Think about:

Design a special stack that has a getMinimum operation with $\Theta(1)$ time complexity (and the other operations have $\Theta(1)$ time complexity as well).

(Trick: use a secondary stack)

<u>Requirements</u>:

- Describe the idea used to solve the problem
- Give the stack representation
- Draw the DS containing elements 37, 93, 25, 11 added in this order into an initial empty container
- Draw the DS containing elements 37, 93, 25, 11, 32, 71, 7 added in this order into an initial empty container
- Implement operation push

Special stack with a getMinimum operation with $\Theta(1)$ complexity (and the other operations have $\Theta(1)$ time complexity as well).

The idea used to solve the problem:

- Keep an auxiliary stack, containing as many elements as the original stack, but containing the minimum value up to each element.
- Use an existing implementation for the stack and work only with the operations from the interface.
- Call the auxiliary stack a **minStack** and the original stack the **elementStack**.

Representation:

SpecialStack:

elementStack: Stack

minStack: Stack

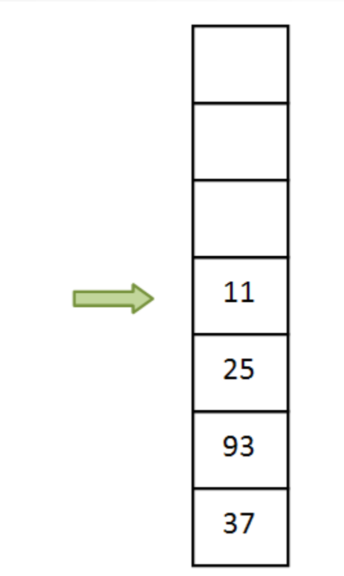
Special stack with a getMinimum operation with $\Theta(1)$ complexity (and the other operations have $\Theta(1)$ time complexity as well).

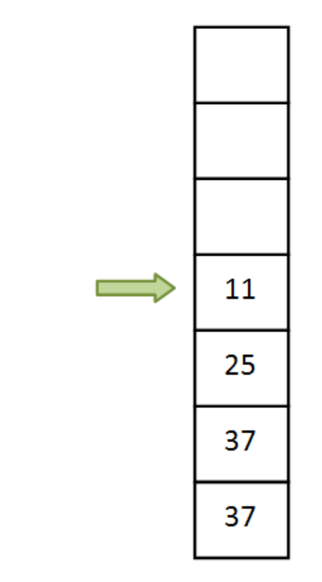
Description of the idea (continuation):

- When a new element is pushed to the element stack, we push a new element to the min stack as well. This element is the minimum between the top of the min stack and the newly added element.
- When an element is popped from the element stack, we will pop an element from the min stack as well.
- The getMinimum operation will simply return the top of the min stack.
- The other stack operations remain unchanged (except init, where you have to create two stacks).

Special stack with a getMinimum operation with $\Theta(1)$ complexity (and the other operations have $\Theta(1)$ time complexity as well).

- Draw the DS containing elements 37, 93, 25, 11 added in this order into an initial empty container



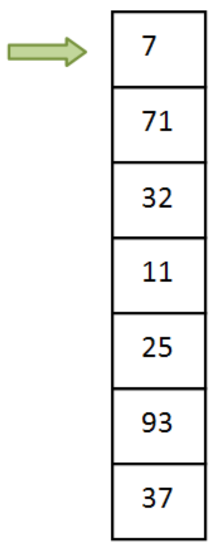


elementStack

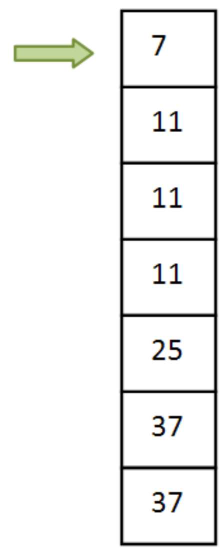
minStack

Special stack with a getMinimum operation with $\Theta(1)$ complexity (and the other operations have $\Theta(1)$ time complexity as well).

- Draw the DS containing elements 37, 93, 25, 11, 32, 71, 7 added in this order into an initial empty container



elementStack



minStack

Special stack with a getMinimum operation with $\Theta(1)$ complexity (and the other operations have $\Theta(1)$ time complexity as well).

```
subalgorithm push(ss, e) is:
    if isEmpty(ss.elementStack) then
        push(ss.elementStack, e)
        push(ss.minStack, e)
    else
        push(ss.elementStack, e)
        currentMin ← top(ss.minStack)
        if currentMin < e then
                push(ss.minStack, currentMin)
        else
                push(ss.minStack, e)
        end-if
    end-if
end-subalgorithm
```

Think about:

Consider ADT Quartiler which contains integer numbers and has the following operations (with the specified complexity requirements).

- init(q) - creates a new, empty Quartiler : $\Theta(1)$: total compl.
- add(q, elem) - adds a new element to the Quartiler q $O(\log_2 n)$
- getTopQuartile(q) - returns the element closest to the 75th percentile. If there is no such element, throws an exception. $\Theta(1)$: total compl.
- deleteTopQuartile(q) - removes the element closest to the 75th percentile. If there is no such element, throws an exception $O(\log_2 n)$: total compl.

Explanation:

- the 75th percentile (called 3rd quartile as well) of a sequence is a value from the sequence, the one below which 75% of the values from the sequence can be found if we sort the sequence. So, if you have the values from 1 to 100 (in any order), the 75th percentile is the value 75. If you have the values 1,2,3,4, the 75th percentile is 3. In case of a tie (for example, if you have values 1,2,3,4,5,6 the value 4 or 5 can be returned).

On short:

- we could consider 3^{rd} quartile as the element on position Round(n*3/4) in the sorted sequence.

Think about:

ADT Quartiler:

- Describe the representation

Assume: We have an implemented binary heap DS, named BinHeap, with the following operations:

init(bh, ...)
use: " $\leq$ " for MIN binary heap
" $\geq$ " for MAX binary heap
add(bh, elem)
top(bh) => elem
remove(bh)
isEmpty(bh)

What is the time complexity for each operation?

Think about:

ADT Quartiler
Describe the representation.

<u>Quartiler</u>:

```
        n:  Integer              // total number of elements
        heap1: BinHeap           // MAX-binary heap
                                 // keeps first 75% of the elements
                                 // (the smallest)
        heap2: BinHeap           // MIN-binary heap
                                 // keeps the largest 25% of the elements
```

Explanation:
element closest to the 75th percentile is top of heap1

=> getTopQuartile – is top of heap1 $\Theta(1)$

add or remove: could use move from heap1 to heap2 (or reversed)

+ add or remove to/from one of the two heaps $O(\log n)$