HP-35 calculator: Looking back at an old algorithm through modern technology

Praanto Samadder

August 29, 2023

Abstract

The objective is to look at two implementations of stacks through implementing the classic Reverse Polish algorithm.

1 Calculator

The `Calculator` class is implemented similar to the problem sheet with one minor difference: the use of `StackType` enum passed through the constructor to use static and dynamic stacks. `expr` is marked with `private val` because of their relevance in the functions in the class.

```
class Calculator (private val expr: List<Item>, stackType: StackType?){
    private var stack: Stack
    private var instructionPointer = 0

    init {
        stack = if (stackType == StackType.STATIC) StaticStack(expr.size)
        else DynamicStack()
    }

    fun run() {
        while (instructionPointer < expr.size) { step() }
    }

    private fun step() {}
}

enum class StackType {
    STATIC, DYNAMIC
}
```

2 Stack

2.1 Stack interface

The stack interface is implemented

```
interface Stack {
    fun push(value: Int)
    fun pop(): Int
}
```

2.2 Static Stack

The size of the stack is determined in the

```
import removeLastAndReturnArray

class StaticStack(size: Int): Stack {
    private var array = IntArray(size)
    private var stackPointer = -1

    override fun push(value: Int) {
        stackPointer++
        array[stackPointer] = value
    }

    override fun pop(): Int {
        val r = array[stackPointer--]
        this.array = array.removeLastAndReturnArray()
        return r
    }
}
```

2.3 Dynamic Stack

Kotlin already has dynamic stacks built into it through it's built-in `mutableArrayOf()` function. However, we will not be using that for our purpose here.

The `push()` and `pop()` function are used to modify the size of the array as soon as an element is added or removed from the stack. Both `push()` and `pop()` functions will create a temporary array that is one element larger or smaller in size respectively and then copy the old array into the temporary array. In the final step, the temporary array replaces the main array.

`pop()` is making use of a Kotlin extension function that removes and

```
import removeLastAndReturnArray

class DynamicStack: Stack {
    private var array = IntArray(0)
    override fun push(value: Int) {
        val tempArray = IntArray(array.size + 1)
        for (each in array.indices) tempArray[each] = array[each]
        tempArray[array.size] = value
        this.array = tempArray
    }

    override fun pop(): Int {
        val lastItem = array.last()
        this.array = array.removeLastAndReturnArray()
        return lastItem
    }
}
```

3 Benchmark

Benchmarking the algorithm is done by measuring the time difference between beginning and the end of the algorithm and by measuring memory allocation of each stack implementation.

3.1 Time benchmark

We are using two functions: `autoPush()` and `autoPop()` to populate and de-populate our stack. The functions are implemented as follows:

```
private fun autoPush() {
    for (each in 0..< 1024) {
        stack.push(Random.nextInt(0, 1000))
    }
}

private fun autoPop() {
    for (each in 0..< 1024) {
        stack.pop()
    }
}
```

The benchmark is done with a test function.

```
@Test
fun benchmark() {
    val startTime = Date().time
    autoPush()
    autoPop()
    val endTime = Date().time
    println("Time taken: " + (endTime - startTime) + "ms")
}
```

3.2 Benchmark device

Benchmarking is done on a 2022 Apple MacBook Air with an M1 processor and 8GB of shared memory.

3.3 Results

When running this test, the static stack takes around 6ms to finish executing whereas