Relaxed Weak Queues and Their Implementation on a Pointer Machine[*]

Amr Elmasry[1,2], Claus Jensen[2], and Jyrki Katajainen[2]
[1]*Max-Planck-Institut für Informatik, Saarbrücken, Germany*
[2]*Department of Computer Science, University of Copenhagen, Denmark*

Abstract

We introduce a relaxed weak queue, a priority queue that is visualized and demonstrated as a collection of binary trees (not multiary trees). It supports all priority-queue operations as efficiently as a run-relaxed heap: *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. A relaxed weak queue uses $3n+O(\lg n)$ pointers besides the space used to store the n elements. All the stated bounds are valid on a pointer machine. Additionally, the operation repertoire can be extended to include *meld* in $O(\min\{\lg m, \lg n\})$ worst-case time, where m and n are the sizes of the priority queues to be melded. However, this increases the worst-case running time of *decrease* to $O(\lg\lg n)$.

1 Introduction

A (minimum) *priority queue* is a data structure used for maintaining a dynamic collection of elements, each drawn from a totally ordered set, such that the minimum element is easily accessible. If Q is a priority queue and elements are stored in nodes, one element per node, the operations to be provided are:

***find-min*(Q).** Return a node in Q that contains a minimum element.

***insert*(Q, p).** Insert node p (which already contains an element) into Q.

***delete-min*(Q).** Delete a node in Q that contains a minimum element.

***delete*(Q, p).** Delete node p from Q.

***decrease*(Q, p, e).** Replace the element at node p with element e, which is not larger than the element earlier stored at p.

[*]CPH STL Report 2010-4. Written in December 2009.

The work of the authors was partially supported by the Danish Natural Science Research Council under contracts 21-02-0501 (project "Practical data structures and algorithms") and 272-05-0272 (project "Generic programming—algorithms and tools"). A. Elmasry was supported by the Alexander von Humboldt Foundation and the VELUX Foundation.

The operation *delete-min*(Q) can be carried out by invoking *find-min*(Q), returning a node p, and thereafter *delete*(Q, p). Alternatively, *delete*(Q, p) can be carried out by emulating *decrease*$(Q, p, -\infty)$ and thereafter *delete-min*(Q). For some applications, it may be necessary to support other operations; the following two are the most relevant:

***extract*(Q).** Extract and return an *unspecified* node from priority queue Q.

***meld*(Q, R).** Move all nodes from priority queues Q and R to a new priority queue, making both empty, and return the new priority queue.

Of these operations, *extract* is non-standard but turned out to be useful for several purposes. First, it can be used when destroying a priority queue; one can just repeatedly invoke *extract* until the data structure becomes empty. Second, it can be used in data-structural transformations for transferring nodes from one priority queue to another while keeping the two priority queues fully operational (see, for example, [11, 12, 13]). Third, it can be used as a subroutine when implementing *delete*, where an extracted node replaces the deleted node to retain structural properties (see, for example, [8, 11, 12]).

To support *decrease* in $O(1)$ worst-case time, the basic idea, applied by Driscoll et al. [8] to a binomial queue [5, 24], 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 its parent. Driscoll et al. described two forms of relaxed heaps: a rank-relaxed heap and a run-relaxed heap. For a rank-relaxed heap, the $O(1)$ time bound for *decrease* is amortized. However, the transformations needed for reducing the number of potential violation nodes are simpler than those required by a run-relaxed heap.

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 [8]: *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 $\log_2(\max\{2, n\})$. We adapt the relaxation technique used for a run-relaxed heap to the binary correspondent of a binomial queue described in [24]; we call the binary structure a weak queue and the relaxed version a relaxed weak queue.

Compared to run-relaxed heaps, our key improvements are threefold. First, we immigrate the transformations into the binary setting, introducing some variations to the transformations in [8]. Moreover, our transformations are decomposed into simple primitives that are conceptually similar to those for a rank-relaxed heap. Second, we show how to implement the priority queue on a pointer machine. Other enhancements include pointing out how to improve the time bound of *find-min* to $O(1)$, and how to efficiently implement *meld*. Third, the space used by a relaxed weak queue can be made less than that required by other structures achieving the same time bounds. (A similar space optimization is known to be possible for binomial queues [5].)

Other data structures having the same asymptotic performance are Brodal's heaps [2], which can in addition meld two priority queues in $O(1)$ worst-case time, and fat heaps [17, 18], which can as well be implemented on a pointer machine. Several other priority-queue structures that are less complicated, but do not achieve the same asymptotic bounds, are known, e.g. binary heaps [25], leftist trees (see [20, Section 5.2.3] or [23, Section 3.3]), binomial queues [5, 24], and ranked priority queues [16].

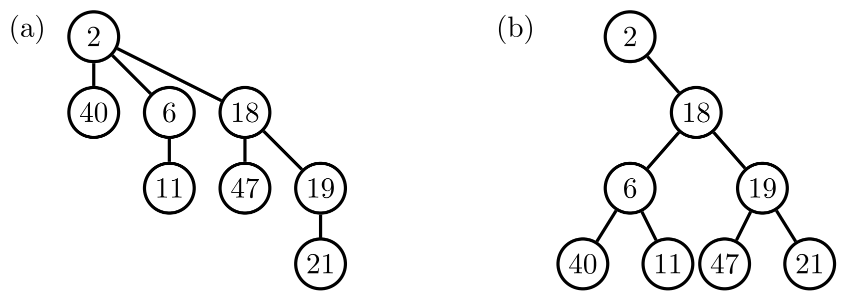

Figure 1: (a) A heap-ordered binomial tree of rank 3 and (b) the corresponding perfect weak heap of height 3.

As our model of computation, we use the high-level version of a *pointer machine* as discussed in [1]. The memory of a pointer machine is a directed graph of cells, each storing a bounded number of pointers to other cells or to the cell itself. All cells are accessed via special centre cells seen as incarnations of registers of ordinary computers. The primitive operations allowed include the creation of cells, destruction of cells, assignment of pointers, and equality comparison of two pointers. By letting some fixed cells represent integers, a pointer stored at a cell can also be interpreted as an integer. Compared to a random-access machine, a pointer machine does not allow general integer arithmetic. In addition, one cannot access cells in a random-access manner. When stating time bounds, we assume that the elements manipulated can be copied and compared in $O(1)$ time, and that all primitive operations require $O(1)$ time. When stating space bounds, we only consider the extra space needed for storing pointers, excluding the space needed for storing the elements.

The structure of the paper is as follows. In Section 2, we recall weak heaps, the basic building blocks that we use in our priority queue construction. We give a brief review of the run-relaxed heaps in Section 3. The transformations used by relaxed-weak-queue operations are described in Section 4. Implementing the priority queue on a pointer machine is discussed in Section 5, including how to incorporate *meld* in logarithmic time. Finally, some conclusions are drawn in Section 6.

2 Weak queues

The building blocks used in relaxed heaps [8] are heap-ordered binomial trees [5, 24]:

1. A *binomial tree* of *rank* 0 is a single node, and a *binomial tree* of *rank* h, for $h > 0$, is composed of a root and its binomial subtrees of rank 0, 1, ..., $h-1$ in this order.

2. The element stored at a node is not smaller than the element stored at its parent.

A heap-ordered binomial tree is a multiary tree, but it can be viewed as a binary tree using the standard child-sibling transformation (see, e.g. [19, Section 2.3.2]). The outcome of this transformation is a *perfect weak heap* storing 2^h elements for an integer $h \geq 0$. The connection between heap-ordered binomial trees and, what we call, perfect

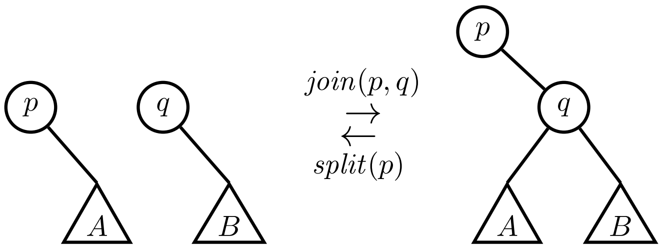

Figure 2: Joining and splitting two perfect weak heaps of the same height (assuming that the element stored at node p is not larger than the element stored at node q).

weak heaps is described in [24]. A perfect weak heap could be defined directly without referring to the corresponding binomial tree as follows (see Figure 1):

1. The root has no left subtree, and its right subtree is a complete binary tree.
2. An element is not larger than the elements in its right subtree.

Since the root has no left subtree, it stores 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 equals the rank of the corresponding subtree in a binomial tree.

Normally, a weak heap is defined in a more general form where the number of elements stored is not necessarily a power of two, and is represented using an array with extra bits [9, 10, 22]. Yet another variant, which is not necessarily perfectly balanced, is called a half-ordered binary tree (see, for example, [14, 22]). A pointer-based representation is preferable when subheaps are frequently relocated, as is the case for priority queues supporting *delete* and *decrease*. 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 [5, 24]. We call such a data structure a *weak queue*.

As for heap-ordered binomial trees, two perfect weak heaps (or subheaps) of the same height can be easily linked together (see Figure 2). 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 the right subtrees of these roots, respectively. Furthermore, let $p.element$ denote the element stored at p. If $p.element \not> q.element$, p is made the root of the resulting perfect weak heap, q the right child of p, A the left subtree of q, and B is kept as the right subtree of q. If $p.element > q.element$, 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. Both a join and a split take $O(1)$ worst-case time. Let p be a node in a binary tree. We say that the *distinguished ancestor* of p is

- the parent of p, if p is a right child; and
- the distinguished ancestor of the parent of p, if p is a left child.

A *weak-heap violation* occurs if the element stored at a node is smaller than the element stored at its distinguished ancestor.

3 Run-relaxed heaps

In this section, we briefly describe the details of run-relaxed heaps. However, we still assume that the reader is familiar with the original paper by Driscoll et al. [8].

A *relaxed binomial tree* [8] is an almost heap-ordered binomial tree where some nodes are *marked*, indicating that the element stored at that node may be smaller than the element stored at its parent. Nodes are marked by *decrease* if the new value is smaller than the value in the parent. By definition, a root is never marked. A marked node is only unmarked when the potential heap-order violation is explicitly removed. A *singleton* is a marked node whose immediate siblings are not marked. A *run* is a maximal sequence of two or more marked nodes that are consecutive siblings. The node of the largest rank in a run is called the *leader* of that run.

A *run-relaxed heap* is a collection of relaxed binomial trees. Let τ denote the number of trees and λ the number of marked nodes in the entire collection. An invariant is maintained that $\tau \leq \lfloor \lg n \rfloor + 1$ and $\lambda \leq \lfloor \lg n \rfloor$ after every heap operation, where n denotes the number of elements stored. To keep track of marked nodes, a *run-singleton structure* is maintained. All singletons are kept in a *singleton table*, which is a resizable array accessed by rank. In particular, this table must be implemented in such a way that growing and shrinking at the tail is possible in $O(1)$ worst-case time, which is achievable by doubling, halving, and incremental copying. Each entry of the singleton table corresponds to a rank; pointers to singletons having this rank are kept in a list. For each entry of the singleton table that has more than one singleton, a counterpart is kept in a *singleton-pair list*. The run leaders are kept in a *run list*. The bookkeeping details of the run-singleton structure are quite straightforward, so we will not repeat them here. The fundamental operations supported are: an addition of a new marked node, a removal of a given marked node, and a removal of an arbitrary marked node if $\lambda > \lfloor \lg n \rfloor$. The worst-case running time of each of these operations is $O(1)$.

Two types of transformations are used when reducing the number of marked nodes: *singleton transformations* are used for combining singletons and *run transformations* are used for making runs shorter. If $\lambda > \lfloor \lg n \rfloor$, the transformations can be applied to reduce λ by at least one. The rationale behind the transformations is that, when there are more than $\lfloor \lg n \rfloor$ marked nodes, there must be at least one pair of singletons that root a subtree of the same rank, or there is a run of neighbouring marked nodes. In both cases, it is possible to apply the transformations. In addition to the pointers required by a binomial queue, each node stores an additional pointer to an object in the run-singleton structure.

To keep track of the roots, a *root table* is maintained, which is also a resizable array accessed by rank. Each entry of the root table corresponds to a rank; pointers to roots having this rank are kept in a list. For each entry of the root table that has more than one root, a counterpart is kept in a *root-pair list*.

A *find-min* operation is implemented by examining all roots and marked nodes. The nodes of a run can be traversed by starting from its leader and following sibling pointers until a non-marked node is reached. Since both the number of trees and the number of marked nodes is $O(\lg n)$, the worst-case running time of *find-min* is $O(\lg n)$. However, if a pointer to the current minimum is maintained by other operations, the worst-case running time of *find-min* becomes $O(1)$.

An *insert* is performed by adding a single-node tree, and appending a pointer pointing to this tree to the list at the corresponding entry in the root table. If this is the second item in this list, a pointer to this root-table entry is appended to the root-pair list. Then the number of trees is reduced, if necessary, by removing a pair of trees corresponding to an entry of the root-pair list, linking the two trees, and adding the resulting tree to the structure. It follows that the worst-case running time of *insert* is $O(1)$.

In *decrease*, after making the element replacement, the new value is compared with the value of the parent. If the node becomes violating, it is marked, an occurrence is inserted into the run-singleton structure, and a reduction is performed on marked nodes if $\lambda > \lfloor \lg n \rfloor$. It follows that the worst-case running time of *decrease* is $O(1)$.

The *delete-min* first invokes *find-min* to locate a node p with the minimum value. The root q of the smallest rank is detached from its tree; this converts the children of q into roots having the smallest ranks in the priority queue. Thereafter, q is merged with the children of p into a new tree with the same rank as p, by repeatedly combining the two nodes of the smallest rank. The new subtree is put in the place originally occupied by p. The root and run-singleton structures are updated accordingly. To maintain the bounds on τ and λ, trees of equal rank are linked until the root-pair list is empty, and a reduction is done on marked nodes if needed. It follows that the worst-case running time of *delete-min* is $O(\lg n)$.

The technique of borrowing the root with the smallest rank can be used to realize *extract*. In the worst case this requires $O(\lg n)$ time. However, if this operation is used for destroying a priority queue of size n, the total amount of work done will be $O(n)$.

4 Transformations

In this section, we assume the same implementation as for run-relaxed heaps, except that we use a weak queue instead of a binomial queue. Considering the binary-tree representation, which should in either case be the way to implement the data structure, a *run* is a maximal sequence of two or more marked nodes that are consecutive on the left spine of a subtree. All nodes of a run have the same distinguished ancestor. More formally, a node is a *member* of a run if it is marked, a left child, and its parent is marked. A node is the *leader* of a run if it is marked, its left child is marked, and it is either a right child or a left child of a non-marked parent. A marked node that is neither a member nor a leader of a run is called a *singleton*.

We modify the original transformations introduced in [8] for a more intuitive set of primitive transformations that constitute the singleton and run transformations. We emphasize that the new transformations are—to an extent—different from those in [8], even when binomial queues are viewed in the binary setting as weak queues.

4.1 Primitive transformations

We have four primitive transformations (see Figure 3). We have kept the names of three of them (cleaning, sibling, pair) the same as the corresponding transformations for rank-relaxed heaps, while stating them in the binary setting. However, our sibling

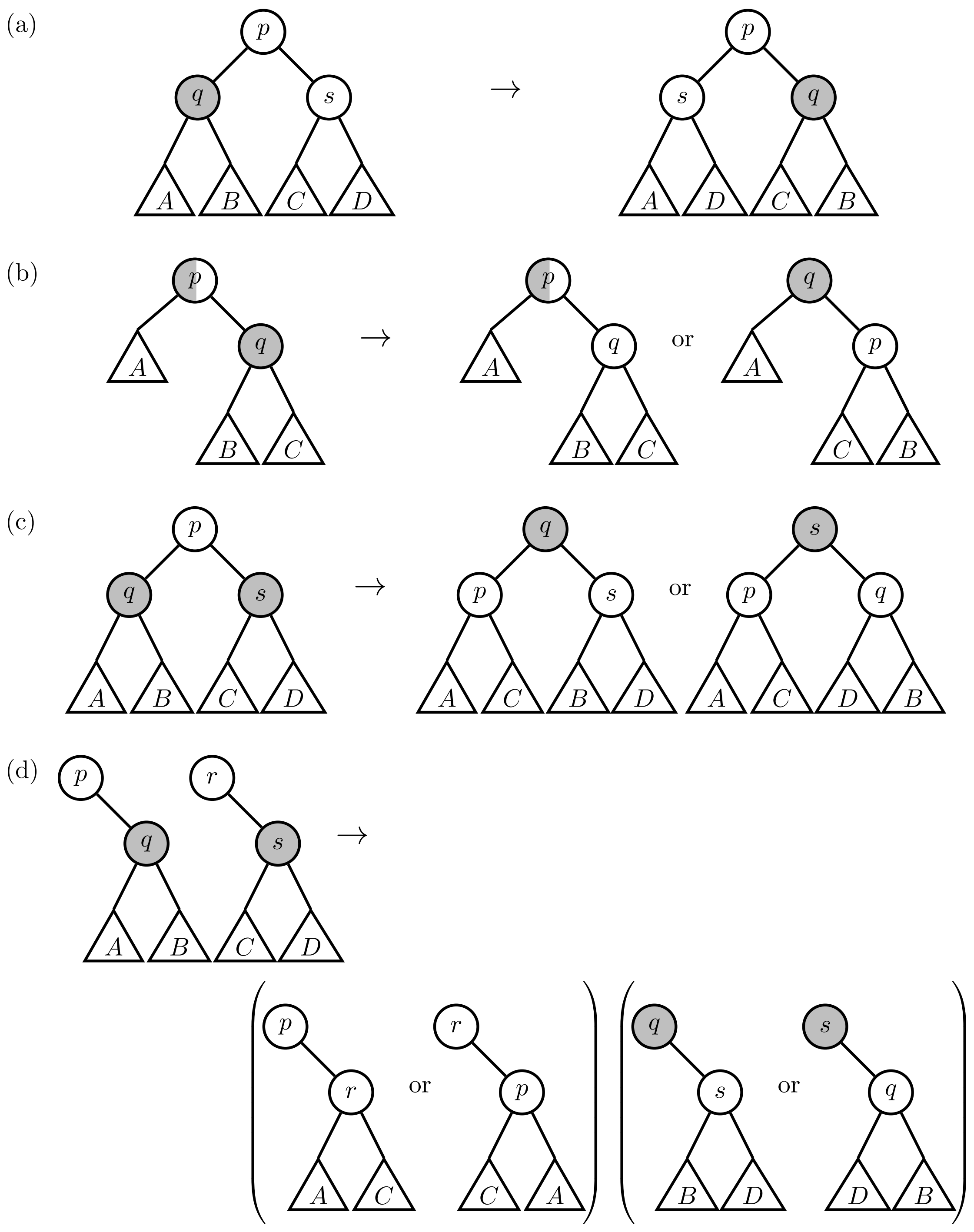

Figure 3: Primitives used in a reduction: (a) cleaning transformation, (b) parent transformation, (c) sibling transformation, and (d) pair transformation. Marked nodes are drawn in grey. A node that is half grey and half white may be marked or unmarked.

transformation is called good-sibling transformation in [8]. In contrary to [8], we are not using their active-sibling transformation as a primitive; instead, we introduce the parent transformation as a new primitive. Note that both the cleaning and parent transformations do not necessarily decrease the number of marked nodes.

Cleaning transformation. Assume that a marked node q is the left child of p, s is the sibling of q, and both p and s are not marked. 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 is to make q a right child. This transformation does not introduce any new weak-heap violations.

Parent transformation. Assume that a marked node q is the right child of p, and that p is marked or non-marked. In this transformation the subheap rooted at p is split, the two subheaps are joined, and the result of the join replaces the subheap originally rooted at p. If p was a root and q becomes a heap root, q is unmarked. As a result of this transformation, either q is unmarked, or q is moved one level up and p is unmarked.

Sibling transformation. Assume that two marked nodes q and s are siblings, and that their parent p is not marked. Furthermore, assume that q is the left child of p. In this transformation 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 joined, and the result of that join replaces the subheap rooted originally at p. As the outcome, the height of a marked node increases by one and one marking disappears. Except for q and s, the distinguished ancestors of all other nodes remain the same, so no new weak-heap violations are introduced. This transformation reduces the number of marked nodes.

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 not marked. This transformation involves three steps. First, the subheaps rooted at p and r are split. Second, the resulting subheaps rooted at p and r are joined and the resulting subheap replaces 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 replaces the subheap originally rooted at p or r, depending on which is still unoccupied after the second step. If q or s becomes a heap root, it is unmarked. This transformation reduces the number of marked nodes.

4.2 Main transformations

We show next how the aforementioned four primitives are employed.

Run transformation. When nodes are marked, the basic idea is to let the markings bubble upwards when two marked nodes have 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 one or two of the top-most marked nodes of a run upwards and at the same time remove at least one marking. Assume that q is the leader of a run, which can be obtained from the run list, and s is the left child of q. 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 decreases, stop. Otherwise, s is now a right child of a non-marked parent. Accordingly, apply the parent transformation once or twice to s; this will reduce the number of marked nodes. (If the first parent transformation unmarks s, then 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 s is not marked. If the sibling of s is marked, apply the sibling transformation to s and its sibling, and stop. Otherwise, apply the cleaning transformation followed by the parent transformation to s and stop if s is unmarked. Otherwise, q and s are marked siblings with a non-marked parent; apply the sibling transformation to them.

Singleton transformation. If the number of marked nodes is larger than $\lfloor \lg n \rfloor$ and there are no runs, at least two singletons must have the same height, and their parents can only be marked if they are right children. A pair of such nodes is found from the singleton-pair list. Assume that q and s are two such singletons. If the sibling of q is marked, 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. In the case that the parent of q is marked, apply the parent transformation to q and stop. If the parent of s is marked, process s as q above and stop. If one or both of q and s are left children, neither the parents nor their siblings can be marked, so apply the cleaning transformation to them, if necessary, thereby ensuring that both are right children of their respective parents. Finally, apply the pair transformation to q and s.

5 Relaxed weak queues on a pointer machine

The implementation of run-relaxed heaps, summarized in Section 3, requires the power of random access. To support $O(1)$-time *insert* a root table is needed, and to support $O(1)$-time *decrease* a singleton table is needed. To avoid using the root table we resort to a simple application of number systems. To avoid using the singleton table, we use a grid that supports two-dimensional increments and decrements by pointer manipulations. If logarithmic-time *meld* is to be supported, we simulate an array on a pointer machine resulting in a slowdown that is logarithmic with respect to the size of the array. As a consequence, the bound for *decrease* becomes $O(\lg \lg n)$.

Next, we describe the auxiliary data structures, which we call the *heap store* and the *node store*, used to keep track of tree roots and potential violation nodes, respectively.

5.1 Heap store

There is a close connection between the sizes of the perfect weak heaps stored in a relaxed weak queue and the binary representation of its size n. If the normal binary representation is used, a perfect weak heap of size 2^h exists 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 corresponds to the addition of two 1-bits in position h propagating a carry to position $h + 1$. The solution described here is based on a redundant binary counter [7] and is easily implementable on a pointer machine. 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. A 2 means there are two heaps, a 1 means there is only one heap, and a 0 means there is no heap of that particular size.

To make the connection precise, we need the following definitions. In a *redundant binary system*, a non-negative integer d is represented as a sequence of digits $\langle d_0 d_1 \ldots d_k \rangle$, d_0 being the least-significant digit, d_k the most-significant digit, and $d_i \in \{0, 1, 2\}$ for all $i \in \{0, 1, \ldots, k\}$, such that $d = \sum_{i=0}^{k} d_i \cdot 2^i$. The redundant binary representation $\langle d_0 d_1 \ldots d_k \rangle$ of d is said to be *regular* [7] if any 2 is preceded by a 0, possibly having a sequence of 1's in between. A subsequence of $\langle d_0 d_1 \ldots d_k \rangle$, which is of the form $\langle 01^{\alpha}2 \rangle$ for $\alpha \in \{0, 1, \ldots, k-1\}$, is called a *block*. That is, every 2 must be part of a block, but there can be 0's and 1's that are not part of a block. For example, the sequence $\langle 102012 \rangle$ (least significant digit given first) represents integer 89 and contains two blocks.

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. Hence, our realization consists of two parts: a *heap sequence* containing perfect weak heaps in increasing order of height, and a *join schedule* with references to all delayed joins also in increasing order of height.

A heap store provides the operation *inject* that inserts a perfect weak heap no larger than any of the existing heaps into the heap store, and the operation *eject* that removes and returns a perfect weak heap of the smallest size from the heap store. Our priority-queue operations rely on a stringent form of *inject*: If d_i is the digit corresponding to the position where injection takes place, then all digits before d_i are 0's and d_i is either a 0, or a 1 that is not part of a block. We only aim at supporting these *strict* injections.

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. (Note that the unused pointer to the left child is reused for this purpose.) 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 height, the join of which is delayed.

Each *inject* involves two steps: an increment followed by a fix. In an *increment* we inject the given perfect weak heap into the heap sequence and, if there is another perfect weak heap of the same height, a pointer to this pair is pushed onto the join schedule. In a *fix* we join the first two perfect weak heaps that are of the same height (if there are any). The perfect weak heaps to be joined are found by consulting the join schedule. More concretely, the pointer at the 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 replaces the two heaps in the heap sequence. If there exists another perfect weak heap of the same height as the resulting weak heap, a pointer indicating this pair is pushed onto the join schedule. Since all such operations are performed in $O(1)$ time, the worst-case cost of *inject* is $O(1)$.

Each *eject* removes the smallest perfect weak heap from the heap sequence, sets the back pointer at the root of that heap to *null*, and pops the top of the join schedule if the heap forms a pair with some other heap. The worst-case cost of *eject* is $O(1)$.

The correctness of *eject* is obvious. As to the correctness of *inject*, the key property is that the stack-based scheduling of delayed joins guarantees that the representation remains regular even if only one join is performed in connection with each injection.

This property is part of computing folklore. It is proved [7, p. 53 ff.] in the special case where only increments of the least-significant digit are allowed (See [21, Section 9.2.4] for a more general case where increments of arbitrary digits are allowed.) To keep the paper self-contained, we prove that the regularity invariant is maintained when strict injections are performed.

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

Proof. Assume that the representation of the number corresponding to the heap sequence is regular before the operation and that the increment involves digit d_i, $i \geq 0$. According to the definition of strict injection, all digits before d_i (if any) must be 0, d_i cannot be 2, and d_i cannot be 1 that is part of a block. Thus, an increment is always possible and after the increment $d_i \in \{1, 2\}$. To complete the proof, we show that the following fix reestablishes the regularity invariant if the increment invalidates it.

Assume that the digit being fixed up is d_j, $j \geq i$. Now we consider three cases depending on the state of digit d_{j+1} following d_j.

Case 1. $d_{j+1} = 0$, which is not part of a block. After the fix, $d_j = 0$, $d_{j+1} = 1$, and the digits before and at d_j are not part of a block. If $i = j$ or if d_i started a block before the increment, the representation becomes regular.

Case 2. $d_{j+1} = 0$, which starts a block. After the fix, $d_j = 0$, $d_{j+1} = 1$, and the digits before d_j are not part of a block. The block is then one digit longer. If $i = j$ or if d_i started a block before the increment, the representation becomes regular.

Case 3. $d_{j+1} = 1$. By the regularity invariant, d_{j+1} cannot be part of a block. After the fix, $d_j = 0$, $d_{j+1} = 2$, and the digits before d_j are not part of a block. This creates a new block of length two. If the representation was not regular after the increment, it becomes regular after the fix. □

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

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

Proof. In the redundant binary system, the binary representation of n has at most $\lfloor \lg n \rfloor + 1$ non-zero digits. By the previous lemma, the representation of the number corresponding to the heap sequence of a relaxed weak queue is maintained regular. Hence, any 2 must be preceded by a 0. This directly implies the claimed upper bound on τ. □

5.2 Node store

The primary purpose of the node store is to keep track of potential violation nodes, and its secondary purpose is to store the heights and types of the nodes. When handling potential violation nodes we follow quite closely the guidelines given in [8], but we avoid the usage of a resizable array and only rely on linked structures. Because of this modification, all node-store operations work on a pointer machine.

For a relaxed weak queue of size n, the node store is a $4 \times (\lfloor \lg n \rfloor + 1)$ *grid* $(a_{i,j})$ of state objects, where $i \in \{0, 1, 2, 3\}$ and $j \in \{0, 1, \ldots, \lfloor \lg n \rfloor\}$. The state objects are identical in one respect: $a_{i,j}$ 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 *rows* $A_i = \langle a_{i,0}, \ldots, a_{i,\lfloor \lg n \rfloor} \rangle$ for $i \in \{0, 1, 2, 3\}$ are used to recall the type (non-marked, member, leader, singleton) and the height of nodes. We assume that the growth and shrinkage of the grid are taken care of by the joins and splits performed at the heap store. The perfect weak heap of the largest height is recalled and each time that heap is involved in a join, a new column is added to the grid. In a similar vein, when the largest perfect weak heap is split, the last column is removed from the grid.

All non-marked nodes share the state objects on row A_0 and no other information is associated with them. Each non-marked node of height h has a pointer to state object $a_{0,h}$. All run members use the state objects on row A_1 in a similar way. For each run leader of height h, we store an additional *leader object* which stores a pointer to that node and another pointer to state object $a_{2,h}$. Correspondingly, each leader has a pointer back to its leader object. All leader objects are kept in a doubly-linked list, called the *run 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, corresponding to a singleton of height h, has a pointer to state object $a_{3,h}$. All singleton objects for singletons of the same height are kept in a doubly-linked list. Each state object $a_{3,h}$ stores a pointer to a *pair object* which indicates whether the singleton-object list starting from $a_{3,h}$ contains more than one object. Each pair object has a pointer back to the state object. If the singleton list contains less than two objects, the pointer in $a_{3,h}$ reserved for a pair object is *null*. All pair objects are kept in a doubly-linked list, called the *pair list*.

The node store must provide the operation *mark* that marks a node or does nothing if the node is a root or if it is marked, the operation *unmark* that unmarks a node and does nothing if the node is not marked, and the operation *reduce* that unmarks an arbitrary marked node if possible. An inserted node is not marked, so its node-store pointer points to $a_{0,0}$. As a result of a priority-queue operation, the height of a node may be increased or decreased one at the time and the type of a node may change as well. In such case, the node-store pointer is updated, but only involving a constant amount of work. The node-store operations *mark*, *unmark*, and *reduce* require $O(1)$ worst-case time on a pointer machine.

5.3 Operations

The minimum node is returned by *find-min*. Since a pointer to this node is maintained, this operation is accomplished in $O(1)$ worst-case time.

In *insert*, considered as a perfect weak heap of height 0, the node to be inserted is placed into the heap store by invoking *inject*. If the element stored at this node is smaller than the current minimum element, the minimum pointer is updated accordingly. Since *inject* requires $O(1)$ worst-case time and possibly one element comparison, *insert* requires $O(1)$ worst-case time and at most two element comparisons.

In *decrease*, after the old element is replaced by the new element, the node is marked by invoking *mark*, and the number of marked nodes is reduced, if necessary, by invoking *reduce*. If the new value is smaller than the current minimum, the minimum pointer is changed to point to the decreased node. Compared to [8], we make the decreased node unconditionally marked because it does not have a pointer facilitating a fast check whether a heap-order violation is introduced or not. It follows that *decrease* requires $O(1)$ worst-case time. At most three element comparisons are done when invoking *reduce* and one more comparison is needed to check the validity of the current minimum, giving a total of four element comparisons.

There are several alternatives for implementing *delete*; the one to be described relies on the extraction technique used in the original implementation of run-relaxed heaps [8] (see also [11, 12]). We carry out *extract*, that returns a node q, 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 until q has no right child: (a) Split the perfect weak heap rooted at q. Let r be the root of the other subheap. (b) If r is marked, remove its marking by invoking *unmark*. (c) Insert the subheap rooted at r into the heap store by invoking *inject*.

To delete a node p, we splice out the subheap rooted at p, use the extracted node q to replace p, reestablish the weak-heap order, and put the resulting subheap in place of the spliced-out subheap. More precisely, the deletion of p is implemented as follows.

Step 3. If p is marked, remove the marking by invoking *unmark*.

Step 4. If p and q are the same node, go to Step 9.

Step 5. Repeat until p has no right child: (a) Split the subheap rooted at p. Let s be the root of the other subheap. (b) If s is marked, remove its marking by invoking *unmark*. (c) Push the subheap rooted at s onto a temporary stack.

Step 6. Repeat until the temporary stack is empty: (a) Pop the top of the stack. Let s be the root of the subheap popped. (b) 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. That is, update the parent and left-child pointers of q, one of the child pointers of the former parent of p, the parent pointer of the left child of p, and consider the special cases where the two neighbouring nodes do not exist.

Step 8. If p was a root, substitute the perfect weak heap rooted at p in the heap store with that rooted at q, else make q a marked node by invoking *mark*.

Step 9. Reduce the number of marked nodes by invoking *reduce*, if necessary (possibly once because of the new marked node introduced, and possibly once more compensating for the decrement of the size of the queue).

Step 10. If p contained the current minimum, scan all roots and marked nodes for a new minimum and update the minimum pointer.

According to our analysis for the heap-store and node-store operations, Step 1, Step 2(b), Step 2(c), Step 3, Step 5(b), Step 8, and Step 9 take $O(1)$ worst-case time. Also, joins and splits in Step 2(a), Step 5(a), and Step 6(b); stack operations in Step 5(c) and Step 6(a); and pointer manipulations in Step 4 and Step 7 take $O(1)$ worst-case time. For a relaxed weak queue of size n, the height of all perfect weak heaps is at most $\lfloor \lg n \rfloor$. Hence, in Step 2, Step 5, and Step 6 the number of iterations is bounded by $\lfloor \lg n \rfloor$, and the total running time of these steps is bounded by $O(\lg n)$.

The number of perfect weak heaps is at most $\lfloor \lg n \rfloor + 1$ and the number of marked nodes is at most $\lfloor \lg n \rfloor$. It is possible to iterate over the roots as well as the marked nodes. Therefore, Step 10 requires $O(\lg n)$ time.

To sum up, *delete* requires $O(\lg n)$ worst-case time. Only Step 2, Step 6, Step 9, and Step 10 involve element comparisons. Actually, Step 2 can be improved by ignoring the joins in Step 2(c); even without them the representation of the number corresponding to the heap sequence will remain regular. By 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 *reduce* at most three element comparisons are performed, so in Step 9 at most six element comparisons are performed. In Step 10 at most $\lfloor \lg n \rfloor + 1$ roots and at most $\lfloor \lg n \rfloor$ marked nodes are visited, so at most $2 \lg n + O(1)$ element comparisons are performed. In total, the number of element comparisons performed is at most $3 \lg n + O(1)$.

The above treatment supports *extract* in logarithmic worst-case time. A constant worst-case-time implementation is still possible by making the structure rely on a different number system. The idea is to use either the zeroless quaternary number system [12],[21, Exercise 9.18], or the segmented redundant number system [21, Exercise 9.12] to ensure that the size of the smallest perfect heap is always a constant.

5.4 Space reduction

Compared to the original transformations given in [8], the advantage we have is that the transformations can be made to work even if the parent pointers are only maintained for the largest children; this saves one pointer. Another way of saving space was to avoid storing the heights within the nodes and instead utilize a pointer to the node store to get the height of a node; this saves another pointer.

Accordingly, it is natural to use four pointers per node when storing a perfect weak heap; one pointer pointing to the left child, one to the right child, one to the parent, and one to the node store. A root is distinguished from other nodes by letting its parent pointer be *null*. For each root, the pointer to its non-existing left child is reused to point to the heap store. By using the same trick as in [5], parent-child relationships can be represented using two pointers per node instead of three. The heap store and the node store only require a logarithmic number of memory cells. This implies that the space usage of a relaxed weak queue can be as low as $3n + O(\lg n)$ pointers in addition to the space needed to store the n elements.

5.5 Extending the operation repertoire with *meld*

A description of *meld* was not given in [8], but it is implementable on a random-access machine in $O(\min\{\lg m, \lg n\})$ worst-case time, where m and n denote the sizes of the two priority queues to be melded. To achieve this bound, a singleton table should be implemented using a resizable array. Let h_Q denote the height of the largest perfect weak heap in priority queue Q. Without loss of generality, we can assume that $h_Q \geq h_R$ for the priority queues Q and R to be melded. A simple approach to *meld* is to *eject* all perfect weak heaps of height at most h_R from both heap sequences, and to *inject* them one by one in descending order of height into the heap sequence of Q. The run-singleton structures are melded by transferring the markings one by one from R to Q, and possibly calling *reduce* after each transfer.

In our pointer machine implementation, which did not involve *meld*, we used a grid to replace, among other things, the singleton table. Every node could efficiently access an object representing its type and height. With *meld* in action all the singletons of the same height from different queues would be grouped together, and hence we could not identify in constant time whether a pair of singletons is from the same queue or not when performing a reduction. On the other hand, if we maintain a separate grid for each queue, we would have to change the pointer that points to the local grid for each node, in at least one queue, to point to the new common grid. In summary, we would sacrifice with the efficiency of either *decrease* or *meld*.

To incorporate an efficient *meld*, we implement the singleton table by simulating a resizable array on a pointer machine. Given the binary representation of a height, the corresponding location in the singleton table can be accessed without relying on the power of a random-access machine by having a complete binary tree built above the grouped singletons. Starting from the root, we sequentially scan the bits in the representation of a height and move either to the right or left until we reach the sought location. We call this tree the *local index*. Instead of the grid, we maintain a doubly-linked list of entities, shared by all queues, representing the heights of the nodes. Every node can access the entity corresponding to its height in constant time by using its node-store pointer. We also build a complete binary tree above these entities. We call this tree the *global index*. Starting from a leaf, the bits in the binary representation of a height are obtained by accessing the tree upwards.

Because the sizes of different priority queues are changing and the global index should be able to service all priority queues, the size of the global index must be adjusted to match the size of the largest priority queue. We maintain four indexes; one index is under construction, one is under destruction, and two are complete (one of them is in use). The sizes of the indexes are consecutive powers of two, and there is always an index that is larger and another that is smaller than the index in use. At the right moment, we make a specific index the current one, start destroying an index, and start constructing a new one (or resume reconstructing one). To accomplish such a change, each height entity must have three pointers, and we maintain a switch that indicates which pointer leads to the index in use. Index constructions and destructions are done incrementally by distributing the work evenly on the operations using the index. This global rebuilding only requires an extra constant amount of work per operation.

To decide the right moment for index switching, we keep track of the sum of the sizes of all priority queues, denoted N. A counter that holds the binary representation of N is maintained, together with a pointer to its most-significant 1-bit; when this pointer is moved one bit either to the right or left, we switch for the appropriate index. On a pointer machine, such a binary counter is represented as a linked list of bits. An increment and a decrement of the binary counter is to be done in worst-case constant time, which is impossible. Fortunately, we can afford to delay a logarithmic number of increments or decrements while updating the counter. (To avoid problems due to delays, we simply use an index that is slightly bigger than what we need.)

The local indexes are maintained and rebuilt in a similar manner as the global one. However, the size of a local index is proportional to $\lg n$ (not to $\lg N$), where n is the current size of the priority queue owning this index. To be able to switch between indexes, a local counter is maintained as the global counter. Moreover, two local counters can be added in time proportional to the length of the shorter counter.

When two queues are melded, in addition to melding their run-singleton structures, we keep the local index of the larger queue and destroy that of the smaller, and add the value of the smaller local counter to that of the larger. The worst-case running time of *meld* is still $O(\min\{\lg m, \lg n\})$. However, *decrease* would now require $O(\lg\lg n)$ worst-case time. To obtain this, we use the global index to get the trailing bits in the representation of the height of the priority queue in question. Although the global index could give about $\lg\lg N$ bits, only $\lg\lg n$ of them are used when accessing the local index (the other leading bits are 0's). Except for the $O(\lg N)$ pointers of the global index, the amount of space used by the priority queues remains the same.

6 Conclusions

We find the connection between perfect weak heaps and heap-ordered binomial trees interesting (see, e.g. [5, 6, 16, 22, 24]). Using this connection, it is possible to implement a worst-case-efficient priority queue using only binary trees. We showed how the relaxation technique used in a run-relaxed heap [8] could be immigrated 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 conceptually simpler. In earlier realizations relying on binary trees, the time bound for *decrease* is $O(1)$ in the amortized sense [16, 22] or $O(\lg n)$ in the worst case [5, 24]. Compared to Fibonacci heaps [15], the time bounds guaranteed by relaxed weak queues are worst-case rather than amortized, but the bounds for *decrease* and *meld* are weaker.

We showed that the amount of space used by a relaxed weak queue can be made as small as $3n + O(\lg n)$ pointers besides the space used to store the n elements. In its original form [8], 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 if the node is a potential violation node. A fat heap [17, 18] uses $4n + O(\lg n)$ pointers, since each node must store three parent-child-sibling pointers and a rank. It should be pointed out that, for any fixed $\varepsilon > 0$ and sufficiently large $n > n(\varepsilon)$, the space requirement of run-relaxed heaps, fat heaps, and relaxed weak queues can be reduced to $(1 + \varepsilon)n$ pointers using the bucketing technique, with the trade-off of increasing the complexity of the underlying data structures [4].

We showed how relaxed weak queues can be implemented on a pointer machine. On a random-access machine, *meld* can be supported in logarithmic worst-case time and *decrease* in $O(1)$ worst-case time. On the other hand, *decrease* takes $O(\lg\lg n)$ worst-case time on a pointer machine, because it is difficult to group singletons of the same height together. If *meld* is to be supported in logarithmic worst-case time, a fat heap [17, 18] also requires the capabilities of a random-access machine. 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 though *meld* can be carried out in $O(1)$ worst-case time. So, the question remains whether it is possible to design a more efficient priority queue—supporting *find-min*, *insert*, *delete*, *decrease*, and *meld*—on a pointer machine.

References

[1] A.M. Ben-Amram, What is a "pointer machine"?, *SIGACT News* 26(2) (1995), 88–95.

[2] G.S. Brodal, Worst-case efficient priority queues, in *Proceedings of the 7th Annual 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] H. Brönnimann, J. Katajainen, and P. Morin, *Putting your data structure on a diet*, CPH STL Report 2007-1, Department of Computer Science, University of Copenhagen (2007).

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

[6] S. Carlsson, J.I. Munro, and P.V. Poblete, An implicit binomial queue with constant insertion time, in *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science, vol. 318, Springer-Verlag (1988), 1–13.

[7] 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).

[8] 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.

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

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

[11] A. Elmasry, C. Jensen, and J. Katajainen, Multipartite priority queues, *ACM Transactions on Algorithms* 5 (2008), Article 14.

[12] A. Elmasry, C. Jensen, and J. Katajainen, Two-tier relaxed heaps, *Acta Informatica* 45 (2008), 193–210.

[13] A. Elmasry, C. Jensen, and J. Katajainen, Two new methods for constructing double-ended priority queues from priority queues, *Computing* 83 (2008), 193–204.

[14] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. The pairing heap: a new form of self_adjusting heap. *Algorithmica* 1(1) (1986), 111–129.

[15] 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.

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

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

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

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

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

[21] C. Okasaki, *Purely Functional Data Structures*, Cambridge University Press (1998).

[22] 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).

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

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

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