

As this example illustrates, a heap can contain multiple entries with the same key. (After all, in a typical simulation, we can't very well outlaw multiple events happening at the same time.)

When we finish, is the heap-order property satisfied? Yes, if the heap-order property was satisfied before the insertion. Let's look at a typical exchange of x with a parent p (right) during the insertion operation. Since the heap-order property was satisfied before the insertion, we know that $p \leq s$ (where s is x 's sibling), $p \leq l$, and $p \leq r$ (where l and r are x 's children). We only swap if $x < p$, which implies that $x < s$; after the swap, x is the parent of s . After the swap, p is the parent of l and r . All other relationships in the subtree rooted at x are maintained, so after the swap, the tree rooted at x has the heap-order property.

For maximum speed, don't put x at the bottom of the tree and bubble it up. Instead, bubble a hole up the tree, then fill in x . This modification saves the time that would be spent setting a sequence of references to x that are going to change anyway.

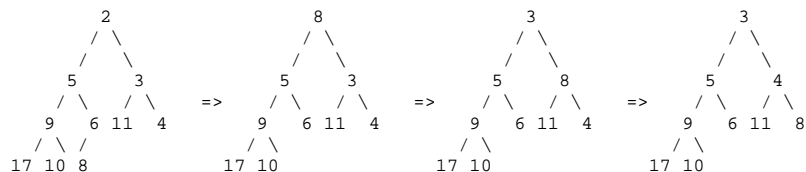
Goodrich & Tamassia have `insert()` return an `Entry` object representing (k, v) . I don't know why.

```
[3] Entry removeMin();
```

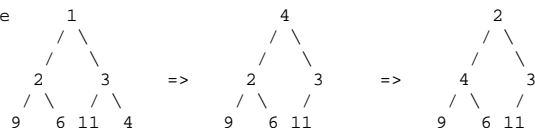
If the heap is empty, return null or throw an exception. Otherwise, begin by removing the entry at the root node and saving it for the return value. This leaves a gaping hole at the root. We fill the hole with the last entry in the tree (which we call " x "), so that the tree is still complete.

It is unlikely that x has the minimum key. Fortunately, both subtrees rooted at the root's children are heaps, and thus the new minimum key is one of these two children. We bubble x down the heap as follows: if x has a child whose key is smaller, swap x with the child having the minimum key. Next, compare x with its new children; if x still violates the heap-order property, again swap x with the child with the minimum key. Continue until x is less than or equal to its children, or reaches a leaf.

Consider running `removeMin()` on our original tree.



Above, the entry bubbled all the way to a leaf. This is not always the case, as the example at right shows.



For maximum speed, don't put x at the root and bubble it down. Instead, bubble a hole down the tree, then fill in x .

Running Times

There are other, less efficient ways we could implement a priority queue than using a heap. For instance, we could use a list or array, sorted or unsorted. The following table shows running times for all, with n entries in the queue.

	Binary Heap	Sorted List/Array	Unsorted List/Array
<code>min()</code>	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
<code>insert()</code>			
worst-case	$\Theta(\log n)$ *	$\Theta(n)$	$\Theta(1)$ *
best-case	$\Theta(1)$ *	$\Theta(1)$ *	$\Theta(1)$ *
<code>removeMin()</code>			
worst-case	$\Theta(\log n)$	$\Theta(1)$ **	$\Theta(n)$
best-case	$\Theta(1)$	$\Theta(1)$ **	$\Theta(n)$

* If you're using an array-based data structure, these running times assume that you don't run out of room. If you do, it will take $\Omega(n)$ time to allocate a larger array and copy the entries into it. However, if you double the array size each time, the `_average_` running time will still be as indicated.

** Removing the minimum from a sorted array in constant time is most easily done by keeping the array always sorted from largest to smallest.

In a binary heap, `min`'s running time is clearly in $\Theta(1)$.

`insert()` puts an entry x at the bottom of the tree and bubbles it up. At each level of the tree, it takes $O(1)$ time to compare x with its parent and swap if indicated. An n -node complete binary tree has $\lceil \log_2 n \rceil$ levels. In the worst case, x will bubble all the way to the top, taking $\Theta(\log n)$ time.

Similarly, `removeMin` may cause an entry to bubble all the way down the heap, taking $\Theta(\log n)$ worst-case time.

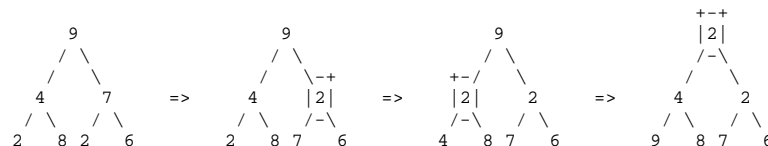
Bottom-Up Heap Construction

Suppose we are given a bunch of randomly ordered entries, and want to make a heap out of them. We could insert them one by one in $O(n \log n)$ time, but there's a faster way. We define one more heap operation.

```
[4] void bottomUpHeap();
```

First, we make a complete tree out of the entries, in any order. (If we're using an array representation, we just throw all the entries into an array.) Then we work backward from the last internal node (non-leaf node) to the root node, in reverse order in the array or the level-order traversal. When we visit a node this way, we bubble its entry down the heap as in `removeMin()`.

Before we bubble an entry down, we know (inductively) that its two child subtrees are heaps. Hence, by bubbling the entry down, we create a larger heap rooted at the node where that entry started.



The running time of `bottomUpHeap` is tricky to derive. If each internal node bubbles all the way down, then the running time is proportional to the sum of the heights of all the nodes in the tree. Page 371 of Goodrich and Tamassia has a simple and elegant argument showing that this sum is less than n , where n is the number of entries being coalesced into a heap. Hence, the running time is in $\Theta(n)$, which beats inserting n entries into a heap individually.

Postscript: Other Types of Heaps (not examinable)

Binary heaps are not the only heaps in town. Several important variants are called "mergeable heaps", because it is relatively fast to combine two mergeable heaps together into a single mergeable heap. We will not describe these complicated heaps in CS 61B, but it's worthwhile for you to know they exist in case you ever need one.

The best-known mergeable heaps are called "binomial heaps," "Fibonacci heaps," "skew heaps," and "pairing heaps." Fibonacci heaps have another remarkable property: if you have a reference to an arbitrary node in a Fibonacci heap, you can decrease its key in constant time. (Pairing heaps are suspected of having the same property, but nobody knows for sure.) This operation is used frequently by an important algorithm for finding the shortest path in a graph. The following running times are all worst-case.

	Binary	Binomial	Skew	Pairing	Fibonacci
insert()	$O(\log n)$	$O(\log n)$	$O(1)$	$O(\log n) *$	$O(1)$
removeMin()	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
merge()	$O(n)$	$O(\log n)$	$O(1)$	$O(\log n) *$	$O(1)$
decreaseKey()	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n) *$	$O(1)$

* Conjectured to be $O(1)$, but nobody has proven or disproven it.

The time bounds given here for skew heaps, pairing heaps, and Fibonacci heaps are "amortized" bounds, not worst case bounds. This means that, if you start from an empty heap, any sequence of operations will take no more than the given time bound on average, although individual operations may occasionally take longer. We'll discuss amortized analysis late this semester.