Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Übungen zur Vorlesung Informatik II (D-BAUG) FS 2017

D. Sidler, F. Friedrich

http://lec.inf.ethz.ch/baug/informatik2/2017

Solution to exercise sheet # 6 **27.3.2017 – 4.4.2017**

Problem 6.1. Selfgrowing Array

By now you have already gained some experience with arrays: once the size of an array is fixed, it cannot be changed. Any attempt to write to an array index bigger then the size of the allocated array will cause a Java exception. (Error and termination of the program if not handled).

The aim of this assignment is to implement an array of integer numbers that provides adaptive growth.

The public interface for the growing array is as follows:

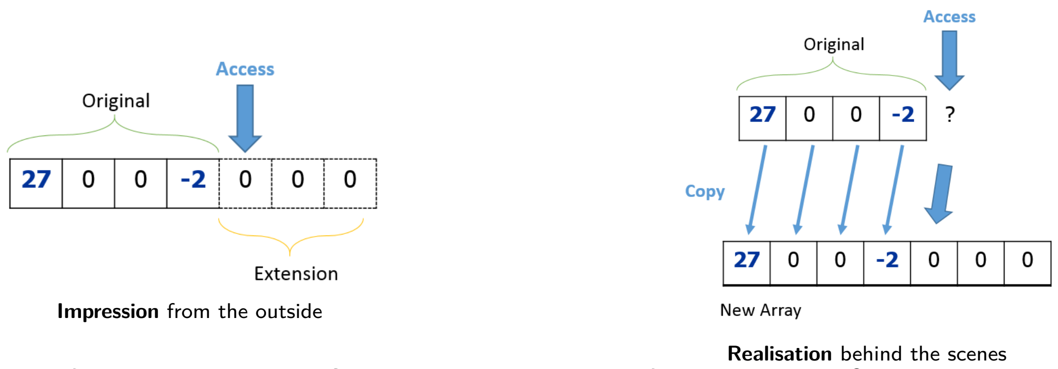
```java
class GrowingArray {
    // constructor
    public GrowingArray()

    // pre: any index ind >= 0
    // post: return value at position ind
    // If no value has been set to ind previously, return 0
    public int get(int idx)

    // pre: any index ind >= 0, any value val
    // post: store value val at position ind
    public void set(int idx, int value)

    // post: returns size of currently allocated storage
    public int size()
}
```

By giving the user access via this object's interface (instead of direct access to an array) it is possible to hide the actual implementation. Actually, there would be many different possibilities to implement such a dynamic array. For this exercise, it should be implemented as indicated by the following illustration:



On the left you can see how the GrowingArray should appear from the outside. Originally it has a size of 4. Then the user tries to access index 4, the array grows automatically to the next power of two which would be 8. On the right you can see how this functionality is implemented internally. As you know an array can not grow. So when the array has to be extended, a new array is allocated with the new size (next power of two). Then all values from the old (smaller) array are copied into this new array and finally the old array is replaced with the new one.

To insert and retrieve value from this array we provide a `main` function which parses the following commands:

```
set 1 5
get 1
```

The `set` command inserts a value (2nd number) at the specified index (first number). The `get` command retrieves the value at the specified index. To verify our implementation at the end of the `main` function all values in the array are printed out.

Next power of two

Growing the array exactly to the size required to host a newly requested element can result in a huge number of allocations when sequentially traversing and resizing the array. In order to avoid this, the array should be grown to the next size which is a power of two. For instance: if the user requests Element 931 we grow the array to size 1024 which corresponds to 2^{10}.
Your first task is to implement the function that returns the next power of two given any input number. You can find the code template at `https://codeboard.ethz.ch/inf2baugex06t01`.

Resize

Finally your task is to implement the function

```
private void resize(int size)
```

which is called within `set`. As motivated earlier it is supposed to

- create a new array using the NextPowerOfTwo function,
- copy the old (smaller) array into the new array,
- replace the internal storage (called "data" in the skeleton code) with the new storage.

Solution of Problem 6.1.

```
class GrowingArray {
        private int[] data;

        public GrowingArray() {
                data = new int[4]; // default size
        }

        private int getNextTwoPower(int num) {
                int result = 2;
                while (result < num) {
                        result *= 2;
                }
                return result;
        }

        private void resize(int size) {
                //create new array
                int tmp[] = new int[getNextTwoPower(size)];
                for (int i = 0; i < data.length; i++) {
                        tmp[i] = data[i];
                }
                //Assign new array to data
                data = tmp;
        }

        public int get(int idx) {
                if (idx >= data.length) {
                        return 0;
                }
                return data[idx];
        }

        public void set(int idx, int value) {
                if (idx >= data.length) {
                        resize(idx+1);
                }
                data[idx] = value;
        }

        public int size() {
                return data.length;
        }
}
```

Problem 6.2. Water Level Monitoring The objective of this exercise is to monitor the levels of several rivers and at any time to be able to report statistics about the level such as, in this simplification, the minimum and maximum value monitored so far. Naturally, this is best implemented using a class that offers an appropriate interface supporting adding new data points and retrieving statistics.

In this task you are asked to implement the class `Level` such that the code offered in the exercise (shown below) works as expected. We do not provide more information about what `Level` has to look like, this is part of the exercise. Make sure you use information hiding (encapsulation) appropriately. Most importantly, first read the code carefully in order to understand what kind of operations `Level` has to support. You can safely assume that any test input always provides at least one data point for each river.

```java
public class Main {
    public static void main(String[] args) {
        // data come from System.in via a scanner
        Scanner scanner = new Scanner(System.in);
        // two rivers Limmat and Sihl are being monitored
        Level limmat = new Level();
        Level sihl = new Level();
        // read in a sequence of data points until "ende" is found
        String river;
        do{
            // read data in the form S <number> or L <number>
            // standing for water level of Sihl or Limmat respectively
            river = scanner.next();
            if (river.equals("S")){
                sihl.put(scanner.nextDouble());
            }
            else if (river.equals("L")){
                limmat.put(scanner.nextDouble());
            }
        } while (!river.equals("ende"));
        // output characteristics.
        System.out.println("Limmat min " + limmat.min() + " max " +
            limmat.max());
        System.out.println("Sihl min " + sihl.min() + " max " + sihl.max());
        scanner.close();
    }
}
```

Implement the class `Level` in this Codeboard project: https://codeboard.ethz.ch/inf2baugex06t02.

Solution of Problem 6.2.

```
class Level{
    // private fields initialized accordingly
    private double min = Double.MAX_VALUE;
    private double max = Double.MIN_VALUE;

    //pre: at least one data point has been entered previously
    //post: report minimum of data previously provided via put
    public double min(){
        return min;
    }

    //pre: at least one data point has been entered previously
    //post: report maximum of data previously provided via put
    public double max(){
        return max;
    }

    //pre: provide one data point
    public void put(double v){
        if (v < min) min = v;
        if (v > max) max = v;
    }
}
```