HP35 Calculator Report.

Adrian Jonsson Sjödin

Fall 2022

Introduction

In this assignment an old fashion calculator that uses reverse polish notation will be implemented. The purpose of this is to gain an understanding of how a *stack* is constructed and works.

Task 1

Complete the code for the different classes outlined by the teacher and implement the stack. Implement both a static stack and a dynamic stack that can expand and shrink. They questions that should be answered are:

- Does the pointer point to the location above the top of the stack or does it point to the top of the stack?
- What is the value of the pointer when the stack is empty?
- What should you do when a program tries to push a value on a full stack?
- What should happen when someone pops an item from an empty stack?

Finally do some benchmarking of the different stack and examine how well the two implementations works.

Method

Completing the code for the different classes **Item, ItemType** and **Calculator** was straightforward since it was just adding some constructor and getters, and complete the case statement in the **Calculator** class.

The **Stack** class however needed to be implemented from scratch. This though is quite straightforward for a static stack since all it need is to be able to push values to the stack, pop values from the stack, and check wether or not the stack is empty or full. This is done by using a pointer to index through the array that represent the stack as seen in the code bellow.

```
public void push(int element) {
  if (isFull()) {
    throw new StackOverflowError("Stack is full");
  }
  arr[++pointer] = element;
}
```

We check if the stack is full by simply looking at what value the pointer is set to, and as long as it is not equal to the length of the array we are free to push to the stack. This is done by first incrementing the pointer and then add the element to the stack. If the stack is full we throw a StackOverflowError.

```
public int pop() {
  if (isEmpty()) {
    throw new EmptyStackException();
}
  return arr[pointer--];
}
```

Popping the stack is implemented in a similar way, with the difference being that we check if the pointer is set to -1, which would mean that the stack is empty. If it is we throw a EmptyStackException, otherwise we return the element that the pointer is currently pointing to and *then* decrement the pointer.

To implement a dynamic stack that can be rescaled one need only implement two new methods in the stack class. Instead of throwing a StackOverflowError when `isFull()` is called, we instead call the method `expandArray()`, which takes the current stack and copies it to a new array with double the capacity.

Similarly to reduce it we add the method `reduceSize()`, which looks at what the pointer is pointing at and if it is less than half the stack capacity it reduces it by half the capacity.

```
public void reduceSize() {
  int curr_length = index + 1;
  if (curr_length < capacity / 2) {
    int[] new_array = new int[capacity / 2];
    :
    }
}
```

Result

To benchmark the different stacks we measured the best time out of n runs that it took for the calculator to calculate an expression with 500 values

followed by 499 add operations.

Stack	Number of calculations	Time in ns	Time increase
Static	1 000	2439	-
Dynamic	1 000	10010	4.1
Static	10 000	2251	-
Dynamic	10 000	6410	2.8
Static	1 000 000	1702	-
Dynamic	1 000 000	5033	2.9

Table 1: Stack benchmarks

Discussion

A static stack is as expected faster than a dynamic stack with a time complexity of $\mathcal{O}(1)$, since it never needs to copy the whole stack over to a bigger/smaller stack. So when it comes to just the time complexity of the different stacks the static stack is better than the dynamic stack with its time complexity of $\mathcal{O}(n)$. However the advantage of the dynamic stack is that it doesn't tie up as much memory as the static stack, since it adjust itself depending on how much space is needed and can also be implemented when one doesn't know exactly how much space might be required. However in cases where one knows in advance how large a stack is required a static stack would be the better alternative.

Task 2

Use the calculator from task 1 to calculate the last digit in your social security number according to the formula

$$10 - ((y_1 *' + y_2 *' + m_1 *' m_2 *' ...) mod_{10})$$

The $*'$ operator means that any multiplication that gives > 10 should instead be though of as $a + b$. To be able to calculate your last digit implement this special operator as well as the modulus operator into the calculator.

Method

The mod_{10} operation is simple to implement. We simply need to add it as a type in the **ItemType** enum class and add the following to the case statement in the **Calculator** class.

```
case MOD -> {
  int y = stack.pop();
  stack.push(y % 10);
}
```

The special operator is little bit tricker but not by much. What I implemented was a method that takes the value at the top of the stack and add together the result of integer division and modulo division by 10, and then pushes it back to the stack. This can be seen in the code bellow.

```
case MULTOADD -> {
  int x = stack.pop();
  int y = x / 10;
  int z = x % 10;
  stack.push(y + z);
}
```

Having implemented these two things we simply need to implement everything correctly into the calculator.

Result

Inputting,

$$10, x, x, *, *', x, x, x, x, x, *, *', x, x, x, *, *', +, +, +, +, +, +, mod$$

where x are my specific social security numbers left blanked for security reason, into the calculator gives me my last digit.

Discussion

When implementing the special operator I choose to implement a version that required that the user knew in advanced that the previous multiplication operation would result in a value greater than 10. I did this because of time constraint and thus choosing the easiest and quickest operation. However for this particular task it could have been better to implement it so that after every multiplication operation the program checked if that multiplication resulted in a value greater than 10, and if it did then immediately do the special operation, thus negating the need to enter the special operator into the calculator. This is however only if the calculator's only job is to calculate the last digit of a social security number, since it would break the multiplication operation for regular operations.

All code for this assignment can be found here: GitHub