An In-Place Priority Queue with $O(1)$ Time for Push and $\lg n + O(1)$ Comparisons for Pop*

Stefan Edelkamp[1], Amr Elmasry[2], and Jyrki Katajainen[3]

[1] Faculty 3—Mathematics and Computer Science, University of Bremen
[2] Department of Computer Engineering and Systems, Alexandria University
[3] Department of Computer Science, University of Copenhagen

Abstract. An *in-place priority queue* is a data structure that is stored in an array, uses constant extra space in addition to the array elements, and supports the operations *top* (*find-min*), *push* (*insert*), and *pop* (*delete-min*). In this paper we introduce an in-place priority queue, for which *top* and *push* take $O(1)$ worst-case time, and *pop* takes $O(\lg n)$ worst-case time and involves at most $\lg n + O(1)$ element comparisons, where n denotes the number of elements currently in the data structure. The achieved bounds are optimal to within additive constant terms for the number of element comparisons, hereby solving a long-standing open problem. Compared to binary heaps, we surpass the comparison bound for *pop* and the time bound for *push*. Our data structure is similar to a binary heap with two crucial differences:

(1) To improve the comparison bound for *pop*, we reinforce a stronger heap order at the bottom levels of the heap such that the element at any right child is not smaller than that at its left sibling.
(2) To speed up *push*, we buffer insertions and allow $O(\lg^2 n)$ nodes to violate heap order in relation to their parents.

1 Introduction

A *binary heap*, invented by Williams [19], is an in-place data structure that

(1) implements a priority queue (i.e. supports the operations *top*, *construct*, *push*, and *pop*);
(2) requires $O(1)$ words of extra space in addition to an array storing the elements; and
(3) is viewed as a nearly complete binary tree where, for every node other than the root, the element at that node is not smaller than the element at its parent (*heap order*).

L.D. Beklemishev (ed.): CSR 2015, LNCS 9139, pp. 1–15, 2015.

Table 1. The worst-case performance of some priority queues. The amount of extra space is measured in words and the complexity of operations in element comparisons. Here n denotes the number of elements stored and w the size of machine words in bits. For all these data structures, the worst-case running time of *top* is $O(1)$, that of *construct* is $O(n)$, and the worst-case running time of *push* and *pop* is proportional to the number of element comparisons (except for heaps on heaps, *push*[‡] is logarithmic).

Data structure	Extra space	*push*	*pop*
Binary heaps [19]	$O(1)$	$\lg n+O(1)$	$2\lg n+O(1)$
Binomial queues [1, 17]	$O(n)$	$O(1)$	$2\lg n+O(1)$
Heaps on heaps [13]	$O(1)$	$\lg\lg n+O(1)^{\ddagger}$	$\lg n+\log^* n+O(1)$
Queues of pennants [5]	$O(1)$	$O(1)$	$3\lg n+\log^* n+O(1)$
Multipartite priority queues [10]	$O(n)$	$O(1)$	$\lg n+O(1)$
Engineered weak heaps [8]	$n/w+O(1)$	$O(1)$	$\lg n+O(1)$
Strengthened lazy heaps [this paper]	$O(1)$	$O(1)$	$\lg n+O(1)$

Letting n denote the number of elements in the data structure, a binary heap supports *top* in $O(1)$ worst-case time, and *push* and *pop* in $O(\lg n)$ worst-case time. For Williams' original proposal [19], the number of element comparisons performed by *push* is at most $\lg n+O(1)$ and that by *pop* is at most $2\lg n+O(1)$. Immediately after the appearance of Williams' paper, Floyd showed [12] how to support *construct*, which builds a heap for n elements, in $O(n)$ worst-case time with at most $2n$ element comparisons.

Since a binary heap does not support all the operations optimally, many attempts have been made to develop priority queues supporting the same set (or even a larger set) of operations that improve the worst-case running time of the operations as well as the number of element comparisons performed by them [1, 3, 5, 6, 8, 10, 13, 17]. In Table 1 we summarize the fascinating history of the problem, considering the space and comparison complexities.

Assume that, for a problem of size n, the bound achieved is $\mathrm{A}(n)$ and the best possible bound is $\mathrm{OPT}(n)$. We distinguish three different concepts of optimality:

Asymptotic optimality: $\mathrm{A}(n) = O(\mathrm{OPT}(n))$.
Constant-factor optimality: $\mathrm{A}(n) = \mathrm{OPT}(n) + o(\mathrm{OPT}(n))$.
Up-to-additive-constant optimality: $\mathrm{A}(n) = \mathrm{OPT}(n) + O(1)$.

As to the amount of space used and the number of element comparisons performed, we aim at up-to-additive-constant optimality. From the information-theoretic lower bound for sorting [15, Sect. 5.3.1], it follows that, in the worst case, either *push* or *pop* must perform at least $\lg n - O(1)$ element comparisons. As to the running times, we aim at asymptotic optimality. Our last natural goal is to support *push* in $O(1)$ worst-case time, because then *construct* can be trivially realized in linear time by repeated insertions.

The binomial queue [17] was the first priority queue supporting *push* in $O(1)$ worst-case time. (This was mentioned as a short note at the end of Brown's paper [1].) However, the binomial queue is a pointer-based data structure requiring $O(n)$ pointers in addition to the elements. For binary heaps, Gonnet and Munro

showed [13] how to perform *push* using at most $\lg\lg n+O(1)$ element comparisons and *pop* using at most $\lg n + \log^* n + O(1)$ element comparisons. Carlsson et al. showed [5] how to achieve $O(1)$ worst-case time per *push* by an in-place data structure that utilizes a queue of pennants. (A *pennant* is a binary heap with an extra root that has one child.) For this data structure, the number of element comparisons performed per *pop* is bounded by $3\lg n + \log^* n + O(1)$. The multipartite priority queue [10] was the first priority queue achieving the asymptotically optimal time and up-to-additive-constant optimal comparison bounds. Unfortunately, the structure is involved and its representation requires $O(n)$ pointers. Another solution [8] is based on weak heaps [7]: To implement *push* in $O(1)$ worst-case time, a bulk-insertion strategy is used—employing two buffers and incrementally merging one with the weak heap before the other is full. The weak heap also achieves the desired worst-case time and comparison bounds, but it uses n additional bits.

Ever since the work of Williams [19], it was open whether there exists an in-place data structure that can match the information-theoretic lower bounds on the number of element comparisons for all the operations. In view of the lower bounds proved in [13], it was not entirely clear if such a structure exists. In this paper we answer the question affirmatively by introducing the strengthened lazy heap that operates in-place, supports *top* and *push* in $O(1)$ worst-case time, and *pop* in $O(\lg n)$ worst-case time involving at most $\lg n+O(1)$ element comparisons.

When a strengthened lazy heap is used in heapsort, the resulting algorithm sorts n elements in-place in $O(n\lg n)$ worst-case time performing at most $n\lg n + O(n)$ element comparisons. The number of element comparisons performed matches the information-theoretic lower bound for sorting up to the additive linear term. Ultimate heapsort [14] is known to have the same complexity bounds, but in both solutions the constant factor of the additive linear term is high.

In a binary heap the number of element moves performed by *pop* is at most $\lg n + O(1)$. We have to avow that, in our data structure, *pop* may require more element moves. On the positive side, we can adjust the number of element moves to be at most $(1+\varepsilon)\lg n$, for any fixed constant $\varepsilon > 0$ and large enough n, while still achieving the desired bounds for the other operations.

Our work shows the limitation of the lower bounds proved by Gonnet and Munro [13] (see also [3]) in their prominent paper on binary heaps. They 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 [13], Carlsson [3] showed that $\lceil \lg n\rceil + \delta(n)$ element comparisons are necessary and sufficient to remove the minimum from a binary heap that has $n > 2^{h_{\delta(n)}+2}$ elements, where $h_1 = 1$ and $h_i = h_{i-1} + 2^{h_{i-1}+i-1}$. One should notice that these lower bounds are valid under the following assumptions:

(1) All the elements are stored in one nearly complete binary tree.
(2) Every node obeys the heap order before and after each operation.
(3) No order relation among the elements of the same level can be deduced from the element comparisons performed by previous operations.

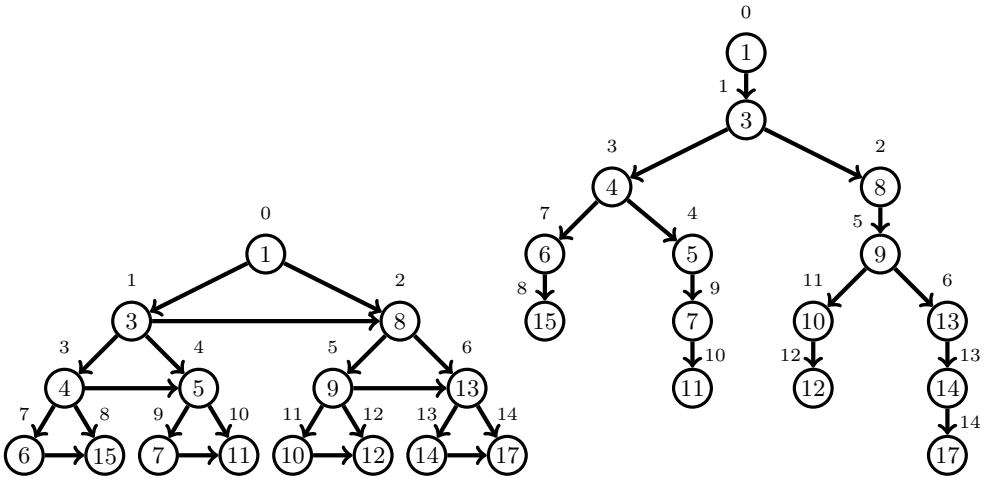

Fig. 1. A strong heap in an array $\mathbf{a}[0:14] = [1, 3, 8, 4, 5, 9, 13, 6, 15, 7, 11, 10, 12, 14, 17]$ viewed as a directed acyclic graph (left) and a stretched tree (right)

We prove that the number of element comparisons performed by *pop* can be lowered to at most $\lg n + O(1)$ if we overrule the third assumption by imposing an additional requirement that the element at any right child is not smaller than that at the left sibling (Sect. 2). We also prove that *push* can be performed in $O(1)$ worst-case time if we overrule the second assumption by allowing $O(\lg^2 n)$ nodes to violate heap order (Sect. 3). Lastly, we combine the two ideas and use them together in our final data structure (Sect. 4).

2 Strong Heaps: Adding More Order

A *strong heap* is a binary heap with one additional invariant: The element at any right child is not smaller than that at the left sibling. This left-dominance property is fulfilled for every right child in a fine heap [4] (or its alternatives [16, 18]), which uses one extra bit per node to maintain the property. Like a binary heap, a strong heap is viewed as a nearly complete binary tree where the lowest level may be missing some nodes at the rightmost (last) positions. Also, this tree is embedded in an array in the same way. If the array indexing starts at 0, the parent of a node at index i ($i \neq 0$) is at index $\lfloor (i-1)/2 \rfloor$, the left child (if any) at index $2i+1$, and the right child (if any) at index $2i+2$.

Two views of a strong heap are exemplified in Fig. 1. On the left, the directed acyclic graph has a nearly complete binary tree as its *skeleton*: There are arcs from every parent to its children and additional arcs from every left child to its sibling indicating the dominance relations. On the right, in the *stretched tree*, the arcs from each parent to its right child are removed as these dominance relations can be induced. In the stretched tree a node can have 0, 1, or 2 children. A node

```
method left-child(i)
return 2i + 1

method sibling(i)
if i = 0
|  return 0
return i + odd(i) − even(i)

method is-leaf(i, n)
if odd(i)
|  return sibling(i) ≥ n
return left-child(i) ≥ n
```

```
method strengthening-sift-down(i, n)
x ← a[i]
while not is-leaf(i, n)
|  j ← sibling(i)
|  if even(i)
|  |  j ← left-child(i)
|  else if j < n and left-child(i) < n and
|          a[left-child(i)] ≤ a[j]
|  |  j ← left-child(i)
|  if x ≤ a[j]
|  |  break
|  a[i] ← a[j]
|  i ← j
a[i] ← x
```

Fig. 2. Implementation of *strengthening-sift-down*; a right child is not accessed directly

has one child if in the skeleton it is a right child that is not a leaf or a leaf that has a right sibling. A node has two children if in the skeleton it is a left child that is not a leaf. If the skeleton has height h (height of a single node being 1), the height of the stretched tree is at most $2h - 1$, and on any root-to-leaf path in the stretched tree the number of nodes with two children is at most $h - 2$.

The basic primitive used in the manipulation of binary heaps is the *sift-down* procedure [12, 19] (see Fig. 4). This operation starts at a node that possibly breaks heap order, traverses down the heap by following the path of children containing the smaller of the elements at any two siblings, and moves the encountered elements one level up until the correct place of the element we started with is found. For strong heaps the *strengthening-sift-down* procedure has the same purpose, and our implementation (see Fig. 2) is similar, with one crucial exception that we operate with the stretched tree instead of the nearly complete tree. Now *pop* can be implemented by replacing the element at the root with the element at the last position of the array (if there is any) and then invoking *strengthening-sift-down* for the root.

Example 1. Consider the strong heap in Fig. 1. If its minimum was replaced with the element 17 taken from the end of the array, the path to be followed by *strengthening-sift-down* would include the nodes ⟨3, 4, 5, 7, 11⟩.

Let n denote the size of the strong heap and h the height of the underlying tree skeleton. When going down the stretched tree, we perform at most $h - 2$ element comparisons due to branching at binary nodes and at most $2h - 1$ element comparisons due to checking whether to stop or not. Hence, the number of element comparisons performed by *pop* is bounded by $3h - 3$, which is at most $3 \lg n$ as $h = \lfloor \lg n \rfloor + 1$.

To build a strong heap, we mimic Floyd's heap-construction algorithm [12]; that is, we invoke *strengthening-sift-down* for all nodes, one by one, processing

them in reverse order of their array positions. One element comparison is needed for every met left child in order to compare the element at its right sibling with that at its left child, making a total of at most $n/2$ element comparisons. The number of other element comparisons is bounded by the sum $\sum_{i=1}^{\lfloor \lg n \rfloor + 1} 3 \cdot i \cdot \lceil n/2^{i+1} \rceil$, which is at most $3n + o(n)$. Thus, *construct* requires at most $3.5n + o(n)$ element comparisons.

For both *pop* and *construct*, the amount of work done is proportional to the number of element comparisons performed, i.e. the worst-case running time of *pop* is $O(\lg n)$ and that of *construct* is $O(n)$.

Lemma 1. *A strong heap of size n can be built in $O(n)$ worst-case time by repeatedly calling strengthening-sift-down. Each strengthening-sift-down operation uses $O(\lg n)$ worst-case time and performs at most $3 \lg n$ element comparisons.*

Next we show how to perform a *sift-down* operation on a strong heap of size n with at most $\lg n + O(1)$ element comparisons. At this stage we allow the amount of work to be higher, namely $O(n)$. To achieve the better comparison bound, we have to assume that the heap is *complete*, i.e. that all leaves have the same depth. Consider the case where the element at the root of a strong heap is replaced by a new element. In order to reestablish strong heap order, the *swapping-sift-down* procedure (Fig. 3) traverses the left spine of the skeleton bottom up starting from the leftmost leaf, and determines the correct place of the new element, using one element comparison at each node visited. Thereafter, it moves all the elements above this position on the left spine one level up, and inserts the new element into this place. If this place is at level g, we have performed g element comparisons. Up along the left spine there are $\lg n - g + O(1)$ remaining levels to which we have moved other elements. While this results in a heap, we still have to reinforce the left-dominance property at these upper levels. In accordance, we compare each element that has moved up with the element at the right sibling. If the element at index j is larger than the element at index $j+1$, we interchange the subtrees T_j and T_{j+1} rooted at positions j and $j+1$ by swapping all their elements. The procedure continues this way until the root is reached.

Example 2. Consider the strong heap in Fig. 1. If the element at the root was replaced with the element 16, the left spine to be followed by *swapping-sift-down* would include the nodes ⟨(3), (4), (6)⟩, the new element would be placed at the last leaf we ended up with, the elements on the left spine would be lifted up one level, and an interchange would be necessary for the subtrees rooted at node (6) and its new sibling (5).

Given two complete subtrees of height h, the number of element moves needed to interchange the subtrees is $O(2^h)$. As $\sum_{h=1}^{\lfloor \lg n \rfloor} O(2^h)$ is $O(n)$, the total work done in the subtree interchanges is $O(n)$. Thus, *swapping-sift-down* requires at most $\lg n + O(1)$ element comparisons and $O(n)$ work.

Lemma 2. *In a complete strong heap of size n, swapping-sift-down runs in-place and uses at most $\lg n + O(1)$ element comparisons and $O(n)$ element moves.*

```
method parent(i)
if i = 0
  | return 0
return ⌊(i − 1)/2⌋

method bottom-up-search(i, j)
while j > i and a[j] ≥ a[i]
  | j ← parent(j)
return j

method swap-subtrees(u, v, n)
j ← 1
while v < n
  | for i ← 0, 1, ..., j − 1
  |   | swap(a[u + i], a[v + i])
  | u ← left-child(u)
  | v ← left-child(v)
  | j ← 2 ∗ j

method leftmost-leaf(i, n)
while left-child(i) < n
  | i ← left-child(i)
return i

method lift-up(i, j, n)
x ← a[j]
a[j] ← a[i]
while j > i
  | swap(a[parent(j)], x)
  | if a[sibling(j)] < a[j]
  |   | swap-subtrees(j, sibling(j), n)
  | j ← parent(j)

method swapping-sift-down(i, n)
k ← leftmost-leaf(i, n)
k ← bottom-up-search(i, k)
lift-up(i, k, n)
```

Fig. 3. Implementation of *swapping-sift-down*

3 Lazy Heaps: Buffering Insertions

In the variant of a binary heap that we describe in this section some nodes may violate heap order because insertions are buffered and unordered bulks are incrementally melded into the heap. The main difference between the present construction and the construction in [8] is that, for a heap of size n, here we allow $O(\lg^2 n)$ heap-order violations instead of $O(\lg n)$, but we still use $O(1)$ extra space to track where the potential violations are. Using *strengthening-sift-down* instead of *sift-down*, the construction will also work for strong heaps.

A *lazy heap* is composed of three parts: *main heap*, *submersion area*, and *insertion buffer*. The main heap together with the submersion area are laid out in the array as a binary heap, and the insertion buffer occupies the last array locations. The following rules are imposed:

(1) New insertions are appended to the insertion buffer at the end of the array.
(2) If the size of the main heap is n', the size of the insertion buffer is $O(\lg^2 n')$.
(3) When the insertion buffer becomes full, a proportion of its elements are treated as an embryo for a new submersion area.
(4) The submersion area is incrementally melded into the main heap by performing a constant amount of work in connection with every modifying operation (*push*/*pop*).
(5) When the insertion buffer is full again, the incremental submersion must have been completed.
(6) When the insertion buffer is empty, the incremental submersion must have been completed. When a *pop* is performed, a replacement element is taken from either the insertion buffer or the main heap (if the former is empty).

```
method sift-down(i, n)
x ← a[i]
while left-child(i) < n
    j ← left-child(i)
    if sibling(j) < n and a[sibling(j)] < a[j]
        j ← sibling(j)
    if x ≤ a[j]
        break
    a[i] ← a[j]
    i ← j
a[i] ← x
```

```
method submersion(n′, n)
r ← n − 1
ℓ ← max{n′, parent(r) + 1}
while r ≠ 0
    ℓ ← parent(ℓ)
    r ← parent(r)
    for i ← r, r − 1, . . . , ℓ
        sift-down(i, n)
```

Fig. 4. Implementation of *submersion*; n' is the size of the main heap and n the size of the main heap plus the submersion area; *sift-down* is from [12]

The insertion buffer should support insertions in $O(1)$ time, and minimum extractions in $O(\lg n)$ time using at most $\lg n + O(1)$ element comparisons. Let $t = \lfloor \lg(1+\lg(1+n')) \rfloor$. We treat the insertion buffer as a sequence of *chunks*, each of size $k = 2^t/4$, and limit the number of chunks to at most k. All the chunks, except possibly the last, will contain exactly k elements. The minimum of each chunk is kept at the first location of the chunk, and the index of the minimum of the buffer is maintained. When this minimum of the buffer is removed, the last element is moved into its place, the new minimum of that chunk is found in $O(k)$ time using $k-1$ element comparisons (by scanning the elements of the chunk), and 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). When *pop* needs a replacement for the old minimum, we have to consider the case where the last element is the minimum of the insertion buffer. In such a case, to avoid losing track of this minimum, before any processing, we swap it with the first element of the buffer. In *push*, a new element is appended to the insertion buffer. Subsequently, the minimum of the last chunk and the minimum of the buffer are adjusted if necessary; this requires at most two element comparisons. Once there are k full chunks, the first half of them are used to form a new submersion area and the elements therein are incrementally melded into the main heap.

The submersion area is treated as part of the main heap even though some of its nodes may not obey heap order. To reestablish heap order, the *submersion* operation (Fig. 4) will traverse the heap bottom up level by level as in Floyd's heap-construction algorithm [12]. Starting with the parents of the nodes containing the initial embryo of the submersion process, for each node we call the *sift-down* procedure. We then consider the parents of these nodes at the next upper level, restoring 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.

In the following analysis we separately consider two phases of the *submersion* procedure. The first phase comprises the *sift-down* calls for the nodes at the levels

with more than two involved nodes. Let b denote the size of the initial bulk. The number of the nodes visited at the jth last level is at most $\lfloor (b-2)/2^{j-1} \rfloor + 2$. For a node at the jth last level, a call to the *sift-down* subroutine requires $O(j)$ work. In the first phase, the amount of work involved is $O(\sum_{j=2}^{\lceil \lg n' \rceil} j/2^{j-1} \cdot b) = O(b)$. The second phase comprises at most $2\lfloor \lg n' \rfloor$ calls to the *sift-down* subroutine; this accounts for a total of $O(\lg^2 n')$ work. Since $b = \Theta(\lg^2 n')$, the overall work done is $O(\lg^2 n')$, i.e. amortized constant per *push*.

Instead of doing a submersion in one shot, we distribute the work by performing $O(1)$ work in connection with every modifying operation. Obviously, such a submersion should be done fast enough to complete before the insertion buffer becomes either full or empty.

To track the progress of the 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 its left and right endpoints, call them (ℓ_1, r_1) and (ℓ_2, r_2). These two intervals are at two consecutive levels, and the parent of the right endpoint of the first interval has an index that is one less than the left endpoint of the second interval, i.e. $\ell_2 - 1 = \lfloor (r_1 - 1)/2 \rfloor$. We say that these two intervals form the *frontier*. While the process advances, the frontier moves upwards and shrinks until it has one or two nodes. The frontier imparts that a *sift-down* is being performed starting from the node whose index is ℓ_2. In addition to the frontier, we also maintain the index of the node that the *sift-down* in progress is currently processing. In connection with every modifying operation, the current *sift-down* progresses a constant number of levels downwards and this index is updated. Once *sift-down* returns, the frontier is updated. When the frontier passes the root, incremental submersion is complete. To summarize, the information maintained to record the state of the *submersion* process is two intervals of indices to represent the frontier plus the node which is under consideration by the current *sift-down*.

As for the insertion buffer, we maintain the index of the minimum on the frontier. We treat each of the two intervals of the frontier as a set of consecutive chunks. Except for the first or 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 is kept at the entry storing the first node among the nodes of the chunk. An exception is the first and last chunks, where we maintain the index for the minimum on each.

To remove the minimum of the submersion area, we know that it must be on the frontier and we readily have its index. This minimum is swapped with the last element of the array and a *sift-down* is performed to remedy the order between the replacement element and the elements in its descendants. We distinguish between two cases:

(1) There are at most two nodes on the frontier.
(2) There are more than two nodes on the frontier.

In the first case, we make the minimum index of the frontier point to the smaller. In the second case, the height of the nodes on the frontier is at most $2 \lg \lg n +$

$O(1)$ so we can afford to do the following. The chunk that contained the removed minimum is scanned to find its new minimum. If this chunk is neither the first nor the last of the frontier, the found minimum is swapped with the element at its first position, followed by a *sift-down* performed on the latter element. The overall minimum of the frontier is then localized by scanning the minima of all the chunks. Extracting the minimum of the submersion area thus requires $O(\lg n)$ time and uses at most $1/2 \cdot \lg n + O(\lg \lg n)$ element comparisons.

In the main heap the *top* and *pop* operations are performed as in a binary heap with the same cost limitations. An exception is that, if *pop* meets the frontier of the submersion area, we stop the execution before crossing it.

To summarize, in a lazy heap, *top* reports the minimum of the three components, *push* is delegated to the insertion buffer, and *pop* is delegated to the component where the overall minimum resides.

Lemma 3. *In a lazy heap of size n, top and push require $O(1)$ worst-case time and pop requires $O(\lg n)$ worst-case time.*

4 Strengthened Lazy Heaps: Putting Things Together

Our final construction is similar to the one of the previous section in that there are three components: main heap, submersion area, and insertion buffer. Here the main heap has two layers: a *top heap* that is a binary heap, and each leaf of the top heap roots a *bottom heap* that is a complete strong heap. The main heap is laid out in the array as a binary heap, and in accordance every bottom heap is scattered throughout the array. As before, the submersion area is contained within the main heap, leading to a possible disobedience of heap order at its frontier. Because the main heap is only partially strong, we call the resulting data structure a *strengthened lazy heap*. Let n' be the size of the main heap, and let $t = \lfloor \lg(1 + \lg(1 + n')) \rfloor$. The height of the bottom heaps is either $t - 3$ and $t - 2$, or $t - 2$ and $t - 1$. In the insertion buffer, the size of a chuck is $k = 2^t/4$ and the size of the buffer is bounded by k^2. To help the reader get a complete picture of the data structure, we visualize it in Fig. 5.

We use a new procedure, that we call *combined-sift-down* (Fig. 6), instead of *sift-down*. Assume we have to replace the minimum of the top heap with another element. To reestablish heap order, we follow the proposal of Carlsson [2]: We traverse down along the path of nodes containing the smaller of the elements at any two siblings until we reach a root of a bottom heap. By comparing the replacement element with the element at that root, we check whether the replacement element should land in the top heap or in the bottom heap. In the first case, in *binary-search-sift-up* we find the position of the replacement element using binary search on the traversed path and thereafter do the required element moves. In the second case, we apply *swapping-sift-down* on the root of the bottom heap.

Let us now recap how the operations are executed and analyse their performance. Here we ignore the extra work done due to the incremental processes.

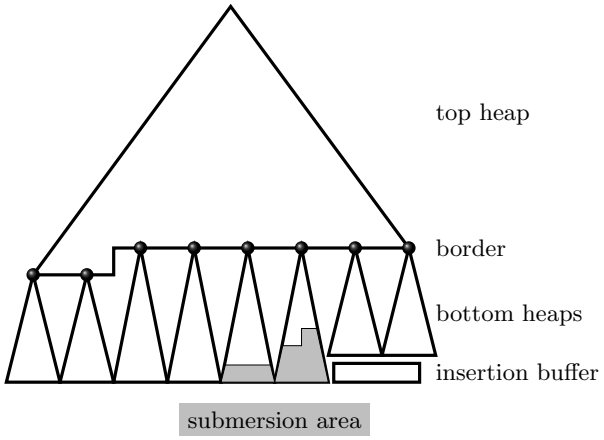

Fig. 5. Schematic view of a strengthened lazy heap

```
method ancestor(i, d)
return ⌊(i + 1)/2^d⌋ − 1

method rotate(i, k, h)
x ← a[i]
for d ← h − 1, h − 2, ..., 0
    a[ancestor(k, d + 1)] ← a[ancestor(k, d)]
a[k] ← x

method correct-place(i, k, h)
d ← h
while i ≠ k
    h′ ← ⌊(h + 1)/2⌋
    j ← ancestor(k, h′)
    h ← h − h′
    if a[i] ≤ a[j]
        k ← j
        d ← d − h′
    else
        i ← ancestor(k, h)
return (i, d)

method binary-search-sift-up(i, k, h)
(j, d) ← correct-place(i, k, h)
rotate(i, j, d)

method combined-sift-down(i, n, h)
j ← i
repeat h times
    k ← left-child(j)
    if a[sibling(k)] < a[k]
        k ← sibling(k)
    j ← k
if a[i] ≤ a[j]
    binary-search-sift-up(i, parent(j), h − 1)
else
    rotate(i, j, h)
    swapping-sift-down(j, n)
```

Fig. 6. Implementation of *combined-sift-down*

Clearly, *top* can be carried out in $O(1)$ worst-case time by reporting the minimum of three elements:

(1) the element at the root of the top heap,
(2) the minimum of the insertion buffer, and
(3) the minimum of the submersion area.

As before, *push* appends the given element to the insertion buffer and updates the minimum of the buffer if necessary. To perform *pop*, we need to consider the different minima and remove the smallest among them.

Case 1. If the minimum is at the root of the top heap, we find a replacement for the old minimum and apply *combined-sift-down* for the root by making sure that we do not cross the frontier. Let n denote the total number of elements. The top heap is of size $O(n/\lg n)$ and the bottom heaps are of size $O(\lg n)$. To reach the root of a bottom heap, we perform $\lg n - \lg\lg n + O(1)$ element comparisons. If we have to go upwards, we perform $\lg\lg n + O(1)$ additional element comparisons in the binary search while applying the *binary-search-sift-up* operation. On the other hand, if we have to go downwards, *swapping-sift-down* needs to perform at most $\lg\lg n + O(1)$ element comparisons. In both cases, the number of element comparisons performed is at most $\lg n + O(1)$ and the work done is $O(\lg n)$.

Case 2. If the overall minimum is in the insertion buffer, it is removed as explained in the previous section. This removal involves $2k + O(1)$ element comparisons and the amount of work done is proportional to that number. Since we have set $k = 2^t/4 = 1/4 \cdot \lg n + O(1)$, this operation requires at most $1/2 \cdot \lg n + O(1)$ element comparisons and $O(\lg n)$ work.

Case 3. If the frontier contains the overall minimum, we apply a similar treatment to that explained in the previous section with a basic exception. If there are more than two nodes on the frontier, the height of the nodes on the frontier is at most $2\lg\lg n + O(1)$. In this case, we use the *strengthening-sift-down* procedure in place of the *sift-down* procedure. This requires at most $1/2 \cdot \lg n + O(\lg\lg n)$ element comparisons and $O(\lg n)$ work. If there are at most two nodes on the frontier, the frontier lies in the top heap. In this case, we apply the *combined-sift-down* procedure instead. This requires at most $\lg n + O(1)$ element comparisons and $O(\lg n)$ work. Either way, for large enough n, the minimum extraction here requires at most $\lg n + O(1)$ element comparisons.

Because of the subtree interchanges made in *swapping-sift-down*, the number of element moves performed by *pop*—even though asymptotically logarithmic—would be larger than the number of element comparisons. Assume that the number of these moves is bounded by $c \lg n$ for some constant c. We can control the number of element moves by adjusting the heights of the bottom heaps. If the maximum height of a bottom heap is set to $t - \lg(c/\varepsilon)$ for some small constant ε, $0 < \varepsilon \leq c$, the number of element moves performed therein will be bounded by $\varepsilon \lg n + O(1)$, while the bounds for the other operations still hold.

Due to the two-layer structure, the incremental remedy processes are more complicated for a strengthened lazy heap than for a lazy heap. Let us consider the introduced complications one at a time and sketch how we handle them.

Complication 1. As the size of the heap changes due to insertions and deletions, we have to move the *border* between the two layers dynamically. To make the bottom heaps one level shallower, we just adjust t and ignore the left-domination property for the nodes on the previous border. To make the bottom heaps one level higher, we need a new incremental remedy process that scans the nodes on the old border and applies *strengthening-sift-down* on each left

child. Again, we only need a constant amount of space to record the state of this process. The total work done in the border lifting is linear so, after the process is initiated, every forthcoming modifying operation has to take a constant share of the work.

There are several special cases to consider.

(1) If *pop* meets the node processed by border-lifting *strengthening-sift-down*, we stop the execution of *pop* and let the incremental process reestablish strong heap order below that node.
(2) If the node where border-lifting *strengthening-sift-down* is to be applied is inside the submersion area, we stop this corrective action and jump to the next since the submersion process has already establish strong heap order below that node.
(3) If the node processed by submersion *strengthening-sift-down* and that by border-lifting *strengthening-sift-down* meet, we stop the border-lifting process and jump to the next since the submersion process will reestablish strong heap order below that node.
(4) If the border-lifting *strengthening-sift-down* meets the frontier, we stop this corrective action before crossing it and jump to the next.
(5) Also, when the node recorded by border-lifting *strengthening-sift-down* is moved by a *swapping-sift-down*, the index to this node is to be updated accordingly.

Complication 2. When extracting the minimum, we use the last element of the insertion buffer as a replacement. However, if the insertion buffer is empty, meaning that the submersion process must have been completed, we need to use an element from the main heap instead. To keep the bottom heaps complete, we move all the elements at the lowest level of the bottom heap that occupies the rear of the array back to the empty insertion buffer. After such a move, the minimum of this piece is not known. Fortunately, we do not need this minimum within the next k *pop* operations, as there are at least a logarithmic number of elements in the main heap that are smaller. Hence, the minimum of the involved chunks can be found incrementally within the upcoming k modifying operations.

Complication 3. If we swapped two subtrees in the bottom heap where the frontier consists of two intervals, there is a risk that we mess up the frontier. Hence, we schedule the submersion process differently: We process the bottom heaps one by one, and lock the bottom heap under consideration to skip subtree interchanges initiated by *pop* in the main heap. Therefore, when the frontier overlaps the bottom heaps, it is cut into several pieces:

(1) the interval corresponding to the unprocessed leaves of the initial bulk,
(2) the two intervals (ℓ_1, r_1) and (ℓ_2, r_2) in the bottom heap under consideration, and
(3) the interval of the roots of the bottom heaps that have been handled by the submersion process.

Locking resolves the potential conflict with *pop*. However, in the currently processed bottom heap there are some nodes between the root and the frontier that

are not yet included in the submersion process and are not in order with the elements above or below. This is not a problem, as none of these elements can be the minimum of the heap except after a logarithmic number of modifying operations. Within such time, these nodes have already been handled by the submersion process.

5 Conclusions

We described a priority queue that

(1) operates in-place,
(2) supports *top* and *push* in $O(1)$ worst-case time, and
(3) supports *pop* in $O(\lg n)$ worst-case time involving at most $\lg n + O(1)$ element comparisons.

The data structure is asymptotically optimal with respect to time, and optimal up to additive constant terms with respect to space and element comparisons.

The related contributions prior to this work can be summarized as follows:

(1) break the $2 \lg n + O(1)$ barrier for the number of element comparisons performed per *pop* when *push* takes $O(1)$ worst-case time [9],
(2) achieve the aforementioned desired bounds using $O(n)$ words of extra space [10],
(3) achieve the desired bounds using $O(n)$ bits of extra space [8],
(4) achieve the desired bounds in-place in the amortized sense [11].

It is remarkable that we could surpass the two lower bounds known for binary heaps [13] by slightly loosening the assumptions that are intrinsic to these lower bounds. To achieve our goals, we simultaneously imposed more order on some nodes, by forbidding some elements at left children to be larger than those at their right siblings, and less order on others, by allowing some elements to possibly be smaller than those at the parents.

In retrospect, we admit that, while binary heaps [19] are practically efficient, our data structure is somewhat impracticable. A solution [11] that achieves the same bounds in the amortized sense is simpler, but our reference implementation is still not competitive with binary heaps. The main questions left open are

(1) whether the number of element moves performed by *pop* can be reduced to $\lg n + O(1)$,
(2) whether our constructions could be simplified, and
(3) whether there are components that are useful in practice.

References

1. Brown, M.R.: Implementation and analysis of binomial queue algorithms. SIAM J. Comput. **7**(3), 298–319 (1978)

2. Carlsson, S.: A variant of Heapsort with almost optimal number of comparisons. Inform. Process. Lett. **24**(4), 247–250 (1987)
3. Carlsson, S.: An optimal algorithm for deleting the root of a heap. Inform. Process. Lett. **37**(2), 117–120 (1991)
4. Carlsson, S., Chen, J., Mattsson, C.: Heaps with bits. Theoret. Comput. Sci. **164**(1–2), 1–12 (1996)
5. Carlsson, S., Munro, J.I., Poblete, P.V.: An implicit binomial queue with constant insertion time. In: Karlsson, R., Lingas, A. (eds.) SWAT 1988. LNCS, vol. 318, pp. 1–13. Springer, Heidelberg (1988)
6. Chen, J., Edelkamp, S., Elmasry, A., Katajainen, J.: In-place heap construction with optimized comparisons, moves, and cache misses. In: Rovan, B., Sassone, V., Widmayer, P. (eds.) MFCS 2012. LNCS, vol. 7464, pp. 259–270. Springer, Heidelberg (2012)
7. Dutton, R.D.: Weak-heap sort. BIT **33**(3), 372–381 (1993)
8. Edelkamp, S., Elmasry, A., Katajainen, J.: Weak heaps engineered. J. Discrete Algorithms **23**, 83–97 (2013). Presented at IWOCA 2012
9. Elmasry, A.: Layered heaps. In: Hagerup, T., Katajainen, J. (eds.) SWAT 2004. LNCS, vol. 3111, pp. 212–222. Springer, Heidelberg (2004)
10. Elmasry, A., Jensen, C., Katajainen, J.: Multipartite priority queues. ACM Trans. Algorithms **5**(1), 14:1–14:19 (2008)
11. Elmasry, A., Katajainen, J.: Towards ultimate binary heaps. CPH STL Report 2013-1, Department of Computer Science, University of Copenhagen (2013)
12. Floyd, R.W.: Algorithm 245: Treesort 3. Commun. ACM **7**(12), 701 (1964)
13. Gonnet, G.H., Munro, J.I.: Heaps on heaps. SIAM J. Comput. **15**(4), 964–971 (1986). Presented at ICALP 1982
14. Katajainen, J.: The ultimate heapsort. In: Lin, X. (ed.) CATS 1998. Australian Computer Science Communications, vol. 20, pp. 87–95. Springer, Singapore (1998)
15. Knuth, D.E.: The Art of Computer Programming: Sorting and Searching, vol. 3, 2nd edn. Addison Wesley Longman, Reading (1998)
16. McDiarmid, C.J.H., Reed, B.A.: Building heaps fast. J. Algorithms **10**(3), 352–365 (1989)
17. Vuillemin, J.: A data structure for manipulating priority queues. Commun. ACM **21**(4), 309–315 (1978)
18. Wegener, I.: The worst case complexity of McDiarmid and Reed's variant of Bottom-up Heapsort is less than $n \log n + 1.1n$. Inform. and Comput. **97**(1), 86–96 (1992). Presented at STACS 1991
19. Williams, J.W.J.: Algorithm 232: Heapsort. Commun. ACM **7**(6), 347–348 (1964)