# Relaxed Weak Queues: An Alternative to Run-Relaxed Heaps

Amr Elmasry[1]      Claus Jensen[2,*]      Jyrki Katajainen[2,*]

[1] *Computer Science Department, Alexandria University*
*Alexandria, Egypt*
[2] *Department of Computing, University of Copenhagen*
*Universitetsparken 1, 2100 Copenhagen East, Denmark*

**Abstract.** A simplification of a run-relaxed heap, called a *relaxed weak queue*, is presented. This new priority-queue implementation supports all operations as efficiently as the original: *find-min*, *insert*, and *decrease* (also called *decrease-key*) in $O(1)$ worst-case time, and *delete* in $O(\lg n)$ worst-case time, $n$ denoting the number of elements stored prior to the operation. These time bounds are valid on a pointer machine as well as on a random-access machine. A relaxed weak queue is a collection of at most $\lfloor \lg n \rfloor + 1$ perfect weak heaps, where there are in total at most $\lfloor \lg n \rfloor + 1$ nodes that may violate weak-heap order. In a pointer-based representation of a perfect weak heap, which is a binary tree, it is enough to use two pointers per node to record parent-child relationships. Due to *decrease*, each node must store one additional pointer. The auxiliary data structures maintained to keep track of perfect weak heaps and potential violation nodes only require $O(\lg n)$ words of storage. That is, excluding the space used by the elements themselves, the total space usage of a relaxed weak queue can be as low as $3n + O(\lg n)$ words.

**ACM CCS Categories and Subject Descriptors.** E.1 [Data Structures]: Lists, stacks, and queues; E.2 [Data Storage Representations]: Linked representations; F.2.2 [Analysis of Algorithms and Problem Complexity]: sorting and searching

**Keywords.** Data structures, priority queues, binary heaps, weak heaps, binomial queues, relaxed heaps

## 1. Introduction

In a generic form, a priority queue (or a heap as it is called in some texts) is a data structure which depends on four type parameters:

$\mathcal{E}$: the type of the *elements* manipulated;

$\mathcal{F}$: the type of the *ordering relation* used in element comparisons;

$\mathcal{C}$: the type of the *compartments* used for storing the elements, but also for storing satellite data, like references to other compartments; and

---

$\mathcal{A}$: the type of the *allocator* which provides methods for allocating new compartments and deallocating old compartments.

When dealing with element comparisons, we use the terms *smaller than* and *greater than*, and the symbols $<$ and $>$, as if the elements were integers. Furthermore, we assume that the elements can be moved and compared in $O(1)$ time; that it is possible to get any information stored at a compartment in $O(1)$ time; and that both allocation and deallocation of compartments take $O(1)$ time.

A *priority queue* $Q$ with type parameters $\langle \mathcal{E}, \mathcal{F}, \mathcal{C}, \mathcal{A} \rangle$ stores a dynamic collection of elements and provides the following set of methods for the manipulation of this collection:

$\mathcal{C}$ *find-min*(). Return the compartment of a minimum element stored in $Q$. The minimum is taken with respect to $\mathcal{F}$. **Requirement.** $Q$ is not empty. The compartment returned is passed by reference.

$\mathcal{C}$ *insert*($\mathcal{E}$ $e$). Insert element $e$ into $Q$ and return its compartment for later use. **Requirement.** There is enough space available to accomplish this operation. Both $e$ and the returned compartment are passed by reference.

*void delete*($\mathcal{C}$ $p$). Remove both the element stored at compartment $p$ and compartment $p$ from $Q$, and return nothing as specified by the type *void* of the return value. **Requirement.** $p$ is a valid compartment.

*void decrease*($\mathcal{C}$ $p$, $\mathcal{E}$ $e$). Replace the element stored at compartment $p$ with element $e$. **Requirement.** $p$ is a valid compartment. $e$ is no greater than the old element stored at $p$. Both $p$ and $e$ are passed by reference.

If desired, the method *delete-min*, which removes the current minimum, can be carried out by invoking *find-min* and thereafter *delete* with the compartment returned by *find-min*. Some additional methods — like a constructor, a destructor, and a set of methods for examining the number of elements stored in $Q$ — are necessary to make the data structure useful, but these are computationally less interesting and therefore not considered here.

In this paper, we are interested in realizations of a priority queue which are efficient in the worst-case sense. In particular, our target is a data structure that supports all priority-queue operations as efficiently as a run-relaxed heap [7]: *find-min*, *insert*, and *decrease* in $O(1)$ worst-case time, and *delete* in $O(\lg n)$ worst-case time, $n$ denoting the number of elements stored prior to the operation and $\lg n$ being a shorthand for $\max\{1, \log_2 n\}$. Other data structures having the same performance are Brodal's heaps [2], which can even meld two priority queues in $O(1)$ worst-case time, and fat heaps [13, 14]. Run-relaxed heaps, Brodal's heaps, and fat heaps are complicated so they are rarely described in textbooks on data structures and algorithms. In [13, 14], where fat heaps were introduced, an attempt was already made to provide a simpler alternative to the other two data structures. Several other, less efficient but also less complicated, priority-queue structures are known, e.g. binary heaps [21] (see also [6, Chapter 6]), leftist trees (see [16, Section 5.2.3] or [19, Section 3.3]), binomial queues [1, 20] (called binomial heaps in [6, Chapter 19]), and ranked priority queues [12].
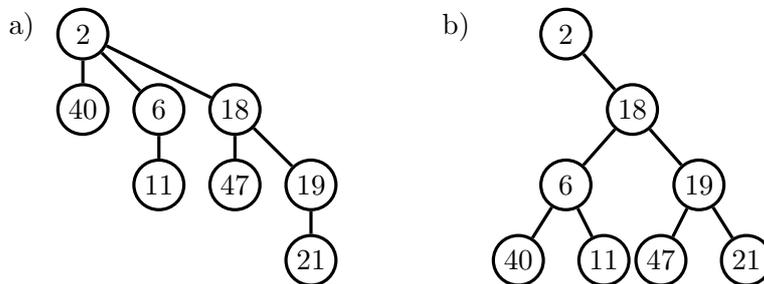
**Figure 1.** a) A heap-ordered binomial tree of rank 3 and b) the corresponding perfect weak heap of height 3.

In a tree, the compartment storing one element is called a *node*. In relaxed heaps [7], the building blocks used are heap-ordered binomial trees [1, 20]:

1. Let $h$ be a nonnegative integer. For $h = 0$ a *binomial tree* of *rank h* is a single node, and for $h > 0$ a *binomial tree* of *rank h* is composed of the root and its binomial subtrees of rank $0, 1, \ldots, h - 1$ in this order.

2. In a *heap-ordered tree*, the element stored at a node is no smaller than the element stored at the parent of that node (i.e. we are considering min-heaps).

A heap-ordered binomial tree is a multiary tree, but it can be transformed into a binary tree using the standard child-sibling transformation (see, e.g. [15, Section 2.3.2]). The outcome of this transformation is a *perfect* weak heap, i.e. a weak heap storing $2^h$ elements for some integer $h \geq 0$. Normally, weak heaps are defined in a more general form where the number of elements stored does not need to be a power of two (see, e.g. [8, 9]).

A *perfect weak heap* could be defined directly without referring to the corresponding heap-ordered binomial tree as follows:

1. The root has no left subtree.
2. The right subtree of the root is a complete binary tree.
3. For each node, every element stored in the right subtree of that node is no smaller than the element stored at that node.

This definition is illustrated in Figure 1. The definition immediately implies that the root must store a minimum element. In a perfect weak heap, we use the term *subheap* to denote a node together with its right subtree. Note that the *height* of a *subheap* (or the root of that subheap) corresponds to the rank of the corresponding subtree in a heap-ordered binomial tree.

Usually weak heaps are represented using an array representation with extra bits [8, 9], but a pointer-based representation is more flexible when subheaps are frequently relocated, as it is the case in our algorithms. Instead of operating on a single weak heap, we maintain a collection of disjoint perfect weak heaps in the same way as one maintains a collection of heap-ordered binomial trees in a binomial queue [20]. We call such a data structure a *weak queue*. Because of *delete* and *decrease* the external references to nodes

inside the data structure must be kept valid. We provide this by not moving the elements after they have been placed into their respective nodes. By employing other ideas earlier applied for binomial queues (see, for example, [10]), we can make *find-min* and *insert* take $O(1)$ worst-case time, and *delete* $O(\lg n)$ worst-case time.

To support *decrease* in $O(1)$ worst-case time, the basic idea, introduced by Driscoll et al. [7] and applied to a binomial queue, is to permit some nodes to violate heap order, i.e. the element stored at a *potential violation node* may be smaller than the element stored at the parent of that node (though this does not need to be the case). A *run* is a sequence of two or more potential violation nodes that are consecutive siblings. Driscoll et al. described two forms of relaxed heaps: a rank-relaxed heap and a run-relaxed heap. Both of them contain at most $\lfloor \lg n \rfloor + 1$ almost heap-ordered binomial trees with at most $\lfloor \lg n \rfloor + 1$ potential violation nodes, $n$ denoting the number of elements stored. We adapt the relaxation technique used for a run-relaxed heap to a weak queue, and call the resulting data structure a *relaxed weak queue*.

For a rank-relaxed heap, the $O(1)$ time bound for *decrease* is only amortized. However, the transformations needed for reducing the number of potential violation nodes, when there are too many of them, are simpler than those required by a run-relaxed heap. The basic observation made in this paper is that the relaxation technique used for a run-relaxed heap can be naturally migrated from multiary trees to binary trees. Moreover, the transformations needed for reducing the number of potential violation nodes can be decomposed into simple primitives that are conceptually similar to those required by a rank-relaxed heap. At the same time, for all priority-queue operations the worst-case time bounds achieved by a relaxed weak queue are the same as those achieved by a run-relaxed heap. Our time bounds are valid on a pointer machine (as defined, for example, in [17]), similarly to those achieved by a fat heap. However, the amount of extra space used by a relaxed weak queue can be made smaller than that required by other realizations of a worst-case efficient priority queue.

The structure of the rest of this paper is as follows. The auxiliary data structures used to keep track of perfect weak heaps and potential violation nodes are described and analysed in Section 2. Realizations of all relaxed-weak-queue operations are described and analysed in Section 3. Finally, a few concluding remarks are made in Section 4.

## 2. Auxiliary data structures

A weak queue is similar to a binomial queue, but it relies on binary trees instead of multiary trees. A relaxed weak queue is a collection of perfect weak heaps where some nodes are denoted to be potential violation nodes. In this section, we describe the auxiliary data structures, which we call the *heap store* and the *node store*, used to keep track of perfect weak heaps and potential violation nodes, respectively.
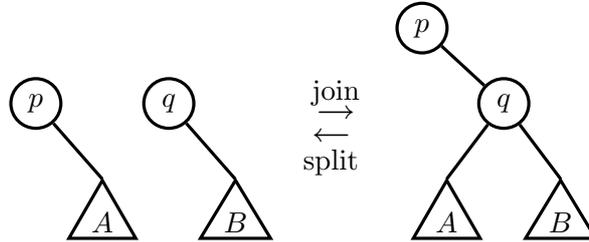
**Figure 2.** Joining and splitting two perfect weak heaps of the same size.

To store a perfect weak heap, it is natural to use three pointers per node, one pointing to the left child, one to the right child, and one to the parent. In addition to this, each node should store one element and a pointer to the node store. Through this additional pointer we can also indirectly get the height and the type of that node. A root is distinguished from other nodes by letting its parent pointer be null. For each root, the pointer to its nonexisting left child is reused to point to the heap store.

We refer to the number of elements stored in a weak heap or a weak queue as its *size*. As for heap-ordered binomial trees, two perfect weak heaps of the same size can easily be linked together (see Figure 2). Later on, we refer to this operation as a *join*. Let $(p, A)$ and $(q, B)$ be the two perfect weak heaps to be joined, where $p$ and $q$ are their roots, and $A$ and $B$ are the right subtrees of these roots, respectively. Furthermore, let $element[p]$ denote the element stored at $p$. If $element[p] \not> element[q]$, $p$ is made the root of the resulting perfect weak heap, $q$ is made the right child of $p$, $A$ is made the left subtree of $q$, and $B$ is kept as the right subtree of $q$. If $element[p] > element[q]$, the same is done, but the roles of the two perfect weak heaps are interchanged. A *split* is the inverse of a join (as illustrated in Figure 2), where the subheap rooted at the right child of the root is unlinked from the given perfect weak heap. Clearly, both a join and a split take $O(1)$ worst-case time.

*Heap store*

There are several alternative ways of implementing a heap store. The array-based scheduling technique used in a run-relaxed heap [7], the array-based counting mechanism used in Brodal's heap [2], or the pointer-based counting mechanism used in a fat heap [13, 14] would all work equally well. The solution described here is implicitly a redundant binary counter, but it is a stack-like structure that needs a restricted form of random access, which can easily be provided by a pointer machine.

As an abstract data structure, a heap store is an ordered sequence of perfect weak heaps, where the perfect weak heaps appear in increasing order of height and where two equal-sized perfect weak heaps are joined whenever

those appear. This would guarantee that at all times the number of perfect weak heaps is bounded above by $\lfloor \lg n \rfloor + 1$, $n$ being the size of the relaxed weak queue in question. To avoid cascading joins, injections of perfect weak heaps into the heap store will be carried out lazily, by not doing all possible joins at once. Therefore, our realization of a heap store consists of two parts: a *heap sequence* containing the perfect weak heaps in increasing order of height and a *join schedule* containing references to all delayed joins.

Let $\mathcal{H}$ denote the type of a perfect weak heap. To support the priority-queue operations for a relaxed weak queue, a heap store $S$ should provide the following methods:

*void inject*($\mathcal{H}$ *H*)**.** Insert perfect weak heap $H$ into $S$. **Requirement.** $H$ should not be larger in size than any perfect weak heap stored in $S$. $H$ is passed by reference.

$\mathcal{H}$ *eject*()**.** Remove and return a perfect weak heap of the smallest size from $S$. **Requirement.** $S$ is not empty. The returned perfect weak heap is passed by reference.

*void replace*($\mathcal{H}$ $H_1$, $\mathcal{H}$ $H_2$)**.** Replace the perfect weak heap $H_1$ by another perfect weak heap $H_2$ of the same size. **Requirement.** Both $H_1$ and $H_2$ are passed by reference.

A concrete representation of a perfect weak heap is simply a pointer to its root and the size of that perfect weak heap can be deduced from the height available at the root. A concrete representation of a heap sequence is a doubly-linked list, where each list item has a pointer to the root of a perfect weak heap. Correspondingly, the root has a pointer back to this heap-list item. A concrete representation of a join schedule is a stack, where each stack item has a pointer to a heap-list item containing the first of the two consecutive perfect weak heaps of the same size, the join of which is delayed.

Let $n$ denote the size of the relaxed weak queue $Q$ under consideration. There is a close connection between the sizes of the perfect weak heaps stored in $Q$ and the binary representation of $n$. If the normal binary representation is used, $Q$ stores a perfect weak heap of size $2^h$ if and only if there is a 1-bit in position $h$ of the binary representation of $n$. A join of two perfect weak heaps of size $2^h$ produces a perfect weak heap of size $2^{h+1}$, which corresponds to the addition of two 1-bits in position $h$ and the propagation of a carry to position $h + 1$. Instead of relying on the normal binary system, we allow each bit position to contain digits 0, 1, or 2. A heap sequence can be seen as a compressed representation of such a number. Digit 2 means that there are two consecutive perfect weak heaps of the same size, digit 1 means that there is only one perfect weak heap of that particular size, and zeros are not represented at all.

To make the connection more precise, we need the following formal definitions. In a *redundant binary system*, a nonnegative integer $d$ is represented as a sequence of digits $\langle d_0 d_1 \dots d_k \rangle$, $d_0$ being the least significant digit, $d_k$ the most significant digit, and $d_i \in \{0, 1, 2\}$ for all $i \in \{0, 1, \dots, k\}$, such that $d = \sum_{i=0}^{k} d_i \cdot 2^i$. For notational convenience, we implicitly assume that

$d_{-1} = 0$. The redundant binary representation $\langle d_{-1}d_0 \ldots d_k \rangle$ of $d$ is said to be *regular* if any digit 2 is preceded by a digit 0, possibly having a sequence of 1's in between. A subsequence of $\langle d_{-1}d_0 \ldots d_k \rangle$, which is of the form $\langle 01^\alpha 2 \rangle$ for $\alpha \in \{0, 1, \ldots, k\}$, is called a *block*. That is, every digit 2 must be part of a block, but there can be digits, 0's and 1's, that are not part of a block. For example, digit sequence $\langle 102012 \rangle$ (least significant digit given first) represents integer 89 and contains two blocks, one starting from position 1 and another starting from position 3.

Let us now consider how the heap-store operations can be carried out. Each *inject* involves two steps. First, in a *fixup step* we join the first two perfect weak heaps that are of the same size, if there are any. The perfect weak heaps to be joined are found by consulting the join schedule. More concretely, the pointer at top of the join schedule is popped, the two consecutive heaps pointed to are removed from the heap sequence, the perfect weak heaps are joined, and the resulting perfect weak heap is put in the place of the two in the heap sequence. If there exists another perfect weak heap of the same size as the resulting weak heap, a pointer indicating this pair is pushed onto the join schedule. Second, in an *increase step* we inject the given perfect weak heap into the heap sequence and, if there is another perfect weak heap of the same size, a pointer to this pair is pushed onto the join schedule. Each *eject* takes the first perfect weak heap, which must be the smallest, from the heap sequence and, if this perfect weak heap forms a pair with some other perfect weak heap, top of the join schedule is also popped. To execute *replace*$(H_1, H_2)$, the pointer at the root of $H_1$ is used to access the corresponding heap-list item. When this is available, the pointer from that item can be updated to point to $H_2$ and the pointer at the root of $H_2$ to point back to that heap-list item.

The correctness of *eject* and *replace* is obvious. As to the correctness of *inject*, the key observation is that the stack-based scheduling of delayed joins guarantees that the representation of the number corresponding to the heap sequence remains regular. That is, when the given perfect weak heap is injected into the heap store, it is necessary to perform at most one join. The proof of the key observation is given in [5, p. 53 ff.] in the case that only increments of the least significant digit are allowed. It is easy to extend the proof for the more general form of increments considered here. In order to keep this paper self-contained, we prove the claim in the following lemma.

**Lemma 1.** *Each inject keeps the representation of the number corresponding to the heap sequence regular.*

**Proof.** Let us first prove that the fixup step propagates a carry properly and keeps the representation of the number corresponding to the heap sequence regular. This is seen to be valid by a case analysis. Assume that the digit being fixed up is $d_j$. Now we consider three cases depending on the state of digit $d_{j+1}$ following $d_j$.

**Case 1.** The block ending at $d_j$ is followed by digit 0, which is not part of a block. After fixing $d_j$, $d_j = 0$ and $d_{j+1} = 1$. This destroys the block and the representation remains regular.

**Case 2.** The block ending at $d_j$ is followed by another block. This means that after the fixup step $d_j = 0$ and $d_{j+1} = 1$. This destroys the block that ended at $d_j$ and makes the other block one digit longer, but the representation remains regular.

**Case 3.** The block ending at $d_j$ is followed by digit 1. In particular, $d_{j+1}$ cannot be part of a block. After the fixup step, $d_{j+1} = 2$ and $d_j = 0$. This destroys the block that ended at $d_j$ and creates a new block of length two, and the representation remains regular.

Let us now prove that the increment step keeps the representation of the number corresponding to the heap sequence regular. Assume that the increment step involves digit $d_i$, and let $d_{i+1}$ be its immediate neighbour. Note that digit $d_i$ cannot be 2 since the first delayed join was propagated forward by the preceding fixup step. Moreover, if after the fixup step $d_i = 1$, $d_i$ cannot be part of a block since the first block was destroyed by the fixup step and the first member of the destroyed block cannot be part of a new block possibly created. Based on these observations it is sufficient to consider the following five cases.

**Case 1.** $d_i = 0$, $d_i$ is not part of a block, and $d_{i+1} = 0$. After the increment step, $d_i = 1$ which must keep the representation regular even if $d_{i+1}$ was part of a block.

**Case 2.** $d_i = 0$ and $d_i$ is part of a block. Thus, $d_{i+1}$ must be 1 or 2. According to the precondition $d_{i-1} = 0$, so the increment step keeps the representation regular.

**Case 3.** $d_i = 0$, $d_i$ is not part of a block, and $d_{i+1} = 1$. Thus, $d_{i+1}$ cannot be part of a block, and the increment step keeps the representation regular.

**Case 4.** $d_i = 1$ and $d_{i+1} \in \{0, 1\}$ and neither of them are part of a block. After the increment step, $d_i = 2$ and according to the precondition $d_{i-1} = 0$, meaning that the representation remains regular.

**Case 5.** $d_i = 1$, $d_{i+1} = 0$, and $d_{i+1}$ is part of a block. As in the previous case, after the increment step $d_i = 2$ and according to the precondition $d_{i-1} = 0$, which means that the representation remains regular. $\square$

The regularity is also essential to get an upper bound on the number of perfect weak heaps that a relaxed weak queue can contain.

**Lemma 2.** *Let $\tau$ denote the number of perfect weak heaps in a relaxed weak queue storing $n$ elements. It must hold that $\tau \leq \lfloor \lg n \rfloor + 1$.*

**Proof.** In the redundant binary system, the binary representation of number $n$ has at most $\lfloor \lg n \rfloor + 1$ nonzero digits. Therefore, the height of any of the perfect weak heaps stored must be between 0 and $\lfloor \lg n \rfloor$. The heap sequence can be interpreted as a regular redundant binary representation of number $n$, where any digit 2 must be preceded by a digit 0, except perhaps the first 2. This would immediately give us an upper bound $\lfloor \lg n \rfloor + 2$ on $\tau$, the number of perfect weak heaps stored.

Suppose that $\tau$ is as large as $\lfloor \lg n \rfloor + 2$. This would mean that there must be at least one digit 2 in the redundant binary representation of $n$. Assume

that the representation of $n$ is $\left\langle n_0, n_1, \ldots, n_{\lfloor \lg n \rfloor} \right\rangle$ and that $n_i$ is the last 2. If $n_j = 1$ for each $j \in \{i+1, \ldots, \lfloor \lg n \rfloor\}$, $n$ must be larger than or equal to $2 \cdot 2^i + \sum_{j=i+1}^{\lfloor \lg n \rfloor} 2^j = 2^{\lfloor \lg n \rfloor + 1}$, which is impossible. That is, there must exist an index $k$, $k \in \{i+1, \ldots, \lfloor \lg n \rfloor\}$, such that $n_k = 0$. We conclude that on average there can be at most one perfect weak heap per every possible height, which establishes the claim. $\square$

The efficiency of the heap-store operations can be summarized as follows:

**Theorem 1.** *All heap-store operations inject, eject, and replace take $O(1)$ worst-case time on a pointer machine.*

**Proof.** Simple pointer manipulations are necessary to accomplish the heap-store operations, and all these take $O(1)$ worst-case time. In connection with each *inject*, at most one element comparison is performed which is assumed to be a constant time operation. *eject* and *replace* do not require any element comparisons. $\square$

*Node store*

The primary purpose of a node store is to keep track of potential violation nodes, and its secondary purpose is to store the heights of the nodes. When handling potential violation nodes we follow quite closely the guidelines given in [7], but we avoid the usage of a resizable array and only rely on list structures. Because of this modification all relaxed-weak-queue operations work on a pointer machine; the capabilities of a random-access machine are not needed to match the performance of a run-relaxed heap.

To define the concept of a potential violation node in a relaxed weak heap, we need some definitions. Let $p$ be a node in a binary tree, and let $q$ be the left child of $p$ and $r$ the right child of $p$. In such a situation, we say that $p$ is the *surrogate parent* of $q$ and the *real parent* of $r$. For a node $s$, we call every ancestor of $s$ that is a real parent of another ancestor of $s$ a *real ancestor* of $s$. Recall that every node is an ancestor of itself. A *weak-heap-order violation* occurs if the element stored at a node is smaller than the element stored at the *first* real ancestor of that node. In a *potential violation node* a weak-heap-order violation may occur.

Clearly, a root cannot be a potential violation node since it has no real ancestors. A node accessed by *decrease* is made a potential violation node immediately after the element replacement. This is necessary since it is not possible to check in $O(1)$ time whether there is a weak-heap-order violation or not. Even if there are no weak-heap-order violations or even if a later priority-queue operation may remove all existing violations, a potential violation node retains its status until the potential weak-heap-order violation between the node and its first real ancestor is explicitly removed.

A node that is a potential violation node is said to be *marked*. Furthermore, a marked node is said to be *tough* if it is the left child of its parent and also the parent is marked. A chain of consecutive tough nodes followed

by a single nontough marked node is called a *run*. All tough nodes of a run are called its *members*; the single nontough marked node of that run is called its *leader*. A marked node that is neither a member nor a leader of a run is called a *singleton*. To summarize, we have divided the set of all nodes into four disjoint categories: unmarked nodes, run members, run leaders, and singletons.

For a relaxed weak queue of size $n$, the backbone of its node store is a $2 \times \lfloor \lg n \rfloor$ matrix $(a_{i,j})$, where $i \in \{0, 1\}$ and $j \in \{0, 1, \ldots, \lfloor \lg n \rfloor - 1\}$. Each *state object* $a_{i,j}$ is linked to its neighbours $a_{i-1,j}$, $a_{i+1,j}$, $a_{i,j-1}$, and $a_{i,j+1}$, if those exist. The row $A_0 = \left\langle a_{0,0}, \ldots, a_{0,\lfloor \lg n \rfloor - 1} \right\rangle$ of this matrix stores information related to unmarked nodes, but it is also relevant for run leaders and singletons. The row $A_1 = \left\langle a_{1,0}, \ldots, a_{1,\lfloor \lg n \rfloor - 1} \right\rangle$ only stores information related to run members. A new column is allocated to this matrix when $n$ is increased and becomes equal to a power of two. Analogously, the last column is deallocated when $n$ is decreased and becomes equal to a power of two minus one. We assume that this dynamization of the matrix is done automatically in connection with the increments and decrements of $n$.

The state objects are identical in one respect: each object $a_{i,j}$ stores the integers $i$ and $j$. All unmarked nodes share the state objects on row $A_0$ such that each unmarked node of height $h$ has a pointer to state object $a_{0,h}$. By consulting this object the node gets its height and its type. All run members use the state objects on row $A_1$ in a similar manner and no other information is associated with these objects.

For each run leader, we store an additional *leader object*. Each leader object stores the height $h$ of the node, the type of that node (integer 2), a pointer to that node, and another pointer to state object $a_{0,h}$. Correspondingly, each leader has a pointer back to its leader object. All leader objects are kept in a doubly-linked list, called the *leader-object list*, so two additional pointers are stored at each object to record the successor and the predecessor of that object in the leader-object list.

Similar to run leaders, we store an additional *singleton object* for each singleton. Every singleton object is linked to the corresponding singleton and vice versa. In addition to this pointer, each singleton object stores the height $h$ of that node, the type of that node (integer 3), and a pointer to state object $a_{0,h}$. All singleton objects for singletons of the same height are kept in a doubly-linked list, called the *singleton-object list*, so two additional pointers need to be stored at each singleton object. A pointer to the beginning of the singleton-object list for height $h$ is stored at $a_{0,h}$.

Each state object $a_{0,h}$ stores a pointer to a *pair object* which indicates whether the singleton-object list associated with $a_{0,h}$ contains more than one object. Each pair object has a pointer back to the state object. If the singleton-object list contains less than two objects, the pointer in $a_{0,h}$ reserved for a pair object has the value null. Finally, all pair objects are kept in a doubly-linked list, called the *pair-object list*, meaning that every pair object should store two additional pointers.

Let $\mathcal{N}$ denote the type of a node in a relaxed weak queue. The abstract interface to a node store must provide the following public methods, in addition to trivial constructors and destructors:

*void mark*($\mathcal{N}$ *q*)**.** Mark node $q$; if $q$ is already marked or a root, do nothing.
   **Requirement.** $q$ is a valid node. $q$ is passed by reference.

*void unmark*($\mathcal{N}$ *q*)**.** Unmark node $q$; if $q$ is already unmarked, do nothing.
   **Requirement.** $q$ is a valid node. $q$ is passed by reference.

*void reduce*()**.** Unmark at least one arbitrary marked node, if possible.

Initially, when a node is created, it is made unmarked (by letting its node-store pointer point to $a_{0,0}$).

We let tuple $(t, h)$, where $t \in \{\text{unmarked}, \text{member}, \text{leader}, \text{singleton}\}$ and $h \in \{0, 1, \ldots, \lfloor \lg n \rfloor - 1\}$, denote the *current state* of a node. In addition to the public methods, the node store provides private methods that make the implementation of the public methods easier. We describe these private methods as *transitions* between states. The requirement for each of the transitions is that the given node $q$ is valid and of the required type. The actions taken in each of the transitions are as follows.

$q$: (unmarked, $h$) $\frown$ (unmarked, $h + 1$)**.** The node-store pointer of $q$ is updated to point to $a_{0,h+1}$ instead of $a_{0,h}$.

$q$: (unmarked, $h$) $\frown$ (unmarked, $h - 1$)**.** The node-store pointer of $q$ is updated to point to $a_{0,h-1}$ instead of $a_{0,h}$, assuming that $h > 0$.

$q$: (unmarked, $h$) $\frown$ (member, $h$)**.** The node-store pointer of $q$ is updated to point to $a_{1,h}$ instead of $a_{0,h}$.

$q$: (member, $h$) $\frown$ (unmarked, $h$)**.** The node-store pointer of $q$ is updated to point to $a_{0,h}$ instead of $a_{1,h}$.

$q$: (unmarked, $h$) $\frown$ (leader, $h$)**.** A new leader object is created, the height stored at $a_{0,h}$ and the address of $a_{0,h}$ are copied to the created object, the type is assigned to be a leader, and the created object is added to the leader-object list. The node-store pointer of $q$ is updated to point to this newly created object instead of $a_{0,h}$.

$q$: (leader, $h$) $\frown$ (unmarked, $h$)**.** The node-store pointer of $q$ is updated to point to $a_{0,h}$ using the pointer available at the old leader object. The leader object is removed from the leader-object list and the leader object is freed.

$q$: (unmarked, $h$) $\frown$ (singleton, $h$)**.** A new singleton object is created, the height stored at $a_{0,h}$ as well as the address of $a_{0,h}$ are copied to the created object, the type is assigned to be a singleton, and the created object is appended to the singleton-object list residing at $a_{0,h}$. If the length of that list becomes two, a new pair object is created, the created object is appended to the pair-object list, and the pair-object pointer at $a_{0,h}$ is updated accordingly. Finally, the node-store pointer of $q$ is updated to point to the newly created singleton object.

$q$: (singleton, $h$) $\frown$ (unmarked, $h$)**.** The node-store pointer of $q$ is updated to point to $a_{0,h}$ instead of the old singleton object using the pointer available at the singleton object. The singleton object is removed from

the singleton-object list, and thereafter the object is freed. If after the removal of the object the length of the singleton-object list is equal to one, the corresponding pair object is removed from the pair-object list and the pair-object pointer at $a_{0,h}$ is updated accordingly.

Using these transition methods, it is straightforward to implement the data-structural transformations performed by the public methods. A simple example of such a transformation is a join, where some pointer modifications must be done and the height of the new root must be increased by one. In all such local transformations, our implementation strategy is to unmark all modified nodes, do all necessary pointer modifications, update the height of the nodes whose height has changed, and mark all nodes that should be marked. For this strategy the set of transition methods described will be sufficient. The increments and decrements of the height of a node can be implemented using the first two transition methods. Let us next consider how markings and unmarkings are done using the transition methods.

If a node to be marked is already marked or a root, *mark* does nothing. To mark an unmarked node $q$ of height $h$, there are six cases depending on the state of the parent of $q$, call it $p$, and the state of the left child of $q$, call it $r$. The transitions to be performed can conveniently be described in a tabular form. In the following table, a bullet • is used to denote the current state of the node in question, because the current state can be deduced from the context.

| state of $r$ / state of $p$ | does not exist or (unmarked, $h-1$) | (leader, $h-1$) or (singleton, $h-1$) |
|---|---|---|
| real parent or surrogate parent (unmarked, $h+1$) | $q{:}\bullet\curvearrowright(\text{singleton}, h)$ | $q{:}\bullet\curvearrowright(\text{leader}, h)$ <br> $r{:}\bullet\curvearrowright(\text{unmarked}, h-1)$ <br> $r{:}\bullet\curvearrowright(\text{member}, h-1)$ |
| surrogate parent (member, $h+1$) | $q{:}\bullet\curvearrowright(\text{member}, h)$ | $q{:}\bullet\curvearrowright(\text{member}, h)$ <br> $r{:}\bullet\curvearrowright(\text{unmarked}, h-1)$ <br> $r{:}\bullet\curvearrowright(\text{member}, h-1)$ |
| surrogate parent (singleton, $h+1$) | $p{:}\bullet\curvearrowright(\text{unmarked}, h+1)$ <br> $p{:}\bullet\curvearrowright(\text{leader}, h+1)$ <br> $q{:}\bullet\curvearrowright(\text{member}, h)$ | $p{:}\bullet\curvearrowright(\text{unmarked}, h+1)$ <br> $p{:}\bullet\curvearrowright(\text{leader}, h+1)$ <br> $q{:}\bullet\curvearrowright(\text{member}, h)$ <br> $r{:}\bullet\curvearrowright(\text{unmarked}, h-1)$ <br> $r{:}\bullet\curvearrowright(\text{member}, h-1)$ |

The transitions to be done in *unmark* can also be described in a tabular form. If the given node is already unmarked, *unmark* does nothing. Therefore, assume that the given node $q$ is marked and has height $h$. Let $p$ be its parent, $r$ its left child, and $s$ the left child of $r$. There are three cases depending on the state of $q$.

**Case 1.** The state of $q$ is (member, $h$). If $r$ does not exist or if the state of $r$ is (unmarked, $h-1$), the following transitions are done.

| **state of $p$** | |
| --- | --- |
| $(\text{member}, h+1)$ | $q\colon \bullet \curvearrowright (\text{unmarked}, h)$ |
| $(\text{leader}, h+1)$ | $p\colon \bullet \curvearrowright (\text{unmarked}, h+1)$<br>$p\colon \bullet \curvearrowright (\text{singleton}, h+1)$<br>$q\colon \bullet \curvearrowright (\text{unmarked}, h)$ |

The other possibility is that the state of $r$ is $(\text{member}, h-1)$. In that case, the transitions to be done depend on the state of $p$ and that of $s$ as follows.

| **state of $s$**<br>**state of $p$** | does not exist<br>or<br>$(\text{unmarked}, h-2)$ | $(\text{member}, h-2)$ |
| --- | --- | --- |
| $(\text{member}, h+1)$ | $q\colon \bullet \curvearrowright (\text{unmarked}, h)$<br>$r\colon \bullet \curvearrowright (\text{unmarked}, h-1)$<br>$r\colon \bullet \curvearrowright (\text{singleton}, h-1)$ | $q\colon \bullet \curvearrowright (\text{unmarked}, h)$<br>$r\colon \bullet \curvearrowright (\text{unmarked}, h-1)$<br>$r\colon \bullet \curvearrowright (\text{leader}, h-1)$ |
| $(\text{leader}, h+1)$ | $p\colon \bullet \curvearrowright (\text{unmarked}, h+1)$<br>$p\colon \bullet \curvearrowright (\text{singleton}, h+1)$<br>$q\colon \bullet \curvearrowright (\text{unmarked}, h)$<br>$r\colon \bullet \curvearrowright (\text{unmarked}, h-1)$<br>$r\colon \bullet \curvearrowright (\text{singleton}, h-1)$ | $p\colon \bullet \curvearrowright (\text{unmarked}, h+1)$<br>$p\colon \bullet \curvearrowright (\text{singleton}, h+1)$<br>$q\colon \bullet \curvearrowright (\text{unmarked}, h)$<br>$r\colon \bullet \curvearrowright (\text{unmarked}, h-1)$<br>$r\colon \bullet \curvearrowright (\text{leader}, h-1)$ |

**Case 2.** The state of $q$ is $(\text{leader}, h)$. Now the following transitions are done depending on the state of $s$.

| **state of $s$** | does not exist<br>or<br>$(\text{unmarked}, h-2)$ | $(\text{member}, h-2)$ |
| --- | --- | --- |
| | $q\colon \bullet \curvearrowright (\text{unmarked}, h)$<br>$r\colon \bullet \curvearrowright (\text{unmarked}, h-1)$<br>$r\colon \bullet \curvearrowright (\text{singleton}, h-1)$ | $q\colon \bullet \curvearrowright (\text{unmarked}, h)$<br>$r\colon \bullet \curvearrowright (\text{unmarked}, h-1)$<br>$r\colon \bullet \curvearrowright (\text{leader}, h-1)$ |

**Case 3.** The state of $q$ is $(\text{singleton}, h)$. In this case only the transition $q\colon (\text{singleton}, h) \curvearrowright (\text{unmarked}, h)$ needs to be done.

This completes the description of *mark* and *unmark*. Next we consider how *reduce* can be implemented.

The rationale behind *reduce* is that, when there are more than $\lfloor \lg n \rfloor$ marked nodes, there must be a run of two or more marked nodes, or at least one pair of marked nodes having the same height. In that case, it is possible to apply at least one of the two transformations — run transformation or singleton transformation — to reduce the number of marked nodes by at least one. The targets for these transformations can be found from the leader-object list or from the pair-object list. These transformations mainly involve pointer modifications; the routines *mark* and *unmark* can be used to do all necessary markings and unmarkings. The first two transition methods

can be used to do all necessary changes in the heights since in both transformations the height of a node may only change by one or two. Later on, one application of the transformations together with all necessary changes to the node store is referred to as a $\lambda$-*reduction*.

**Lemma 3.** *Let $\lambda$ denote the total number of potential violation nodes in a relaxed weak heap storing $n$ elements. Due to $\lambda$-reductions, it must hold that $\lambda \leq \lfloor \lg n \rfloor + 1$ at all times.*

**Proof.** Since a root cannot be violating, a node having the maximum height can never be a potential violation node. That is, the height of a potential violation node must be between 0 and $\lfloor \lg n \rfloor - 1$. After increasing $\lambda$ by one, if $\lambda = \lfloor \lg n \rfloor + 1$, there must be a run of two or more consecutive potential violation nodes, or a pair of two potential violation nodes of the same height. Therefore, premises for a $\lambda$-reduction are fulfilled and it can be used to reduce $\lambda$ by at least one. $\square$

Both run and singleton transformations employ the following set of primitive transformations (see Figure 3) which we have given the same names as the corresponding transformations in a rank-relaxed heap [7]:

**Cleaning transformation.** Assume that a marked node $q$ is the left child of its parent $p$, $s$ is the sibling of $q$, and both $p$ and $s$ are unmarked. In this transformation the subheap $(q, B)$ rooted at $q$ (i.e. $q$ and its right subtree $B$) and the subheap $(s, D)$ rooted at its sibling $s$ are swapped. The purpose of this transformation is to make the parent of $q$ the first real ancestor of $q$. To show the correctness of this transformation, let $f$ denote the first real ancestor of $p$. It is known that $element[f] \not\succ element[p]$ and $element[p] \not\succ element[s]$ so by transitivity $element[f] \not\succ element[s]$. There may be a weak-heap-order violation between $f$ and $q$ as well as between $p$ and $q$, so $q$ should remain marked. Therefore, the swap of the two subheaps is legal and does not introduce any new weak-heap-order violations.

**Parent transformation.** Assume that a marked node $q$ is the right child of its parent $p$, and that $p$ is marked or unmarked. In this transformation the subheap rooted at $p$ is split, the two subheaps are joined, and the result of the join is put in the place of the subheap originally rooted at $p$. If $p$ was a root and $q$ becomes a root, $q$ is unmarked. This transformation moves a marked node one level up and eliminates one marked node if $p$ is marked or a root. The correctness of this transformation directly follows from the correctness of a join and a split.

**Sibling transformation.** Assume that two marked nodes $q$ and $s$ are siblings, and that their parent $p$ is unmarked. Furthermore, assume that $q$ is the left child of $p$ and that $s$ is the right child of $p$. This transformation involves two steps. First, the subheap rooted at $p$ is split, the resulting subheap rooted at $p$ and the subheap rooted at $q$ are swapped, the subheaps rooted at $q$ and $s$ are virtually joined by moving $q$ above $s$ without comparing the elements stored at $q$ and $s$. Second, the parent transformation is applied to $s$. As the outcome of the first step the
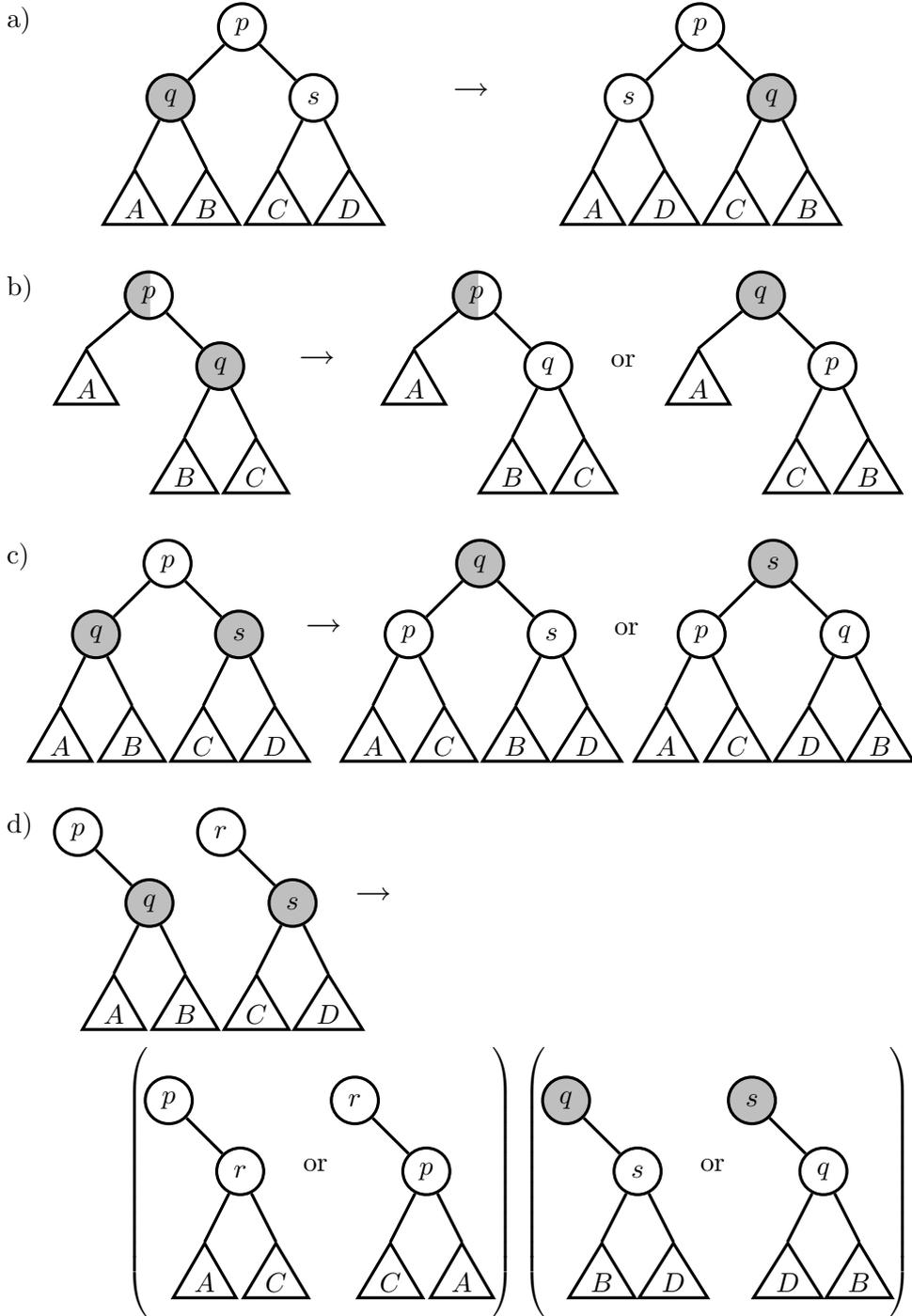
**Figure 3.** Primitives used in a λ-reduction: a) cleaning transformation, b) parent transformation, c) sibling transformation, and d) pair transformation. Marked nodes are drawn in gray. A node that is half gray and half white may be marked or unmarked.

height of one marked node is increased by one, and as the outcome of the second step one marking will disappear. The correctness of the first step immediately follows from the fact that for all nodes, except $s$, their first real ancestors remain the same, so no new weak-heap-order violations are introduced. For $s$, the new real parent does not make any harm since $s$ is already marked. The correctness of the second step follows from that of the parent transformation.

**Pair transformation.** Assume that two marked nodes $q$ and $s$ do not have the same parent and that they are of the same height. Furthermore, assume that $q$ and $s$ are the right children of their respective parents $p$ and $r$, which both are unmarked. This transformation involves three steps. First, the subheaps rooted at $p$ and $r$ are split. Second, the produced subheaps rooted at $p$ and $r$ are joined and the resulting subheap is put in the place of the subheap originally rooted at $p$ or $r$, depending on which becomes the root of the resulting subheap. Third, the two remaining subheaps rooted at $q$ and $s$ are joined and the resulting subheap is put in the place of the subheap originally rooted at $p$ or $r$, depending on which is still unoccupied after the second step. If after the third step $q$ or $s$ becomes a root, the node is unmarked. By this transformation at least one marked node is eliminated. The correctness of this transformation follows from the correctness of splits and joins.

When nodes are marked, the basic idea is to let the markings bubble upwards until two marked nodes meet or we have two marked nodes of the same height. The runs cause a complication in this process since these can only be unravelled from above.

The purpose of a *run transformation* is to move the two top-most marked nodes of a run upwards and at the same time remove at least one marking. Assume now that $q$ is the leader of a run taken from the leader-object list and that $r$ is the first member of that run. There are two cases depending on the position of $q$.

**Case 1.** $q$ is a right child. Apply the parent transformation to $q$. If the number of marked nodes decreased, stop. Now the parent of $r$ is unmarked. If the sibling of $r$ is marked, apply the sibling transformation to $r$ and its sibling, and stop. Thereafter, apply the parent transformation once or twice to $r$ to reduce the number of marked nodes. (Note that the first parent transformation may unmark $r$, so it is legitimate to stop.)

**Case 2.** $q$ is a left child. If the sibling of $q$ is marked, apply the sibling transformation to $q$ and its sibling, and stop. Otherwise, apply the cleaning transformation to $q$, thereby making it a right child. Now the parent of $r$ is unmarked. If the sibling of $r$ is marked, apply the sibling transformation to $r$ and its sibling, and stop. Otherwise, apply the cleaning transformation followed by the parent transformation to $r$. Now $q$ and $r$ are marked siblings with an unmarked parent; apply the sibling transformation to them to reduce the number of marked nodes.

If the number of marked nodes is larger than $\lfloor \lg n \rfloor$ and there are no runs, at least two marked nodes must have the same height, and their parents can only be marked if they are real parents. A pair of such nodes is found via the pair-object list. Assume that $q$ and $s$ are two such singletons. In a *singleton transformation* these nodes are processed as follows. If the sibling of $q$ is marked (which can be $s$), apply the sibling transformation to $q$ and its sibling, and stop. Similarly, if the sibling of $s$ is marked, apply the sibling transformation to $s$ and its sibling, and stop. Now if one or both of $q$ and $s$ are left children, neither their parents nor their siblings can be marked, so apply the cleaning transformation to them, if necessary, thereby making both the right children of their respective parents. In the case that the parent of $q$ is marked, apply the parent transformation to $q$ to remove one marking, and stop. If the parent of $s$ is marked, process $s$ as $q$ above and stop. After ensuring that both parents are unmarked, apply the pair transformation to $q$ and $s$ to remove at least one marking.

The performance of the node-store operations is summed up in the following theorem.

**Theorem 2.** *The node-store operations mark, unmark, and reduce take $O(1)$ worst-case time on a pointer machine.*

**Proof.** All transitions between states involve simple pointer manipulations, object allocations, and object deallocations. Both *mark* and *unmark* perform a constant number of invocations of the transition methods. Therefore, both of them take $O(1)$ worst-case time.

In a run transformation and a singleton transformation only a constant number of pointer modifications, markings, unmarkings, and height updates are done, so both transformations can be accomplished in $O(1)$ worst-case time. That is, the worst-case running time of *reduce* is also $O(1)$. In a run transformation at most three and in a singleton transformation at most two element comparisons are made. Other node-store operations do not involve any element comparisons. $\square$

## 3. Realizations of priority-queue operations

Often in the literature the best worst-case implementations of binomial-queue operations are not described. For example, in [20] and [6, Chapter 19] the given implementation of *find-min* requires $O(\lg n)$ worst-case time and that of *insert* $O(\lg n)$ worst-case time; and in [7] the implementation of *find-min* requires $O(\lg n)$ worst-case time. In this section we give a full description of all relaxed-weak-queue operations. In the comparison-based model, the implementations given are asymptotically optimal.

A relaxed weak queue can be represented by maintaining the following information:

1. A pointer to a heap store is maintained. The heap store pointed to is given full responsibility for keeping track of the perfect weak heaps currently stored.

2. A pointer to a node store is maintained. The node store is given full responsibility for keeping track of all potential violation nodes currently in the data structure.

3. A variable $n$ is used to recall the number of elements stored in the data structure. Another variable is used to recall $\lfloor \lg n \rfloor$. To avoid the usage of a whole-number logarithm function, $\lfloor \lg n \rfloor$ can be calculated by maintaining the interval $[2^k \mathinner{.\,.} 2^{k+1})$ in which $n$ lies. When $n$ moves outside the interval, the logarithm and the interval are updated accordingly. Each time the value of $\lfloor \lg n \rfloor$ is increased or decreased, a column is allocated to or deallocated from the backbone of the node store, respectively.

4. To facilitate a fast *find-min*, a pointer to the node storing the current minimum is maintained and continuously updated by all modifying operations. This *minimum pointer* refers to a root or to a potential violation node.

Now we are ready to prove the main result of this paper:

**Theorem 3.** *A relaxed weak queue supports find-min, insert, and decrease in $O(1)$ worst-case time, and delete in $O(\lg n)$ worst-case time, where $n$ denotes the number of elements stored in the data structure prior to the operation. These time bounds are valid on a pointer machine as well as on a random-access machine.*

**Proof.** We give the proof by considering all the priority-queue operations separatively.

*find-min*

The minimum pointer points to the node storing the current minimum, so this node can just be returned. That is, this operation can be accomplished in $O(1)$ worst-case time.

*insert*

To insert an element into the data structure, a new node is allocated, the given element is placed into this node, and the new node which is also a perfect weak heap of height 0 is placed into the heap store by invoking *inject*. If the given element is smaller that the element stored at the node pointed to by the minimum pointer, the minimum pointer is updated to point to the newly created node.

According to our assumptions a new node can be allocated and an element can be moved in $O(1)$ worst-case time. Also, *inject* takes $O(1)$ worst-case time. Similarly, an element comparison and a pointer update take $O(1)$ worst-case time. That is, *insert* requires $O(1)$ worst-case time. In *inject* one element comparison may be necessary, so in total at most two element comparisons are made.

*delete*

There are several alternative ways of implementing *delete*; the implementation to be described relies on the borrowing technique used in [7, 10], but other approaches could have been chosen (see, e.g. [6, Section 19.2] or [20]). The borrowing technique has two advantages compared to the other approaches considered. First, the structure of the heap store can be kept simple, i.e. a stack-like structure as described will be powerful enough to support *delete*. Second, the constant factor in the leading term of the time bound can be made lower than that achievable by other approaches (cf. [10]).

Let us assume that node $p$ is to be deleted. To delete $p$, the basic idea is to extract the subheap rooted at $p$ from the perfect weak heap, in which it resides, borrow another node $q$ from the smallest perfect weak heap to fill in the hole created by $p$, and put the new subheap in the place of the extracted subheap. More precisely, the deletion of $p$ is done as follows:

**Step 1.** Eject the smallest perfect weak heap from the heap store by invoking *eject*. Let $q$ be the root of that perfect weak heap.

**Step 2.** Repeat the following until $q$ has no children.
  **a)** Split the perfect weak heap rooted at $q$. Let $r$ be the root of the other subheap created.
  **b)** Remove the marking of $r$, if any, by invoking *unmark*, since a root may not be a potential violation node.
  **c)** Insert the subheap rooted at $r$ into the heap store by invoking *inject*.

**Step 3.** If $p$ and $q$ are the same node, go to Step 11.

**Step 4.** Extract the subheap rooted at $p$ from the perfect weak heap, in which it resides, and remember the neighbouring nodes of $p$.

**Step 5.** Repeat the following until $p$ has no children.
  **a)** Split the subheap rooted at $p$. Let $s$ be the root of the other subheap created.
  **b)** Push the subheap rooted at $s$ onto a temporary stack.

**Step 6.** Repeat the following until the temporary stack is empty.
  **a)** Pop the top of the temporary stack. Let $s$ be the root of the subheap popped.
  **b)** Remove the marking of $s$, if any, by invoking *unmark*, since the subtree rooted at $s$ will be orderly joined.
  **c)** Join the subheaps rooted at $q$ and $s$; independent of the outcome denote the new root $q$.

**Step 7.** Put $q$ in the place originally occupied by $p$.

**Step 8.** Make $q$ a potential violation node by invoking *mark*.

**Step 9.** If $p$ was a root, substitute the perfect weak heap rooted at $q$ for that rooted at $p$ in the heap store by invoking *replace*.

**Step 10.** Remove the marking of $p$, if any, by invoking *unmark* to update the node store.

**Step 11.** If the minimum pointer points to $p$, scan all roots and all potential violation nodes to find a new minimum element and update the minimum pointer.

**Step 12.** Reduce the number of potential violation nodes, if possible, by invoking *reduce* twice (once because of the new potential violation node introduced and once more because of the decrement of $n$).

**Step 13.** Free $p$ and return.

According to our earlier analysis, Step 1, Step 8, Step 9, Step 10, and Step 12 take $O(1)$ worst-case time. In addition, according to our assumptions, Step 3, Step 4, Step 7, and Step 13 all take $O(1)$ worst-case time. The height of all perfect weak heaps is bounded by $\lfloor \lg n \rfloor$. Therefore, in Step 2, Step 5, and Step 6 the number of repetitions is bounded by $\lfloor \lg n \rfloor$. In these steps, all the substeps only take $O(1)$ worst-case time so the total running time of these steps is bounded by $O(\lg n)$.

By Lemma 2 the number of perfect weak heaps is at most $\lfloor \lg n \rfloor + 1$ and by Lemma 3 the number of potential violation nodes is also at most $\lfloor \lg n \rfloor + 1$. The heap store can easily be extended to provide iterators so that the roots of the perfect weak heaps can be visited. Pointers to the roots are available in the heap sequence, which can be traversed incrementally. Also, the node store can be extended to provide iterators in order to iterate over all potential violation nodes. All singletons are found by scanning through all singleton-object lists (even the empty ones). All run leaders are readily available in the leader-object list, and the members of every run can be found by starting from the run leader and repeatedly accessing the left children until a nonmember is met. Therefore, Step 11 takes at most $O(\lg n)$ time.

To sum up, *delete* requires $O(\lg n)$ worst-case time. Only Step 2, Step 6, Step 11, and Step 12 involve element comparisons. Actually, Step 2 can be improved by ignoring the joins in Step 2c; even without them the representation of the number corresponding to the heap sequence will remain regular. After this optimization, no element comparisons are necessary in Step 2. In Step 6 each join requires one element comparison and the loop is repeated at most $\lfloor \lg n \rfloor$ times, so at most $\lfloor \lg n \rfloor$ element comparisons are performed. In Step 11 at most $\lfloor \lg n \rfloor + 1$ roots and at most $\lfloor \lg n \rfloor + 1$ potential violation nodes are visited, so at most $2 \lg n + O(1)$ element comparisons are performed. According to our earlier analysis in a $\lambda$-reduction at most three element comparisons are performed, so in Step 12 at most six element comparisons are performed. To summarize, the number of element comparisons performed is bounded above by $3 \lg n + O(1)$.

*decrease*

Let $p$ be the node given as an argument for *decrease*. To accomplish the operation, the old element stored at $p$ is replaced by the given element, $p$ is made a potential violation node by invoking *mark*, and the number of potential violation nodes is reduced, if possible, by invoking *reduce*. As for the correctness of *decrease*, it is important to note that the element replacement does not cause any new weak-heap-order violations below the given node and that the potential weak-heap-order violations above the given node are handled correctly by $\lambda$-reductions.

By our assumption an element move takes $O(1)$ time, and by our earlier analysis both *mark* and *reduce* can be carried out in $O(1)$ worst-case time. That is, *decrease* also takes $O(1)$ worst-case time. Only *reduce* involves element comparisons; and by our earlier analysis at most three element comparisons are performed in a $\lambda$-reduction. □

## 4. Concluding remarks

We find the connection between perfect weak heaps and heap-ordered binomial trees interesting. Using this connection (as done earlier, for example, in [4, 12]) it is possible to implement a worst-case efficient priority queue using only binary trees. The main contribution in this paper was to take the relaxation technique used in a run-relaxed heap [7] into the binary-tree setting. A run-relaxed heap and a relaxed weak queue provide the same worst-case performance for all priority-queue operations, except that a relaxed weak queue is simpler to program and works on a pointer machine.

By using the child-sibling representation of binary trees and by reusing the unused sibling pointers as parent pointers (see, e.g. [19, Section 4.1]), a perfect weak heap can be represented using two pointers per node. This representation makes an access to the parent slower, so in practice it may be better to retain the parent pointers. The auxiliary data structures maintained to keep track of perfect weak heaps and potential violation nodes only require $O(\lg n)$ words of storage. Therefore, taking into account the extra pointer stored at each node, the total space usage of a relaxed weak queue can be as low as $3n + O(\lg n)$ words plus the space needed to store the elements themselves.

Earlier realizations of a worst-case efficient priority queue have higher space requirements. In its original form [7] a run-relaxed heap uses $6n + O(\lg n)$ words plus $n$ bits of storage. Each node stores four pointers to record parent-child-sibling relationships, a rank, a pointer to the violation structure, and a bit indicating whether the node is a potential violation node or not. A fat heap [13, 14] uses $4n + O(\lg n)$ words of storage, since each node must store three pointers and a rank. In earlier binary-tree realizations (see, e.g. [12, 18]), the $O(1)$ time bound for *decrease* is only amortized.

We would like to point out that relaxed weak queues could be used in the two-tier framework described in [10], thereby reducing the number of element comparisons performed by *delete*. By substituting relaxed weak queues for run-relaxed heaps, a realization of a priority queue is obtained which reduces the number of element comparisons performed by *delete* from $3 \lg n + O(1)$ to $\lg n + O(\lg \lg n)$. For a fat heap the corresponding bound with the two-tier framework is $2 \log_3 n + O(\lg \lg n)$ ($2 \log_3 n \approx 1.27 \lg n$). The asymptotic running times of all priority-queue operations are retained by this data-structural transformation.

There are two natural variations of a relaxed weak queue that may have practical relevance. First, if it is enough to support *decrease* in logarithmic

worst-case time, it can be accomplished by invoking *delete* followed by *insert*. The whole node store could be eliminated and the extra pointers at the nodes could be removed. It would be enough to store the height of each weak heap. In *delete* the root should be accessed first and the path from the root to the given node should also take part in the split and join processes (cf. [20]). Such a weak queue would only use $2n + O(\lg n)$ words of storage, in addition to the space used by the $n$ elements; and the number of element comparisons performed by *delete* would reduce to $2\lg n + O(1)$. Second, it would be possible to support *meld* in $O(\min\{\lg m, \lg n\})$ worst-case time, where $m$ and $n$ denote the size of the subcollections to be melded. To obtain this, the node store could be implemented as in a run-relaxed heap. Moreover, each node should store its height and a pointer to the node store, which now only contains information on potential violation nodes. That is, the space requirements would increase to $4n + O(\lg n)$ words.

We consciously avoided the use of random access in the manipulation of a relaxed weak queue. However, if *meld* should also be supported and the violation structure used in a run-relaxed heap is employed, the capabilities of a random-access machine are needed. Also, a fat heap requires the capabilities of a random-access machine if *meld* is to be provided in logarithmic worst-case time. Brodal's heap [2] relies on resizable arrays, and thereby on random access as well. The purely functional priority queue of Brodal and Okasaki [3] is pointer-based, but it does not offer any support for general *delete* or *decrease*, even if *meld* can be carried out in $O(1)$ worst-case time.

We hope that in textbooks on data structures and algorithms binomial queues [20] and Fibonacci heaps [11] will be replaced by relaxed weak queues, or some simpler data structure. A binomial queue and a relaxed weak queue provide the same worst-case performance for all priority-queue operations, except that a relaxed weak queue can carry out *decrease* in $O(1)$ worst-case time. Compared to a Fibonacci heap, the time bounds guaranteed by a relaxed weak queue are worst-case rather than amortized, the implementations of the priority-queue operations are straightforward, and the worst-case analysis is easy to understand. Though in applications, where it is essential to support *meld* in $O(1)$ time, one may consider using Fibonacci heaps or Brodal's heaps. However, when supporting both *meld* and *decrease*, it may be difficult to avoid solving a union-find problem outside the data structure, as pointed out in [13], since *decrease* requires that it is known to which subcollection the given node belongs.

## References

[1] M. R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing* **7** (1978), 298–319.

[2] G. S. Brodal. Worst-case efficient priority queues. *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*. ACM/SIAM (1996), 52–58.

[3] G. S. Brodal and C. Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming* **6** (1996), 839–857.

[4] S. Carlsson, J. I. Munro, and P. V. Poblete. An implicit binomial queue with constant

insertion time. *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science* **318**. Springer-Verlag (1988), 1–13.

[5]   M. J. Clancy and D. E. Knuth. A programming and problem-solving seminar. Technical Report STAN-CS-77-606. Department of Computer Science, Stanford University (1977).

[6]   T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, 2nd Edition. The MIT Press (2001).

[7]   J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM* **31** (1988), 1343–1354.

[8]   R. D. Dutton. Weak-heap sort. *BIT* **33** (1993), 372–381.

[9]   S. Edelkamp and I. Wegener. On the performance of Weak-Heapsort. *Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science* **1770**. Springer-Verlag (2000), 254–266.

[10]   A. Elmasry, C. Jensen, and J. Katajainen. A framework for speeding up priority-queue operations. CPH STL Report 2004-3. Department of Computing, University of Copenhagen (2004).

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

[12]   P. Høyer. A general technique for implemention of efficient priority queues. *Proceedings of the 3rd Israel Symposium on the Theory of Computing Systems*. IEEE (1995), 57–66.

[13]   H. Kaplan, N. Shafrir, and R. E. Tarjan. Meldable heaps and boolean union-find. *Proceedings of 34th Annual ACM Symposium on Theory of Computing*. ACM (2002), 573–582.

[14]   H. Kaplan and R. E. Tarjan. New heap data structures. Technical Report TR-597-99. Department of Computer Science, Princeton University (1999).

[15]   D. E. Knuth. *Fundamental Algorithms, The Art of Computer Programming* **1**, 3rd Edition. Addison Wesley Longman (1997).

[16]   D. E. Knuth. *Sorting and Searching, The Art of Computer Programming* **3**, 2nd Edition. Addison Wesley Longman (1998).

[17]   M. Penttonen and J. Katajainen. Notes on the complexity of sorting in abstract machines. *BIT* **25** (1985), 611–622.

[18]   G. L. Peterson. A balanced tree scheme for meldable heaps with updates. Technical Report GIT-ICS-87-23. School of Information and Computer Science, Georgia Institute of Technology (1987).

[19]   R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM (1983).

[20]   J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM* **21** (1978), 309–315.

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