HP-35 Pocket Calculator using Stacks in Java

Malte Berg

HT 2024

Introduction

This report covers the second assignment assigned in the course ID1021. The assignment is to create a version of the HP-35 pocket calculator in Java. The calculator uses Reverse Polish Notation, a postfix notation where the operator comes after the operands.

Stacks

The implementation of stacks in the assignment is as self-written classes and not using pre-written libraries. Two stacks were to be implemented: static and dynamic. As to not complicate things too much, the stacks only hold integers instead of accepting any data type. The two main functions of a stack is `push()` and `pop()`, where `push()` adds, or ”pushes”, an integer on the stack and `pop()` removes, or ”pops”, an integer from the stack.

Stacks work with the Last-In-First-Out (LIFO) principle, where the last added integer is the first to be removed. Unlike an array, stacks only keep track of the index of the last added integer; it can only access the ”top” of the stack. To access integers lower down in the stack, one has to first pop above integers.

Static Stack

The static stack has a fixed size that is decided when the stack is created. Just like an array, it can only hold a set amount of integers before being full. If one tries to add more integers to the stack, a stack overflow occurs as an integer is added beyond the bounds of the stack.

Dynamic Stack

The dynamic stack has a variable size that increases or decreases during the course of the running program. The reasoning for the size increases and decreases for this implementation will later be presented.

Implementing Stacks in Java

The stacks implemented in this assignment are built from an abstract class `Stack`.

```
public abstract class Stack {
    int[] stack;
    int size;
    int top;

    public abstract void push(int val);
    public abstract int pop();
}
```

The class contains the two necessary methods for a stack, `push()` and `pop()`, the integer array that contains the integers, `stack`, the number of items of the stack, `size`, as well as the index for the latest added integer, `top`. From this, the classes `StackStatic` and `StackDynamic` were implemented.

Note: `top` *is initialized to* 0 *in the following implementations.*

Implementing Static Stack in Java

```
public StackStatic(int size) {
    if (size < 1) {
        throw new IllegalArgumentException(
                "Stack size has to be at least 1");
    }
    stack = new int[size];
}
```

When initializing the static stack, a size argument is passed. If the size is less than 1, an exception is thrown as the stack should be able to contain at least one item. As long as the size is 1 or greater, the stack is initialized.

```
public void push(int val) {
    try {
        stack[top++] = val;
    } catch (IndexOutOfBoundsException e) {
        System.out.println("Stack full.");
        throw e;
    }
}
```

When calling `push()`, a try-catch is used to try and add the integer `val` to the stack. If the stack is full, i.e. `top` > `size`, an exception is thrown. If not, the integer is added to the stack and `top` is incremented (`top++`).

```
public int pop() {
    int popped;
    try {
         popped = stack[--top];
    } catch (IndexOutOfBoundsException e) {
        System.out.println("Empty stack.");
        throw e;
    }
    return popped;
}
```

When calling `pop()`, a try-catch is once again used. If the stack is empty, i.e. `top` − 1 < `0`, an exception is thrown. If not, `top` is decremented first (pre-decremented with `--top`), and the top of the stack is returned.

Implementing Dynamic Stack in Java

```
int size = 4;
public StackDynamic() {
    stack = new int[size];
}
```

The dynamic stack has been implemented with a set size in when initializing. The size has been set to 4 as a reasonable smallest size. As the stack is dynamic, there is no reason to take in a size argument as the user should not have to worry about it and the program should do the work instead.

```
public void push(int val) {
    if (top == size) {
        size *=2;
        int[] newStack = new int[size];
        System.arraycopy(stack, 0, newStack, 0, top);
        this.stack = newStack;
    }
    stack[top++] = val;
}
```

The lack of a try-catch can be seen in the method `push()` in the `StackDynamic` class. This is not needed as the size of the stack will increase if needed. In the method, if `top` is equal to `size` when trying to push an integer, `stack` recieves double the space thanks to doing a copy of the current stack to a new stack, `newStack`. `stack` is then instead pointed to `newStack` which contains all the previous elements. When finished, or if `top` is not equal to `top`, the pushed `val` is added to the stack and `top` is incremented.

```java
public int pop() {
    /*
        [...]
    */

    if (size > 4 && top <= size / 2 - size / 8) {
        this.size /= 2;
        int[] newStack = new int[size];
        System.arraycopy(stack, 0, newStack, 0, top);
        this.stack = newStack;
    }
    return popped;
}
```

The code from the implementation of the static stack can be reused, with the addition of a new if-statement checking if it's reasonable to decrease the size of the stack. In this implementation, the conditions are:

- `size` > 4

If `size` is 4, there is no reason to decrease it as 4 was picked as the smallest reasonable size for the stack. If `size` is bigger than 4, it's fine to decrease.

- `top` $<=$ `size`$/2 -$ `size`$/8$

If `top` has decremented below a certain threshold it's fine to decrease the stack size. So what should the threshold be? For the current stack to fit in a new stack with half the size, `top` has to be at most the same value as `size`$/2$. To then make sure that popping and pushing in rapid succession at exactly `size`$/2$ does not increase and decrease the array size continuously putting excess stress with the `arrayCopy` method, the threshold to decrease is `top` $<=$ `size`$/2 -$ `size`$/8$ which adds a varying amount of small leeway depending on the stack size.

If this threshold is passed when popping, the stack size is halved just like the way it was doubled and all the values in the current stack are copied to the new one.

Running the program

The program takes input from the terminal using `BufferedReader` to allow the user to write in their own expression.

```
Stack s = new StackDynamic();
boolean run = true;
BufferedReader br = new BufferedReader(
                            new InputStreamReader(System.in));
while (run) {
    System.out.print("> ");
    String input = br.readLine();
    int first, second;
    switch (input) {
        case "+": [...]
        case "-":
            second = s.pop();
            first = s.pop();
            s.push(first - second);
            break;
        case "*": [...]
        case "/": [...]

    [...]
```

All the cases are the same inside with the exception of the operator. The reason the second value is popped first and the first value is popped second is due to the LIFO principle. If the code was written the other way around, the expression "5 3 −" would be resulting in a calculation of $3 - 5$ instead of the wanted $5 - 3$. Therefor the second value must be read in first.

```
    [...]
        case "":
            run = false;
            break;

        default:
            s.push(Integer.parseInt(input));
            break;
    }
}
System.out.printf("The result is: %d\n\n", s.pop());
```

If the user types a number, the number is added to the stack. If the user types nothing, the program is ended and the result of the calculation is printed, which is the lowest item in the stack if an expression is correctly typed. If for example the number of operators were to be more than can be used by the number of operands, an exception would occur as the program would try to pop from the stack when there are no available items, going below the bounds of the stack.

Given Example

The assignment had a given example to check if the calculator worked.

```
4 2 3 * 4 + 4 * + 2 -
```

This is, of course, The Answer to the Great Question of Life, the Universe and Everything.

```
4 2 3 * 4 + 4 * + 2 -
4 (2 * 3) 4 + 4 * + 2 -
4 6 4 + 4 * + 2 -
4 (6 + 4) 4 * + 2 -
4 10 4 * + 2 -
4 (10 * 4) + 2 -
4 40 + 2 -
(4 + 40) 2 -
44 2 -
44-2
42
```