Assignment 5A: Patient Queue

Assignment by Marty Stepp. Based on Priority Queue assignment by Jerry Cain.

Links Description Implementation Binary Heap Extension Style FAQ Extras

In the problem, you will implement a collection called a *priority queue*. This problem focuses on linked lists and pointers, along with classes and objects. We also strongly suggest that you implement a binary heap as an extension.

This is a **pair assignment**. You are allowed to work individually or work with a single partner. If you work as a pair, **comment both members' names** on top of every submitted code file. Only one of you should submit the assignment; do not turn in two copies.

Links:

Starter Code

We provide a ZIP archive with a starter project that you should download and open with Qt Creator. You will edit and turn in only the following file. The ZIP contains other files/libraries; do not modify these. Your code must work with the other files unmodified. If you want to declare function prototypes, declare them at the top of your **.cpp** file, not by modifying our provided **.h** file.

- **linkedpq.cpp**, the C++ implementation for your class
- **linkedpq.h**, the C++ header for your class
- **heappq.cpp** (extension only)
- **heappq.h** (extension only)

demo JAR

If you want to further verify the expected behavior of your program, you can download the provided sample solution demo JAR and run it. If the behavior of our demo in any way conflicts with the information in this spec, you should favor the spec over the demo.

"How do I run the assignment solution demos?"

Our assignments offer a solution 'demo' that you can run to see the program's expected behavior. On many machines, all you have to do is download the .jar file, then double-click it to run it. But on some Macs, it won't work; your operating system will say that it doesn't know how to launch the file. If you have that issue, download the file, go to the Downloads folder in your Finder, right-click on the file, and click Open, and then press Confirm.

Some Mac users see a security error such as, "cs106b-hw2-wordladder-demo.jar can't be opened because it is from an unidentified developer." To fix this, go to System Preferences → Security & Privacy. You will see a section about downloaded apps. You should see a prompt asking if you would like to allow access to our solution JAR. Follow the steps and then you should be able to open the demo.

If all else fails, you could run the demo JAR from a terminal. Every operating system allows you to open a "terminal" or "console" window for typing raw system commands. Open your operating system's terminal or console window (Google if you need to learn how to do this), and then type:

```
cd DIRECTORY_WHERE_DEMO_JAR_IS_STOREDjava -jar JAR_FILE_NAME
```

For example, on a Mac machine, if your user name is jsmith12 and you have saved a demo JAR named hw2.jar in your Documents/106b directory, you would type:

```
cd /users/jsmith12/Documents/106bjava -jar hw2.jar
```

Or on a Windows machine, if your user name is jsmith12 and you have saved a demo JAR named hw2.jar in your Documents/106b directory, you would type:

```
cd C:\users\jsmith12\Documents\106bjava -jar hw2.jar
```

output logs:

- log #1
- log #2

Try pressing **Ctrl+1** and **Ctrl+2** in the console window to automatically type the user input and compare your output to these logs!

Problem Description:

In this problem you will write a class called a *patient queue* ("PQ" for short) 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 most urgent priority (smallest integer value) comes out first, then the second-smallest, and so on, with the least urgent priority (largest integer value) item coming out last. A queue that processes its elements in order of increasing priority like this is also called a *priority queue*.

LinkedPQ:

Your implementation of the patient queue will be a `LinkedPQ` class that uses 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.

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. The following is a diagram of the internal linked list state that a `LinkedPQ` should have after enqueuing the elements listed previously:

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

Notice that some of the elements have the same priority, such as 5:Dolores, 5:William, and 5:Teddy. The relative order of these tied elements is based on the order they were added, with Dolores appearing first because she was added earliest to the queue, and so on.

Implementation Details:

(It is worth noting that outside of the context of this homework assignment, internally a given implementation of a priority queue might not actually store its elements in sorted order; all that matters is that when they 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.)

As stated previously, your `LinkedPQ` class must use a sorted linked list as its internal data storage. This class is required to have only a single private member variable inside it: a pointer to the front node of your list. This pointer must be set to a null value (`NULL`) if the list is empty.

We supply you with a **PNode** structure (in **pnode.h**/**cpp**) that is a small structure representing a single node of the linked list. Each **PNode** stores information about one patient in the queue, including a string for the patient's name and an integer for their priority, along with a pointer to a next node. You should use this structure to store the elements of your queue along with their priorities. Its declaration looks like the following:

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

    // constructor - each parameter is optional
    PNode(string name = "", int priority = 0, PNode* next = NULL);
};
```

As stated previously, 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 dequeuing, you do not need to search the linked list to find the element with most urgent priority to remove/return it; it is always at the front of the list. 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 list.

As shown previously, the following is a diagram of the internal linked list state of a `LinkedPQ` after enqueuing the elements listed previously:

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

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, suppose you want 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
                                                                    ^
                                                                    |
                                                           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}
                                                                    ^      |
                                                                    |      v
                                                           current ---+     {5:Stub
```

Suppose you next wanted to insert the value "Maeve" with priority 1 into the list shown above. You would insert the new node at the front of the list, as shown below:

```
front     {2:Ford} -> {4:Bernard} -> {5:Dolores} -> {5:William} -> {5:Teddy} -> {
    |      ^
    v      |
    {1:Maeve}
```

LinkedPQ Operations:

Your class must support all of the following operations. 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
`LinkedPQ()`	In this parameterless constructor you should initialize a new empty queue (with a null front for your internal linked list).
`~LinkedPQ()`	In this destructor you must free up any memory used by your linked list nodes.
***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 and should be added just like any other value. Any string is a legal value, and any integer is a legal priority; there are no invalid values that can be passed. This is equivalent to an **`add`** or **`enqueue`** operation on your collection.
***pq*.processPatient()**	In this function you should remove 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. This is equivalent to a **`remove`** or **`dequeue`** operation on your collection.
***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. You should throw a string **exception** if the queue does not contain any patients. This is equivalent to a **`peek`** operation on your collection.
***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. You should throw a string **exception** if the queue does not contain any patients. This is equivalent to a **`peek`** operation on your collection that peeks at the priority rather than the value.
***pq*.upgradePatient(*name*, *oldPriority*, *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 from the old value to the new value, giving this patient a more urgent priority (a smaller integer) than its old value. Perhaps this is needed because the patient's condition has gotten worse, so they need to be seen by the hospital with more haste. If the given patient is present in the queue and has exactly the given old priority as their priority, you are to change the patient's priority to the given new priority and alter the ordering of your list so that the list remains in sorted order. If the given patient name occurs multiple times in the queue with the given old priority, you should alter the priority of the first occurrence you find that has the given old priority when searching your linked list from the start. If the given new priority is not more urgent (lesser integer value) than the old priority, or if the given patient is not present anywhere in the queue with exactly the given old priority value, your function should throw a string **exception**.
***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.
***pq*.clear();**	In this function you should remove all elements from the patient queue, freeing memory for all nodes that are removed.
***pq*.debug();**	In this function you can do anything you want. Our client program calls **`debug()`** on your list after every operation it performs. So, for example, if you want to print some debugging information about the state of your linked list to **`cout`** before or after each operation, you could put that code here in your **`debug`** member function. You don't have to put any code in this function; it is fine to leave it completely blank if you like.

`ostream << pq`	You should write a **`<<`** operator for printing your patient queue to the console. The elements should be printed out in front-to-back order and must be in the form of *value*:*priority* with **`{}`** braces and separated by a comma and space, such as: **`{2:Ford, 4:Bernard, 5:Dolores, 5:William, 5:Teddy, 8:Arnold}`** The **PNode** 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.

*members you must write in **`LinkedPQ`** 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 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 previously 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. Make a member function and/or parameter **`const`** if it does not perform modification of the object's state.

You will also need to do the following:

Add descriptive comments to linkedpq.h/cpp. Both 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 linkedpq.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 linkedpq.cpp.

Make member functions `const` as appropriate. Some of the member functions you will write do not modify the state of the patient queue. For all such functions, mark them as **`const`** at the end of their headers in the .h and .cpp files in your code. For example, if a member named **`foo`** doesn't modify the list, you should change its header to:

```
class LinkedPQ {
    ...
    void functionName() const;
};
```

Other than adding **`const`** to some headers, you should not modify the provided public member headers in any way.

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 enqueuing 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 first one found). 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, **`PNode**`**). If you like, you are allowed to use a reference to a pointer (e.g. **`PNode*&`**).

You should not create any more **PNode** objects than necessary. For example, if a **`LinkedPQ`** contains 6 elements, there should be exactly 6 **PNode** 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 PNode`** 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 **PNode**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 different list states and transitions between them in your code as you add and remove elements. You should also think about the effect of adding a node in the front, middle, and back of the list of a given state.

zero nodes

```
front /
```

one node

```
              +---+---+
front -->     | ? | / |
              +---+---+
```

N nodes

```
              +---+---+     +---+---+         +---+---+    +---+---+
front -->     | ? |   | --> | ? |   |--> ... | ? |   |--> | ? | / |
              +---+---+     +---+---+         +---+---+    +---+---+
```

Heap Implementation (optional extension)

The second 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       8
      +-------+-------+-------+-------+-------+-------+-------+-------+-------+--
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 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       8
      +-------+-------+-------+-------+-------+-------+-------+-------+-------+--
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       8
      +-------+-------+-------+-------+-------+-------+-------+-------+-------+--
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 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.

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 the heap PQ, 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.

Hint: In order to test your binary heap, add the line **`#include heappq.h`** to **hospitalmain.cpp**, and the console will give you an option of which PQ to test.

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 in your **`LinkedPQ`** is a **`PNode*`** 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 PQ 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: In your **`LinkedPQ`**, 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 **`PNode`** structures than necessary. For example, if the client has called **`newPatient`** 6 times on a **`LinkedPQ`**, your code should have created exactly 6 total **`PNode`** objects; no more, no less. You shouldn't, say, have a seventh empty "dummy" node at the front or back of the **`LinkedPQ`**'s 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. The one exception to this rule is `upgradePatient`, which is allowed to make two passes, one to find/remove the element and a second to add it with its new priority.

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 walk across 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 `LinkedPQ` object. It is difficult to be certain that code does not contain memory leaks; if you want to try to verify this, you could consider putting a print statement or counter in the list node class that prints every time a node is created or destroyed. Counting the number of these messages would tell you whether every allocated node was freed by your program.

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.

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: 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? Do I really need it?
A: Free up any dynamic memory that you previously allocated with the `new` keyword. Yes, you need it.

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 `PNode` and `PNode*` ? Which one should I use?
A: You literally never want to create a variable of type `PNode` ; you want only `PNode*` . The former is an object, the latter is a pointer to an object. You always want pointers to `PNode` 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 `PNode`?
A: Since the linked list PQ needs to keep its memory around dynamically, you should use the `new` keyword. Like this:

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

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

```
PNode* node = new PNode(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!
PNode node;            // bad bad bad
node.data = 42;        // bad bad bad
node.next = NULL;   // bad bad bad
...
```

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.

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.
- **Other kinds of PQs:** Write additional classes that implement the PQ operations in different ways. For example, learn about various kinds of *heaps* that can be used to implement a priority queue. If you write files **extrapq.h** and **extrapq.cpp**, our `main` file can be modified without too much work to make it use your queues alongside the required ones.
- **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.

Honor Code Reminder:

You are expected to follow the **Stanford Honor Code**.

- In your file's comment header, list your name (and your partner's name, if you worked in a pair); also cite *all* sources of help, including books, web pages, friends, section leaders, etc.
- Do not consult any assignment solutions that are not your (pair's) own.
- Do not attempt to disguise any code that is not your (pair's) own.

- Do not give out your assignment solution to another student.
- Do not post your homework solution code online. (e.g. PasteBin, DropBox, web forums).
- Please take steps to ensure that your work is not easily copied by others.

Remember that we run **similarity-detection software** over all solutions, including this quarter and past quarters, as well as any solutions we find on the web.

If you need help solving an assignment, we are happy to help you. You can go to the LaIR, or the course message forum, or email your section leader, or visit the instructor/Head TA during their office hours. **You can do it!**