20200121

Material covered at the beginning of today's class was included in the posted notes for July 17.

Picking up right where those notes left off …

Now, what can we say about this postfix evaluation problem in terms of its complexity? It should be clear that this problem is in $\Omega(n)$ - where n is the length of the postfix expression - since we must at least look at every token in the expression.

Furthermore, you can see that the algorithm given is in $O(n)$ since the amount of work done for each token is bounded by a constant (remember: all arithmetic operations take constant time – this is part of our model of computation). Thus we have an algorithm with O classification equal to the Ω classification of the problem. Thus this problem is in $\Theta(n)$ and we know that no algorithm for this problem can have a lower O classification.

Wait a minute ... there's a big *unstated* assumption in that last paragraph ...

The claim that the algorithm is in $O(n)$ is **only** true if each of the stack operations is in $O(1)$ (i.e. takes constant time) ... and that may not be true!

So now we need to look at the actual implementation of a stack.

There are two simple solutions: **store the stack in a one-dimensional array** or **store the stack in a linked list**

Array: we can use an array with indices in the range [0..k] for some k. We store the stack in locations 0, 1, 2 etc, with location 0 holding the first item pushed onto the stack, etc. We can use a variable called **top** to keep track of the top of the stack. Our Stack class might look something like this:

```
class Stack():

   # constructor
   def init():
      this.array1 = new array[0..k]
      this.top = -1                    # the stack is empty

   def push(x):
      if this.top == k:
          ERROR("Stack overflow")
      else:
          this.top += 1
          this.array1[top] = x

   def pop():
      if this.top == -1:
          ERROR("Can't pop from empty stack")
      else:
          x = this.array1[top]
          this.top -= 1
          return x

   def isEmpty():
      return this.top == -1
```

This is very fast and simple - but of course the maximum size of the stack is limited. This can be handled by allocating a new, larger array when needed – but that's a time-costly action.

Linked List: For the linked list Stack implementation, we need to create a **Node** object, containing two fields:

value - the value being stored

next - a pointer to another Node object

```
class Node():

        #instance variables
        value: integer
        next: Node

        # constructor
        def init(x):
                this.value = x
                this.next = nil
```

Now our Stack class might look like:

```
class Stack():

        #instance variables
        top : Node


    # constructor
    def init():
        this.top = nil


    def push(x):
        newNode = new Node(x)
        newNode.next = this.top
        this.top = newNode
```

```
def pop():
    if this.top == nil:
        ERROR("Can't pop from empty stack")
    else:
        x = this.top.value
        this.top = this.top.next
        return x


def isEmpty():
    return this.top == nil
```

This involves more operations per push and pop than the array version, and so will be a bit slower in practice. However it has the benefit that there is no upper limit on the size of the stack.

Now we can verify that with either of these implementations, all stack operations take $O(1)$ time ... so the $\Theta(n)$ classification of the problem is correct.

******* BONUS MATERIAL*******

In class I mentioned that there is a well-known algorithm to translate expressions from infix notation to postfix notation. I'm including this algorithm in the notes here – it's worth looking at. I'm explaining it in detail in these notes but you can treat this as "enrichment" material. I won't test you on this algorithm.

If you examine the postfix expression we first looked at, you may notice that the operands (the numbers) are in exactly the same order as they were in the original infix expression. This gives a clue to how we might design an algorithm to do the translation:

- leave the operands in the same order
- working in decreasing order of precedence, push each operator to the right until it is just to the right of the operands that it applies to

The problem with this is that it requires multiple passes over the infix expression. To avoid this we take a different approach: we step through the infix expression from left to right, passing the operands straight through, but keeping track of the operators as we go and "holding them in reserve". When we encounter an operator with higher precedence than the previous one, we add it to the ones we are holding. When we encounter an operator with equal or lower precedence than the previous one, we "bring back" (ie "output") the high precedence operator(s) we are holding on to, then we "hold on to" the new operator.

Here are two tiny examples. Suppose our infix expression is 3 * 4 + 5 * 6

We output 3, then hold onto the *, then output 4. The next token is + . This has lower precedence than the * we are holding, so it effectively separates the * operation from whatever comes after the + . The low precedence of + means that we want to evaluate what's to its left and what's to its right before we apply the + . So at this point we bring back the * we are holding, and output it. The output so far is 3 4 *. Now we hold onto the + . We output the 5 (output so far is 3 4 * 5). The next token is * . This has higher precedence than the + we are holding so we hold onto it too (we are now holding + *). Next we see and output the 6. We are left with the two operators we are holding so we output them, last to first. Our completed output is 3 4 * 5 6 * +

Second example: 3 + 4 * 5 + 6

We output 3, hold onto +, output 4, hold onto * (it has higher precedence than the + we are holding), then output 5. We see the + , which has lower precedence than the * we are holding, so we output the * (total output so far is 3 4 5 *) We can also output the + we are holding because we have evaluated the values before and after it. We hold onto the + that we just obtained because we have not yet seen its second operand. Now we see and output 6. Finally we are left with the held + operator so we output it. The completed output is 3 4 5 * + 6 +

The key concept is that we defer each operator until we are sure that its operands have been completely added to the output sequence.

The last key concept of the algorithm is that parentheses effectively embed complete expressions within the overall expression. When we encounter parentheses we make sure we completely evaluate what's between them before carrying on.

As a larger example, consider the infix expression **6 + 8 * 4 / 9 - 5**

6 is an operand, so we simply output it **6**
+ is an operator, so we hold onto it
8 is an operand so we output it **8**
* is an operator, its precedence is higher than the previous one (+) so we hold onto it
4 is an operand so we output it **4**
/ is an operator with equal precedence to the previous one, so we output the previous one, and hold onto the /
*
9 is an operand so we output it **9**
- is an operator with lower precedence than the previous one (/) so we bring back the /
/
Now we compare the **-** to the other operator we are holding onto (+). The + also has precedence $\geq$ the new operator so we output it too
+
5 is an operand so we output it **5**
We are left holding onto the **-** and there are no more numbers so we output the -
-

Thus we get **6 8 4 * 9 / + 5 -** which is a valid postfix form of the original infix expression

The question is, how are we going to hold onto those operators, and get them back in reverse order when we need them? The answer is ... you guessed it … a **stack**.

Holding onto things and giving them back in reverse order what stacks do, and it is exactly what we need for our infix-to-postfix algorithm. Here's the algorithm. I've highlighted the actual pseudo-code because there are a lot of explanatory comments.

```
InfixToPostfix(e):   # e is an expression in infix notation, in which we can
                     # identify the individual tokens (operands, operators and
                     # parentheses)
    # We will assume e is well-formed
    S = new Stack()          # S will be used to hold operators until we need them
    postFix = empty list
    for t in e:
        if t is an operand:
            append t to postFix     # we don't add operands to the stack, we just
                                    # pass them through
        else if t is a left parenthesis "(":
            S.push(t)
        else if t is a right parenthesis ")":
            # pop off all stored operators, back to the
            # matching left parenthesis
            x = S.pop()
            while x != "(" :
                append x to postFix
                x = S.pop()
        else:
            # The only remaining possibility is that t is an operator.  We will add it to the
            # stack, but first we must pop off any higher precedence operators that need to be
            # added to the Postfix expression now.

            if not S.isEmpty():
                # pop stored operators until we find one with lower precedence
                # than t
                x = S.pop()
                while precedence(x) >= precedence(t):
                    # for the purpose of this algorithm,  the precedence of a
                    # left parenthesis (  must be 0
                    append x to postFix
                    # it's time for this operator to join the postFix expression
                    if not S.isEmpty():
                        x = S.pop()
                        # get the next one from the stack
                    else:
```

```
                    break                    # exit the while loop because the stack is empty
                if precedence(x) < precedence(t):
                        # if the last thing we removed from the stack has lower
                        # precedence than t, it needs to go back on the stack
                        S.push(x)
            # now the stack is ready for the new operator to be added
            S.push(t)
                # the new operator always goes on the stack, waiting until
                # its operands are ready
        # now we have reached the end of the infix expression – we need to clear out any
        # operators still in the stack
        while not S.isEmpty():
            # add any operators still in S to the postfix expression
            x = S.pop()
            append x to postFix

    return postFix
```

That's a long-winded, complex-looking algorithm, and you should work through it by hand for a couple of examples to see how it works. But you don't need to memorize it.

Here it is again without the annotation:

```
InfixToPostfix(e):

    S = new Stack()
    postFix = empty list
    for t in e:
        if t is an operand:
            append t to postFix
         else if t is a left parenthesis "(":
              S.push(t)
        else if t is a right parenthesis ")":
              x = S.pop()
              while x != "(" :
                  append x to postFix
                  x = S.pop()
        else:
              if not S.isEmpty():
                 x = S.pop()
                 while precedence(x) >= precedence(t):
                      append x to postFix
                      if not S.isEmpty():
                         x = S.pop()
                      else:
                           break
                      if precedence(x) < precedence(t):
                           S.push(x)
              S.push(t)

       while not S.isEmpty():
              x = S.pop()
              append x to postFix

    return postFix
```

If you're still reading, congratulations! This is the end of our brief look at stacks – hopefully I have convinced you that even though they are simple, they are extremely powerful.