Stacks

Tim "Dr. T" Chamillard

Introduction

A stack is a data structure with specific access characteristics. Think of making and using a stack of books. How do we do it? Well, to add a book to the stack, we place it on the stack; in geek-ese, this is called pushing the book onto the stack. To remove a book from the stack, we take the top book off the stack; this is called popping from the stack. Note that we're only allowed to take the top book off the stack, we can't remove a book from the middle of the stack, the bottom of the stack, or anywhere else. We can also look at the top book on the stack without removing it from the stack; this is called peeking at the stack.

Because the last thing we added to a stack is the first thing we remove from the stack, a stack is a LIFO (Last In First Out) data structure. We can implement the stack using an array, a list, or a variety of other underlying structures, but any stack implementation needs to enforce LIFO access.

Stacks are useful for solving numerous problems. When we explore recursion, we'll look at the call stack in the debugger to see how the recursive method calls are organized. We'll essentially be using the call stack to store our intermediate results as we go through the recursion. Using a stack to organize method calls is a standard technique, because we always want to return from the methods we call in reverse order, making a LIFO data structure the perfect choice. Another example where stacks are useful is in compilers, especially when we need to match open and closing parentheses or curly braces. One final example – stacks can be very useful when doing path-finding and other searching tasks in artificial intelligence problems.

Although stacks can expose many operations, we'll implement a stack that provides a Count property and Push, Pop, and Peek methods to support both our understanding of how a stack could be implemented and our complexity analysis of those operations.

The `System.Collections.Generic` namespace in C# provides a `Stack` class that does the same things we'll do below and more. The underlying implementation in that class is an array rather than a list, but we get the same behavior from Push, Pop, Peek, and Count. In practice, you should simply plan to use the `Stack` class provided in C# when you need a stack (and you will).

Instance Variables and the Constructor

Let's start building our stack by adding the instance variables and a constructor. We're of course going to make our stack generic so we can store any data type on the stack.

Here's the start of our `Stack` class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Stacks
{
    /// <remarks>
    /// A stack
    /// </remarks>
    class Stack<T>
    {
        List<T> contents = new List<T>();

        #region Constructor

        /// <summary>
        /// Constructor
        /// </summary>
        public Stack()
        {
        }

        #endregion
```

The only instance variable we need is something to hold the contents of the stack. We're using a list for this, but you have a variety of choices for that data type. You should also notice that we're creating the new list object when we declare the `contents` field. Another option would be to just declare the `contents` field without initializing it and actually create the new list object in the constructor. Either way works fine, though I prefer just doing it when I declare the variable.

We also seem to be occasionally writing constructors that don't do anything! We of course need a constructor so we can instantiate the class to create new objects of the class, so this isn't wasted effort. You'll also find that lots of constructors initialize instance variables and do other initialization tasks; we just don't need any of that in our `Stack` class.

C# will actually automatically give us a no-argument constructor for the classes we define, but I prefer to explicitly include the constructor in my code.

Count

The `Count` property simply returns the number of items currently on the stack. Because this is exactly the same as the number of items in our `contents` field, this is a pretty simple property:

```
        #region Properties

        /// <summary>
```

```
        /// Gets the number of items in the stack
        /// </summary>
        public int Count
        {
            get { return contents.Count; }
        }

        #endregion
```

Notice that we can get the value of this property but we can't set it. This makes sense, because we shouldn't be able to directly manipulate this count. The only way we can (indirectly) change this property's value is by pushing items on and popping items from the stack.

Algorithmic Complexity

The documentation for the `List Count` property says "Retrieving the value of this property is an O(1) operation." That means the get for our `Count` property is also O(1).

Push

The `Push` method adds a new item to the top of the stack. A list doesn't have a top and bottom, but it does have a front and back. We'll simply treat the back of our list as the top of the stack. Here's the method:

```
        #region Public methods

        /// <summary>
        /// Pushes the item on the stack
        /// </summary>
        /// <param name="item"></param>
        public void Push(T item)
        {
            contents.Add(item);
        }
```

So, it might seem more intuitive (at least it was to me) to treat the front of the list as the top of the stack. Why didn't we do that instead? Read the next section to find out!

Algorithmic Complexity

The complexity for the `List Add` method is just like for the `Add` method for our dynamic arrays. If we don't have to grow the list, this is an O(1) operation, but if we do have to grow the list this is an O(n) operation.

Why didn't we treat the front of the list as the top of the stack? Because then we'd have to use the `List Insert` method to push onto the stack (passing 0 as the location). So what, you might be thinking! The `Insert` method is an O(n) operation when we add to

the front of the list (do you see why?), and we'd always see the worst case because we'd always be inserting at the front of the list.

So basically, we could pick between a Push that's almost always O(1) and a Push that's always O(n). Hopefully you see why we picked the O(1) approach.

Pop

The `Pop` method removes the top item from the stack and returns it. If the stack is empty, the method throws an `InvalidOperationException`. Here's the code:

```
/// <summary>
/// Pops the top item from the stack
/// </summary>
/// <returns>the popped item</returns>
public T Pop()
{
    // throw exception if try to pop from an empty stack
    if (contents.Count == 0)
    {
        throw new InvalidOperationException(
            "Can't pop from an empty stack");
    }
    else
    {
        // retrieve top of stack, remove, and return
        T item = contents[contents.Count - 1];
        contents.RemoveAt(contents.Count - 1);
        return item;
    }
}
```

Algorithmic Complexity

Accessing the `contents.Count - 1` element of the list is an O(1) operation. Unfortunately, `RemoveAt` is an O(n) operation, where n = Count – the index of the element we want to remove. The good news, though, is that we're always removing the element at the end of the list; when we do the math, we find that in our case `RemoveAt` is O(n) where n = Count – (Count – 1) = 1. Pop is therefore an O(1) operation.

Peek

The `Peek` method also returns the top item on the stack, but it does NOT remove that item from the stack. Like the `Pop` method, if the stack is empty the `Peek` method throws an `InvalidOperationException`. Here's the code:

```
/// <summary>
/// Peeks at the top item on the stack
/// </summary>
/// <returns>the top item</returns>
```

```
public T Peek()
{
    // throw exception if try to peek at an empty stack
    if (contents.Count == 0)
    {
        throw new InvalidOperationException(
            "Can't peek at an empty stack");
    }
    else
    {
        // return top of stack
        return contents[contents.Count - 1];
    }
}
```

Algorithmic Complexity

As noted above, accessing the `contents.Count - 1` element of the list is an O(1) operation, so Peek is an O(1) operation.

Testing the Stack Class

You should take a look at the `TestStack` code in the provided project to see how we can thoroughly test the `Stack` class. My original set of test cases did some fancy building of expected results to try to build really general test cases. Unfortunately, they didn't catch the fact that I had implemented my `Pop` and `Peek` methods incorrectly! I went back and simplified them because they weren't working.

In general, here's what we test in that class:

Stack Property/Method	**Tested Characteristics**
Constructor	Count is 0 Stack has no items
Count	Correct based on # of items on stack
Push	Count increases by 1 Contents reflect new item on top of stack
Pop (empty stack)	InvalidOperationException thrown
Pop (non-empty stack)	Correct item returned Count decreases by 1 Contents reflect item removed from top of stack
Peek (empty stack)	InvalidOperationException thrown
Peek (non-empty stack)	Correct item returned Count doesn't change Contents of stack don't change