Stacks: The Hp-35 Calculator

Ruxandra-Stefania Tudose

Fall 2023

Introduction

This report aims to walk through the implementation of the HP-35 Calculator, but most importantly to ultimately utilize the obtained understanding of the stack through this real-life example in order compare and contrast the dynamic and static versions of this data structure by running a benchmark using its core operations: *push and pop*.

The General Approach

I started this assignment by first setting up the HP-35 Calculator. In other words, I implemented the missing methods for the different operations and as indicated in the file, I decided to work with an abstract class in order to create the two different classes for the dynamic and static stack. Although it is not the focus of this report, I checked the calculator's accuracy using the Reversed Polish Notation, which served as a really helpful example in terms of a concrete way of how we can make use of stacks, since I was able to follow the execution of the mathematical expression by displaying on the screen the variation of the counter and the value included or excluded from the stack.

Now, the next two sections will dive into the different ways of implementing the two methods that stay at the core of this data structure: *push(int value)* and *pop()*.

The Static Stack: Implementation

When initialized, the static stack is provided an initial, fixed size of the array, whose pointer is set to -1, namely one position away from the actual first position in the stack. Every time an element is pushed, the pointer increases and the new element is placed on the stack. Once the maximum space is reached and since the array is not flexible in terms of changing its size, the program simply throws an exception. On the other hand, once

a value is popped, the pointer decreases and the element is removed from the stack. If the stack is empty (the pointer reaches value -1) and the pop method is called, an exception will be thrown once again.

The Dynamic Stack: Implementation

Similarly, when initialized, the dynamic stack is provided an initial, fixed size of the array, whose pointer is set to -1, however once the maximum size is reached, the size of the stack is increased. All elements are copied to a temporary array of double length, whose reference is then attributed to the initial one and finally, the new element is pushed, while the pointer is increased. Therefore, no exception is thrown as the full dynamic stack case is dealt with by increasing the array every time we need it. As far as the pop method is concerned, apart from returning the needed value, its new feature is presented under the ability of shrinking the array once there is a certain amount of unused space in the array (this topic will be furter explored after the Time Analysis section). The counter decreases when an element is popped and if the stack is empty, an exception is thrown.

Time Analysis

First and foremost it should be taken into consideration that given this time analysis, the minimum execution time will be used for drawing further conclusions. A new class called Bench has been created, which consists of methods that measure the execution of pushing and popping elements for a given number of times. Although it presents no particular interest to our analysis, a call of the time measurement method has been executed in order to prevent time mispredictions caused by the cache memory.

As it can be noticed from the table below, the dynamic stack is on average **4.6** times more expensive in terms of the execution time compared to the static stack. In spite of this conclusion, it should also be taken into consideration that the dynamic stack has been improved timewise (e.g: given the push operation, the stack's size is always doubled every time it gets full and **not** increased by a fixed value).

Moreover, it should also be noticed that the ratio between the static and dynamic time execution is unlikely to be predictable. And this unpredictability is actually *inherited* from the dynamic execution time, which varies during execution and is not linear compared to the static stack (as the number of operations doubles, the time approximately does the same). That is precisely why, the columns for the individual static and dynamic time execution have been included in the table.

NoOfOps	Static	Dynamic	Ratio
100	0.15	0.58	4.0
200	0.29	0.93	3.2
400	0.57	2.22	3.9
800	1.10	5.59	5.0
1600	2.22	13.17	5.92
3200	4.38	24.15	5.51

Table 1: The ratio between the static and dynamic stack time execution given the number of operations.

Overall Slight Difficulties

The problems I encountered during this assignment are both codewise and implementationwise, which will now be further discussed:

Implementing the shrinking dynamic stack

As my stack pointer is initialised with the value -1, I had to insert a double condition to check when shrinking the stack should take place. And that is because I popped the value whenever the pointer was not equal to -1. At the same time, -1 would always be less than a quarter of the stack's size. I, therefore, added the second condition for the counter to be different than -1 as presented below:

```
if(counter < dynstack.length/4 && counter != -1) {
    int [] shrinkstack = new int [dynstack.length/2];
    ...
}

if(counter != -1)
  return dynstack[counter--];
else
  throw new ArrayIndexOutOfBoundsException ("Empty pop");
```

When should the shrinking of the dynamic stack take place?

Before joining the lecture on Friday, I used to shrink the stack when the unused space was less than half of the stack size minus one. However, I realised that this alternative is shrinking the stack a bit too early and that it highly relates on luck since all my next operations could be push operations. This way, what I was apparently saving in terms of memory, I was paying on the other hand, in terms of time by copying all elements every time in order to extend the stack immediately after having reduced its size. I, therefore,

decided to shrink it to half when the unused space was less than a quarter of the stack's size.

Conclusion

All in all, the implementation of a dynamic and static stack varies quite a lot. If I were to find myself in a case when I knew with high certainty the space I would need to allocate and that under no circumstances that space would be exceeded during execution, I would definitely pick the static stack in order to solve the given problem. This way, the execution would be fast, time would be predictable (linear increase) and the memory usage efficient. On the other hand, if allocated more than what is needed in terms of space, the static stack easily becomes a waste of memory usage.

However, since rarely is this the case in real life, dynamic stacks are a more viable solution. Although at first they might seem slower (time is lost copying all elements when extending or shrinking the stack), in time (through the amortised cost) it will make use of the extra allocated space in memory and it will turn into a more convenient solution thanks to its flexibility. In other words, it balances the time and memory efficiency by proportionally shrinking or increasing the stack.

Criteria	StaticStack	DynamicStack
timewise	faster	slower
memorywise	unneccessary allocated space	flexible stack
overall usage	in limited situations only	a more viable option

Table 2: Analsys of the static and dynamic stack.

Note: In the table above, it should be taken into consideration that the static stack is not memory efficient if one allocates more space that the problem actually requires (e.g: allocate a stack of 32 and only use 8 positions). Moreover, the *overall usage* refers to the situations when the either types of stacks can be used efficiently.