Stanford University, CS 106X
Homework Assignment 5: Priority Queue
Binomial Heap Optional Extension

Extension description by Jerry Cain.

This document describes an optional extension to the assignment. It is very challenging and not recommended for most students, but we are posting it in case it is of interest.

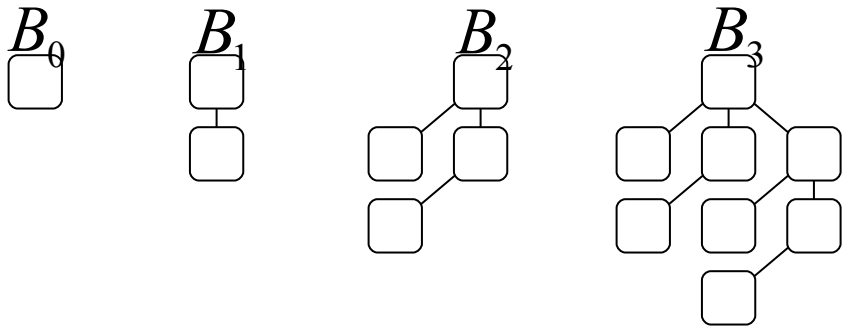
Optional Implementation: Binomial Heap

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:

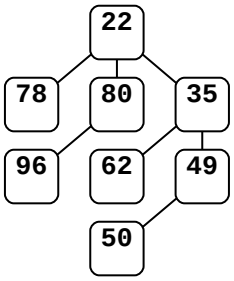
- A binomial tree of order 0 is a single node with no children.
- A binomial tree of order k is a single node at the root with k children, indexed 0 through k – 1. The 0^{th} child is a binomial tree of order 0, the 1^{st} child is a binomial tree of order 1, …, the m^{th} child is a binomial tree of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

Some binomial trees:

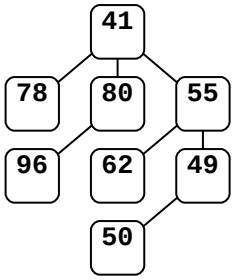


One property to note—and one that will certainly be exploited in the coming paragraphs, is that one can assemble a binomial tree of order k + 1 out of two order k trees by simply appending the second to the end of the first's list of children. A related property: each binomial tree of order k has a total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

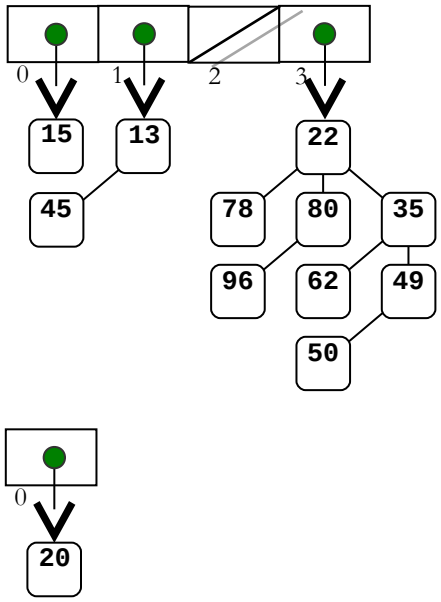
Now, if binary heaps can back priority queues, then so can binomial heaps. But the number of elements held by a priority queue can't be constrained to be some power of 2 all the time. So priority queues, when backed by binomial heaps, are really backed by a **`Vector`** of binomial heaps.

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!

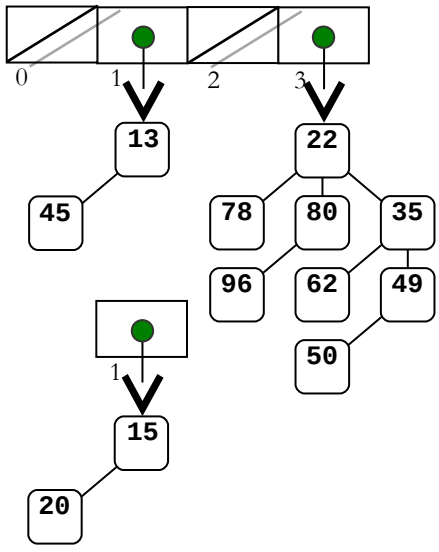
Binomial Heap Insert (Enqueue)

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

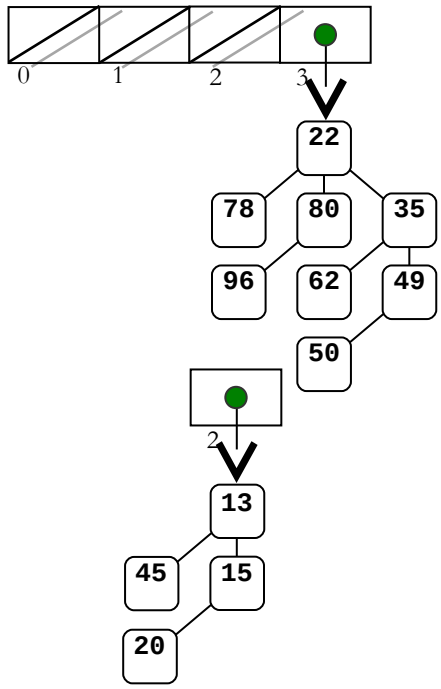
The size of the priority queue goes from 11 to 12—or rather, from 1011 to 1100. We'll understand how to add this new element, all the time maintaining the heap ordering property within each binomial tree, if we emulate binary addition as closely as possible. That emulation begins by creating binomial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



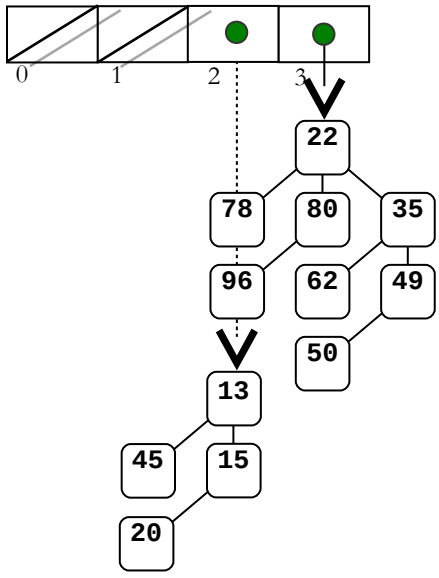
Now, when we add 1 and 1 in binary, we get 0, and carry a 1, right? We do the same thing when merging two order-0 binomial heaps, order-0 plus order-0 equal NULL, carry the order-1. One key difference: when you merge two order-0 heaps into the order-1 that gets carried, you need to make sure the heap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



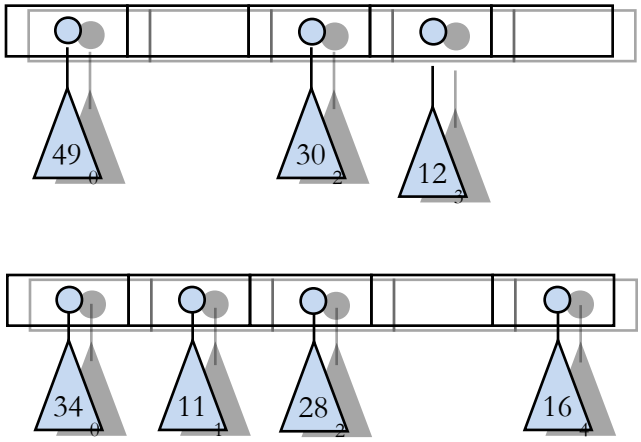
Had there been an order-2 in the original figure, the cascade of merges would have continued. But because there's no order-2 binomial heap in the original, the order-2 carry can assume that position in the collection and the cascade can end. In our example, the original binomial heap collection would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

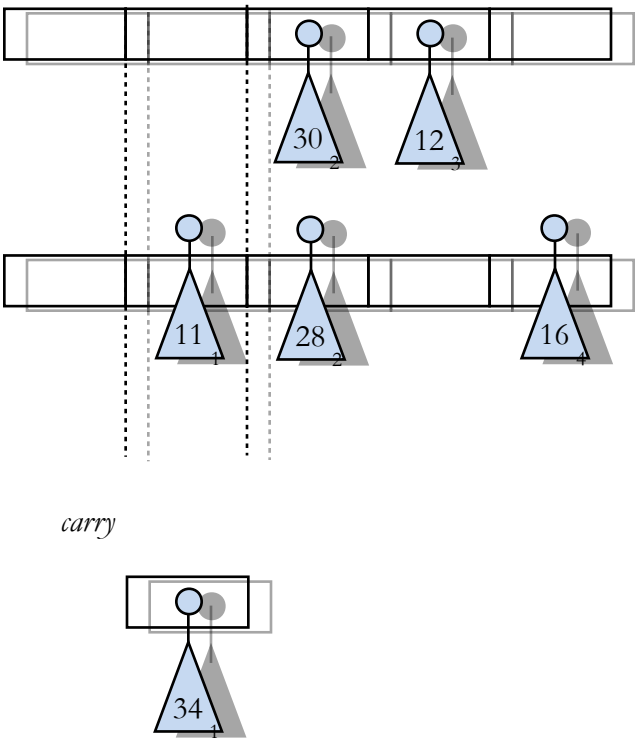
You can merge two heaps using an extension of the binary addition emulated while discussion enqueue. As it turns out, enqueuing a single node is really the same as merging an arbitrarily large binomial heap with a binomial heap of size 1. The generic merge problem is concerned with the unification of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



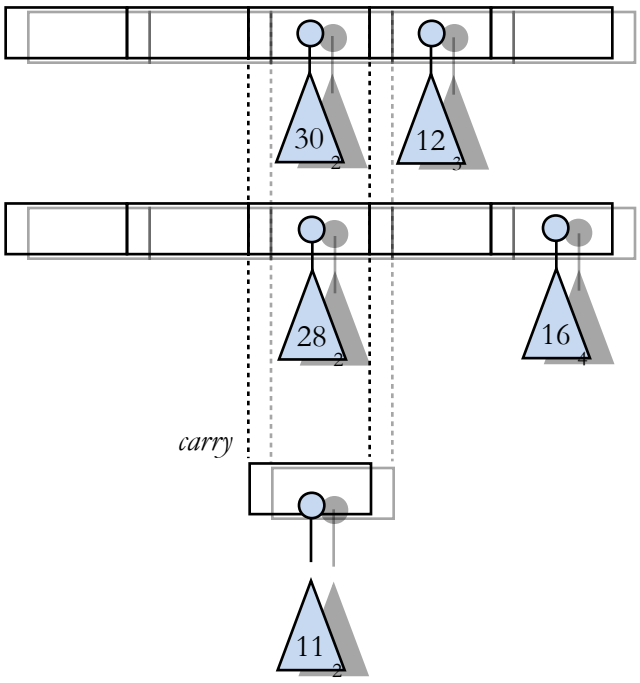
The triangles represent binomial trees respecting the heap ordering properties, and the subscripts represent their order. The numbers within the triangles are the root values—the smallest in the tree, and the blanks represent **NULL**. (We don't have space for the more elaborate pictures used to illustrate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

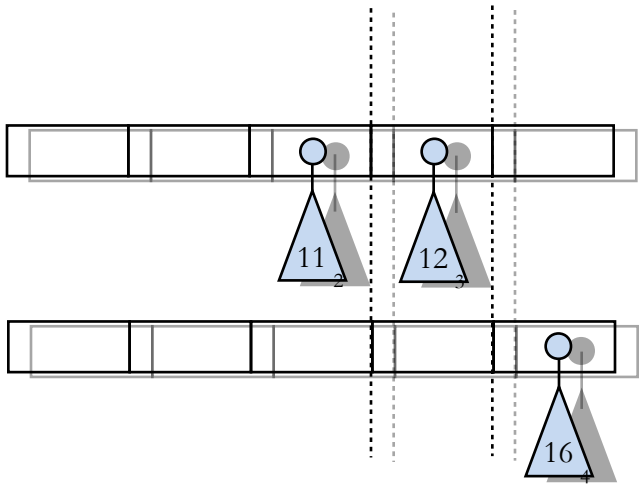
1. Merge the two order-0 trees to produce an order-1 (with 34 at the root) that carries.

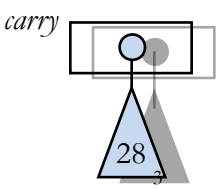


2. Merge the two order-1 trees to produce an order-2 tree carry, with 11 at the root.

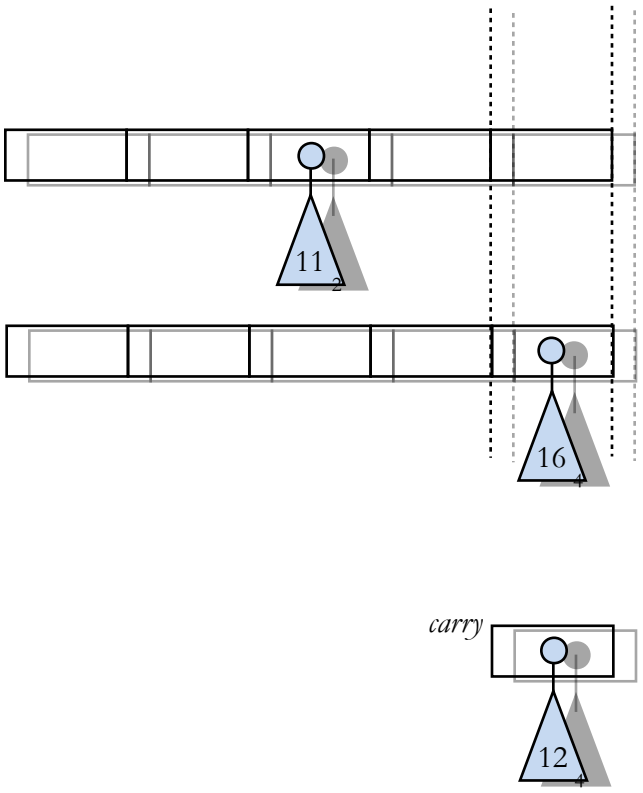


3. Merge the three (three!) order-2 trees! Leave one of the three alone (we'll leave the 11 in place, though it could have been any of the three) and merge the other two to produce an order-3 (with the smaller of 28 and 30 as the root).

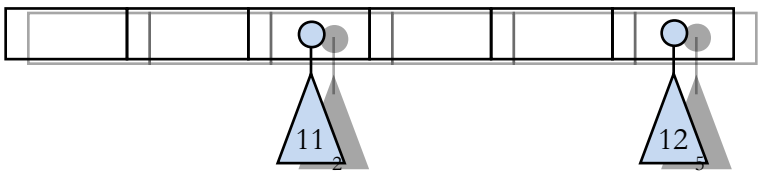




4. Merge the two order-3 trees to produce an order-4 tree carry, with 12 as the root:



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 dequeue

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). **dequeue** 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 **dequeue** 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 {
    string value;
    Vector<BinomialHeap *> 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.