Towards ultimate binary heaps

Amr Elmasry[1] Jyrki Katajainen[2]

[1] *Computer and Systems Engineering Department, Alexandria University*
Alexandria 21544, Egypt
[2] *Department of Computer Science, University of Copenhagen*
Universitetsparken 1, 2100 Copenhagen East, Denmark

Abstract. A binary heap is a classical in-place priority queue described in most textbooks on data structures and algorithms. A binary heap is known to support *insert* and *extract-min* in $O(\lg n)$ worst-case time, where n denotes the number of elements stored in the data structure. We introduce an in-place variant of binary heaps that supports *insert* in $O(1)$ time, and *extract-min* in $O(\lg n)$ time with at most $\lg n + O(1)$ element comparisons, all in the amortized sense. Our construction is conceptually simple and can be presented in basic textbooks. Furthermore, we show how to support *insert* in $O(1)$ worst-case time and simultaneously support *extract-min* as efficiently as binary heaps. Finally, we show how a sequence of *extract-min* operations can be performed in-place using at most $\lg n + O(1)$ element comparisons per operation after linear preprocessing. The upper bounds we prove bypass the lower bounds known for both the *insert* and *extract-min* operations of binary heaps.

1. Introduction

A *binary heap* [15] is a priority-queue structure having the following properties:

Structure: It is a binary tree in which each node stores one element. This tree is almost complete in the sense that all levels are full, except perhaps the last level where elements are stored at the leftmost nodes.

Element ordering: The elements are kept in *heap order*, i.e. for each node its element is not larger than the elements at its descendants.

Representation: The elements are stored in breadth-first order in an array $A = [a_i \mid i \in \{1, \ldots, n\}]$. Then the left child of a_i is stored at a_{2i}, the right child at a_{2i+1}, and the parent at $a_{\lfloor i/2 \rfloor}$ if $i > 1$.

The main virtue of a binary heap is that it is an in-place data structure that, in addition to the element array, only needs $O(1)$ words of memory. Binary heaps support *minimum* in $O(1)$ worst-case time, and *insert* and *extract-min* in $O(\lg n)$ worst-case time. As originally introduced by Williams [15], the number of element comparisons performed by *extract-min* is at most $2 \lg n + O(1)$.

Our main motivation for writing this paper was to show the limitation of the lower bounds proved by Gonnet and Munro in their classical paper

CPH STL Report 2013-1. Written in April 2013.

on binary heaps [10] (see also [2]). More specifically, Gonnet and Munro showed that $\lceil \lg \lg(n+2) \rceil - 2$ element comparisons are necessary to insert an element into a binary heap. In addition, slightly correcting [10], Carlsson showed that $\lceil \lg n \rceil + \delta(n)$ element comparisons are necessary and sufficient to delete the minimum element from a binary heap with $n > 2^{h_{\delta(n)}+2}$ elements, where $h_1 = 1$ and $h_i = h_{i-1} + 2^{h_{i-1}+i-1}$. The assumptions behind these theorems are that the elements are stored in a single binary heap and that the heap order must be valid *everywhere* before and after each operation.

In Section 2 we show how to modify binary heaps to achieve better amortized bounds. More specifically, our in-place priority queue supports *insert* in $O(1)$ time, and *extract-min* in $O(\lg n)$ time using at most $\lg n + O(1)$ element comparisons, all in the amortized sense. We show that the $\Omega(\lg \lg n)$ lower bound per insertion is no more valid if we allow $O(\lg^2 n)$ nodes to violate the heap order. Even though this number is non-constant, we only need few words to keep track of the locations of these violations. That is, the resulting data structure is still in-place. To bypass the lower bound for *extract-min*, the main idea is to partition the element array into two zones around the median; this idea is borrowed from Katajainen's ultimate heapsort [11]. We build a binary heap using only the small elements, and use the large elements as a backlog to fill in the holes created by *extract-min* operations. In particular, the heap order may be violated among the nodes storing filler elements. The amount of extra space used is still $O(1)$ words since we do not track where the heap-order violations are.

In Section 3 we show how to obtain worst-case constant-time *insert* by applying a straightforward deamortization approach; some technical details make the construction somehow involved. In connection with this worst-case solution we can apply any of the *extract-min* algorithms known for standard binary heaps. However, we cannot match the $\lg n + O(1)$ amortized bound for the number of element comparisons performed by *extract-min* in the worst case. By using the *extract-min* algorithm of Gonnet and Munro [10] we achieve the bound of $\lg n + \log^* n + O(1)$ element comparisons per *extract-min*.

In Section 4 we extend the two-zone solution to a six-zone solution that achieves the bounds for *extract-min* in the worst case per operation after an $O(n)$ preprocessing time; for this latter scenario, we start with n elements and do not allow new insertions. In other words, we show that, starting with n elements and after linear-time preprocessing, we can perform n *extract-min* operations each in worst-case $O(\lg n)$ time using at most $\lg n + O(1)$ element comparisons per operation. This construction is suitable for online sorting.

Related work

Even though the time bounds proved are better than those possible for binary heaps, similar bounds can be achieved by other data structures: There are several priority queues that support *insert* in $O(1)$ amortized time [5, 13],

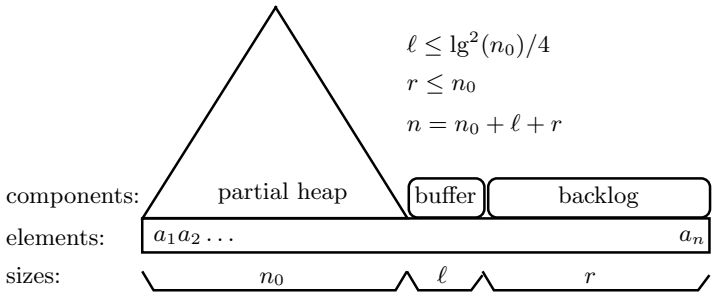

Figure 1. The components of our in-place priority queue.

and even in $O(1)$ worst-case time [4, 7, 8]. One can in fact transform any priority queue to support *insert* in $O(1)$ worst-case time on the expense of moving the cost of *insert* to *extract-min* [1]; this transformation also increases the space used.

The only in-place data structure providing the same *asymptotic* worst-case bounds as ours is that of Carlsson et al. [4]. Their data structure is composed of a collection of heaps, so it can bypass the lower bound on the cost of *insert* for binary heaps this way. However, our data structure has a smaller constant factor in the number of element comparisons performed by the *extract-min* operation.

The aforementioned bounds on the number of element comparisons ($O(1)$ per *insert* and $\lg n + O(1)$ per *extract-min*) can be achieved—even in the worst case—using the multipartite priority queues of Elmasry et al. [7]. However, their data structure is complicated; and more critically, it is pointer-based. Another motive for the current paper was to study whether the same comparison complexity can be achieved by an in-place data structure.

It is also possible to achieve at most $\lg n + O(1)$ element comparisons in the worst case per *extract-min*, while not allowing insertions, by using the construction in [6]; this data structure uses $O(n)$ extra bits, so it is not in-place.

2. Amortized solution

In this section we prove the following theorem:

Theorem 1. *Let n denote the number of elements stored in a data structure prior to an operation. There exists an in-place priority queue that supports minimum and insert in $O(1)$ time, and extract-min in $O(\lg n)$ time involving at most $\lg n + O(1)$ element comparisons. All these bounds are amortized.*

To handle *insert* efficiently, we rely on the buffering technique (where the elements are inserted into a *buffer* and when the buffer becomes full it is submerged with the main structure). The buffering technique is used in multipartite priority queues [7] and some other data structures (see, e.g. [1, 5]). To handle *extract-min* operations such that every *extract-min* performs

at most $\lg n + O(1)$ element comparisons in the amortized sense, we use a *partial heap* where $O(n)$ nodes may violate the heap order. The idea of using a partial heap, instead of a standard binary heap, is taken from [11]. More precisely, the data structure that we use to prove Theorem 1 consists of three components that are consecutive subarrays of the element array (for an illustration, see Fig. 1):

Partial heap: A partial heap has the same structure and representation as a binary heap, but not all elements obey the heap order. The root is *good*, and any other node is good if its parent is good and its element is not smaller than the element at its parent. That is, the elements in good nodes are in heap order. Within the remaining bottom subtrees the heap order may be violated, but the elements stored there are larger than or equal to any of the elements in the good nodes. The objective is that the elements in these bottom subtrees are by no means influential when performing the *extract-min* operations.

Insert buffer: The buffer is a subarray lying in between the other two components, and we maintain a separate pointer to the minimum element stored in there. If the size of the partial heap is n_0, the buffer is never allowed to become larger than $\Theta(\lg^2(n_0))$. The buffer should support insertions in constant time, minimum extractions in $O(\lg(n_0))$ time, and it should be possible to submerge a buffer into the partial heap efficiently.

Backlog: Every element in this subarray is larger than or equal to any element stored in a good node of the partial heap. An element from this component is used to fill in a leaf of the partial heap whenever one becomes vacant.

The data structure works in phases. Every phase is initiated by an *extract-min* operation once the backlog is empty. At the beginning of the phase, the median of all the elements is found in-place in linear time [12], and the element array is partitioned into two zones around this median; the larger elements form the new backlog. A binary heap is then built for the smaller elements in the first zone using Floyd's linear-time heap-construction algorithm [9].

Implementing insertions

The key idea behind our amortized constant-time solution is to insert the elements into the heap in bulks. Noting that ℓ elements can be inserted into a heap of size n_0 in $O(\ell + \lg^2(n_0))$ time with an efficient bulk-insertion procedure, an amortized constant-time insertion is indeed plausible by setting the maximum size of the buffer to $\Theta(\lg^2(n_0))$.

To be able to extract the minimum of the buffer in $O(\lg n_0)$ time, we treat the buffer as consecutive *chunks* of elements each of size $k = \lceil \lg(n_0)/2 \rceil$ and limit the number of these chunks to at most k chunks. All the chunks, except possibly the last one, will contain exactly k elements. The minimum of each chunk is kept at the first location of the chunk. When the minimum

of the buffer is extracted, the new minimum of its chunk is found in $O(k)$ time using $k-1$ element comparisons (by scanning the elements of the chunk), then the new overall minimum of the buffer is found in $O(k)$ time using $k-1$ element comparisons (by scanning the minima of the chunks). In our implementation we maintain the variables: n_0 to recall the number of elements in the heap, ℓ to recall the number of elements in the buffer, and b_{min} to keep track of the index of the minimum element of the buffer. Alternatively, instead of using b_{min}, we could have kept the minimum of the buffer at its first (or last) location. Still, we opted not to use this alternative as it would make the worst-case solution intricate.

For the *insert* operation (see the pseudo-code in Fig. 2), a new element is inserted into the buffer as long as the buffer size is below the threshold. Correspondingly, the backlog is rotated one position to the right. After the insertion, the minimum of the last chunk as well as the overall minimum of the buffer are adjusted if necessary; this requires at most two element comparisons. Once the threshold is reached, a *bulk-insert* operation is performed by moving all the elements of the buffer to the heap.

We make use of Williams' *sift-down* subroutine [15] to restore the heap

```
procedure: insert
input: A[1..n]: in-place priority queue; e: element
data structures: A[1..n_0]: partial heap; A[n_0+1..n_0+ℓ]: buffer;
                 A[n_0+ℓ+1..n]: backlog; b_min: index
ℓ++
n++
a_n ← e
swap(a_{n_0+ℓ}, a_n)
if ℓ = 1
|  b_min ← n_0 + 1
|  k ← ⌈lg(n_0)/2⌉
p ← n_0 + k * ⌊(ℓ − 1)/k⌋ + 1
if a_{n_0+ℓ} < a_p
|  // the new element is the smallest in its chunk
|  swap(a_{n_0+ℓ}, a_p)
|  if a_p < a_{b_min}
|  |  // the new element is the smallest in the buffer
|  |  b_min ← p
if ℓ = k^2
|  // the buffer is full
|  bulk-insert(A, n_0, ℓ)
|  n_0 ← n_0 + ℓ
|  ℓ ← 0
```

Figure 2. The pseudo-code for *insert*.

```
procedure: bulk-insert
input: A[1 .. n_0 + ℓ]: array; n_0: index; ℓ: index
data structures: A[1 .. n_0]: partial heap; A[n_0 + 1 .. n_0 + ℓ]: buffer
right ← n_0 + ℓ
left ← max {n_0 + 1, ⌊(n_0 + ℓ)/2⌋}
while right ≠ 1
    left ← ⌊left/2⌋
    right ← ⌊right/2⌋
    for j ∈ {right, right − 1, ··· , left}
        sift-down(A, j, n_0 + ℓ)
```

Figure 3. The pseudo-code for *bulk-insert*.

order between a node (that may be violating) and those in its subtrees (that are binary heaps). We could as well use any of the alternative realizations for this subroutine [3, 9, 14]. We equip the subroutine with three parameters: the element array under consideration, the index of the element to be sifted down, and the index of the last element. More precisely, *sift-down*(A, j, m) dictates a *sift-down* starting at the jth element of array A that has m elements.

To perform a bulk insertion in $O(\ell + \lg^2(n_0))$ time (see the pseudo-code in Fig. 3), the heap order is reestablished bottom up level by level. Starting with the parents of the nodes of the buffer, for each node we call the *sift-down* subroutine. We then consider the parents of these nodes at the next upper level, restoring the heap order up to this level. This process is repeated all the way up to the root. As long as there are more than two nodes that are considered at a level, the number of such nodes almost halves at the next level. The correctness of *bulk-insert* easily follows by induction; after calling the *sift-down* subroutine for a range of nodes at a level, the heap order is established for all their descendants up to this level. This must be the case since we start with all the elements of the buffer and consider all the ancestors of this range. Observe that this *bulk-insert* algorithm works both for partial heaps and binary heaps.

The work done by *insert*, other than for the bulk insertion, is obviously constant. We separately consider two parts of the *bulk-insert* procedure. The first part comprises the *sift-down* calls for the nodes at the levels with more than two involved nodes. The total number of those nodes at the jth last level is at most $\lfloor(\ell - 2)/2^{j-1}\rfloor + 2$. Here we use the fact that the number of parents of a contiguous block of b elements in the array representing a heap is at most $\lfloor(b-2)/2\rfloor + 2$. For a node at the jth last level, a call to the *sift-down* subroutine requires $O(j)$ work. It follows that the amount of work involved in the first part is $O(\sum_{j=2}^{\lceil \lg r\rceil} j/2^{j-1} \cdot \ell) = O(\ell)$. The second part comprises at most $2\lceil\lg(n_0)\rceil$ calls to the *sift-down* subroutine; this accounts for a total of $O(\lg^2(n_0))$ time. Since $\ell = \Theta(\lg^2(n_0))$, the overall work done is $O(\lg^2(n_0))$ and can be amortized as a constant per operation.

```
procedure: extract-root
input: A[1 .. n]: in-place priority queue
data structures: A[1 .. n_0]: partial heap; A[n_0 + 1 .. n_0 + ℓ]: buffer;
                 A[n_0 + ℓ + 1 .. n]: backlog; b_min: index
if n_0 + ℓ = n
|  // the backlog is empty
|  rebuild-structure(A, n, n_0, ℓ)
i = 1
while i ≤ ⌊n_0/2⌋
|  if A[2i] < A[2i + 1]
|  |  A[i] ← A[2i]
|  |  i ← 2i
|  else
|  |  A[i] ← A[2i + 1]
|  |  i ← 2i + 1
A[i] ← A[n]   // move a backlog element to the vacant leaf
n--
```

Figure 4. The pseudo-code for the *extract-root* operation that uses amortized $\lg n + O(1)$ element comparisons.

Implementing minimum extractions

There are two cases to consider depending on whether the overall minimum is the minimum of the heap or the minimum of the buffer (the overall minimum cannot be in the backlog). Let us consider the case where the minimum is in the heap; we call the operation *extract-root* (see the pseudo-code in Fig. 4). After removing the element at the root of the heap, this operation compares the two elements stored at the children of the vacant node and promotes the smaller to the parent node. The process of moving the smaller of the two children is repeated until a vacant leaf is created. The operation concludes its work by moving an element from the backlog (the last element in the array) to this vacant leaf.

The correctness of the *extract-root* operation follows from the following facts:

- At the beginning of every phase all the elements in the heap are smaller than those in the backlog.
- All the time, the elements that have been moved from the backlog to the heap are descendants of those who were initially in the heap.
- The heap order is messed up only among the nodes coming from the backlog.
- Every *extract-root* operation ensures that the backlog is not empty.

Since one element comparison is used per level, the *extract-root* operation performs at most $\lg n + O(1)$ element comparisons. The work done at the beginning of a phase to rebuild the structure (median finding and heap construction) is linear in the existing number of elements. Using straightforward arguments, this work can be amortized as $O(1)$ cost per *insert*.

```
procedure: extract-min
input: A[1..n]: in-place priority queue
data structures: A[1..n_0]: partial heap; A[n_0+1..n_0+ℓ]: buffer;
                 A[n_0+ℓ+1..n]: backlog; b_min: index
if ℓ = 0 or a_1 < a_{b_min}
    // the overall minimum is in the heap
    extract-root(A)
    return
// move the new minimum of the chunk starting at b_min to a_{b_min}
x ← a_{n_0+ℓ}
ix ← n_0 + ℓ
j ← min{b_min + k − 1, n_0 + ℓ − 1}
for i = b_min + 1 to j
    if a_i < x
        x ← a_i
        ix ← i
a_{b_min} ← a_{ix}
a_{ix} ← a_{n_0+ℓ}
// update b_min by checking the minima of the chunks
x ← a_{n_0+1}
i ← n_0 + k + 1
while i ≤ n_0 + ℓ
    if a_i < x
        x ← a_i
        b_min ← i
    i ← i + k
a_{n_0+ℓ} ← a_n
ℓ--
n--
```

Figure 5. The pseudo-code for *extract-min*.

If the overall minimum is in the buffer (see the pseudo-code of *extract-min* in Fig. 5), we find the new minimum within the chunk that contained the extracted minimum and then we find the new minimum of the buffer; this involves at most $2k - 2$ element comparisons.

Using one element comparison, we can decide whether the minimum is in the buffer or in the heap. The extraction of the minimum of the heap takes $O(\lg n)$ time and involves at most $\lg n + O(1)$ element comparisons. If the minimum is in the buffer, we have a pointer to it. Scanning the elements in the chunk that contains the minimum takes $O(k) = O(\lg n)$ time. Scanning the minima of the chunks also takes $O(k) = O(\lg n)$ time (as we are keeping these minima at the first location of every chunk, and their indices can be calculated in constant time each). In total, deleting the minimum of the buffer takes $O(\lg n)$ time. The number of element comparisons performed either way is bounded by $\lg n + O(1)$.

3. Deamortizing insertions

In this section we show how to deamortize insertions. Minimum extractions can be performed using any of the known procedures for binary heaps. The result to be proved can be summarized as follows:

Theorem 2. *Let n denote the number of elements stored in a data structure prior to an operation. There exists an in-place priority queue that supports minimum and insert in $O(1)$ worst-case time, and extract-min in $O(\lg n)$ worst-case time involving at most $\lg n + \log^* n + O(1)$ element comparisons.*

Our key idea here is to do the bulk insertion incrementally with the upcoming insert operations and keep the heap fully functional under this submersion. We maintain up to two buffers as extensions to a binary heap; one accepting new insertions and occupying the last elements of the array, while the other is being incrementally submerged with the heap. The details follow.

We call the first buffer the *insertion buffer*, and the second buffer the *submersion buffer*. An *insert* operation is performed exactly as before in the insertion buffer. Once the insertion buffer is full, the submersion buffer must have been already submerged with the heap (the heap order is established). We then declare the submersion buffer as part of the heap, make the insertion buffer the new submersion buffer, and initiate an empty insertion buffer. As will be illustrated next, the melding of the submersion buffer with the heap should be done fast enough before the insertion buffer becomes full.

From the amortized solution, we know that $O(\lg^2(n_0))$ time is enough to finish the bulk insertion for the submersion buffer; this means that a constant amount of work is done in connection with every upcoming insertion. As before, we perform *sift-down* operations starting from the nodes of the buffer bottom up level by level. To keep track of the progress done in this submersion process, we maintain two intervals that represent the nodes up to which the *sift-down* subroutine has been called. Each such interval is represented by two indices indicating the left and right nodes in the interval, call them (l_1, r_1) and (l_2, r_2). Note that these two intervals are at two consecutive levels of the heap, and that the parent of the left node of one interval has an index that is one more than the right node of the second interval, i.e. $\lfloor l_1/2 \rfloor = r_2 + 1$. We call these two intervals the *frontier* of the submersion process. Obviously we need a constant number of words to keep track of the frontier. While the submersion process advances, the frontier moves upwards and shrinks until it has one or two nodes. This frontier records that the *sift-down* now in progress is being performed starting from the node whose index is r_2+1. As a special case, at some points of time there may exist only one interval (l, r) in the frontier. In such a case, the *sift-down* now in progress is being performed starting from the node whose index is $\lfloor l/2 \rfloor$. In addition to the frontier we also maintain the index of the node where the *sift-down* in progress is currently handling. With every insertion, the current *sift-down* progresses a constant number of steps downwards and this index is updated. Once the *sift-down* returns, the frontier is updated

as follows. If there are two intervals, the upper interval is extended by r_2+1 and the lower interval shrinks by removing l_1 and possibly l_1+1 if both of them are siblings. If there is only one interval, it shrinks by removing l and possibly $l+1$, and a new interval is added to the frontier that contains only one node the index of which is $\lfloor l/2 \rfloor$. When the frontier reaches the root, the submersion process is complete. To summarize, the information maintained to record the state of the incremental submersion is: two intervals of indices to represent the frontier plus the node where the current *sift-down* is.

As for the insertion buffer, we also maintain the index of the minimum among the elements on the frontier. We treat each of the two intervals of the frontier as a set of consecutive chunks (subintervals). Except for the first and last chunk on each interval that may have less nodes, every other chunk has k nodes. In addition, we maintain the invariant that the minimum within every chunk on the frontier, except for the first and last chunks, is kept at the entry storing the first node among the nodes of the chunk. To accomplish this, we keep track of the minimum of the last chunk of the second interval while the frontier is expanding from the end. Once the number of nodes in this chunk reaches the threshold k, we swap the minimum with the first node in the chunk. This would be followed by an extra *sift-down* call to the node that receives the larger element; this *sift-down* is again performed incrementally. As we do this swap at most once with every chunk, the time bounds are still manageable.

Next we turn to the *extract-min* operation. To find the overall minimum, we need to consider the minima of: the heap, the insertion buffer, and the submersion buffer; the smallest among the three is to be deleted. We swap the minimum with the last element in the array; that is an element of the insertion buffer if it is not empty, otherwise it is an element of the submersion buffer if it not empty, otherwise it is an element of the heap. A subtle point to be considered (that has not popped up in the amortized solution) is when this last element is the minimum of the buffer it belongs to. In such a case, we swap this minimum element with the first element of the buffer.

1) If the minimum is in the heap, we call *sift-down* starting from the root. However, the *sift-down* subroutine must be modified such that it does not cross the frontier if both meet. More precisely, if the *sift-down* subroutine reaches a node that is on the frontier it stops; a condition that can be easily checked. The reason is that the frontier structure holds information about the minima of its chunks, and we do not want to lose track of any of those elements. In addition, the nodes below the frontier are not yet submerged with the rest of the heap, so we need not bother with fixing their order with their ancestors.

2) If the minimum is in the insertion buffer, we delete it while maintaining the structure and updating the minimum of its chunk and the overall minimum of the insertion buffer (as for the amortized solution).

3) If the minimum is in the submersion buffer, we know that this minimum must be on the frontier of the submersion process. As we readily have the index of the minimum of the frontier, this minimum is swapped with the

last element of the array and a *sift-down* call is performed to remedy the order between the frontier node and its descendants. When the minimum on the frontier is replaced, the nodes of the chunk that contained it are scanned for the new minimum, which is in turn swapped with the element at the first position of the chunk again followed by a *sift-down* call for the larger element. The overall new minimum on the frontier is then localized by scanning the fist and last chunks in addition to the nodes that are at the beginning of every other chunk; there are $O(k)$ such nodes. It follows that *extract-min* still takes $O(\lg n)$ worst-case time, while *insert* is performed in $O(1)$ worst-case time as claimed.

4. Deamortizing minimum extractions

In this section we consider online sorting. The task is to use linear time for constructing a data structure and then support *extract-min* operations at optimal worst-case cost. Our result can be summarized as follows:

Theorem 3. *Let n denote the number of elements stored in a data structure prior to an operation. There exists an in-place priority queue that can be constructed in $O(n)$ worst-case time, and supports minimum extraction in $O(\lg n)$ worst-case time involving at most $\lg n + O(1)$ element comparisons.*

Our objective is to partition the elements into six zones $Z_1, Z_2, \ldots, Z_6$, such that the elements in Z_i are not larger than those in Z_{i+1} and $|size(Z_i) - size(Z_j)| \leq 1$, for all $i, j \in \{1, 2, \ldots, 6\}$. Moreover, we build a binary heap for the elements in each of Z_1, Z_2 and Z_3. This initial configuration can be constructed in linear time, accounting for $O(1)$ amortized cost per element. We would prefer to think about the sequence of *extract-min* operations being executed in epochs; in each epoch half the existing elements are deleted. Consider a realizable snapshot for our zones at the beginning of phase t, where we only have five zones, $Z_1^{(t)}, Z_2^{(t)}, Z_3^{(t)}, Z_4^{(t)}, Z_5^{(t-1)}$, fulfilling the above conditions except for $Z_5^{(t-1)}$ for which $|size(Z_5^{(t-1)}) - 2 \cdot size(Z_j^{(t)})| \leq 1$, for all $j \in \{1, 2, 3, 4\}$, and that the first three zones each has its elements forming a binary heap. At the end of an epoch, there will again be five zones with the same conditions fulfilled. Every epoch is divided into three consecutive phases as follows:

1. The elements of $Z_1^{(t)}$ are deleted using $Z_4^{(t)}$ as the backlog. In the meantime, the elements of $Z_5^{(t-1)}$ are incrementally split around their median into two partitions $Z_5^{(t)}$ and $Z_6^{(t)}$.
2. The elements of $Z_2^{(t)}$ are deleted using $Z_5^{(t)}$ as the backlog. In the meantime, the elements of $Z_4^{(t)}$ (that have replaced $Z_1^{(t)}$) are incrementally split around their median into two partitions $Z_1^{(t+1)}$ and $Z_2^{(t+1)}$, and a binary heap is incrementally built within each of the two zones.
3. The elements of $Z_3^{(t)}$ are deleted using $Z_6^{(t)}$ as the backlog. In the meantime, the elements of $Z_5^{(t)}$ (that have replaced $Z_2^{(t)}$) are incrementally

split around their median into two partitions $Z_3^{(t+1)}$ and $Z_4^{(t+1)}$, and a binary heap is incrementally built in $Z_3^{(t+1)}$.

Note that when the last element of a zone Z_i is moved to a preceding zone, its array-entry is filled up by moving the last element of Z_{i+1} in its place, and this is repeated until we vacate the last element of the last zone.

One final subtle issue is what to do when the number of elements in a zone Z_i, $i \in \{1, 2, 3\}$, is not exactly equal to that of its backlog zone Z_{i+3}; there are two possible cases: If $size(Z_i) = size(Z_{i+3}) - 1$, after deleting all the elements of Z_i, we only need to move the remaining element of Z_{i+3} to Z_i. If $size(Z_i) = size(Z_{i+3}) + 1$, after consuming all the elements of Z_{i+3}, to delete the last element of Z_i we simply move the last element of Z_i itself to replace the deleted element (this last deletion takes constant time).

5. Afterword

In our view, it is interesting that we could bypass two lower bounds—believed to be solid for a long time—by slightly loosening the constraints that are intrinsic to the lower bounds. Indeed, we attested a heap structure that guarantees constant worst-case insertion time and uses n array entries to store n elements. We also came up with an algorithm to build a structure in-place in linear time, which would then support minimum extraction in $O(\lg n)$ worst-case time and at most $\lg n + O(1)$ element comparisons per operation. To achieve our goals we allowed some nodes to violate the heap order; one may wonder why we would need to keep the heap fully ordered after every operation. It is still open whether it is possible to bypass the lower bound on the number of element comparisons for *extract-min* while guaranteeing $O(1)$ worst-case cost per insertion. Another open question is whether a binary heap, permitting violation nodes, could accommodate *decrease* in constant time per operation.

References

[1] S. Alstrup, T. Husfeldt, T. Rauhe, and M. Thorup, Black box for constant-time insertion in priority queues, *ACM Trans. Algorithms* **1**, 1 (2005), 102–106.

[2] S. Carlsson, An optimal algorithm for deleting the root of a heap, *Inform. Process. Lett.* **37**, 2 (1991), 117–120.

[3] S. Carlsson, A note on Heapsort, *Comput. J.* **35**, 4 (1992), 410–411.

[4] S. Carlsson, J. I. Munro, and P. V. Poblete, An implicit binomial queue with constant insertion time, *SWAT 1988*, *LNCS* **318**, Springer (1988), 1–13.

[5] S. Edelkamp, A. Elmasry, and J. Katajainen, The weak-heap data structure: Variants and applications, *J. Discrete Algorithms* **16** (2012), 187–205.

[6] A. Elmasry, Three sorting algorithms using priority queues, *ISAAC 2003*, *LNCS* **2906**, Springer (2003), 209–220.

[7] A. Elmasry, C. Jensen, and J. Katajainen, Multipartite priority queues, *ACM Trans. Algorithms* **5**, 1 (2008), Article 14.

[8] A. Elmasry, C. Jensen, and J. Katajainen, Two skew-binary numeral systems and one application, *ACM Trans. Comput. Syst.* **50**, 1 (2012), 185–211.

[9] R. W. Floyd, Algorithm 245: Treesort 3, *Commun. ACM* **7**, 12 (1964), 701.

[10] G. H. Gonnet and J. I. Munro, Heaps on heaps, *SIAM J. Comput.* **15**, 4 (1986), 964–971.
[11] J. Katajainen, The ultimate heapsort, *CATS 1998*, *Australian Computer Science Communications* **20**, Springer-Verlag Singapore (1998), 87–95.
[12] T. W. Lai and D. Wood, Implicit selection, *SWAT 1988*, *LNCS* **318**, Springer (1988), 14–23.
[13] J. Vuillemin, A data structure for manipulating priority queues, *Commun. ACM* **21**, 4 (1978), 309–315.
[14] I. Wegener, Bottom-up-Heapsort, a new variant of Heapsort beating, on an average, Quicksort (if n is not very small), *Theoret. Comput. Sci.* **118**, 1 (1993), 81–98.
[15] J. W. J. Williams, Algorithm 232: Heapsort, *Commun. ACM* **7**, 6 (1964), 347–348.