CMPSCI 187: Programming With Data Structures

Lecture 10: The ArrayStack Class
30 September 2011

The ArrayStack Class

- Implementing a Stack With an Array
- A Generic Stack is an Array of What?
- The Fields and the Basic Methods
- Stack Overflow and Dynamic Resizing
- Running Time Analysis
- Exception Handling
- Project 3: The Dropout Stack

Implementing a Stack With an Array

- An array is a powerful data structure -- we have **random access** to it, meaning that we can get out any item if we know its address.
- A stack has limited access, only at the top, which is not hard to simulate with an array.
- Arrays are fixed in size, while stacks (at least in principle) have unbounded memory available.
- The idea is that we will mark the position of the top of the stack, which will move within the array as we push and pop.
- Pushes and pops will not affect the position of the other elements and so may be implemented in O(1) time.

A Generic Stack is an Array of What?

- We want to create a **generic** class ArrayStack<T>, where T will be a variable ranging over a type inside our code. An ArrayStack<Dog> will be implemented by an array of Dog elements. Inside our code we will speak of an array of T elements, whose type is T [ ].

- But in our constructor we can't say "T [ ] stack = new T[capacity];" because the interpreter can't create objects that don't come from a clearly defined class.

- So instead we say "T [ ] stack = (T[ ]) (new Object[capacity]);", creating an array of Object elements and **casting** it into a T [ ] variable. The compiler will generate a warning but allow the cast.

- The array really *is* an Object array, but as long as all the objects in it *are* T objects, we won't have problems with overloaded methods.

The Fields and Basic Methods

- ArrayStack<T> has two fields: `int top` and `T[ ] stack`, plus a constant `DEFAULT_CAPACITY`.
- We push a new element on by increasing top by one and setting `T[top]` to the new element. The only complication comes when the array is full.
- We peek by returning `T[top-1]`, unless of course the stack is empty.
- We pop by returning `T[top-1]` and decrementing `top`, again unless the stack is empty (`top == 0`).
- The size of the stack is always `top` even if it is 0, so the isEmpty method just returns `(top == 0)`.

Stack Overflow and Dynamic Resizing

- As we said, arrays have a fixed size but stacks do not.
- We could use an ArrayList instead of an array, but instead we will just redevelop the ArrayList's **dynamic resizing**.
- When our stack fills the whole array and we get a push, we double the size of the array. (Actually we make a new array that is twice as big, copy the old array into the new one, and replace the original with the new one.) Then we can implement the push as normal.
- If we cared about using an array that was too big, we could downsize any array that was less than half full. But L&C do not do this.
- What's the cost of this resizing?

Running Time Analysis

- The five basic stack operations are pretty clearly O(1) time each, *except* that a push might cause a resizing. So *in the worst case*, pushes are O(n), where n is the total number of inputs being processed by the stack.

- Suppose I start with an array of size 100 (as in L&C) and push 6400 elements onto it. It gets resized six times during that sequence of pushes, to 200, 400, 800, 1600, 3200, and 6400.

- Creating and copying into an array of size n takes O(n) time. We have about log n resizings of O(n) time each, for O(n log n) time.

- But it's better than that -- the total time is proportional to 100 + 200 + 400 + 800 + 1600 + 3200 + 6400 = 2*(6400) - 200. In general, the *total* resizing time for n pushes is still proportional to n, which we call O(n). We say that the **amortized** time per push is still O(1), though the worst-case time is O(n).

Exception Handling

- We need our class to behave reasonably if the user attempts to peek or pop when the stack is empty. We can't implement the action, but we can throw an exception.

- L&C define an EmptyCollectionException in their `jss2.exceptions` package. We'll use this to signal stack underflow.

- Since we want our pop and peek methods to be able to throw this exception to their calling method, we need the clause `throws new EmptyCollectionException("Stack")`. The "Stack" tells the user what kind of Collection threw the exception.

- The other methods can't cause this exception and thus have no throws clause.

Project 3: The Dropout Stack

- If we were not able to resize our array in ArrayStack, we would have to throw a StackOverflowException when we could no longer act like a stack.
- An alternative is to act *sort of like* a stack, discarding the oldest element in a full stack in order to make room for a new element.
- In Project 3 you'll implement this with a **circular array**, where both the top and the bottom of the stack move around the array as we push and pop. We will later use the circular array to implement a queue.
- You'll add a resize method to your DropoutStack class, which will let you define an extension of DropoutStack that is a fully correct implementation of the stack ADT.
- The project will be posted later today and is due Tuesday 11 October.