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.

Due: Wednesday, August 2, at 12:00 PM (noon). You may work in pairs for this assignment.

Assignment Overview

In class we have learned about queues that process elements in a first-in, first-out (FIFO) order. But FIFO is not the best order for all problems—for example, for a problem like assisting patients in a hospital, since some patients have more critical injuries than others. In this problem, you will write several implementations of a collection class modeled on this type of queue—also known as a **priority queue**— for a hospital waiting room.

For the **`PatientQueue`** classes that you will write, you can imagine that as each new patient checks into the hospital, the staff assesses the patient's injuries and gives them an integer **priority** rating. Smaller integers represent 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 priority (i.e., with the lowest number rating) is seen first.

Your assignment has two parts: first, you will implement a **`PatientQueue`** class using a Stanford Libraries Vector. Then, you'll implement a **`PatientQueue`** class using a linked list that you design. Though the implementation details will differ, these two classes should have **identical** behavior from the user's perspective. This assignment is designed to give you more practice with creating your own classes and handling pointers. The second implementation is trickier than the first, because it requires a fair amount of pointer gymnastics. Start early, proceed slowly, and draw lots of pictures. If you do so, we think you'll find that this assignment is not as complex as it originally seems!

Files and Links

Starter files:

Project Starter ZIP

(open **PatientQueue.pro**)

- **VectorPatientQueue.cpp**
- **VectorPatientQueue.h**
-
- **LinkedListPatientqueue.cpp**
- **LinkedListPatientqueue.h**
- **HeapPatientQueue.cpp** (extension only)
- **HeapPatientQueue.h** (extension only)

Output logs:

- output #1
- output #2
- output #3
- output #4
- output #5

Turn in via paperless: https://cs198.stanford.edu/paperless

Overview of Patient Priority Queues

A priority queue can be understood as storing a set of keys (priorities) and their associated values (in our case, patient names).

The basic design of the priority queue is that, regardless of the order in which you add/enqueue the elements, when you remove/dequeue them, the one with the lowest number comes out first, then the second-lowest, and so on, with the highest-numbered (i.e., least urgent) item coming out last.

In your implementations of the **`PatientQueue`**, if two or more patients in the queue have the same priority, you should break ties by choosing the patient who arrived earliest. 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, suppose 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

If you were to dequeue the patients to process them, they would come out in this order: Ford, Bernard, Dolores, William, Teddy, Arnold. (Note that tie breaking works differently in the optional binary heap implementation; see details in that section.)

A priority queue can internally 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. The two parts of this assignment will require you to experiment with both kinds of storage: your Vector must be unsorted, and your linked list must be sorted. The difference between the external expected behavior of the priority queue and its true internal state can lead to interesting differences in implementation, as you will learn while completing this assignment.

PatientQueue Operations

For each of your implementations of the `PatientQueue`, you must write all of the following member functions. For the linked list version, each member must run within the Big-O runtime noted on the far right of the table; the *N* in this context means the length of the linked list. The headers of every member function must match those specified here. **Do not change the parameters or function names;** doing so will result in a deduction. The make-up of your linked list nodes and Vector elements will be described below, under "Implementation Details" for each part.

Member Name	Description	Vector Big-O	Linked List Big-O
`PatientQueue()`	In this parameterless constructor you should initialize a new empty queue (with a null front for your internal linked list, and an empty Vector for your Vector class).	O(1)	O(1)
`~PatientQueue()`	In this destructor you must free up any memory used—for example, you will need to free your linked list nodes (note that you may not need to do anything in your destructor for your Vector class).	O(1)	O(*N*)
***pq*.newPatient(*name*, *priority*);**	In this function you should add/enqueue the given person into your patient queue with the given priority. **Duplicate names and priorities are allowed**. Any string is a legal value, and any integer is a legal priority; there are no invalid values that can be passed.	O(1)	O(*N*)
***pq*.processPatient()**	In this function you should remove/dequeue the patient with the most urgent priority from your 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. The string should be a helpful, grammatically correct and concise message that informs the user what went wrong.	O(*N*)	O(1)
***pq*.frontName()**	In this function you should return the name of the most urgent patient (the person in the front of your patient queue), **without removing it or altering the state of the queue (i.e., peek name)**. You should throw a string **exception** if the queue does not contain any patients.	O(*N*)	O(1)
***pq*.frontPriority()**	In this function you should return the integer priority of the most urgent patient (the person in the front of your patient queue), **without removing it or altering the state of the queue (i.e., peek priority)**. You should throw a string **exception** if the queue does not contain any patients.	O(*N*)	O(1)

***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** (i.e., a smaller integer) than its current value, perhaps because the patient's condition has gotten worse. If the given patient is present in the queue and already has a more urgent priority than the given 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 was placed into the queue.	O(*N*)	O(*N*)
***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.	O(1)	O(1)
***pq*.clear();**	In this function you should remove all elements from the patient queue, freeing memory for all nodes that are removed.	O(1)	O(*N*)
toString()	You should write a **`toString()`** function for printing your patient queue to the console. The function should return a string in the following format: The elements should be printed out in front-to-back order and must be in the form of *priority*:*value* with **`{}`** braces and separated by a comma and space, such as: **`{2:Ford, 4:Bernard, 5:Dolores, 5:William, 5:Teddy,8:Arnold}`** The **`PatientNode`** structure has a **`<<`** operator that may be useful to you. Your formatting and spacing should match exactly. Do not place a **`\n`** or **`endl`** at the end of your output. Note: you only have to format the string in priority order for the linked list version of your p-queue; it can be in any order for your Vector implementation.	O(*N*)	O(*N*)

Members you must write for the ***`PatientQueue`*** *class*

Constructor/destructor: As the table above indicates, your class must define a constructor with no parameters. Since, as we shall see, your linked list implementation must allocate dynamic memory (using **`new`**), you must ensure that there are no memory leaks by freeing any such allocated memory at the appropriate time. This means that you will need a destructor to free the memory for all of your nodes.

Helper functions: The members listed on the previous pages 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.

Private members: 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.

Vector Implementation Details

For part one of this assignment, you will write a **`VectorPatientQueue`** class that will use an unsorted Stanford Vector to hold the queue of patients. Because the Vector is unsorted, when de-queueing/processing a patient, you will need to traverse the Vector to find the element with the smallest element. This process is inherently inefficient, but don't worry—that's why you'll try out the implementation with sorted linked list storage next.

Inside the **`VectorPatientQueue.h`** file, you will fine the interface for the **`VectorPatientQueue`** class. Your task is to implement the methods exported in this header file. To do so, you will need to define the private fields inside the class and to implement the class's methods in the **`VectorPQueue.cpp`** source file.

You should create a **`struct`** inside the **`VectorPatientQueue.h`** file that holds a patient name and a priority. We strongly suggest that you also define an integer "timestamp" parameter. This will let you distinguish between patients with the same priority who come in at different times—in an unsorted list, you must determine who came in first, and a timestamp is a good way to do that. This timestamp can be based on an incrementing counter elsewhere in your class; every time a patient is enqueued, you update the timestamp and add it to the patient's struct.

Try not to overthink the Vector patient queue— its main goals is to help you get used to the idea of a priority queue, and its implementation should be relatively straightforward.

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.

Linked List Implementation Details

For the second part of this assignment, you will write a `LinkedListPatientQueue` class that will use a sorted **singly linked list** as its internal data storage. 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. As new elements are enqueued, you should add them at the appropriate place in the linked list so as to maintain the sorted order. You should implement the bodies of all member functions and constructors in PatientQueue.cpp.

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 dequeueing/peeking is very fast, and overall performance is fairly good.

For this implementation, we supply you with a `PatientNode` structure (in **patientnode.h**/**cpp**), a small struct representing a single node of the linked list. You should use this structure to store the elements of your queue along with their priorities: each `PatientNode` contains a string value, an integer priority, and a pointer to the next node. Note that while in the Vector implementation you needed a timestamp to keep track of when patients arrive, 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 (to be 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);
     ...

};
```

To understand the internal state of the linked list behind a `PatientQueue`, let's return to the example used above. Suppose again that 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

The following is a diagram of the internal linked list state of a **`PatientQueue`** after enqueuing these elements:

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

The 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:

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

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

```
front ->{2:Ford} ->{4:Bernard}->{5:Dolores} ->{5:William} -> {5:Teddy}       {8:Arnold}
                                                               ^      |        ^
                                                               |      v        |
                                                     current ---+     {5:Stubbs}
```

Additional Constraints for Linked List Implementation

Don't make unnecessary passes over the linked list. For example, when enqueuing an element, a poor implementation would 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/how long it is.

Duplicate patient names and priorities are allowed. For example, the **`upgradePatient`** operation should only affect a single occurrence of a patient's name (specifically, 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.

Do not ***use a `sort` function or library*** to arrange the elements of your list.

Do not ***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.

No pointers to pointers. 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*&`**).

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).

Lastly, ***don't create unnecessary PatientNode objects***. 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 just 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. (This is an unnecessary use of memory.) You can, however, declare as many local variable pointers to **PatientNode**s 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 --> | ? |   | --> | ? |   |--> ... | ? | / |`
	`          +---+---+     +---+---+        +---+---+`

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 specs for Homeworks 1-4, such as those regarding 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 ensure you're using good practices with pointers and linked lists.)

Commenting: Add descriptive comments to your .h and .cpp files. Both files should have top-of-file header comments that summarize what your program does, and how. 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 noted above, 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 the "data" of existing nodes.

Restrictions on creating new PatientNode structs: As noted above, you should not create any more **`PatientNode`** structures than necessary. You can, however, declare as many local variable *pointers* to nodes as you like.

Restrictions on traversal of the list: Also as noted above, any function that modifies your linked list should do so by traversing 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.

Also, 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 the traversal 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, we receive various frequent student questions. The answers to some of those questions are below:

Q: How do I compare strings to see which comes earlier in ABC order?
A: C++ string objects support the standard comparison operators like <, <=, >, >=, ==, and != .

Q: How can I implement `operator <<` for printing a priority queue? It seems like the operator would need access to the private data inside of the priority queue object.
A: The **`<<`** operator in our assignment is declared with a special keyword called `friend` that makes it so that this operator is able to directly access the private data inside the priority queue if needed.
Q: What am I supposed to do in a destructor?
A: Free up any dynamic memory that you previously allocated with the `new` keyword.

Q: What is the difference between a destructor and the `clear` method? Don't they do the same thing, deleting all elements from the queue?

A: A `clear` method is called explicitly when a client wants to wipe the elements and then start over and use the same list to store something else. A destructor is called implicitly by C++ when an object is being thrown away forever; it won't ever be used to store anything else after that. The implementations might be similar, but their external purpose is different.

Q: What is the difference between `PQNode` and `PQNode*` ? Which one should I use?
A: You literally never want to create a variable of type `PQNode` ; you want only `PQNode*` . The former is an object, the latter is a pointer to an object. You always want pointers to `PQNode` objects in this assignment because objects created with `new` live longer; they are not cleaned up when the current function exits.

Q: How do I declare a `PQNode`?
A: Since the linked list PQ needs to keep its memory around dynamically, you should use the `new` keyword. Like this:

```
PQNode* node = new PQNode();
```

Or, you can pass any of the `value`, `priority`, and `next` values on construction:

```
PQNode* node = new PQNode(value, priority, next);
```

You *must* declare all your nodes as pointers; do not declare them as regular objects like the following code, because it will break when the list node goes out of scope:

```
// do not do this!
PQNode node;           // bad bad bad
node.data = 42;        // bad bad bad
node.next = nullptr;   // bad bad bad
...
```

Q: For the linked list PQ, how do I make sure that the strings stay in sorted order?
A: You have to do this yourself. You can compare strings to each other using the standard comparison operators: `>= =< > < == !=`

Q: Whenever I try to test my linked list PQ, the program "unexpectedly finishes." Why?
A: This is a very common bug when using pointers. It means you tried to dereference (->) a null pointer or garbage pointer. Run the program in Debug Mode (F5) and find out the exact line number of the error. Then trace through the code, draw pictures, and try to figure out how the pointer on that line could be NULL or garbage. Often it involves something like checking a value like `current->next->data` before checking whether `current` or `current->next` are `NULL`.

Q: Is there a difference between "deleting" and "freeing" memory?
A: We use the terms somewhat interchangeably. But what we mean is that you must call 'delete' on your dynamically allocated memory. There is a function named 'free' in C++, but we don't want you to use that.

And that’s it—congratulations!

You’re done. If you like, consider adding extra features (see next page).

Possible Extra Features

For this problem, most good extra feature ideas involve adding operations to your queue beyond those specified. There's a longer extension below that involves implementing the PatientQueue using a third data structure, the binary heap; here are some ideas for more minor additions and extra features that you could add to your existing 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 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, 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 identify the code for them easily). This also applies to the heap implementation described below.

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. (*This also applies to the heap implementation described below*.) Our submission system saves every submission you make, so if you make more than one we will be able to view all of them; your previously submitted files will not be lost or overwritten.

Optional Extension: Heap Implementation

Write a PatientQueue class that uses 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 "children's" indexes do. 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 on 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       8       9
      +-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+
value |       | "t":2 | "m":5 | "b":4 | "x":5 | "q":5 | "a":8 |       |       |       |
      +-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+
size = 6  capacity = 10
```

As discussed in lecture, adding (enqueuing) a new element into a heap involves placing it into the first empty index (7, in this case) and then "bubbling" 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, we would first place it into index 7, then bubble it up by swapping it with "b":4 from index 3, because its priority (3) is less than b's priority (4). This is where the bubbling or swapping would stop, however, 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       8       9
      +-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+
value |       | "t":2 | "m":5 | "y":3 | "x":5 | "q":5 | "a":8 | "b":4 |       |       |
      +-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+
size = 7  capacity = 10
```

As we discussed in lecture, removing (dequeuing) the most urgent element from a heap is not a simple matter of removing the "root" or start element at index 1. If we were to simply remove the element at the root index (say, by re-indexing our array), our tree would no longer have the proper structure (look at a picture of a binary heap tree to see why, if this is not yet clear). So instead, after we extract the data from our root element, we move the element from the last occupied index (7, in this case) all the way up to the "root" or "start" index 1, replacing the root that was there before. Then, we "bubble" or "percolate" that element down: as long as it has a less urgent (i.e., higher) priority index than its child, we swap it with that child (located at index i*2 or i*2+1). For example, if we removed "t":2, we would first swap up the element "b":4 from index 7 to index 1, then bubble "b":4 down one level, 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 child (which is an 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       8       9
      +-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+
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—this is why most priority queues are now implemented using binary heaps, as we noted above. 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 runs in just O(1), since the most urgent-priority element 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. in lexicographic order, where "a" comes before "b"). Compare strings using the standard relational operators like <,<=,>,>=,==,and!=. Do not make assumptions about the lengths of strings.

Changing the priority of an existing value requires you to loop over the heap to find that value; once you find it, set its new priority and bubble that value up 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 indices in which 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.

Finally, please write your assessment of the Big-O efficiency of your enqueue and dequeue operations in the comments in **`HeapPriorityQueue.h`**.