Calculator

Axel Månson Lokrantz

Spring Fall 2022

Introduction

In the 'Calculator' assignment the students were asked to implement a calculator capable of calculating expressions using the reverse polish notation.

Task 1: Static stack

The calculator itself was required to do fundamental mathematical operations such as addition, subtraction, division and multiplication through a very simplistic design using a stack and an instruction pointer. To understand how a basic stack works, the students were asked to make a static stack, which basically translates to a stack with a fixed stack size, and a dynamic one which can expand / collapse depending on the input data.

The basic structure of the calculator was already pre-written as well as a few functions to push / pop values and mathematical operations such as ADD or SUB.

```
public class StaticStack {

    public int[] staticStack = new int[4];
    int sp = -1;

    public void push (int i){
        if(sp == 4){
            System.out.println("Stack is full.");
        }
        else{
            staticStack[++sp] = i;
        }
    }
    public int pop(){
        if (sp == -1){
            return Integer.MIN_VALUE;
```

```
        }
        return staticStack[sp--];
    }
}
```

The code for our StaticStack can be seen above. We originally set the stack size to 4, which worked well with example expression from the assignment. The stack pointer is set to -1 to avoid any out of bound errors, but it could have been 0 as well with a few adjustments in the code. If the user tries to insert too many values an error message appears and if the users tries to pop an already empty stack the program returns MIN_VALUE. With a little more time it would have been better to throw an exception with a catch block instead.

The timings did not change when the stack size and input data was adjusted, therefor the algorithm has a performance of O(1).

Task 2: Dynamic Stack

The dynamic stack was a little more complicated since we had to find a solution on when to expand or collapse the array depending on the amount of elements. The solution that was used in our program was whenever the stack is full, we double the size of the stack. So a 4 element long array would expand to 8, 16, 32 ... elements. To collapse the stack two conditions had to be full filled: The stack needs a minimum size, which was set to 4. The other condition was, if the stack pointer is pointing on an element which is 1/4 of the total stack size we divide the stack by 2. When doubling the stack 9 times we got an array with the size of 1024. This pattern repeats for 19, 29 etc. Log2(1024) is 10, therefor we have an algorithm of Olog(n).

```
public class DynamicStack {

    int stackSize = 4;
    int sp = -1;
    public int[] dynamicStack = new int[stackSize];
    public void push (int i){
        if(sp == (stackSize-1)){
            extendStack();
        }
        sp++;
        dynamicStack[sp] = i;
    }
    public int pop(){
        if (stackSize >= 4 && sp <= (stackSize/4)){
            reduceStack();
```

```
        }
        return dynamicStack[sp--];
    }
    private void extendStack(){
        stackSize *= 2;
        int[] tempStack = new int[stackSize];
        for (int i = 0; i < stackSize/2; i++){
            tempStack[i] = dynamicStack[i];
        }
        dynamicStack = tempStack;
    }
    private void reduceStack(){
        stackSize /= 2;
        int[] tempStack = new int[stackSize];
        for (int i = 0; i < stackSize; i++){
            tempStack[i] = dynamicStack[i];
        }
        dynamicStack = tempStack;
    }
}
```

Task 3: Benchmark

Below is the code used for the benchmark tests.

```
public double benchmark(){
        int k = 1_000_000;
        long t0 = System.nanoTime();
        for(int i = 0; i < k; i++){
            run();
            resetPointer();
        }
        long t_access = (System.nanoTime() - t0);
        return ((double)(t_access))/((double)k);
    }
    public void resetPointer(){
        this.ip = 0;
    }
```

The test is very similar to the testing done in the previous assignment. The program does 1.000.000 loop iterations and finds a median value. Throughout the testing we noticed that no matter how many iterations we did the time remained about the same. This was because the ip pointer got stuck at the last element. To fix the issue we had to implement a method to

reset the pointer which solved the issue. The calculator was given the same problem over and over again which is not optimal considering the cache. A better benchmark test would have been to randomize the problem for each loop iteration.

Below you can find the results from some of the benchmark testing. The static stack is roughly twice as fast as the dynamic stack.

Static stack	Dynamic stack	Ratio
74.2	172.4	2.3
76.1	163.3	2.1
72.6	122.7	1.6
85.7	181.2	2.1
82.6	180.0	2.1

Table 1: Run time in nanoseconds.

Task 4: Luhn algorithm

In the fourth and final task we were asked to calculate the last digit in our personal number using Luhn algorithm in reversed polish notation. To be able to compute the calculation we had to expand our calculator with two new operations - modulo 10 and a special multiplier where 9*2 equals 9. The multiplier first takes any digit and multiplies it with 2, if you are left with two digits, let's say 18 for example, you would end up with 1+8 = 9.

```
case SPECIAL : {
                int x = stack.pop();
                x = x*2;
                if (x >= 10){
                    int x2 = x/10;
                    int x3 = x%10;
                    x = x2 + x3;
                }
                    stack.push(x);
                    break;
            }
            case MOD10 : {
                int x = stack.pop();
                x = x % 10;
                stack.push(x);
                break;
            }
```

The following expression was inserted in revered polish notation where val1 represent 1, val2 represent 2 etc. The program returns a 7, which also happens to be the last digit in my personal number.

```
Item[] expr = {val10, val9, spec, val0, add, val0, spec,
add, val1, add, val2, spec, add, val5, add, val2, spec,
add, val9, add, val5, spec, add, mod, sub};
```