CS106B Handout 29
Autumn 2012 November 2nd, 2012

Assignment 5: Priority Queue

Assignment idea, handout, and binary heap implementation by Julie Zelenski.
Binomial heap extension by Jerry.

So, it's finally time for you to implement a class of your own: a priority queue, which is a variation on the standard queue described in the reader. The standard queue is a collection of elements managed in a first-in, first-out manner. The first element added to the collection is always the first element extracted; the second is second; so on and so on.

In some cases, a FIFO strategy may be too simplistic for the activity being modeled. A hospital emergency room, for example, needs to schedule patients according to priority. A patient who arrives with a more serious problem should preempt others even if they have been waiting longer. This is a *priority queue*, where elements are added to the queue in arbitrary order, but when time comes to extract the next element, it is the highest priority element in the queue that is removed. Such an object would be useful in a variety of situations. In fact, you can even use a priority queue to implement sorting; insert all the values into a priority queue and extract them one by one to get them in sorted order.

The main focus of this assignment is to implement a priority queue class in several different ways. You'll have a chance to experiment with arrays and linked structures, and in doing so you'll hopefully master the pointer gymnastics that go along with it.

Due: Monday, November 12th at 3:00 p.m.

The `PQueue` Interface

The priority queue will be a collection of strings. Lexicographically smaller strings should be considered **higher** priority than lexicographically larger ones, so that **`"ping"`** is higher priority than **`"pong"`**, regardless of insertion order.

Here are the methods that make up the **`public`** interface of all priority queues:

```
class PQueue {
public:
   void enqueue(const string& elem);
   std::string extractMin();
   std::string peek();
   static PQueue *merge(PQueue *one, PQueue *two);
private:
   // implementation dependent member variables and helper methods
};
```

`enqueue` is used to insert a new element to the priority queue. **`extractMin`** returns the value of highest priority (i.e., lexicographically smallest) element in the queue and removes it. **`merge`** destructively unifies the incoming queues and returns their union as a new queue. For the detailed descriptions on how these methods behave, see the **`pqueue.h`** interface file included in the starter files.

Implementing the priority queue

While the external representation may give the illusion that we store everything in sorted order behind the scenes at all time, the truth is that we have a good amount of flexibility on what we choose for an internal representation. Sure, all of the operations need to work properly, and we want them to be fast. But we might optimize not for speed but for ease of implementation, or to minimize memory footprint. Maybe we optimize for one operation at the expense of others.

This assignment is all about client expectations, implementation and internal representation. Yes, you'll master arrays and linked lists in the process, but the takeaway point of the assignment—or the most important of the many takeaway points—is that you can use whatever machinery you deem appropriate to manage the internals of a new, object-oriented container.

You'll implement the priority queue in three (or four if you opt for the extra credit) different ways. Two are fairly straightforward, but the third is nontrivial, and fourth is very challenging (although so neat and clever and beautiful that it's worth the struggle.)

Implementation 1: Optimized for simplicity and for the **`enqueue`** method by backing your priority queue by an unsorted **`Vector<string>`**. **`merge`** is pretty straightforward, but **`peek`** and **`extractMin`** are expensive, but the expense might be worth it in cases where you need to get a version up and running really quickly for a prototype, or a proof of concept, or perhaps because you need to **`enqueue`** 50,000 elements and extract a mere 50. I don't provide much in terms of details on this one, as it's pretty straightforward.

Implementation 2: Optimized for simplicity and for the **`extractMin`** operation by maintaining a sorted doubly linked (**`next`** and **`prev`** pointers required) list of strings behind the scenes. **`peek`** and **`extractMin`** will run super fast, but **`enqueue`** will be slow, because it needs to walk the list from front to back to find the insertion point (and that takes time that's linear in the size of the priority queue itself. **`merge`** can (and should) be implemented to run in linear time, for much the same reason Merge from merge sort can be.

Implementation 3: Balance insertion and extraction times by implementing your priority queue in terms of a binary heap, discussed in detail below. When properly implemented, **`peek`** runs in O(1) time, **`enqueue`** and **`extractMin`** each run in O(lg n) time, and **`merge`** runs in O(n) time.

Optional
Implementation 4: **`enqueue`**, **`extractMin`**, and **`merge`** all run in O(lg n), but only because we use a binomial heap behind the scenes. The binomial heap is by far the most intense of the four data structures used in Assignment 4, and it's completely optional. Only tackle this if you're up for a challenge and would like some extra credit.

Implementation 1: Unsorted `Vector`

This implementation is layered on top of our **`Vector`** class. **`enqueue`** simply appends the new element to the end. When it comes time to **`extractMin`** the minimum element (i.e. the one with the highest priority in our version), it performs a linear search to find it. This implementation is straightforward as far as layered abstractions go, and serves more as an introduction to the architecture of the assignment than it does as an interesting implementation. Do this one first.

Aside

As you implement each of the subclasses, you'll leave **`pqueue.h`** and **`pqueue.cpp`** alone, and instead modify the interface (**`.h`**) and implementation (**`.cpp`**) files for each of the subclasses. In the case of the unsorted **`Vector`** version, you'll be concerned with **`pqueue-vector.h`** and **`pqueue-vector.cpp`**. **`pqueue-vector.h`** defines the public interface you're implementing, but its **`private`** section is empty:

```
class VectorPQueue : public PQueue {
public:
   VectorPQueue();
   ~VectorPQueue();

   static VectorPQueue *merge(VectorPQueue *one, VectorPQueue *two);

   void enqueue(const std::string& elem);
   std::string extractMin();
   std::string peek();

private:
   // update the private section with the list of
   // data members and helper methods needed to implement
   // the vector-backed version of the PQueue.
};
```

Not surprisingly, the **`private`** section shouldn't be empty, but instead should list the items that comprise your internal representation. You should erase the comment I've provided and insert the list of data members and private helper functions you think should be there.

The **`pqueue-vector.cpp`** file provides dummy, placeholder implementations of everything, just so that the project cleanly compiles. In a few cases, the dummy implementations actually do the right thing, but a large majority of them need updates to include real code that does real stuff.

Note that the parent **`PQueue`** class defines a **`protected`** field called **`logSize`**, which means you have access to it. You should ensure that **`logSize`** is always maintained to house the logical size of the priority queue—both here and in the other two (or three, if you do the extra credit) implementations. By unifying the **`logSize`** field to the parent class, we can implement **`size`** and **`isEmpty`** at the **`PQueue`** class level (I already did) and they work automatically for all subclasses.

As you advance through the implementations, understand that you'll be modifying different pairs of interface and implementation files (**`pqueue-heap.h`** and **`pqueue-heap.cpp`** for the binary heap version, etc). In all cases, the **`private`** sections of the interface are empty and need to be fixed, and in all cases the implementation files have placeholder implementations to sedate the compiler into complacency.

Implementation 2: Sorted doubly-linked list

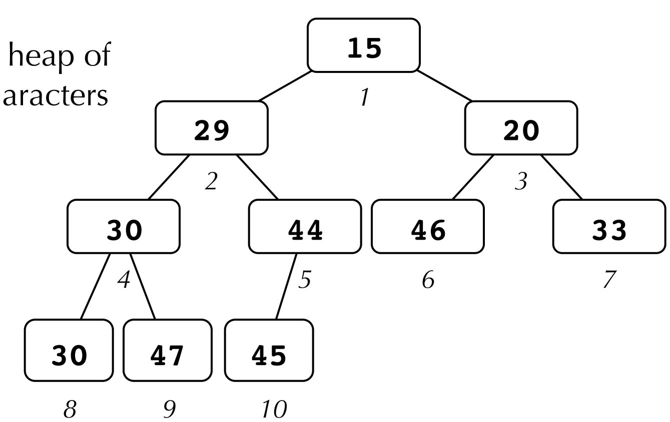
The linked list implementation is a doubly linked list of values, where the values are kept in sorted order (i.e., smallest to largest) to facilitate finding and removing the smallest element quickly. Insertion is a little more work, but made easier because of the decision to maintain both **`prev`** and **`next`** pointers. **`merge`** is conceptually simple, although the implementation can be tricky for those just learning pointers and linked lists for the first time.

Implementation 3: Binary Heap

Although the binary search trees we'll eventually discuss in lecture might make a good implementation of a priority queue, there is another type of binary tree that is an even better choice in this case. A *heap* is a binary tree that has these two properties:

- It is a *complete* binary tree, i.e. one that is full in all levels (all nodes have two children), except for possibly the bottom-most level which is filled in from left to right with no gaps.
- The value of each node is less than or equal to the value of its children.

Here's a conceptual picture of a small heap of integer strings (i.e. strings where all characters are digits)

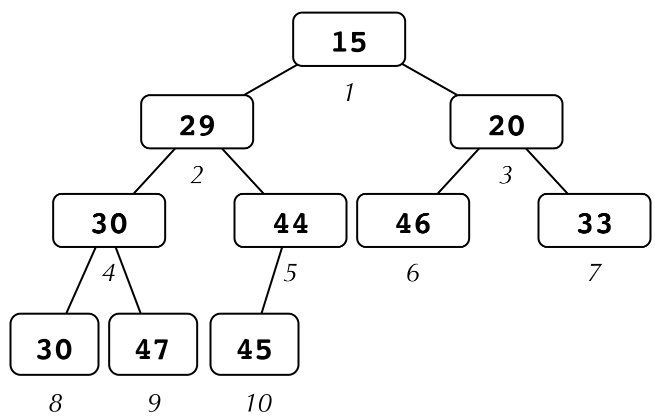


Note that a heap differs from a binary search tree in two significant ways. First, while a binary search tree keeps all the nodes in a sorted arrangement, a heap is ordered in a much weaker sense. Conveniently, the manner in which a heap is ordered is actually enough for the efficient performance of the standard priority queue operations. The second important difference is that while binary search trees come in many different shapes, a heap must be a complete binary tree, which means that every heap containing ten elements is the same shape as every other heap of ten elements.

Representing a heap using an array

One way to manage a heap would be to use a standard binary tree node definition and wire up left and right children pointers to all nodes. We can exploit the completeness of the tree and create a simple array representation and avoid the pointers.

Consider the nodes in the heap to be numbered level by level like this:



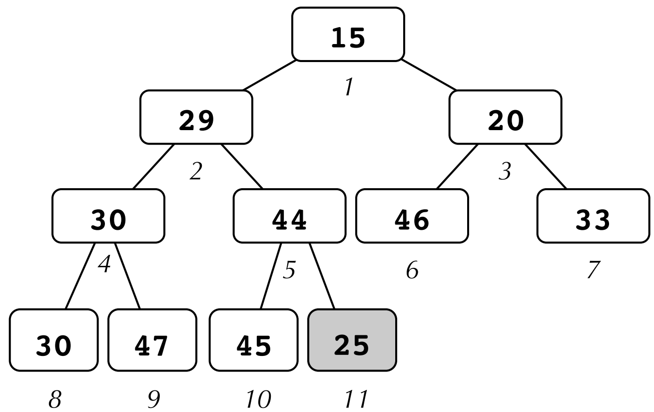
and now check out this array representation of the same heap:

15	29	20	30	44	46	33	30	47	45
1	2	3	4	5	6	7	8	9	10

You can divide any node number by 2 (discarding the remainder) to get the node number of its parent. For example, the parent of node 9 is node 4. The two children of node *i* are 2*i* and 2*i* + 1, e.g. node 3's two children are 6 and 7. Although many of the drawings in this handout use a tree diagram for the heap, keep in mind you will actually be representing the heap as a string array.

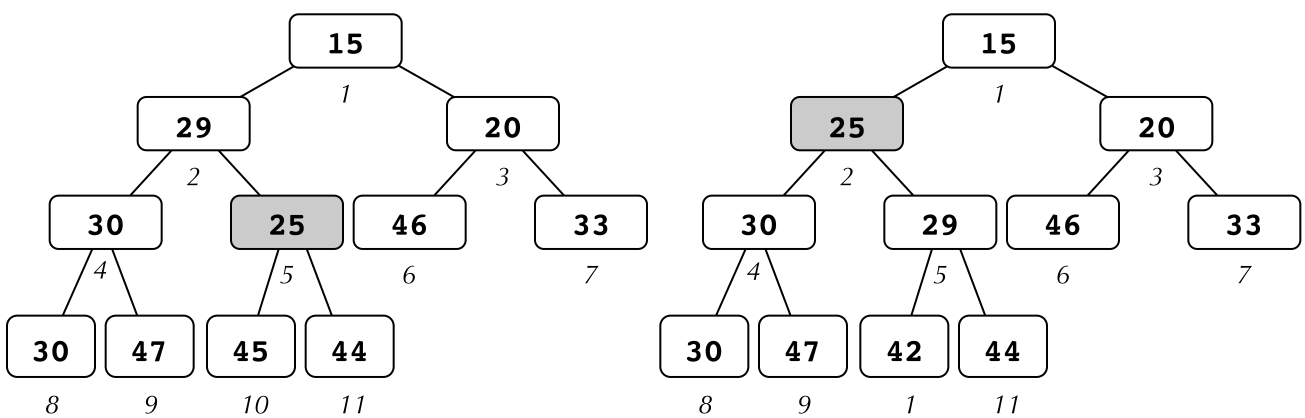
Heap insert

Inserting into a heap is done differently than its functional counterpart in a binary search tree. A new element is added to the very bottom of the heap and it rises up to its proper place. Suppose, for example, we want to insert **`"25"`** into our heap. We add a new node at the bottom of the heap (the insertion position is equal to the size of the heap):



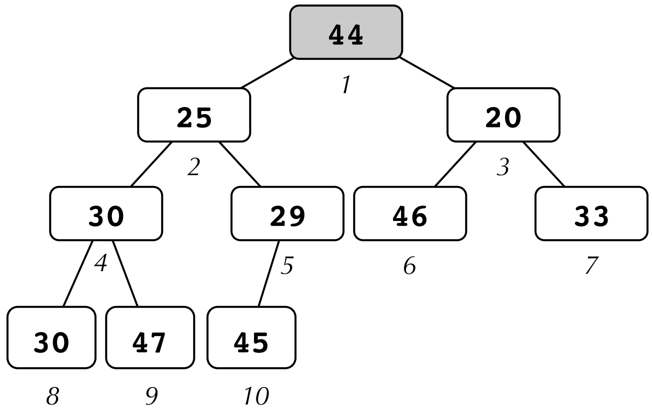
15	29	20	30	44	46	33	30	47	45	25
1	2	3	4	5	6	7	8	9	10	11

We compare the value in this new node with the value of its parent and, if necessary, exchange them. Since our heap is actually laid out in an array, we "exchange" the nodes by swapping array values. From there, we compare the moved value to its new parent and continue moving the value upward until it needs to go no further. This is sometimes called the *bubble-up* operation.

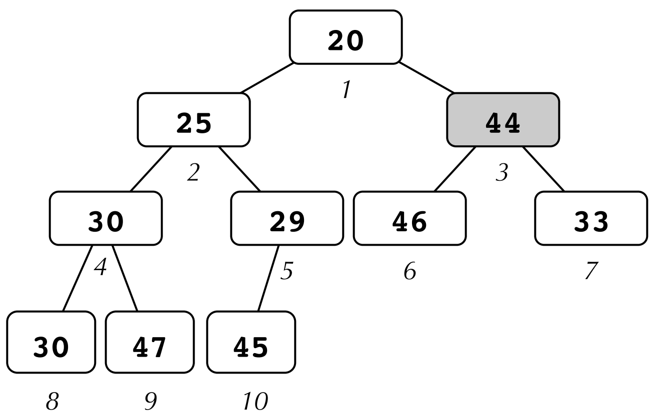


Heap `extractMin`

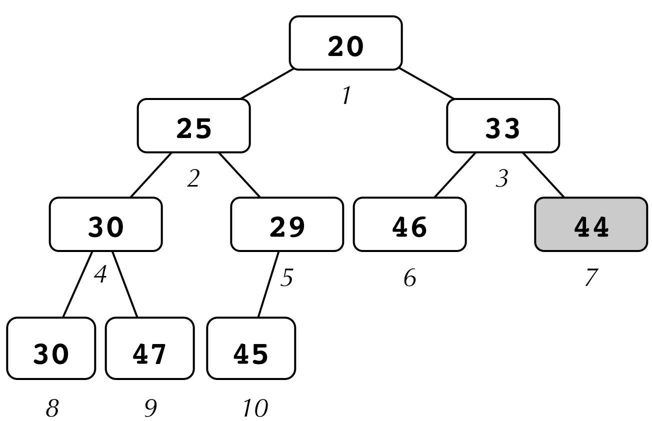
Where is the smallest value in the heap? Given heap ordering, the smallest value resides in the root, where it can be easily accessed. However, after removing this value, you generally need to rearrange the nodes that remain. Remember the completeness property dictates the shape of the heap, and thus it is the bottommost node that needs to be removed from the structure. Rather than re-arranging everything to fill in the gap left by the root node, we can leave the root node where it is, copy the **value** from the last node to the root node, and remove the last node.



We have a complete tree again, and the left and right sub-trees are heaps. The only potential problem: a violation of the heap ordering property, localized at the root. In order to restore the min-heap property, we need to trickle that value down to the right place. We will use an inverse to the strategy that allows us to float up the new value during the `enqueue` operation. Start by comparing the value in the root to the values of its two children. If the root's value is larger than the values of either child, swap the value in the root with that of the smaller child:



This step fixes the heap ordering property for the root node, but at the expense of possibly breaking the sub-tree of the child we switched with. The sub-tree is now another heap where only the root node is out of order, so we can iteratively apply the same re-ordering on the sub-tree to fix it up and so-on down through its sub-trees.



You stop trickling downwards when the value sinks to a level such that it is smaller than both of its children or it has no children at all. This recursive action of moving the out-of-place root value down to its proper place is often called *heapify*-ing.

You should implement the priority queue as a heap array using the strategy shown above. This array should start small and grow dynamically as needed.

Merging two heaps

The merge operation—that is, destroying two heaps and creating a new one that's the logical union of the two originals—can be accomplished via the same heapify operation discussed above. Yes, you could just insert elements from the second into the first, one at a time, until the second is depleted and the first has everything. But it's actually faster—asymptotically so, in fact—to do the following:

- Create an array that's the logical concatenation of the first heap's array and the second heap's array, without regard for the heap ordering property. The result is a complete, array-backed binary tree that in all likelihood isn't even close to being a heap.
- Recognize that all of the leaf nodes—taken in isolation—respect the heap property.
- Heapify all sub-heaps rooted at the parents of all the leaves.
- Heapify all sub-heaps rooted at the grandparents of all the leaves.
- Continue heapify increasingly higher ancestors until you reach the root, and heapify that as well.

Binary Heap Implementation Notes

<u>*Manage your own raw memory*</u>. It's tempting to just use a **`Vector<string>`** to manage the array of elements. But using the vector introduces an extra layer of code in between your **`HeapPQueue`** and the memory that actually store the elements, and in practice, a core container class like the **`HeapPQueue`** would be implemented without that extra layer. Make it a point to implement your **`HeapPQueue`** in terms of raw, dynamically allocation arrays of **`string`**s instead of a **`Vector<string>`**.

<u>*Freeing memory*</u>. 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. We recommend getting the entire data structure working without deleting anything, and then going back and taking care of it.

<u>*Think before you code*</u>. The amount of code necessary to complete the implementation work is not large, but you will find it requires a bit of thinking getting it to work correctly. It will help to sketch things on paper and work through the boundary cases carefully before you write any code.

Test thoroughly. I know we've already said this, but it never hurts to repeat it a few times. You don't want to be surprised when our grading process finds a bunch of lurking problems that you didn't discover because of inadequate testing. The code you write has some complex interactions and it is essential that you take time to identify and test all the various cases. I've provided you with a minimal test harness to ensure that the **`HeapPQueue`** works in simple scenarios, but it's your job to play villain and try to break your implementation, knowing that you're done when you fail to do so.

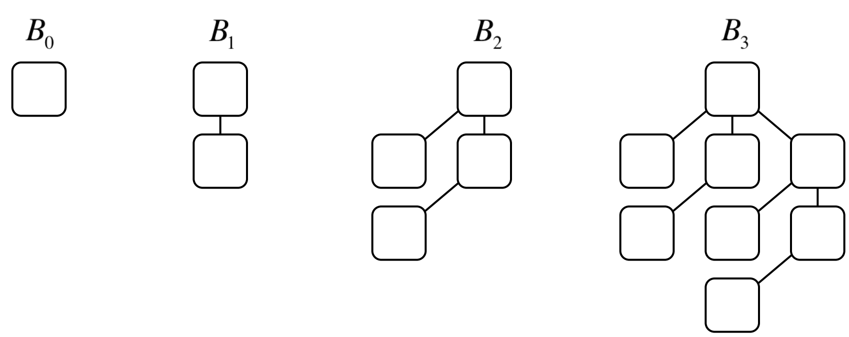
Optional Implementation 4: Binomial Heap

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.

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

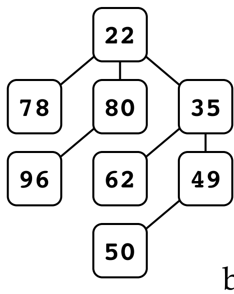
- A binomial tree of order 0 is a single node with no children.
- A binomial tree of order k is a single node at the root with k children, indexed 0 through k – 1. The 0^{th} child is a binomial tree of order 0, the 1^{st} child is a binomial tree of order 1, …, the m^{th} child is a binomial tree of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

Some binomial trees:

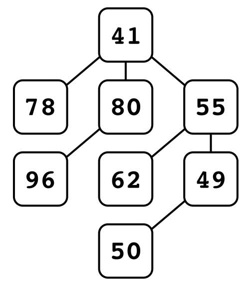


One property to note—and one that will certainly be exploited in the coming paragraphs, is that one can assemble a binomial tree of order k + 1 out of two order k trees by simply appending the second to the end of the first's list of children. A related property: each binomial tree of order k has a total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

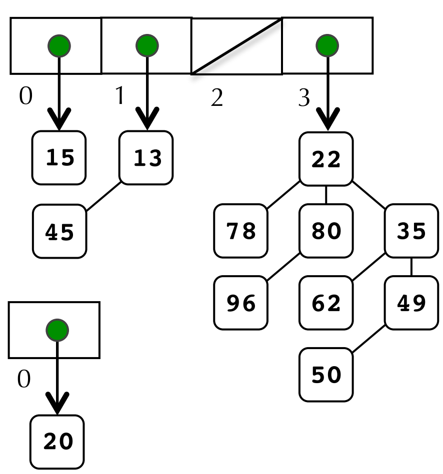
Now, if binary heaps can back priority queues, then so can binomial heaps. But the number of elements held by a priority queue can't be constrained to be some power of 2 all the time. So priority queues, when backed by binomial heaps, are really backed by a **`Vector`** of binomial heaps.

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!

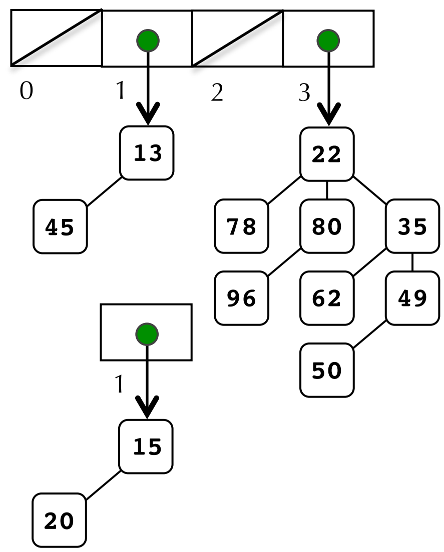
Binomial Heap Insert

What happens when we introduce a new element into the mix? More formally: What happens when you enqueue a new string into the **`Vector<node *>`**-backed priority queue? Let's see what happens when we enqueue a **"20"**.

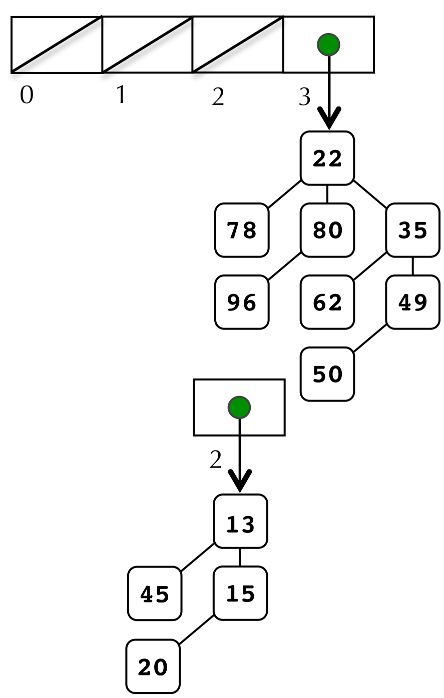
The size of the priority queue goes from 11 to 12—or rather, from 1011 to 1100. We'll understand how to add this new element, all the time maintaining the heap ordering property within each binomial tree, if we emulate binary addition as closely as possible. That emulation begins by creating binomial 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<node *>`**.



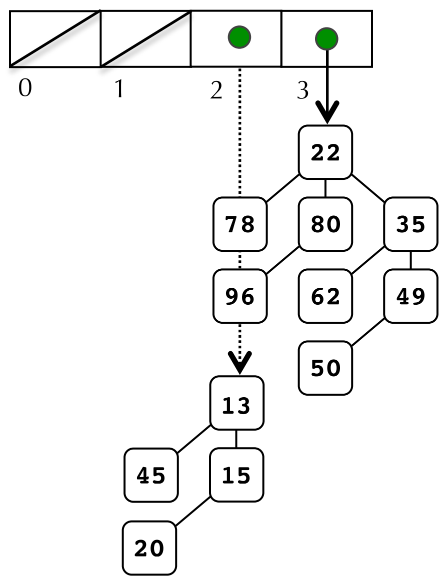
Now, when we add 1 and 1 in binary, we get 0, and carry a 1, right? We do the same thing when merging two order-0 binomial heaps, order-0 plus order-0 equal **NULL**, carry the order-1. One key difference: when you merge two order-0 heaps into the order-1 that gets carried, you need to make sure the 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.



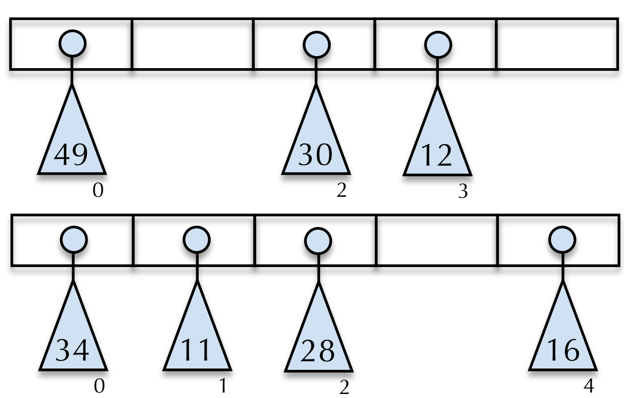
Had there been an order-2 in the original figure, the cascade of merges would have continued. But because there's no order-2 binomial heap in the original, the order-2 carry can assume that position in the collection and the cascade can end. In our example, the original binomial heap collection would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

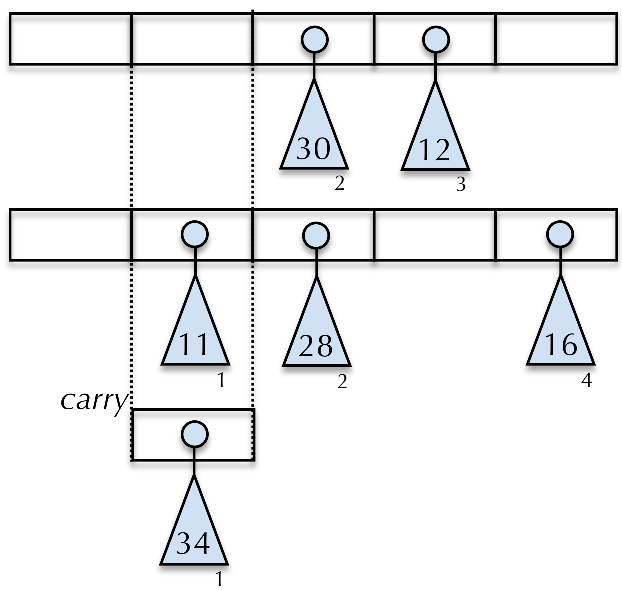
You can merge two heaps using an extension of the binary addition emulated while discussion enqueue. As it turns out, enqueuing a single node is really the same as merging an arbitrarily large binomial heap with a binomial heap of size 1. The generic merge problem is concerned with the unification of 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:



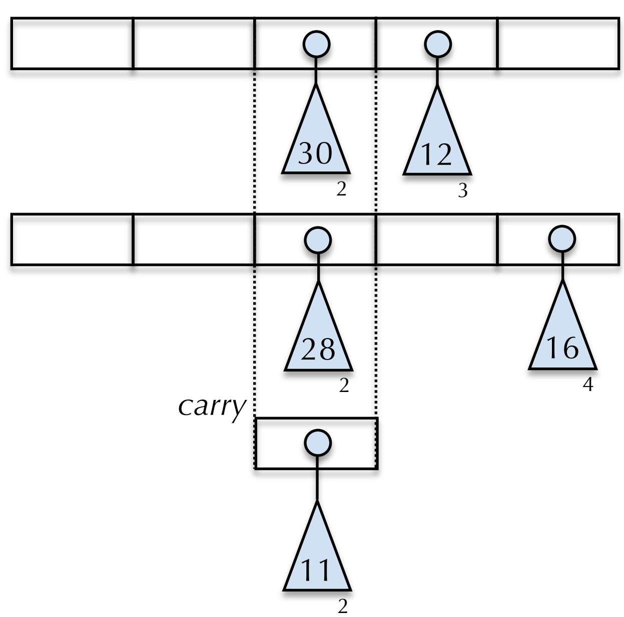
The triangles represent binomial trees respecting the heap ordering properties, and the subscripts represent their order. The numbers within the triangles are the root values—the smallest in the tree, and the blanks represent **`NULL`**. (We don't have space for the more elaborate pictures used to illustrate **`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:

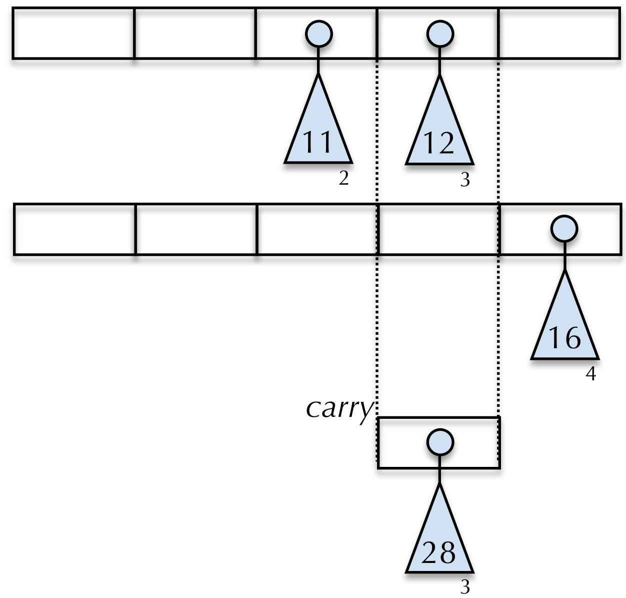
1. Merge the two order-0 trees to produce an order-1 (with 34 at the root) that carries.



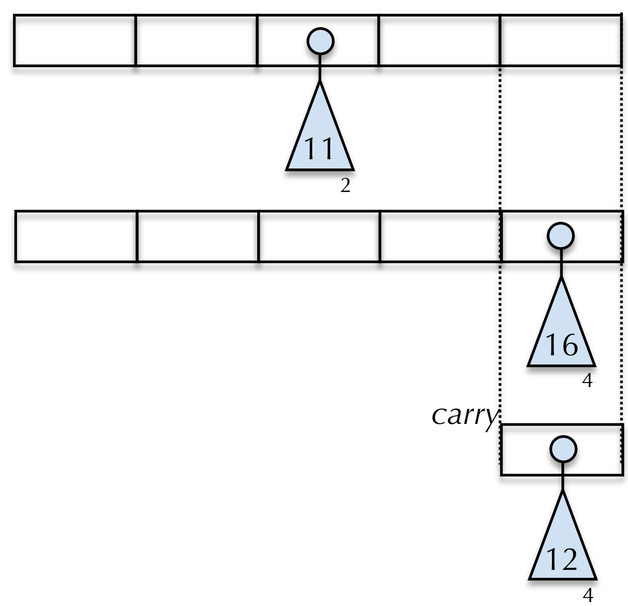
2. Merge the two order-1 trees to produce an order-2 tree carry, with 11 at the root.



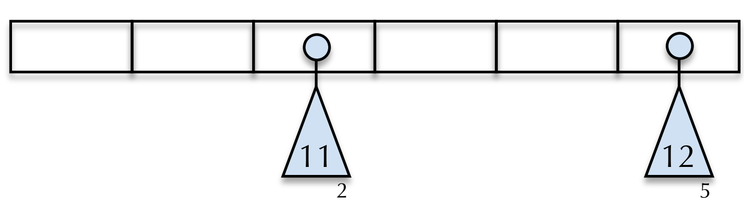
3. Merge the three (three!) order-2 trees! Leave one of the three alone (we'll leave the 11 in place, though it could have been any of the three) and merge the other two to produce an order-3 (with the smaller of 28 and 30 as the root).



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 node {
   std::string elem;
   Vector<node *> 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.

Accessing files

On the class web site, there are two starter projects: one for XCode and a one for Visual Studio. Each project contains these files:

`qqueue-test.cpp`	Test harness to assist in development and testing.
`pqueue`	Interface and partial implementation of base **`PQueue`** class. The primary purpose of the **`PQueue`** class is to define the interface that all concrete priority queue implementations should agree on.
`pqueue-vector`	Interface and implementation file housing the unsorted vector version of the priority queue.
`pqueue-linked-list`	Interface and implementation file housing the sorted, doubly linked list version of the priority queue.
`pqueue-heap`	Interface and implementation file housing the version of the priority queue that's backed by the array-packed binary heap.
`pqueue-binomial-heap`	Interface and implementation file housing the version of the priority queue that's backed by the binomial heap. Remember that this is totally optional.