Report Assignment 2: Stack

Thomas Gustafsson

September 2022

Introduction

The purpose of this assignment is to learn how a *stack* can be used. All code is written in Java. The goal is to implement a calculator that can calculate mathematical expressions described using *Reverse Polish Notation, RPN*. The HP35 calculator is a well known of using this method of calculating. It is simply writing mathematical expressions with the operand last. Instead of writing 2 + 4 normally, it is instead written 24+ in the HP35 calculator. Using RPN no parenthesis is needed. In a stack, the basic operations *push* and *pop* are used to add and remove items. The stack is a first in - last out system. A way to visualize this is by picturing a pile of plates. To pick out the second to highest plate, the highest plate needs to be taken out of the pile first.

Using the stack we can push a value to the stack, push another value, and use an operator type such as "ADD" or "DIV" to perform the calculation: Pop the second value, pop the first value and push them back to the stack with the operation applied (eg. push first + second).

An expression

First thing to do is to create a way to represent an expression. Doing it as a string "2 4 + 2 1 - *" would require parsing of integers; but seeing as it is not relevant for the purpose of the assignment, I am going to represent an expression by an array of *items*. Each item is a two element object with a type and a value. Continuing from the code snippets given in the instructions, inside the *Item* class:

```
public Item(int value) {
    this.value = value;
    this.type = ItemType.VALUE;
}

public Item(ItemType itemType) {
    this.value = 0;
    this.type = itemType;
}
```

Now the Item class almost is complete. An item will have a value and a type. But first a set of enums should be created, a step in making parsing avoidable: ADD, SUB, MUL, DIV, VALUE. More on that in the next section. In the class, a method getValue and another method getType are created which return value and type.

The calculator

The format of the code given in the instruction of the assignment if continued upon in the *Calculator* class because it seems like an effective way around the problem. In method *step()*, cases for all operands are added. Following the structure:

```
case MUL -> {
    int y = stack.pop();
        int x = stack.pop();
        stack.push(x * y);
}
```

And this is done similarly for the operands just different in what's pushed. SUB means stack.push(x - y);. For an integer entered when using the calculator is the case VALUE:

```
case VALUE -> {
    stack.push(nxt.getValue());
}
```

Implementing the stack

Now for the stacks. One static, and one dynamic. They are used separately for testing different things. A requirement is to implement them from scratch without using any Java libraries (except output libraries).

a static stack

The first stack has a fixed size of four. As an array, the stack navigated by using a stack pointer and manipulated by created methods *push* and *pop*. Simply put, adding and removing values to then perform an operation and push the result back into the array.

```
int pop(){
        return stack[pointer--];
}

void push(int i){
        stack[++pointer] = i;
}
```

The pointer points to the top of the stack, but when the stack is empty the value of the pointer is zero. Pushing a value on a full stack will result in a stack overflow, and this can be communicated to the user by throwing a RunTimeException error when trying to add an item to the Calculator when stack array is full. Same thing can be done when someone pops an item from an empty stack, to throw an error saying that the stack already is empty.

a dynamic stack

An actual solution for a push operation resulting in stack overflow is extending the size of the stack. This is done by allocating a new larger array and copying the items from the original array to the new one. It's more complex but can be fixed with about five lines of code for the within pop and push method.

Inside push:

```
if(pointer == stack.length) {
        int[] newStack = new int[stack.length * 2];
        for(i=0;i<stack.length;i++) {
                newStack[i] = stack[i];
        }
    stack = newStack;
}
```

Inside pop:

```
if(pointer < (stack.length/2 - 1)) {
        int[] newStack = new int[(stack.length / 2)];
        for(int i=0;i<stack.length/2;i++) {
                newStack[i] = stack[i];
        }
        stack = newStack;
}
```

The way the dynamic stack now works is if the stack is full when trying to push a value, a new array double the size as the current one will be allocated. The values in the current stack will be copied into the new stack. Now, the new stack is set to be the current stack. It works similarly when trying to pop a value. If the highest item in the stack is at an index lower than half of the stack size, the items will be transferred to a new array half the size of the current one. Could be done better though. An improvement could be halving the stack when the highest item is at an index 1/4 of the stack size. To avoid unnecessary array transfers.

benchmarks

Benchmarks are done by using *System.nanoTime()* at two points which enables seeing the resolution time of the two implementations. Short and simple, it

seems like the time complexity varies depending on how big of a stack the program is working with. Seems to be O(1) at smaller sizes, but once the arrays become substantially larger it shows times of about thirty to forty percent compared to the static stack. So, O(n) for the dynamic stack.

calculate your last digit

To calculate the last digit of a personal number was a bit more difficult to figure out the solution to. We are given the rest of the numbers. But following the same structure as before, a case is created as following:

```
case MOD -> {
        int y = stack.pop();
        int x = stack.pop();
        stack.push(x - (x/y)*y);
}
```

..and introducing modulo as an operation. To then calculate the last digit is done by multiplying each digit by 2,1,2,1 meaning multiplying every second digit by 2. Then the sum of this by modulo 10, then subtracting the rest by 10. The last number of the personal number will be shown.