# THE POST-ORDER HEAP

*Nicholas J. A. Harvey and Kevin Zatloukal*
MIT Computer Science and Artificial Intelligence Laboratory
{*nickh,kevinz*}*@mit.edu*

*Abstract*

We propose the post-order heap, a new data structure for implementing priority queues. Our structure is a simple variant of the binary heap that requires only $\Theta(1)$ amortized time for *Insert* operations and $O(\log n)$ time in the worst case for *Delete-Min* operations. Like the binary heap, the post-order heap is an implicit data structure, meaning that a structure containing $n$ elements can be stored using only the first $n$ locations of an array and $\Theta(1)$ additional words of storage.

## 1. *Introduction*

The binary heap, due to Williams [4], is the canonical example of a priority queue. It supports the *Insert* and *Delete-Min* operations in worst-case $\Theta(\log n)$ time and also supports a batch insert operation called *Build-Heap*, which constructs a binary heap of $n$ elements in $\Theta(n)$ time. In this paper, we present the post-order heap, a variant of the binary heap, which performs insertions by executing *Build-Heap* online. This allows us to improve the cost of *Insert* to $\Theta(1)$ in the amortized sense, without affecting the cost of *Delete-Min*.

Post-order heaps are not the first priority queue to perform *Insert* in $\Theta(1)$ time and *Delete-Min* in $O(\log n)$ time. Fibonacci heaps [3] achieve these bounds in the amortized sense and support *Union* and *Decrease-Key* operations in $\Theta(1)$ amortized time. However, each node of a Fibonacci heap requires several pointers and additional state. In contrast, post-order heaps are implicit, requiring only $\Theta(1)$ storage in addition to an array of elements. Carlsson et al. [1] presented the first implicit priority queue to support *Insert* in $\Theta(1)$ time and *Delete-Min* in $O(\log n)$ time, both in the worst-case sense. Unlike their complex construction, however, post-order heaps are a natural extension of the binary heap that is both simple and practical.

## 2. *Overview*

The binary heap's *Insert* operation adds each new item at the bottom of the tree and then swaps the item *upward* until the heap order is satisfied. This makes the worst-case running time proportional to the depth of the inserted node, which is $\Theta(\log n)$ on average. On the other hand, the *Build-Heap* operation, due to Floyd [2], takes advantage of the insight that the average height of a node (i.e., distance to a leaf) is less than 2, whereas the average depth is $\Theta(\log n)$. The *Build-Heap* operation adds each new item at the top of a tree of previously inserted items and then swaps the item *downward* until the heap order is satisfied. This process of swapping an item downward in the tree is called a "heapify". The running time of a heapify is proportional to the height of the inserted node, which is $\Theta(1)$ on average.

As with binary heaps, the nodes of a post-order heap are organized into a partially-constructed, heap-ordered binary tree. Whereas binary heaps insert new nodes into this tree in the order given by a level-order traversal, post-order heaps insert new nodes according to a post-order traversal, as shown in Figure 1. Since each new node is the parent of two subtrees that have already been heapified, the post-order heap only needs to heapify at the new node to restore the heap order. Thus, insertion requires time proportional to the height of the inserted node, which is $\Theta(1)$ on average.

To store a heap in an array, we need to define a mapping between tree nodes and array locations. As with the binary heap, we define the mapping for a post-order heap to be the insertion order of the nodes. Thus, the labels on the nodes in Figure 1 also indicate the locations of those nodes in the array. Since our insertion order differs from that of a binary heap, the resulting data structure has many different properties. We examine some of these properties in the next section.

## 3. *Basic Properties*

We can think of the nodes in a binary heap of size $n$ as the first $n$ nodes visited by a level-order traversal of a perfect binary tree of infinite size, that is, a tree with a fixed root but growing infinitely downward. Similarly, we can think of the nodes in a post-order heap of size $n$ as the first $n$ nodes visited by a post-order traversal on a perfect binary tree $\mathcal{G}$ of infinite size. In this case, though, the tree $\mathcal{G}$ has a fixed left-most leaf and grows infinitely upward and to the right from that node.

The subgraph of $\mathcal{G}$ induced by these $n$ nodes is a forest $F_n$. For example, Figure 1 shows the forest $F_{24}$, which contains four trees. Let $T_n = \{t_1, ..., t_k\}$ be the trees of $F_n$ ordered by the roots' positions in the array. Each tree $t_i$ is a perfect binary tree since the post-order traversal visits a node only after completely visiting both of its subtrees. To simplify our notation, we let $t_i$ refer to both the tree and its root node.

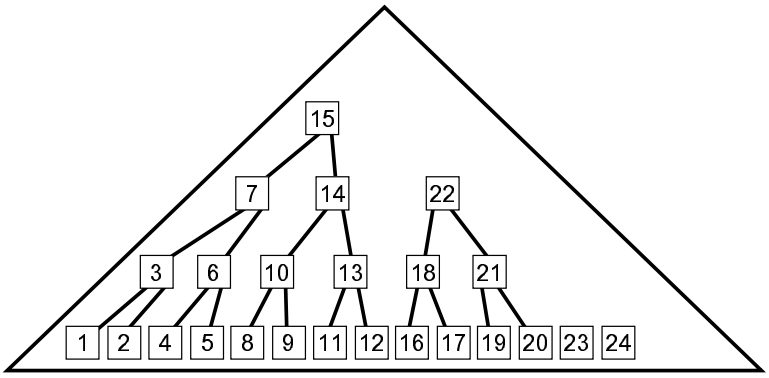


**Figure 1:** A post-order heap inserts nodes in the order given by a post-order traversal. The numbers in the nodes indicate the insertion order, not the actual values stored in those nodes. At this point, the heap consists of four trees. Notice that the last two trees are both of size 1. The next node inserted would be the parent of those two trees.

PROPOSITION 3.1 *For every $i < k$, the root $t_i$ is the left child of its parent in $\mathcal{G}$. That is, every root $t_i$ is a left child except possibly for $t_k$. For any pair $i < j$, $t_j$ is contained in the right subtree of $t_i$'s parent.*

*Proof*: Consider any root $t_i$ with parent $p_i$. Since $t_i$ is a root, we must not yet have inserted $p_i$. The post-order insertion rule at $p_i$ says to insert the left subtree, then the right subtree, and then $p_i$. If $t_i$ is right child of $p_i$, then $p_i$ would be the next node to be inserted after $t_i$. Since it has not been inserted, $t_i$ must be the last node in the array, which means $i = k$. If $i < k$, then $t_i$ must be a left child. Since $p_i$ has not been inserted, all of the nodes inserted after $t_i$ must be in the right subtree of $p_i$. □

COROLLARY 3.2 *The heights of the trees are decreasing, with equality only possible for $t_{k-1}$ and $t_k$.*

*Proof*: For any $t_i$ with $i < k$, $t_{i+1}, ..., t_k$ are contained in the right subtree of $t_i$'s parent, so clearly their heights are no greater than $t_i$'s. Equality is only possible if $t_{i+1}$ is the entire right subtree, in which case $t_{i+1}$ is a right child, so $i + 1 = k$. □

Since each tree $t_i$ is perfect, a tree of height $h$ has exactly $2^{h+1} - 1$ nodes. If a post-order heap of size $n$ consists of exactly one tree of height $h$, then we must have $n = 2^{h+1} - 1$ and $h = \lg(n + 1) - 1$. In general, if we know the number of nodes in a post-order heap, we can deduce the height of the first tree $t_1$.

DEFINITION 3.3 *Let* $s(h) = 2^{h+1} - 1$ *and* $h(n) = \lfloor \lg(n+1) \rfloor - 1$.

THEOREM 3.4 *In a post-order heap containing* $n$ *nodes, the height of* $t_1$ *is* $h(n)$.

*Proof*: We argued the case $k = 1$ above, so suppose that $k > 1$. Let $h_1$ be the height of $t_1$. Proposition 3.1 shows that the trees $t_2, ..., t_k$ are all contained in the right sibling of $t_1$. It follows that $t_1$ contains at least $n/2$ nodes: $s(h_1) = 2^{h_1+1} - 1 \geq n/2 \implies h_1 \geq \lg(n+2) - 2 > \lfloor \lg(n+1) \rfloor - 2$. Since $t_2$ exists, we know that $t_1$ contains fewer than $n$ nodes: $s(h_1) = 2^{h_1+1} - 1 < n \implies h_1 < \lg(n+1) - 1 \leq \lfloor \lg(n+1) \rfloor$. Since $h_1$ is an integer, we must have $h_1 = \lfloor \lg(n+1) \rfloor - 1 = h(n)$. Thus, tree $t_1$ has height $h(n)$. □

Corollary 3.2 implies that no tree is taller than $t_1$. Thus, Theorem 3.4 tells us that the maximum height of any tree is $h(n) = \lfloor \lg(n+1) \rfloor - 1$. Futhermore, since $t_i$ is strictly taller than $t_j$ if $i < j < k$, it follows that the number of trees, $k$, is at most $\lfloor \lg(n+1) \rfloor + 1$.

We now consider how to determine the height of an arbitrary node in a post-order heap. Let $v_i$ be the $i$-th node in the array, and let

$$H(i) = \begin{cases} h(i) & \text{if } i = s(h(i)) \\ H\big(i - s(h(i))\big) & \text{otherwise.} \end{cases}$$

THEOREM 3.5 $H(i)$ *is the height of node* $v_i$.

*Proof*: Imagine that $v_i$ has just been inserted. By Theorem 3.4, we can have $i = s(h(i))$ if and only if $t_1$ is the only tree. In that case, the post-order insertion rule tells us that $v_i$, the last node inserted, must be the root of $t_1$, whose height is $h(i)$. This proves the first case in the definition of $H$. Now, suppose that $v_i$ is not the root of $t_1$. Then, by Proposition 3.1, node $v_i$ must be in $t_i$'s right sibling since $v_i$ was inserted later. By symmetry, the $j$-th node inserted in $t_1$ has the same height as the $j$-th node inserted in $t_1$'s right sibling. If $v_i$ is the latter node, then the former node has index $i - s(h(i))$ since $s(h(i))$ is the size of tree $t_1$. □

## 4. *The Insert operation*

In this section, we show that the *Insert* operation requires only $\Theta(1)$ amortized time. Inserting the $n$-th element into a post-order heap involves copying the new value into the node $v_n$ and then heapifying at $v_n$. To perform the heapify, we must compute the location of $v_n$'s left and right children in the array so that we can compare their values to $v_n$'s new value. We now show how to compute these array locations.

The post-order insertion rule tells us that the left and right child of $v_n$ were the last nodes inserted in each of their respective subtrees and that the nodes in the right subtree were added just after the left child. This means that the right child was inserted just before $v_n$, so its index is $n-1$. The index of the left child must be $n-1$ minus the number of nodes in the right subtree. Since $H(n)$ is the height of $v_n$, the size of the right subtree is $s(H(n)-1)$, so the index of the left child is $n-1-s(H(n)-1)$.

This gives us a way of computing the locations of $v_n$'s left and right child. Unfortunately, computing $H(n)$ directly from its definition may require $\Theta(\log n)$ time in the worst-case. We will now show how we can compute the height of $v_n$ in $\Theta(1)$ time by keeping track of some additional information.

Let $p_n(H(n)+j)$ denote the node that is the $j$-th ancestor of $v_n$ in $\mathcal{G}$. For example, $p_n(H(n))$ is $v_n$ itself and $p_n(H(n)+1)$ is the parent of $v_n$. We now define $D_n$, an infinite $\{0,1\}$ string, which records the whether $v_n$'s ancestors are left or right children of their respective parents. Let $D_n(h)$ denote the $h$-th bit of $D_n$. Then, we define

$$D_n(h) = \begin{cases} 0 & \text{if } h < H(n) \\ 0 & \text{if } p_n(h) \text{ is the left child of } p_n(h+1) \\ 1 & \text{if } p_n(h) \text{ is the right child of } p_n(h+1). \end{cases}$$

PROPOSITION 4.1 *$D_n(h) = 0$ for $h > h(n)$.*

*Proof*: By Theorem 3.4, the height of $t_1$ is $h(n)$. Since its parent, $p_1$, has not yet been inserted, every existing node is a descendant of $p_1$. The post-order insertion rule implies that the left subtree is fully inserted before any nodes in the right subtree. Thus $p_1$ must be the left child of its parent. Similarly, $p_1$'s parent must be a left child, etc. Since the height of $p_1$ is $h(n)+1$, we have shown that $D_n(h) = 0$ for $h > h(n)$. □

This proposition shows that we can store $D_n$ in $O(\log n)$ bits: we need not record $D_n(h)$ for $h > h(n)$ since we know it is 0. In particular, we will record $D_n$ as an integer and compute $D_n(h)$ by standard arithmetic as $\lfloor D_n/2^h \rfloor \bmod 2$. This will give a result of 0 for any $h > h(n)$. We will now show that $D_{n+1}$ and $H(n+1)$ are easily computed from $D_n$ and $H(n)$.

THEOREM 4.2 *$D_{n+1}$ and $H(n+1)$ are related to $D_n$ and $H(n)$ by*

$$\begin{aligned} D_{n+1} &= \begin{cases} D_n + 2^{H(n)} & \text{if } D_n(H(n)) = 0 \\ D_n - 2^{H(n)} & \text{if } D_n(H(n)) = 1 \end{cases} \\ H(n+1) &= \begin{cases} 0 & \text{if } D_n(H(n)) = 0 \\ H(n)+1 & \text{if } D_n(H(n)) = 1 \end{cases} \end{aligned}$$

*Proof*: First, suppose that $D_n(H(n)) = 0$. This means that $v_n$ is the left child of its parent, $p_n$. The post-order insertion rule requires the next node inserted, $v_{n+1}$, to be the leftmost leaf in $p_n$'s right subtree, so its height $H(n+1)$ is 0. Nodes $v_n$ and $v_{n+1}$ have the same ancestor $p_n$ at height $H(n)+1$, so $D_n(h) = D_{n+1}(h)$ for $h > H(n)$. Since $v_{n+1}$ is the leftmost leaf in the right subtree of $p_n$, all of its ancestors up to height $H(n)-1$ are left children, which means that $D_{n+1}(h) = 0$ for $h < H(n)$, and by definition, we have $D_n(h) = 0$ for $h < H(n)$. Lastly, whereas $v_n$ is the left child of $p_n$, node $v_{n+1}$ is a descendant of the right child of $p_n$, which means that $D_{n+1}(H(n)) = 1$ while $D_n(H(n)) = 0$. In summary, we have shown that $D_{n+1} = D_n + 2^{H(n)}$.

Now, suppose that $D_n(H(n)) = 1$. This means that $v_n$ is the right child of $p_n$. The post-order insertion rule requires $v_{n+1}$ to be $p_n$. Thus, we will have $H(n+1) = H(n)+1$. Since $v_n$ and $v_{n+1}$ have the same ancestors at height $H(n+1)$ and above, we have $D_n(h) = D_{n+1}(h)$ for $h > H(n)$. By definition, we know that $D_{n+1}(h) = 0$ for $h \leq H(n)$. Since $D_n(h) = 0$ for $h < H(n)$ and $v_n$ is the right child of $p_n$, the only difference between $D_{n+1}$ and $D_n$ is that $D_{n+1}(H(n)) = 0$ while $D_n(H(n)) = 1$. Therefore, we have shown that $D_{n+1} = D_n - 2^{H(n)}$. □

This shows that we can compute the height and ancestry string of $v_{n+1}$, namely $H(n+1)$ and $D_{n+1}$, given the height and ancestry string of $v_n$, namely $H(n)$ and $D_n$. Given $H(n+1)$, we can efficiently compute the array locations of $v_{n+1}$'s left and right children: $n - s(H(n+1)-1)$ and $n$ respectively. The heights of the children are one less than the height of $v_{n+1}$, so we can compute the indices of their children, and so on. This means that we have enough information to efficiently perform a heapify at $v_{n+1}$. Thus, in order to perform the *Insert* operation, we maintain (1) the index $n$ of the last node in the array, (2) the height $H(n)$ of $v_n$, and (3) its ancestry string $D_n$. By Proposition 4.1, this data can be stored in three words. By Theorem 4.2, it can be updated in $\Theta(1)$ time.

THEOREM 4.3 *We can implement the Insert operation in $\Theta(1)$ amortized time.*

*Proof*: As we saw above, the *Insert* operation does only constant work aside from the heapify at $v_{n+1}$. To prove that the heapify runs in $\Theta(1)$ amortized time, we use a potential function that is the sum of the heights of each tree $t_i$. When the inserted node has height 0, the heapify does only constant work and the potential is unchanged, so the amortized cost is $\Theta(1)$. Otherwise, by the post-order insertion rule, we are inserting the new node of height $h$ as the parent of two subtrees of height $h-1$. The real cost of the heapify is proportional to $h$, but the potential decreases by $2(h-1) - h = h - 2$, so the amortized cost is just 2. □

## 5. *The DeleteMin operation*

To delete the minimum value in a post-order heap, we must first search for the node containing the minimum value. Since each tree $t_i$ is heap-ordered, we know that the minimum value must be at the root of some $t_i$. Below, we will see how to enumerate the tree roots in $O(\log n)$ time. Once we have found the root $t_i$ with the minimum value, we swap its value with the value in the last node $v_n$. Then, we must heapify at $t_i$ to restore the heap-order, which takes $O(\log n)$ time. Finally, we remove $v_n$, which now contains the minimum value. Below, we will also see how to update the values of $H(n)$ and $D_n$ in $O(\log n)$ time.

We can enumerate the roots of each tree $t_i$ as follows. By the post-order insertion rule, the root of $t_k$ must be the last node, $v_n$. Now, consider the previous tree $t_{k-1}$. The rule applied to the parent of $t_{k-1}$ implies that $t_k$ and all of its descendants were inserted just after $t_{k-1}$, which means that the index of root $t_{k-1}$ is $n - s(H(n))$. If $h$ is the height of $t_{k-1}$, then we know that $t_k$'s ancestor at height $h$ is the right sibling of $t_{k-1}$, which means that $D_n(h) = 1$. In general, if $D_n(h) = 1$, then the left sibling of $v_n$'s ancestor at height $h$ is some root $t_i$. Thus, we can find the heights of each of the trees by finding the 1 bits in $D_n$. Futhermore, we can compute the index $j_i$ of the previous tree $t_i$ at height $h$ by the formula $j_i = j_{i+1} - s(h)$ in $\Theta(1)$ time. This shows that we can enumerate the roots of the trees in $O(\log n)$ time.

After removing the minimum value, we must compute the values of $H(n-1)$ and $D_{n-1}$. If $H(n) > 0$, then node $v_{n-1}$ is the right child of $v_n$, so we have $H(n-1) = H(n) - 1$ and $D_{n-1} = D_n + 2^{H(n-1)}$. Otherwise, node $v_{n-1}$ is the left sibling of the first ancestor of $v_n$ that is a right child of its parent. Thus $H(n-1)$ is the position of the rightmost 1 bit in $D_n$ and $D_{n-1} = D_n - 2^{H(n-1)}$. This discussion shows that we can update $H(n)$ and $D_n$ in $O(\log n)$ time, which gives us the following result.

THEOREM 5.1 *We can implement the Delete operation in $O(\log n)$ time.*

## 6. *Experiments*

To verify the practicality of the post-order heap, we experimentally compared it to the standard binary heap. Both heaps were implemented in a comparable manner in the C# programming language. In each experiment, we inserted $n$ 32-bit integers into the heap, where $n$ is randomly chosen between 1 and $10^6$. We performed experiments with the data inserted in increasing, decreasing, and random order. For each ordering, we performed 300 experiments and recorded both the number of comparisons and the actual running time in seconds. Table 1 shows the average number of comparisons for

each *Insert* operation and the total time required to run all experiments. We ran the experiments on a 2.4GHz Pentium 4 machine with 512 MB of RAM.

For the binary heap, the best-case performance is achieved on increasing input. Since each new item is placed at the bottom of the tree, it will not need to be moved because its parent contains a smaller value. For the post-order heap, increasing input is actually a bad case. Since each new item is placed at the top of a tree, it will need to move all the way to the bottom. However, we proved above that the average height of an inserted node, over a sequence of insertions, is just 2. We can see in Table 1 that the post-order heap performed only 2 comparisons per insert, which is quite respectable.

On uniformly random input, the binary heap has similar performance since each item will need to move only a constant number of places on average (most items will stay at the bottom). We can see in Table 1 that the binary heap averages 2.38 comparisons per insert. On the other hand, the post-order heap averages only 1.87 comparisons per insert, more than 20% fewer.

The worst case for the binary heap is achieved on decreasing input. Here, each item is moved to the top of the heap, costing $\Theta(\log n)$ comparisons per operation. Table 1 shows that the binary heap averages 17.32 comparisons per insert, a huge jump over the increasing and uniformly random cases. Decreasing input is the best case for the post-order heap since each item does not move at all. The result is only 1 comparison per insert, which is a significant improvement over the binary heap.

Table 1 also shows the running time of the *Insert* operations on each input ordering. Here, we can see the cost of the extra bookkeeping required for the post-order heap. However, it is important to note that these running times are for 32-bit integer data, for which the cost of a comparison is extremely small. For larger integers, strings, or other objects with less efficient comparisons, the running time would more closely match the shape of the comparison data. As the time for a comparison increases to infinity, the total running time will be proportional to the number of comparisons.

| **Ordering** | **Binary Heap** | | **Post-Order Heap** | |
|---|---|---|---|---|
| | Comparisons | Total Time | Comparisons | Total Time |
| Increasing | 1.00 | 2.31 | 2.00 | 7.59 |
| Random | 2.38 | 5.66 | 1.87 | 7.53 |
| Decreasing | 17.32 | 27.09 | 1.00 | 5.00 |

**Table 1:** Experimental comparison of the running times of the *Insert* operations of binary and post-order heaps on three different types of input orderings. Since these results are for small integer data, the running times would trend towards the comparison values for data types with slower comparisons.

We also tested the performance of *Delete* operations for both data structures. The number of comparisons were roughly equivalent between the two: the post-order heap performed 15% more on random input but more than 20% less on increasing and decreasing input. However, for *Delete* operations, the extra costs for the post-order heap are more prominent, as it has to search through the $O(\log n)$ trees to find the minimum. The running time for the post-order heap is a factor of 2.0-2.5 worse than that of the binary heap. Again, it is important to keep in mind, though, that this is for 32-bit integer data, and that for other data types, this factor will decrease.

These results show that the post-order heap does indeed smooth out the poor worst-case behavior of the binary heap, albeit at the cost of a small constant factor slowdown in other cases. As noted above, the size of this constant factor will decrease on data types with slower comparisons, shrinking to less than 1 in the limit for *Insert* operations on uniformly random inputs. These results show that the post-order heap is a practical data structure, suitable for use in real-world applications.

## 7. *Conclusion*

We have presented the post-order heap, a variant of the standard binary heap. Like the binary heap, the post-order heap is an implicit data structure, requiring only three words of storage in addition to the array of elements. Unlike the binary heap, however, an *Insert* operation requires only $\Theta(1)$ amortized time. Unlike the data structure of Carlsson et al. [1], the post-order heap is quite simple, as we saw above and as is clear from the pseudocode in the appendix. We also presented experimental results demonstrating that the post-order heap is a practical data structure that should be considered for real-world applications. Thus, we have seen that the post-order heap is a simple and practical variant of the binary heap that delivers improved performance on *Insert* operations with only three words of additional storage and little added complication.

## *References*

[1] S. Carlsson, J. I. Munro, and P. V. Poblete. An implicit binomial queue with constant insertion time. In *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory*, volume 318 of *Lecture Notes in Computer Science*, pages 1–13, July 1988.

[2] R. W. Floyd. Algorithm 245 (Treesort). *Communications of the ACM*, 7(12):701, 1964.

[3] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[4] J. W. J. Williams. Algorithm 232 (Heapsort). *Communications of the ACM*, 7(1):347–348, 1964.

# A. *Pseudocode*

The pseudocode maintains four variables: *A*, the array containing the heap values, *n*, the number of nodes in the heap, *H*, the value of $H(n)$, and *D*, the value of $D_n$. The $H$-th bit of the integer $D$ is denoted $D[H]$. The pseudocode also uses the function $s(h) = 2^{h+1} - 1$, as given in Definition 3.3.

```
▷ Get left child of node i, which is at height h.
LEFT-CHILD(i, h)
1  return i − 1 − s(h − 1)

▷ Get right child of node i, which is at height h.
RIGHT-CHILD(i, h)
1  return i − 1

▷ Restore heap property at node i.
HEAPIFY(i, h)
 1  if h = 0 then
 2     return
 3  l ← LEFT-CHILD(i, h)
 4  r ← RIGHT-CHILD(i, h)
 5  if A[l] < A[i]
 6     then smallest ← l
 7     else smallest ← i
 8  if A[r] < A[smallest] then
 9     smallest ← r
10  if smallest ≠ i then
11     A[i] ↔ A[smallest]
12     HEAPIFY(smallest, h − 1)

▷ Insert a new value into heap.
INSERT(key)
 1  ▷ Compute D(n + 1) and H(n + 1)
 2  if n > 0 then
 3     if D[H] = 0
 4        then D[H] ← 1
 5             H = 0
 6        else D[H] ← 0
 7             H ← H + 1
 8
 9  ▷ Add new node to array; restore heap property
10  n ← n + 1
11  A[n] ← key
12  HEAPIFY(n, H)

▷ Remove min element from heap and return it.
DELETE-MIN()
 1  ▷ Enumerate all roots and find min element
 2  minLoc ← n
 3  minHeight ← H
 4  x ← n − s(H)
 5  h ← H − 1
 6  while x > 0
 7     do repeat h ← h + 1 until D[h] = 1
 8        if A[x] < A[minLoc] then
 9           minLoc ← x
10           minHeight ← h
11        x ← x − s(h)
12
13  ▷ Swap min to end; restore heap property
14  A[minLoc] ↔ A[n]
15  HEAPIFY(minLoc,minHeight)
16
17  ▷ Update n, H, and D
18  minNode ← A[n]
19  n ← n − 1
20  if n > 0 then
21     if H > 0
22        then H = H − 1
23             D[H] = 1
24        else while D[H] = 0 do H ← H + 1
25             D[H] = 0
26  return minNode
```