Exercise 3 Counter Queue (*exam 19/02/2018*)

A well known supermarket chain, BZR, tasked you with the implementation of a piece of software for the management of queues at counters in their stores. After an environmentally responsible choice, BZR will install card readers at each counter, hence customers will not pick a number from the usual paper roll, but they will virtually enter a line via scanning their fidelity card.
The existing store management software represents customer's identifiers with `string`s. Your class has to provide three main methods:

1. the first method takes an ID as argument and adds the corresponding customer in line
2. the second method returns the ID of the next client to be served
3. the third method checks whether there is any client who is waiting.

Beware that customers must not be allowed to reserve more than one position in the queue and the system is expected to notify such failures to get in line. Your contract requires methods to be optimized for the **average case**.
In addition, state the complexity of all the implemented methods, both in the average and worst case, with a particular focus on the design choices you made with respect to data structures and algorithms.
At last, discuss at high level how you would change the project if BZR wanted to give precedence to premium clients and how this would affect the computational complexity.

Exercise 3 - Solution

We need a `std::queue` to keep track of customers' arrival order. Furthermore, we need a fast way to check who is in line, then we also use a `std::unordered_set` for its quick lookups. Hash tables are better than binary trees in the average case, which is why we prefer the unordered alternative over `std::set`.

********** file counter_queue.h **********

```
#ifndef _COUNTER_QUEUE_
#define _COUNTER_QUEUE_

#include <queue>
#include <string>
#include <unordered_set>

namespace supermarket
{
 class counter_queue
 {
   std::queue<std::string> m_order;
   std::unordered_set<std::string> m_in_line;

 public:
   bool
   pick_number (const std::string & customer_id);

   std::string
   next_customer (void);

```

```
22    bool
23    empty (void) const;
24  };
25 }
26
27 #endif // _COUNTER_QUEUE_
```

The method pick_number (see lines 5 to 15 of file "counter_queue.cpp" for the implementation) relies on the methods std::unordered_set::insert and std::queue::push. Indeed, in line 11, it calls insert, which returns a std::pair containing and iterator (that points either to the newly inserted element or to an element with the same key, if it was already stored in the set) and a bool (which is true if the element is successfully inserted). Then, in line 13, the method checks if the new element is effectively added to the set and, in this case, it calls std::queue::push in order to append the new customer to the end of the queue.
The method next_customer (see lines 17 to 24) picks the first element of the queue, which is the next customer to serve, and it erases it both from the queue and from the set of customers in line.

********** file counter_queue.cpp **********

```
1  #include "counter_queue.h"
2
3  namespace supermarket
4  {
5    bool
6    counter_queue::pick_number (const std::string & customer_id)
7    {
8      typedef
9        std::pair<std::unordered_set<std::string>::iterator, bool>
10       insert_return_type;
11     const insert_return_type outcome = m_in_line.insert (customer_id);
12     const bool added_in_queue = outcome.second;
13     if (added_in_queue) m_order.push (customer_id);
14     return added_in_queue;
15   }
16
17   std::string
18   counter_queue::next_customer (void)
19   {
20     const std::string next = m_order.front ();
21     m_order.pop ();
22     m_in_line.erase (next);
23     return next;
24   }
25
26   bool
27   counter_queue::empty (void) const
28   {
29     return m_order.empty ();
30   }
31 }
```

By the choice of containers we have made, the methods pick_number, next_customer, and empty have $O(1)$ complexity on average, while, being N the number of customers in line,

`pick_number` and `next_customer` have complexity $O(N)$ in the worst case.
In order to prioritize some premium customers over others, the `std::queue` should be substituted with a `std::priority_queue`, with some other minor changes. The main difference in terms of complexity is in `pick_number` and `next_customer`, which become $O(\log N)$ both in the average and worst case, due to the underlying heap.

Exercise 4 Streaming System (*exam 21/07/2017*)

You have to develop a micro-batch streaming system. In particular, you have to compute the average, minimum, and maximum values of the data samples received within a time window. A time window includes a fixed integer number of time slots and the data coming from each time slot are stored in a micro-batch. The time window is moved as the time elapses and old data are dropped according to the overlapping window paradigm. As an example, Figure 6.1 shows a sliding window including four times slots, hence four micro-batches.
The definition of the `MicroBatch` class is the following:

```
1  class MicroBatch
2  {
3   std::vector<double> samples;
4
5  public:
6   MicroBatch (const std::vector<double> & v)
7     : samples (v) {}
8
9   MicroBatch() {}
10
11  double get_min() const;
12  double get_max() const;
13  double get_sum() const;
14
15  unsigned get_n_samples() const;
16
17  void set_samples (const std::vector<double> & v);
18 };
```

In particular a `MicroBatch` object includes a varying number of samples received within a single time slot.
Provide the implementation of the `ContinousStream` class, which includes the following methods:

1. The constructor, which receives as input the number of time slots.
2. The `get_min()`, `get_max()`, and `get_average()` methods, which return the minimum, maximum, and average of all the values currently stored in the time window.
3. The `add_batch` method that adds a new micro-batch in the time window, discarding the oldest.

Store the micro batches in a circular vector (see Figure 6.2) in a way that, when a new micro-batch is added in the time window, the oldest one is removed. In particular, store in the circular vector shared pointers to micro-batch objects. Moreover, since you can receive multiple requests to the `get` methods within a time slot, cache the minimum, maximum, and sum of the samples in a micro-batch in vectors and update their values within the `add_batch` method.