CMPS 6100 Lab 05

In this lab, you will implement and use a Linked List and Stack built off of it to solve a practical problem: the matched parenthesis problem. This is a problem relevant to running programs since computer programs are not valid unless all open parentheses are matched by closing ones. In the event a program contains mismatched parentheses, the programmer must be notified. In this lab, you will implement an algorith to detect this case.

In this lab, you will be implementing parts of `linked_list.py`, `stack.py`, and `main.py`.

Refer back to the README.md for instruction on git, how to test your code, and how to submit properly to get all the points you've earned.

Introduction

As you've seen while writing python programs, it is easy to end up with nested open and close parenthesis, for example:

```
for i in range(len(dict.keys())):
```

Python instructions are only valid if every open '(' is matched by a closing ')'. The python interpreter must ensure that this is true for an entire program before trying to execute it.

You will implement a function to ensure that all open parentheses are matched by closing ones in a given string.

However, before we get there, we need to implement our data structures to use.

As Carl Sagan said, "If you wish to make an apple pie from scratch, you must first invent the universe."

We won't go that far, but we will implement a Stack, which, as you've seen, will require us to implement a Linked List for efficiency.

While we only need a handful of methods from a Linked List for a Stack, we will implement a fully functional Doubly Linked List. In the next lab, we will use it to implement a Queue.

Code Organization

Since we have multiple parts, Linked List, Stack, Parentheses Matcher, we are implementing each in its own file and using import statements to pull everything together.

There are three python files in this lab: `linked_list.py`, `stack.py`, and `main.py`, and `unit_tests.py`. The parentheses matcher goes in `main.py`. All pytests are in `unit_tests.py` `linked_list.py` and `stack.pyo` are self-explanatory.

Breaking our program into individual files is a good idea. There are a couple of reasons not to bundle everything in one long source file.

1) Long source files are hard to read
2) We want to minimize import bloat. If you want to use a linked list in a later program, you can cleanly import your linked list without also being forced to import a stack and parentheses matcher too.

All pytests are in `unit_tests.py` for ease of setting up and running the tests on GitHub.

Linked List

We will start this lab by implementing a Linked List. A partially implemented Linked List (the same as presented in the reading), is provided for you in `linked_list.py`.

Your job is to implement all incomplete stubbed out methods: `prepend`, `get`, `remove_last`, and `remove`.

Just like presented in the notes, we highly recommend that you draw out examples as you implement these functions. Stepping through these operations visually is helpful for thinking through them, realizing special cases, debugging mistakes, and implementing them correctly. The drawings in the notebook were not only for your benefit. The author was implementing the Linked List code being presented while writing that notebook.

1. Implement `prepend` and ensure that `test_empty_list_prepend` and `test_prepend` passes. (4 pts)
2. Implement `get` and ensure that `test_get` passes. For efficiency, implement `get` so that if the index requested is the last element in the list, directly retrieve it through the TAIL rather than iterating to it. (3 pts)
3. Implement `remove_last` and ensure that `test_remove_last` passes. (3 pts)
4. Implement `remove` and ensure that `test_remove_special_cases` and `test_remove_general` passes. Note that `remove` takes in the `element` to remove. If that element is not in the list, nothing happens. If that element is in the list, the earliest instance of it is removed. (4 pts)

   For example, after `l.remove(3)` is called on the list:

   ```
   [2]<->[3]<->[5]<->[3]
   ```

   it will have been modified to be:

   ```
   [2]<->[5]<->[3]
   ```

A Note on Unit Tests

Unit Tests are provided for the entire Linked List class, not just the part you are implementing.

Stack

Now having implemented a Linked List, you will find that implementing a Stack is much easier!

Recall that for a stack, you can use a Linked List and implement its `push`, `pop`, and `top` operations by always inserting, removing, and accessing from the same end of the Linked List.

5. Implement `push`, `pop`, and `top` in `stack.py`, and ensure that `test_stack` passes. (4 pts)

Parentheses Checker

With a Stack finally implemented, it's time to implement a parentheses checker.

You will implement a method `are_parentheses_matched(string)` which takes in a string `string` and returns `True` if all open and closed parentheses are matched by their conjugate. An empty string is by default considered matched.

The string `string` will only contain the characters `(` and `)`.

Examples:

- `""` is matched
- `"("` is unmatched
- `")"` is unmatched
- `"()"` is matched
- `"())"` is unmatched
- `"(()"` is unmatched
- `"()()()"` is matched
- `"(())"` is matched

How can we use a Stack to solve this problem?

Consider the string `"(())"`. We can iterate over it. Every time we encounter a `(`, we push it onto the stack. When we encounter a `)`, we pop an element off of the stack. It will be the matching open parenthesis. If we

ever encounter a closing parenthesis, but the stack it empty, that parenthesis is unmatched. If, after iterating over the entire string, the stack still has elements on them, they are unmatched open parentheses. Using these ideas, you can implement a matching parentheses checker.

6. Implement `are_parentheses_matched(string)` which takes in a string and returns `True` if all parentheses are matched and `False` if there are any unmatched parentheses. An empty string is considered to be matched. Ensure that all six parentheses checker tests pass. (7 pts)

Epiloque

Linked Lists are tricky to implement. They are very detail oriented and due to the connected state of the list, subtle mistakes in one operation can result in hard to trace bugs in other operations. That all said, once implemented, they are useful and foundational for other data structures. You have now used your Linked List to implement a Stack which you've used in turn to solve an important problem for all programming languages.