At this point we finally turned our attention to a data structure: the stack.

A stack is our first example of an Abstract Data Type: we specify the operations we need to be able to perform on the data we will store, but we do not specify the details of the implementation. Of course when we actually write code we do need to choose a specific implementation, and the choice we make will often have significant impact on the efficiency of our program.

A stack must provide (at least) three operations:

- `push(x)` - add the value x to the stack
- `pop()` - remove the most recently added value, and return it
- `isEmpty()` - return True if there are no values in the stack, and False otherwise

In keeping with popular practice, we will imagine that we have implemented a Stack class and that we can create a stack with a statement such as

```python
S = new Stack()
```

Then the operations listed above become methods attached to the stack we create.

Stacks are often described as a LIFO (Last In First Out) data structure: the most recently added (pushed) value is the first one removed (popped).

The first thing to notice about a stack is that it automatically reverses the order of a sequence:

```python
S = new Stack()
S.push(1)
S.push(3)
S.push(7)
S.push(9)
```

```python
print S.pop(), S.pop(), S.pop(), S.pop()
```

will print 9 7 3 1
Stacks are so limited in structure and operation that it may seem that they can’t be very useful. However their simplicity is an advantage. Stack-based algorithms often run very fast and can be proved to be correct without too much trouble. There is a language called Forth that is basically designed around stack operations – this makes the compiler/interpreter very small and simple, which means it can run on platforms with very limited processing resources. This becomes more and more important as the Internet of Things becomes reality and every object has a small onboard processor.

Actually, we deal with a stack every time we run a program that has method calls. Whenever a program calls a method, a “snapshot” of the current status (values of variables, point of execution, etc.) is pushed onto a stack that is often called “the execution stack”. If that method calls another method (or calls itself recursively), another “snapshot” is pushed onto the stack. When a method returns, the top snapshot on the stack is popped off and used to continue execution from the point at which the method was called. The only time we are likely to be reminded of this stack is when we accidentally fall into infinite recursion – the error message is likely to contain the phrase “stack overflow”.

We will talk later about important reasons for using stacks, but first I want to introduce a very practical application of the stack data structure.

Consider the problem of evaluating an arithmetic expression such as $3 + 4 \times 7 + 8 \times (3 - 1)$

Most people in North America have been taught that parentheses have highest precedence, followed by exponentiation, then multiplication and division, then addition and subtraction, so the expression above evaluates to $3 + 28 + 16 = 47$

But these precedence rules are completely arbitrary. For example, we could keep the rule about parentheses but do everything else in simple left-to-right order ... which would give 114 ... or right-to-left order ... which would give 103 ... or give addition higher precedence than multiplication ... which would give 210 (assuming I have done the calculations properly)

The notation we have used here to write down the expression is called infix notation because the operators (*, +, etc) are placed between the operands (3, 4, 7, etc)

In order to evaluate an infix expression correctly we need to know exactly what rules of precedence were intended by the person who created the expression. Wouldn’t it be wonderful if there were a universal way to represent an expression so that no matter what rules of precedence are in use, the method of evaluating the expression is always the same?

There is! It is called postfix notation, and it was invented in 1924 by Jan Łukaseiwicz ...
(which I have trouble pronouncing, but it is something like Wuhkashavicks). Because he was Polish, this is sometimes called Polish Postfix notation. In postfix notation, operators come after their operands, so "3 * 4" (infix) becomes "3 4 *" (postfix)

The expression we started with: \(3 + 4 * 7 + 8 * (3 - 1)\) can be written as \(3 4 7 * 8 3 1 - * + +\)

We can evaluate this in simple left-to-right order ... we keep going until we hit an operator (the first one is *) and then we apply it to the two numbers just before it: \(4 * 7 = 28\), and we put the result in place where the \(4 7 *\) was, so now the expression is \(3 28 8 3 1 - * + +\)

The next thing we see is the -, which we apply to the numbers just in front of it (3 and 1) and put the result back in the expression, giving

\(3 28 8 2 * + +\)

The next operator we find is *, which we apply to 8 2. The expression is now

\(3 28 16 + +\)

The next operator is +, applied to the 28 16. This gives

\(3 44 +\)

We apply the + to the numbers before it, giving a final result of 47 ... which is exactly the result we expect from the original expression using standard rules of precedence ... but note that we did not need to know those precedence rules. Once we have the expression in postfix form we just evaluate from left to right.

This is really useful because if we are writing the calculation in a program, if we want to write it in infix notation then we have to be sure the compiler is going to use exactly the same precedence rules as we do. But if we write the expression in postfix notation the compiler does not have to know anything about precedence rules or parentheses – all it has to know is that the expression is to be evaluated from left to right using the simple process outlined above.

But something magic happened there - I just pulled the postfix version of the expression out of thin air. Can we find a way to convert any infix expression to an equivalent postfix expression?

Of course we can – as long as we know the precedence rules, we can construct an algorithm that scans the infix expression and converts it to a postfix expression.

One approach is to build it up one piece at a time ... for example, we can look at

\(3 + 4 * 7 + 8 * (3 - 1)\)

and see the parenthesized part "(3 - 1)" which we can immediately write as "3 1 -" and now that this is taken care of, we can work on the next level of precedence: multiplication and division. We can see that "4 * 7" will become "4 7 *", and that the "8 *" combines with the "3 1 -" to give "8 3 1 - *". Now all that we have left to deal with are the additions. We can resolve these in different ways, but perhaps the simplest is to put them at
the end. Following this process we see that the original expression could be translated into postfix notation as $3 4 7 * 8 3 1 - * + +$. An alternative for placing the “+” signs is to insert each one right after the things it applies to, which gives $3 4 7 * + 8 3 1 - * +$. This demonstrates that there can be more than one correct postfix version of an expression.

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. If you wish to skip over it, look for the line that starts ********

If you examine the two postfix expressions just given above, 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.

The key concept is that we defer each operator until deferring it any longer would create an error.
As an example, consider the infix expression $6 + 8 * 4 / 9 - 5$ ... we process it from left to right

6 is an operand, so we simply output it
+ is an operator, so we hold onto it
8 is an operand so we output it
* is an operator, its precedence is higher than the previous one (+) so we hold onto it
4 is an operand so we output it
/ 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
- 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 (+). This also has precedence $\geq$ the new operator so we output it too

+  
5 is an operand so we output it
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 ... a stack.
Holding onto things and giving them back in reverse order is exactly what we need for our infix-to-postfix algorithm. The algorithm looks like this:

```plaintext
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()
    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:
            # The only remaining possibility is that t is an operator. We will add it to the
            # stack, but first we must pop off any 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):
                    # assume precedence(x) returns the precedence level of x
                    # for example, precedence("*") would be > precedence("+")
                    x = S.pop()
                # for the purpose of this algorithm, precedence("(") - ie. 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()
                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)

S.push(t)
    # the new operator always goes on the stack, waiting until
    # its operands are ready

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.

*******

We’ll pick up the thread by showing that an expression in postfix form can be evaluated with
a simple algorithm ... also using a stack. Don’t worry – this algorithm is a lot simpler than the
one that creates the postfix form.
Let E be a string that represents an expression in postfix form
Assume E has a “next” method that returns the next token in E
Let S be a stack

while we haven’t processed all of E:
    x = E.next()
    if x is a value:
        S.push(x)
    else:
        # x is an operator
        let n be the number of values required for x (usually 2)
        pop the top n values from S
        y = the result of applying operator x to the values just popped off S
        S.push(y)
return S.pop()  #return the final computed value

This algorithm will fail if E is not well-formed. As an exercise, improve the algorithm by using the isEmpty() method to avoid problems.

It is worth noting that postfix notation is very important in computer science because it gives a good model how arithmetic is actually carried out in a computer. When we write a high-level statement like
C = A + B
it gets translated into assembly language sort of like this:
load the contents of address A into a CPU register
load the contents of address B into another CPU register
add the contents of those two registers and store the result in another CPU register
copy that register to address C

In other words the addition is really carried out in a postfix way: we identify the operands, then execute the operation on them

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 $\mathcal{O}$ classification equal to the $\Omega$ classification of the problem. Thus this problem is in $\Theta(n)$ and we know that no algorithm for this problem can have a lower $\mathcal{O}$ 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.

... and that’s where we ended for the day.