Assignment 1 – Doubling Stack (4 points)

The class `DoublingStack` maintains a stack as doubly-linked list of arrays where each array has twice the size of the previous one. This ensures that new memory does not need to be allocated in every `push` operation, but only if the respective top array is full. The following sketch visualizes a doubling stack after pushing the five elements 6, 8, 2, 7, 7.

Given below is the program code of the class (as well as the helper-class for the `Node`).

```cpp
struct Node { // for one array in the list
    int* values; // data members
    int size;
    Node* next;
    Node* prev;

    Node(int s, Node* p)
    : values(new int[s]), size(s), next(0), prev(p) // constructor
    {};

    ~Node() { delete[] values; } // destructor

    Node(const Node& node) { ... } // copy-constructor (not printed)

    Node& operator=(const Node& node) { ... } // assignment operator (not printed)
};

class DoublingStack {
    Node* first; // data members
    Node* top:
    int pos;

    DoublingStack() :
    first(new Node(1, 0)), top(first), pos(0) // default constructor
    {};

    DoublingStack(const DoublingStack& ds) {...} // copy-constructor (not printed)

    DoublingStack& operator=(const DoublingStack& ds) {...} // (not printed)

    // POST: puts value on the stack
```
Solve the following tasks!

i) How many arrays does an initially empty DoublingStack contain after its function push has been called \( N \) times for
   \( N = 7 \)
   \( N = 33 \)

ii) Implement the destructor \(~\text{DoublingStack}\) such that it frees all allocated memory!

```cpp
~DoublingStack() {
    Node* node = first;
    while (node != [1]) {
        Node* tmp = [2];
        node = [3];
        delete [4];
    }
}
```

iii) Implement the function pop of the class DoublingStack which first deletes the top array if it is empty, then removes the top element from the stack and returns its value!

```cpp
int pop() {
    if (pos == [5]) {
        top = [6];
        pos = [7];
        delete [8];
    }
    return [10];
}
```

This exercise can be handed in via Codeboard! However, if you prefer, you can also hand in your solutions on paper as before.

Submission:  https://codeboard.ethz.ch/ifmp16E13T1
Assignment 2 – Queue (4 points)

A queue is a data structure whose core functionality consists of the following two operations:

1. Insertion of a new element at the end (“element queues up”);
2. Removal of the first element (“element gets served”).

Implement a type `ifmp::queue` that supports these operations, with the following public member functions (and keys of type `int`).

```cpp
// POST: key is added as last element
void push_back(int key);

// POST: returns whether *this is empty
bool empty() const;

// PRE: !empty()
// POST: first element of *this is returned
int front() const;

// PRE: !empty()
// POST: last element of *this is returned
int back() const;

// PRE: !empty()
// POST: first element of *this is removed
void pop_front();
```

Implement `operator<<` to output the elements in the queue from front to back. In addition, as a queue is a dynamic type, you must also provide copy constructor, assignment operator, and destructor.

Furthermore, the Codeboard-template provides a `main`-function to test your implementation.

**Hint:** Queues work similarly to stacks, thus you can take the codes for `ifmp::stack` as a reference.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>operations</td>
<td>p -1 p 2 p -3 p 4 f r f s</td>
</tr>
<tr>
<td></td>
<td>-1 2</td>
</tr>
</tbody>
</table>

Submission: [https://codeboard.ethz.ch/ifmp16E13T2](https://codeboard.ethz.ch/ifmp16E13T2)

Assignment 3 – Vector (4 points)

In this exercise we ask you to implement a class `ifmp::vector` for type `int` that provides similar functionality as `std::vector<int>`, at least as far as dynamic memory management is concerned. A `std::vector` is a container that represents arrays that can change in size. Internally, the class `std::vector` uses dynamically allocated arrays to store the elements. Like for normal arrays, the elements are stored in contiguous locations in the memory. This allows for efficient access of the elements with pointer arithmetic.
The class `std::vector` cannot only allocate memory of arbitrary size at the moment it is created, but it can also dynamically change its size. For instance, you can add an element `e` to the end of the vector with the function `push_back(e)`, and the size increases by 1. To add an element at the end of a vector of size `s`, it is not enough to simply reserve some memory for this single element and store it there. The memory occupied by the vector might not be contiguous. Instead you will have to reserve a larger range in memory and copy over the entire vector to the new range. It would be a wise decision at this point to reserve a range which is larger by more than one element (good choice is: larger by a factor of 2, i.e. `2s` elements) and hide the additional elements from the user. This way, later `push_back` calls can simply write into these spare elements instead of having to copy over the entire vector to a larger range each time `push_back` is called.

The Codeboard-link provides you with a template listing the member functions of our `ifmp::vector`. Your task is to implement these member functions! If you want, you can write more functions! Furthermore, the Codeboard-template also provides a `main`-function to test your implementation.

Of course you are not allowed to use any data structure from the Standard Library that already provides dynamic memory allocation for this exercise, but you should call `new` and `delete` directly.

**Note:** The class `ifmp::vector` stores three pointers: `[begin_, end_of_memory_)` denotes the range of allocated memory, and `(end_of_memory_-begin_)` is called the capacity of the vector. `[begin_, end_)` denotes the range of used memory by the elements of the vector, and `(end_-begin_)` is called the size of the vector. The following illustration depicts the pointer layout. (The name `end_of_memory_` is shortened to `eom_`.)

---

**I/O-Examples**


<table>
<thead>
<tr>
<th>operations</th>
<th><code>p 1 p 2 p 3 f b s</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>1 3</code></td>
</tr>
</tbody>
</table>

**Submission:** [https://codeboard.ethz.ch/ifmp16E13T3](https://codeboard.ethz.ch/ifmp16E13T3)