Homework 2

Shane Stephenson - 06/12/2023

All code used in this assignment and the markdown files used to build this assignment can be found in GitHub *here*.

This PDF was generated from a custom extended markdown file. For details of the extension and build instructions, it can be found in GitHub *here*.

All finalized PDFs generated by the build process can be found in GitHub *here*

Problems

Problem Problem 1a

Pseudo-code implementation. In this, `i` is the element to insert at base of stack and `s` is the input stack:

```
FUNCTION INPUTS i: element, s: stack -> OUTPUT stack
  # stack s is the input stack and i is the element to place at bottom
  LET s_0 = INITIALIZE EMPTY STACK

  WHILE s IS NOT EMPTY:
      s_0.push(s.pop) # pop from one stack and immediately move to second stack

  s.push(i)

  WHILE s_0 IS NOT EMPTY:
      s.push(s.pop)

  RETURN s
```

Problem 1b

Pseudo-code implementation. In this, `i` is the element to insert in the third position of the stack and `s` is the input stack:

```
FUNCTION INPUTS i: element, s: stack -> OUTPUT stack
  # stack s is the input stack and i is the element to place in the third position
  LET s_0 = INITIALIZE EMPTY STACK

  WHILE s IS NOT EMPTY:
      s_0.push(s.pop) # pop from one stack and immediately move to second stack

  s.push(s_0.pop)
  s.push(s_0.pop)
  s.push(i)

  WHILE s_0 IS NOT EMPTY:
      s.push(s_0.pop)

  RETURN s
```

Problem 2a

The below table represents each character as it's being iterated over, what the stack looks like, and any stack operation that is being applied:

Character	Stack	Stack Operation
{	{	Push '{'
[	[{	Push '['
A	[{	None
+	[{	None
B	[{	None
]	{	Pop '['
-	{	None
[	[{	Push '['
(	([{	Push '('
C	([{	None
-	([{	None
D	([{	None
)	[{	Pop '('
]	{	Pop '['
END	{	Check empty

After iterating through all characters, stack is non-empty so delimiters are not properly matching.

Problem 2b The below table represents each character as it's being iterated over, what the stack looks like, and any stack operation that is being applied:

Character	Stack	Stack Operation
(	(	Push '('
(	((	Push '('
H	((	None
)	(	Pop '('
*	(	None
{	{(	Push '{'
(	({(	None
[	[({(	Push '['
J	[({(	None
+	[({(	None
K	[({(	None
]	({(	Pop '['
)	{(	Pop '('
}	(	Pop '{'
)		Pop '('
END		Check empty

After iterating through all characters, stack is empty so delimiters are properly matching.

Problem 3 Pseudo-code implementation checking mirrored strings:

```
FUNCTION INPUTS w: string -> OUTPUT bool
  # string w is input string of format xCy
  LET stack = INITIALIZE EMPTY STACK
  LET found_c = FALSE
  LET output = TRUE

  FOR char IN w:
    IF NOT found_c AND char != 'C':
      stack.push(char)
    ELSE IF char == 'C':
      LET found_c = TRUE
    ELSE:
      IF stack.pop != char:
        LET output = FALSE

  RETURN output
```

A Python implementation of the above algorithm with some test cases is as follows. As these problems start to require more modular code, it becomes easier for me to just write the code in Python than writing consistent pseudo-code:

```python
class Stack(list):
    # This is just a wrapper class around Python lists to so that we can interact
    # with the lists in a way that idiomatic of a stack with methods to push and check
    # if the stack is empty.
    def __init__(self):
        super().__init__()

    def push(self, item):
        self.append(item)

    def is_empty(self):
        return len(self) == 0


def check_mirrored(s: str) -> bool:
    stack = Stack()
    found_c = False
    output = True
    for char in s:
        if (not found_c) and (char != 'C'):
            stack.push(char)
        elif char == 'C':
            found_c = True
        else:
            if stack.pop() != char:
                output = False
    return output and stack.is_empty() # using 'and' here to validate stack is empty
```

```
if __name__ == "__main__":
    print("Running test cases...")
    assert check_mirrored('xCx') == True
    assert check_mirrored('xyCyx') == True
    assert check_mirrored('xyCy') == False
    assert check_mirrored('xyCxy') == False
    assert check_mirrored('xyyzaCazyyx') == True
    assert check_mirrored('xyyzaCazyyu') == False
```

Problem 4

The below code is a slight modification and extension of the above code. For this implementation, we use two stacks. One stack collects everything inbetween the 'D's (the 'D-segments') and the second stack is used within the function to test if the extracted D-segment sequence stack represents a valid xCy mirrored string. Again, test strings are asserted at the end.

```
class Stack(list):
    # This is just a wrapper class around Python lists to so that we can interact
    # with the lists in a way that idiomatic of a stack with methods to push and check
    # if the stack is empty.
    def __init__(self):
        super().__init__()

    def push(self, item):
        self.append(item)

    def is_empty(self):
        return len(self) == 0


def check_mirrored(s: Stack) -> bool:
    stack = Stack()
    found_c = False
    for char in s:
        if (not found_c) and (char != 'C'):
            stack.push(char)
        elif char == 'C':
            found_c = True
        else:
            if stack.pop() != char:
                return False
    return stack.is_empty() # Need to ensure stack is empty after popping all matching elements


def check_each_section(s: str) -> bool:
    section_stack = Stack()
    for char in s:
        if char != 'D':
            section_stack.push(char)
        else:
            if not check_mirrored(section_stack):
                return False
            section_stack = Stack()
    return check_mirrored(section_stack) # Need to check last D-segment


if __name__ == "__main__":
    print("Running test cases...")
```

```
assert check_each_section('xCxDxCx') == True
assert check_each_section('DxCxDxCx') == True
assert check_each_section('xCxDxCxD') == True
assert check_each_section('xyCyxDxyCyx') == True
assert check_each_section('xyCxyDxyCyx') == False
assert check_each_section('xyyzaCazyyxDxyCyxDuwqCqwu') == True
assert check_each_section('xyyzaCazyyxDxyyzaCazyyu') == False
```

Problem 5

For this, I'll be using the same Stack class defined above and simply extending it with a method for inserting an element in the stack and reading an element within the stack. For this example, we'll implement it so that the elements are indexed from oldest (bottom of stack) as 0 to newest (top of stack) as stack length.

```
class Stack(list):
    def __init__(self):
        super().__init__()

    def push(self, item):
        self.append(item)

    def is_empty(self):
        return len(self) == 0

    def peek(self):
        return self[-1]

    def insert(self, item, index: int) -> 'Stack':
        # This implementation of insert works by popping every item
        # from the stack and onto a secondary stack until the number
        # of items popped is equal to the index. Then the item is
        # popped onto the stack and then all additional items from
        # the secondary stack are popped back onto the original stack.
        secondary_stack = Stack()
        for _ in range(index):
            secondary_stack.push(self.pop())
        self.push(item)
        for _ in range(index):
            self.push(secondary_stack.pop())
        return self

    def read(self, index: int):
        # This implementation works the same way that the `insert` method
        # above works, except when the index is reached the stack head is peeked.
        # Once all items are moved back onto the original stack from the
        # secondary stack, the peeked item is returned.
        secondary_stack = Stack()
        for _ in range(index):
            secondary_stack.push(self.pop())
        output = self.peek()
        for _ in range(index):
            self.push(secondary_stack.pop())
        return output


if __name__ == "__main__":
```

```
print("Running test cases...")
# create the test stack Stack(a, b, c, d)
stack = Stack()
stack.push('a')
stack.push('b')
stack.push('c')
stack.push('d')

# check if each item in the stack returns proper index
assert stack.read(0) == 'd'
assert stack.read(1) == 'c'
assert stack.read(2) == 'b'
assert stack.read(3) == 'a'

# insert 'e' into index 2 of the stack
stack.insert('e', 2)
assert stack.read(0) == 'd'
assert stack.read(1) == 'c'
assert stack.read(2) == 'e'
assert stack.read(3) == 'b'
assert stack.read(4) == 'a'

# insert 'z' into index 5 of the stack
stack.insert('z', 5)
assert stack.read(0) == 'd'
assert stack.read(1) == 'c'
assert stack.read(2) == 'e'
assert stack.read(3) == 'b'
assert stack.read(4) == 'a'
assert stack.read(5) == 'z'
```