Homework 2

Shane Stephenson - 06/12/2023

All code used in this assignment, the markdown files used to build this PDF, and the PDF building script can be found in GitHub here.

Problems

Problem 1a

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

```
FUNCTION [i: ELEMENT, s: STACK] -> STACK
START
  # 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
END
```

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 [i: ELEMENT, s: STACK] -> STACK
START
  # 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
END
```

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. In this function, `w` is the input string in form xCy that will be tested for being mirrored:

```
FUNCTION [w: STRING] -> BOOLEAN
START
  # 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
END
```

A Python implementation of the above algorithm with some test cases is as follows. As these problems become more complex, I feel it more necessary to convert the pseudo-code into real code that I can use to test various test cases.

```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:
```

```python
                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

Within the below pseudo-code, the function named `check_each_section` takes an input string and returns `true` if it matches the pattern and `false` if there is a non-symmetric xCy substring.

```
FUNCTION check_mirrored [input_stack: STACK] -> BOOLEAN
START check_mirrored:
  # stack inpt_stack is input stack representation with format xCy
  LET stack = INITIALIZE EMPTY STACK
  LET found_c = FALSE
  WHILE NOT s.is_empty:
    LET char = s.pop
    IF NOT found_c AND char != 'C':
      stack.push(char)
    ELSE IF char == 'C':
      LET found_c = TRUE
    ELSE:
      IF stack.pop != char:
        RETURN FALSE
  RETURN stack.is_empty
END check_mirrored

FUNCTION check_each_section [w: STRING] -> BOOLEAN
START check_each_section:
  LET section_stack = INITIALIZE EMPTY STACK
  FOR char IN w:
    IF char != 'D':
      section_stack.push(char)
    ELSE:
      IF NOT check_mirrored(section_stack):
        RETURN FALSE
      LET section_stack = INITIALIZE EMPTY STACK
  RETURN check_mirrored(section_stack)
END check_each_section
```

The below code is a slight modification and extension of the above Python 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.

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

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

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

```python
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

Within the pseudo-code implementation below, the function `insert` shows how a second stack can be used to insert a new element into the primary stack and the function `read` will read the element from the stack.

For both these functions, the input parameters `i` are the index to insert/read at, `s` is the stack to do the reading/insertion in, and `e` is the element to insert (for the insertion function).

In this implementation, the bottom of the stack will be indexed at 0 and will increase by increments of +1 for each additional element added to the stack. Once the stack has been popped a number of times equivalent to the index, the element to insert is pushed onto the stack before moving everything from the secondary stack back onto the primary stack. The `read` function works similarly to the `insert` function, but it simply peeks once it gets to the appropriate index instead of pushing a new element onto the stack.

```
FUNCTION insert [e: ELEMENT, i: INDEX, s: STACK] -> STACK
# function which uses a second stack to insert into another stack
START insert
  LET secondary_stack = INITIALIZE EMPTY STACK
  FOR _ FROM 0 to i:
    secondary_stack.push(s.pop)
  s.push(e)
  FOR _ FROM 0 to i:
    s.push(secondary_stack.pop)
  RETURN s
END insert

FUNCTION read [i: INDEX, s: STACK] -> STACK
# This function works almost the same as 'insert', just
# without doing the actual insertion
START read
  LET secondary_stack = INITIALIZE EMPTY STACK
  FOR _ FROM 0 to i:
    secondary_stack.push(s.pop)
  LET output = s.read
  FOR _ FROM 0 to i:
    s.push(secondary_stack.pop)
  RETURN output
END read
```

Python implementation with test cases are shown below:

```
FUNCTION insert [e: ELEMENT, i: INDEX, s: STACK] -> STACK
# function which uses a second stack to insert into another stack
START insert
  LET secondary_stack = INITIALIZE EMPTY STACK
  FOR _ FROM 0 to i:
    secondary_stack.push(s.pop)
  s.push(e)
```

```
  FOR _ FROM 0 to i:
    s.push(secondary_stack.pop)
  RETURN s
END insert

FUNCTION read [i: INDEX, s: STACK] -> STACK
# This function works almost the same as 'insert', just
# without doing the actual insertion
START read
  LET secondary_stack = INITIALIZE EMPTY STACK
  FOR _ FROM 0 to i:
    secondary_stack.push(s.pop)
  LET output = s.read
  FOR _ FROM 0 to i:
    s.push(secondary_stack.pop)
  RETURN output
END read
```

Problem 6

Need to come back to complete.

```
# Design a method for keeping two stacks within a single linear array
# s[SPACESIZE] so that neither stack overflows until all of memory is used and an
# entire stack is never shifted to a different location within the array. Write
# methods push1, push2, pop1, and pop2 to manipulate the two stacks. (Hint:
# the two stacks grow toward each other.)
```

```
TODO!
```

Problem 7a

We can parse the expression `(A+B)*(C$(D-E)+F)-G` for postfix in the following steps. The stack starts with '(' inside.

Character	Stack	Current expression
(	((	NONE
A	((	A
+	((+	A
B	((+	AB
)	(	AB+
*	(*	AB+
(	(*(	AB+
C	(*(	AB+C
\$	(*(	AB+C$
(	(*((	AB+C$
D	(*((	AB+C$D
-	(*((-	AB+C$D
E	(*((-	AB+C$DE
)	(*(	AB+C$DE-
+	(*(+	AB+C$DE-
F	(*(+	AB+C$DE-F
)	(*	AB+C$DE-F+
-	(-	AB+C$DE-F+*
G	(-	AB+C$DE-F+*G
)	NONE	AB+C$DE-F+*G-

So postfix notation for the expression is `AB+C$DE-F+*G-`

For prefix notation, we can parse as follows:

Character	Stack	Current expression
G	(	G
-	(-	G
(	(-(	G
F	(-(	GF
+	(-(+	GF
(	(-(+(	GF
E	(-(+(	GFE
-	(-(+(-	GFE
D	(-(+(-	GFED
)	(-(+	GFED-
$.g	(-(+	GFED-$
C	(-(+	GFED-$.gC
)	(-	GFED-$.gC+
*	(-*	GFED-$.gC+
(	(-*(	GFED-$.gC+
B	(-*(	GFED-$.gC+B
+	(-*(+	GFED-$.gC+B
A	(-*(+	GFED-$.gC+BA
)	(-*	GFED-$.gC+BA+
)	NONE	GFED-$.gC+BA+*-

So the prefix notation for that expression is `GFED-$C+BA+*-`.

Problem 7b

We can parse the expression `A+(((B-C)*(D-E)+F)/G)$(H-J)` in postfix notation using the following steps:

Character	Stack	Current expression
A	(	A
+	(+	A
(	(+(	A
(	(+((	A
(	(+(((	A
B	(+(((	AB
-	(+(((-	AB
C	(+(((-	ABC
)	(+((	ABC-
*	(+((*	ABC-
(	(+((*(	ABC-
D	(+((*(	ABC-D
-	(+((*(-	ABC-D
E	(+((*(-	ABC-DE
)	(+((*	ABC-DE-
+	(+((+	ABC-DE-*
F	(+((+	ABC-DE-*F
)	(+(	ABC-DE-*F+
/	(+(/	ABC-DE-*F+
G	(+(/	ABC-DE-*F+G
)	(+	ABC-DE-*F+G/
$	(+	ABC-DE-*F+G/$
(	(+(	ABC-DE-*F+G/$
H	(+(	ABC-DE-*F+G/$H
-	(+(-	ABC-DE-*F+G/$H
J	(+(-	ABC-DE-*F+G/$HJ
)	(+	ABC-DE-*F+G/$HJ-
)	NONE	ABC-DE-*F+G/$HJ-+

So the postfix notation for the expression is `ABC-DE-*F+G/$HJ-+`.

To parse the expression into prefix notation, we complete the following steps:

Character	Stack	Current expression
(	((	NONE
J	((	J
-	((-	J
H	((-	JH
)	(	JH-
$	(	JH-$
(	((	JH-$
G	((	JH-$G
/	((/	JH-$G
(	((/(	JH-$G
F	((/(	JH-$GF
+	((/(+	JH-$GF
(	((/(+(	JH-$GF
E	((/(+(	JH-$GFE
-	((/(+(-	JH-$GFE
D	((/(+(-	JH-$GFED
)	((/(+	JH-$GFED-
*	((/(+*	JH-$GFED-
(	((/(+*(	JH-$GFED-
C	((/(+*(	JH-$GFED-C
-	((/(+*(-	JH-$GFED-C
B	((/(+*(-	JH-$GFED-CB
)	((/(+*	JH-$GFED-CB-
)	((/	JH-$GFED-CB-*+
)	(	JH-$GFED-CB-*+/
+	(+	JH-$GFED-CB-*+/
A	(+	JH-$GFED-CB-*+/A
)	NONE	JH-$GFED-CB-*+/A+

So the prefix notation for the expression is `JH-$GFED-CB-*+/A+`

Problem 8a

Out of time.