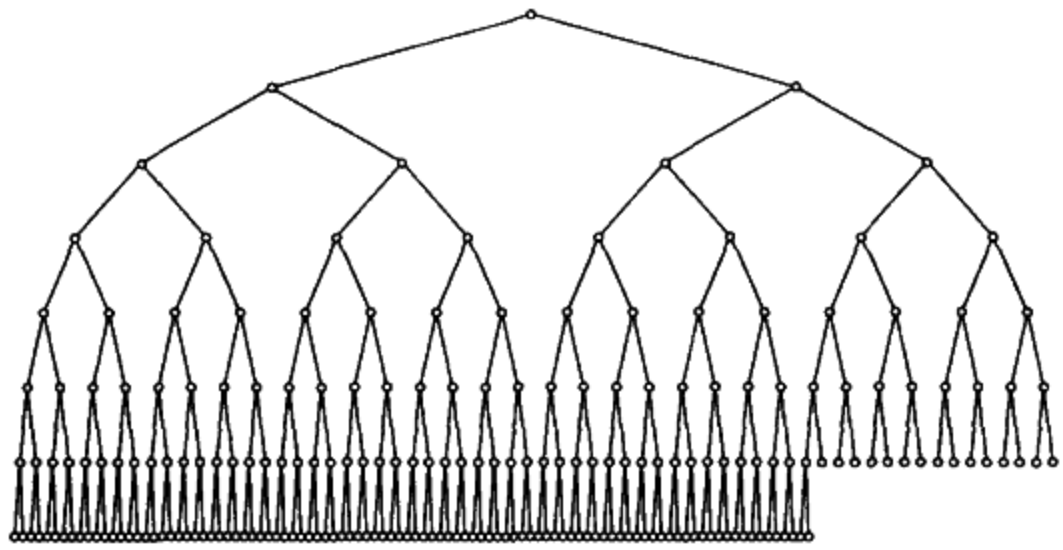

图 6-13 一棵巨大的完全二叉树

如果我们假设通过某种其他方法得知每一个元素的位置，那么就有几种其他操作的开销变小。下述前三种操作均以对数最坏情形时间运行。

decreaseKey(降低关键字的值)

decreaseKey(p, Δ)操作降低在位置 p 处的项的值，降值的幅度为正的量 Δ。由于这可能破坏堆序性质，因此必须通过上滤对堆进行调整。该操作对系统管理员是有用的：系统管理员能够使他们的程序以最高的优先级来运行。

increaseKey(增加关键字的值)

increaseKey(p, Δ)操作增加在位置 p 处的项的值，增值的幅度为正的量 Δ。这可以用下滤来完成。许多调度程序自动地降低正在过多地消耗 CPU 时间的进程的优先级。

delete(删除)

delete(p)操作删除堆中位置 p 上的节点。该操作通过首先执行 decreaseKey(p, ∞)然后再执行 deleteMin()来完成。当一个进程被用户中止(而不是正常终止)时，它必须从优先队列中除去。

buildHeap(构建堆)

有时二叉堆是由一些项的初始集合构造而得。这种构造方法以 N 项作为输入，并把它们放到一个堆中。显然，这可以使用 N 个相继的 insert 操作来完成。由于每个 insert 将花费 $O(1)$ 平均时间以及 $O(\log N)$的最坏情形时间，因此该算法的总的运行时间是 $O(N)$平均时间而不是 $O(N \log N)$最坏情形时间。由于这是一种特殊的指令，没有其他操作干扰，而且我们已经知道该指令能够以线性平均时间来执行，因此，期望能够保证线性时间界的考虑是合乎情理的。

一般的算法是将 N 项以任意顺序放入树中，保持结构特性。此时，如果 percolateDown(i)从节点 i 下滤，那么图 6-14 中的 buildHeap 程序则可以由构造方法用于创建一棵堆序的树(heap-ordered tree)。

图 6-15 中的第一棵树是无序树。从图 6-15 到图 6-18 中其余 7 棵树表示出 7 个 percolateDown 中每一个的执行结果。每条虚线对应两次比较：一次是找出较小的儿子节点，另一个是较小的儿子与该节点的比较。注意，在整个算法中只有 10 条虚线(可能已经存在第 11 条——在哪里?)，它们对应 20 次比较。

为了确定 buildHeap 的运行时间的界，我们必须确定虚线的条数的界。这可以通过计算堆中所有节点的高度的和来得到，它是虚线的最大条数。现在我们想要说明的是：该和为 $O(N)$。