HP35

Pontus Hector

Fall 2022

Introduction

This is the second assignment in the course ID1021. In this task one explores the functionality of stacks by creating a dynamic and static stack for a calculator with polish notation and compare their differences.

Static stack implementation

The static stack was static in the meaning that it had a fixed and predetermined size before the calculator starts pushing values onto it. In the static stack an index variable was used to point to the index of the most recently inserted element in the stack (the element at the top of the stack). To insert (or push) an element, we incremented the stack pointer and then placed the new element at that index. Similarly, to extract (or pop) the element, the element on the current index was ejected and then the stack pointer index was decremented. An empty stack was represented by an empty queue where the stack pointer is equal to -1. With this implementation the time complexity for using the static stack is constant (O(1)) since the stacks size is not dependent on the amount of elements pushed onto the stack.

Dynamic stack implementation

The dynamic stack should be able to grow as we add more items when that is if the original array is overflowed. To do this a method was created that kept track of the current index of the stack pointer and compared it to the size of the stack each time an element was pushed or popped. If the next position in the stack was equal to the current size of the array when **pushing**, a new array of twice the size was created where all the elements from the previous array were copied to the new array. To enlarge the current size of the dynamic stack the following method was used:

```
public void expandArray() {
                int curr_size = index + 1;
                int[] new_array = new int[curr_size * 2];
```

```
        for (int i = 0; i < curr_size; i++) {
                new_array[i] = stack[i];
        }
        stack = new_array;
        capacity = new_array.length;
}
```

Similarly if the current arrays length was equal to half of current size of the stack when **popping** a new array of half the size was created where all the elements from the previous array were copied to the new array. To reduce the current size of the dynamic stack the following method was used:

```
public void reduceSize() {
        int curr_length = index + 1;
        if (curr_length < capacity / 2) {
                int[] new_array = new int[capacity / 2];
                System.arraycopy(stack, 0, new_array, 0, new_array.length);
                stack = new_array;
                capacity = new_array.length;
        }
}
```

With this implementation the time complexity for the stack is now in worst case O(n) since the stacks size is dependent on the amount of elements pushed onto the stack. The time complexity is in the best case O(1) but this only occurs whenever the dynamic stack does not need to expand. After the first expansion of the size of the stack, the time complexity varies between being O(log(n)) and O(n) depending on how many values are pushed to the stack in relation to its size. Right before an expansion of the stack size the time complexity is O(log(n)) while immediately after the expansion the time complexity is O(n).

Benchmark

In the benchmark we compare the time elapsed between the dynamic stack calculator and the static stack calculator when running the same amount of workload. Both of the calculators start by pushing 16 values onto the stack and then performing 15 operations onto the values. The values are randomized for each iteration (an integer between 1 and 10) to ensure that the compiler will not "guess" the answer on beforehand. The operations are the same for both calculators and no division is included since this sooner or later would lead to a division by 0 when running these values for hundreds-of-thousands of times.

The measurements below further confirms the reasoning that the dynamic stack calculator is slower than the static stack calculator. While this may be true it does not necessarily mean that the static stack calculator is the better calculator.

Iterations	Average runtime	Minimal runtime
100000	242ns	100ns

Table 1: Static stack with 16 pushed and popped elements, start-size 16

Iterations	Average runtime	Minimal runtime
100000	555ns	200ns

Table 2: Dynamic stack with 16 pushed and popped elements, start-size 4

The static stack calculator is faster since it does not need to expand and decrease the stack but this also means that the allocated memory is not adapted to the user-input. More often than not the static calculator will allocate either more memory than one needs, which is bad since memory is valuable, or less memory than one needs, which is bad since a stack overflow would occur.

Calculate your last digit

To calculate the last digit in ones personal number with the calculator provided one needs to implement a mod10 operation and an operation that may receive a number between 10-18 and break the number down. As an example 12 is treated as 1 + 2 when we do the final summation to calculate the last digit and to do this while storing values on the stack one needs to implement a new operation.

The mod10 operation is implemented through the following code:

```
int y = stack.pop();
stack.push(y % 10);
```

The *' operation is implemented through the following code:

```
int a = stack.pop();
int y = a / 10;
int x = a % 10;
a = x + y;
stack.push(a);
```

After these implementations the following input would be expected on a calculator using polish notation for the personal number 0101014454:

$10, 0, 1, 0, 1, 0, 1, 4, 2, *, 4, 5, 2, *, *', +, +, +, +, +, +, +, +, mod10, -$