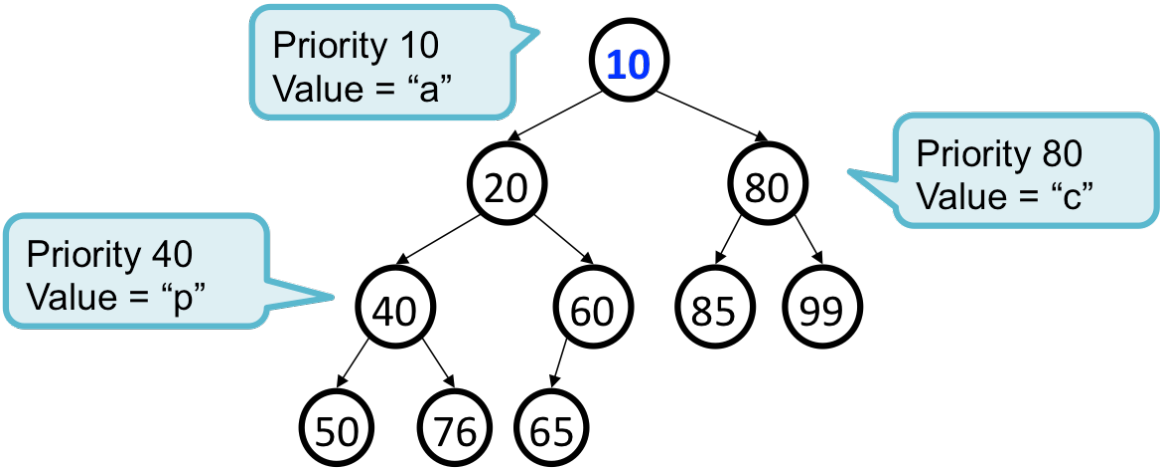
Assignment 5: PQueue

Created by Julie Zelenski, with edits by Jerry Cain, Keith Schwarz, Marty Stepp, Chris Piech and others.

NOVEMBER 7TH, 2016



This assignment focuses on implementing a priority queue ("PQ") collection using several internal data structures, using pointers, arrays, dynamic memory allocation, linked lists, and heaps. The starter code for this project is available as a ZIP archive. A demo is availible as a JAR (see handout on how to run a jar):

Starter Code

Demo Jar

Turn in the following files:

1. **`ArrayPriorityQueue.h / .cpp`** , a PQ implementation using an unsorted array as internal storage
2. **`LinkedPriorityQueue.h / .cpp`**, the C++ code for all of your recursion functions.
3. **`HeapPriorityQueue.h / .cpp`**, the C++ code for all of your recursion functions.

We provide you with several other files to help you, but you should not modify them.

This is a pair assignment. You may work in a pair or alone. Find a partner in section or use the course message board. If you work as a pair, comment both members' names atop every code file. Only one of you should submit the program; do not turn in two copies. Submit using the Paperless system linked on the class web site.

Due Date: PQueue is due Wednesday November 16th at 12:00pm.

Y.E.A.H hours:

Wed, Nov 9th

T.B.A.

T.B.A.

The Priority Queue Collection

A priority queue is a collection that is similar to a queue, except that the elements are enqueued with "priority" ratings. Then when the elements are dequeued later, they always come out in increasing order of priority. 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 queues are useful for modeling tasks where some jobs are more important to process than others, like the line of patients at an emergency room (a severely injured patient has higher "priority" than one with a cold).

We will use the convention that a smaller priority number means a greater urgency, such that a priority-1 item would take precedence over a priority-2 item. Because terms like "higher priority" can be confusing, since a "higher" priority means a lower integer value, we will follow the convention in this document of referring to a "more urgent" priority for a smaller integer and a "less urgent" priority for a greater integer.

In this assignment, you will be writing three different implementations of a priority queue class that stores strings. If two strings in the queue have the same priority, you will break ties by considering the one that comes first in alphabetical order to come first. Use C++'s built-in relational operators (<, >, <=, etc.) to compare the strings.

For example, if you enqueue these strings into a priority queue:

- enqueue "x" with priority 5
- enqueue "b" with priority 4
- enqueue "a" with priority 8
- enqueue "m" with priority 5
- enqueue "q" with priority 5
- enqueue "t" with priority 2

Then if you were to dequeue the strings, they would be returned in this order:

- "t", then "b", then "m", then "q", then "x", then "a"

You could think of a priority queue as a **sorted queue** where the elements are sorted by priority, breaking ties by comparing the string elements themselves. But internally the 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. An actual priority queue implementation is not required to store its internal data in any particular order, so long as it dequeues its elements in increasing order of priority. As we will see, this difference between the external expected behavior of the priority queue and its true internal state can lead to interesting differences in implementation.

Your task: You will implement each of these three versions of PQueue. For each implementation in addition to writing the implementation, in the comments include the worst case big-O for enqueue and dequeue.

Priority Queue Implementations:

1) **`ArrayPriorityQueue`**: The first priority queue implementation you will write uses an unsorted array 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. The elements of the array are not stored in sorted order internally. As new elements are enqueued, you should simply add them to the end of the array. When dequeuing, you must then search the array to find the smallest element and remove/return it. This implementation is simple to write and optimized for fast enqueuing but has slow dequeue/peeking and poor overall performance. The following is a diagram of the internal array state of an **`ArrayPriorityQueue`** after enqueuing the elements listed on the previous section:

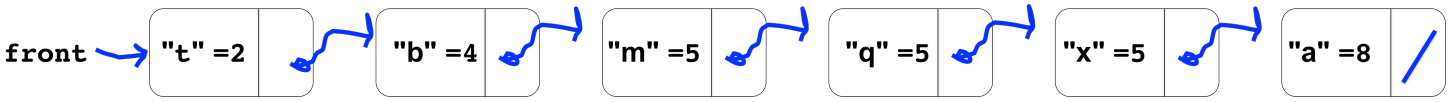
```
index     0       1       2       3       4       5       6       7       8       9
      +-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+
value | "x":5 | "b":4 | "a":8 | "m":5 | "q":5 | "t":2 |       |       |       |       |
      +-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+
```

We supply you with a **`PQEntry`** structure that is a small object containing a string value and an integer priority. You should use this structure to store the elements of your priority queue along with their priorities in the array.

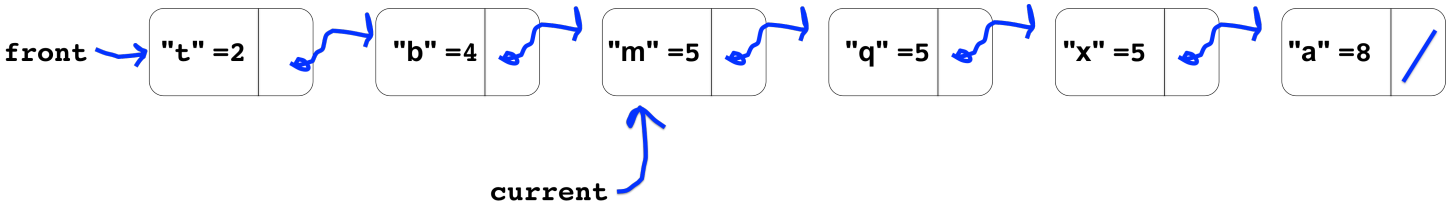
Remember to include the big-O of enqueue and dequeue in the comments in **`ArrayPriorityQueue.h`**.

---

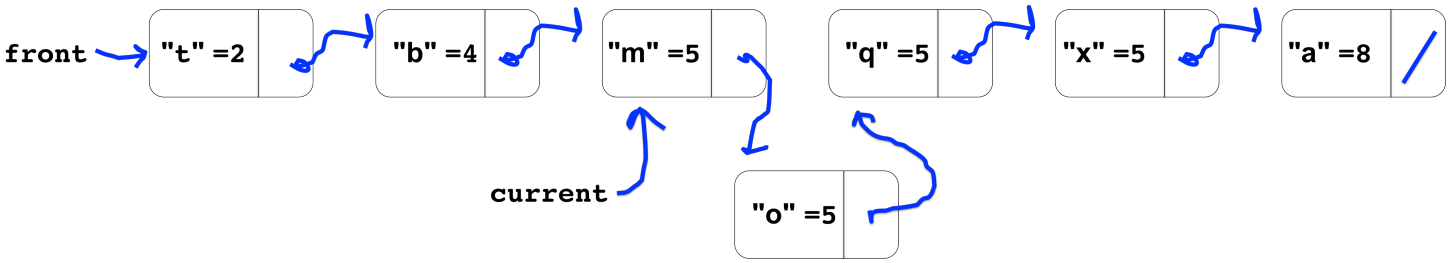
2) **`LinkedPriorityQueue`**: The second priority queue implementation you will write uses a sorted linked list as its internal data storage. This class is only allowed to have a single private member variable inside it: a pointer to the front of your list. The elements of the linked list are stored 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. When dequeuing, 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. This implementation is harder to write than the array implementation, and enqueuing to it is slower, but it is much faster for dequeue/peeking and has better overall performance. The following is a diagram of the internal linked list state of a LinkedPriorityQueue after enqueuing the elements listed on the previous section:



The hardest part of this implementation is inserting a new node in the proper place when enqueue is called. You must look for the proper insertion point by finding the first element whose priority is at least as large as the new value to insert, breaking ties by comparing the strings. 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 "o" with priority 5 into the list shown above , your code should iterate until you have a pointer to the node containing "m", as shown below:



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



We supply you with a **`ListNode`** structure that is a small object representing a single node of the linked list. Each ListNode stores a string value and integer priority in it, and a pointer to a next node. You should use this structure to store the elements of your priority queue along with their priorities.

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

3) **HeapPriorityQueue**: The third priority queue implementation you will 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       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 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       9
      +-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+
value |       | "t":2 | "m":5 | "y":3 | "x":5 | "q":5 | "a":8 | "b":4 |       |       |
      +-------+-------+-------+-------+-------+-------+-------+-------+-------+-------+
size = 7
capacity = 11
```

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       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. 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!=. Donotmakeassumptionsaboutthelengthsofthestrings.

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`**.

Priority Queue Operations:

Each of your three priority queue implementations must support all of the following operations.

Member	Description
`pq.enqueue(value, priority);`	In this function you should add the given string value into your priority queue with the given priority. Duplicates are allowed. Any string is a legal value, and any integer is a legal priority; there are no invalid values that can be passed.
`pq.dequeue()`	In this function you should remove the element with the most urgent priority from your priority queue, and you should also return it. You should throw a string exception if the queue does not contain any elements.
`pq.peek()`	In this function you should return the string element with the most urgent priority from your priority queue, without removing it or altering the state of the queue. You should throw a string exception if the queue does not contain any elements.
`pq.peekPriority()`	In this function you should return the integer priority that is most urgent from your priority queue (the priority associated with the string that would be returned by a call to peek), without removing it or altering the state of the queue. You should throw a string exception if the queue does not contain any elements.
`pq.changePriority(value, newPriority);`	In this function you will modify the priority of a given existing value in the queue. The intent is to change the value's priority to be more urgent (smaller integer) than its current value. If the given value is present in the queue and already has a more urgent priority to the given new priority, or if the given value is not already in the queue, your function should throw a string exception. If the given value occurs multiple times in the priority queue, you should alter the priority of the first occurrence you find when searching your internal data from the start.
`pq.isEmpty()`	In this function you should return true if your priority queue does not contain any elements and false if it does contain at least one element.
`pq.size()`	In this function you should return the number of elements in your priority queue.
`pq.clear();`	In this function you should remove all elements from the priority queue.
`out << pq`	You should write a << operator for printing your priority queue to the console. The elements can print out in any order and must be in the form of "value":priority with {} braces, suchas {"t":2,"b":4,"m":5,"q":5,"x":5,"a":8}. The PQEntry and ListNode structures both have >>operators that may be useful. Your formatting and spacing should match exactly. Do not place a \n or endl at the end.

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

Constructor/destructor: Each class must also define a parameterless constructor. If the implementation allocates any dynamic memory, you must ensure that there are no memory leaks by freeing any allocated memory at the appropriate time. This will mean that you will need a destructor for classes that dynamically allocate memory.

Helper functions: The members listed on the previous page represent a large fraction of each class's behavior. But you should add other members 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.

Member variables: We have already specified what member variables you should have. Here are some other constraints:

- Don't make something a member variable if it is only needed by one function. Make it local. Making a variable into a data member that could have been a local variable or parameter will hurt your Style grade.
- All data member variables in side each of your classes should be private.

Other Implementation Details:

Here are a few other constraints we expect you to follow that don't fit neatly into any other section.

- The ArrayPriorityQueue has many similar aspects to the ArrayList written in lecture. It's completely fine to borrow any of the code from that lecture and use it in your Array PQ as needed. (Cite it as a source.)
- The ArrayPriorityQueue's operations like changePriority, enqueue, and dequeue should not needlessly rearrange the elements of the array any more than necessary. For example, when changing an element's priority, don't remove and re-add that element to the array.
- The LinkedPriorityQueue's operations 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 spot to insert, and then make a second traversal to get back to the spot to insert and add the new element. Do not make such multiple passes. Also, keep in mind that your LinkedPriorityQueue 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.
- The HeapPriorityQueue must implement its operations efficiently using the "bubbling" or "percolating" described in this handout. It is important that these operations run in O(log N) time.
- Duplicates are allowed in your priority queue, so be mindful of this. For example, the changePriority operation should affect only a single occurrence of a value (the first one found). If there are other occurrences of that same value in the queue, a single call to changePriority shouldn't affect them all.
- You are not allowed to use a sort function to arrange the elements of any collection, nor are you allowed to create any temporary or auxiliary data structures inside any of your priority queue implementations. They must implement all of their behavior using only their primary internal data structure as specified.
- You will need pointers for several of your implementations, but you should not use pointers-to-pointers (for example, ListNode**) or references to pointers (e.g. ListNode*&).
- You should not create any more ListNode objects than necessary. For example, if a LinkedPriorityQueue contains 6 elements, there should be exactly 6 ListNode objects in the chain, 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. You can declare as many local variable pointers to ListNodes as you like.

Development Strategy and Hints:

- **Order of Implementation**: While you are free to implement the three priority queues in any order you wish, we strongly recommend that you implement them in the order we specified: Array, then Linked, then Heap. This goes from simplest to most difficult.
- **Draw pictures**. When manipulating linkedlists, it is often helpful to draw pictures of the linkedlist 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.
- **Don't panic**. You will be doing a lot of pointer gymnastics in the course of this assignment, and you will almost certainly encounter a crash in the course of writing your program. If your program crashes, resist the urge to immediately make changes to your code. Instead, look over your code methodically. Use the debugger to step through the code one piece at a time, or use the provided testing harness to execute specific commands on the priority queue. The bug is waiting there to be found, and with persistence you will find it. If your program crashes with a specific error message, try to figure out exactly what that message means. Don't hesitate to get in touch with your section leader, and feel free to stop by the LaIR or office hours.
- **Testing**: Extensively test your program. Run the sample solution posted on the class web site to see the expected behavior of your queue classes. It allows you to interactively test each queue by calling any member functions in any order you like. Some provided expected outputs are posted on the class web site, but we do not guarantee that those outputs cover all possible cases. You should perform your own exhaustive testing.

Possible Extra Features:

Thats all! You are done. Consider adding extra features.

Though your solution to this assignment must match all of the specifications mentioned previously, it is allowed and encouraged for you to add extra features to your program if you'd like to go beyond the basic assignment. Here are some example ideas for extra features that you could add to your program.

1. **Sorted Unfilled Array**: Similar to arrayPQ, but store elements in sorted order.
2. **Map of Queues**: Use a Map with integer priorities as its keys and queues of strings as the values associated with those keys. This puts all elements with the same priority into an inner queue together.
3. **Doubly-Linked List**: Write a linked list whose nodes have both prev and next pointers. This makes it easy to move both forward and backward in the list, which can speed up certain operations like changing priority. Also declare both a front and a back pointer that point to the first and last element in the list respectively.
4. **Binomial Heap**: A binomial heap is a variation on the binary heap already described that is especially efficient for certain common priority queue operations. See separate handout on class web site describing this idea.

Another good idea is to add operations to each heap beyond those specified:

1. **Merge**: Write a member function that accepts another priority queue of the same type and adds all of its elements into the current priority queue. Do this merging "in place" as much as possible; for example, if you are merging two linked list PQs, directly connect the node pointers of one to the other as appropriate.
2. **Deep Copy**: Make your priority queues properly support the = assignment statement, copy constructor, and deep copying. See the C++ "Rule of Three" and follow that guideline in your implementation.
3. **Iterator**: Write a class that implements the STL iterator and a begin and end function in your priority queues, which would enable "for-each" over your PQ. This requires knowledge of the C++ STL library.

Do you have an interesting idea for an extra feature? Ask the head TA or instructor.

Submitting with extra features: If you complete any extras, please list them in your comment headings. Also please submit your program twice: first without extra features (or with them disabled), and a second time with the extensions.
