PA9 PatientQueue

This assignment is based on an assignment written by instructors at Stanford University. It represents a lot of hard work by the lecturers at Stanford University who have taught CS106B/X over the years.

Due: Wednesday 4/6 by 11:30PM
Submission:

- PatientQueue.java - implementation of a Patient Queue.

Overview

The goal of this assignment is to practice implementing your own data structure. Specifically you will implement a priority queue using a binary minimum heap. Remember to use good coding practices. Decompose common code into separate private helper functions.

We hope you have gotten plenty of practice with generics by now. So for this assignment, your priority queue will be specific to Patient objects. You will not need to make it generic.

Assignment

You will write a class called PatientQueue. 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.

For a binary minimum heap, the element with the minimum priority value is frontmost. That is, regardless of the order in which you add/enqueue the elements, the minimum priority element is dequeued first, then the second-smallest, and so on, with the largest priority element coming out last.

For example, if the following patients arrive at the hospital in this order and the priority they are assigned is shown in parentheses:

- ”Anat” (4)
- ”Ben” (9)
- ”Sasha” (8)
- ”Wu” (7)
- ”Rein” (6)
- ”Ford” (2)

If you were to place them into a patient queue and dequeue them, they would come out in this order:

- Ford, Anat, Rein, Wu, Sasha, Ben

Binary Heap

The patient queue is a specific application of a well-known abstract data type called a priority queue. You can think of a priority queue as a sorted queue where the elements are sorted by priority. But the priority queue need not internally store its elements in sorted order; all that matters is that the dequeue operation accesses the frontmost element (which will be the minimum priority). As we will see, this difference between the external expected behavior of the priority queue and its true internal state can lead to an interesting implementation.

The priority queue implementation you are to write stores the elements into an unfilled array. The array elements are organized into an arrangement called a binary heap. A binary heap that is arranged to put the minimum priority element frontmost is classified as a min-heap.

As discussed in lecture, a binary heap is an array that obeys a heap ordering property. Each index i is thought of as the ”parent” of the two ”child” indexes, i * 2 and i * 2 + 1. (To simplify the index math, we strongly suggest leaving index 0 unused and starting the data at index 1.) A parent is in front of its children in the queue. For a min-heap, this means the parent must have a smaller priority than its children, i.e. it is the minimum of the ”family”.

One very desirable feature of a binary heap is that the frontmost element (the one that should be returned by peek or dequeue) is always located at index 1. For example, suppose we added the six patients from the bulleted list in the previous section to a min-heap. The frontmost element is stored at index 1; that element is ”Ford” which has minimum priority value 2.

index	0	1	2	3	4	5	6	7	8	9
value	..	”Ford” (2)	”Rein” (6)	”Anat” (4)	”Ben” (9)	”Wu” (7)	”Sasha” (8)	..	..	..

Enqueue

Adding (enqueuing) to a heap is done by placing the new element at the end of the array in the first empty index and then ”bubbling up” that element to its proper position according to heap order. Effectively, this starts the new element near the ”back” of the queue and jostles its way toward the front by moving ahead of its parent(s) where appropriate. If you would like to see a full trace of the steps necessary to add the preceding six elements to the heap, they are shown below.

The sequence of enqueue operations would produce the following states of the min-heap:

- enqueue ”Anat” (4)

index	0	1	2	3	4	5	6	7	8	9
value	..	”Anat” (4)	..	..	..	..	..	..	..	..

- enqueue ”Ben” (9)

index	0	1	2	3	4	5	6	7	8	9
value	..	”Anat” (4)	”Ben” (9)	..	..	..	..	..	..	..

- enqueue ”Sasha” (8)

index	0	1	2	3	4	5	6	7	8	9
value	..	”Anat” (4)	”Ben” (9)	”Sasha” (8)	..	..	..	..	..	..

- enqueue ”Wu” (7)

index	0	1	2	3	4	5	6	7	8	9
value	..	”Anat” (4)	”Ben” (9)	”Sasha” (8)	”Wu” (7)	..	..	..	..	..

(Wu bubbles up from index 4 to 2)

index	0	1	2	3	4	5	6	7	8	9
value	..	”Anat” (4)	”Wu” (7)	”Sasha” (8)	”Ben” (9)	..	..	..	..	..

- enqueue ”Rein” (6)

index	0	1	2	3	4	5	6	7	8	9
value	..	”Anat” (4)	”Wu” (7)	”Sasha” (8)	”Ben” (9)	”Rein” (6)	..	..	..	..

(Rein bubbles up from index 5 to 2)

index	0	1	2	3	4	5	6	7	8	9
value	..	”Anat” (4)	”Rein” (6)	”Sasha” (8)	”Ben” (9)	”Wu” (7)	..	..	..	..

- enqueue ”Ford” (2)

index	0	1	2	3	4	5	6	7	8	9
value	..	”Anat” (4)	”Rein” (6)	”Sasha” (8)	”Ben” (9)	”Wu” (7)	”Ford” (2)	..	..	..

(Ford bubbles up from index 6 to 3 to 1)

index	0	1	2	3	4	5	6	7	8	9
value	..	”Ford” (2)	”Rein” (6)	”Anat” (4)	”Ben” (9)	”Wu” (7)	”Sasha” (8)	..	..	..

The heap order for a min-heap dictates that a parent must be the minimum of the family. If the child just added is smaller than its parent, it belongs in front of it, so the two elements must be swapped. If after swapping, the element that moved ahead is smaller than its new parent, it swaps again. The swapping continues until the element has moved frontward to the proper position and heap order has been restored.

Let’s trace adding ”Eve” with priority 3 to the above min-heap. She is first placed into index 7:

index	0	1	2	3	4	5	6	7	8	9
value	..	”Ford” (2)	”Rein” (6)	”Anat” (4)	”Ben” (9)	”Wu” (7)	”Sasha” (8)	”Eve” (3)	..	..

But this can temporarily break the heap ordering, so we must fix it. An element at index i has its parent at index i/2, so Eve’s parent index is 7/2 = 3 (using integer division). Because Eve’s priority (3) is smaller than her parent Anat’s (4), these two elements are swapped to move Eve in front of Anat. Eve stops bubbling here as her priority (3) is not smaller than her new parent Ford’s (2).

The final heap contents after enqueuing ”Eve” would be:

index	0	1	2	3	4	5	6	7	8	9
value	..	”Ford” (2)	”Rein” (6)	”Eve” (3)	”Ben” (9)	”Wu” (7)	”Sasha” (8)	”Anat” (4)	..	..

Dequeue

Removing (dequeuing) an element from a heap is done by replacing the element at index 1 with the element from the last occupied index of the array and then ”bubbling down” that element to the proper position according to heap order. Effectively, this takes an element that was formerly near the ”back” of the queue, temporarily puts it in the very front, and then sinks it backward to the proper place by moving it behind its children where appropriate.

Heap order for a min-heap dictates that a parent must be the minimum of the family. If a parent is larger than either/both children, it belongs behind it, so the smaller of the two children must be swapped with the parent. If after swapping, the element that moved behind

is still larger than either/both children, it must swap again. The element continues swapping toward the back until it settles in the proper position and heap order has been restored.

Let's trace dequeuing from the min-heap above. The element at index 7, "Anat" (4), is initially used to replace the dequeued element at index 1:

index	0	1	2	3	4	5	6	7	8	9
value	..	"Anat" (4)	"Rein" (6)	"Eve" (3)	"Ben" (9)	"Wu" (7)	"Sasha" (8)	..	..	..

But this can temporarily break the heap ordering, so we must fix it. Anat's priority is compared to her two children at indexes 2 and 3. Because Anat (4) is not smaller than its child Eve (3), Anat must swap with Eve to bring the smaller child in front. No further swapping is needed because Anat's priority (4) is smaller than the priority of its new only child, Sasha (8).

The final heap contents after dequeuing "Ford" would be:

index	0	1	2	3	4	5	6	7	8	9
value	..	"Eve" (3)	"Rein" (6)	"Anat" (4)	"Ben" (9)	"Wu" (7)	"Sasha" (8)	..	..	..

Change Priority

To change the priority of an existing element in a binary heap, you must first loop over the entire heap to find the element. Once found, you set its new priority and then "bubble" the element forward or backward from its current position to restore heap order.

For example, if starting with the min-heap above, changing the priority of Eve to 10 would cause her to move backwards in the queue, first swapping with Anat and then again with Sasha. The final heap contents would be:

index	0	1	2	3	4	5	6	7	8	9
value	..	"Anat" (4)	"Rein" (6)	"Sasha" (8)	"Ben" (9)	"Wu" (7)	"Eve" (10)	..	..	..

A key benefit of using a binary heap implementation for a priority queue is efficiency. The peek operation is very fast since the frontmost element is always stored at index 1. The "bubbling" process moves an out-of-place element to its proper position in step sizes of increasing powers of 2, which means only a few steps are needed to cover a lot of ground. This makes the enqueue and dequeue operations both fast.

Patient Class

We supply you with a Patient type in Patient.java. This type represents a single entry in the queue. Each patient stores information about one patient, including a String for the patient's name and an integer for their priority.

Since the class is small and simple, it uses public member variables that you can access directly. (You should not have public member variables in your PatientQueue class).

Required Methods

Your PatientQueue class must support the following operations. The header for each member function must match those specified below. Do not change the parameter or return types or function names.

Member Name	Description
PatientQueue()	In this constructor you should initialize a new empty queue that does not contain any patients. Initially your queue's internal heap array should have a capacity of 10 elements. The frontmost element is stored at index 1 in your array.
void enqueue(String name, int priority)	In this function you should add the given person into your patient queue with the given priority. You must "bubble up" appropriately to keep your array in proper heap order. 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. If there is no room in your queue's internal array, you must resize it to a larger array with **twice the capacity**.
void enqueue(Patient patient)	This function is the same as the above function but takes in a Patient object directly instead of a name and priority.
String dequeue()	In this function you should remove the frontmost patient in your queue, and return their name as a string. You should throw an exception if the queue is empty.
String peek()	In this function you should return the name of the frontmost patient in the queue, without removing or altering the state of the queue. You should throw an exception if the queue is empty.
int peekPriority()	In this function you should return the integer priority of the frontmost patient in the queue without removing it or altering the state of the queue. You should throw an exception if the queue is empty.
void changePriority(String name, int newPriority)	In this function you should modify the priority of a given existing patient in your queue. Perhaps this is needed because the patient's condition has gotten better or worse, so they need to be seen by the hospital with more or less haste. You must maintain the proper heap ordering after the priority change. This may require you to "bubble" the patient forward or backward in the queue. For example, in a min-heap, changing to a smaller integer priority value might cause the patient to move frontward ahead of others in the queue, or changing to a larger integer priority might cause the patient to move backward behind others. A request to change a patient's priority to the same value it already has should have no effect on the queue. If the given patient name occurs multiple times in the queue, you should alter the priority of the first occurrence you find when searching your array from the start index. If the given name is not in the queue, nothing should happen.
boolean 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.
int size()	In this function you should return the number of elements in your patient queue.
void clear()	In this function you should remove all elements from your patient queue.
String toString()	You should override the toString method so that your PatientQueue has a useful way to print itself. Each element is printed in the format value (priority). Elements are separated by a comma and space and the entire sequence is enclosed in braces. The elements should be printed in the order they appear in the heap array. Your formatting and spacing should match our example exactly. {Anat (4), Rein (6), Sasha (8), Ben (9), Wu (7), Eve (10)}

Priority Ties & Constraints

- If there is ever a tie in priority, break ties by comparing the strings themselves, treating a string that comes earlier in the alphabet as being in front (e.g. "Anat" before "Ben"). Do not make assumptions about the string lengths.
- Duplicate patient names and priorities are allowed in your queue. The changePriority operation should affect only a single occurrence of a patient's name (the first one found). If the queue contains other occurrences of that same name, a single call to changePriority shouldn't affect them all.
- You should not make unnecessary passes over the queue. For example, when enqueuing an element, a poor implementation would be to traverse the entire queue. The point of using a binary heap is that you should not need to do that.
- You are not allowed to use a sort function or library to arrange the elements of your queue.
- **You are not allowed to use any Java collections. You may only use an array.**
- You are not allowed to create any temporary or auxiliary data structures (such as additional arrays) anywhere in your code, other than when you are resizing to a larger array on an enqueue operation. You must implement all behavior using only the one heap array of patients as specified.

Tips and Advice

- It is useful to implement toString early on in development so that you can use this function to debug your other functions.
- Write helper functions: The members listed previously represent your class's required public behavior. But you may add other "helper" member functions to help you implement all of the appropriate behavior. Any other member functions you add 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 sure to exhaustively test your program. Cover as many scenarios as possible.

Grading Criteria

We are not providing testcases for this PA.

We encourage you to write your own JUnit testcases to ensure your classes work properly, but we will not be collecting or grading these test files. We will test your classes with our own testcases.

Your grade will consist of similar style and code clarity points we have looked for in earlier assignments.

Write your own code. We will be using a tool that finds overly similar code. Do not look at other students' code. Do not let other students look at your code or talk in detail about how to solve this programming project. Do not use online resources that have solved the same or similar problems. It is okay to look up, "How do I do X in Java", where X is indexing into an array, splitting a string, or something like that. It is **not** okay to look up, "How do I solve {a programming assignment from CSc210}" and copy someone else's hard work in coming up with a working algorithm.