Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Lecturer: Prof. B. Gärtner

Informatik für Mathematiker und Physiker HS16

Exercise Sheet 13

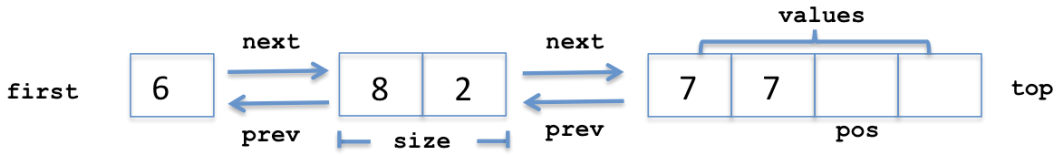
Submission deadline: 15:15 - Tuesday 20th December, 2016

Course URL: http://lec.inf.ethz.ch/ifmp/2016/

Assignment 1 – Doubling Stack (4 points)

[from: Exam Summer 2016, ex.8]

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`s).

```
1  struct Node { // for one array in the list
2    int* values;  int size;  Node* next;  Node* prev; // data members
3
4    Node (int s, Node* p)
5      : values (new int[s]), size (s), next (0), prev (p) // constructor
6    {}
7
8    ~Node() { delete[] values; }  // destructor
9
10   Node (const Node& node) { ... } // copy-constructor (not printed)
11
12   Node& operator= (const Node& node) { ... } // assignment operator (not printed)
13 };
14
15
16 class DoublingStack {
17   Node* first;  Node* top;   int pos; // data members
18 public:
19   DoublingStack ()
20     : first (new Node (1, 0)), top (first), pos (0) // default constructor
21   {}
22
23   DoublingStack (const DoublingStack& ds) {...} // copy-constructor (not printed)
24
25   DoublingStack& operator= (const DoublingStack& ds) {...} // (not printed)
26
27   // POST: puts value on the stack
```

```
28    void push (int value) {
29      if (pos == top->size) { // top array full
30        top->next = new Node (2 * top->size, top);
31        top = top->next; pos = 0;
32      }
33      top->values[pos++] = value;
34    }
35  ...
36  };
```

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`

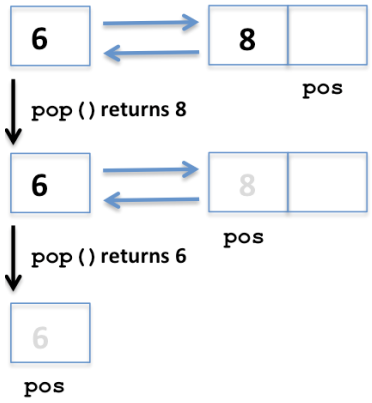
`N = 33`

ii) Implement the destructor `~DoublingStack` such that it frees all allocated memory!

```
1  ~DoublingStack() {
2    Node* node = first;
3    while (node != [1]) {
4      Node* tmp = [2];
5      node = [3];
6      delete [4];
7    }
8  }
```

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!

```
1  // PRE: *this is not empty
2  // POST: ...
3  int pop() {
4    if (pos == [5]) {
5      top = [6];
6      pos = [7];
7      delete [8];
8      [9];
9    }
10   return [10];
11 }
```



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)

[Skript-Aufgabe 160]

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`).

```
// 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.

I/O-Examples (Explanation: http://lec.inf.ethz.ch/ifmp/2016/codeboard.html)

```
operations
p -1 p 2 p -3 p 4 f r f s
-1 2
```

Submission: https://codeboard.ethz.ch/ifmp16E13T2

Assignment 3 – Vector (4 points)

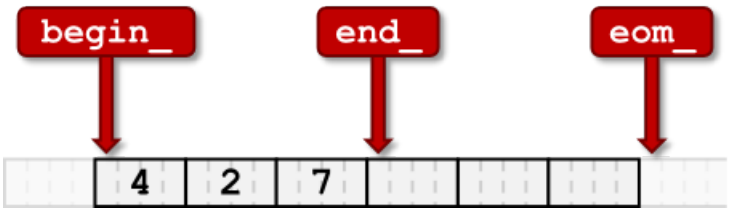
In this exercise we ask you to implement a class `ifmp::vector` for type `int` that provides *similar* functionality as `std::vector<int>`, at least as far as dynamic memory management is concerned. A `std::vector` is a container that represents arrays that can change in size. Internally, the class `std::vector` 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` can not 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 (Explanation: http://lec.inf.ethz.ch/ifmp/2016/codeboard.html)

```
operations
p 1 p 2 p 3 f b s
1 3
```

Submission: https://codeboard.ethz.ch/ifmp16E13T3