

Assignment 5: Patient Queue

Assignment by Chris Gregg and Anton Apostolatos, based on an assignment by Marty Stepp and Victoria Kirst. Originally based on a problem by Jerry Cain.

MAY 11, 2017

Due Date:

The patient queue assignment is due Thursday, May 18th at 12:00pm.















Outline:

Description Implementation Style FAQ Extras

This problem focuses on implementing a priority queue using a Vector and also implementing a priority queue using a singly-linked list.

In the problem, you will implement a collection called a *priority queue* in two different ways: using a Stanford Library Vector and a linked-list. As an extension, you can also implement a priority queue using a heap (note: most priority queues are implemented as a heap). This problem focuses on linked lists and pointers, along with classes and objects. This is an **pair programming assignment**. You may choose to work in a pair/group on this program.

Files and Links:

 Project Starter ZIP (open PatientQueue.pro)	 Turn in:	 output logs:
	 VectorPatientQueue.cpp  VectorPatientQueue.h  LinkedListPatientQueue.cpp  LinkedListPatientQueue.h  HeapPatientQueue.cpp (extension only)  HeapPatientQueue.h (extension only)	 output #1  output #2  output #3  output #4  output #5

Problem Description:

In this problem you will write a **PatientQueue** class that manages a waiting line of patients to be processed at a hospital.

In this class we have learned about queues that process elements in a first-in, first-out (FIFO) order. But FIFO is not the best order to assist patients in a hospital, because some patients have more urgent and critical injuries than others. As each new patient checks in at the hospital, the staff assesses their injuries and gives that patient an integer **priority** rating, with smaller integers representing greater urgency. (For example, a patient of priority 1 is more urgent and should receive care before a patient of priority 2.)

Once a doctor is ready to see a patient, the patient with the most urgent (smallest) priority is seen first. That is, regardless of the order in which you add/enqueue the elements, when you remove/dequeue them, the one with the lowest priority number comes out first, then the second-smallest, and so on, with the largest priority item coming out last. Priority ties are broken by first dequeuing the same-priority patients in the order they were enqueued. A queue that processes its elements in order of increasing priority like this is also called a *priority queue*.

Parts

There are **two** different classes to write for this assignment. The Vector and Linked-List implementations should have **identical** behavior. If you choose to implement the optional heap extension, it is easier to make it slightly different (specifically: patients with the same priority must be dequeued in the order they were enqueued, first-in-first-out, in the Vector and Linked-List implementations, and this is not as easy in a Heap implementation). See below for a description of each part of the assignment.

Overview of Patient Queues

A priority queue stores a set of keys (priorities) and their associated values. This assignment models a hospital waiting room as a priority queue. If two patients in the queue have the same priority, you will break ties by choosing the patient who arrived earliest with that priority. This means that if a patient arrives at priority K and there are already other patients in the queue with priority K , your new patient should be placed after them.

For example, if the following patients arrive at the hospital in this order:

- "Dolores" with priority 5
- "Bernard" with priority 4
- "Arnold" with priority 8
- "William" with priority 5

- "Teddy" with priority 5
- "Ford" with priority 2

Then if you were to dequeue the patients to process them, they would come out in this order: Ford, Bernard, Dolores, William, Teddy, Arnold.

Implementation Details for both Classes:

Internally a given implementation of a priority queue can store its elements in sorted or unsorted order; all that matters is that when the elements are dequeued, they come out in sorted order by priority. This difference between the external expected behavior of the priority queue and its true internal state can lead to interesting differences in implementation, which you will find while completing this assignment.

You will write a **VectorPatientQueue** class that uses an **un-sorted** Vector and you will write a second class, **LinkedListPatientQueue** that uses a **sorted** linked list as its internal data storage.

We supply you with a **PatientNode** structure (in **patientnode.h/cpp**) that is a small structure representing a single node of the linked list. Each **PatientNode** stores a string value and integer priority in it, and a pointer to a next node. In the linked-list implementation, you should use this structure to store the elements of your queue along with their priorities. In the Vector Patient Queue, you should probably write your own struct to hold the priorities and elements, and we strongly suggest that you also define an integer "timestamp" parameter that simply increments every time a value is added to the queue. The reason for the timestamp parameter is to enable you to differentiate between patients with the same priority who come in at different times -- in an unsorted list, you need to have the ability to determine who came in first, and a timestamp is a good way to do that. For the linked list implementation, you won't need a timestamp because the list will be sorted, and patients with the same priority will be sorted in the order in which they arrived.

The following is the PatientNode struct (used in the LinkedListPatientQueue class):

```
struct PatientNode {
    string name;
    int priority;
    PatientNode* next;

    // constructor - each parameter is optional
    PatientNode(string name, int priority, PatientNode* next);
    ...
};
```

Note: all of the XPatientQueue classes have an addition ": public PatientQueue" designator in the header files. This is called "inheritance" and is beyond the scope of cs106b -- we are using it to test your various Patient Queue classes in hospital.cpp. Feel free to ask about inheritance during office hours or on Piazza, but you do not need to concern yourself with it for this assignment.

PatientQueue Operations:

Your class must support all of the following operations. For the linked-list implementation, each member must run within the Big-Oh runtime noted; the *N* in this context means the length of the linked list.

Member Name	Description
PatientQueue()	In this parameterless constructor you should initialize a new empty queue (with front for your internal linked list, and an empty Vector for your Vector class).
~PatientQueue()	In this destructor you must free up any memory used -- for example, your linked nodes (you may not need to do anything in your destructor for your Vector class).
pq.newPatient(name, priority);	In this function you should add the given person into your patient queue with the given priority. Duplicate names and priorities are allowed. Any string is a legal name and any integer is a legal priority; there are no invalid values that can be passed.
pq.processPatient()	In this function you should remove the patient with the most urgent priority from the queue, and you should also return their name as a string. You should throw a string exception if the queue does not contain any patients.
pq.frontName()	In this function you should return the name of the most urgent patient (the person at the front of your patient queue), without removing it or altering the state of the queue. You should throw a string exception if the queue does not contain any patients.
pq.frontPriority()	In this function you should return the integer priority of the most urgent patient (the person at the front of your patient queue), without removing it or altering the state of the queue. You should throw a string exception if the queue does not contain any patients.
pq.upgradePatient(name, newPriority);	In this function you will modify the priority of a given existing patient in the queue. The intent is to change the patient's priority to be more urgent (a smaller integer) than its current value, perhaps because the patient's condition has gotten worse. If the patient is present in the queue and already has a more urgent priority than the new priority, or if the given patient is not already in the queue, your function should throw a string exception . If the given patient name occurs multiple times in the queue, you should alter the priority of the highest priority person with that name that is placed into the queue.
pq.isEmpty()	In this function you should return true if your patient queue does not contain any elements and false if it does contain at least one patient.

<code>pg.clear() ;</code>	In this function you should remove all elements from the patient queue, freeing memory for all nodes that are removed.
<code>toString()</code>	You should write a <code>toString()</code> function for printing your patient queue to the console. The function should return a string as defined as follows: The elements should be printed out in front-to-back order and must be in the form of <i>priority</i> with <code>{ }</code> braces and separated by a comma and space, such as: <code>{2:Ford, 4:Bernard, 5:Dolores, 5:William, 5:Teddy, 8:Arnold}</code> The <code>PatientNode</code> structure has a <code><<</code> operator that may be useful to you. Your formatting and spacing should match exactly. Do not place a <code>\n</code> or <code>endl</code> at the end of your output. <i>Note: you only have to format the string in priority order for the linked list code and it can be in any order for your Vector code.</i>

members you must write in `PatientQueue` class

The headers of every member must match those specified above. Do not change the parameters or function names.

Constructor/destructor: Your class must define a parameterless constructor. Since your linked-list implementation allocates dynamic memory (using `new`), you must ensure that there are no memory leaks by freeing any such allocated memory at the appropriate time. This will mean that you will need a destructor to free the memory for all of your nodes.

Helper functions: The members listed on the previous page represent your class's required behavior. But you may add other member functions to help you implement all of the appropriate behavior. Any other member functions you provide must be **private**. Remember that each member function of your class should have a clear, coherent purpose. You should provide private helper members for common repeated operations.

You will also need to do the following:

Add descriptive comments to `LinkedListPatientQueue.h/cpp` and `VectorPatientQueue.h/cpp`. All files should have top-of-file headers; one file should have header comments atop each member function, either the `.h` or `.cpp`; and the `.cpp` should have internal comments describing the details of each function's implementation.

Declare your necessary private member variable in `PatientQueue.h`, along with any private member functions you want to help you implement the required public behavior. Your inner data storage *must* be a singly linked list of patient nodes; do not use any other collections or data structures in any part of your code.

Implement the bodies of all member functions and constructors in `PatientQueue.cpp`.

As stated previously, the elements of the patient queue are stored in **sorted order** internally. As new elements are enqueued, you should add them at the appropriate place in the vector or linked list so as to maintain the sorted order. The primary benefit of this implementation is that when dequeuing, you do not need to search the vector or linked list to find the element with most urgent priority to remove/return it. Enqueuing new patients is slower, because you must search for the proper place to enqueue them, but dequeuing a patient to process them is very fast, because they are at the front of the vector or Vector list.

Vector Implementation

Your class will use an unsorted `Stanford Vector` to hold the queue of patients. Because the `Vector` is unsorted, you will need to traverse the list to find the element with the smallest element, which is, notably, inefficient.

You should create a struct inside the `VectorPatientQueue.h` file that holds a patient name and a priority, and it should also hold an integer timestamp for breaking ties. This timestamp can be based on an incrementing counter elsewhere in your class, and every time a patient is enqueued you can update the timestamp and add it to the patient's struct.

Do not overthink the `Vector` patient queue -- this should be a relatively straightforward part of the project so you can get used to the idea of a priority queue.

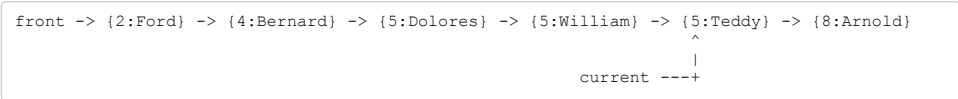
Linked List Implementation

Your class will use a singly **linked list** as its internal data storage. The elements of the linked list are stored in **sorted order** internally. As new elements are enqueued, you should add them at the appropriate place in the linked list so as to maintain the sorted order. The primary benefit of this implementation is that when removing a patient to process them, you do not need to search the linked list to find the smallest element and remove/return it; it is always at the front of the list. Enqueuing is slower, because you must search for the proper place to enqueue new elements, but dequeue/peeking and general overall performance are fairly good.

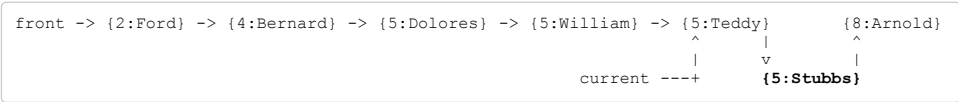
The following is a diagram of the internal linked list state of a `PatientQueue` after enqueueing the elements listed previously:

```
front -> {2:Ford} -> {4:Bernard} -> {5:Dolores} -> {5:William} -> {5:Teddy} -> {8:Arnold}
```

A tricky part of this implementation is **inserting a new node** in the proper place when a new patient arrives. You must look for the proper insertion point by finding the last element whose priority is at least as large as the new value to insert. Remember that, as shown in class, you must often stop one node early so that you can adjust the next pointer of the preceding node. For example, if you were going to insert the value "Stubbs" with priority 5 into the list shown above, your code should iterate until you have a pointer to the node for "Teddy", as shown below:



Once the **current** pointer shown above points to the right location, you can insert the new node as shown below:



Here are a few other constraints we expect you to follow in your implementation:

You should not make unnecessary passes over the linked list. For example, when enqueueing an element, a poor implementation would be to traverse the entire list once to count its size and to find the proper index at which to insert, and then make a second traversal to get back to that spot and add the new element. Do not make such multiple passes. Also, keep in mind that your queue class is not allowed to store an integer size member variable; you must use the presence of a null **next** pointer to figure out where the end of the list is and how long it is.

Duplicate patient names and priorities are allowed in your queue. For example, the **upgradePatient** operation should affect only a single occurrence of a patient's name (the one with lowest priority). If there are other occurrences of that same value in the queue, a single call to **upgradePatient** shouldn't affect them all.

You are not allowed to use a **sort** function or library to arrange the elements of your list, nor are you allowed to create any temporary or auxiliary data structures anywhere in your code. You must implement all behavior using only the one linked list of nodes as specified.

You will need pointers for several of your implementations, but you should not use pointers-to-pointers (for example, **PatientNode****). If you like, you are allowed to use a reference to a pointer (e.g. **PQNode*&**).

You should not create any more **PatientNode** objects than necessary. For example, if a **PatientQueue** contains 6 elements, there should be exactly 6 **PatientNode** objects in its internal linked list; no more, no less. You shouldn't, say, have a seventh empty "dummy" node at the front or back of the list that serves only as a marker, nor should you create a **new PatientNode** that is just thrown away or discarded without being used as part of the linked list. You can declare as many local variable pointers to **PatientNodes** as you like.

Development Strategy and Hints:

Draw pictures: When manipulating linked lists, it is often helpful to draw pictures of the linked list before, during, and after each of the operations you perform on it. Manipulating linked lists can be tricky, but if you have a picture in front of you as you're coding it can make your job substantially easier.

List states: When processing a linked list, you should consider the following states and transitions between them in your code as you add and remove elements. You should also, for each state below, think about the effect of adding a node in the front, middle, and back of the list.

zero nodes	front /
one node	front --> ? /
N nodes	front --> ? --> ? --> ... ? --> ? /

Heap Implementation (optional extension)

The third priority queue implementation you may optionally write uses a special array structure called a binary heap as its internal data storage. The only private member variables this class is allowed to have inside it are a pointer to your internal array of elements, and integers for the array's capacity and the priority queue's size.

As discussed in lecture, a binary heap is an unfilled array that maintains a "heap ordering" property where each index *i* is thought of as having two "child" indexes, *i* * 2 and *i* * 2 + 1, and where the elements must be arranged such that "parent" indexes always store more urgent priorities than their "child" indexes. To simplify the index math, we will leave index 0 blank and start the data at an overall parent "root" or "start" index of 1. One very desirable property of a binary heap is that the most urgent-priority element (the one that should be returned from a call to peek or dequeue) is always at the start of the data in index 1. For example, the six elements listed in the previous pages could be put into a binary heap as follows. Notice that the most urgent element, "t":2, is stored at the root index of 1.

index	0	1	2	3	4	5	6	7
value		"t":2	"m":5	"b":4	"x":5	"q":5	"a":8	
size =	6							
capacity =	10							

As discussed in lecture, adding (enqueueing) a new element into a heap involves placing it into the first empty index (7, in this case) and then "bubbling up" or "percolating up" by swapping it with its parent index (*i*/2) so long as it has a more urgent (lower) priority than its parent. We use integer division, so the parent of index 7 = 7/2 = 3. For example, if we added "y" with priority 3, it would first be placed into index 7, then swapped with "b":4 from index 3 because its priority of 3 is less than b's priority of 4. It would not swap any further because its new parent, "t":2 in index 1, has a lower priority than y. So the final heap array contents after adding "y":3 would be:

index	0	1	2	3	4	5	6	7
value		"t":2	"m":5	"y":3	"x":5	"q":5	"a":8	"b":4
size	= 7							
capacity	= 10							

Removing (dequeuing) the most urgent element from a heap involves moving the element from the last occupied index (7, in this case) all the way up to the "root" or "start" index of 1, replacing the root that was there before; and then "bubbling down" or "percolating down" by swapping it with its more urgent-priority child index ($i*2$ or $i*2+1$) so long as it has a less urgent (higher) priority than its child. For example, if we removed "t":2, we would first swap up the element "b":4 from index 7 to index 1, then bubble it down by swapping it with its more urgent child, "y":3 because the child's priority of 3 is less than b's priority of 4. It would not swap any further because its new only child, "a":8 in index 6, has a higher priority than b. So the final heap array contents after removing "t":2 would be:

index	0	1	2	3	4	5	6	7
value		"y":3	"m":5	"b":4	"x":5	"q":5	"a":8	
size	= 6							
capacity	= 10							

A key benefit of using a binary heap to represent a priority queue is efficiency. The common operations of enqueue and dequeue take only $O(\log N)$ time to perform, since the "bubbling" jumps by powers of 2 every time. The peek operation takes only $O(1)$ time since the most urgent-priority element's location is always at index 1.

If nodes ever have a tie in priority, break ties by comparing the strings themselves, treating strings that come earlier in the alphabet as being more urgent (e.g. "a" before "b"). Compare strings using the standard relational operators like $<$, $<=$, $>$, $>=$, $=$, and $!=$. Do not make assumptions about the length of the strings.

Changing the priority of an existing value involves looping over the heap to find that value, then once you find it, setting its new priority and "bubbling up" that value from its present location, somewhat like an enqueue operation.

For both array and heap PQs, when the array becomes full and has no more available indexes to store data, you must resize it to a larger array. Your larger array should be a multiple of the old array size, such as double the size. You must not leak memory; free all dynamically allocated arrays created by your class.

Remember to include the big-O of enqueue and dequeue in the comments in `HeapPriorityQueue.h`.

Style Details:

As in other assignments, you should follow our **Style Guide** for information about expected coding style. You are also expected to follow all of the general style constraints emphasized in the Homework 1-4 specs, such as the ones about good problem decomposition, parameters, redundancy, using proper C++ idioms, and commenting. The following are additional points of emphasis and style constraints specific to this problem. (Some of these may seem overly strict or arbitrary, but we need to constrain the assignment to force you to properly practice pointers and linked lists as intended.)

Commenting: Add descriptive comments to your .h and .cpp files. Both files should have top-of-file headers. One file should have header comments atop each member function (either the .h or .cpp; your choice). The .cpp file should have internal comments describing the details of each function's implementation.

Restrictions on pointers: The whole point of this assignment is to practice pointers and linked lists. Therefore, do not declare or use any other **collections** in any part of your code; all data should be stored in your linked list of nodes. There are some C++ "smart pointer" libraries that manage pointers and memory automatically, allocating and freeing memory as needed; you should not use any such libraries on this assignment.

Restrictions on private member variables: As stated previously, the only member variable (a.k.a. instance variable; private variable; field) you are allowed to have is a **PatientNode*** pointer to the front of your list. You may not have any other member variables. Do not declare an **int** for the list size. Do not declare members that are pointers to any other nodes in the list. Do not declare any collections or data structures as members.

Restrictions on modifying member function headers: Please do not make modifications to the **PatientQueue** class's public constructor or public member functions' names, parameter types, or return types. Our client code should be able to call public member functions on your queue successfully without any modification.

Restrictions on creation and usage of nodes: The only place in your code where you should be using the **new** keyword is in the **newPatient** function. No other members should use **new** or create new nodes under any circumstances. You also should not be modifying or swapping nodes' **name** values after they are added to the queue. In other words, you should implement all of the linked list / patient queue operations like **upgradePatient** by manipulating node pointers, not by creating entirely new nodes and not by modifying "data" of existing nodes.

You also should not create any more **PatientNode** structures than necessary. For example, if the client has called **newPatient** 6 times, your code should have created exactly 6 total **PatientNode** objects; no more, no less. You shouldn't, say, have a seventh empty "dummy" node at the front or back of the list that serves only as a marker, and you shouldn't accidentally create a temporary node object that is lost or thrown away. You can declare as many local variable *pointers* to nodes as you like.

Restrictions on traversal of the list: Any function that modifies your linked list should do so by traversing across your linked list a single time, not multiple times. For example, in functions like **newPatient**, do not make one traversal to find a node or place of interest and then a second traversal to manipulate/modify the list at that place. Do the entire job in a single pass.

Do not traverse the list farther than you need to. That is to say, once you have found the node of interest, do not unnecessarily process the rest of the list; break/return out of code as needed once the work is done.

Avoiding memory leaks: This item is listed under Style even though it is technically a functional requirement, because memory leakage is not usually visible while a program is running. To ensure that your class does not leak memory, you must delete all of the node objects in your linked list whenever data is removed or cleared from the list. You must also properly implement a destructor that deletes the memory used by all of the linked list nodes inside your **PatientQueue** object.

Frequently Asked Questions (FAQ):

For each assignment problem, we receive various frequent student questions. The answers to some of those questions can be found by clicking the link below.

Patient Queue FAQ (click to show)

Possible Extra Features:

For this problem, most good extra feature ideas involve adding operations to your queue beyond those specified. Here are some ideas for extra features that you could add to your program:

- **Known list of diseases:** Instead of, or in addition to, asking for each new patient's priority, ask what illness or disease they have, and use that to initialize the priority of newly checked-in patients. Then keep a table of known diseases and their respective priorities. For example, if the patient says that they have the flu, maybe the priority is 5, but if they say they have a broken arm, the priority is 2.
- **Merge two queues:** Write a member function that accepts another patient queue of the same type and adds all of its elements into the current patient queue. Do this merging "in place" as much as possible; for example, if you are merging two linked lists, directly connect the node pointers of one to the other as appropriate.
- **Deep Copy:** Make your queue properly support the `=` assignment statement, copy constructor, and deep copying. Google about the C++ "Rule of Three" and follow that guideline in your implementation.
- **Iterator:** Write a class that implements the STL **iterator** type and **begin** and **end** member functions in your queue, which would enable "for-each" over your PQ. This requires knowledge of the C++ STL library.
- **Other:** If you have your own creative idea for an extra feature, ask your SL and/or the instructor about it.

Indicating that you have done extra features: If you complete any extra features, then in the comment heading on the top of your program, please list all extra features that you worked on and where in the code they can be found (what functions, lines, etc. so that the grader can look at their code easily).

Submitting a program with extra features: Since we use automated testing for part of our grading process, it is important that you submit a program that conforms to the preceding spec, even if you want to do extra features. If your feature(s) cause your program to change the output that it produces in such a way that it no longer matches the expected sample output test cases provided, you should submit two versions of your program file: a first one with the standard file name without any extra features added (or with all necessary features disabled or commented out), and a second one whose file name has the suffix **-extra.cpp** with the extra features enabled. Please distinguish them in by explaining which is which in the comment header. Our turnin system saves every submission you make, so if you make multiple submissions we will be able to view all of them; your previously submitted files will not be lost or overwritten.

The linked-list class is only allowed to have a single private member variable inside it: a pointer to the front of your list (for the linked list implementation). Vector that holds the list (for the Vector implementation),

© Stanford 2017 | Created by Chris Gregg. CS106B has been developed over decades by many talented teachers.