Assignment 5: Priority Queue

作业 5：优先队列

Assignment idea, handout introduction, and binary heap specifications come from by Julie Zelenski.

作业构思、讲义介绍和二进制堆规范来自 Julie Zelenski。

Everything else, including the binomial heap implementation and rest of the handout, comes from Jerry.

其他所有内容，包括二项式堆的实现和讲义的其他部分，都来自杰里。

It's finally time for you to implement a class of your own. That class is a priority queue, and it's a variation on the standard queue described in lecture and in the course textbook.

现在终于到了你自己实现一个类的时候了。这个类是优先级队列，是讲座和教材中描述的标准队列的变体。

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, however, a FIFO strategy may be too simplistic for the problem being solved. A hospital emergency room, for example, needs to schedule patients according to priority. A patient who arrives with a more serious problem should pre-empt 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.

标准队列是以先进先出方式管理的元素集合。添加到集合中的第一个元素总是第一个被提取出来的元素；第二个是第二个；以此类推。然而，在某些情况下，先进先出的策略对于要解决的问题来说可能过于简单。例如，医院急诊室需要根据优先级安排病人的就诊时间。问题更严重的病人即使等待时间更长，也应优先于其他病人。这是一个优先级队列，元素以任意顺序添加到队列中，但当需要提取下一个元素时，队列中优先级最高的元素会被移除。

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 master the pointer gymnastics that go along with it.

本作业的重点是用几种不同的方法实现一个优先级队列类。您将有机会尝试使用数组和链接结构，并在此过程中掌握与之相关的指针技巧。

Monday, November 4th at 11:59pm

11月 4th 日星期一晚上11:59

The PQueue Interface PQueue 接口

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.

优先级队列是字符串的集合。词法上较小的字符串优先级应高于词法上较大的字符串，因此无论插入顺序如何，"ping "的优先级都高于 "pong"。

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

以下是构成所有优先级队列公共接口的方法：

```
class PQueue {
public:
    void enqueue(const std::string& elem);
    std::string extractMin();
    const 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 supplied queues and returns their union as a new priority queue. For detailed descriptions of how these methods behave, see the pqueue.h interface file included in the starter files.

enqueue 用于向优先级队列中插入新元素。 extractmin 返回队列中优先级最高（即词典中最小）的元素值，并将其删除。 merge 对提供的队列进行破坏性统一，并将其合并为一个新的优先级队列返回。有关这些方法的详细说明，请参阅启动文件中的 pqueue.h 接口文件。

Implementing the priority queue

实施优先队列

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 as the 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.

虽然外部表示法可能会给人一种错觉，以为我们一直在幕后按排序存储所有内容，但事实上，我们在选择内部表示法时有很大的灵活性。当然，所有操作都需要正常运行，而且我们希望它们运行得更快。但是，我们可能不是为了速度而优化，而是为了便于实现，或者是为了尽量减少内存占用。也许我们优化某个操作时会牺牲其他操作。

This assignment is all about client expectations, implementation and internal representation. 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 an abstraction.

这项任务主要涉及客户期望、实施和内部表示。在这一过程中，你将掌握数组和链表，但这项作业的收获--或者说众多收获中最重要的一点--是你可以使用任何你认为合适的机制来管理抽象的内部结构。

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

你将用四种不同的方法实现优先队列。其中两种简单明了，但第三种并非易事，第四种则非常具有挑战性（尽管它是如此整洁、巧妙和美观，值得我们为之奋斗）。

Implementation 1: Optimized for simplicity and for the enqueue method by backing your priority queue by an unsorted vector<string>. merge is pretty straightforward, and 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 5. I don't provide much in terms of details on this one, as it's pretty simple.

实现 1：通过以无排序向量<string> 作为优先队列的后盾，优化了 enqueue 方法的简洁性。合并非常简单，而 peek 和 extractmin 的代价很高，但在需要快速启动并运行原型或概念验证版本，或者需要 enqueue 50,000 个元素并提取区区 5 个元素的情况下，这种代价可能是值得的。关于这一点，我没有提供太多细节，因为它非常简单。

Implementation 2: 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.

实现 2：通过二进制堆来实现优先级队列，从而平衡插入和提取时间。如果实现得当，peek 运行时间为 $O(1)$ ，enqueue 和 extractMin 运行时间分别为 $O(\lg n)$ ，而合并运行时间为 $O(n)$ 。

Implementation 3: 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.

实现 3：通过在幕后维护一个已排序的双链路（需要 next 和 prev 指针）字符串列表，对简单性和 extractMin 操作进行了优化。

Implementation 4: enqueue, extractMin, and merge all run in $O(\lg n)$, but only because we use a collection of binomial heaps behind the scenes. Binomial heaps are by far the most intense of the four data structures used in Assignment 5, so leave plenty of time ahead of the deadline to work on it.

实现 4：enqueue、extractMin 和 merge 都在 $O(\lg n)$ 中运行，但这只是因为我们在幕后使用了二叉堆集合。到目前为止，二叉堆是作业 5 中使用的四种数据结构中最复杂的一种，因此请在截止日期前留出充足的时间来处理它。

Implementation 1: Unsorted Vector

实施 1：未排序向量

This implementation is layered on top of our vector class. enqueue simply appends the new element to the end. When it comes time to extract the highest priority element, it performs a linear search to find it. The implementation is straightforward as far as layered abstractions go and serves more as an introduction to the assignment architecture than it does as an interesting implementation. Do this one first.

enqueue 只是将新元素添加到尾部。当需要提取优先级最高的元素时，它会执行线性搜索来找到它。就分层抽象而言，该实现非常简单，与其说是有趣的实现，不如说是赋值架构的介绍。先做这个。

Aside

As you implement each of the subclasses, you'll leave pqueue. **h** and pqueue.cpp alone, and instead be modifying 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:

在实现每个子类的过程中，你将不去管 pqueue. **h** 和 pqueue.cpp，而是修改每个子类的接口 (. **h**) 和实现 (. **cpp**) 文件。pqueue-vector.h 定义了要实现的公共接口，但其私有部分是空的：

```
class VectorPQueue : public PQueue {
public:
    VectorPQueue();
    ~VectorPQueue();
    static VectorPQueue *merge(VectorPQueue *one, VectorPQueue *two);
    void enqueue(const std::string& elem);
    std::string extractMin();
    const 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 provided comment and insert the collection of data members and private helper functions you need.

毫不奇怪，私有部分不应该是空的，而应该列出构成内部表示的项目。您应该擦除所提供的注释，然后插入所需的数据成员集合和私有辅助函数。

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

pqueue-vector. cpp 文件提供了所有内容的虚拟、占位符实现，以便项目能顺利编译。在少数情况下，虚拟实现会做一些正确的事情，但绝大多数情况下都需要更新，以包含能做真正事情的真实代码。

Important: 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 three implementations. By unifying the logSize field to the parent class, we can implement size and isEmpty at the PQueue class level (I already did for you), and all subclasses just inherit it.

重要：父 PQueue 类定义了一个名为 logSize 的受保护字段，这意味着你可以访问它。你应该确保 logsize 始终保持优先队列的逻辑大小，在这里和其他三个实现中都是如此。通过将 logSize 字段统一到父类，我们可以在 PQueue 类级别实现 size 和 isEmpty（我已经为你实现了），所有子类都只需继承它即可。

As you advance through the implementations, understand that you'll be modifying different pairs of interface and implementation files (pqueue-heap.h and pqueueheap. cpp for the binary heap version, etc). In all cases, the private sections of the interface are empty and need to be fleshed out, and in all cases the implementation files have placeholder implementations to seduce the compiler into thinking everything is good.

在逐步实现的过程中，请注意您将修改不同的接口和实现文件（二进制堆版本的 pqueue-heap.h 和 pqueueheap.）在所有情况下，接口的私有部分都是空的，需要加以充实；在所有情况下，实现文件都有占位符实现，以诱使编译器认为一切正常。

Implementation 2: Binary Heap

实现 2：二进制堆

Although the binary search trees we'll eventually discuss 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.

虽然我们最终要讨论的二叉搜索树可能是优先级队列的良好实现方式，但在这种情况下，还有一种二叉树是更好的选择。

A heap is a binary tree that has these two properties:

堆是具有这两个特性的二叉树：

It is a complete binary tree, i.e. one that is full in all levels (all nodes have two children), except for possibly the bottommost level, which is filled in from left to right with no gaps.

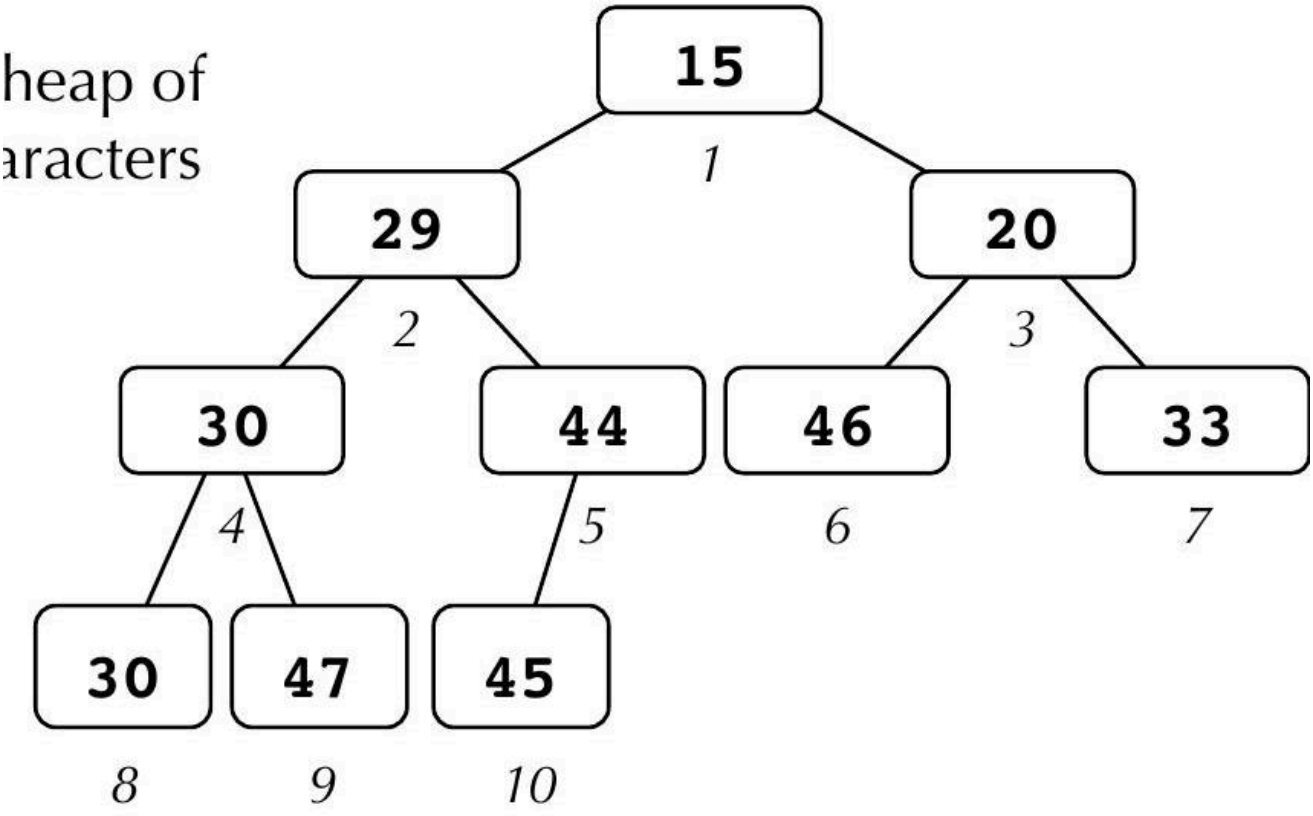
它是一棵完整的二叉树，即所有层次都是完整的（所有节点都有两个子节点），可能最底层除外，因为最底层从左到右都是完整的，没有空隙。

The value of each node is less than or equal to the value of its children.

每个节点的值小于或等于其子节点的值。

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

下面是一小堆整数字符串（即所有字符都是数字的字符串）的概念图：



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 more loosely. Conveniently, the manner in which a heap is ordered is 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 binary heap containing ten elements is the same shape as every

other binary heap with ten elements.

请注意，堆与二叉搜索树有两个重要的不同点。首先，二叉搜索树将所有节点排序，而堆的排序则更为松散。方便的是，堆的排序方式足以保证标准优先级队列操作的高效执行。第二个重要区别是，二叉搜索树的形状多种多样，而堆必须是一棵完整的二叉树，这意味着每个包含 10 个元素的二叉堆与其他包含 10 个元素的二叉堆形状相同。

Representing a heap using an array

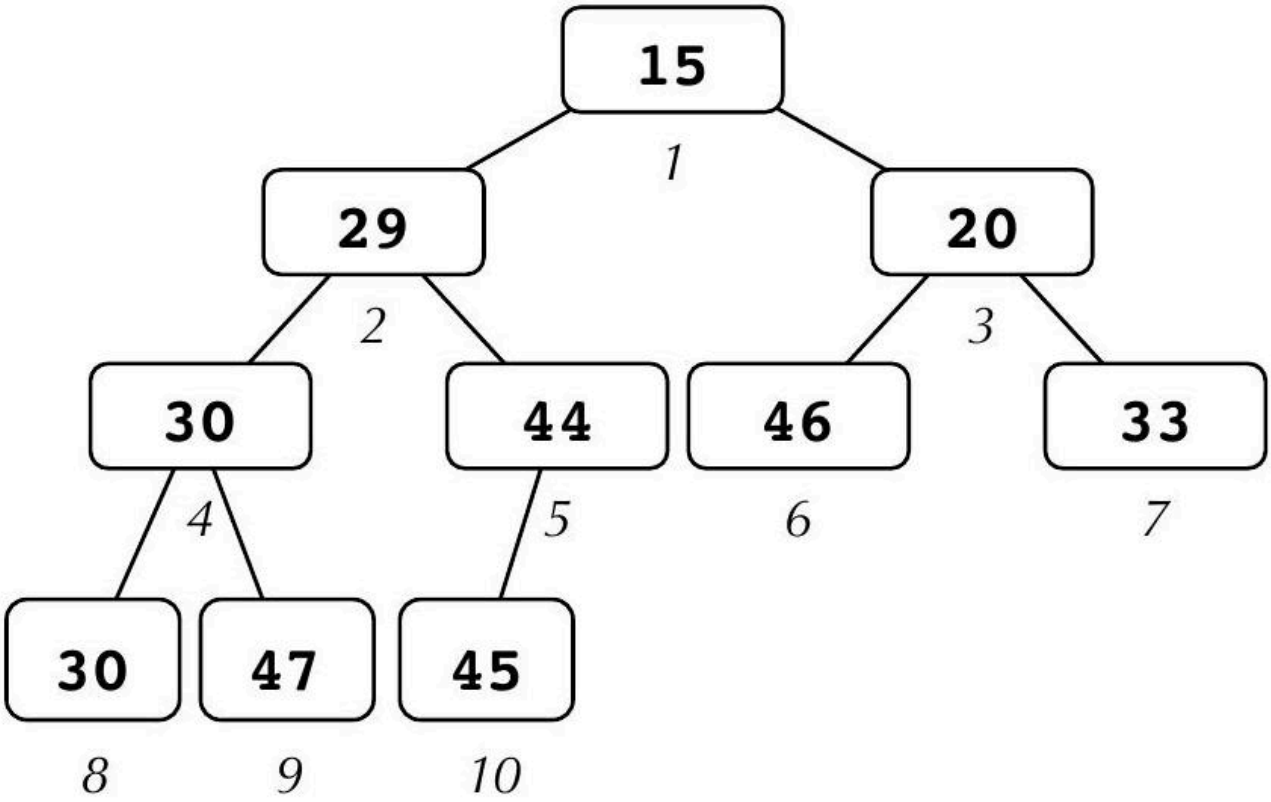
使用数组表示堆

One way to manage a heap would be to use a standard binary tree node definition and wire up left and right child pointers to sub-trees. However, we can and will exploit the completeness of the binary heap's tree and create a simple array representation and avoid the pointers.

管理堆的一种方法是使用标准的二叉树节点定义，并为子树连接左右子指针。不过，我们可以并将利用二叉堆树的完整性，创建一个简单的数组表示法，并避免使用指针。

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

将堆中的节点逐级编号，就像这样：



and check out this array representation of the same heap:

并查看同一堆的数组表示：

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 (e.g., the parent of 9 is 4). The two children of node i are $2i$ and $2i + 1$, e.g. node 3 's two children are 6 and 7 .

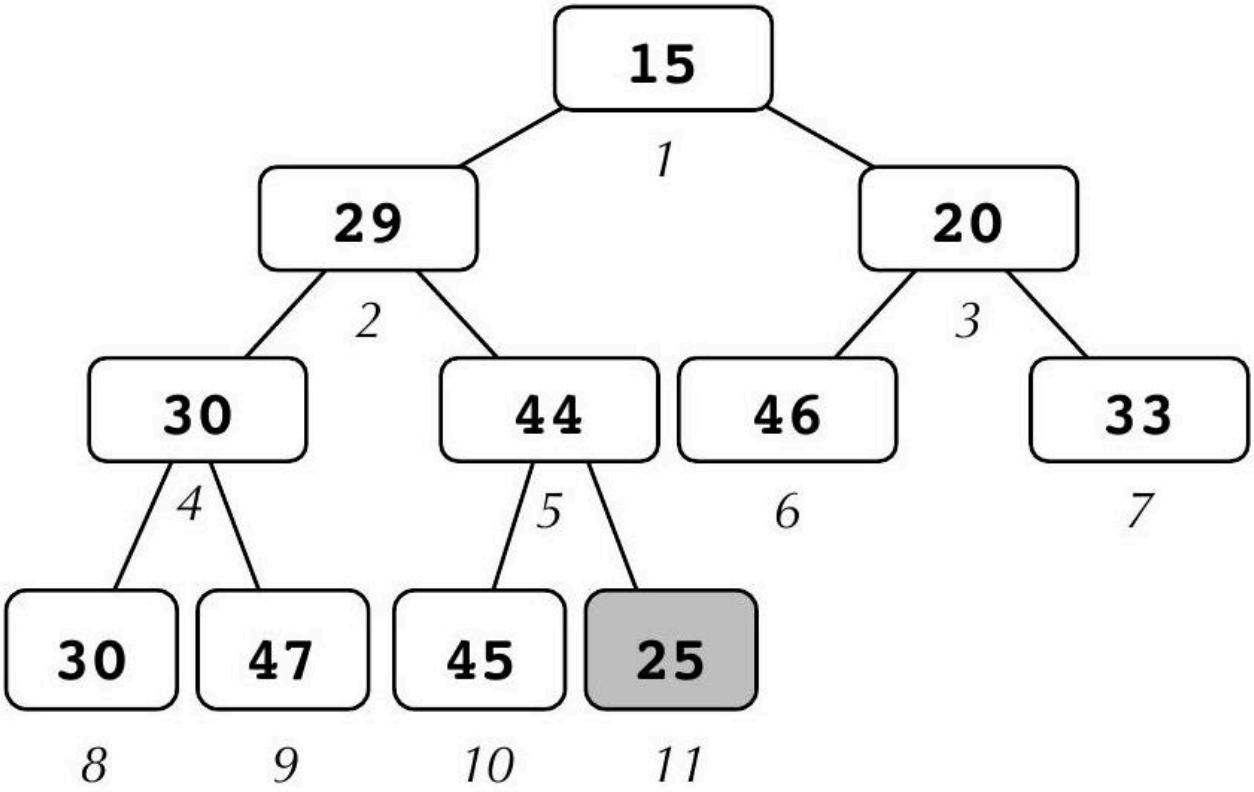
Although many of the drawings in this handout use a tree diagram for the heap, keep in mind you will actually be representing the binary heap with a dynamically allocated array, much like the vector does.

你可以用任何节点编号除以 2（去掉余数），得到它的父节点编号（例如，9 的父节点编号是 4 ）。节点 i 的两个子节点是 $2i$ 和 $2i+1$，例如，节点 3 的两个子节点是 6 和 7。虽然本讲义中的许多图都使用树形图来表示堆，但请记住，您实际上是在用动态分配的数组来表示二进制堆，就像向量一样。

Heap insert 堆插入

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 a “25”. We add a new node at the bottom of the heap (the insertion position is equal to the size of the heap), as with:

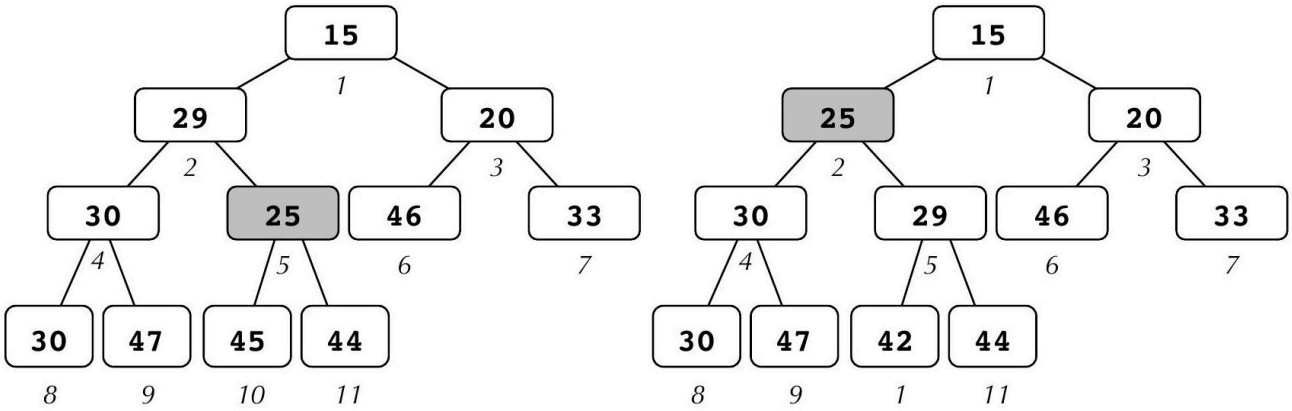
一个新元素会被添加到堆的最底层，然后上升到合适的位置。例如，假设我们要插入一个 "25"。我们在堆的底部添加一个新节点（插入位置等于堆的大小），如下所示：



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 ascended value to its new parent and continue moving the value up until it need go no further.

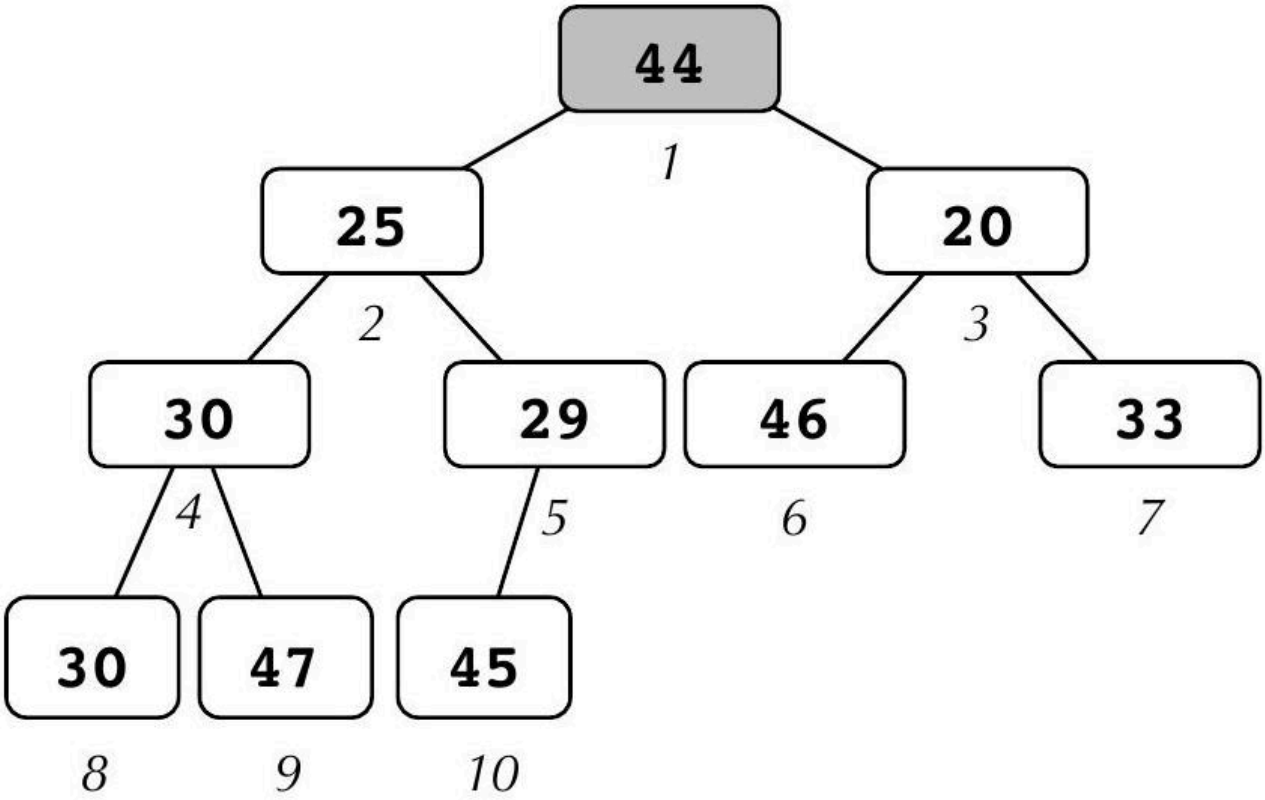
我们将新节点的值与其父节点的值进行比较，如有必要，则交换它们。由于我们的堆实际上是一个数组，因此我们通过交换数组值来 "交换 "节点。然后，我们将上升的值与其新的父节点进行比较，并继续向上移动该值，直到不再需要为止。



Heap remove 堆删除

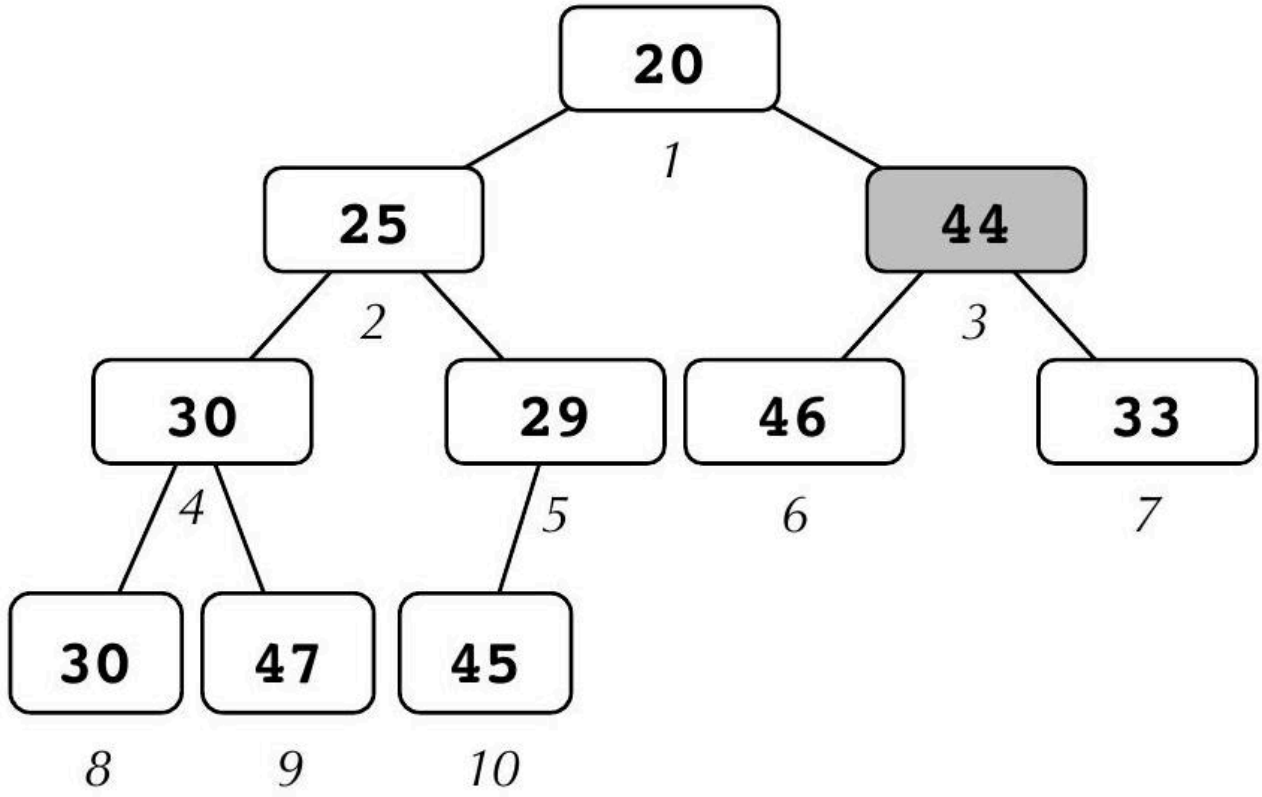
The binary heap property guarantees that the smallest value resides at the root, where it can be easily accessed. After removing this value, you typically need to rearrange those that remain. The completeness property dictates the shape of the heap, so it's the bottommost node that needs to be removed. Rather than re-arranging everything to fill in the gap left at the root, we leave the root node where it is, copy the value from the last node to the root, and remove the last node.

二进制堆属性保证了最小值位于根部，便于访问。移除该值后，通常需要重新排列剩余的值。完整性属性决定了堆的形状，因此需要移除的是最底层的节点。与其重新排列所有节点来填补根节点留下的空白，我们不如保留根节点，将最后一个节点的值复制到根节点，然后删除最后一个节点。



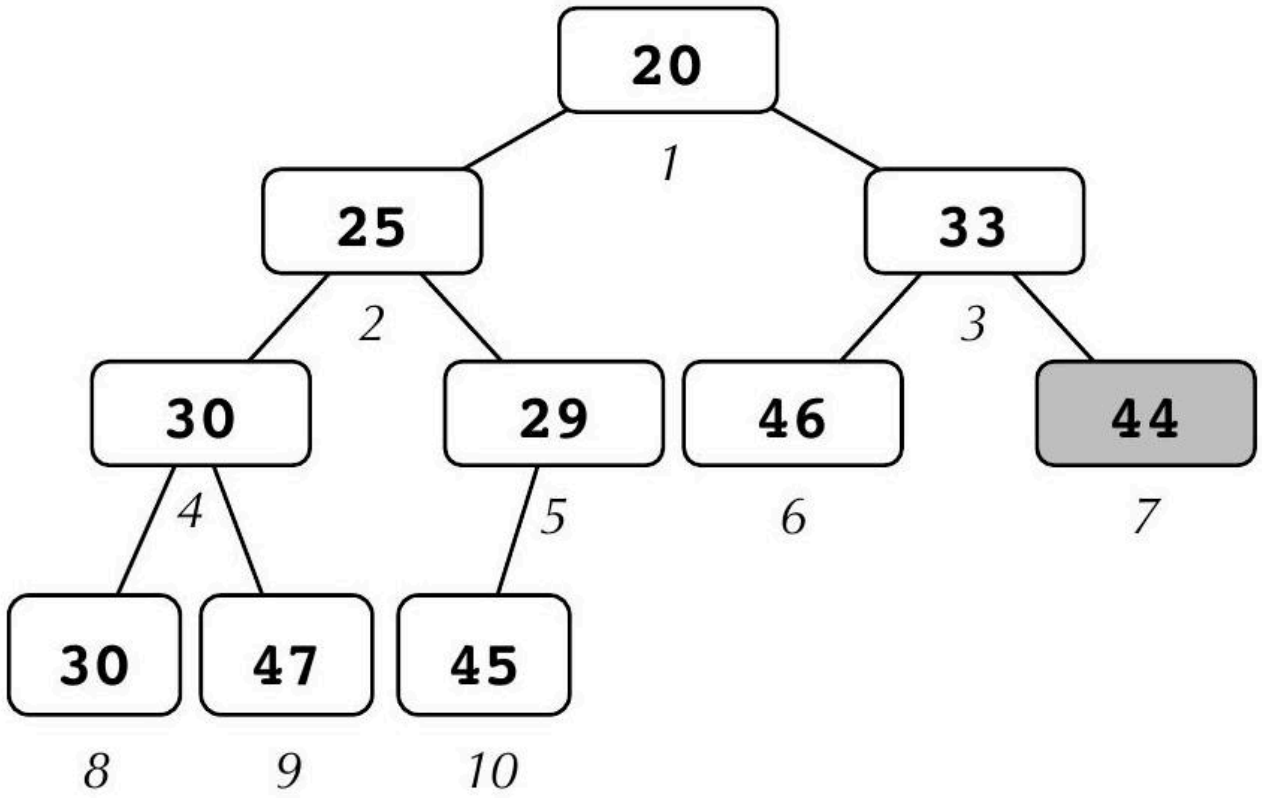
We still have a complete tree with one less value, and the left and right sub-trees are still legit binary heaps. The only potential problem: a violation of the heap ordering property, localized at the root. In order to restore the min-heap property, we need to trickle that value down to the right place. 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 one:

我们仍然有一棵少了一个值的完整树，左右子树仍然是合法的二叉堆。唯一的潜在问题是：违反了堆排序属性，而且是在根部。为了恢复最小堆属性，我们需要将该值下传到正确的位置。首先将根的值与其两个子代的值进行比较。如果根中的值大于任一子代的值，则将根中的值与较小子代的值交换：



This step fixes the heap ordering property for the root node, but at the expense of the impacted sub-tree. The sub-tree is now another heap where its root node is out of order, so we can iteratively apply the same reordering on the sub-tree to fix it and any other impacted sub-trees.

这一步修复了根节点的堆排序属性，但却牺牲了受影响的子树。现在，该子树是另一个堆，其根节点的顺序已经失效，因此我们可以对该子树反复应用相同的重排序，以修复它和其他受影响的子树。



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 bubble-down algorithm is often referred to as heapify-ing.

当数值下降到比其两个子代都小或没有子代时，就停止向下堆积。这种向下冒泡的算法通常被称为堆化算法（heapify-ing）。

You should implement the priority queue as a binary heap array using the strategy outlined above. This array should start small (initially allocated space for 4 strings) and grow as needed.

应使用上述策略将优先级队列作为二进制堆数组来实现。这个数组一开始应该很小（最初为 4 个字符串分配空间），然后根据需要不断扩大。

Merging two heaps 合并两个堆

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:

合并操作，即销毁两个堆并创建一个新堆，它是两个原始堆的逻辑结合。是的，你可以把第二个堆中的元素一个一个地插入第一个堆中，直到第二个堆中的元素用完，第一个堆中的元素全部用完。但实际上，用下面的方法会更快--实际上是近似地更快：

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

创建一个数组，它是第一个堆的数组和第二个堆的数组的逻辑连接，不考虑堆的排序属性。数组的

result is a complete, array-backed binary tree that in all likelihood isn't even close to being a binary heap.

结果是一棵完整的、有数组支持的二叉树，但它很可能连二叉堆都算不上。

Recognize that all of the leaf nodes-taken in isolation—respect the heap property.

认识到所有叶节点--孤立地看--都尊重堆属性。

Heapify all sub-heaps rooted at the parents of all the leaves.

堆化扎根于所有叶子父代的所有子堆。

Heapify all sub-heaps rooted at the grandparents of all the leaves.

堆化扎根于所有叶子祖父母的所有子堆。

Continue to heapify increasingly higher ancestors until you reach the root, and heapify that as well.

继续堆砌越来越高的祖先，直到堆砌到根部，然后也堆砌到根部。

Binary Heap Implementation Notes

二进制堆实施说明

Manage your own raw memory. It's tempting to just use a Vector<string> to manage the array of elements. But using a 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 strings instead of a Vector<string>.

管理自己的原始内存使用矢量（Vector<string>）来管理元素数组很有诱惑力。但使用矢量会在 HeapPQueue 和实际存储元素的内存之间引入额外的代码层，而在实际应用中，像 HeapPQueue 这样的核心容器类的实现是不需要额外代码层的。请务必使用原始的、动态分配的字符串数组来实现 HeapPqueue，而不是使用 Vector<string>。

Freeing memory. You are responsible for properly managing heap-allocated memory. Your implementation should not orphan any memory during its operations and the destructor should free everything it needs to.

释放内存。您有责任正确管理堆分配的内存。您的程序在运行过程中不应释放任何内存，析构函数应释放所有需要释放的内存。

Think before you code. The amount of code necessary to complete the implementation is not large, but you will find it requires a good amount of thought. It'll help to sketch things on paper and work through the boundary cases carefully before you write any code.

在编写代码之前先思考。完成实施所需的代码量并不大，但你会发现这需要大量的思考。在编写任何代码之前，先在纸上画出草图，并仔细研究边界情况，这将会有所帮助。

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

彻底测试。我知道这一点我们已经说过了，但重复几遍总不会有错。如果我们在评分过程中发现了一堆潜伏的问题，而你因为测试不充分而没有发现，你可千万不要大吃一惊。您编写的代码有一些复杂的交互，您必须花时间识别和测试所有不同的情况。我为您提供了最基本的测试工具，以确保

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 can't break it any longer.

HeapPQueue 可以在简单的情况下工作，但你的任务是扮演反派角色，尝试破坏你的实现，当你无法再破坏它时，你就大功告成了。

Implementation 3: Sorted doubly linked list

实现 3：排序双链表

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 it's 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. This is the first linked structure you'll be writing for us, and you should review the textbook's implementation of the queue and read over the All About Linked Lists handout beforehand.

链表的实现是值的双链表，其中的值按排序顺序（即从小到大）保存，以方便快速查找和删除最小的元素。Merge 在概念上很简单，但对于初次学习指针和链表的人来说，实现起来可能比较棘手。这是你要为我们编写的第一个链表结构，你应该先复习一下教科书中队列的实现，并阅读一下《关于链表》讲义。

Implementation 4: Binomial Heaps

实施 4：二项式堆

The binomial heap expands on the idea of a binary heap by maintaining a collection of binomial trees, each of which respect a property very similar to the heap order property discussed for Implementation #2.

二叉树堆对二叉树堆的概念进行了扩展，它维护着一个二叉树集合，每个二叉树都尊重与实现 #2 中讨论的堆顺序属性非常相似的属性。

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

k 阶（k 为正整数）的二叉树是递归定义的：

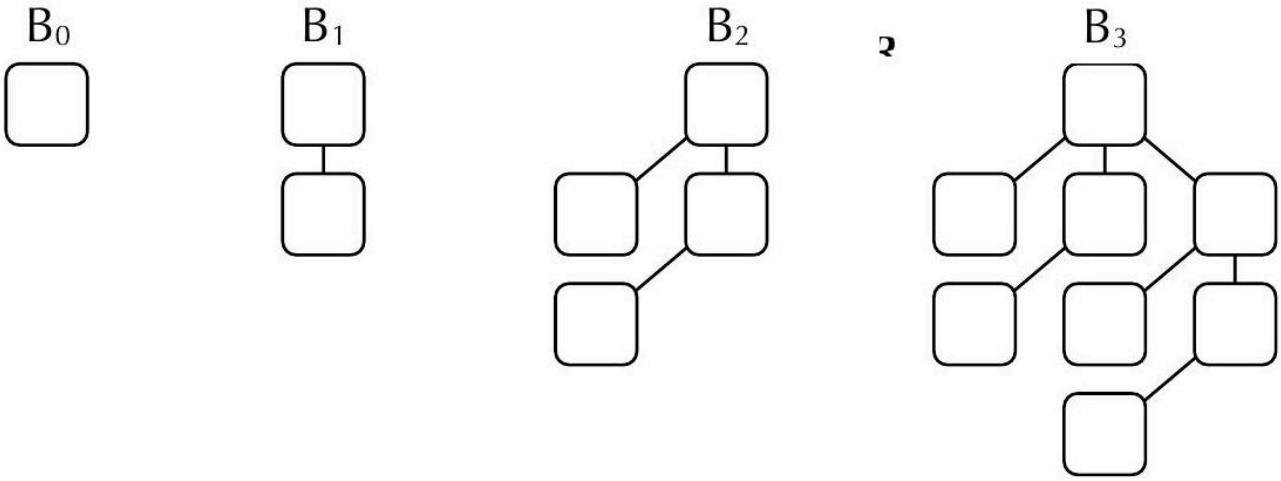
A binomial tree of order 0 is a single node with no children.

阶数为 0 的二叉树只有一个节点，没有子节点。

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 $1, \cdots$, the m^{th} child is a binomial tree of order m, and the $k-1^{\text{st}}$ child is a binomial tree of order $\mathrm{k}-1$.

阶数为 k 的二叉树是指根部有一个节点，节点上有 k 个子节点，索引为 0 至 $k-1$ 。子节点 0^{th} 是一棵阶为 0 的二叉树，子节点 1^{st} 是一棵阶为 $1, \cdots$ 的二叉树，子节点 m^{th} 是一棵阶为 m 的二叉树，子节点 $k-1^{\text{st}}$ 是一棵阶为 $\mathrm{k}-1$ 的二叉树。

Some binomial trees: 一些二叉树



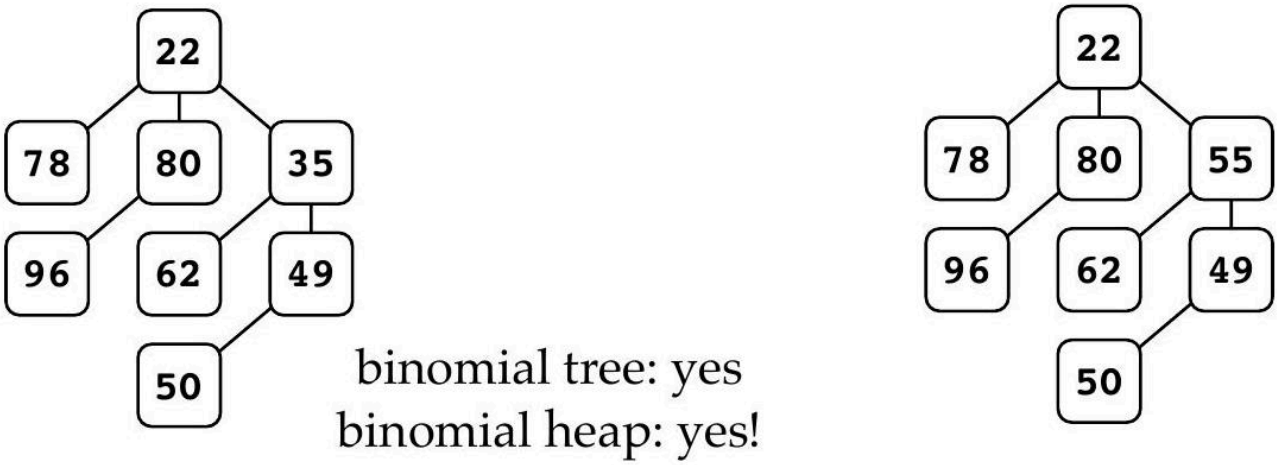
One property to note-and one that will certainly be exploited in the coming paragraphs, is that one can assemble a binomial tree of order $k+1$ out of two order k trees by simply appending the root of the second to the end of the first root's sequence of children. A related property: each binomial tree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

值得注意的一个特性--在接下来的段落中肯定会用到--是我们可以用两棵阶数为 k 的树组装出一棵阶数为 $k+1$ 的二叉树，只需将第二棵树的根添加到第一棵树根的子序列末尾即可。一个相关的特性是：每阶 k 的二叉树共有 2^{k} 个节点。

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 lexicographically greater than the “49”):

阶数为 k 的二叉树是一棵阶数为 k 的二叉树，在这棵树中，堆属性自始至终都被递归遵守，也就是说，每个节点中的值在词典上都小于或等于其子节点中的值。在二叉树只存储数字字符串的世界里，左边的二叉树也是二叉堆，而右边的二叉树不是（因为 "55 "在词法上大于 "49"）：



binomial tree: yes binomial heap: no!

二叉树：是 二叉堆：否！

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

现在，如果二叉堆可以支持优先级队列，那么二叉堆也可以。但优先级队列所持有的元素数不能被限制为某个 2 的幂次

all the time. So priority queues, when backed by binomial heaps, are really backed by a vector of them.

一直存在。因此，优先队列由二叉堆支持时，实际上是由一个二叉堆向量支持。

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_{10} is 1011_2 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!

如果一个优先级队列需要管理 11 个元素，那么它将保留三个阶数分别为 0、1 和 3 的二叉堆来存储 $2^0 + 2^1 + 2^3 = 1 + 2 + 8 = 11$ 元素。 11_{10} 的二进制表示为 1011_2 并非巧合。1011 中的 "1 "为整个数字贡献了 $2^3, 2^1$ 和 2^0 。这三个指数告诉我们需要什么样的二项式堆阶数才能容纳所有 11 个元素。真不错！

Binomial Heap Insert 二项式堆插入

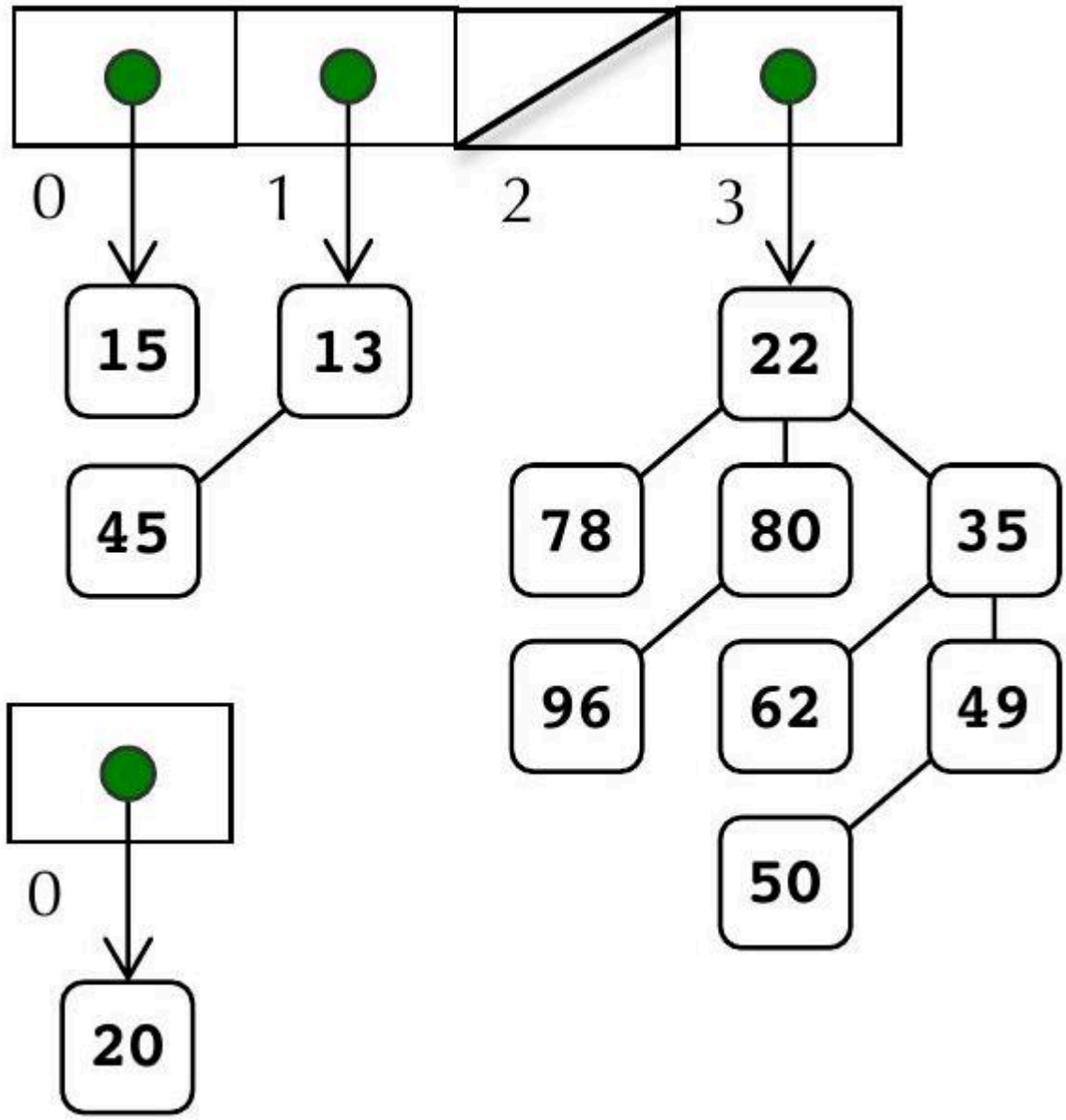
What happens when we introduce a new element into the mix? More formally: What happens when you enqueue a new string into the Vector<node *>-backed priority queue? Let’s see what happens when we enqueue a “20”.

当我们引入一个新元素时会发生什么？更正式地说：当你向矢量<node *>支持的优先级队列中queue一个新字符串时，会发生什么？让我们看看当我们枚举一个 "20 "时会发生什么。

The size of the priority queue goes from 11 to 12 -or rather, from 1011_2 to 1100_2. 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

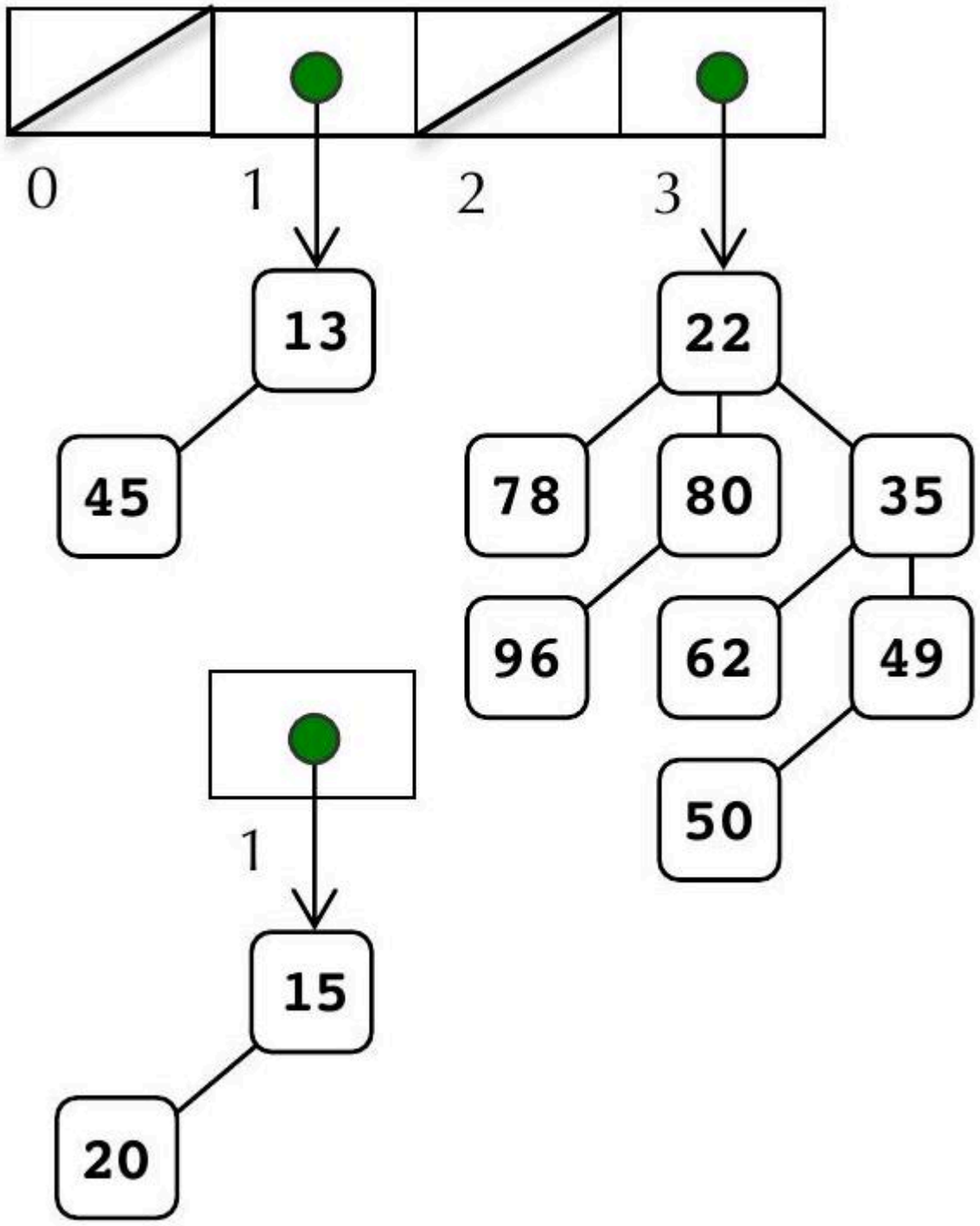
new element-a “20” in this example-and align it with the 0^{th} order entry of the Vector<node *>.

优先级队列的大小将从 11 变为 12，或者说，从 1011_2 变为 1100_2 。如果我们尽可能地模拟二进制加法，就会明白如何在每个二叉树中添加这个新元素，并始终保持堆排序属性。模拟的第一步是围绕新元素--本例中的 "20"--创建阶数为 0 的二叉树，并将其与 Vector 的 0^{th} 阶条目对齐。



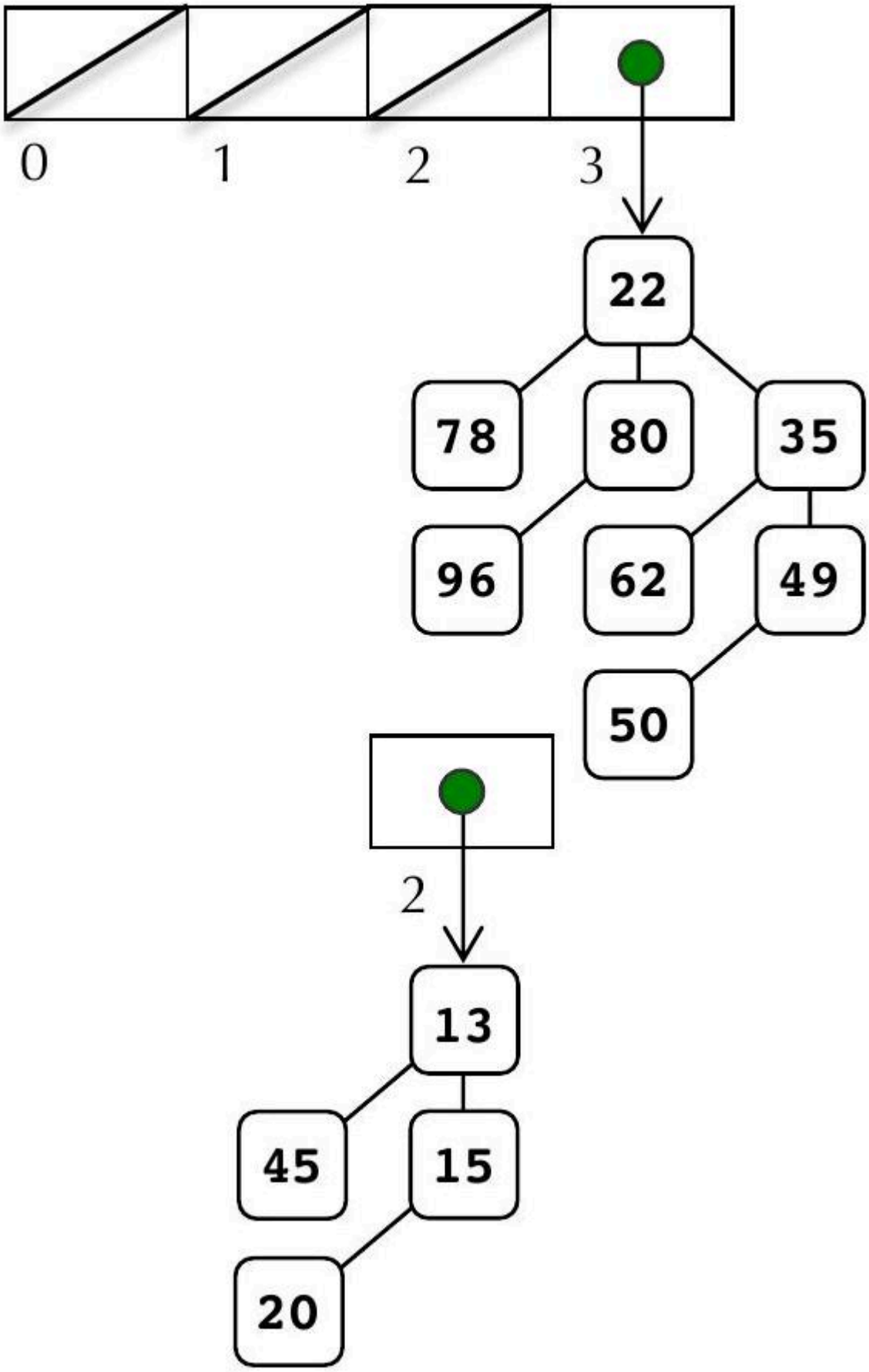
Now, when we add 1 and 1 in binary, we get 0 , and carry a 1 , right? We do the same thing when merging two order-0 binomial heaps, order-0 plus order-0 equal NULL, carry the order-1. One key difference: when you merge two order-0 heaps into the order-1 that gets carried, you need to make sure the heap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

现在，当我们在二进制中将 1 和 1 相加时，我们得到 0 ，并携带一个 1 ，对吗？我们在合并两个 0 阶二项式堆时也是这样做的，0 阶加 0 阶等于空，然后携带 1 阶。一个关键区别是：当把两个 0 阶堆合并为 1 阶堆时，需要确保合并时尊重堆属性。由于 15 小于 20，这就意味着 15 会得到一个 order-0 作为子堆，而这个 15 会成为 order- 1 的根。



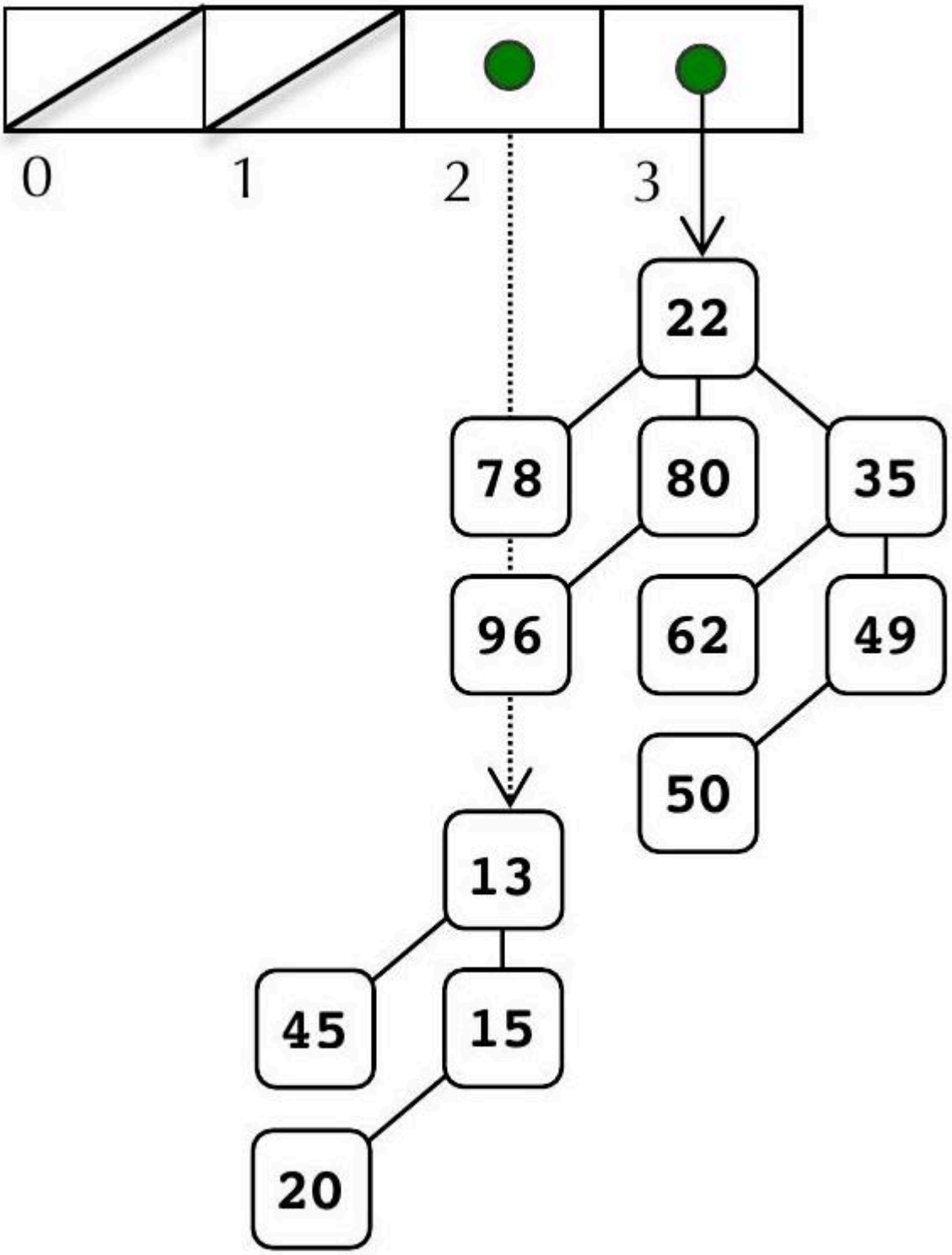
The carry now contributes to the merging at the order-1 level. The carry (with the 15 and the 20) and the original order-1 contribution (the one with the 13 and the 45) similarly merge to produce a NULL order-1 with an order-2 carry.

现在，进位有助于阶-1 级的合并。进位（包含 15 和 20）和原来的阶-1 级贡献（包含 13 和 45）同样合并，产生一个包含阶-2 进位的 NULL 阶-1。



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:

如果原图中有一个阶 2，级联合并就会继续。但是，由于原图中没有阶次 2 的二叉堆，阶次 2 的携带就可以占据集合中的这个位置，级联就可以结束。在我们的例子中，原来的二叉堆集合将转化为



Binomial Heap Merge 二项式堆合并

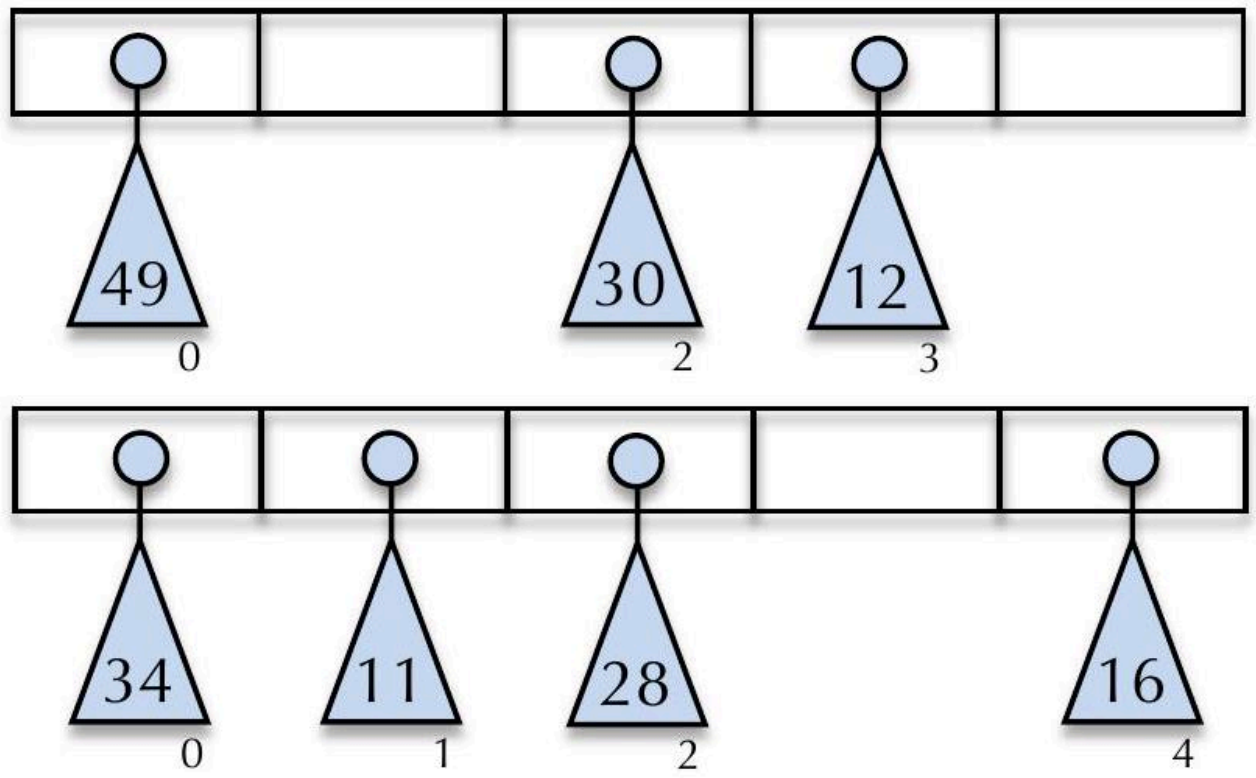
The primary perk the binomial heap has over the more standard binary heap is that it, if properly implemented, supports merge much more quickly. In fact, two binomial heaps as described above can be merged in $O(\lg n)$ time, where n is the size of the larger binomial heap.

与更标准的二进制堆相比，二进制堆的主要优势在于，如果实现得当，它可以更快地支持合并。事实上，上述两个二叉堆可以在 $O(\lg n)$ 时间内合并，其中 n 是较大二叉堆的大小。

You can merge two heaps using an extension of the binary addition emulated while discussing 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:

在讨论 enqueue 时，我们可以使用二进制加法的扩展来合并两个堆。事实证明，enqueue 一个节点实际上等同于合并一个任意大的二叉堆和一个大小为 1 的二叉堆。通用合并问题关注的是两个任意大小的二叉堆的统一。因此，为了便于说明，假设我们要合并两个大小分别为 13 和 23 的二叉堆，如下所示：



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 pictures.)

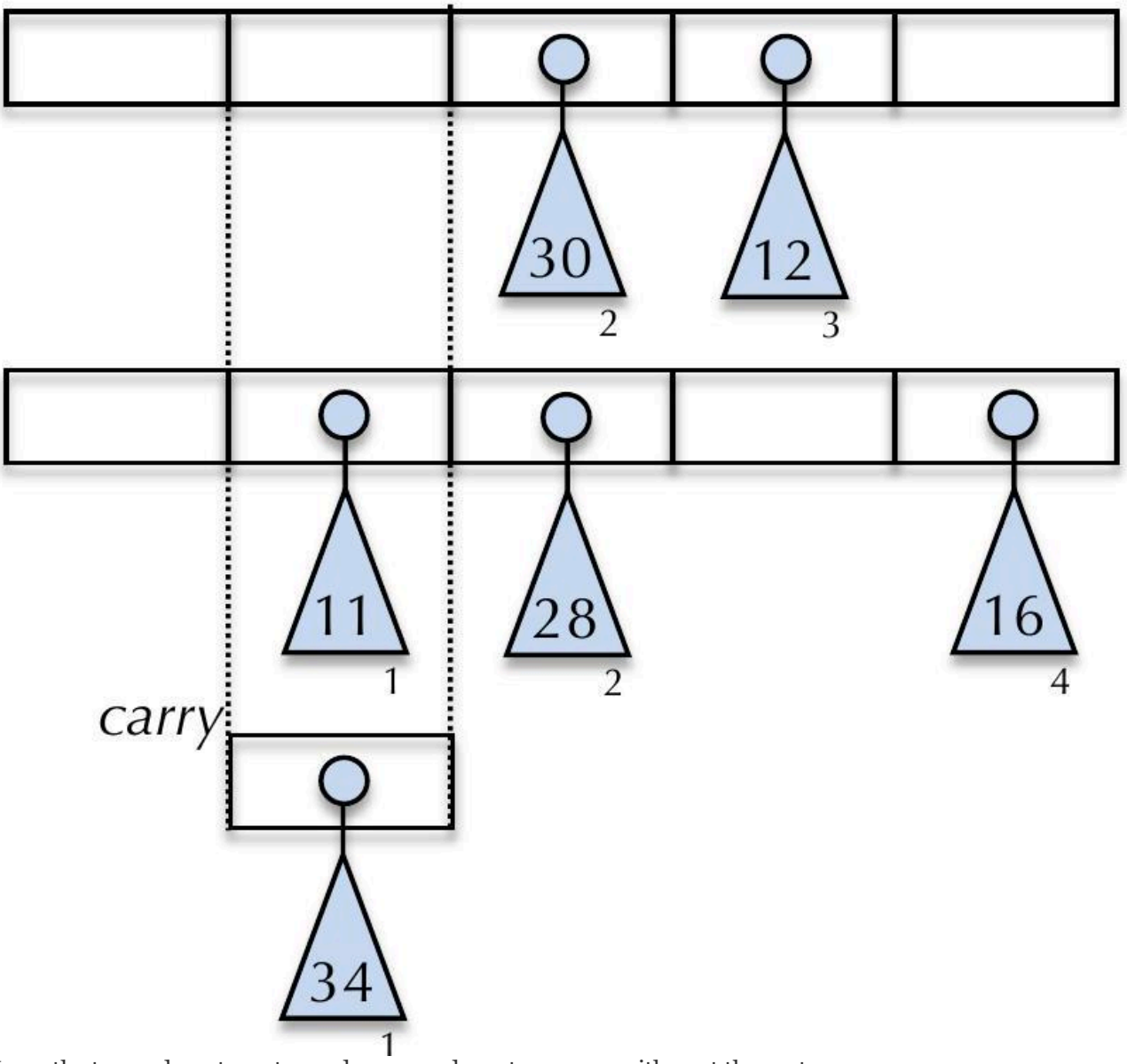
三角形代表尊重堆排序属性的二叉树，下标代表它们的顺序。三角形内的数字是根值--树中最小的值，空白代表 NULL。(我们没有足够的空间来展示用于说明 enqueue 的更复杂的图片，所以我使用了更骨架化的图片）。

To merge is to emulate binary arithmetic, understanding that the 0 s and 1 s of pure binary math have been upgraded to be NULLs 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:

合并就是模拟二进制运算，纯二进制数学中的 0 和 1 已升级为 NULL 和二叉树根地址。合并从任何 0 阶树开始，然后从低阶到高阶--在图中从左到右依次进行。这种特殊的合并（从图形上将第二阶合并到第一阶）可以用动画的形式逐一播放：

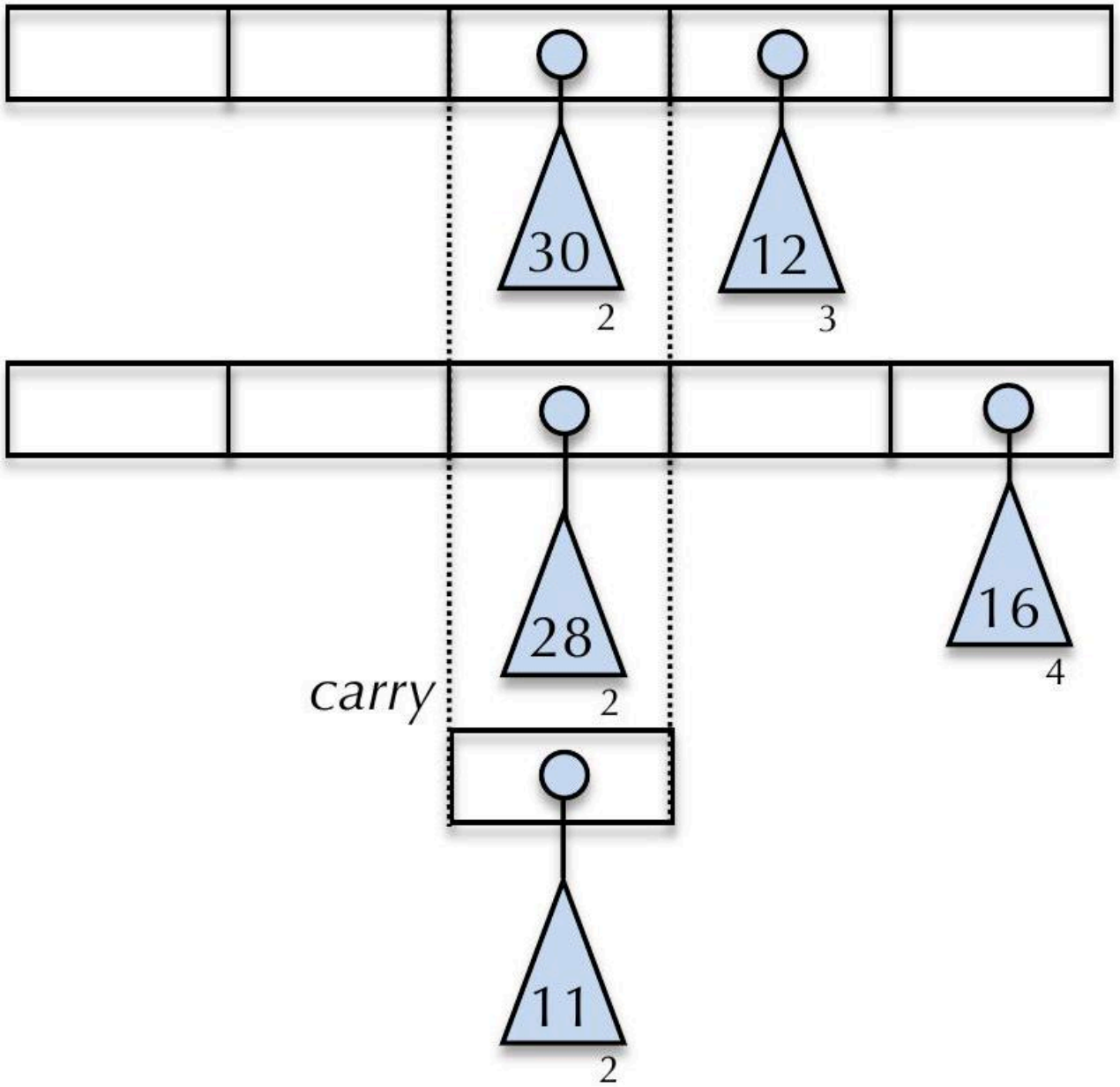
Merge the two order-0 trees to produce an order-1 (with 34 at the root) that carries.

合并两棵 0 号树，生成一棵 1 号树（根部为 34 号树）。



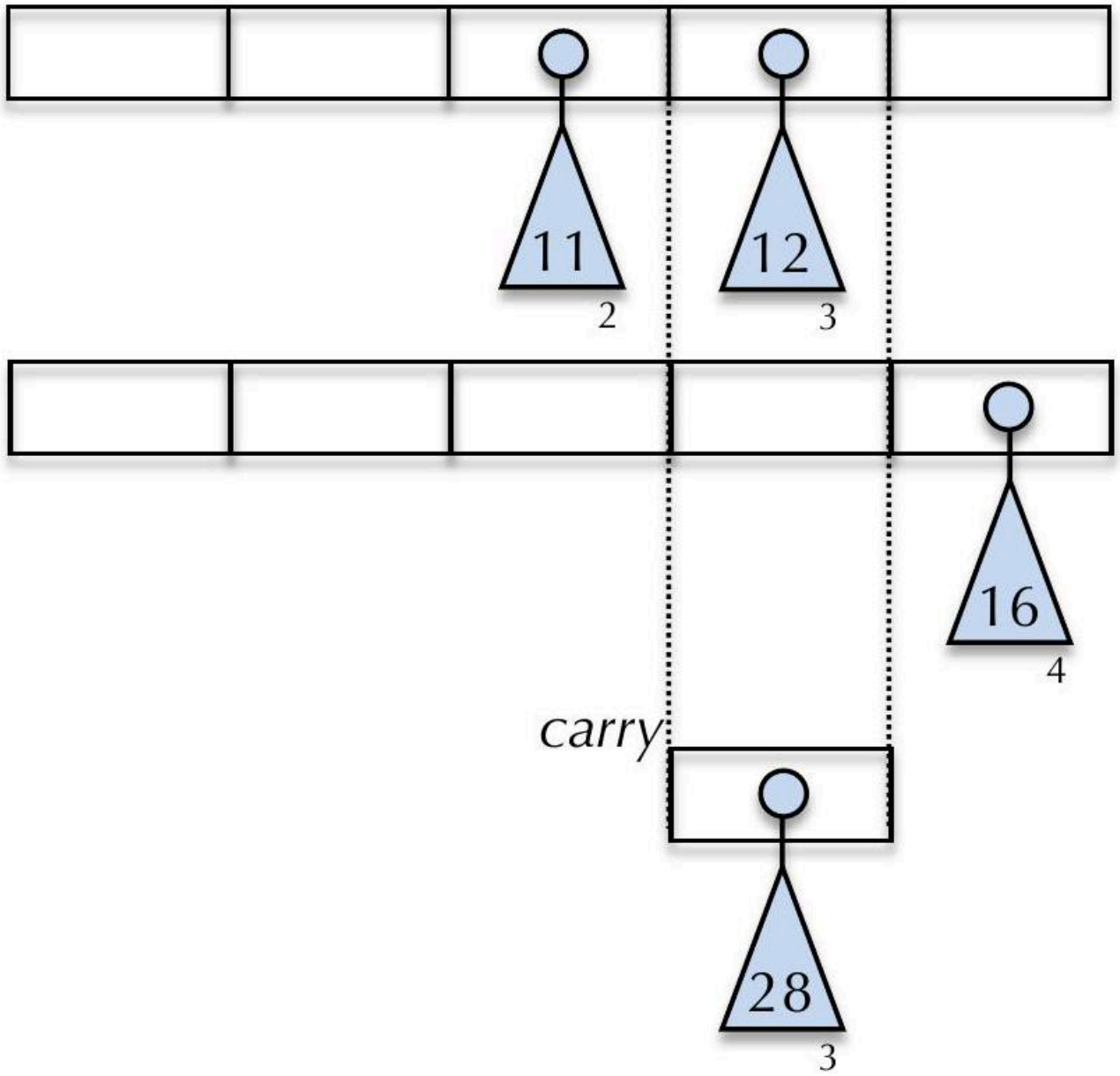
Merge the two order-1 trees to produce an order-2 tree carry, with 11 at the root.

合并两棵阶 1 树，生成一棵阶 2 树，树根为 11。



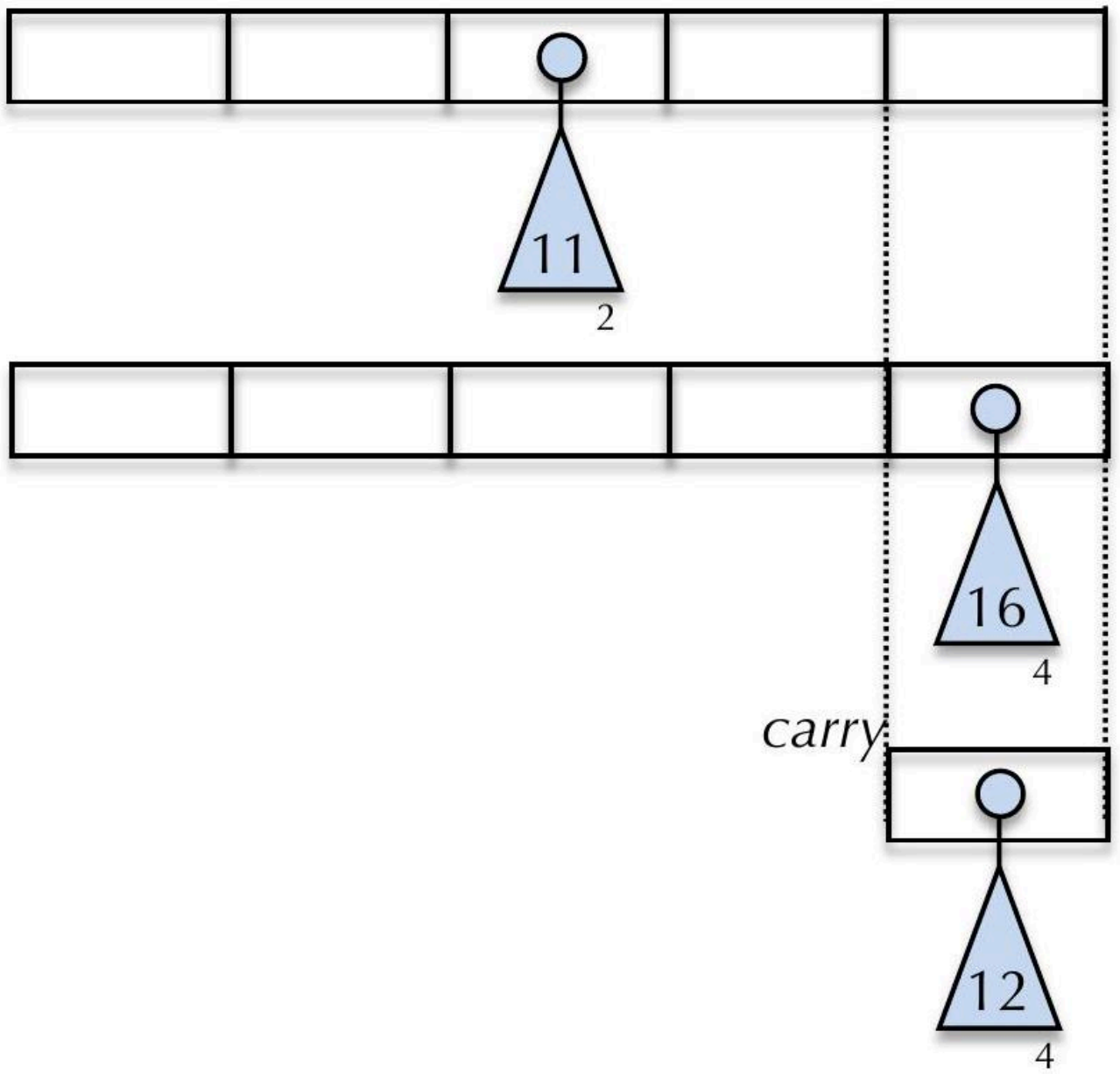
Merge the three (three!) order-2 trees! Leave one of the three alone (we'll leave the 11 in place, though it could have been any of the three) and merge the other two to produce an order-3 (with the smaller of 28 and 30 as the root).

合并三棵（三棵！）阶次 2 树！保留三棵树中的一棵（我们保留 11 号树，尽管它可能是三棵树中的任何一棵），合并另外两棵树，生成一棵阶 3 树（以 28 和 30 中较小的一棵为根）。



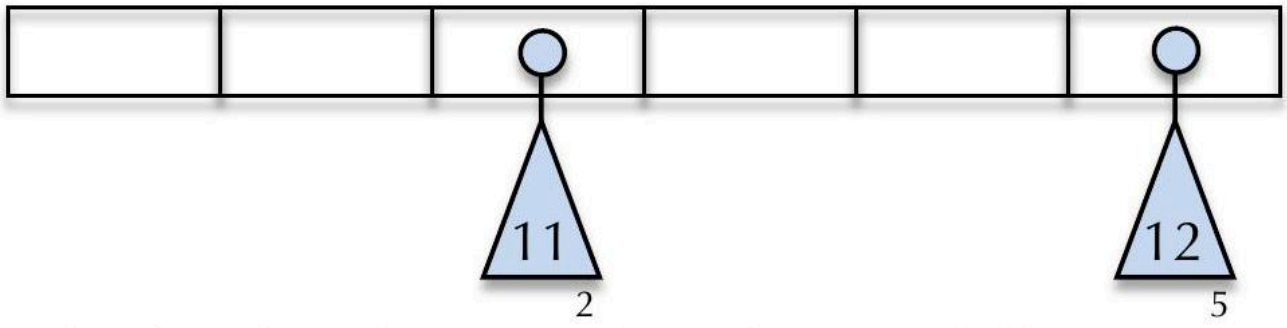
Merge the two order-3 trees to produce an order-4 tree carry, with 12 as the root:

合并两棵阶 3 树，生成以 12 为根的阶 4 树：



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.

最后，合并两个阶 4，形成一棵阶 5 树，12 位于树根。因为这显然是最后一次合并，所以我们只画最后一张图。



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 .

上文反映了这样一个事实，即合并后的乘积应该有 $13 + 23$ 等于 36_{10} 等于 100100_2 的元素，而事实也确实如此：阶 2 树包含 4 个元素，阶 5 树包含 32 个元素。

Binomial Heap Peek and extractMin

二叉堆偷窥和 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.

peek 可以用一个简单的 for 循环来实现，循环遍历集合中的所有二叉堆，并返回遇到的所有根元素中最小的元素（运行时间为 $O(\lg n)$ ）。当然，容纳最小元素的二叉堆必须确保它的所有子元素都被正确地重新并入数据结构，而不会成为孤儿。如上图所示，这些子堆会以与第二个二叉堆相同的方式合并到剩余结构中。

Binomial Heap Implementation Notes

二项式堆实施说明

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 start typing, hoping it'll all just work out. You'll only succeed 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 operations that update the binomial heap itself.

在编码之前先思考。关于二进制堆，我们说过同样的话，但在这里更为重要。你不能假装了解二叉堆，然后开始键入，希望一切都能顺利进行。只有清楚地了解 enqueue、merge 和 extractMin 的工作原理，并编写与之相一致的代码，才能取得成功。尤其是，在处理任何更新二叉堆本身的操作之前，你绝对必须理解上述的一般合并操作。

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:

结合使用内置结构和自定义结构。二叉堆中的每个节点都应使用类似下面这样的数据结构来建模：

```
struct node {
    string elem;
    Vector<node *> children;
};
```

As opposed to the binary heap, the binomial heap is complicated 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.

与二进制堆相比，二项式堆足够复杂，你需要依靠合理选择的内置程序，并在其上进行分层。我们鼓励你在实现时使用上述节点类型，只有在有充分理由的情况下才偏离它。

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.

释放内存您有责任释放堆分配的内存。在操作过程中，您的实现不应释放任何内存，析构函数应释放对象的所有内部内存。

Accessing files 访问文件

We've assembled a Qt Creator project for you already, and that project encapsulates many interface (**h**) and implementation (. **cpp**) files.

我们已经为您创建了一个 Qt Creator 项目，该项目封装了许多接口 (**h**) 和实现 (. **cpp**) 文件。

pqueue-test.cpp pqueue pqueue-test.cpp pqueue	Test harness to assist in development and testing. Interface and partial implementation of base 协助开发和测试的测试线束。接口和部分实施基础
	PQueue class. The primary purpose of the PQueue class is to define the interface that all concrete priority queue implementations should agree on. PQueue 类。PQueue 类的主要目的是定义所有具体的优先级队列实现都应同意的接口。
pqueue-vector	Interface and implementation file housing the unsorted vector version of the priority queue. 容纳无排序矢量版优先级队列的接口和实现文件。
pqueue-heap	Interface and implementation file housing the version of the priority queue that's backed by the array-packed binary heap. 接口和实现文件容纳优先级队列的版本，该版本由数组堆栈二进制堆支持。
pqueue-linked-list	Interface and implementation file housing the sorted, doubly linked list version of the priority queue. 为优先级队列的排序双链表版本提供接口和实现文件。
pqueue-binomial-heap	Interface and implementation file housing the version of the priority queue that's backed by binomial heaps. 接口和实现文件，容纳由二叉堆支持的优先级队列版本。