Assignment 5: Priority Queue

Priority queue assignment by Julie Zelenski and Jerry Cain with edits by Keith Schwarz.

Now that we've started discussing class implementation techniques, it's time for you to implement your own collection class: the *priority queue*. A priority queue is a modified version of a queue in which elements are not dequeued in the order in which they were inserted. Instead, elements are removed from the queue in order of *priority*. For example, you could use a priority queue to model a hospital emergency room: patients enter in any order, but more critical patients are seen before less critical patients. Similarly, if you were building a self-driving car that needed to process messages from multiple sensors, you might use a priority queue to respond to extremely important messages (say, that a pedestrian has just walked in front of the car) before less important messages (say, that a car two lanes over has just switched on its turn signal).

In the course of this assignment, you will implement a priority queue in several different ways. In doing so, you will learn to master linked lists and dynamic allocation, and will construct a powerful collection class that will serve as a critical building block in later assignments.

Due: Wednesday, May 23rd at 10:00AM

YEAH Hours: Tuesday, May 15 in 380-380C, 4:15PM – 5:45PM

The Priority Queue at a Glance

In this assignment, you will be implementing a priority queue class that stores strings. You can enqueue strings in any order you wish. Whenever you extract a string from the priority queue, the priority queue will remove and return the lexicographically first string in the queue ("lexicographical order" is the fancy computer-scientist way of saying "alphabetical order"). For example, if you insert these five strings into a priority queue:

"Goldilocks," "Mama bear," "Papa bear," "Baby bear"

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

"Baby bear," "Goldilocks," "Mama bear," "Papa bear"

As an FYI, if you compare **`string`**s using C++'s built-in relational operators (**`<`**, **`>`**, **`<=`**, etc.), the comparisons are done lexicographically.

The priority queue interface contains these five operations:

- **`size`**: Returns how many elements are in the priority queue.
- **`isEmpty`**: Returns whether the priority queue is empty.
- **`enqueue`**: Inserts an element into the priority queue.
- **`peek`**: Returns, but does not remove, the lexicographically first string in the queue. If the queue is empty, **`peek`** should call **`error`** to report an error.
- **`dequeueMin`**: Returns and removes the lexicographically first string in the queue. If the queue is empty, **`dequeueMin`** should call **`error`** to report an error.

The Assignment

As you have seen in lecture, there are multiple ways to implement each of the collections classes. Your assignment is to implement a priority queue class in four different ways. Those are:

- **Unsorted `Vector`**. The elements in the priority queue are stored unsorted in a **`Vector`**.
- **Sorted linked list**. The elements are stored in a sorted, singly-linked list.
- **Unsorted linked list**. The elements are stored in an unsorted, doubly-linked list.
- **Binary heap**. The elements are stored in a *binary heap*, a specialized data structure well-suited to priority queues.

While you are free to write these implementations in any order that you wish, we strongly suggest implementing them in the above order.

Implementation One: Unsorted `Vector`

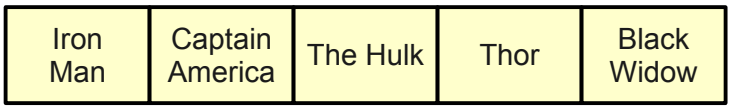
Your first implementation of the priority queue will be backed by an unsorted **`Vector`**. This implementation is one of the simplest, and is designed to help you acclimate to class design and our testing harness.

Inside the **`pqueue-vector.h`**, you will find the interface for a **`VectorPriorityQueue`** 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 **`pqueue-vector.cpp`** source file.

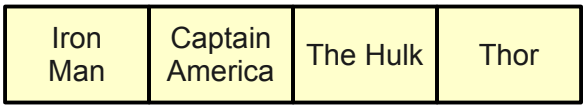
Representing a priority queue with an unsorted **`Vector`** is reasonably straightforward. Whenever an element is enqueued into the priority queue, you simply append it to the underlying **`Vector`**. For example, if you were to enqueue these five strings in order:

"Iron Man," "Captain America," "The Hulk," "Thor," "Black Widow"

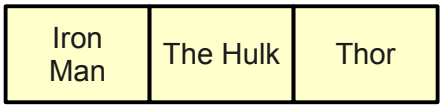
Then the **`Vector`** would look as follows:



To dequeue a string from the priority queue, you simply scan across the **`Vector`** to determine which string comes lexicographically first, remove that string from the **`Vector`**, then return it. For example, calling **`dequeueMin`** on the above priority queue would return "Black Widow" and leave the priority queue holding



Calling **`dequeueMin`** a second time would return "Captain America" and leave the priority queue holding



We have provided you with a test harness in **`pqueue-test.cpp`**. This testing code contains two important tools. First, it contains an interactive testing environment in which you can issue individual commands to your priority queue. This is designed to let you test your priority queue on specific inputs. Once you are comfortable that your priority queue works correctly, you can run our more

elaborate automated test suite. This will push your class implementation hard by running numerous tests specifically designed to smoke out edge cases. If your class passes these tests, it probably works correctly.

After you have implemented the **`VectorPriorityQueue`**, take a minute to think over these questions:

- What is the best-case runtime, in big-O notation, of inserting an element into the priority queue? What is the worst-case runtime?
- What is the best-case runtime, in big-O notation, of removing an element from the priority queue? What is the worst-case runtime?
- You can sort a sequence of elements by inserting them all into a priority queue, then removing them one at a time. If you use the **`VectorPriorityQueue`** as your priority queue for this, you will end up with a well-known sorting algorithm. What sorting algorithm is it?

You don't need to submit answers to these questions, but it's good to think them over.

Implementation Two: Sorted Linked List

When backing the priority queue with an unsorted **`Vector`**, insertions are fast but dequeues are expensive. An alternative approach would be store the elements in the priority queue in sorted order. This increases the time required to insert something into the priority queue, but decreases the amount of work you have to do to find the next element to remove; after all, the smallest element will be right at the front of the sequence.

For your next task, you will implement the priority queue as a sorted, singly-linked list. That is, you will store the elements in a linked list and enforce that the elements are always stored in sorted order. The header file **`pqueue-linkedlist.h`** contains the interface for the **`LinkedListPriorityQueue`** class. Your should implement this class by adding the appropriate fields, methods, and types to the **`private`** section of this class, then implementing the member functions in **`pqueue-linkedlist.cpp`**.

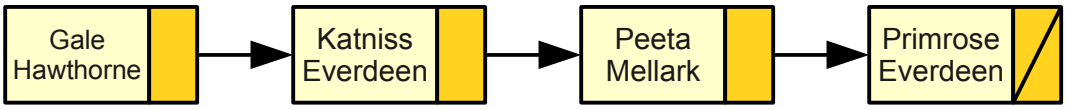
To better understand how this class should be implemented, suppose that you were to insert the following three strings into the **`LinkedListPriorityQueue`**:

"Primrose Everdeen," "Katniss Everdeen," "Peeta Mellark"

In this case, the priority queue would look like this:

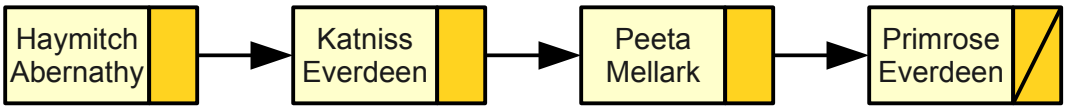
Notice that the elements are in sorted order, even though they weren't added in this order.

If you then insert the string "Gale Hawthorne," the priority queue will contain the following:

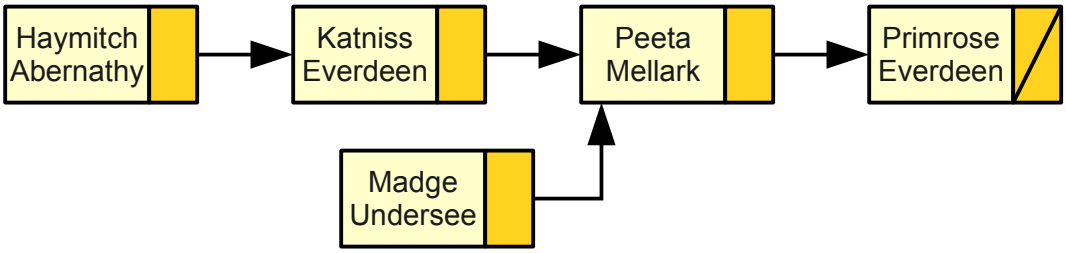


If you now insert "Haymitch Abernathy," then the priority queue will contain

Dequeuing an element from this priority queue is reasonably straightforward. Since the elements of the priority queue are stored in sorted order, you can just remove the first cell from the linked list and return its contents. For example, dequeuing from the above priority queue would yield "Gale Hawthorne" and leave the priority queue structured as



The hardest part of the implementation is determining how to insert an element into the linked list. You will have to search the list for the first element whose value is greater than the new value. For example, to insert "Madge Undersee" into the above linked list, you would determine that it needs to be inserted before "Peeta Mellark," as shown here:



However, in order to splice the new node into the list, you have to change the **`next`** pointer in the *preceding* cell – Katniss Everdeen – so that it points to the new node. To do so, we suggest that you implement your insertion function by iterating across the linked list and keeping track of two pointers: one for the current element, and one for the element just before that in the sequence. That way, when you find where the new element should go, you can splice it into the list by updating the preceding cell.

After you implement the **`LinkedListPriorityQueue`**, think about the following questions. As before, you don't need to submit your answers.

- What is the best-case runtime, in big-O notation, of inserting an element into the priority queue? What is the worst-case runtime?
- What is the best-case runtime, in big-O notation, of removing an element from the priority queue? What is the worst-case runtime?
- You can sort a sequence of elements by inserting them all into a priority queue, then removing them one at a time. If you use the **`LinkedListPriorityQueue`** as your priority queue for this, you will end up with a well-known sorting algorithm. What sorting algorithm is it?

Implementation Three: Unsorted Doubly-Linked List

An alternative representation of a linked list is a *doubly-linked list*, in which each cell stores two pointers – one to the next cell in the list, and one to the previous cell in the list. That way, given a pointer to any cell, you can advance to the next or previous elements by just following a single pointer.

In the third part of this assignment, your job is to implement the priority queue as an unsorted, doubly-linked list. Whenever a new element is enqueued into the priority queue, you should prepend it to the linked list. For example, if you were to insert the strings

"Marten Reed," "Faye Whitaker," "Dora Bianchi"

into the doubly-linked list priority queue in that order, the priority queue would hold

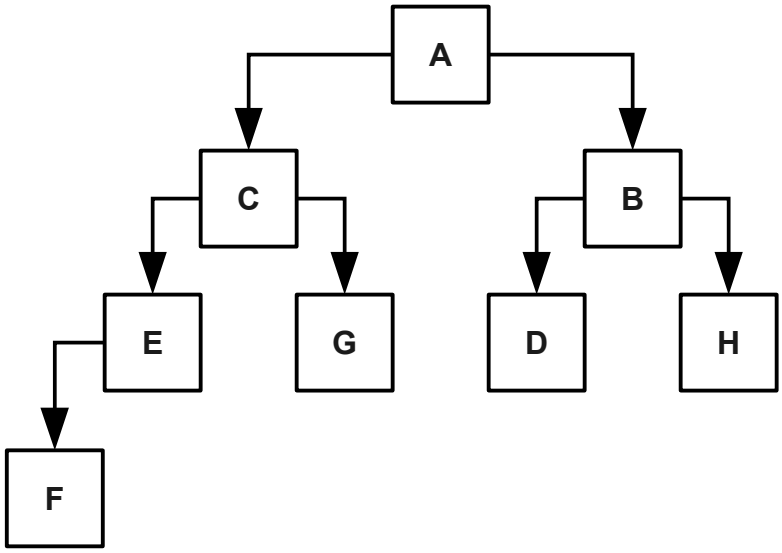
To dequeue the minimum element from the priority queue, you can scan across the linked list to determine where the minimum element is. You then remove that element from the doubly-linked list and return it. Since the list is doubly-linked, you can splice the cell out of the list without having to use a separate pointer to keep track of the previous element. You will probably want to draw some pictures of how exactly you will splice the cell out of the linked list before you attempt to code it up; this step is a bit complex.

The header file **`pqueue-doublylinkedlist.h`** contains the interface for the sorted, doubly-linked list priority queue. You should implement this class by adding a **`private`** section to the interface and implementing the appropriate methods in the **`pqueue-doublylinkedlist.cpp`** source file.

Implementation Four: Binary Heap

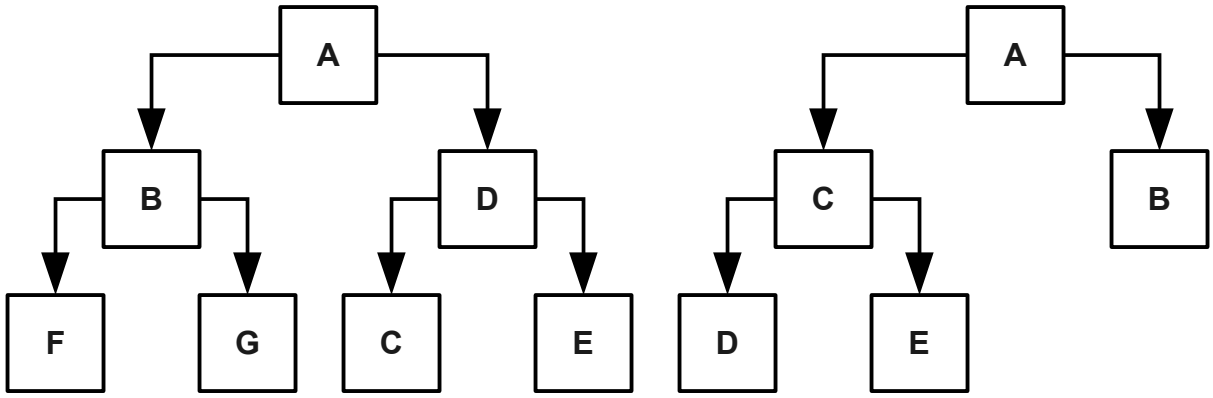
Your final implementation of the priority queue will be backed by a *binary heap*, a fast and clean data structure optimized for use in priority queues.

Binary heaps are best explained by example. Below is a binary heap containing the letters A – H:



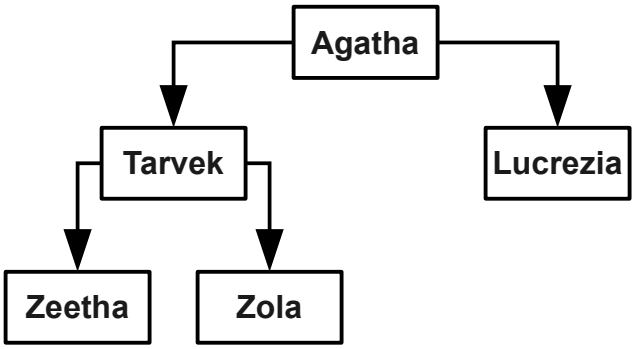
Let's look at the structure of this heap. Each value in the heap is stored in a *node*, and each node has zero, one, or two *child nodes* descending from it. For example, the node A has two children, holding C and B, while the node E has just one child (F) and the node G has no children at all.

In a binary heap, we enforce the rule that every row of the heap, except for the last, must be full. That is, the first row should have one node, the second row two nodes, the third row four nodes, the fourth row eight nodes, etc., up until the last row. You can see this in the above example – the first three rows are all filled in, and only the last row is partially filled. Here are two other examples of binary heaps, each of which obey this rule:

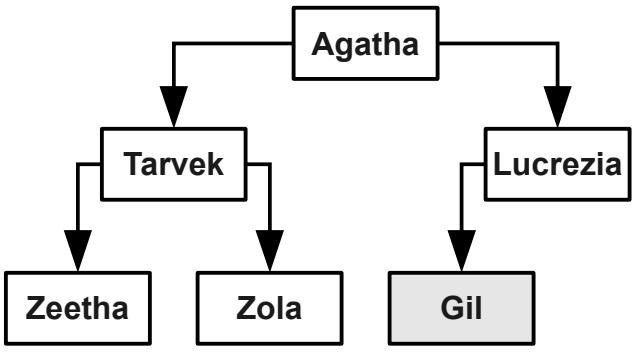


Inside a binary heap, we also enforce one more property – no child node comes lexicographically before its parent. All three of the heaps you've seen so far obey this rule. However, there are no guarantees about how strings can be ordered within a row; as you can see from the examples, within a row the ordering is pretty much arbitrary.

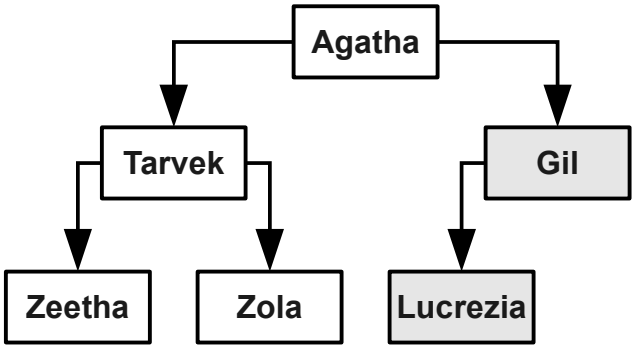
If the elements of a priority queue are stored in a binary heap, it is easy to read off which element is the smallest – it's the one at the top of the heap. It is also efficient to insert an element into a binary heap. Suppose, for example, that we have this binary heap:



Let's add the string "Gil" to this heap. Since a binary heap has all rows except the last filled, the only place we can initially place Gil is in the first available spot in the last row. This is as the left child of Lucrezia, so we place the new node for Gil there:

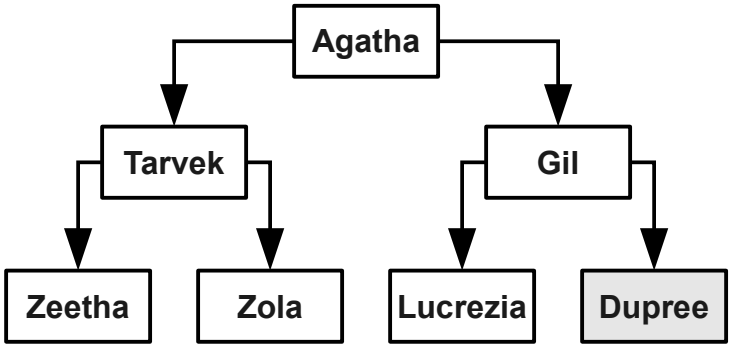


At this point, the binary heap is invalid because Gil lexicographically precedes Lucrezia. To fix this, we run a ***bubble-up*** step and continuously swap Gil with its parent node until it is in its proper place. This means that we exchange Gil and Lucrezia, as shown here:

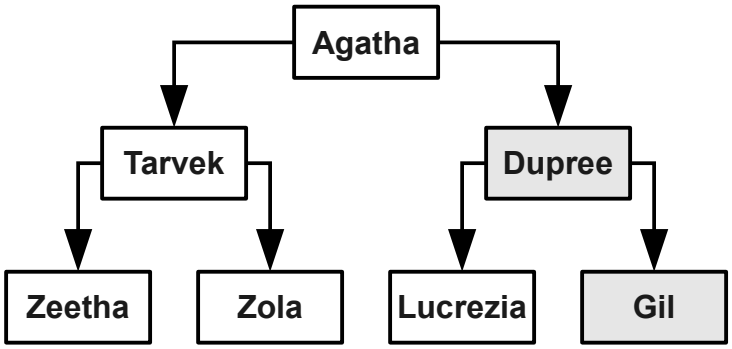


At this point we are done. We now have a binary heap containing all of the original values, plus Gil.

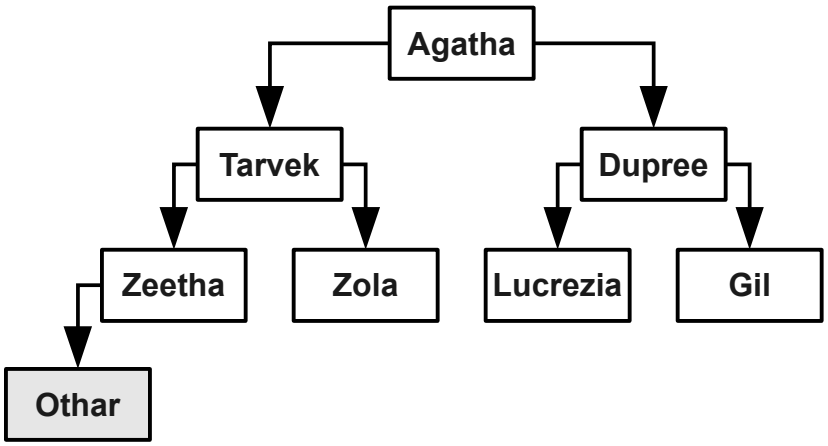
Let's suppose that we now want to insert the string “Dupree” into the heap. We begin by placing it at the next free location in the last row, which is as the right child of Gil:



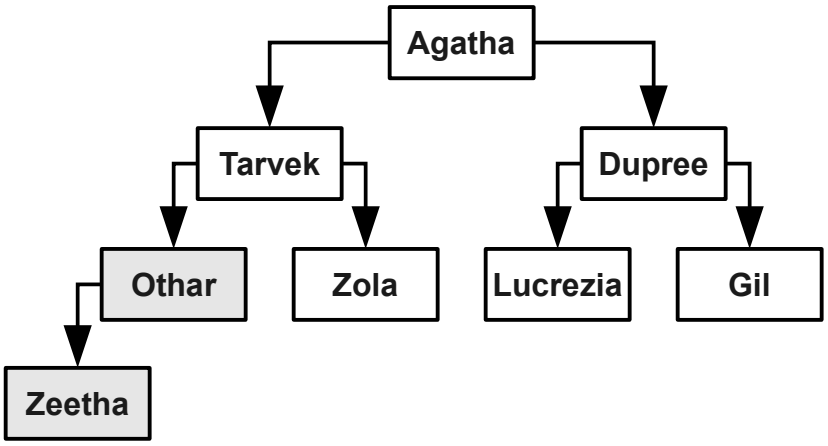
We then bubble Dupree up one level to fix the heap:



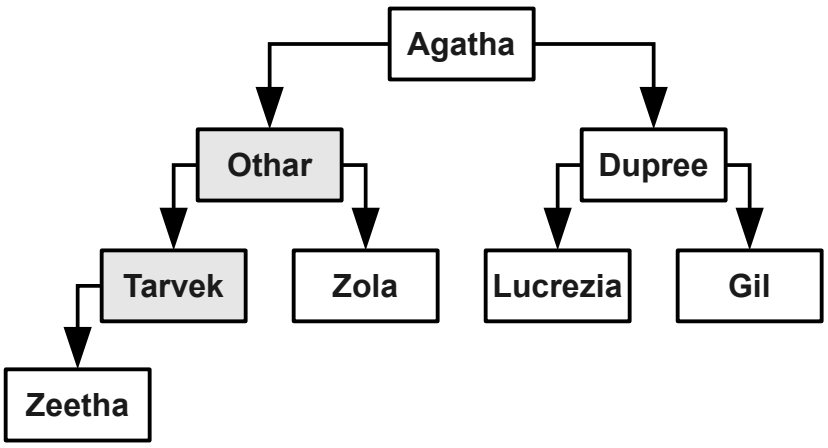
And, again, have a new heap containing these elements. As a final example, suppose that we want to insert “Othar” into this heap. We begin by putting it into the first free spot in the last row, which in this case is as the left child of Zeetha. This is shown here:



We now do a bubble-up step. We first swap Othar and Zeetha to get

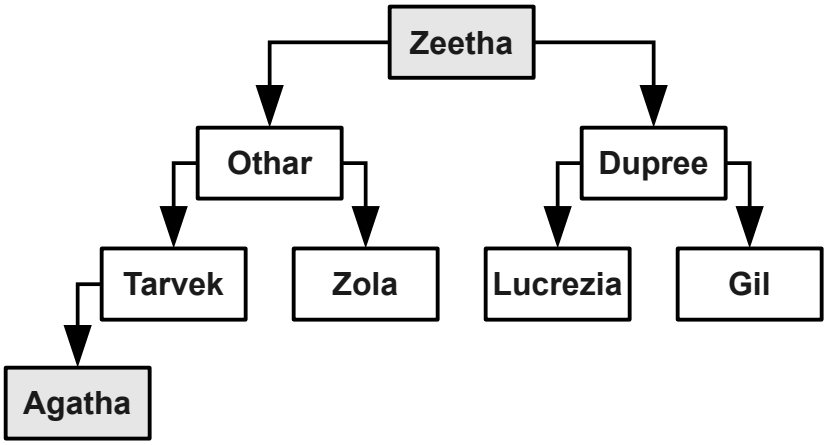


And then swap Othar and Tarvek to get

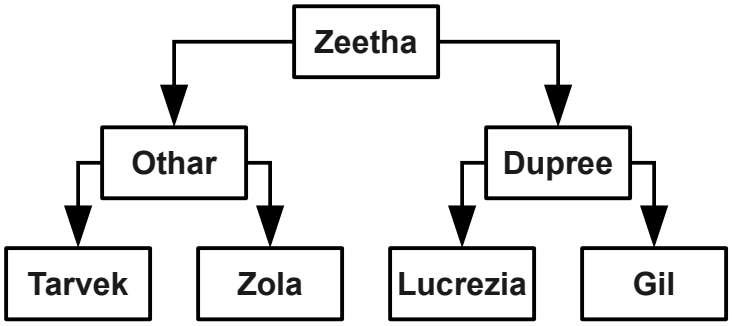


This step runs very quickly. With a bit of math we can show that if there are n nodes in a binary heap, then the height of the heap is at most O(log n), and so we need at most O(log n) swaps to put the new element into its proper place. Thus the enqueue step runs in time O(log n).

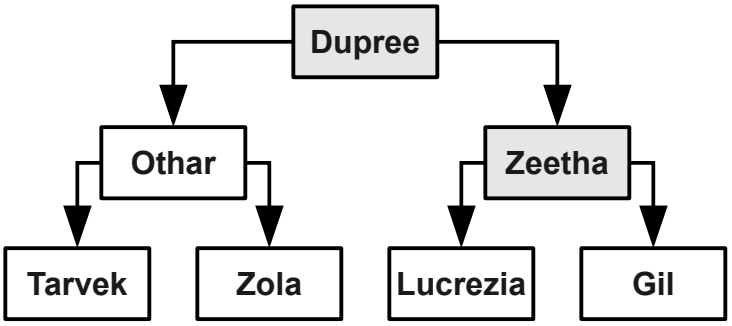
We now know how to insert an element into a binary heap. How do we implement dequeue-min? We know that the minimum element of the binary heap is atop the heap, but we can't just remove it – that would break the heap into two smaller heaps. Instead, we use a more clever algorithm. First, we swap the top of the heap for the very last node in the heap, as shown here:



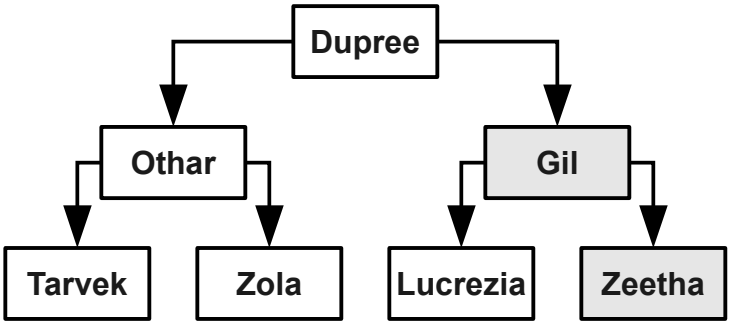
Now, we can remove Agatha from the heap. This leaves:



Unfortunately, what we are left with is not a binary heap because the top element (Zeetha) is one of the lexicographically last values in the heap. To fix this, we will use a ***bubble-down*** step and repeatedly swap Zeetha with its smaller child until it comes to rest. First, we swap Zeetha with Dupree to get this heap:



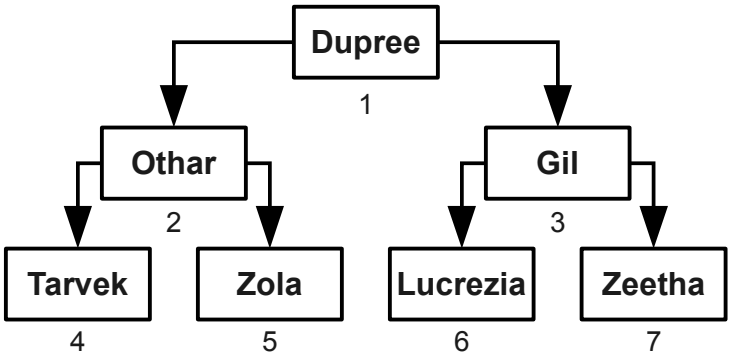
Since Zeetha is not at rest yet, we swap it with the smaller of its two children (Gil) to get this:



And we're done. That was fast! As with enqueue, this step runs in time O(log *n*), because we make at most O(log *n*) swaps. This means that enqueuing *n* elements into a binary heap and then dequeuing them takes time at most O(*n* log *n*). This method of sorting values is called *heapsort*.

How do we represent a binary heap in code? You might think that, like a linked list, we would implement the heap as cells linked together with pointers. This implementation, while possible, is difficult. Instead, we will implement the binary heap using nothing more than a dynamic array.

"An array**?**," you might exclaim.* "How is it possible to store that complicated heap structure inside an array?" The key idea is to number the nodes in the heap from top-to-bottom, left-to-right. For example, we might number the nodes of the previous heap like this:



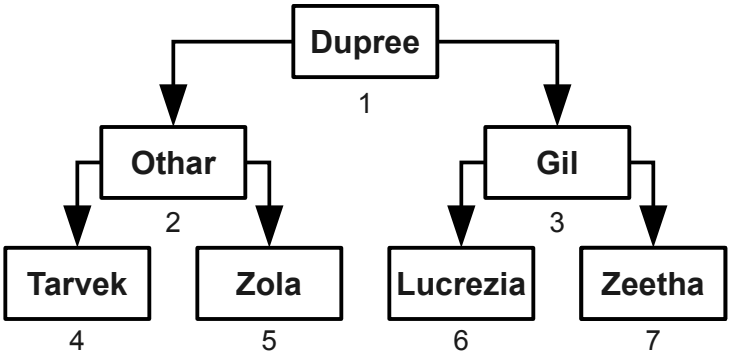
This numbering system has some amazing properties:

- Given a node numbered *n*, its children (if any) are numbered 2*n* and 2*n* + 1.
- Given a node numbered *n*, its parent is numbered *n* / 2, rounded down.

You can check this yourself in the above tree. That's pretty cool, isn't it? The reason that this works is that the heap has a rigid shape – every row must be filled in completely before we start adding any new rows. Without this restriction, our numbering system wouldn't work.

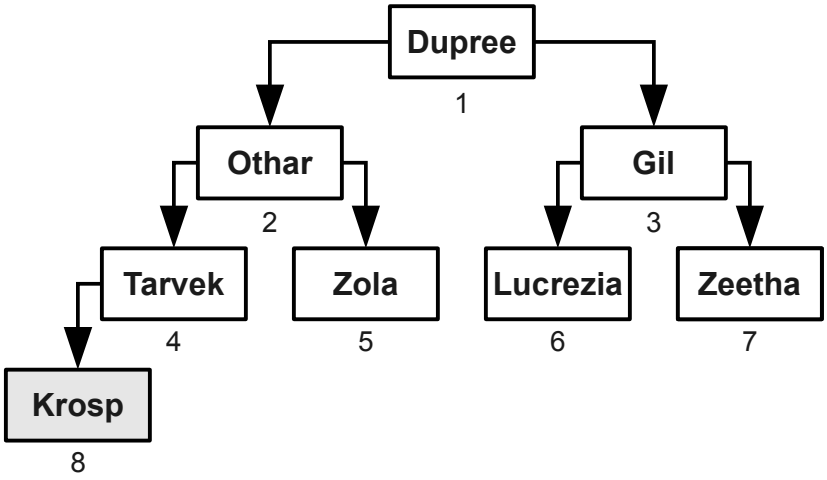
Because all of our algorithms on a binary heap only require us to navigate from parent to child or child to parent, it's possible to represent binary heap using just an array. Each element will be stored at the index given by the above numbering system. Given an element, we can then do simple arithmetic to determine the indices of its parent or its children.

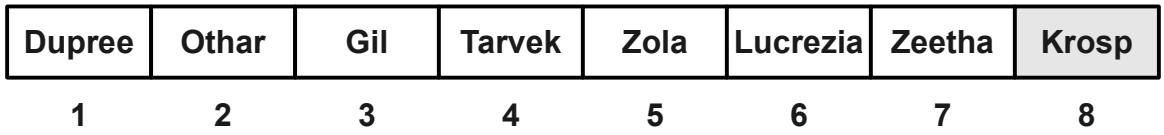
For example, we might encode the above heap using the following array:



* That's an **interrobang** you're looking at – a combination of an exclamation point and a question mark. Crazy, isn't it**?**

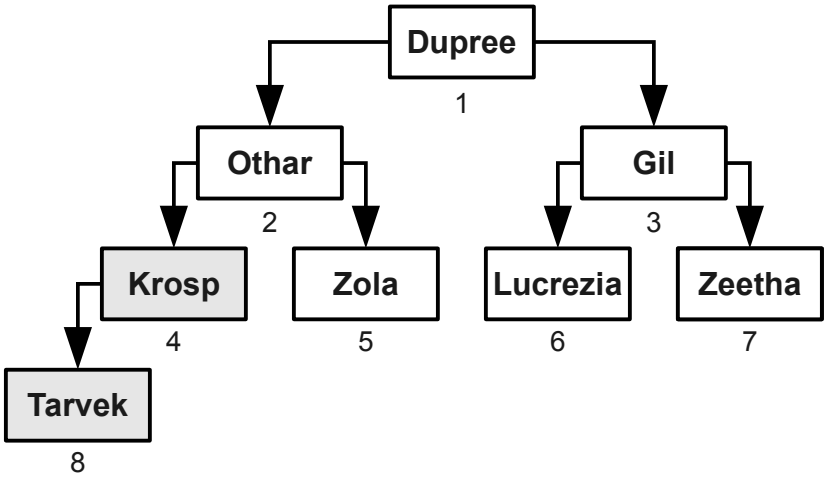
The enqueue and dequeue-min algorithms we have developed for binary heaps translate beautifully into algorithms on the array representation. For example, suppose that we want to insert the string "Krosp" into this binary heap. We begin by adding it into the heap, as shown here:

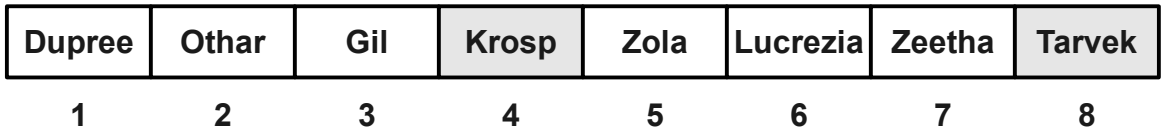




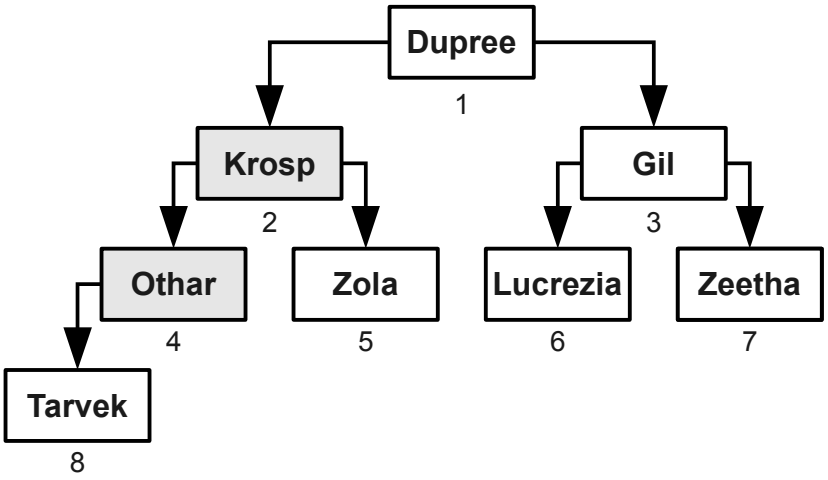
Notice that Krosp is at index 8, which is the last position in the array. This is not a coincidence; whenever you add a node to a binary heap, it always goes at the end of the array.

We then bubble Krosp up into its final position by repeatedly comparing it to its parent. Since Krosp is at position 8, his parent (Tarvek) is at position 4. Since and Krosp precedes Tarvek, we swap them:





Krosp's parent is now at position 2 (Othar), so we swap Krosp and Othar to get the final heap:



As your final implementation task for the assignment, you should implement the **`HeapPriorityQueue`** class in the **`pqueue-heap.h`** and **`pqueue-heap.cpp`** source files. Although in practice you would layer this class on top of the **`Vector`**, for the purposes of this assignment you must do all of your own memory management. This means that you should dynamically allocate and deallocate the underlying array in which your heap is represented.

One word of caution – in our examples, we assumed that the array was one-indexed. Remember that C++ arrays are zero-indexed. When implementing your binary heap, you must make sure to take this into account. There are many ways to do this – perhaps you will have a dummy element at the start of your array, or perhaps you'll adjust the math to use zero-indexing – but be sure that you keep this in mind when designing your implementation.

Extra Credit Opportunity: Build Your Own Priority Queue!

The four heap implementation strategies you will implement in this assignment are only a small sampling of the myriad implementations of priority queues. For extra credit, research one of the following priority queue implementations, then implement the **`ExtraPriorityQueue`** class using one of these more complex (but more efficient!) designs:

- Binomial heap
- Fibonacci heap
- Skew heap
- Leftist heap
- Pairing heap
- Leonardo heap
- Poplar heap
- Splay heap

Advice, Tips, and Tricks

Here are a few pointers that might make your life easier as you work through this assignment:

- **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.
- **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.
- **Test thoroughly.** We have provided you a fairly comprehensive testing system that you can use to verify your code. When you run the provided starter code, you will have the option to manually or automatically test all four of the priority queue implementations (plus the optional extra fifth). The manual tests are good for initial debugging; they allow you to directly issue commands to the priority queue and see what happens. Once you're comfortable that your implementation is mostly correct, you can run the automated tests. The automated tests will subject your priority queue to a battery of tests that will cover a lot of cases. We cannot guarantee that our automatic tests will cover every case, though, and you're strongly encouraged to add your own testing code.

Good luck!