The Facts About Stacks

Robin Dawes

February 17, 2021

Discusses the practical implementation of the Stack data type, and details the application of a stack to the problem of converting an expression from infix to postfix.

.

Where Were We?

Picking up right where the previous note left off ...

What can we say about the computational complexity of the postfix evaluation problem? 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. Thus we have an algorithm with its Ω classification equal to its O classification. Thus this problem is in $\Theta(n)$ and we know that no algorithm for this problem can have a lower O classification.

Remember: all arithmetic operations take constant time - this is part of our model of computation.

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 $O(1)$ (ie. takes constant time) ... and at this point I've given you no reason to believe that!

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

Implementing a Stack

THERE ARE TWO simple approaches to implementing a *Stack* class:

- store the stack in a one-dimensional array
- store the stack in a linked list

Each has advantages and drawbacks.

ARRAY IMPLEMENTATION: we can use an array with indices in the range $[0..k]$ for some k. We store the stack in locations 0, 1, 2 ... k, 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 potentially time-costly action.

In a later note we will analyze this cost more carefully - in some circumstances it can be quite practical.

LINKED LIST IMPLEMENTATION: 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.

Infix to Postfix

Earlier I mentioned that there is an $O(n)$ algorithm to translate expressions from infix notation to postfix notation.

I'm explaining it in detail in these notes but you can treat this as "enrichment" material.

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 method is that it requires multiple passes over the infix expression. To avoid this we take a different approach: we step through the infix expression just once from left to right, passing the operands straight through, but keeping track of the operators as we go and "holding them in reserve" until they are needed. When we encounter an operator with higher precedence than the previous one, we add it to the ones we are holding. Because the stack is a LIFO structure, the higher precedence operator will be popped off and added to the expression *before* the other, lower precedence operators.

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 because we must have already seen their operands, then we "hold on to" the new operator.

EXAMPLE 1: $3 * 4 + 5 * 6$

As we scan across this from left to right, we hold onto the "*" . When we see the "+" we know that the "4" just before it belongs to the "*" we are holding, so we add the "*" to the postfix expression we

are building right after the 4. Then we hold onto the "+". When we see the next "*", we see it has higher precedence than the "+" we are holding so we don't bring back the "+" to own the "5" - the "5" belongs to the "*" we just found. We keep going until we see there is nothing after the "6" ... at which point we bring back the "*" to own the "5" and "6", then the "+" to do the final addition.

Taking that one step at a time:

1. We see and output 3
2. We see and hold onto *
3. We see and output 4
4. We see + . 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 whatever is to its left and whatever is to its right before we apply the + . So at this point we bring back the * we are holding, and we output it. The output so far is 3 4 *
5. We hold onto the +
6. We see and output the 5 (output so far is 3 4 * 5).
7. We see * . This has higher precedence than the + we are holding so we hold onto it too (we are now holding + *)
8. We see and output 6
9. 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

1. We see and output 3
2. We see and hold onto +
3. We see and output 4
4. We see and hold onto * (it has higher precedence than the + we are holding)
5. We see and output 5
6. We see the + . It has lower precedence than the * we are holding, so we output the * (total output so far is 3 4 5 *).
7. 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.
8. We see and output 6
9. 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 other key concept of the algorithm (not shown in either example) 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.

ONE MORE EXAMPLE: $6 + 8 * 4 / 9 - 5$

1. We see and output 6
2. We see and hold onto +
3. We see and output 8
4. We see and hold onto *
5. We see and output 4
6. We see / This is an operator with equal precedence to the previous one, so we output the previous one (*), and hold onto the /
7. We see and output 9
8. We see - This is an operator with lower precedence than the previous one (/) so we bring back and output the /
9. Now we compare the - to the other operator we are holding onto (+). The + also has precedence $\geq$ the new operator so we output the + too.
10. We see and output 5
11. 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 is what stacks do, and it is exactly what we need for our infix-to-postfix algorithm. Here's the algorithm (on its own page for readability).

```
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
```

S is used to store operators until we are ready for them.

We don't add operands (ie. values) to the stack, we just pass them through to the output list.

```
        else if t is a left parenthesis "(":
            S.push(t)
```

Left parentheses go directly to the stack.

```
        else if t is a right parenthesis ")":
            x = S.pop()
            while x != "(" :
                append x to postFix
                x = S.pop()
```

For a right parenthesis, pop off all stored operators back to the matching left parenthesis.

```
        else:
            if not S.isEmpty():
                x = S.pop()
```

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 before this operator.

```
                while precedence(x) >= precedence(t):
```

Pop stored operators until we find one with lower precedence than t.

```
                    append x to postFix
```

It's time for this saved high-precedence operator to be added to the postFix expression.

```
                    if not S.isEmpty():
                        x = S.pop()
```

Get the next operator from the stack.

```
                    else:
                        break
```

Exit the while loop because the stack is empty.

```
                if precedence(x) < precedence(t):
                    S.push(x)
```

If the last thing we removed from the stack has lower precedence than t, it needs to go back on the stack.

```
            S.push(t)
```

The new operator always goes on the stack because its right operand has not been seen yet.

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

We have reached the end of the infix expression - we need to clear out any operators still in the stack.

Add any operators still in S to the postfix expression.

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.