A Calculator

Isak Wilkens TIDAB

Fall 2022

Introduction

This is a report on the second assignment "A Calculator" on the course "ID2021" TIDAB, KTH. The assignment is to implement a calculator that can calculate mathematical expressions in *reversed Polish Notation*. We do this since the notation only requires a stack data structure to function and will be a good exercise in how a stack data structures works. The calculator will firstly be implemented using a static stack and then with a dynamic stack. To test the calculator further than the most simple tasks it will be trialed with the "Luhn-algorithm" as a final test. The two different types of stack will then be bench-marked against each other to analyze pros and cons.

The calculator

The calculator uses an instruction array and a stack array with pointers to these respectively. The framework for running the calculator and a constructor for the instruction types is given in the assignment. The task is to implement the calculator functions, the stack(s) and the means to benchmark the two.

calculator functions

The functions that were implemented are add, subtract, multiply, divide, take number value, modulo10, and a special version of multiply by two. This special version takes a number, multiplies it with two and then separates the numbers of the result and adds them up.

Here follows three examples of the implemented functions:

```
case DIV : {
    int y = stack.pop();
    int x = stack.pop();
    stack.push(x / y);
    break;
```

```
        }
        case VALUE : {
            stack.push(nxt.value);
            break;
        }
        case SPECIAL : {
            int x = stack.pop();
            x *=2;
            if (x >= 10 ){
                int x3 = x % 10;
                int x4 = x / 10;
                x = (x3+x4);
            }
            stack.push(x);
            break;
        }
```

Stack implementation

static stack

The static stack was implemented with a fixed size and a stack pointer starting at -1, outside of the stack. When something is pushed onto the stack the pointer is incremented and the pushed value is added to that position in the stack array. The pop function copies the value on the current position of the stack pointer and decrements the pointer. This assignment does not focus on exceptions so if a value higher than the fixed size is pushed onto the stack pointer the program will throw the standard out of bounds exception and if the stack pointer is popped too low (-1) the function will just return an extremely low value so that the user can see that something went wrong. Since the static stack is fixed no matter the user input it will have a time complexity of O(1), unchanged with regards to data size. The code for the static stack is shown below:

```
public class Stack {
    private int[] stack = new int[10];
    public int sp = -1;
    public void push(int value){
        if(sp == 10){
            System.out.println("STACKOVERFLOW!");
            return;
        }
        sp++;
        stack[sp] = value;
```

```
}
public int pop(){
    if(sp == -1){
        return Integer.MIN_VALUE;
    }
    int value = stack[sp];
    sp--;
    return value;
}
```

dynamic stack

The stack now gets added functionality to where is size is dynamically changed after need from the input of the user. The dynamic stack starts at the array size of four, if another value if attempted to be pushed onto the stack when its pointer is at four the stack size will be doubled through the function "extendStack" (see code below). A new stack double the size is created and the values of the current one is copied onto it. Similarly when values are popped from the stack and the stack pointer decremented enough the stack size will halved. To prevent the stack size form being reduced unnecessarily for example in a function that does both pop and push the stack size is not reduced immediately but instead when the stack pointer is at one fourth the length of the stack. There is also a conditional statement making sure that the stack length is not shorter than four as a minimum value. The same procedure of coping over the values onto a new stack is used when reducing the stack size. The dynamic stack has a time complexity that depends on user input and in this case since it is doubled after a certain input, its on the logarithmic scale. If the starting value is 2 it will have to be doubled nine times to reach a stack size of 1000. So the time complexity for the dynamic stack will be O(log(n)). The code for the dynamic stack is shown below:

```
public void push(int value){
    if(sp == (stackSize-1)){
        extendStack();
    }
    sp++;
    stack[sp] = value;
}
public int pop(){
    if(sp == -1){
        return Integer.MIN_VALUE;
    }
    if( stackSize >= 4 && sp <= (stackSize/4)){
```

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

Luhn-algorithm

To be able to use the calculator for Luhn's algorithm the previously mentioned special multiplication had to be added as a function. Since we cannot work with parentheses and a large calculation is to be deducted from 10 it needs to be input in a special way to this calculator. The normal way to write it is (special multiplication noted as $*'$) $10 - ((y1 * 2 + y2 + m1 *' 2 + m2...)mod10)$ but when working with *reversed Polish Notation* and this calculator the 10 is put in first into the stack. Then the following first digits are added to each other with every other one manipulated by special multiplication. Then modulo 10 is used to deduct the last digit of this sum to lastly use the subtract function which will then take the 10 we put into the stack memory and then deduct this last digit from 10. In my case the final number is one, the sum is 39 and the final subtraction will then be $10 - 9 = 1$.

Benchmarks and Conclusion

Benchmarks were done with an added code function which was used to run the calculation from 10 up to 1.000.000 times with both stacks. The

calculation that was done was to push 10 number 3s onto the stack and then add the last two. This means that the dynamic stack had to double twice. The average time was taken from these runs to compare the two stack methods. The results showed for the first runs that the dynamic stack indeed was slower but then when there were a large number of repetitions they looked similar. The reason for this similarity, apart from the given reduction when there is a larger number of tries that shares the delay, the software seems to be doing something to optimize making the difference between the two hard to spot. Improvements that might have given a better result would have been to use randomized numbers, analyse minimum values as well ass mean which is an interesting indicator of algorithm efficiency and also maybe to try with different stack sizes where the stack is also reduced. The measurements for the bench-marking is shown in table 1. The code for the bench-marking is shown below:

```
public double benchmark(){
    for (int i = 0; i < j; i++){
        run();
        resetPointer();
        stack.resetStack();
    }
    t_total = (System.nanoTime() - t0);
    return t_total/(double)k;
}
```

Repetitions(j)	**10**	**20**	**50**	**100**	**1000**	**10000**	**100000**	**1000000**
static(ns)	110k	50k	15k	14k	5,2k	0,83k	0,47k	0,13k
dynamic(ns)	210k	160k	64k	40k	5,1k	1,6k	0,40k	0,13k

Table 1: Measurements of calculating the input on the static and dynamic stack. Run-time in nano seconds.

In conclusion the dynamic stack is great since it can adopt to user input without crashing but as seen from benchmarks we loose some performance. Static memory is somewhat faster and requires less code and conditional statements. Physical memory in a computer is precious and to not over use, in this case and many other the dynamic stack gives more advantages than the static one.