HP-35 calculator in Java

Kinga Anna Koltai

Fall 2024

Introduction

In this second assignment I am going to implement the HP-35, the world's first **scientific pocket calculator** in a terminal. It will be using the reverse Polish notation[1] and a stack (implemented with arrays). The main goal of the assignment is to see how a stack can be used and to get familiar with its implementation using arrays.

Implementing the stack

First I am going to look at two possible implementations of a stack, the main difference being that one of them is static and the other is dynamic.

the static stack

For the static stack the main focus was getting used to handling the stack pointer. I decided to set it to 0 to point to the first element of the array, on position 0. For the `push()` operation, I am incrementing the pointer after I wrote an element to where it points, so that it always points at the top of the stack, and for the `pop()` I decrement before returning the value.

Below is the code snippet for the `push()` and `pop()` operations:

```
public void push(int val) {
    if (top<stack.length) stack[top++]=val;
    else System.out.println("Stackoverflow");
}
```

As you can see, before doing anything, I first check if there is still space to write on the stack. For now I just decided to print an error message showing me that the if-statement worked as intended, but in the final code I would use exceptions and include some code that terminates the calculator

[1] https://en.wikipedia.org/wiki/Reverse_Polish_notation

program, because there is no logical way to move forward with a full static stack.

```
public int pop() {
    if (top>0) return stack[--top];
    else return Integer.MIN_VALUE;
}
```

Similarly in the `pop()` I also check to see it makes sense to perform the operation by checking if there is any value stored on the stack. If there is none, I decided to return the smallest value the `int` datatype can hold, because it seems rather unlikely to come up in calculation, and I also added a `print()` method that translates `Integer.MIN_VALUE` into text telling the user that the `pop()` was unsuccessful.
For testing purposes I removed popped elements from the array (set them to 0) to be able to see how it works.

```
public int pop(){
    for (int i = 0; i<this.stack.length; i++){
        System.out.println(this.stack[i]);
    }
    int tmp = stack[top];
    stack[top] = 0;
    return tmp;
}
```

Using this I confirmed that my stack pointer worked consistently, always pointing at an empty position or at the end of the array.

the dynamic stack

Next I implemented a dynamic stack using very similar functions. The only difference is that when our static stack returned a `Stack Overflow` exception, this dynamic stack doubles in size instead and copies all the elements from the previous array to the new one. Another important question was whether to decrease the size back if the stack empties out again. The task warned us about the possibility of increasing the size from e.g. 8 to 16 and then right back to 8 and maybe to 16 again. Considering this and the fact that since the calculator operates on a PC and has a decent amount of available space on the stack, we are very unlikely to run out of memory and so I would not implement this decrease mechanism. However, if memory was an issue I would probably implement it in the `pop()` method, decreasing the array size only when the `top` index has been consistently below half the size for a while (e.g. for 5 consequent iteration).

benching the stacks

For both versions of the stack I decided to benchmark the push operation since that is the one that can get very costly after a certain size.
First I set up my benchmark to measure pushing n elements onto a static stack k=200 times and taking the minimum. The way I implemented the static stack would create an infinite loop in case of a stack overflow, so I only measured the time it takes to push n elements on a stack of size n. For the dynamic version on the other hand, I was curious to see how costly it really was to copy everything over to a new array, so I decided to always start with an array of 100 elements. This way my first measurement goes without creating a new array.
During setup I made the rookie mistake of not resetting my `min` variable in the loop, so I kept measuring a constant time complexity which was of course rather suspicious. After fixing this error, I measured the following:

n	exec time (static stack) [µs]	exec time (dynamic stack) [µs]
100	0.2	0.2
200	0.5	0.5
400	1	1.3
800	1.8	3
1600	3.7	4

I concluded that for these sizes there are no significant differences between the two (except maybe for n=800, but I suspect that is not really that bad either, might just be due to rounding or some other factors.

The calculator

Using the dynamic version of the stack, I implemented a calculator that can perform the 4 basic mathematical operation (+,-,*,/). It takes one number or operand per line, and needs an additional enter to get the result at the end. Its working mechanism is based on the reverse polish notion which works very well with a stack. It uses a string reader to read from the input and converts numbers to an int before pushing them on the stack until an operand comes, in which case it pops the two topmost elements and performs the given operation on them, pushing the result back onto the stack. It was my decision to use the dynamic stack, it could obviously also be implemented using the static one. There are, as always, advantages and disadvantages for both versions, however, for this use case I believe the dynamic stack comes out on top. The only comparable solution would be giving a static stack a rather big working space so that we are less likely to run into a stackoverflow. I personally decided on the dynamic, because having the calculator stop working after you entered a good amount of numbers would

simply be far more annoying than having to wait a little for it to generate the new array, especially since a lot of the times I would assume more time is spent on figuring out what comes next in the polish notation rather than waiting on the computer.
During testing I noticed that my initial solution could not handle non-integer values (except of course for the operands), so if it encountered for example the letter 'a' it terminated with an error. I added a try-catch expression so that when the value entered cannot be converted to int and pushed on the stack, it writes the message "Not a number or operand(+,-,*,/). Please enter a single digit or operand" on the screen and allows the calculator to continue working.
Also found that because of my value for an unsuccessful `pop()` was the minimum value for an int, if I for example only pressed enter without entering any calculations, the result I got was the minimum value for ints. To fix this, I added some code at the end of my program to check whether there is a (seemingly) correct result. My thought process was that if...

1. ...the top of the list is already at 0, we have nothing to pop()
2. ...the top of the list is higher than 1, we have several values on the stack

...and therefore cannot correctly evaluate the result. In this case I print out a message for the user. It would be a good idea to also clear the stack and jump back to the beginning of the program letting the user try again after a failed attempt, but this would require careful implementation and I am just a lazy programmer so for the purpose of this assignment, I leave it up to the user to restart the calculator program if they did not get the desired result.