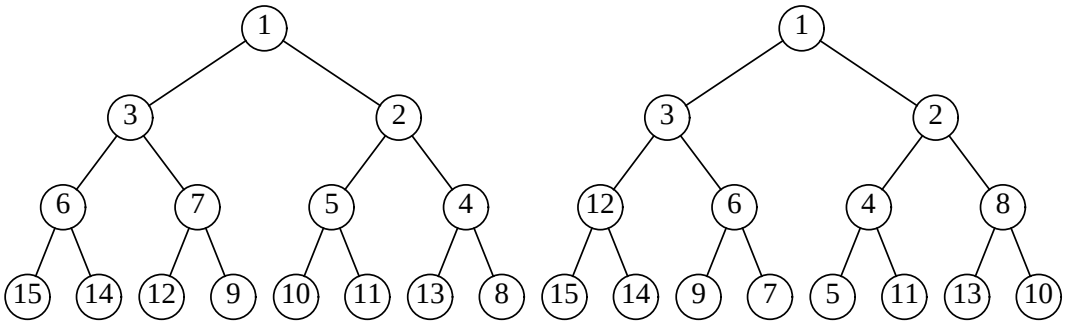
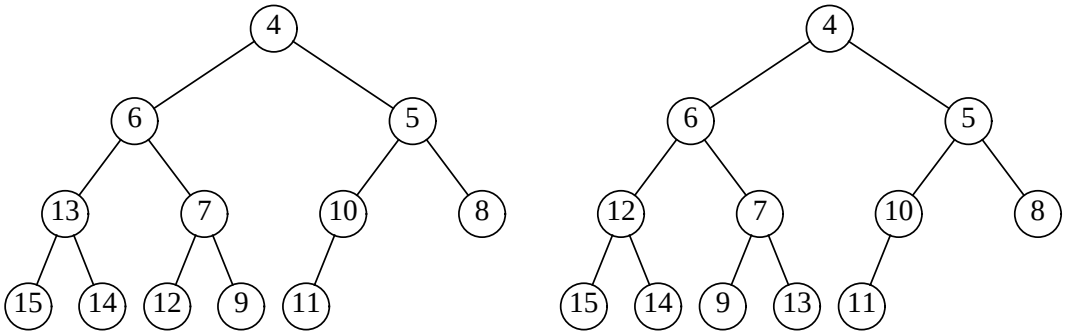
Chapter 6: Priority Queues (Heaps)

6.1 Yes. When an element is inserted, we compare it to the current minimum and change the minimum if the new element is smaller. *DeleteMin* operations are expensive in this scheme.

6.2



6.3 The result of three *DeleteMins,* starting with both of the heaps in Exercise 6.2, is as follows:



6.4

6.5 These are simple modifications to the code presented in the text and meant as programming exercises.

6.6 225. To see this, start with $i=1$ and position at the root. Follow the path toward the last node, doubling i when taking a left child, and doubling i and adding one when taking a right child.

6.7 (a) We show that $H(N)$, which is the sum of the heights of nodes in a complete binary tree of N nodes, is $N - b(N)$, where $b(N)$ is the number of ones in the binary representation of N. Observe that for $N = 0$ and $N = 1$, the claim is true. Assume that it is true for values of k up to and including $N-1$. Suppose the left and right subtrees have L and R nodes, respectively. Since the root has height $\lfloor \log N \rfloor$, we have

$$\begin{aligned} H(N) &= \lfloor \log N \rfloor + H(L) + H(R) \\ &= \lfloor \log N \rfloor + L - b(L) + R - b(R) \\ &= N - 1 + (\lfloor \log N \rfloor - b(L) - b(R)) \end{aligned}$$

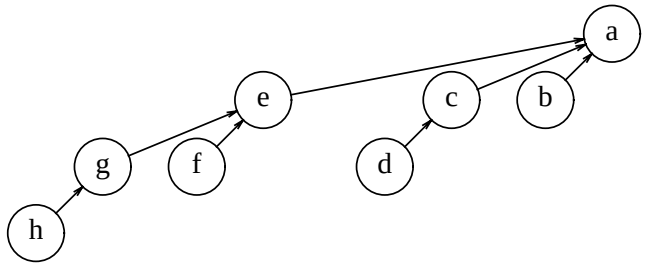
The second line follows from the inductive hypothesis, and the third follows because $L + R = N - 1$. Now the last node in the tree is in either the left subtree or the right subtree. If it is in the left subtree, then the right subtree is a perfect tree, and $b(R) = \lfloor \log N \rfloor - 1$. Further, the binary representation of N and L are identical, with the exception that the leading 10 in N becomes 1 in L. (For instance, if N = 37 = **10**0101, L = **1**0101.) It is clear that the second digit of N must be zero if the last node is in the left subtree. Thus in this case, $b(L) = b(N)$, and

$$H(N) = N - b(N)$$

If the last node is in the right subtree, then $b(L) = \lfloor \log N \rfloor$. The binary representation of R is identical to N, except that the leading 1 is not present. (For instance, if N = 27 = **1**01011, L = 01011.) Thus $b(R) = b(N) - 1$, and again

$$H(N) = N - b(N)$$

(b) Run a single-elimination tournament among eight elements. This requires seven comparisons and generates ordering information indicated by the binomial tree shown here.



The eighth comparison is between b and c. If c is less than b, then b is made a child of c. Otherwise, both c and d are made children of b.

(c) A recursive strategy is used. Assume that $N = 2^k$. A binomial tree is built for the N elements as in part (b). The largest subtree of the root is then recursively converted into a binary heap of 2^{k-1} elements. The last element in the heap (which is the only one on an extra level) is then inserted into the binomial queue consisting of the remaining binomial trees, thus forming another binomial tree of 2^{k-1} elements. At that point, the root has a subtree that is a heap of $2^{k-1} - 1$ elements and another subtree that is a binomial tree of 2^{k-1} elements. Recursively convert that subtree into a heap; now the whole structure is a binary heap. The running time for $N = 2^k$ satisfies $T(N) = 2T(N/2) + \log N$. The base case is $T(8) = 8$.

6.8 Let $D_1, D_2, ..., D_k$ be random variables representing the depth of the smallest, second smallest, and k^{th} smallest elements, respectively. We are interested in calculating $E(D_k)$. In what follows, we assume that the heap size N is one less than a power of two (that is, the bottom level is completely filled) but sufficiently large so that terms bounded by $O(1/N)$ are negligible. Without loss of generality, we may assume that the k^{th} smallest element is in the left subheap of the root. Let $p_{j,k}$ be the probability that this element is the j^{th} smallest element in the subheap.

Lemma: For $k > 1$, $E(D_k) = \sum_{j=1}^{k-1} p_{j,k}(E(D_j) + 1)$.

Proof: An element that is at depth d in the left subheap is at depth $d+1$ in the entire subheap. Since $E(D_j + 1) = E(D_j) + 1$, the theorem follows. □

Since by assumption, the bottom level of the heap is full, each of second, third, ..., $k-1^{th}$ smallest elements are in the left subheap with probability of 0.5. (Technically, the probability should be $½ - 1/(N-1)$ of being in the right subheap and $½ + 1/(N-1)$ of being in the left, since we have already placed the k^{th} smallest in the right. Recall that we have assumed that terms of size $O(1/N)$ can be ignored.) Thus

$$p_{j,k} = p_{k-j,k} = \frac{1}{2^{k-2}} \binom{k-2}{j-1}$$

Theorem: $E(D_k) \le \log k$.

Proof: The proof is by induction. The theorem clearly holds for $k = 1$ and $k = 2$. We then show that it holds for arbitrary $k > 2$ on the assumption that it holds for all smaller k. Now, by the inductive hypothesis, for any $1 \le j \le k-1$,

$$E(D_j) + E(D_{k-j}) \le \log j + \log k-j$$

Since $f(x) = \log x$ is convex for $x > 0$,

$$\log j + \log k-j \le 2\log(k/2)$$

Thus

$$E(D_j) + E(D_{k-j}) \le \log(k/2) + \log(k/2)$$

Furthermore, since $p_{j,k} = p_{k-j,k}$,

$$p_{j,k}E(D_j) + p_{k-j,k}E(D_{k-j}) \le p_{j,k}\log(k/2) + p_{k-j,k}\log(k/2)$$

From the lemma,

$$\begin{aligned} E(D_k) &= \sum_{j=1}^{k-1} p_{j,k}(E(D_j) + 1) \\ &= 1 + \sum_{j=1}^{k-1} p_{j,k}E(D_j) \end{aligned}$$

Thus

$$E(D_k) \le 1 + \sum_{j=1}^{k-1} p_{j,k}\log(k/2)$$

$$\leq 1 + \log (k/2) \sum_{j=1}^{k-1} p_{j,k}$$

$$\leq 1 + \log (k/2)$$

$$\leq \log k$$

completing the proof. □

It can also be shown that asymptotically, $E(D_k) \simeq \log (k-1) - 0.273548$.

6.9 (a) Perform a preorder traversal of the heap.

(b) Works for leftist and skew heaps. The running time is $O(Kd)$ for d-heaps.

6.11 Simulations show that the linear time algorithm is the faster, not only on worst-case inputs, but also on random data.

6.12 (a) If the heap is organized as a (min) heap, then starting at the hole at the root, find a path down to a leaf by taking the minimum child. The requires roughly $\log N$ comparisons. To find the correct place where to move the hole, perform a binary search on the $\log N$ elements. This takes $O(\log \log N)$ comparisons.

(b) Find a path of minimum children, stopping after $\log N - \log \log N$ levels. At this point, it is easy to determine if the hole should be placed above or below the stopping point. If it goes below, then continue finding the path, but perform the binary search on only the last $\log \log N$ elements on the path, for a total of $\log N + \log \log \log N$ comparisons. Otherwise, perform a binary search on the first $\log N - \log \log N$ elements. The binary search takes at most $\log \log N$ comparisons, and the path finding took only $\log N - \log \log N$, so the total in this case is $\log N$. So the worst case is the first case.

(c) The bound can be improved to $\log N + \log^* N + O(1)$, where $\log^* N$ is the inverse Ackerman function (see Chapter 8). This bound can be found in reference [16].

6.13 The parent is at position $\lfloor (i + d - 2)/d \rfloor$. The children are in positions $(i - 1)d + 2$, ..., $id + 1$.

6.14 (a) $O((M + dN)\log_d N)$.

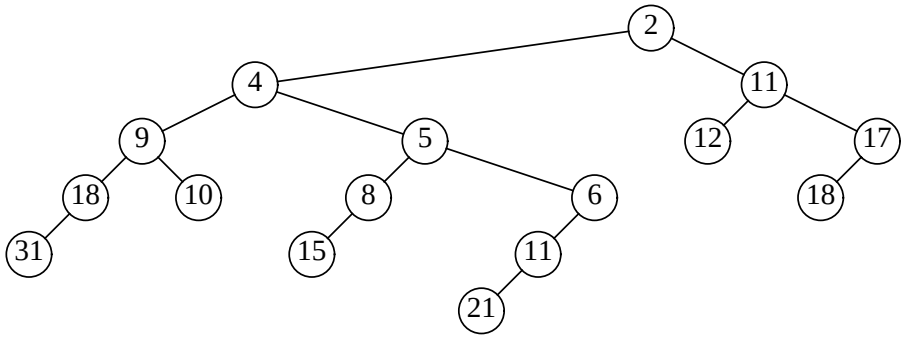
(b) $O((M + N)\log N)$.

(c) $O(M + N^2)$.

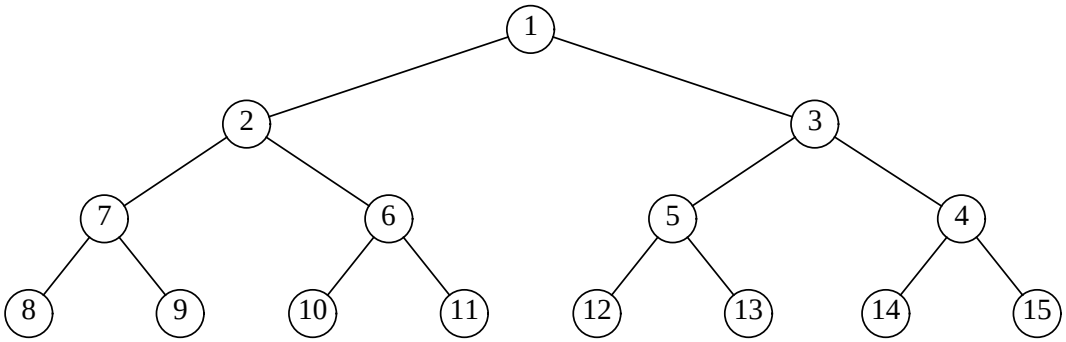
(d) $d = \max(2, M/N)$.

(See the related discussion at the end of Section 11.4.)

6.16



6.17



6.18 This theorem is true, and the proof is very much along the same lines as Exercise 4.17.

6.19 If elements are inserted in decreasing order, a leftist heap consisting of a chain of left children is formed. This is the best because the right path length is minimized.

6.20 (a) If a *DecreaseKey* is performed on a node that is very deep (very left), the time to percolate up would be prohibitive. Thus the obvious solution doesn't work. However, we can still do the operation efficiently by a combination of *Delete* and *Insert*. To *Delete* an arbitrary node x in the heap, replace x by the *Merge* of its left and right subheaps. This might create an imbalance for nodes on the path from x's parent to the root that would need to be fixed by a child swap. However, it is easy to show that at most $\log N$ nodes can be affected, preserving the time bound. This is discussed in Chapter 11.

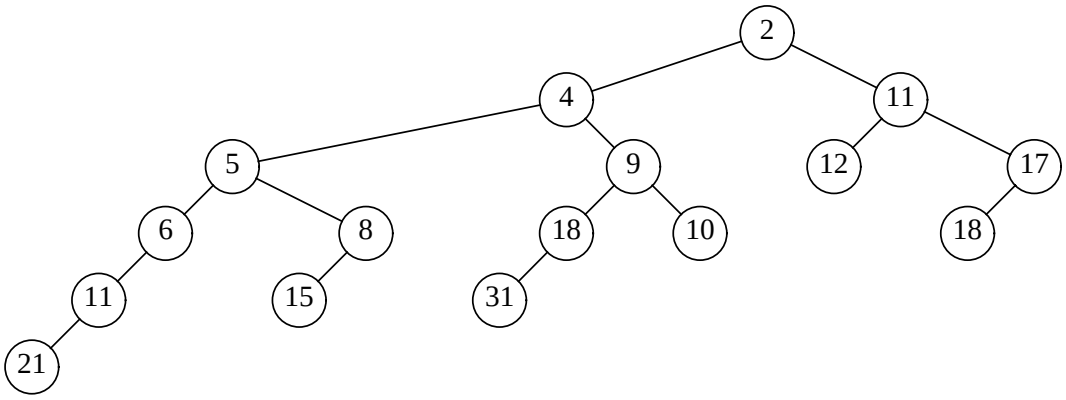
6.21 Lazy deletion in leftist heaps is discussed in the paper by Cheriton and Tarjan [9]. The general idea is that if the root is marked deleted, then a preorder traversal of the heap is formed, and the frontier of marked nodes is removed, leaving a collection of heaps. These can be merged two at a time by placing all the heaps on a queue, removing two, merging them, and placing the result at the end of the queue, terminating when only one heap remains.

6.22 (a) The standard way to do this is to divide the work into passes. A new pass begins when the first element reappears in a heap that is dequeued. The first pass takes roughly

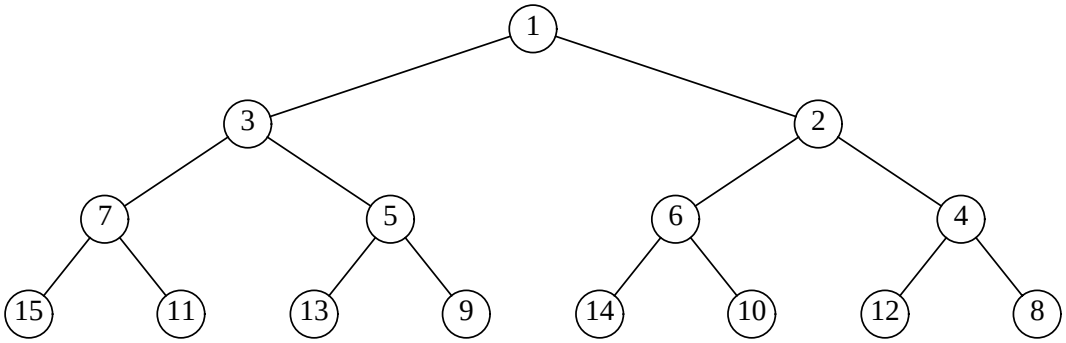
$2*1*(N/2)$ time units because there are $N/2$ merges of trees with one node each on the right path. The next pass takes $2*2*(N/4)$ time units because of the roughly $N/4$ merges of trees with no more than two nodes on the right path. The third pass takes $2*3*(N/8)$ time units, and so on. The sum converges to $4N$.

(b) It generates heaps that are more leftist.

6.23



6.24



6.25 This claim is also true, and the proof is similar in spirit to Exercise 4.17 or 6.18.

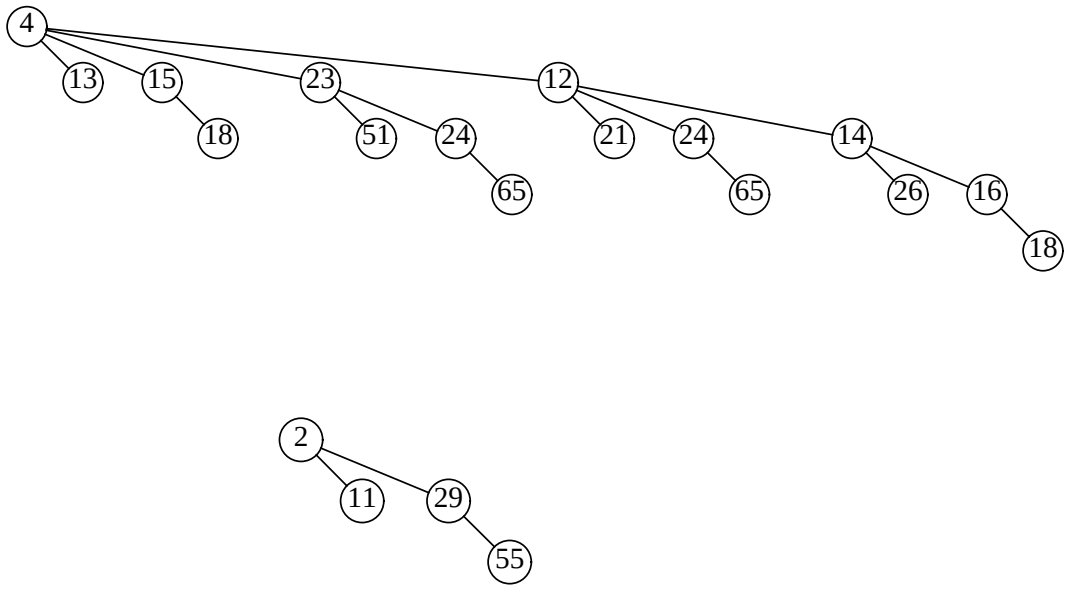
6.26 Yes. All the single operation estimates in Exercise 6.22 become amortized instead of worst-case, but by the definition of amortized analysis, the sum of these estimates is a worst-case bound for the sequence.

6.27 Clearly the claim is true for $k = 1$. Suppose it is true for all values $i = 1, 2, ..., k$. A B_{k+1} tree is formed by attaching a B_k tree to the root of a B_k tree. Thus by induction, it contains a B_0 through B_{k-1} tree, as well as the newly attached B_k tree, proving the claim.

6.28 Proof is by induction. Clearly the claim is true for $k = 1$. Assume true for all values $i = 1, 2, ..., k$. A B_{k+1} tree is formed by attaching a B_k tree to the original B_k tree. The original

thus had $\binom{k}{d}$ nodes at depth d. The attached tree had $\binom{k}{d-1}$ nodes at depth $d-1$, which are now at depth d. Adding these two terms and using a well-known formula establishes the theorem.

6.29



6.30 This is established in Chapter 11.

6.31 The algorithm is to do nothing special – merely *Insert* them. This is proved in Chapter 11.

6.35 Don't keep the key values in the heap, but keep only the difference between the value of the key in a node and the value of the key in its parent.

6.36 $O(N + k \log N)$ is a better bound than $O(N \log k)$. The first bound is $O(N)$ if $k = O(N / \log N)$. The second bound is more than this as soon as k grows faster than a constant. For the other values $\Omega(N / \log N) = k = o(N)$, the first bound is better. When $k = \Theta(N)$, the bounds are identical.