# The Weak-Heap Family of Priority Queues in Theory and Praxis

**Stefan Edelkamp**[1] **Amr Elmasry**[2] **Jyrki Katajainen**[2]

[1] Faculty 3—Mathematics and Computer Science, University of Bremen
PO Box 330 440, 28334 Bremen, Germany
Email: edelkamp@tzi.de

[2] Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark
Email: {elmasry,jyrki}@di.ku.dk

**Abstract**

In typical applications, a priority queue is used to execute a sequence of $n$ *insert*, $m$ *decrease*, and $n$ *delete-min* operations, starting with an empty structure. We study the performance of different priority queues for this type of operation sequences both theoretically and experimentally. In particular, we focus on weak heaps, weak queues, and their relaxed variants. We prove that for relaxed weak heaps the execution of any such sequence requires at most $2m + 1.5n \lg n$ element comparisons. This improves over the best bound, at most $2m + 2.89n \lg n$ element comparisons, known for the existing variants of Fibonacci heaps. We programmed six members of the weak-heap family of priority queues. For random data sets, experimental results show that non-relaxed versions are performing best and that rank-relaxed versions are slightly faster than run-relaxed versions. Compared to weak-heap variants, the corresponding weak-queue variants are slightly better in time but not in the number of element comparisons.

*Keywords:* Data structures, priority queues, weak heaps, weak queues, shortest paths

## 1 Introduction

A priority queue is an important data structure that is used for implementing many fundamental algorithms, like Dijkstra's algorithm for computing shortest paths (Dijkstra 1959) and Prim's algorithm for finding minimum spanning trees (Prim 1957). For a comparison function operating on a totally ordered set of elements, priority queues often support the operations *insert*, *delete*, *delete-min* (extracting an element with the minimal value), and *decrease* (decreasing the value of a given element).

For some applications, it is natural to require *find-min* (determining the location of the current minimum) to be a constant-time operation. This means that the other operations are responsible for updating the pointer to the minimum after each modification to the data structure. However, in many applications, fast *find-min* is not essential since it is always followed by *delete*. Hence, instead of updating the minimum pointer after each modification, *delete-min* finds the minimum before the deletion.



To guarantee a theoretically-optimal behaviour in applications, as those mentioned above, a priority queue that provides *decrease* in $O(1)$ (amortized or worst-case) time must be used. Fibonacci heaps (Fredman and Tarjan 1987) were the first priority queues that achieve this. In fact, they provide optimal amortized time bounds for all operations ($O(1)$ per *insert* and *decrease*, and $O(\lg n)$ per *delete-min*, where $n$ is the size of the heap). In previous experimental studies (Stasko and Vitter 1987, Cho and Sahni 1998, Bruun et al. 2010), binary heaps (Williams 1964), pairing heaps (Fredman et al. 1986), leftist trees (Crane 1972), or weak queues (Vuillemin 1978) have been reported to perform best, even though none of them is theoretically optimal. In addition, several recent papers present alternatives to Fibonacci heaps and claim their superiority (Takaoka 2003, Haeupler et al. 2009, Chan 2009, Elmasry 2010).

In this paper we investigate the theoretical and practical performance of weak heaps, weak queues, and their relaxed variants. A perfect weak heap is a binary-tree representation of a heap-ordered binomial tree (Vuillemin 1978). A weak queue, as it was named in (Elmasry et al. 2005), is a binomial queue represented using binary trees, i.e. it is a collection of perfect weak heaps. In general, a weak heap (Dutton 1993) is not necessarily perfect, i.e. its size need not be a power of two.

Relaxed heaps were introduced by Driscoll et al. (1988) to support *decrease* in $O(1)$ worst-case time. The basic idea, applied by Driscoll et al. to binomial queues (Vuillemin 1978), is to permit some nodes to violate heap order, i.e. the element stored at a potential violation node may be, but is not necessarily, smaller than the element stored at its parent. Driscoll et al. described two forms of relaxed heaps: run-relaxed heaps and rank-relaxed heaps. Run-relaxed heaps achieve the same bounds as Fibonacci heaps, even in the worst case per operation. For rank-relaxed heaps, the $O(1)$ time bound for *decrease* is amortized, but the transformations needed for reducing the number of potential violation nodes are simpler than those required by run-relaxed heaps.

We are interested in priority queues that support all operations as efficiently as relaxed heaps. Furthermore, we work towards optimizing the bounds on the number of element comparisons needed to perform the underlying operations. The core difference between our structures and those in (Driscoll et al. 1988) is that the latter rely on heap-ordered binomial trees (Vuillemin 1978) while ours use pointer-based weak heaps. In comparison to relaxed heaps, our key improvements are twofold. First, we immigrate the transformations of Driscoll et al. into the binary-tree setting; we call the underlying forest of relaxed weak heaps a *relaxed weak queue*. Second, to improve

the bound on the number of element comparisons per *delete-min* operation, we use a single pointer-based weak heap instead of a weak queue, and show how to perform the priority-queue operations accordingly; we call the resulting structure a *relaxed weak heap*. We remark that run-relaxed weak queues were introduced in (Elmasry et al. 2005) and rank-relaxed weak queues in (Edelkamp 2009). The description of both weak-queue variants in this paper is a refinement to the ones in the two technical reports.

We also show that relaxed weak heaps can be made competitive to, and even improve over, all existing priority queues. Starting with an empty structure, the execution of any sequence of $n$ *insert*, $m$ *decrease*, and $n$ *delete-min* operations requires at most $2m + 1.5n \lg n$ element comparisons for rank-relaxed weak heaps, while the best bound known for Fibonacci heaps is $2m + 2.89n \lg n$ element comparisons, and for other priority queues even higher.

Experimental results show that our implementations of weak queues and weak heaps are competitive with highly-tuned implementations of binary, Fibonacci, and pairing heaps, even in an application like the computation of shortest paths. For weak queues, one additional advantage is small space consumption. We show that, at any given time, rank-relaxed weak queues store at most one potential violation node per level. This leads to the use of at most $2n + O(\lg n)$ full-length words and $n$ words of $\lg \lg n + O(1)$ bits each, in addition to the space used by the elements.

The main body of the text consists of three parts. After reviewing some related structures in Section 2, we recall the functionality of relaxed heaps and introduce relaxed weak queues as their binary-tree-based variants in Section 3. In Section 4, we introduce relaxed weak heaps as their single-heap variants that achieve a better bound on the number of element comparisons performed. In Section 5, we report the experimental results comparing our priority-queue implementations to their natural competitors, when computing the single-source shortest paths using Dijkstra's algorithm, and when handling some other synthetic operation sequences. We end the paper with a short conclusion in Section 6.

## 2 Related Structures

In the basic setting, we consider operation sequences that consist of $n$ *insert*, $m$ *decrease*, and $n$ *delete-min* operations, starting with an empty structure. Using some known data structures, reviewed briefly in this section, our reference sequence can be executed in $\Theta((m+n) \lg n)$ or $\Theta(m+n \lg n)$ worst-case time. The difference between these two bounds is significant in theory, but that seems not to be typically the case in practice. For $m = O(n)$ the bounds are the same, and for $m = \Omega(n \lg n)$ the difference is a logarithmic factor. Often, like in Dijkstra's algorithm, the *decrease* operations are conditional: Only if the given value is smaller than the current value, a *decrease* operation is executed; otherwise, nothing is done. Hence, data structures supporting *decrease* in $\Theta(\lg n)$ time are noteworthy candidates for everyday use.

Below, we give a summary of the related structures that we compare against in this paper.

**Binary heaps.** Assuming that all *insert* operations are executed first, binary heaps can process our sequence of operations in $O((m + n) \lg n)$ worst-case time, involving at most $m \lg n + 2n \lg n + 2n$ element comparisons. Often in the literature, binary heaps are considered in a context where only *insert* and *delete-min* operations are to be supported. To support *decrease* (and *delete*), one must ensure that pointers given to the elements remain valid; this means that the elements are to be stored indirectly. In a typical case, when operating on random data sets, $O(1)$ work is expected in connection with each *insert* and *decrease*; and *delete-min* is expected to perform $\lg n + O(1)$ element comparisons (when relying on a bottom-up heapifying strategy).

**Fibonacci heaps.** With the potential function used in (Fredman and Tarjan 1987), for our reference sequence, Fibonacci heaps can be shown to perform at most $3m + 2.89n \lg n$ element comparisons. However, using a slight modification, the bound can be brought down to $2m + 2.89n \lg n$. The number of trees in the heap can grow up to $n$, and a single *delete-min* can take $\Theta(n)$ time. We can restrict the number of roots to $1.44\lceil \lg n \rceil$, but *decrease* may then require $\Theta(\lg n)$ time in the worst case (Kaplan and Tarjan 2008). Hence, Fibonacci heaps cannot match the worst-case performance of run-relaxed heaps. Our experiments show that the reference sequence can be handled faster by a carefully-coded Fibonacci-heap implementation than by an existing implementation in LEDA (Mehlhorn and Näher 1999). In turn, we raise the question about the reliability of experimental results reported in earlier studies that used an implementation of Fibonacci heaps as a baseline.

**Pairing heaps.** In the theory community, the analysis of pairing heaps has attracted lots of attention. Fredman (1999) was able to show that the amortized cost of *decrease* is not a constant. In spite of this, for a long time, it has been known (see, e.g. (Stasko and Vitter 1987)) that the practical performance of pairing heaps is good. The number of element comparisons performed per *delete-min*, though $O(\lg n)$ in the amortized sense, can be up to $\Theta(n)$ in the worst case. In spite of the theoretical setback, because of its simplicity and ease of implementation, a carefully-coded pairing-heap implementation is practically relevant.

**2-3 heaps.** This data structure was one of the early alternatives to Fibonacci heaps. It performs at most $2m + 3n \lg n + n$ element comparisons when handling our reference sequence (Takaoka 2003). A 2-3 heap is also efficient in practice as verified by our experiments. The main drawback is maintainability: very few people understand the internals of its implementation, and accordingly are able to change its code.

**Violation heaps.** This data structure is one of the recent alternatives to Fibonacci heaps (Elmasry 2010). The theoretical analysis hides larger constants than those proved for Fibonacci heaps. However, this can be an artifact of the analysis, not the data structure itself. Our experiments showed that in several cases this data structure performs well.

## 3 Relaxed Weak Queues

A *weak queue* is a binomial queue (Vuillemin 1978) implemented as a collection of binary trees. The root of a tree has only one child, the right child if any, which in turn roots a complete binary tree. Hence, the size of every tree is a power of two. The *rank* of a tree is the binary logarithm of its size. The *distinguished ancestor* of a node $x$ is the parent of $x$ if $x$ is a right child, and the distinguished ancestor of the parent of $x$ if $x$ is a left child. The *weak-heap ordering* enforces that no element is smaller than that at its distinguished ancestor. These trees (with the weak-heap-order property) are called *perfect weak heaps*.

Two basic primitives used in the manipulation of perfect weak heaps are *join* and *split*. A *join* links

two trees of rank $r$ by making the root with the larger value, say $y$, the child of the other root, say $x$. The right subtree of $x$ becomes the left subtree of $y$, and the rank of $x$ is increased to $r + 1$. A *split* is the reverse of a *join*; a tree of rank $r + 1$ is divided into two trees of rank $r$.

### 3.1 Registries

The data structure that keeps track of the tree roots is called a *root registry*. In the standard implementation (Vuillemin 1978), there is at most one tree per rank. Several alternatives have been proposed to achieve better per-operation bounds in the worst case (see, e.g. (Driscoll et al. 1988, Elmasry et al. 2008a)). In the implementation proposed in (Driscoll et al. 1988), any number of trees per rank is allowed, but an upper bound on the total number of trees is set. A *root table* is maintained, which is a resizable array accessed by rank. Each entry of the root table contains a pointer to the beginning of a list of roots having this particular rank. For each entry referring to more than one root, a counterpart is kept in a *root-pair list*.

It must be possible to add a root to a root registry, extract a given root, and reduce the number of roots when it exceeds a threshold. To support extraction, each root should know its location within the list where it is in. An easy way of implementing the collision lists is to use the left-child pointer (which is unused) and the parent pointer (which is also unused) of each root for linking. Then an extraction is just a removal from a doubly-linked list. For reduction, the root-pair list is consulted and two trees of the same rank are joined.

In a *relaxed weak queue*, the trees are not necessarily weak-heap ordered; a *marked node* is potentially weak-heap-order violating. A root is always non-marked. The data structure that keeps track of the marked nodes is called a *mark registry*. It must be possible to add a new marked node to such a registry, extract a given marked node, and reduce the number of the marked nodes if it exceeds a threshold.

The key ingredient is a set of transformations used to reduce the number of marked nodes (for a detailed discussion, see (Elmasry et al. 2005)). Each transformation involves a constant number of structural changes. The primitive transformations are visualized in a pictorial form in Figure 1. A *cleaning transformation* makes a marked left child into a marked right one, provided its sibling and parent are both non-marked. A *parent transformation* reduces the number of marked nodes or pushes the marking one level up. A *sibling transformation* reduces the number of markings by eliminating the markings of two siblings, while generating a new marking at the level above. A *pair transformation* has a similar effect, but it operates on two non-sibling nodes of the same rank. The cleaning transformation does not require element comparisons, each of the parent and the sibling transformations involves one element comparison, and the pair transformation involves two element comparisons.

For run-relaxed versions, we adopt a lazy strategy where the number of markings is reduced by one, if possible, as compensation for each new marking. For rank-relaxed versions, we adopt an eager strategy where the transformations are employed until they can no longer reduce the number of markings.

### 3.2 Run-Relaxed Weak Queues

Let $\tau$ denote the number of trees and $\lambda$ the number of marked nodes. An invariant is maintained that $\tau \leq \lfloor \lg n \rfloor + 1$ and $\lambda \leq \lfloor \lg n \rfloor$ after each operation.

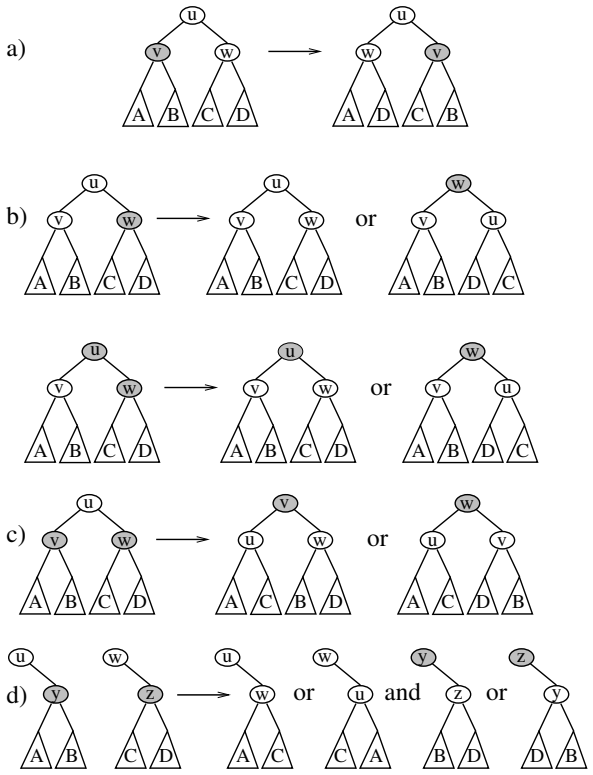


Figure 1: Primitive transformations: a) cleaning transformation at $v$, b) parent transformation at $w$, c) sibling transformation at $v$, and d) pair transformation for $y$ and $z$. Grey nodes are marked.

A marked node is a *member* if it is a left child and its parent is marked. A marked node is a *leader* if its left child is a member, and if it is either a right child or a left child whose parent is non-marked. A maximal chain of members preceded by a leader is called a *run*. A marked node that is neither a member nor a leader is called a *singleton*. To summarize, we divide the set of nodes into four disjoint categories: non-marked nodes, run members, run leaders, and singletons.

The mark registry can be implemented as follows (Driscoll et al. 1988). All run leaders are kept in a *run list*, and all run members are accessed via their leaders. All singletons are kept in a *singleton table*, which is a resizable array accessed by rank. Each entry of the singleton table contains a pointer to the beginning of a list of singletons having this particular rank. For each rank referring to more than one singleton, a counterpart is kept in a *singleton-pair list*. When a node becomes a singleton, a new item pointing to this node is added to the appropriate list of singletons and the singleton-pair list is updated if necessary. To facilitate extraction, each node stores a back pointer to its item once created. The worst-case time of marking and unmarking routines is $O(1)$.

Two types of compound transformations are used to reduce the number of marked nodes: 1) the *singleton transformation* is used for reducing the number of singletons, and 2) the *run transformation* is used for making runs shorter. If $\lambda$ exceeds the threshold, the transformations can be applied to reduce $\lambda$ by one. The rationale behind the transformations is that, when there are more than $\lfloor \lg n \rfloor$ marked nodes, there is a run of marked nodes, or there is at least one pair of singletons that root a subtree of the same rank. Hence, it is possible to apply one of the transformations. We show next how the primitive transforma-

tions are employed in the two compound transformations. For all cases, at most two element comparisons are performed to remove a marked node.

- **Run transformation.** When dealing with marked nodes, the basic idea is to let the markings bubble upwards until two marked nodes have the same rank. The runs cause a complication in this process since these can only be unravelled from above. Let $q$ be the leader of the given run, $r$ its left child, and $p$ its parent. We consider two cases:

  $q$ **is a left child (zig-zig case).** If the sibling of $q$ is marked, apply the sibling transformation at $q$ and stop. If the sibling of $r$ is marked, apply the parent transformation at that sibling and stop. Otherwise, apply the cleaning transformation at $q$. If the new sibling of $r$ is marked, apply the sibling transformation at $r$ and stop. Otherwise, apply the cleaning transformation followed by the parent transformation at $r$. If $r$ is still marked, $q$ and $r$ are marked siblings; apply the sibling transformation at $r$. In total, at most two element comparisons are necessary: one may be done by the parent transformation and another by the sibling transformation.

  $q$ **is a right child (zig-zag case).** Perform a split at $p$, apply the parent transformation at $r$, then join the resulting tree and that rooted at $q$. Finally, attach the root of the result of the join to the place where $p$ was originally. Two element comparisons are done: one by the join and another by the parent transformation.

- **Singleton transformation.** If $\lambda$ exceeds the threshold and there are no runs, at least two singletons must have the same rank and they have marked parents only if they are right children. A pair of such nodes is found with the help of the singleton-pair list. Assume that $q$ and $s$ are such singletons. If the parent of $q$ is marked, apply the parent transformation at $q$ and stop. Similarly, if the parent of $s$ is marked, apply the parent transformation at $s$ and stop. If the sibling of $q$ is marked, apply the sibling transformation at $q$ (or at its sibling depending on which of the two is a left child) and stop. Similarly, if the sibling of $s$ is marked, apply the sibling transformation at $s$ (or at its sibling) and stop. In accordance, if one or both of $q$ and $s$ are left children, neither their parents nor their siblings are marked. Apply the cleaning transformation at $q$ and $s$ to ensure that both are right children of their respective parents. Finally, apply the pair transformation for $q$ and $s$.

The *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 left-child pointers until a non-marked node is reached. Since the number of trees is at most $\lfloor \lg n \rfloor + 1$ and the number of marked nodes is at most $\lfloor \lg n \rfloor$, the worst-case time of *find-min* is $O(\lg n)$, involving at most $2\lfloor \lg n \rfloor$ element comparisons.

An *insert* is performed by adding a single-node tree, and appending a pointer to this node to the list at the first entry of the root table. If this is the second item in this entry, a pointer to this root-table entry is appended to the root-pair list. The number of trees is then reduced, if necessary, by removing a pair of trees from an entry of the root-pair list, joining the two trees, and adding the resulting tree to the structure. It follows that the worst-case time of *insert* is $O(1)$, involving at most one element comparison.

In *decrease*, after making the element replacement, the affected node is marked and an occurrence is inserted into the mark registry. A reduction of $\lambda$ is performed if possible. It follows that the worst-case time of *decrease* is $O(1)$, involving at most two element comparisons.

The *delete-min* operation invokes *find-min* (at most $2\lfloor \lg n \rfloor$ element comparisons) to locate a node $x$ with the minimum value. The distinguished ancestors of $x$ up to the root of the tree are identified, and each of them is sifted down leaving the root of the tree of $x$ vacant. As a result, each node on the path from the child of the deleted root to the leftmost leaf together with its right subtree forms a perfect weak heap; the heap and mark registries are updated accordingly. To maintain the bound on $\tau$, trees of equal rank are repeatedly joined until the threshold is reached (at most $\lfloor \lg n \rfloor$ element comparisons). To maintain the bound on $\lambda$, a reduction of $\lambda$ is performed if possible (at most two element comparisons). Hence, the worst-case time of *delete-min* is $O(\lg n)$, involving at most $3\lfloor \lg n \rfloor + 2$ element comparisons.

The performance of run-relaxed weak queues can be summarized as follows.

**Theorem 1** *For run-relaxed weak queues, starting with an empty structure, the execution of any sequence of $n$ insert, $m$ decrease, and $n$ delete-min operations requires at most $2m + 3n\lfloor \lg n \rfloor + n$ element comparisons. The worst-case execution time of this sequence is $O(m + n \lg n)$. Moreover, each insert and decrease runs in $O(1)$ worst-case time, and each delete-min in $O(\lg n')$ worst-case time, where $n'$ denotes the size of the heap prior to the operation.*

**Proof.** The number of $\lambda$-reductions is bounded by the total number of nodes that have ever been marked, which is at most $m$. Each mark removal involves at most two element comparisons. This accounts for at most $2m$ element comparisons. The number of $\tau$-reductions is bounded by the total number of trees that have ever been created, which is one per *insert* and at most $\lfloor \lg n \rfloor$ per *delete-min*. This accounts for at most $n\lfloor \lg n \rfloor + n$ element comparisons. The number of element comparisons involved in scanning to locate the minima is at most $2n\lfloor \lg n \rfloor$. The theorem follows by combining these bounds. □

### 3.3 Rank-Relaxed Weak Queues

Here the basic idea is to reduce the number of markings as much as possible after every new marking. Otherwise, the operations are executed as in run-relaxed weak queues. To remove markings eagerly, we enforce the following stronger invariants:

1. There exists at most one marked node per rank.
2. The parent of a marked node is non-marked.
3. A marked node is always a right child.

The last invariant forces us to make a modification to the *join* operation: If a marked node is made a left child, apply the cleaning transformation to it immediately after a *join*. After this change, the operations in the root registry do not break these invariants.

Assume that the invariants are valid, and consider what to do when a node is marked. If the right child of that node is marked, we apply the parent transformation at that child. Hereafter, we can be sure that both children of the marked node are non-marked. To

reestablish the invariants, we have to lift the marking upwards until we reach the root, or until none of the neighbouring nodes is marked and no other marked node of the same rank exists.

Let $q$ be the most-recently marked node, and let *parent*$(q)$ denote its parent. The propagation procedure has several cases; some of them can lift the marking one or two levels up:

$q$ **is a root.** Since a root cannot be marked, remove this marking and stop.

$q$**'s sibling is marked.** Since the sibling is marked, $q$ must be a left child. Apply the sibling transformation at $q$, and repeat the case checks for the resulting marked node at the level above.

$q$ **is a left child and all the neighbours of** $q$ **are non-marked.** Apply the cleaning transformation at $q$, and check for the next case.

$q$ **is a right child and** *parent*$(q)$ **is non-marked.** If there is a marked node of the same rank as $q$, apply the pair transformation for $q$ and this marked node, and repeat the case checks for the resulting marked node; otherwise stop.

$q$ **is a left child and** *parent*$(q)$ **is marked.** Since the parent is marked, it must be a right child. Remove one marking from this length-two run as in the zig-zag case of the run transformation, and repeat the case checks for the resulting marked node.

$q$ **is a right child and** *parent*$(q)$ **is marked.** Apply the parent transformation at $q$ and stop.

For this structure, run and singleton transformations are not applied, and marking and unmarking routines are easier. Our implementations of the associated data structures—root registry and mark registry—are simple and space-economical. A root registry is a resizable array storing at each entry a pointer to a root of that particular rank, if any. Additionally, two bit-vectors, each stored in a single word, are maintained. These give the ranks where there exist roots and the ranks where there are two or more roots, respectively. A mark registry is even simpler; it is a resizable array storing at each entry a pointer to a marked node of that particular rank, if any. Since we only aim at achieving good amortized performance, the standard doubling-and-halving technique can be used to implement the resizable arrays; no worst-case efficient resizable arrays are needed. A standard implementation of a weak-queue node uses three pointers, and a word storing the rank, a bit indicating whether the node is a root or not, and another bit indicating whether the node is marked or not. Hence, the amount of space used is $4n + O(\lg n)$ words in addition to the elements stored. The amount of extra words can be reduced to $3n + O(\lg n)$ by storing the parent-child relationships cyclically (Brown 1978).

The performance of rank-relaxed weak queues can be summarized as follows.

**Theorem 2** *For rank-relaxed weak queues, starting with an empty structure, the execution of any sequence of $n$ insert, $m$ decrease, and $n$ delete-min operations requires at most $2m + 3n\lfloor \lg n \rfloor$ element comparisons. The worst-case execution time of this sequence is $O(m + n \lg n)$.*

**Proof.** The proof is similar to the proof of Theorem 1, while noting the difference that $\lambda \leq \lfloor \lg n \rfloor - 1$ holds; this follows as a result of the second invariant. Hence, the number of element comparisons involved in finding the minima is at most $2n\lfloor \lg n \rfloor - n$. □

## 4 Relaxed Weak Heaps

Concerning the number of element comparisons performed for our reference sequence, we can do even better by relying on a single weak heap instead of a collection of perfect weak heaps. A *weak heap* is a binary tree whose root only has a right child if any, and whose elements obey the weak-heap ordering. Except for the root, the nodes that have at most one child are at the last two levels only.

In its original form (Dutton 1993, Edelkamp and Wegener 2000), a weak heap is implemented using an array of elements and an array of bits. In our treatment, we implement weak heaps using a linked representation. Also, we allow nodes that may violate the weak-heap ordering as discussed in Section 3. As for $\lambda$-reductions, we can either adopt the lazy strategy (applying a reduction after each new marking), or the eager strategy (removing the markings whenever possible). We call the data structures using these strategies a *run-relaxed weak heap* and a *rank-relaxed weak heap*, respectively.

Here, there are two differences compared to our previous treatment. First, every node stores its *depth*, not its rank (the depth of the root being zero). Second, we have to keep track of the leaves to make it possible to expand and contract the heap at the last level. We call this structure a *leaf registry*. In its implementation, we maintain two doubly-linked lists, one for each of the last two levels of the heap. All the nodes of the last level are leaves and are accordingly kept in the first list in arbitrary order. All the nodes that have at most one child at the second-to-last level are kept in the second list in arbitrary order. Using these two lists, a leaf can be appended to or removed from the last level of the weak heap in a straightforward manner. Naturally, a join is performed between two subheaps of the same depth. This ensures that the leaves are still at the last two levels, and the two lists of the leaf registry need not be altered.

For a relaxed weak heap of size $n$, we can settle $\lambda \leq \lceil \lg n \rceil - 1$. Recall that the height of the heap is $\lceil \lg n \rceil + 1$ and the root is never marked. When $\lambda = \lceil \lg n \rceil$, there are two marked nodes at the same level and/or the child of the root is marked. For both such cases, a $\lambda$-reduction is always possible.

Accompanying each $\lambda$-reduction, a constant number of nodes need to be swapped, and accordingly a constant number of pointers are updated. It is significant that subtrees are only swapped with other subtrees having the same depth; this ensures that the leaves stay at the last two levels, and the two lists of the leaf registry need not be altered. However, when a node is moved (by one of the transformations) from the last level to the second-to-last level and vice versa, the leaf registry must be updated accordingly.

An *insert* is performed by adding the new node, say $x$, as a leaf at the last level. To do that, we first pick a node, say $y$, from the list at the second-to-last level (currently having at most one child). We add the new node $x$ as a child of $y$, and accordingly append $x$ to the list of nodes at the last level, and remove $y$ from the other list if it now has two children. The node $x$ is then marked indicating that it may be violating. The *insert* operation is followed by reducing the number of marked nodes as appropriate.

A *decrease* is executed, as in the relaxed weak queues, by marking the affected node, and reducing the number of marked nodes as appropriate.

Assume that we are deleting a node $x$. Let $y$ be the last node on the *left spine* (the path from the root of a subtree to the leftmost leaf in that subtree) of the subtree rooted at the right child of $x$. Node $y$

can be identified by starting from the right child of $x$, and repeatedly traversing left children until reaching a node that has no left child. Furthermore, let $z$ be a node *borrowed* from the last level of the heap. Naturally, the leaf registry must be updated accordingly. Now, each node on the path from $y$ to the right child of $x$ (both included) is seen as a root of a weak heap. To create a subheap that can replace the subheap rooted at $x$, we traverse the path upwards; we start by joining $y$ and $z$, then we continue by joining the resulting subheap and the subheap rooted at the parent of $y$, and so on. At last, $x$ is removed and the root of the result of the repeated joins is attached in its place. Since this attached node may be violating, it is marked. Because of the possible increase of $\lambda$ and the reduction in the number of elements, the operation is followed by $\lambda$-reductions as needed.

The correctness of this procedure follows from the correctness of *join*; that is, the resulting subheap is weak-heap ordered. It should be noted that it is fully appropriate to use *join* here. By setting the depth of $z$ equal to that of $y$ at the beginning, in any later phase the depth of the root of the resulting subheap is set to be one less than its earlier depth. The depths of all other nodes remain unchanged.

For a *delete-min* operation, the minimum is either at a marked node or at the root. After the minimum node is localized, it is removed as described above. Observe that, if the minimum is at a marked node, the final marking will not increase $\lambda$, and if the minimum is at the root, there is no need to mark the new root.

Since $\lambda$ is bounded by $\lceil \lg n \rceil - 1$, finding the minimum requires at most $\lceil \lg n \rceil - 1$ element comparisons. Each *join* requires $O(1)$ time and involves one element comparison. The number of nodes on the path from $y$ to the right child of $x$ is at most $\lceil \lg n \rceil$. Thus, the number of joins, as well as the number of element comparisons, performed to restore the weak-heap ordering is at most $\lceil \lg n \rceil$. A further optimization can be done; we either borrow the node $y$, if it is a leaf at the last level, or an arbitrary leaf $z$ as before. By starting the repeated joins from the second-to-last level, the number of joins will be at most $\lceil \lg n \rceil - 1$. It follows that, in total, the number of element comparisons performed by *delete-min* is at most $2\lceil \lg n \rceil - 2$ plus those performed when removing marked nodes.

The performance of run-relaxed weak heaps can be summarized as follows.

**Theorem 3** *For run-relaxed weak heaps, decrease and insert require $O(1)$ worst-case time using at most two element comparisons per operation, and delete-min requires $O(\lg n)$ worst-case time using at most $2\lceil \lg n \rceil$ element comparisons.*

For rank-relaxed weak heaps, the bounds on the number of element comparisons can be further reduced with the following improvement in *delete-min*.

- If $\lambda < \frac{1}{2}\lceil \lg n \rceil$, perform *delete-min* as above.
- If $\lambda \geq \frac{1}{2}\lceil \lg n \rceil$, use the transformations to remove all the existing markings. This is done bottom up; starting with the mark at the lowest level, we repeatedly lift it up using the parent transformation until this mark meets the first mark at a higher level. We then apply either the sibling or the pair transformation to remove the two markings and introduce one mark at the next higher level. These actions are repeated until all the markings are removed. We then proceed with the *delete-min* operation as above, while noting that a minimum element is now at the root.

**Theorem 4** *For rank-relaxed weak heaps, starting with an empty structure, the execution of any sequence of $n$ insert, $m$ decrease, and $n$ delete-min operations requires $O(m + n \lg n)$ time using at most $2m + 1.5n \lg n$ element comparisons.*

**Proof** The total number of markings created by $m$ *decrease* and $n$ *insert* operations is $m + n$ (one marking per operation); no other operation will increase this number. Since every $\lambda$-reduction gets rid of at least one marking, the total number of reductions is at most $m + n$. Since each $\lambda$-reduction performs at most two element comparisons, the total number of element comparisons involved is at most $2m + 2n$.

Consider the *delete-min* operation. The number of involved joins, and accordingly the number of element comparisons, performed to restore the weak-heap ordering is at most $\lceil \lg n \rceil - 1$. For the first case, when $\lambda < \frac{1}{2}\lceil \lg n \rceil$, the number of element comparisons involved in *find-min* is less than $\frac{1}{2}\lceil \lg n \rceil$. Then, the number of element comparisons charged for such case is at most $1.5\lceil \lg n \rceil - 1$. For the second case, when $\lambda \geq \frac{1}{2}\lceil \lg n \rceil$, the number of levels that have no markings is at most $\frac{1}{2}\lceil \lg n \rceil$. Our bottom-up procedure executes at most one parent transformation for each of these levels to lift a making one level up. Since a parent transformation performs one element comparison, the total number of element comparisons involved in such transformations (which do not reduce the number of marked nodes) is at most $\frac{1}{2}\lceil \lg n \rceil$. After the mark removals, as the root now has the minimum value, there are no element comparisons involved in *find-min*. Then, the number of element comparisons charged for such case is also at most $1.5\lceil \lg n \rceil - 1$.

When considering the $n$ *delete-min* operations, the ceiling in the bound can be dropped by paying attention to the shrinking number of elements in the repeated *delete-min* operations. For these, the worst-case scenario occurs with weak heaps of size $n, n-1, \ldots, 1$; in this scenario $n$ *insert* operations are followed by $n$ *delete-min* operations (Dutton 1993). Hence, the total number of element comparisons charged for the *delete-min* operations is at most $1.5(\sum_{i=1}^{n} \lceil \lg i \rceil) - n < 1.5n \lg n - 2n$. Here we use the inequality $\sum_{i=1}^{n} \lceil \lg i \rceil \leq n \lg n - 0.914n$ (Edelkamp and Stiegeler 2002). Together with the $2m + 2n$ element comparisons required by the $\lambda$-reductions, we perform a total of at most $2m + 1.5n \lg n$ element comparisons for the whole sequence of operations. □

## 5 Experiments

To verify the practical relevance of our theoretical findings, we implemented seven data structures: weak heap, weak queue, run-relaxed weak heap, run-relaxed weak queue, rank-relaxed weak heap, rank-relaxed weak queue, and violation heap. All our implementations have been made part of the CPH STL (`www.cphstl.dk`). For comparison purposes, we also considered two implementations from LEDA (Mehlhorn and Näher 1999): Fibonacci heap and pairing heap; and four other from the CPH STL: array-based binary heap, array-based weak heap, Fibonacci heap, and pairing heap. In this section, we report the results of the experiments carried out for these data structures. We performed two types of experiments. First, we looked at operation sequences encountered when computing shortest paths. Second, we looked at synthetic operation sequences.

The priority queues available at the CPH STL, old and new, support the interface of a meldable priority queue. To avoid redundant code, the design is based on the bridge design pattern: The interface provided for the users has many convenience functions, whereas the realizators—the data structures—just provide the key functionality. In their basic form, for all realizators, the evaluation of *find-min* can take logarithmic worst-case time. However, if necessary, the realizators can be decorated to support *find-min* in $O(1)$ worst-case time. In the experiments reported here, slow *find-min* was used; so, in *delete-min* it was necessary to find the minimum before it could be extracted.

The library interface of a meldable priority queue is parameterized with: the type of elements, a comparison function, an allocator (with the standard allocator as default), a realizator, and iterators used to access the elements stored by the realizator. In addition, realizators accept more specialized type parameters like: the type of the root registry, the mark registry, and some other policies. The type parameters enable the compiler to inline code and facilitate policy-based benchmarking (Bruun et al. 2010).

Here is another note about our competitors:

**Array-based binary heap** (Williams 1964)**.** The CPH STL implementation employs a bottom-up heapifier, and stores elements indirectly using handles to retain referential integrity. That is, each array entry stores a pointer to an element and from that element there is a reference back to the entry.

**Array-based weak heap** (Dutton 1993)**.** As above, elements are stored indirectly. This implementation was one of the fastest implementations considered in (Bruun et al. 2010).

**Fibonacci heap** (Fredman and Tarjan 1987)**.** The CPH STL implementation is lazy: *insert*, *decrease*, and *delete* operations only add nodes to the root list and leave all the work for the forthcoming *find-min* operations, which consolidate roots of the same rank.

**Pairing heap** (Fredman et al. 1986)**.** The CPH STL implementation is a reconstruction of the no-auxiliary-buffer two-pass approach used in LEDA.

**2-3 heap** (Takaoka 2003)**.** Compared to the original proposal, the CPH STL implementation includes a few modifications to support the CPH STL interface.

**Violation heap** (Elmasry 2010)**.** We used a root registry that relies on a singly-linked list. To support three-way join operations, we used a bit-vector to indicate the ranks that have more than two roots.

When developing the programs for the experiments, we used the code written for the experiments reported in (Bruun et al. 2010) as the starting point. Our goal was to simplify the code base in order to make maintenance simpler. Therefore, we made a complete rewrite of the code base. In particular, we aimed that the data structures use the same set of transformations. We built two component frameworks, one to realize all weak-queue variants and another to realize all weak-heap variants. The frameworks can be characterized as follows:

**Weak-queue framework.** The root registry uses the numeral-system approach described, for example, in (Elmasry et al. 2008a). Its advantage is that no collision lists are necessary because at each rank there are at most two roots. Three mark registries were written: the naive registry that avoids markings by repeatedly visiting distinguished ancestors (weak queue), the lazy registry described in Section 3.2 (run-relaxed weak queue), and the eager registry described in Section 3.3 (rank-relaxed weak queue).

Table 1: Approximative LOC counts for various priority-queue realizators in the CPH STL. (Comments, assertions, debugging aids, and lines with a single parenthesis are ignored, and long statements are counted as single lines.)

| Realizator | LOC |
|---|---|
| weak heap | 565 |
| weak queue | 513 |
| run-relaxed weak heap | 1 021 |
| run-relaxed weak queue | 964 |
| rank-relaxed weak heap | 883 |
| rank-relaxed weak queue | 826 |
| array-based binary heap | 205 |
| array-based weak heap | 214 |
| Fibonacci heap | 296 |
| pairing heap | 204 |
| 2-3 heap | 546 |
| violation heap | 459 |

**Weak-heap framework.** The leaf registry was implemented as described in Section 4. The mark registries were reused: the naive registry (weak heap), the lazy registry (run-relaxed weak heap), and the eager registry (rank-relaxed weak heap).

The nodes used by all priority queues reserved space for three pointers pointing to the parent, left child, and right child, respectively. Depending on the priority queue in use, additional information (like rank and depth) was associated with each node. To speed up the grouping of nodes at the same level, we associated a static array of size 64 (we assumed $n < 2^{64}$) for the advanced mark-registry implementations. To find the first occupied entry in this array, we maintained an additional bit-array in a single word and exploited the computation of the most significant 1-bit (via the built-in leading-zero-count command).

All experiments were performed on one core of a desktop computer (model Intel i/7 CPU 2.67 GHz) running Ubuntu 10.10 (Linux kernel 2.6.28-11-generic). This computer had 32 KB L1 cache, 256 KB L2 cache, 8 MB (shared) L3 cache, and 12 GB main memory. All programs, written in C++, were compiled using GNU C++ compiler (`gcc` version 4.3.3) with options `-Wall -O3 -fno-strict-aliasing -lleda`.

The length of a program says something about its complexity, even though this metric does not exactly capture the intellectual challenge of creating the programming artifact in hand. To give a big picture of the code complexity of the different priority-queue implementations, we list the LOC counts for many of the data structures available in the CPH STL in Table 1. For most priority queues four pieces—node, root registry, mark registry, and priority-queue engine—are clearly visible in the code. We report the total amount of code needed for implementing these central pieces in each data structure. There are pieces that are shared by the components (like the transformations), but their LOC counts are included in the total amounts for all data structures that use them.

### 5.1 Shortest-Paths Computation

We measured the number of distance comparisons performed and the CPU time consumed for the single-source shortest-paths computation. We chose Dijkstra's algorithm as a good fit for our setting since, for a graph of $n$ vertices and $m$ edges, at most $n$ *insert* and *delete-min* operations and $m$ *decrease* operations are executed. For generating random graphs of $n$ vertices and $m$ edges, we used the graph gener-

Table 2: Effect of the graph representation on the running time of Dijkstra's algorithm when $n = 2^k$ and $m = 2n \lg n$ (PH: pairing heap, FH: Fibonacci heap, $r$: LEDA random graph $c$: LEDA compact graph, $e$: engineered graph); each test result (time/$(n \lg n + m)$ [ns]) is the average of 10 repetitions.

| $k$ | $\mathrm{PH}_r$ | $\mathrm{PH}_c$ | $\mathrm{PH}_e$ | $\mathrm{FH}_r$ | $\mathrm{FH}_c$ | $\mathrm{FH}_e$ |
|---|---|---|---|---|---|---|
| 13 | 42.4 | 47.7 | 15.9 | 47.7 | 53.0 | 26.5 |
| 14 | 41.4 | 43.5 | 17.4 | 52.3 | 52.3 | 30.5 |
| 15 | 54.7 | 54.7 | 19.9 | 64.1 | 65.2 | 29.4 |
| 16 | 70.6 | 71.7 | 23.9 | 83.9 | 83.9 | 33.5 |
| 17 | 81.2 | 81.9 | 27.1 | 93.7 | 94.2 | 36.3 |
| 18 | 96.0 | 96.4 | 35.4 | 109.7 | 110.4 | 46.1 |
| 19 | 108.2 | 108.8 | 43.8 | 124.0 | 124.4 | 55.6 |
| 20 | 111.5 | 111.8 | 47.6 | 126.8 | 127.3 | 60.1 |

ators from LEDA (version 6.2). Edge weights were integers drawn randomly from the range $[1..m]$ and converted to double-precision floating-point numbers.

We modified LEDA's code for Dijkstra's algorithm as follows: 1) We computed the shortest-path distances from the source to all other vertices, not the shortest paths themselves. 2) We maintained the state of each vertex; a vertex could be *scanned*, *labelled*, or *unlabelled*. 3) We ignored all back edges, i.e. edges to a scanned vertex, from further consideration when updating the tentative distances. 4) We inserted a vertex into the priority queue when its state changed from unlabelled to labelled.

In our experiments, violation heaps and 2-3 trees showed good performance. However, these structures were beaten by other competitors for both the running time and the number of distance comparisons. The priority queues from LEDA turned out to be slower than their CPH-STL counterparts, and hence they were excluded from further consideration. Also, the array-based weak-heap implementation was outperformed by a small margin by the new pointer-based weak-heap implementation and was accordingly excluded from further consideration. The explanation is that, for referential integrity, the array-based implementations have to use indirection and rely on pointers, too. We also excluded run-relaxed weak heaps and run-relaxed weak queues as they provide worst-case guarantees which was not relevant for the present application. Moreover, the rank-relaxed variants were always better and are a good replacement.

As a standard, a graph is represented using adjacency lists. However, as pointed out by the developers of LEDA (Mehlhorn and Näher 1999) and others, this may lead to a poor cache behaviour. Therefore, LEDA offers a more compact graph representation based on adjacency arrays. To make further application engineering possible, we implemented our own graph data structure based on adjacency arrays. In our engineered version, each edge stores its endpoints and its length. This information is kept compactly in an array while storing all edges outgoing from the same vertex consecutively in memory. Moreover, the graph and the priority queue use the same set of nodes. Each vertex stores its tentative distance, its state, and a pointer to the first of its outgoing edges. For example, in the case when the underlying priority queue is a Fibonacci heap, each node also stores a degree, a mark, and four pointers to two siblings, the parent, and a child. If the edge lengths are double-precision floating-point numbers taking two words each, and if each pointer takes one word, for a graph of $n$ vertices and $m$ edges, the data structures would require $4m + 8n + O(1)$ words.

Table 2 shows that, when this engineered graph representation was used with Fibonacci or pairing heaps in the implementation of Dijkstra's algorithm, the resulting implementation was about a factor of 2-3 faster compared to that relying on the graph structures of LEDA. The main problem was the interconnection between the two data structures. Somehow it was necessary to recall for each vertex its location inside the priority queue and to indicate for each priority element the corresponding vertex in the graph; this required space and indirect memory accesses. Also, because of the tight coupling of the two structures, we could avoid all dynamic memory management. The memory for the graph was allocated from the stack, and the priority queues could reuse the same nodes. Because of these advantages, we used this graph representation in further experiments.

The results of the experiments on Dijkstra's algorithm are given in Figures 2–4. We varied the edge density of the input graphs from $m = 4n$, to $m = 2n \lg n$, up to $m = n^{3/2}$.

For all graphs, the number of distance comparisons was smallest for weak heaps. In general, the weak-queue variants performed more distance comparisons than the weak-heap variants, while the rank-relaxed versions performed better than the run-relaxed ones. As to the CPU time, binary heaps are mostly the winners followed by pairing heaps, while there is a visible advantage of weak queues compared to weak heaps. For both performance measures, relaxed variants are often worse than their non-relaxed counterparts.

The case of dense graphs seems intriguing. As $m$ is much larger than $n$, the manipulation of the graph is the dominating factor. The results for the number of distance comparisons indicate that this value con-

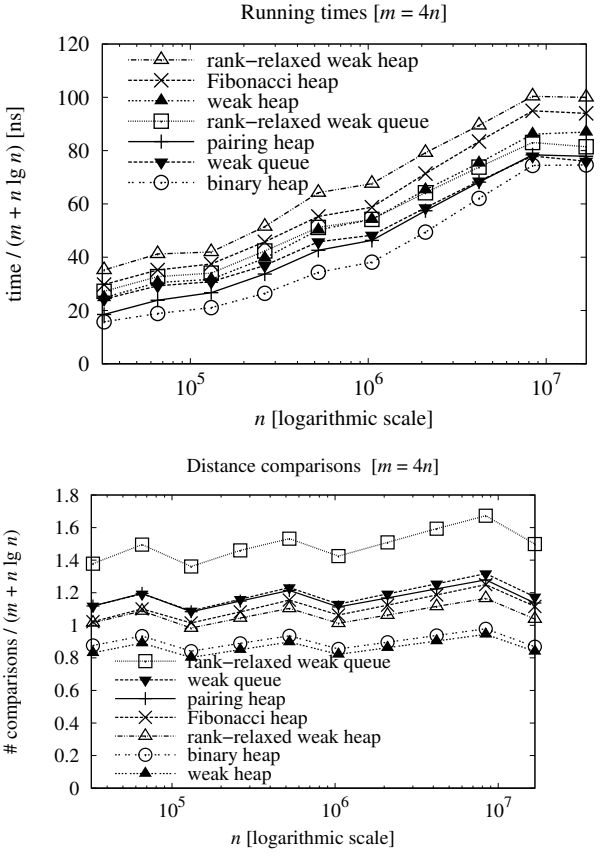


Figure 2: Performance of Dijkstra's algorithm on graphs with $m = 4n$ for different priority queues.

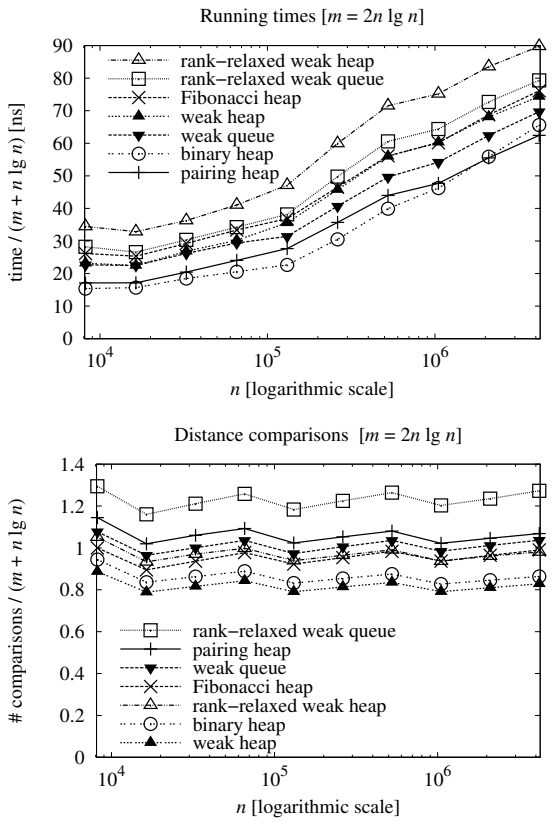


Figure 3: Performance of Dijkstra's algorithm on graphs with $m = 2n \lg n$ for different priority queues.

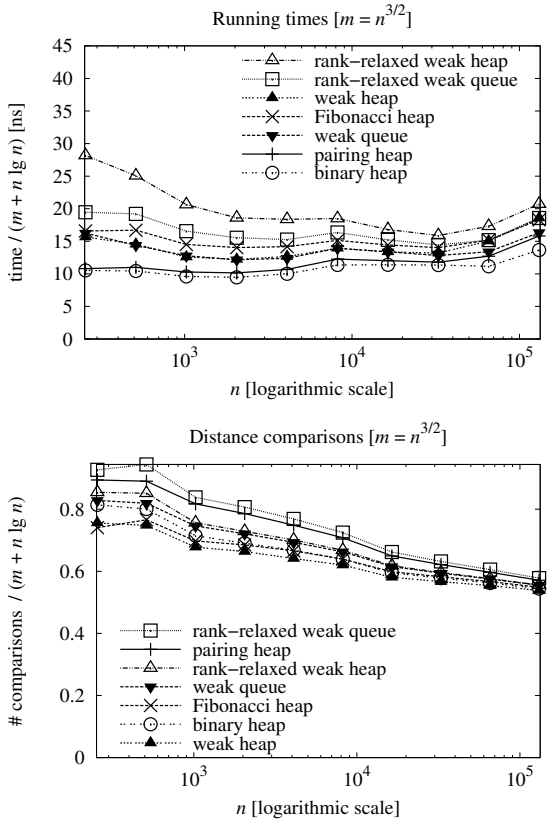


Figure 4: Performance of Dijkstra's algorithm on graphs with $m = n^{1.5}$ for different priority queues.

Table 3: Performance of different priority queues in our synthetic tests.

| | #Comparisons | | Time [ns] | |
|---|---|---|---|---|
| *insert*/ $n$ | $10^6$ | $10^7$ | $10^6$ | $10^7$ |
| array-based binary heap | 17.95 | 21.32 | 163 | 188 |
| weak heap | 10.06 | 11.87 | 273 | 332 |
| weak queue | 0.99 | 0.99 | 86 | 79 |
| run-relaxed weak heap | 1.84 | 1.84 | 320 | 328 |
| run-relaxed weak queue | 0.99 | 0.99 | 76 | 78 |
| rank-relaxed weak heap | 1.47 | 1.49 | 166 | 173 |
| rank-relaxed weak queue | 0.99 | 0.99 | 76 | 75 |
| *decrease*/ $n$ | $10^6$ | $10^7$ | $10^6$ | $10^7$ |
| array-based binary heap | 18.95 | 22.32 | 831 | 1 832 |
| weak heap | 10.90 | 12.85 | 1 006 | 1 686 |
| weak queue | 9.85 | 11.34 | 1 836 | 3 482 |
| run-relaxed weak heap | 1.99 | 1.99 | 570 | 606 |
| run-relaxed weak queue | 1.99 | 1.99 | 413 | 526 |
| rank-relaxed weak heap | 1.99 | 1.99 | 413 | 527 |
| rank-relaxed weak queue | 1.99 | 1.99 | 280 | 381 |
| *delete-min* / $n \lg n$ | $10^6$ | $10^7$ | $10^6$ | $10^7$ |
| array-based binary heap | 0.98 | 0.95 | 46 | 80 |
| weak heap | 0.94 | 0.92 | 57 | 80 |
| weak queue | 1.31 | 1.34 | 53 | 76 |
| run-relaxed weak heap | 1.09 | 1.19 | 54 | 76 |
| run-relaxed weak queue | 1.31 | 1.34 | 52 | 73 |
| rank-relaxed weak heap | 0.94 | 1.30 | 61 | 88 |
| rank-relaxed weak queue | 1.31 | 1.34 | 53 | 74 |

verges to about 1/2 distance comparison per edge; see Figure 4. This is justified as performing (on average) one distance comparison per forward edge, and none for back edges. In other words, distance comparisons are done to check whether to execute a *decrease* or not; after this check, very few distance comparisons are performed by the *decrease* operations.

### 5.2 Synthetic Operation Sequences

If the edge weights are uniformly-distributed random numbers, the expected number of *decrease* operations executed in Dijkstra's algorithm is $O(n \lg(m/n))$ (Noshita 1985). In addition, for this type of input, the cost per *decrease* is small for non-relaxed structures like binary and weak heaps. In accordance, we conducted additional experiments to measure the worst-case performance of individual operations.

*insert* **test.** Start with an empty data structure, perform $n$ *insert* operations. The elements were given in decreasing order.

*decrease* **test.** After $n$ *insert* operations, perform $n$ *decrease* operations. The elements were inserted in random order. The value of each element was decreased once to a value smaller than the active minimum, and *decrease* operations were performed in decreasing order according to the element values.

*delete-min* **test.** After $n$ *insert* operations, perform $n$ *delete-min* operations. The elements were inserted in random order.

In the last two tests, the cost of building the structures was not included in the test results. Table 3 provides the number of element comparisons and the observed CPU times when $n = 10^6$ and $10^7$.

Consider the number of element comparisons performed. Except for weak heaps and binary heaps, other structures performed well for *insert*. All the non-relaxed structures showed logarithmic behaviour for *decrease*. For both tests, all relaxed structures performed at most two element comparisons per operation. For *delete-min*, the performance of weak heaps was the best and close to the optimum.

Consider the CPU time consumed. The weak-queue variants are superior to the weak-heap variants for *insert*. For *decrease*, though the relaxed structures are superior to the non-relaxed ones, the overhead due to the $\lambda$-reduction transformations is still visible. On the other hand, the performance of *delete-min* is quite similar for all structures.

## 6 Conclusion

We were mainly interested in the comparison complexity of priority-queue operations when handling an operation sequence that appears in typical applications. Introducing relaxed weak heaps, we showed how to perform a sequence of $n$ *insert*, $m$ *decrease*, and $n$ *delete-min* operations using at most $2m + 1.5n \lg n$ element comparisons. For all other known data structures, the proved bounds are higher. In particular, Fibonacci heaps are not optimal with respect to the number of element comparisons performed.

Our experiments indicated that this improvement is primarily analytical. In the test scenarios considered by us, for many priority queues, the number of element comparisons performed was observed to match, or to be better than, the number achieved by relaxed weak heaps. We once more note that there is a gap between the theoretical worst-case bounds and the actual performance encountered in practice. One could ask whether the worst-case analysis of some of the other structures could be improved as well.

Although it is possible to achieve $n \lg n + 2n \lg \lg n + O(m+n)$ element comparisons for the operation sequence considered (Elmasry et al. 2008b), the constant in the $O(\cdot)$ term for $m$ and $n$ is bigger than 2 (about 5). A theoretical question arises whether it is possible or not to achieve a bound of $2m + n \lg n + o(n \lg n)$ element comparisons for our reference sequence of operations.

## References


Brown, M. R. (1978), 'Implementation and analysis of binomial queue algorithms', *SIAM Journal on Computing* **7**(3), 298–319.

Bruun, A., Edelkamp, S., Katajainen, J. & Rasmussen, J. (2010), Policy-based benchmarking of weak heaps and their relatives. *in* '9th International Symposium on Experimental Algorithms', Vol. 6049, LNCS, Springer-Verlag, pp. 424–435.

Chan, T. M. (2009), Quake heaps: A simple alternative to Fibonacci heaps, Unpublished manuscript.

Cho, S. & Sahni, S. (1998), 'Weight-biased leftist trees and modified skip lists', *ACM Journal of Experimental Algorithmics* **3**(2), Article 2.

Crane, C. A. (1972), Linear lists and priority queues as balanced binary trees, Ph.D. Thesis, Computer Science Department, Stanford University.

Dijkstra, E. W. (1959), 'A note on two problems in connexion with graphs', *Numerische Mathematik* **1**, 269–271.

Driscoll, J. R., Gabow, H. N., Shrairman, R. & Tarjan, R. E. (1988), 'Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation', *Communications of the ACM* **31**(11), 1343–1354.

Dutton, R. D. (1993), 'Weak-heap sort', *BIT* **33**(3), 372–381.

Edelkamp, S. & Stiegeler, P. (2002), 'Implementing Heapsort with $n \log n - 0.9n$ and Quicksort with $n \log n + 0.2n$ comparisons', *ACM Journal of Experimental Algorithmics* **7**, Article 5.

Edelkamp, S. & Wegener, I. (2000), On the performance of Weak-Heapsort, *in* '17th Annual Symposium on Theoretical Aspects of Computer Science', Vol. 1770, LNCS, Springer-Verlag, pp. 254–266.

Edelkamp, S. (2009), Rank-relaxed weak queues: Faster than pairing and Fibonacci heaps?, Technical Report 54, Faculty 3—Mathematics and Computer Science, University of Bremen.

Elmasry, A., Jensen, C. & Katajainen, J. (2005), Relaxed weak queues: An alternative to run-relaxed heaps, CPH STL Report 2005-2, Department of Computer Science, University of Copenhagen.

Elmasry, A., Jensen, C. & Katajainen, J. (2008), 'Multipartite priority queues', *ACM Transactions on Algorithms* **5**(1), Article 14.

Elmasry, A., Jensen, C. & Katajainen, J. (2008), 'Two-tier relaxed heaps', *Acta Informatica* **45**(3), 193–210.

Elmasry, A. (2010), 'The violation heap: A relaxed Fibonacci-like heap', *Discrete Mathematics, Algorithms and Applications* **2**(4), 493–503.

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

Fredman, M. L., Sedgewick, R., Sleator, D. D. & Tarjan, R. E. (1986), 'The pairing heap: A new form of self-adjusting heap', *Algorithmica* **1**(1), 111–129.

Fredman, M. L. (1999), 'On the efficiency of pairing heaps and related data structures', *Journal of the ACM* **46**(4), 473–501.

Haeupler, B., Sen, S. & Tarjan, R. E. (2009), Rank-pairing heaps, *in* '17th Annual European Symposium on Algorithms', Vol. 5757, LNCS, Springer-Verlag, pp. 659–670.

Kaplan, H. & Tarjan, R. E. (2008), 'Thin heaps, thick heaps, *ACM Transactions on Algorithms* **4**(1), Article 3.

Mehlhorn, K. & Näher, S. (1999), *The LEDA Platform of Combinatorial and Geometric Computing*, Cambridge University Press.

Noshita, K. (1985), 'A theorem on the expected complexity of Dijkstra's shortest path algorithm', *Journal of Algorithms* **6**(3), 400–408.

Prim, R. C. (1957), 'Shortest connection networks and some generalizations', *Bell System Technical Journal* **36**, 1389–1401.

Stasko, J. T. & Vitter, J. S. (1987), 'Pairing heaps: Experiments and analysis', *Communications of the ACM* **30**(3), 234–249.

Takaoka, T. (2003), 'Theory of 2-3 heaps', *Discrete Applied Mathematics* **126**(1), 115–128.

Vuillemin, J. (1978), 'A data structure for manipulating priority queues', *Communications of the ACM* **21**(4), 309–315.

Williams, J. W. J. (1964), 'Algorithm 232: Heapsort', *Communications of the ACM* **7**(6), 347–348.