A Calculator

Algorithms and data structures ID1021

Johan Montelius

Fall term 2022

Introduction

In this assignment you should implement a calculator that can calculate mathematical expressions described using *reverse Polish notation*. We do this in order to see how a *stack* can be used. In all your programming courses so far you have been working using a stack but you might not have used one explicitly like we shall do now. You have probably never heard of the reverse Polish notation but everything will be clear in a minute (or hour).

The HP-35 calculator and reverse Polish notation and the stack

Reverse Polish notation is simply writing mathematical expressions with the operand last. Instead of writing $5 + 3$ we write $53+$. This sounds weird but it has its advantages and the first programmable computer, Z3, used reveres Polish notation when expressing mathematical formulas (start googling now). If you took this class in 1972 rather than 2022 you would probably be the proud owner of a HP-35 pocket calculator that also used this form of entering expressions.

The good thing with reversed Polish notation is that you can do a away with parenthesis. The expression $(4 + 5) * 6$ is simply written $45 + 6*$. The biggest advantage is that we have a very simple way of calculating the result, all we need is a *stack*.

A stack is data structure that allows two basic operations: *push* and *pop*. An item can be pushed on the stack and is then at top of the stack. A pop operation will remove, and return, the item at the top of the stack, if the stack is not empty. The item below the removed item is then at the top of the stack. We can have other operations: check if the stack is empty, peek at a value some steps below the top of the stack etc but the two operations that changes the state is *push* and *pop*.

It turns out that the reversed notation and the stack are made for each other. If we have the expression 34+ then we simply push 3 on the stack, push 4 on the stack and then do the addition by popping two items from the stack and push the result back.

Take a pen and paper and go through the steps to calculate $34 + 24 + *$, if you end up with 42 on the stack you got the point.

An expression

The first thing we need to do is to represent an expression. We could of course represent it as a string `"3 4 + 2 4 + *"` but we would then need to do some parsing of integers which is not very interesting at this point. Let's represent an expression by an array of *items*. Each `Item` is a two element object with a `type` and `value`.

```
public class Item {
  private ItemType type;
  private int value = 0;
  :
  :
}
```

The type is of the class `ItemType` and determines if the item is an operation or a value. If it is a `VALUE` then the `value` element holds the value.

```
public enum ItemType {
  ADD,
  SUB,
  MUL,
  DIV,
  VALUE
}
```

The `Item` class will of course have methods to construct different items, return the type and return the value.

If we have the `Item` data structure we can construct an array of items that represent arbitrary complex arithmetic expressions in reversed Polish notation.

The calculator

The calculator it self is very simple in its design. It will hold an arithmetic expression (represented as an array of items in reversed Polish notation), a *instruction pointer* and a stack.

```
public class Calculator {

  Item[] expr;
  int ip;
  Stack stack;

  public Calculator(Item[] expr) {
    this.expr = expr;
    this.ip = 0;
    this.stack = new Stack();
  }

  :
}
```

When using the calculator we will provide the expression and then `run` the "program" by executing one `step` at a time.

```
public int run() {
  while ( ip < expr.length ) {
    step();
  }
  return stack.pop();
}
```

In each `step` we simply look at the type of the item and the switch to the right procedure. The `ItemType` must be visible to the `Calculator` so it should be in a separate file.

```
public void step() {

  Item nxt = expr[ip++];

  switch(nxt.type()) {

    case ADD : {
      int y = stack.pop();
      int x = stack.pop();
      stack.push(x + y);
      break;
      :
      :
    }
```

Now for the part that you need to work on - how do you implement the stack?

Implementing the stack

You will implement the stack in two versions, one static and one dynamic. You should implement them from scratch and you're not allowed to use for example ArrayList or other Java libraries to solve the problem (you're of course allowed to use output libraries).

a static stack

The first implementation will be a fixed sized stack let's say of size four (that is a small stack). The stack should then allocate an array of four item and keep track of a *stack pointer* (an index). For simplicity we implement a stack that can only hold integers.

The two methods `push` and `pop` are fairly simple to implement and the only thing you need to keep track of is the stack pointer and make sure that you do not push items outside of the array.

Questions you need to consider:

- 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 (stack overflow)?
- What should happen when someone pops an item from an empty stack?

a dynamic stack

Slightly more complex is to handle a stack that can grow as we add more items. In a push operation that would generate a stack overflow for the dynamic stack we simply extend the size of the stack by allocating a new larger array and copy the items from the original array to the new array. One question is how much larger the new stack should be, should we increase by only one item (no), a fixed amount (maybe) or something else?

Creating a larger stack should be only a few lines of code but how do you do if you should also be able to shrink the stack? Assume that you extend the stack from size 8 to 16 but then pop a few items, you then decrease the size to 8 again. Yet you do not want extend to 16 and then immediately decrease to 8 and then maybe immediately increase to 16. There should be some mechanism in the system that only decreases the size of the stack after a while.

benchmarks

Do some benchmarks where you examine how well the two implementations work. Is there a noticeable penalty when using the dynamic stack?

calculate your last digit

To calculate the last digit in a personal number you first multiply each digit (first 9) with 2,1,2,1,.... and then sum the result. The sum is then taken mod 10 and we subtract it from 10. The tricky part is that the multiplication is very special $6 * 2$ is of course 12 but this is treated as $1 + 2$ when we do the summation so the special multiplier could return 3 from the beginning.

If we use $*'$ for this strange multiplier and use a special operator for mod10 we could write it like this:

$$10 - ((y1 *' 2 + y2 + m1 *' 2 + m2...)mod_{10})$$

What does this look like in reversed Polish notation? Implement the mod_{10} operation and the strange $*'$ multiplier and calculate your own last digit.