Algorithms and Data Structures: Week 4

Johnson rev. Bell (Michaelmas 2024)

Question 1

Would you use a stack or a queue for each of the following applications?

(a) Managing a waiting list.

(b) Recording pages visited by a web browser for use with back button.

(c) Scheduling jobs for a supercomputer.

(d) Maintaining an undo sequence in a text editor.

Provide a justification for at least two of your answers. Can you think of another application of a stack, not listed here? Can you think of another application of a queue, not listed here?

Question 2

Given an initially empty queue complete the table below with the output of each operation and the updated contents of the queue.

operation	output	queue
enqueue(4)		
enqueue(6)		
dequeue		
enqueue(2)		
dequeue		
front		
dequeue		
dequeue		
isEmpty		
enqueue(1)		
enqueue(2)		
size		
enqueue(6)		
enqueue(4)		
dequeue		

Question 3

Given an initially empty stack complete the table below with the output of each operation and the updated contents of the stack.

operation	output	stack
push(2)		
push(9)		
push(8)		
pop		
pop		
top		
pop		
pop		
isEmpty		
push(1)		
push(3)		
push(5)		
size		
push(2)		
pop		

Question 4

An arithmetic expression written in *postfix* notation is evaluated by reading from left to right and when an operator (plus, minus, multiply, divide) is met it is evaluated on the last two operands (numbers) seen, and the value obtained is substituted in to the expression. For example: to evaluate

$$4\ 5\ 1\ 2\ +\ -\ /$$

we read across until we reach the $+$ which we then evaluate on the last two numbers: 1 and 2. The result obtained is 3 which replaces them in the string to give

$$4\ 5\ 3\ -\ /$$

We now read the minus sign and evaluate it on the 5 and 3 (keeping that ordering) to get

$$4\ 2\ /$$

and so finally we evaluate $4/2$ to obtain the result 2. Note that in the usual *infix* notation we would have written this as

$$4/(5-(1+2))$$

Write pseudocode that evaluates a postfix expression. Use a stack. You may test for the type of a value in the condition clause of an **if** statement.

Implementation Notes

Recall the presentation of a solution to the bracket-matching problem from lecture:

Input: A string X consisting of a sequence of parentheses
Output: Returns **true** if the parentheses in X are balanced, **false** otherwise

```
Initialize an empty stack S
for each character x in X do
    if x is an open parenthesis then
        S.push(x)
    else
        if x is a close parenthesis then
            if S is empty then
                return false
            end if
            if S.pop() does not match x then
                return false
            end if
        end if
    end if
end for
if S is empty then
    return true
else
    return false
end if
```

We looked in lecture about how to use arrays and linked lists to implement Stacks. You may know that Python does not (straightforwardly) allow for the use of arrays. There are, of course, Python `lists` but these are not arrays in the sense we defined in lecture. Why not? This might not be immediately obvious, but it is likely that you know that Python `lists` will happily contain values of multiple types:

```
some_list = [3, 'orange', [1, 2, 3]]
```

This would appear to be at odds with the property that arrays (at least in ADS) are homogeneous. If you want to test your understanding of an array-backed Stack implementation, you can treat Python `lists` a bit like they are arrays, by "declaring" an empty "array" of size N, and a variable t with the usual meaning as follows:

```
fake_A = [None] * N
t = -1
```

and writing some functions that implement the Stack's behaviour described in lecture.

Another option, which we can look at here, is to use the Python `list` "out-of-the-box" to simulate some of the behaviours of a Stack (though we can't guarantee that we get the performance benefits of doing so—this is a problem for the developers of Python). We can think about this as a "third" way to implement a Stack, which is not assessable. (You do, however, need to know about the array-backed and the linked-list–backed approaches.) To do this, we note that Python `lists` support an `.append(e)` method, which behaves similarly to Stack.push(e) and a `.pop()` method. Knowing this, can you rearrange the lines in Listing 1 so that it creates a valid Python implementation that matches the pseudocode description of the solution to the bracket-matching problem from lecture?

```
1     S = []
2         if x == '(':
3                 if S == []:
4     else:
5             if x == ')':
6
7             S.append(x)
8                     return False
9         else:
10        return False
11        return True
12                if S.pop() != '(':
13                    return False
14    if S == []:
15
16    for x in X:
17def is_balanced_parens(X):
```

Listing 1: Reorder the lines: Balanced Parentheses

Extension

If you finish the above questions, consider the following *optional* tasks.

- Implement your solution to Question 4 above using Python, with the help of the stack implementation in the Jupyter Notebook on Ultra.
- Try Extra Question 1 below.
- Try Extra Question 2 below.
- Try Project Euler Problem 164 (`https://projecteuler.net/problem=164`).

Extra Question 1

Consider the following table of data. The first row denotes the day, the second is the price of a share (so the array shows how the price changes over time).

0	1	2	3	4	5	6	7	8	9
221	198	203	214	213	219	220	216	218	222

We would like to add a third row telling us how many days it is since the price was higher than it is currently. So in the above example we would get (assume we write $*$ if the price has never been higher):

0	1	2	3	4	5	6	7	8	9
221	198	203	214	213	219	220	216	218	222
$*$	1	2	3	1	5	6	1	2	$*$

Use pseudocode to describe an algorithm that given an array containing the data on the second row, returns a new array with the data for the third row. Your algorithm should read each piece of data in the input array as seldom as possible. (Unsurprising hint: use a stack.)

Extra Question 2

Write some pseudocode that converts an infix expression to an equivalent postfix expression – for example, $2 * 3 - 8/4$ should become $2\ 3\ *\ 8\ 4\ /\ -$. Note that this expression evaluates to 4 since multiplication and division take precedence over addition and subtraction. For operators of equal precedence, the one farthest to the left takes precedence.

You can assume the infix expression contains only single-digit numbers and operators (no brackets). You can also assume there exists a simple syntax to test for operator precedence in pseudocode (something like "*X* has precedence over Y" that you may use in a conditional statement). (After you have completed the exercise, think about how you might implement this.)

One possible approach: to begin, initialise an empty stack and an empty string. Your algorithm will repeatedly add to the string so that finally it is the desired expression. The infix expression should be read just once, left to right.

The order of the numbers need not change so when a number is read in the infix expression it can be added immediately to the postfix string.

Operators are either added to the string or the stack. How do you decide? What should be done if the stack is empty? If it is not empty (so you can look and see what is at the top), when should you pop, when should you push? Think about what possible state the stack may have once all the numbers have been processed. How to do deal with this?