# Policy-Based Benchmarking of Weak Heaps and Their Relatives

Asger Bruun[1], Stefan Edelkamp[2,⋆], Jyrki Katajainen[1,⋆⋆], and Jens Rasmussen[1]

[1] Department of Computer Science, University of Copenhagen,
Universitetsparken 1, 2100 Copenhagen East, Denmark
[2] TZI, Universität Bremen,
Am Fallturm 1, 28357 Bremen, Germany

**Abstract.** In this paper we describe an experimental study where we evaluated the practical efficiency of three worst-case efficient priority queues: 1) a weak heap that is a binary tree fulfilling half-heap ordering, 2) a weak queue that is a forest of perfect weak heaps, and 3) a run-relaxed weak queue that extends a weak queue by allowing some nodes to violate half-heap ordering. All these structures support *delete* and *delete-min* in logarithmic worst-case time. A weak heap supports *insert* and *decrease* in logarithmic worst-case time, whereas a weak queue reduces the worst-case running time of *insert* to $O(1)$, and a run-relaxed weak queue that of both *insert* and *decrease* to $O(1)$. As competitors to these structures, we considered a binary heap, a Fibonacci heap, and a pairing heap. Generic programming techniques were heavily used in the code development. For benchmarking purposes we developed several component frameworks that could be instantiated with different policies.

## 1 Introduction

In this paper, we study addressable priority queues which store dynamic collections of elements and support the operations *find-min*, *insert*, *decrease* (or *decrease-key*), *delete*, *delete-min*, and *meld*. For addressable priority queues, *delete* and *decrease* take a handle to an element as an argument, and *find-min* and *insert* return a handle. These handles must be kept valid even though elements are moved around inside the data structures.

The most prominent priority queues described in textbooks (see, e.g. [7, 25]) include binary heaps [31], binomial queues [29], Fibonacci heaps [16], and pairing heaps [15]. Of these, a Fibonacci heap is important for many applications since it supports *decrease* in $O(1)$ amortized time. Also, it supports the other priority-queue operations in optimal amortized bounds: *find-min*, *insert*, and *meld* in $O(1)$ time; and *delete* and *delete-min* in $O(\lg n)$ time, $n$ being the number of elements stored prior to the operation.

⋆ Research partially supported by DFG grant ED 74/8-1.
⋆⋆ Partially supported by the Danish Natural Science Research Council under contract 09-060411 (project "Generic programming—algorithms and tools").

| Framework | Structure | *delete* | *insert* | *decrease* |
|---|---|---|---|---|
| single heap | weak heap | $\lceil \lg n \rceil$ | $\lfloor \lg n \rfloor + 1$ | $\lceil \lg n \rceil$ |
| multiple heap | weak queue | $2 \lg n + O(1)$ | 2 | $\lfloor \lg n \rfloor$ |
| relaxed heap | run-relaxed weak queue | $3 \lg n + O(1)$ | 2 | 4 |

**Table 1.** Worst-case number of element comparisons performed by the most important operations on a weak heap and its variants (when *find-min* takes $O(1)$ worst-case time). Here $n$ denotes the size of the data structure just prior to an operation.

After the publication of Fibonacci heaps, two questions were addressed: 1) Can the same time bounds be achieved in the worst case? 2) Can the time bounds be achieved by a simpler data structure? The first question was settled by Brodal [3] in a practically unsatisfactorily manner since in his solution, when considering the number of element comparisons performed, the constant factor involved in the complexity of *delete-min* is much higher than $\lg n$ for all reasonable values of $n$ [18]. Relaxed heaps proposed by Driscoll et al. [9] are more practical, but they support *meld* in logarithmic worst-case time, which is suboptimal. The second question has been studied by several authors, but there does not seem to be an agreement, whether the question has been settled or not. For more information about this issue, consult any of the recent articles [6, 13, 17, 26] and the references mentioned therein.

The research reported in this paper is related to both of the foregoing questions. Our primary objective was to evaluate the usefulness of various implementation strategies when programming weak heaps [10] and their close relatives: weak queues and relaxed weak queues [11, 14]. Our secondary objective was to get a complete picture of the field and compare the performance of these structures to that of some well-known competitors: binary heaps [31], Fibonacci heaps [16], and pairing heaps [15, 28]. Of the studied data structures, a weak heap has the same asymptotic performance as a binary heap [31], a weak queue the same as a binomial queue [29], and a rank/run-relaxed weak queue the same as a rank/run-relaxed heap [9]. To get an insight of the performance characteristics of the studied data structures, in Table 1 we list the number of element comparisons performed by the most important operations.

To make the experimental comparison as fair as possible, we relied on policy-based design (see, for example, the book by Alexandrescu [2]). For similar priority queues, a separate component framework was developed. Three parameterized frameworks were written: 1) a single-heap framework that can realize a binary heap, relying on top-down or bottom-up heapifying, and a weak heap (Section 3); 2) a multiple-heap framework that can realize a weak queue and a binomial queue (Section 4); and 3) a relaxed-heap framework that can realize a run-relaxed and rank-relaxed weak queue (Section 5).

In a popular-scientific form, our results could be summarized as follows:

1) Read the masters! The original implementation of a binomial queue [29], in essence a weak queue, turned out to be one of the best performers mainly because of the focus put on implementation details in its description.

2) Priority queues that guarantee good performance in the worst-case setting have difficulties in competing against solutions that guarantee good performance in the amortized setting. Hence, worst-case efficient priority queues should only be used in applications where worst-case efficiency is essential.
3) Memory management is expensive. In our early code many unnecessary memory allocations were performed. Micro benchmarking revealed that memory management caused a significant performance slowdown.
4) In current computers, caching effects are significant. Memory-saving and bit-packing techniques turned out to be effective.
5) For most practical values of $n$, the difference between $\lg n$ and $O(1)$ is small! Often in the literature, in particular in theoretical papers, the significance of $O(1)$-time *insert* and *decrease* has been exaggerated. For heaps, for which *decrease* requires logarithmic time, the loop sifting up an element is extremely tight. Unless we make element comparisons noticeable expensive, it is difficult to come up with a faster solution.
6) For random data, the typical running time of *insert*, *decrease*, and *delete* (but not *delete-min*) is $O(1)$ for binary heaps, weak heaps, and weak queues. Hence, more advanced data structures can only beat these data structures for pathological input instances.
7) Generic component frameworks help algorithm engineers to carry out unbiased experiments. Changing policies helped us to tune the programs significantly while keeping the code base small.

## 2 Parameterized Design

The frameworks written for this study have been made part of the CPH STL [8]. In this section we give a brief overview of the overall design of our programs. As to the actual code, we refer to the CPH STL design documents [5, 20, 27].

When doing the implementation work, we followed the conventions set for the CPH STL project. For example, the application programming interface (API) for a meldable priority queue is specified in [19] (and corrected in [20]). All *containers*, as they are called in STL parlance, are interfaces that are decoupled from their actual implementations. These interfaces are designed to be user friendly, but to implement them only a smaller *realizator* is needed. There is a clear division of labour between the container and its realizator: 1) A client gives an element to the container, which allocates a node and puts the element into that node, and gives the node further to the realizator. When a realizator extracts a node, it gives the node back to the container which takes care of the deallocation of the node. 2) The container also provides (unidirectional) *iterators* to traverse through the elements. Iterators can also be used as handles to elements.

As an example, the single-heap framework is parameterized to accept seven type arguments: the type of the elements (or values) manipulated; the type of the comparator used in element comparisons; the type of the allocator providing an interface to allocate, construct, destroy, and deallocate objects; the type of the nodes (or encapsulators) used for storing the elements; the type of the heapifier

used when re-establishing heap order after an element update; the type of the resizable array used for storing the heap; and the type of the surrogate proxy used (by iterators) for referring to the realizator (this is needed for supporting *swap* in $O(1)$ worst-case time).

Our goal was to make the frameworks generic such that they only use the methods provided by the policies given as type parameters and make as few assumptions on their functionality as possible. The key point is that the implementation of priority-queue operations *find-min*, *insert*, *decrease*, *delete*, *delete-min*, and *meld* is exactly the same for all realizators that a framework can create.

Our parameterized design has several advantages: a high level of code reuse, and ease of maintenance and benchmarking. By changing the parameters, one can easily see what the effect of a particular change is. The parameterized design also has its disadvantages: component frameworks can be difficult to understand, the design and development can be time consuming compared to quick-and-dirty programming, and sometimes generic programming can be difficult because of inadequate tool support. Moreover, a framework can become a hindrance for code optimizations, even though we did not experience any performance slowdown because of this type of design. However, we did experience that sometimes it was a challenge to make a change to a framework; this required that the programmer knew many data structures well and understood consequences of the change.

## 3 Single-Heap Framework

Recall that in a *heap-ordered tree* the element stored at a node is no smaller than the element stored at the parent of that node. The main difference between a binary heap [31] and a weak heap [10] is that the latter is only partially ordered. A *weak heap* has the following properties: 1) The element stored at a node is smaller than or equal to any element stored in the right subtree of that node (half-heap ordering). 2) The root of the entire structure has no left child. 3) The right subtree of the root is a complete binary tree (in the meaning defined in [22, Section 2.3.4.5]). In a *perfect weak heap*, the right subtree of the root is a perfect binary tree (i.e. a complete binary tree where even the last level is full).

A weak heap of size $n$ has a clever array embedding that utilizes $n$ auxiliary bits $r_i$, $i \in \{0, \ldots, n-1\}$. For location $i$, the left child is found at location $2i + r_i$ and the right child at location $2i + 1 - r_i$. For this purpose, $r_i$ is interpreted as an integer in the range $\{0, 1\}$, initialized to 0. By flipping the bit, the status of being a left or a right child can be exchanged, which is an essential property to join two weak heaps in $O(1)$ worst-case time. It is possible to construct a weak heap of size $n$ using $n - 1$ element comparisons, while for weak-heapsort the number of element comparisons performed is at most $n \log n + 0.09n$ [12], a value remarkably close to the lower bound of $n \log n - 1.44n$ element comparisons required by any sorting algorithm [23, Section 5.3.1].

Similar to binary heaps, array-embedded weak heaps can be extended to work as priority queues. For *delete-min*, after exchanging the element stored at the root with that stored at the last location, half-heap ordering is restored

bottom-up, joining the weak heaps that lie on the left spine of the subtree rooted at the right child of the root. For *insert*, we sift up the element until half-heap ordering is re-established. Similarly, for *decrease*, we start at the node that has changed its value, and propagate the change upwards. For *delete* we move the element at the last location into the place of the deleted element, and sift that element down as in *delete-min* and up as in *decrease*.

*Framework engineering.* Instead of using a linked representation, we decided to use a resizable array. To keep the iterators valid at all times, we store the elements indirectly and maintain pointers between the array and the elements as proposed in [7, Chapter 6]. This does not destroy the worst-case complexity, since for a resizable array the worst-case running time of the grow and shrinkage operations can be kept constant (see, for example, [21]). By accepting different heapifier policies, which provide methods for sifting down and up, we can easily switch between weak heaps and different implementations of binary heaps. For example, one may use an alternative bottom-up sift-down strategy (see [23, Section 5.2.3, Exercise 18] or [30]).

## 4 Multiple-Heap Framework

The key idea behind an improved worst case of *insert* is to maintain a sequence of perfect weak heaps instead of keeping all elements in a single heap. As this makes relocation of subheaps frequent, we rely on a pointer-based representation. Recall that a *binomial queue* is a collection of heap-ordered binomial trees [29], and that a *binomial tree* is a multiary tree that stores $2^h$ elements for some integer $h \geq 0$. A *weak queue* is just like a binomial queue, but each multiary tree is transformed into a binary tree by applying the standard child-sibling transformation (see, e.g. [22, Section 2.3.2]). A binary-tree variant of a binomial tree was already utilized by Vuillemin [29] in his tuned implementation of binomial queues, even though he did not give any name for the data structure.

For brevity, we call the perfect weak heaps maintained just heaps. Furthermore, we call the data structure used to keep track of the heaps a *heap store*. The heaps are maintained in size order, starting from the smallest. The basic operations to be supported include *inject* which inserts a new heap into the heap store, and *eject* which removes the smallest heap from the heap store. For *inject* it is essential that the size of the new heap is no greater than that of the smallest heap currently in a weak queue.

After *delete-min*, when determining the root that contains the new minimum, we have to iterate over the heaps. Therefore, the number of heaps has to be kept low; the worst-case minimum for the number of heaps is $\lfloor \lg n \rfloor + 1$. That is, when new heaps arrive, occasional joins are necessary. A simple way to maintain a heap store is to utilize the connection to binary numbers: If a weak queue stores $n$ elements and the binary representation of $n$ is $\langle b_0, b_1, \ldots, b_{\lfloor \lg n \rfloor} \rangle$, the heap store contains a heap of size $2^i$ if and only if $b_i = 1$. The main problem with this invariant is that sometimes *inject* has to perform a logarithmic number of joins, each taking $O(1)$ worst-case time.

There are several alternative strategies to implement the heap store such that both *inject* and *eject* take $O(1)$ time in the worst case still keeping the number of heaps logarithmic. One of the simplest approaches, mentioned already in [4], is to rely on redundant numbers. If $d_i$ denotes the number of heaps of height $i$ and $d_i \in \{0, 1, 2\}$, the heap store can keep the *cardinality sequence* $\langle d_0, d_1, \ldots, d_{\lfloor \lg n \rfloor} \rangle$ *regular*, i.e. in the form $\big(0 \mid 1 \mid 01^*2\big)^*$ using the normal syntax for regular expressions (see, for example, [1]). If after each *inject* the first two heaps of the same size are joined, the regularity of the cardinality sequence will be preserved and the number of heaps will never be larger than $\lfloor \lg n \rfloor + 1$ [14]. For *eject*, the smallest heap is extracted and the cardinality sequence is updated accordingly.

*Framework engineering.* There were several alternative ways of implementing the nodes. In our baseline version, every node stores an element and pointers to its left child, right child, and parent. To make swapping of nodes cheaper, we also tried variants where elements were stored indirectly, but these versions turned out to be slower than the baseline version. In an extreme case, only two pointers per node would be necessary to cover the parent-child relationships, as observed by Brown [4], but this space optimization did not pay off; the space optimized versions were considerably slower than the baseline version.

The framework supports two types of heap stores: one that maintains a proxy for each heap and keeps these proxies in a linked list, and another that maintains the roots in a linked list by reusing the pointers at the nodes. The latter idea goes back to Vuillemin [29]. Also, following Vuillemin's original proposal the heights of the heaps are maintained in a bit vector, which can be stored in a single word, since the heights are between 0 and $\lfloor \lg n \rfloor + 1$. Both types of heap stores could be equipped with the binary number system or redundant binary number system. For the redundant system, different strategies for maintaining the information about the pairs of heaps having the same size were tried. The best alternative turned out to a preallocated stack storing pointers to the first member of each pair. In general, all solutions relying on dynamic storage management were noticeable slower than the versions that avoided it. Overall, the overhead incurred by the redundant system turned out to be negligible.

We observed that there was a huge difference in the typical running times for the two known ways of dealing with *delete*. Brown [4] called the two strategies top-down and bottom-up. The *top-down strategy* sifts up the node being deleted to the root and removes the root, whereas the *bottom-up strategy* finds a replacement node for the node being deleted, makes the replacement, and sifts down or up the new node. In a typical case, assuming that we are not deleting the minimum, the amount of work done by the bottom-up approach is $O(1)$, whereas the amount of work involved in the top-down approach is logarithmic.

## 5 Relaxed-Heap Framework

In relaxed weak queues the new ingredient is that the half-ordering violations incurred by *decrease* operations are resolved by marking. When there are too many marked nodes, the number of marked nodes is reduced. Driscoll et al. [9]

introduced this idea in their relaxed heaps, and Elmasry et al. [14] observed that the idea carries over into the binary-tree setting. The other operations *find-min*, *insert*, *delete*, *delete-min*, and *meld* can be implemented as for weak queues.

We call the data structure used to keep track of all markings a *mark store*. The fundamental operations to be supported include *mark* which marks a node to denote that a half-ordering violation may occur at that node, *unmark* which removes a marking, and *reduce* which removes one or more unspecified markings. A *run* is a maximal sequence of two or more marked nodes that are consecutive on the left spine of a subtree. 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*. If at some height there are more than one singleton, these singletons form a *team*. To summarize, the set of all nodes is divided into four disjoint categories: non-marked nodes, members, leaders, and singletons.

A pair $(type, height)$ with *type* being either non-marked, member, leader, or singleton; and *height* being a value in the range $\{0, 1, \ldots, \lfloor \lg n \rfloor\}$ denotes the *state* of a node. The states are stored explicitly at the nodes. Transformations used when reducing the number of marked nodes induce a constant number of state transitions. A simple example of such a transformation is a join, where the height of the new root is increased by one.

Other transformations (see Fig. 1) are cleaning, parent, sibling, and pair transformations. A *cleaning transformation* rotates a marked left child to a marked right one, provided that its sibling and parent are 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 two markings, while generating a new marking one level up. A *pair transformation* has a similar effect, but it operates on disconnected trees. These four transformations are combined to perform a *singleton* or *run transformation*.

In a *run-relaxed weak queue* [14], which is similar to a run-relaxed heap [9], an invariant is maintained that, after each priority-queue operation, the number of markings is never larger than $\lfloor \lg n \rfloor$. When this bound is exceeded, a singleton or a run transformation is applied to restore the invariant. The running time of *decrease* can be guaranteed to be $O(1)$ in the worst case. In a *rank-relaxed weak queue* [11], which is similar to a rank-relaxed heap [9], the transformations are applied in an eager way by performing as many *reduce* operations as possible after each priority-queue operation that introduces new markings. The worst-case cost of *decrease* can be logarithmic, but the amortized cost is a constant. From a practical perspective, amortization leads to a slightly more efficient implementation, as verified in [11].

*Framework engineering.* The first implementation of a mark store that supports *mark*, *unmark*, and *reduce* in $O(1)$ worst-case time was described in [9]. In this solution it was necessary to maintain a doubly-linked list of leaders, a doubly-linked list of teams, a doubly-linked list of singletons at each height, and a resizable array of pointers to the beginning of each singleton list. In our engi-

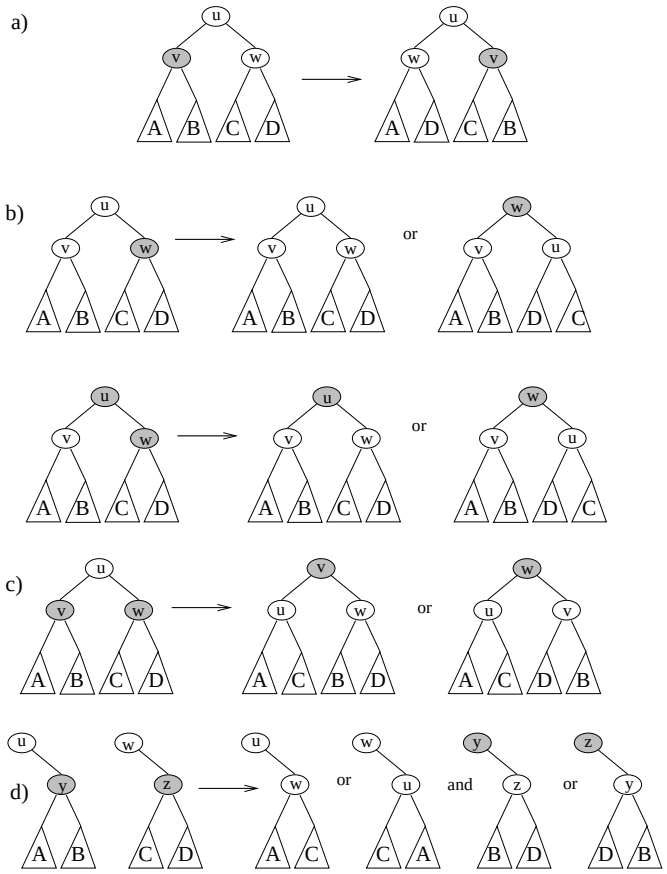


**Fig. 1.** Primitives used by *reduce*: a) cleaning transformation, b) parent transformation, c) sibling transformation, and d) pair transformation.

neered implementation we use no lists, but keep pointers to the marked nodes in a preallocated array and maintain another preallocated array of bit vectors, each occupying a single word, indicating which of the marked nodes are singletons of particular height. Additionally, we need one bit vector to denote which of the marked nodes are leaders and another to indicate which of the singleton sets have more than one node. To allow fast *unmark*, every marked node stores an index referring to the pointer array maintained in the mark store. The bit-vector class features the fast selection of the most significant 1-bit.

## 6 Experiments

In our benchmarks we compared priority queues from the weak-heap family (weak heap, weak queue, and run-relaxed weak queue) to their closest competitors (binary heap [31], Fibonacci heap [16], and pairing heap [28]). The last two

were taken from LEDA (version 6.2) [24]. The other data structures were the engineered versions that we implemented for the CPH STL [8].

We carried out several experiments for different types of input data (worst-case and randomly-generated instances) in different environments (compilers and computers varied) considering different kinds of performance indicators (number of element comparisons, clock cycles, and CPU time). The results obtained did not vary much across the tested environments. Also, the results obtained by the clock-cycle and CPU-time measurements were similar.

When engineering our implementations we carried out several micro-benchmarks. Due to space limitations, we do not report any detailed results on them, but refer to [5]. For randomly-generated data, the average running time of *insert*, *decrease*, and *delete* (but not *delete-min*) is $O(1)$ for binary and weak heaps. Since these structures are simple, other more advanced structures have difficulties in beating them. For the structures having good amortized time bounds, *insert* and *decrease* are fast because most work is delayed till *delete* and *delete-min*. Due to space limitations, we leave out the results for randomly-generated input data, but present them in the full version of this paper.

We find the results of synthetic benchmarks involving the basic operations interesting and report these results here. These benchmarks were conducted on a laptop computer (model Intel® Core$^{\text{TM}}$2 CPU T5600 @ 1.83GHz) under Ubuntu 9.10 operating system (Linux kernel 2.6.31-19-generic) using `g++` C++ compiler (`gcc` version 4.4.1 with options `-DNDEBUG -Wall -std=c++0x -pedantic -x c++ -fno-strict-aliasing -O3`). The size of L2 cache of this computer was about 2 MB and that of the main memory 1 GB. The input data was integers of type `long long` and, for the LEDA data structures, pairs of type (`long long`, `struct empty`) since in LEDA the elements are expected to be (priority, information) pairs. Besides comparing integer elements with their built-in comparison function, the comparator increased a global counter to gather comparison counts.

In order to avoid the problem caused by a bounded clock granularity, which in the test computer was 10 milliseconds, for given $n$ we repeated each experiment $\lceil 10^6/n \rceil$ times, each time with a new priority queue. The standard dual-loop strategy was used to eliminate the time taken by all initializations. All running times are reported in microseconds, and they are average times per operation.

In Fig. 2, 3, 4, and 5 we give the average running times used and the number of element comparisons performed per *insert*, *decrease*, *delete*, and *delete-min*, respectively. In the *insert* experiment, the integers between 0 and $n-1$ were inserted in reversed sorted order which forced binary and weak heaps to use logarithmic time for each operation. In spite of this, the running times were competitive. In the *decrease* experiment, the integers were inserted in random order, and thereafter the values were updated such that each new value became the new minimum element; the time used by *decrease* operations was measured. This arrangement guaranteed that *decrease* took logarithmic time on an average for a binary heap, weak heap, and weak queue. Even in such extreme situation, these three data structures were competitive against theoretically more robust solutions. In the *delete* experiment, the integers were inserted in random order

and extracted in their insertion order; the time used by *delete* operations was measured. All our implementations relied on the bottom-up deletion strategy; this experiment confirmed that this was a good choice. In the *delete-min* experiment, the integers were inserted in random order, and the minimum was extracted until the data structure became empty; the time used by *delete-min* operations was measured. Even if a weak heap is optimal with respect to the number of element comparisons, a weak queue was faster. For all problem sizes, a Fibonacci heap used about 3 times more time than a weak queue.

The average running times reported can be used to estimate the overhead caused by the worst-case behaviour. For data structures that do not provide good performance in the worst-case setting, the running times of individual operations can fluctuate considerably. For example, for binary and weak heaps the worst-case running time of a single *insert* was linear since we relied on `std::vector`, not on a worst-case efficient resizable array. For Fibonacci and pairing heaps the worst-case running time of a single *delete-min* is linear. Even for Vuillemin's implementation of a weak queue the running times of *insert* and *decrease* can vary between $\Theta(1)$ and $\Theta(\lg n)$. In applications, where such fluctuations are intolerable, only a run-relaxed weak queue can guarantee stable behaviour, but as shown, this stability has its price.

In particular for randomly-generated input data, the performance of simple data structures like binary and weak heaps is good. These simple data structures fall in short only when melding has to be efficient. Namely, for our implementations, melding two weak heaps (or binary heaps) of size $m$ and $n$, $m \leq n$, takes $\Theta(m \lg n)$ time in the worst case. Even though more efficient implementations are possible, one should consider using some of other data structures instead.

## References


1. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools*, 2nd Edition, Pearson Education, Inc. (2007).
2. A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley (2001).
3. G. S. Brodal, Worst-case efficient priority queues, *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM/SIAM (1996), 52–58.
4. M. R. Brown, Implementation and analysis of binomial queue algorithms, *SIAM Journal on Computing* **7**, 3 (1978), 298–319.
5. A. Bruun, Effektivitetsmåling på krydsninger af svage og binomiale prioritetskøer, CPH STL Report **2010-2**, Department of Computer Science, University of Copenhagen (2010).
6. T. M. Chan, Quake heaps: A simple alternative to Fibonacci heaps, Unpublished manuscript (2009).
7. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3th Edition, The MIT Press (2009).
8. Department of Computer Science, University of Copenhagen, The CPH STL, Website accessible at http://cphstl.dk/ (2000–2010).
9. 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**, 11 (1988), 1343–1354.

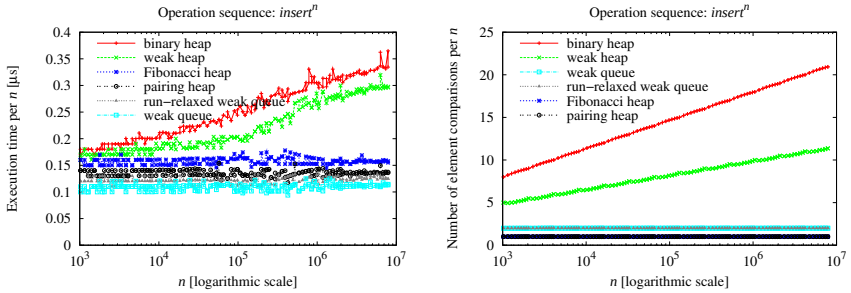


**Fig. 2.** *insert*: CPU times and comparison counts for different priority queues.

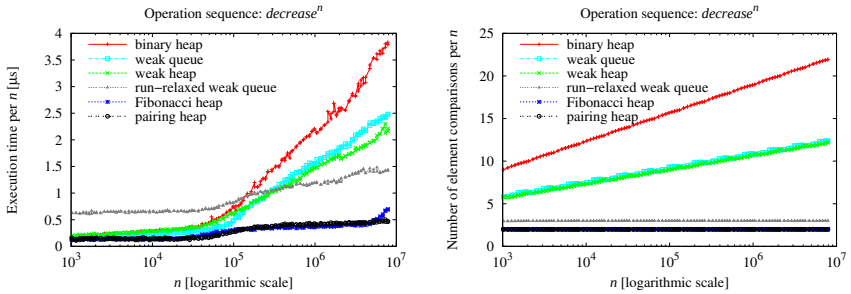


**Fig. 3.** *decrease*: CPU times and comparison counts for different priority queues.

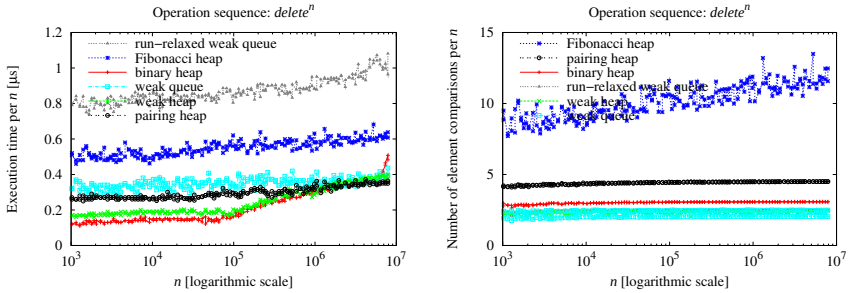


**Fig. 4.** *delete*: CPU times and comparison counts for different priority queues.

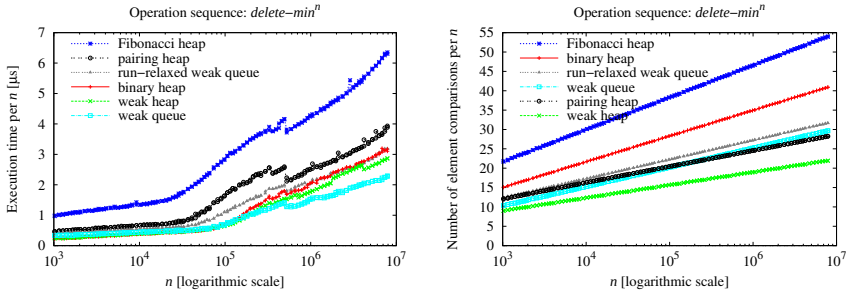


**Fig. 5.** *delete-min*: CPU times and comparison counts for different priority queues.

10. R. D. Dutton, Weak-heap sort, *BIT* **33**, 3 (1993), 372–381.
11. S. Edelkamp, Rank-relaxed weak queues: Faster than pairing and Fibonacci heaps?, Technical Report **54**, TZI, Universität Bremen (2009).
12. S. Edelkamp and I. Wegener, On the performance of Weak-Heapsort, *Proceedings of the 17th Annual Symposium on Theoretical Aspects of Computer Science*, *Lecture Notes in Computer Science* **1770**, Springer-Verlag (2000), 254–266.
13. A. Elmasry, Violation heaps: A better substitute for Fibonacci heaps, E-print **0812.2851v1**, arXiv.org (2008).
14. A. Elmasry, C. Jensen, and J. Katajainen, Relaxed weak queues: An alternative to run-relaxed heaps, CPH STL Report **2005-2**, Department of Computer Science, University of Copenhagen (2005).
15. M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan, The pairing heap: A new form of self-adjusting heap, *Algorithmica* **1**, 1 (1986), 111–129.
16. M. L. Fredman and R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *Journal of the ACM* **34**, 3 (1987), 596–615.
17. B. Haeupler, S. Sen, and R. E. Tarjan, Rank-pairing heaps, *Proceddings of the 17th Annual European Symposium on Algorithms*, *Lecture Notes in Computer Science* **5757**, Springer-Verlag (2009), 659–670.
18. C. Jensen, Private communication (2009).
19. J. Katajainen, Project proposal: A meldable, iterator-valid priority queue, CPH STL Report **2005-1**, Department of Computer Science, University of Copenhagen (2005).
20. J. Katajainen, Priority-queue frameworks: Programs, CPH STL Report **2009-7**, Department of Computer Science, University of Copenhagen (2009).
21. J. Katajainen and B. B. Mortensen, Experiences with the design and implementation of space-efficient deques, *Proceedings of the 5th International Workshop on Algorithm Engineering*, *Lecture Notes in Computer Science* **2141**, Springer-Verlag (2001), 39–50.
22. D. E. Knuth, *Fundamental Algorithms*, *The Art of Computer Programming* **1**, 3rd Edition, Addison Wesley Longman (1997).
23. D. E. Knuth, *Sorting and Searching*, *The Art of Computer Programming* **3**, 2nd Edition, Addison Wesley Longman (1998).
24. K. Mehlhorn and S. Näher, *The LEDA Platform of Combinatorial and Geometric Computing*, Cambridge University Press (1999).
25. K. Mehlhorn and P. Sanders, *Algorithms and Data Structures: The Basic Toolbox*, Springer-Verlag (2008).
26. R. Paredes, Graphs for metric space searching, Ph.D. Thesis, Department of Computer Science, University of Chile (2008).
27. J. Rasmussen, Implementing run-relaxed weak queues, CPH STL Report **2008-1**, Department of Computer Science, University of Copenhagen (2008).
28. J. T. Stasko and J. S. Vitter, Pairing heaps: Experiments and analysis, *Communications of the ACM* **30**, 3 (1987), 234–249.
29. J. Vuillemin, A data structure for manipulating priority queues, *Communications of the ACM* **21**, 4 (1978), 309–315.
30. I. Wegener, Bottom-up-Heapsort, a new variant of Heapsort beating, on an average, Quicksort (if $n$ is not very small), *Theoretical Computer Science* **118** (1993), 81–98.
31. J. W. J. Williams, Algorithm 232: Heapsort, *Communications of the ACM* **7**, 6 (1964), 347–348.