ID1021: A calculator

Alexander Lundqvist

01-09-2022

Introduction

This report we try to implement a calculator that can perform calculations on mathematical expressions written in reverse polish notation (RPN). To implement the calculator we utilize the stack data structure. A stack can be described as an array where

We then try to benchmark the different implementations to see how they get affected by different sizes of mathematical expressions.

Lastly we use the calculator to calculate the last digit in our personal number.

Method

To start off we create the main project following the instructions in the task. I soon realized that I wanted to separate the code into different classes for better management and problem solving. The Main class controls the program flow and is also responsible for the benchmark testing of the different implementations.

Static stack

The instructions suggested to implement the static stack with only integers and with a small size of 4, but I wanted some flexibility in the calculator and I also wanted to be able to handle decimals (As all online RPN calculators were able to do it) so the static stack can have any size since it gets determined by the size of the expression array.

Dynamic stack

The dynamic stack code is mostly the same as the static stack apart from the resizing method which is described below.

```
private void resize(int amount) {
    Item[] newStack = new Item[amount];
```

```
        for (int i = 0; i < this.stackPointer; i++) {
            newStack[i] = this.stack[i];
        }

        this.stack = newStack;
        newStack = null;
    }
```

This code works by creating an auxiliary array where the size is determined by the amount parameter. The code allows for both increasing the array size as well as decreasing it. As we don't want to resize the stack all the time (which leads to more stress on the CPU) we always double the size when we want to increase it. This gets determined when the value of the stack pointer is larger than the size of the stack, as the stack pointer points to the top of the stack not the last element. You can however use a stack pointer that points to the element at the top of the stack, however this can be bothersome when emptying the stack as you have to account for negative values.

To decrease the stack size we go the other way and look for when the stack pointer has reached a quarter of the stack size, then we halve the size of the stack. This might look weird but it is done to leave some air in the stack in case we start adding elements to it again and want to avoid any unnecessary increasing of the stack.

```
    if (this.stackpointer > 0 && this.stackpointer == size()/4)
            resize(size()/2);
```

Calculating last digit

To be able to perform this calculation I had to implement two extra mathematical operators. These were just added to the end of the calculator code and are described here. The code is quite self explained as Java already has the modulus operator built in.

```
    case MOD : {
                double x = this.stack.pop().getValue();
                this.stack.push(new Item(x % 10));
                break;
            }
            case MULX : {
                double y = this.stack.pop().getValue();
                double x = this.stack.pop().getValue();
                double sum = y*x;
                if(sum < 10){
```

```
            stack.push(new Item(sum));
        }
        else {
            this.stack.push(new Item((sum - 10) + 1));
        }
        break;
    }
```

Result

Here are the results from the benchmark testing and also the result of the calculation of last digit of my personal number.

Benchmarks

prgm	**100**	**1000**	**10000**
Static	0.17ms	0.47ms	1.44ms
Dynamic	0.12ns	0.66ms	1.64ms

Table 1: Test of the different implementations for different expression lengths.

Calculate your last digit

There isn't really much to say about the result. I am omitting my own personal number here as this report will be available online but the calculations were correct as the last element in the stack was 7 which is the last digit in my personal number.

Discussion

Since I have taken this course before I already have some knowledge in how to implement the stack data structure. As such, no problems arose during the actual development of the calculator. However, the concept of reverse polish notation was foreign to me which led to quite a bit of research into how it works.

The biggest issue was regarding multiple digit integers. As I didn't want to hard code anything I had some problems to figure out how to differentiate between integers as 123+ could evaluate as either 12 + 3 or 1 + 23. Fortunately after some reading regarding how actual physical calculators using RPN worked, being separating each integer input with a special enter-button, I got the idea. I first implemented a simple method but realised that for the benchmark it would be more interesting with large and uniform expressions so in the end I settled on making an expression generator where the user could define the size.

There is no error handling implemented in the code whatsoever, but I figure I should've implemented my own errors to deal with things like stack overflow, out-of-bounds errors and null values which would've helped with the development but as alot of input is hard coded I feel that it is enough for this exercise as it took longer to solve and debug the code then I wanted.

Static vs. dynamic stack

Since a static stack has a fixed size the time complexity will always be constant for push and pop operations. The same can be said for the memory complexity as we allocate a fixed memory size. However, an issue with a static stack is that you could potentially waste memory since we don't resize the stack when popping items. If the stack doesn't gets eaten by the garbage collector, then we are left with allocated memory that is not being used.

The dynamic stack on the other hand gets resized when adding additional elements by using an auxiliary array. As the size of the stack increases, the stack will get doubled and while you would normally not use mathematical expressions that use thousands of operators this can get out of hand when using the stack for other purposes. Say for instance you have a random stack with some arbitrary elements, when you have filled your stack of 100MB (Crazy number yes, but just to illustrate.) and want to add another element, you would (in my code) make a new stack of 200MB of memory.

In normal computers this wouldn't be an issue but this could potentially be tricky when dealing with microsystems and built-in systems as they often have very little memory.

I didn't have any time to perform any discussion on the last task.