# Exercises: Dynamic Structures (ADTs)

Adolfo Villafiorita

04.11.2024

# Premiss


If you want, continue with the exercises assigned last time.

(It is much better taking something from start to end, rather than keep starting new applications which are never completed... the difficult part is in the parts you add at the end!!!)

# Stack implementations: Array


With arrays: keep filling from top to bottom, extract from bottom

| | |
|---|---|
| 0 | top |
| 1 | |
| 2 | |
| 3 | current |
| 0 | |
| 0 | |

Make sure `current` is non negative (stack underflow) and does not exceed the array size (stack overflow!).

# Stack implementations: Linked lists


With a linked list. Possible implementations

1. Always add on top. Always remove from top. Need to take into account one corner case (top of the list is `nullptr`; need to use reference to pointers)

Other naive implmentations presented below are inefficient and complex to implement, providing no advantage, unless you are resuing code:

1. Always add to end. Always remove from end.
2. Double linked list. Efficient (you have instant access to the last element … but when you do, you are basically using it like the proposed implementation)

# RPN Calculator


Operands first, operators last

```
2 3 + => 2 + 3
3 4 + 6 * => (3 + 4) * 6
```

- Used by Hewlett-Packard calculators
- Does not require parentheses
- Evaluation based on a stack

# RPN Calculator in CPP


Write a program which waits for user input and if the user input is:

- a number: pushed onto the stack
- an operation (+, -): first two elements of the stack are pulled, operator applied to the operands, and the result pushed back onto stack
- `s`: the first two numbers in the stack are swapped
- `p`: the stack is printed (from top to bottom)
- `q`: the calculator quits

# Other math functions


Implement also the following useful functions:

- `^` elevates the second number in the stack to the power of the first number in the stack: `2 3 ^ => 8`. Use library functions or implement your own.
- `sqrt` computes the square root (better if you use library function in `cmath`)
- (`eql` puts 1 in the stack if the two first two numbers in the stack are equal, `0` otherwise.)

# Variations: Registers


Implement a register system. The registers work as follow:

- the store to register X instruction takes the top of the stack, removes it from the stack, and stores it in register X
- the recall from register X instruction takes the value currently stored in register X and puts it on top of the stack

Simplifying hypothesis: we have three registers, called A, B, C and the corresponding (new) commands: `sto_a`, `sto_b`, `sto_c`, `rcl_a`, `rcl_b`, `rcl_c`

# Variations: Registers (II)


Implement an infinite (ahem, large) number of registers.

- the store instruction now takes two arguments from the stack, the register number and the value to store.
- recall takes one argument from the stack, the register from which the value has to be taken.

(Hint: you need a linked list to store register values.)

Examples.

- `10 3 store` stores number 10 in register 3
- `3 recall` takes the value of register 3 and puts it into the stack

# Variations: read a full set of instructions


Modify the program so that it can read a sequence of instructions from file (or stdin) and applies them in order.

Example

```
2 3 + 4 5 + * // returns 45
```

… if you think about it, you are writing your first language interpreter.

## Variations: program + args


Modify the program so that it takes the following arguments:

- the first argument is the name of the file storing a sequence of instructions for the calculator
- the remaining arguments are numbers to seed the stack with

```
./a.out pitagora.txt 3 4
```

What happens when “pitagora.txt” has the following text?

```
sto_a 2 ^ rcl_a 2 ^ + sqrt
```