Programming Assignment 2:
Design and Implementation of a Service Queue ADT

DUE DATE: Saturday, March 2 by 11:59PM

Note that you will submit two items separetely:

1. **An analysis of a given implementation (submitted via gradescope).**
2. **Your class implementation (file: ServiceQueue.h)**

(details below).

You go to a popular restaurant that doesn't take reservations. When you get there, you just have to wait in line for a table. These days, most restaurants will give you some kind of buzzer and will signal you when your table is ready.

The restaurant essentially has to manage a queue of buzzers. Implementation of an ADT supporting this management is what you will be doing in this assignment.

Overview of ServiceQueue behavior

(greater detail can be found in the banner comments in ServiceQueue.h).

- "Buzzers": for our purposes, a "buzzer" is represented by a unique integer ID. In principle, there is an infinite number of buzzers starting from 0.
- Operation **give_buzzer()**: this is the enqueue operation (a new customer is getting in line). The following happens:
  - An unused buzzer ID is determined (see Buzzer Policy below) and
  - an entry for that buzzer is added to the end of the queue and
  - the selected buzzer ID is returned.
- Operation **seat()**: this is the "dequeue" operation: if the queue is not-empty, the first entry is removed from the front; the removed buzzer ID can now be "re-used" (again, see Buzzer Policy below). If the queue is empty, false is returned and no changes are made (on success, true is returned).
- Operation **take_bribe()**: A person holding a particular buzzer may offer the system a bribe to move to the front of the queue. This function takes a buzzer ID and does the following:
  - If the buzzer is indeed somewhere in the queue, it is plucked out and moved to the front of the queue and true is returned.
  - Otherwise (the buzzer isn't even in the queue), the function simply returns false.
- Operation **kick_out():** (for sending trouble causers home). This operation takes a buzzer ID and does the following:

    - If the buzzer is indeed somewhere in the queue, it is removed and true is returned. The buzzer now becomes reusable (again, see Buzzer Policy).
    - Otherwise (the buzzer isn't even in the queue), the function simply returns false.
- Operation **snapshot():** This function takes an integer vector as a reference parameter and populates it with the current queue contents (i.e., sequence of buzzer IDs) in sequence.
- Operation **length():** reports the current length of the queue.

Buzzer Policy:

The buzzer ID assigned by a call to give_buzzer is determined by the following policy:

1. If there are no reusable buzzers (i.e., from a previous seat or kick_out operation), the ***smallest*** unused buzzer ID must be used. (As a consequence, the first call to give_buzzer will always use ID zero).
2. If there ***are*** reusable buzzers, the buzzer that became reusable most recently ***must*** be used (i.e., from the most recent seat or kick_out operation).

Understanding Behavior:

You have been given two complete implementations of the ADT described above. However, these implementations, while behaviorally correct, **do NOT meet the runtime requirements that *you* must meet!** The "slow" implementations are given in two files (they both use the class name ServiceQueue):

- **ServiceQueueSlow.h:** this makes extensive use of C++ vectors including some arcane stuff with removal, iterators, etc.
- **ServiceQueueSlow2.h**: this also uses C++ vectors and the same overall structure as ServiceQueueSlow.h. However, some operations are done "by hand" on the underlying vectors instead of using the functions provided by the vector class. Reason: you may find some of the vector functions called in ServiceQueueSlow.h to be a bit strange (depending on your level of C++ experience). This version may have fewer distractions for you (otherwise, in terms of behavior and runtime, the two "slow" versions are effectively the same.

You have also been given a simple interactive driver program in **Driver.cpp**. It can be compiled to use any of the three ServiceQueue implementations (one of the two slow ones or yours).

To compile using ServiceQueue.h, you do nothing fancy (produces executable Driver):

```
g++ -std=c++11 Driver.cpp -o Driver
```

To compile using ServiceQueueSlow.h:

```
g++ -std=c++11 Driver.cpp -D_SLOW -o DriverSlow
```

To compile using ServiceQueueSlow2.h:

```
g++ -std=c++11 Driver.cpp -D_SLOW2 -o DriverSlow2
```

Or... simply type "make" and the provided makefile will compile all three.

Play with the driver program: To make sure you understand the correct behavior, compile and run DriverSlow and/or DriverSlow2 and follow the interactive menu.

A sample session has been included in an appendix.

Runtime Requirements:

Now the fun part! Your implementation of the ServiceQueue class must meet the following runtime requirements:

```
give_buzzer():  O(1) runtime (technically O(1) "amortized runtime")
seat():         O(1) runtime.
take_bribe():   O(1) runtime
kick_out():     O(1) runtime
snapshot():     O(n) runtime (where n is the queue length).
length():       O(1) runtime
```

Deliverables

1. (via gradescope) **Analysis of a Slow implementation:** You will study the "Slow" implementations and do a (tight) worst-case runtime analysis for each of the functions above. Note that both ServiceQueueSlow.h and ServiceQueueSlow2.h have the same runtime behavior, so study the one you feel most comfortable with. This will be submitted via gradescope as a separate item. Details:
    a. You must include an analysis of each of the six operations above clearly labeled.
    b. Each analysis must give a rationale for each of your conclusions (i.e., why your claimed worst-case runtime bound is correct).
    c. In the event that an operation has an amortized runtime which differs from the worst case, report that result as well.
2. Your implementation of ServiceQueue.h meeting the requirements above.

Brainstorming

The week 06 lab will be dedicated to group brainstorming on ideas for how the runtime requirements can be met! In the meantime, think, think, think! Remember, you have lots of freedom here as to how you might organize your data structures. After you have completed your brainstorming lab sessions, we will discuss in lecture a sketch of a strategy which meets the runtime requirements. So, even if you weren't able to converge on a design that meets all requirements, you still can submit a final implementation that does!

Restrictions:

You may use vectors from the standard template library, but that is about the limit -- everything else must be done by you!

Hints:

- You may find doubly-linked list nodes handy!
- It's cliche, but try to think outside the box. Think about your toolbox -- you have structs, vectors, pointers, etc.. Be creative about how to use them!

Appendix: Sample Session Compiling and Running the Driver Program With the given ServiceQueueSlow.h implementation.

```
Johns-MacBook-Air:src lillis$ make
g++ -Wall -std=c++11 Driver.cpp -o Driver
g++ -Wall -std=c++11 -D_SLOW Driver.cpp -o DriverSlow
g++ -Wall -std=c++11 -D_SLOW2 Driver.cpp -o DriverSlow2
Johns-MacBook-Air:src lillis$ ./DriverSlow

Welcome to the simple service-queue interactive program

      (SOURCE FILE USED FOR ServiceQueue CLASS: ServiceQueueSlow.h)

   An empty service queue has been created for you
   Commands:
    d          : display queue
    l          : report length of queue
    g          : give out a buzzer
    s          : serve the first buzzer in line
    k <buzzer> : kick specified buzzer out!
    b <buzzer> : take a bribe to move specified buzzer to front!
    q          : quit
-----------------------------------

cmd > g
  buzzer: 0
cmd > g
  buzzer: 1
cmd > g
  buzzer: 2
cmd > g
  buzzer: 3
cmd > d
  [ 0 1 2 3 ]
cmd > s
  seating buzzer: 0
cmd > d
  [ 1 2 3 ]
cmd > g
  buzzer: 0
cmd > g
  buzzer: 4
cmd > d
  [ 1 2 3 0 4 ]
cmd > b 3
```

```
  VIP coming through!
cmd > d
  [ 3 1 2 0 4 ]
cmd > g
  buzzer: 5
cmd > k 1
  1 is outta here!
cmd > d
  [ 3 2 0 4 5 ]
cmd > s
  seating buzzer: 3
cmd > d
  [ 2 0 4 5 ]
cmd > k 4
  4 is outta here!
cmd > d
  [ 2 0 5 ]
cmd > g
  buzzer: 4
cmd > d
  [ 2 0 5 4 ]
cmd > b 4
  VIP coming through!
cmd > d
  [ 4 2 0 5 ]
cmd > g
  buzzer: 3
cmd > d
  [ 4 2 0 5 3 ]
cmd > s
  seating buzzer: 4
cmd > g
  buzzer: 4
cmd > d
  [ 2 0 5 3 4 ]
cmd > q
  goodbye...
```