PONDER 04 : NOW SERVING

Due Saturday at 11:59 PM MST

The fourth programming assignment will be to implement the deque data structure and use it to implement a program to help lab assistants serve students in the Linux Lab more effectively.

Deque

Create a class encapsulating the notion of a deque. A deque is a double-ended queue. In other words, it is a combination of a queue and a stack. The deque we will create will work exactly like the [std::deque](http://www.cplusplus.com/reference/deque/deque/) class. The deque must be implemented as a circular deque the same as your circular queue. Of course, any data-type will need to be supported, so your class will be a template class. It will need to be defined in its own header file (deque.h). The class name must beDeque and will need to support the following operations:

* **Constructors**: Default constructor (create a deque with zero items in it), a non-default constructor (taking a capacity value as a parameter), and the copy constructor. If allocation is not possible, the following error will be thrown:  
  ERROR: Unable to allocate a new buffer for deque
* **Destructor**: When finished, the class should delete all the allocated memory.
* **operator=**: Assignment operator. This method takes a Deque as a parameter and copies all the elements to this. If the current buffer size is sufficient, not allocation is made. If the current buffer size is not sufficient, enough space is allocated to accomodate the new data. If there is insufficient memory to allocate a new buffer, then the following exception is thrown:  
  ERROR: Unable to allocate a new buffer for Deque. It also returns \*this by-reference as all assignment operators should.
* **empty()**: Test whether the deque is empty. This method takes no parameters and returns a Boolean value.
* **size()**: Return the deque size. This method takes no parameters and returns an integer value.
* **capacity()**: Return the current capacity of the deque. That is, the number of elements the deque is able to store without reallocating.
* **clear()**: Empties the deque of all items. There are no parameters and no return value.
* **push\_back()**: Adds an item to the back of the deque. This method takes a single parameter (the item to be added to the end of the deque) and has no return value. Note that if the deque is full, then the capacity will be doubled. In the case of an allocation error, the following c-string exception will be thrown:  
  ERROR: Unable to allocate a new buffer for deque
* **push\_front()**: Adds an item to the front of the deque exactly the same as push\_back().
* **pop\_back()**: Removes an item from the back of the deque, serving to reduce the size by one. Note that if the deque is already empty, the following c-string exception will be thrown:  
  ERROR: attempting to pop from an empty deque
* **pop\_front()**: Removes an item from the front of the deque exactly the same aspop\_front().
* **front()**: Returns the item currently at the front of the deque. This item is returned by-reference, so the last item can be changed through the front() method. If the deque is currently empty, the following exception will be thrown:  
  ERROR: attempting to access an item in an empty deque
* **back()**: Returns the item currently at the back of the deque exactly the same as front().

Note that the only way to access elements in a deque is through the front() and back() method. This means that is no iterator for Deque.

A driver program will be provided the same as previous assignments. A few hints that may come in handy when implementing this part of the assignment:

* Start simple! Start with a fixed array of 100 items and put the head and the tail in the middle (iHead = 50; iTail = 49). Make sure you get the push and pop methods working in this case.
* Get tail wrapping working. With the same 100 item array, move the head and the tail to the end (iHead = 99; iTail = 98). Make sure that push\_back() works in this case.
* Handle a negative index. Now move to the head and tail pointers to the start of the array (iHead = 0; iTail = -1). Make sure that push\_front() works in this case.
* Finally, work on the grow. Remember that when you push data onto a full deque, there are two steps: first to grow the buffer, then to perform a normal push. Make sure to test these two cases independently.
* One last hint: write a debug function to display the contents of the deque. This will enable you to see what is happening and give you important clues as to where your bugs are.

Now Serving

In the Linux Lab, there are a small number of lab assistants serving a large number of students. In order to make best use of the lab assistants' time and to keep the lab operating in an orderly manner, the Now Serving system was developed. This system works as follows: when a student needs help, the student enters a help request into the system. This help request consists of the class (CS 235), the student name (Sam), and the number of minutes that will be required (5). This help request is put onto the back of the deque. If there is no student currently being helped by a lab assistant, then the assistant takes the next request off of the front of the deque and assists him or her for the desired number of minutes. Occasionally there is an emergency. A student can submit an emergency help request by preceding his request with two exclamation marks (!!). When this happens, the emergency help request is placed at the head of the deque.

Every minute, the system decrements the time-remaining value on the current help request. If the time-remaining value reaches zero then the next help request (if there is one) is taken from the deque. Also, every minute a new help request can be entered into the system. To see how this works, consider the following execution:

Every prompt is one minute. The following input is accepted:

<class> <name> <#minutes> : a normal help request

!! <class> <name> <#minutes> : an emergency help request

none : no new request this minute

finished : end simulation

<0> CS124 Sam 4

Currently serving Sam for class CS124. Time left: 4

<1> CS165 Sue 2

Currently serving Sam for class CS124. Time left: 3

<2> none

Currently serving Sam for class CS124. Time left: 2

<3> none

Currently serving Sam for class CS124. Time left: 1

<4> none

Currently serving Sue for class CS165. Time left: 2

<5> none

Currently serving Sue for class CS165. Time left: 1

<6> none

<7> none

<8> finished

End of simulation

For a second example, Sam and Sue ask for 2 minutes of help. Then Steve needs some emergency help. Note how Steve jumps ahead of Sue in line:

Every prompt is one minute. The following input is accepted:

<class> <name> <#minutes> : a normal help request

!! <class> <name> <#minutes> : an emergency help request

none : no new request this minute

finished : end simulation

<0> CS124 Sam 2

Currently serving Sam for class CS124. Time left: 2

<1> CS165 Sue 2

Currently serving Sam for class CS124. Time left: 1

<2> !! CS124 Steve 2

Emergency for Steve for class CS124. Time left: 2

<3> none

Emergency for Steve for class CS124. Time left: 1

<4> none

Currently serving Sue for class CS165. Time left: 2

<5> none

Currently serving Sue for class CS165. Time left: 1

<6> none

<7> none

<8> finished

End of simulation

A few hints that may come in handy when implementing this part of the assignment are:

* After every prompt, the currently served person (if there is one) is displayed. There is a tab character immediately before each of these lines.
* You can assume there is no space in the student's name nor in the class name.
* Every prompt contains the number of minutes that have passed since the simulation began.
* You might want to create a class or two to help with this problem. It will "greatly simplify" the complexity of the resulting program.

As with the previous lessons, you must use your own Deque class to get full credit. If your class does not work, use the standard template library std::deque from #include <deque>. If you do this, you will loose points for the first half of the assignment, but not the second.

Common Mistakes

The most common mistakes students make with this assignment include the following:

* **Currently-helped student**. The currently-helped student should not be in the deque. Otherwise, adding an emergency item to the deque will cause the currently-helped student to be interrupted.
* **Not designing first**. This may sound silly, but the most common mistake made with this assignment is to start writing code without a clear design in mind. Though no pseudocode, structure charts, or UML class diagrams are required for this assignment, you should seriously consider using them for your own needs. It is easy to get lost when you don't have a map!

Test Bed

The testBed for this assignment is:

testBed cs235/week04 week04.tar

You can also run testBed on the executable:

testBed cs235/week04 a.out

Of course, you will need to pass testBed to get full credit on the assignment.

Submitting

You will submit this assignment using the Linux submit command. Please:

1. Create a TAR file built from the makefile, which will contain five files:
   * makefile: Directly from /home/cs235/week04/makefile except with your edits on the comment block.
   * deque.h: Your class definition for Deque.
   * nowServing.h: Containing the prototype for nowServing() and any other functions or classes you may need.
   * nowServing.cpp: Implementation for all the functions and classes necessary for the Now Serving program.
   * week04.cpp: Unmodified from /home/cs235/week04/week04.cpp.
2. Run the program by hand a few times through all four test cases as well as the Now Serving program.
3. Verify your solution with testBed.
4. Submit your file using the submit command. The submit command will prompt you for your instructor, the class (cs235), and the assignment (week04). You submit your file with:

submit week04.tar

Your program will be graded according to the following rubric:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Exceptional 100% | Good 90% | Acceptable 70% | Developing 50% | Missing 0% |
| Deque Interface  20% | The interfaces are perfectly specified with respect to const, pass-by-reference, etc. | week04.cppcompiles without modification | All of the methods in Deque match the problem definition | Deque has many of the same interfaces as the problem definition | The public methods in the Deque class do not resemble the problem definition |
| Deque Implementation  20% | Passes all fourDeque testBed tests | Passes three testBed tests | Passes two testBed tests | Passes one testBed test | Program fails to compile or does not pass any testBed tests |
| Now Serving  30% | The code demonstrates Object-Oriented design principles | Passes the Now Serving testBed test | The code essentially works but with minor defects | Elements of the solution are present | The Now Serving code was not attempted |
| Code Quality  20% | There is no obvious room for improvement | All the principles of encapsulation and modularization are honored | One function is written in a "backwards" way or could be improved | Two or more functions appears "thrown together." | The code appears to be written without any obvious forethought |
| Style  10% | Great variable names, no errors, great comments | No obvious style errors | A few minor style errors: non-standard spacing, poor variable names, missing comments, etc. | Overly generic variable names, misleading comments, or other gross style errors | No knowledge of the BYU-I code style guidelines were demonstrated |

Please make sure to fill out the program header in the makefile with the following information: the amount of coding time required to complete the assignment, and what was the most difficult part. Failure to do this will result in a loss of 10% on the assignment.