CS106X
Autumn 2014 October 20, 2014

Assignment 4 - PriorityQueue

This assignment focuses on implementing a collection using several internal data structures, using pointers, arrays, dynamic memory allocation, linked lists, and heaps. Turn in the following files:

- `VectorPriorityQueue.h/.cpp` : a priority queue implementation using a Vector for storage
- `LinkedPriorityQueue.h/.cpp` : a priority queue implementation using a linked list internally
- `HeapPriorityQueue.h /.cpp` : a priority queue implementation using a "heap" array internally
- `BinomialHeapQueue.h/.cpp` : a priority queue implementation using a binomial heap internally

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

Assignment Due: Wednesday, October 29th, 2014 at 5:00 pm

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 **four 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.

Priority Queue Implementations:

1) **`VectorPriorityQueue`**: The first priority queue implementation you will write uses an **unsorted Vector** as its internal data storage. This class is only allowed to have a single **private member variable** inside it: your vector. The elements of the vector are ***not* stored in sorted order** internally. As new elements are enqueued, you should simply add them to the end of the vector. When dequeuing, you must then search the vector 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 vector state of a `VectorPriorityQueue` after enqueuing the elements listed on the previous page:

```
index       0       1       2       3       4       5
        +-------+-------+-------+-------+-------+-------+
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 vector.

2) **`LinkedPriorityQueue`**: The linked list implementation is a **doubly linked list** of values, where the values are kept in sorted order (i.e., smallest to largest) to facilitate finding and removing the smallest element quickly. Insertion is a little more work, but made easier because of the decision to maintain both `prev` and `next` pointers. Note: when writing this code, and when debugging this, remember the first rule of dynamic memory structures: DRAW A PICTURE! The only **private member variables** this class is allowed to have inside it are a pointer to the front of your list.

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 = 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       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 `!=`. Do not make assumptions about the lengths of the strings.

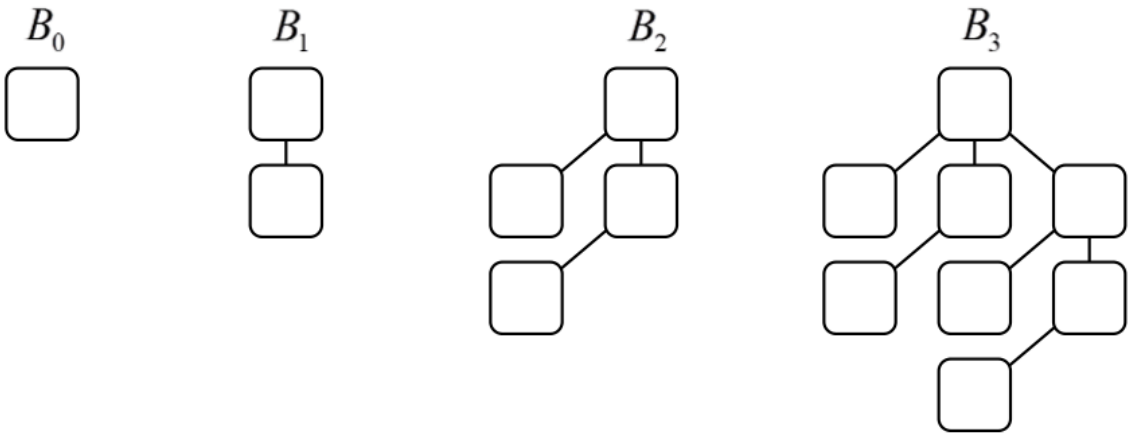
As with the `ArrayList` class written in lecture, when the heap array becomes full and has no more available indexes to store data, you must **resize** it to a larger array. Your larger array should be a multiple of the old array size, such as double the size. You must not **leak memory**; free any dynamically allocated arrays created by your class.

4) **`BinomialHeap:`** The binomial heap expands on the idea of a binary heap by maintaining a collection of binomial trees, each of which respects a property very similar to the heap order property discussed for Implementation #3. For this implementation, you are only allowed to store a data structure containing the pointers to binomial trees (explained in the section below) and the number of elements in the heap as **private member variables**.

A binomial tree of order k (where k is a positive integer) is recursively defined:

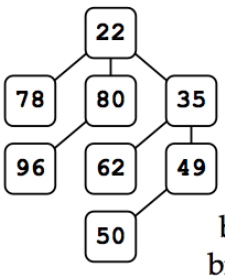
- A binomial tree of order 0 is a single node with no children.
- A binomial tree of order k is a single node at the root with k children, indexed 0 through k – 1. The 0^{th} child is a binomial tree of order 0, the 1^{st} child is a binomial tree of order 1, ..., the m^{th} child is a binomial tree of order m, and the k – 1^{st} child is a binomial tree of order k - 1.

Some binomial trees:

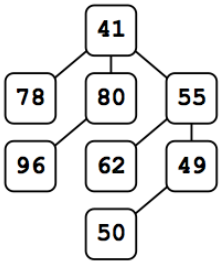


One property to note—and one that will certainly be exploited in the coming paragraphs, is that one can assemble a binomial tree of order k + 1 out of two order k trees by simply appending the second to the end of the first's list of children. A related property: each binomial tree of order k has a total of 2^k nodes.

A binomial heap of order k is a binomial tree of order k, where the heap property is recursively respected throughout—that is, the value in each node is lexicographically less than or equal to those held by its children. In a world where binomial trees store just numeric strings, the binomial tree on the left is also a binomial heap, whereas the one on the right is not (because the `"55"` is alphabetically greater than the `"49"`):



binomial tree: yes
binomial heap: yes!



binomial tree: yes
binomial heap: no!

Now, if binary heaps can back priority queues, then so can binomial heaps. But the number of elements held by a priority queue can't be constrained to be some power of 2 all the time. So priority queues, when backed by binomial heaps, are really backed by a `Vector` of binomial heaps.

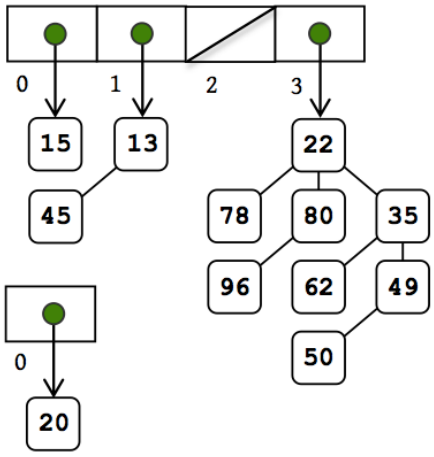
If a priority queue needs to manage 11 elements, then it would hold on to three binomial heaps of orders 0, 1, and 3 to store the $2^0 + 2^1 + 2^3 = 1 + 2 + 8 = 11$ elements. The fact that the binary representation of 11 is 1011 isn't a coincidence. The 1's in 1011 contribute 2^3, 2^1, and 2^0 to the overall number. Those three exponents tell us what binomial heaps orders are needed to accommodate all 11 elements. Neat!

Binomial Heap Insert

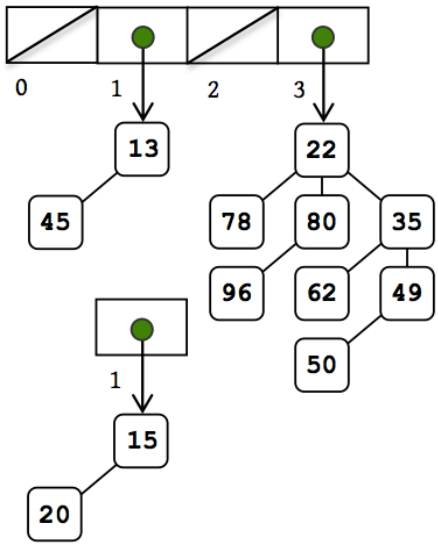
What happens when we introduce a new element into the mix? More formally: What happens when you enqueue a new string into the `Vector<BinomialHeapNode *>`-backed priority queue? Let's see

what happens when we enqueue a `"20"`.

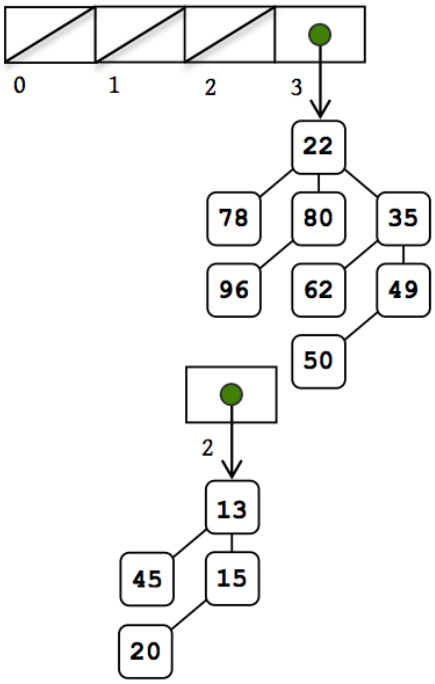
The size of the priority queue goes from 11 to 12—or rather, from 1011 to 1100. We'll understand how to add this new element, all the time maintaining the heap ordering property within each binomial tree, if we emulate binary addition as closely as possible. That emulation begins by creating binomial tree of order 0 around the new element—a `"20"` in this example—and align it with the 0^{th} order entry of the `Vector<BinomialHeapNode *>`.



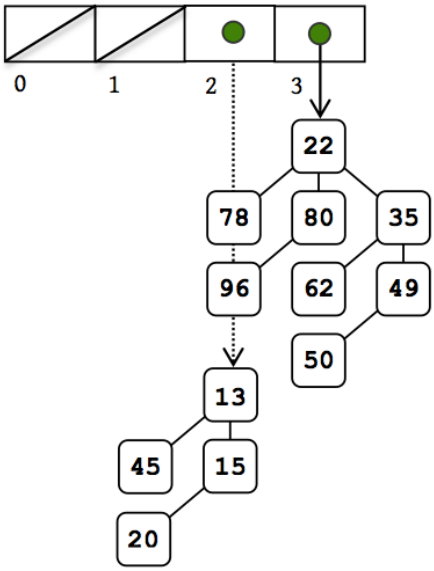
Now, when we add 1 and 1 in binary, we get 0, and carry a 1, right? We do the same thing when merging two order-0 binomial heaps, order-0 plus order-0 equal `NULL`, carry the order-1. One key difference: when you merge two order-0 heaps into the order-1 that gets carried, you need to make sure the heap property is respected by the merge. Since the 15 is smaller than the 20, that means the 15 gets an order-0 as a child, and that 15 becomes the root of the order-1.



The carry now contributes to the merging at the order-1 level. The carry (with the 15 and the 20) and the original order-1 contribution (the one with the 13 and the 45) similarly merge to produce a `NULL` order-1 with an order-2 carry.



Had there been an order-2 in the original figure, the cascade of merges would have continued. But because there's no order-2 binomial heap in the original, the order-2 carry can assume that position in the collection and the cascade can end. In our example, the original binomial heap collection would be transformed into:

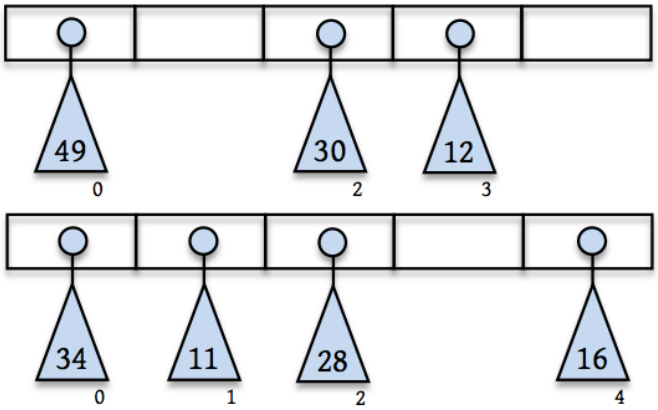


Binomial Heap Merge

The primary perk the binomial heap has over the more standard binary heap is that it, if properly implemented—supports merge much more quickly. In fact, two binomial heaps as described above can be merged in O(lg n) time, where n is the size of the larger binomial heap.

You can merge two heaps using an extension of the binary addition emulated while discussion enqueue. As it turns out, enqueuing a single node is really the same as merging an arbitrarily large

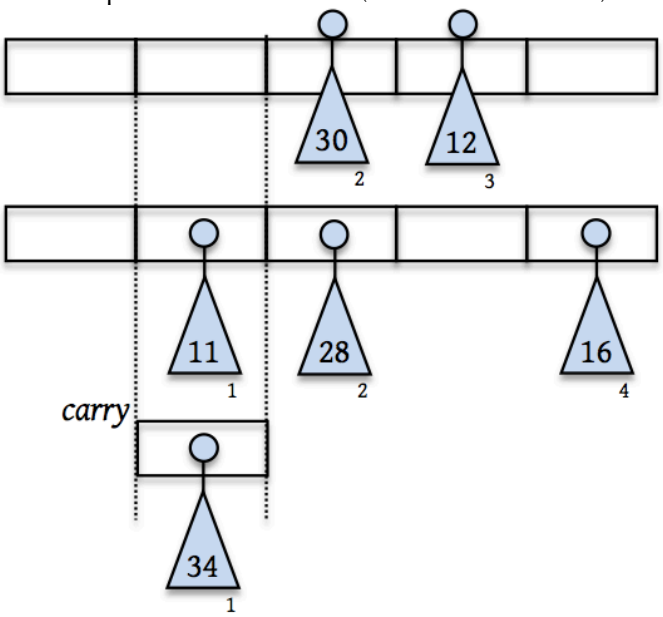
binomial heap with a binomial heap of size 1. The generic merge problem is concerned with the unification of two binomial heaps of arbitrary sizes. So, for the purposes of illustration, assuming we want to merge two binomial heaps of size 13 and 23, represented below:



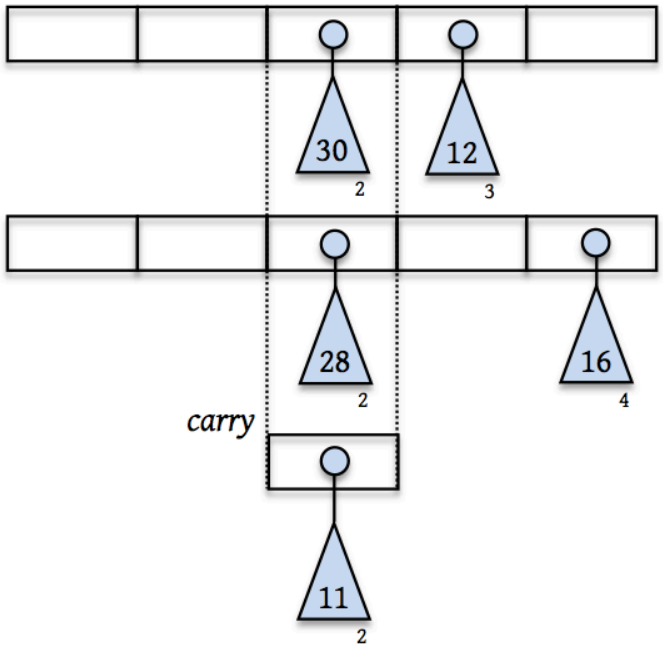
The triangles represent binomial trees respecting the heap ordering properties, and the subscripts represent their order. The numbers within the triangles are the root values—the smallest in the tree, and the blanks represent `NULL`. (We don't have space for the more elaborate pictures used to illustrate enqueue, so I'm going with more skeletal but I'm hoping equally helpful pictures.)

To merge is to emulate binary arithmetic, understanding that the 0s and 1s of pure binary math have been upgraded to be `NULL`s and binomial tree root addresses. The merge begins with any order-0 trees, and then ripples from low to high order—left to right in the diagram. This particular merge (which pictorially merges the second into the first) can be animated play-by- play as:

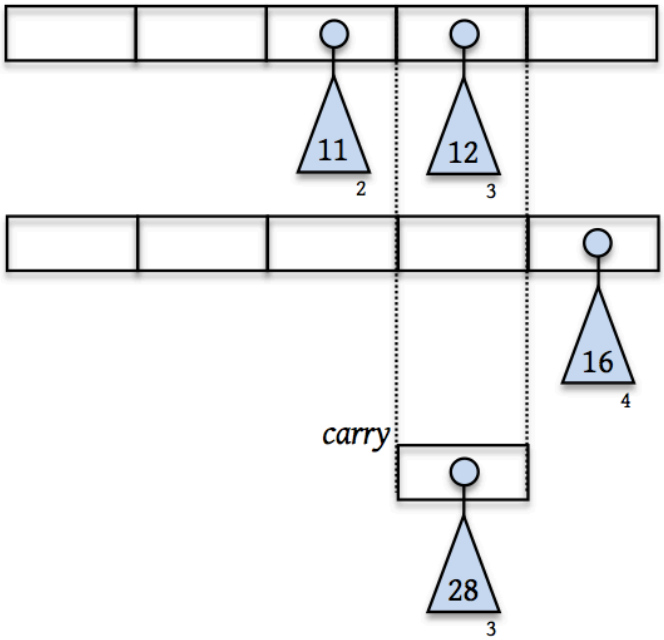
1. Merge the two order-0 trees to produce an order-1 (with 34 at the root) that carries.



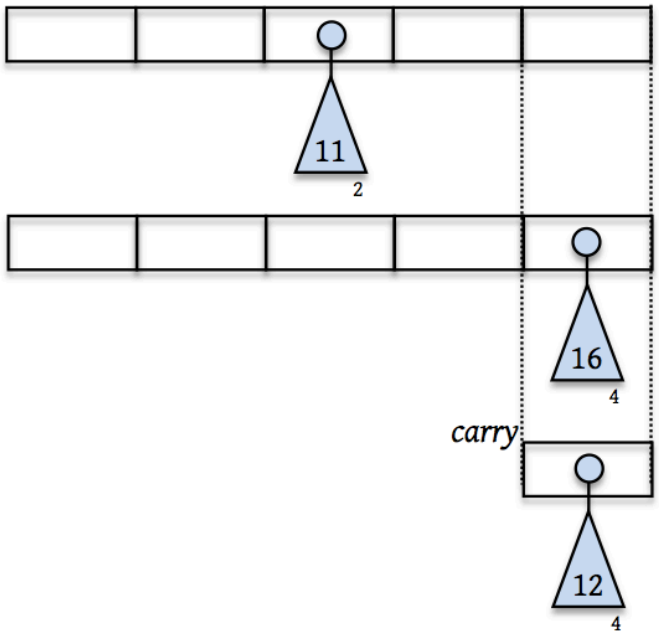
2. Merge the two order-1 trees to produce an order-2 tree carry, with 11 at the root.



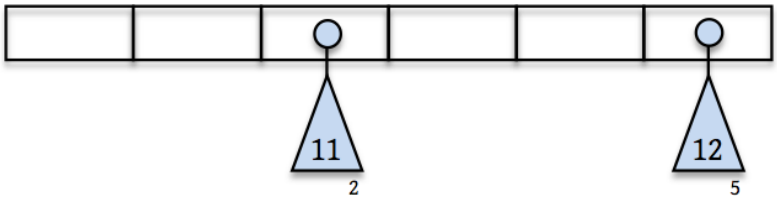
3. Merge the three (three!) order-2 trees! Leave one of the three alone (we'll leave the 11 in place, though it could have been any of the three) and merge the other two to produce an order-3 (with the smaller of 28 and 30 as the root).



4. Merge the two order-3 trees to produce an order-4 tree carry, with 12 as the root:



5. Finally, merge the two order-4s to materialize an order-5 tree with the 12 at the root. Because this is clearly the last merge, we'll draw just one final picture.



The above reflects the fact that the merged product should have 13 + 23 equals 36_{10} equals 100100_2 elements, and it indeed does: The order-2 tree houses 4 elements, and the order-5 houses 32.

Binomial Heap `peek` and `extractMin`

`peek` can be implemented as a simple for loop over all of the binomial heaps in the collection, and returning the smallest of all of the root elements it encounters (and it runs in O(lg n) time). `extractMin` runs like `peek` does, except that it physically removes the smallest element before returning it. Of course, the binomial heap that houses the smallest element must make sure all of its children are properly reincorporated into the data structure without being orphaned. Each of those children can be merged into the remaining structure in much the same way a second binomial heap is, as illustrated above.

Binomial Heap Implementation Notes

Think before you code: We said the same thing about the binary heap, but it's even more important here. You can't fake an understanding of binomial heaps and code, hoping it'll all just kind of work out. You'll only succeed with this final implementation if have a crystal clear picture of how `enqueue`, `merge`, and `extractMin` all work, and you write code that's consistent with that understanding. In particular, you absolutely must understand the general merge operation described above before you tackle any of operations that update the binomial heap itself.

Use a combination of built-ins and custom structures: Each node in a binomial heap should be modeled using a data structure that looks something like this:

```
struct BinomialHeapNode {
    PQEntry entry;
    Vector<BinomialHeapNode *> children;
};
```

As opposed to the binary heap, the binomial heap—at least the first time you implement it—is sophisticated enough that you'll want to rely on sensibly chosen built-ins and layer on top of those. You're encouraged to use the above node type for your implementation, and only deviate from it if you have a compelling reason to do so.

Freeing memory: You are responsible for freeing heap-allocated memory. Your implementation should not orphan any memory during its operations and the destructor should free all of the internal memory for the object. As opposed to before, it's probably best for you to free memory as you go along, since the memory management issues for this version of more elaborate, and going back and patching up memory problems will be more difficult.

Priority Queue Operations:

Each of your 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 raise an `error` using the `error(...)` function 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 raise an `error` 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 raise an `error` if the queue does not contain any elements.
`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.

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 inside 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 `VectorPriorityQueue`'s operations like `enqueue` and `dequeue` should not needlessly rearrange the elements of the vector any more than necessary.
- 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.
- 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 `ListNode`s as you like.

Development Strategy and Hints:

- *Order of Implementation:* While you are free to implement the priority queues in any order you wish, we strongly recommend that you implement them in the order we specified: Vector, then Linked, then Heap, then Binomial Heap. This goes from simplest to most difficult.
- *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.
- *Testing, testing, testing:* Make sure to 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.

Style Guidelines:

In general, items mentioned in the "Implementation and Grading" from the previous assignment(s) specs also apply here. Please refer to those documents as needed. Note the instructions in the previous assignments about procedural decomposition, variables, types, parameters, value vs. reference, and commenting. Don't forget to **cite any sources** you used in your comments. Refer to the course **Style Guide** for a more thorough discussion of good coding style.

Commenting: Since all of the queue classes have the same public members, we will allow you to comment the public member functions (`enqueue`, `dequeue`, etc.) a single time in `VectorPriorityQueue`, and then in the other classes you can simply write, "See `VectorPriorityQueue.h` for documentation member functions." But we do expect you to put a descriptive comment header on each queue class file and explain that implementation and its pros and cons. **Also put a comment atop every member function that states its Big-Oh**. For example, in a `HeapPriorityQueue` the `peek` operation runs in constant time, so you should put a comment on that function that says "O(1)."

Redundancy: Redundancy is another major grading focus; avoid repeated logic as much as possible. Your classes will be graded on whether you make good choices about what members it should have, and other factors such as which are `public` vs. `private`, and `const`-correctness, and so on. You may find that there are some operations that you have to repeat in all of your classes, like checking whether a queue is empty before dequeuing from it. We do not require you to reduce redundancy across multiple classes; but we do expect you to remove redundancy within a single class. If one implementation has a common operation, make a **private helper** function and call it multiple times in that file.

Please remember to follow the **Honor Code** on this assignment. Submit your own work; do not look at others' solutions. Cite sources. Do not give out your solution; do not place a solution on a public web site or forum. NOTE: The Stanford C++ library includes a priority queue implementation that is somewhat similar to the `HeapPriorityQueue` you are supposed to implement. For this assignment, we ask that you do not look at that file's source code because it is too similar to what you are asked to do here. You must solve the problem yourself.

Possible Extra Features:

A good extra would be to write a PQ implementation(s) beyond those required by this assignment, such as:

- **Sorted Unfilled Array:** Similar to vector PQ, but use a growable array and store elements in sorted order.
- **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.

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

- **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.
- **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.
- **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.
- **Other:** 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.