The Weak-Heap Data Structure: Variants and Applications[1]

Stefan Edelkamp[a], Amr Elmasry[b], Jyrki Katajainen[b]

[a] *Faculty 3—Mathematics and Computer Science, University of Bremen*
PO Box 330 440, 28334 Bremen, Germany
edelkamp@tzi.de
[b] *Department of Computer Science, University of Copenhagen*
Universitetsparken 1, 2100 Copenhagen East, Denmark
{*elmasry, jyrki*}*@diku.dk*

Abstract

The weak heap is a priority queue that was introduced as a competitive structure for sorting. Its array-based form supports the operations *find-min* in $O(1)$ worst-case time, and *insert* and *delete-min* in $O(\lg n)$ worst-case time using at most $\lceil \lg n \rceil$ element comparisons. Additionally, its pointer-based form supports *delete* and *decrease* in $O(\lg n)$ worst-case time using at most $\lceil \lg n \rceil$ element comparisons. In this paper we enhance this data structure as follows:

1. We improve the array-based form to support *insert* in $O(1)$ amortized time. The main idea is to temporarily store the inserted elements in a buffer, and, once the buffer is full, to move its elements to the heap using an efficient bulk-insertion procedure. As an application, we use this variant in the implementation of adaptive heapsort. Accordingly, we guarantee, for several measures of disorder, that the formula expressing the number of element comparisons performed by the algorithm is optimal up to the constant factor of the high-order term. Unlike other previous constant-factor-optimal adaptive sorting algorithms, adaptive heapsort relying on the developed priority queue is practically workable.
2. We improve the pointer-based form to support *insert* and *decrease* in $O(1)$ worst-case time per operation. The expense is that *delete* then requires at most $2\lceil \lg n \rceil$ element comparisons, but this is still better than the $3\lfloor \lg n \rfloor$ bound known for run-relaxed heaps. The main idea is to allow some nodes to violate the weak-heap ordering; we call the resulting priority queue a *relaxed weak heap*. We also develop a more efficient amortized variant that provides *delete* guaranteeing an amortized bound of $1.5\lceil \lg n \rceil$ element comparisons, which is better than the $2\lceil \log_\phi n \rceil$ bound known for

[1] The material on adaptive sorting was presented at the 22nd International Workshop on Combinatorial Algorithms held in Victoria, Canada, in June 2011; and the material on shortest paths was presented at the 18th Computing: The Australasian Theory Symposium in Melbourne, Australia, in February 2012.

Fibonacci heaps, where ϕ is the golden ratio. As an application, we use this variant in the implementation of Dijkstra's shortest-paths algorithm. Experimental results indicate that weak heaps are practically efficient; they are competitive with other priority-queue structures when considering the number of element comparisons performed, and lose by a small margin when considering the actual running time.

---

1. Introduction

A priority queue is an important data structure that can be used to solve many fundamental problems like the sorting problem [1], the single-source shortest-paths problem [2], and the minimum-spanning-tree problem [3]. For a comparison function operating on a totally ordered set of elements, a priority queue $\mathcal{Q}$ usually supports the following operations:

find-min. Return an element with the minimum value in $\mathcal{Q}$.

insert. Insert a given element into $\mathcal{Q}$.

delete-min. Remove an element with the minimum value from $\mathcal{Q}$.

delete. Remove a specified element from $\mathcal{Q}$.

decrease. Decrease the value of a specified element in $\mathcal{Q}$ to a given value.

We call a priority queue *elementary* if it supports the operations *find-min*, *insert*, and *delete-min*; and *addressable* if it, in addition, supports the operations *delete* and *decrease* (for both, an efficient implementation requires that handles to elements are provided). In some applications, like sorting, only an elementary priority queue is needed. The most prominent priority queues that support these operations include (see, e.g. [4]): binary heaps [1], binomial queues [5], and Fibonacci heaps [6]. Of these, a Fibonacci heap is important for the implementation of many graph algorithms, since it supports *insert* and *decrease* in $O(1)$ amortized time, and *delete-min* in $O(\lg n)$ amortized time.[2]

In this paper we study the weak-heap data structure [7]. A *weak heap* has three important properties that distinguish it from an ordinary binary heap [1]:

1. The weak heap is represented as a binary tree, where the root has no left child. Sometimes, this kind of tree is called a *half tree* (see, e.g. [8]).
2. Except for the root, the nodes that have at most one child are at the last two levels only. However, leaves at the last level can be scattered, i.e. the last level is not necessarily filled from left to right.

[2] Throughout the paper we use n to denote the number of elements stored in the data structure prior to the operation in question and $\lg n$ as a shorthand for $\log_2(\max\{2, n\})$.

3. Each node stores an element that is less than or equal to every element in the right subtree of that node, but the relation to the elements in the left subtree is arbitrary. We call this property the *weak-heap ordering*; in the literature, the terms *variant heap property* [9] and *half-ordered binary trees* [8, 10] have also been used.

A weak-heap-ordered half tree arises naturally when a heap-ordered multi-way tree is viewed as a binary tree (rightmost-child left-sibling representation). A *perfect weak heap* that stores exactly 2^r elements is a binary-tree representation of a heap-ordered binomial tree of rank r [5]. On the other hand, a heap-ordered binomial tree is a compact representation of a perfect tournament [5]. So, tournament trees, binomial queues, and weak heaps are closely related data structures. The following are other distinguishable properties of a weak heap: 1) it can be imperfect (in contrast to a binomial tree); 2) it is a single tree (in contrast to a binomial queue, which is a collection of perfect trees); and 3) it is fairly balanced (in contrast to, for example, a pairing heap [10]).

An array-based implementation of weak heaps was described in details by Dutton [7], but the structure was introduced earlier in a technical report by Peterson [9]. Even though Peterson sketched the idea, Dutton implemented the data structure and showed its practical significance when used for sorting. To support subtree rotations efficiently in an array representation, a weak heap requires one additional bit per element. As a priority queue, a weak heap shares similarities with an ordinary binary heap; an insertion of an element is performed by starting from a leaf upwards until the heap order is reestablished, while deleting the minimum is initiated starting from the root. An array-based weak heap supports the operations *find-min* in $O(1)$ worst-case time, and *insert* and *delete-min* in $O(\lg n)$ worst-case time using at most $\lceil \lg n \rceil$ element comparisons. A weak heap of size n can be built using $n-1$ element comparisons. In Section 2, we review how the basic weak-heap operations are performed.

Weak heaps were originally introduced in the context of sorting [7, 9]. For a sequence of n elements, the worst-case number of element comparisons performed by weak-heapsort is $n\lceil \lg n \rceil - 2^{\lceil \lg n \rceil} + n - 1 \leq n \lg n + 0.09n$ [11], a value remarkably close to the lower bound of $n \lg n - 1.44n$ element comparisons. Algorithms for which the number of primitive operations is optimal up to the leading term are called *constant-factor-optimal*. Other members in this exclusive group of constant-factor-optimal heapsort algorithms are Katajainen's ultimate heapsort [12], which is fully in-place, and McDiarmid's and Reed's variant of bottom-up heapsort [13], which requires n additional bits as weak-heapsort. An improved version of weak-heapsort is used to sort indices using at most $n \lg n - 0.91n$ element comparisons, and a weak-heap variant of quicksort requires $n \lg n + 0.2n$ element comparisons on average [14].

In Section 3, our main motivation is to come up with an elementary priority queue that supports *insert* in $O(1)$ amortized time and *delete-min* in $O(\lg n)$ time using at most $\lg n + O(1)$ element comparisons. The simple—but powerful—tool we use is a buffer, into which the new elements are inserted. When the buffer becomes full, all its elements are moved to the heap by an

efficient bulk-insertion procedure. We use this variant of the weak-heap data structure in the implementation of adaptive heapsort [15]. To demonstrate that our approach is practically workable, we compare our implementation of adaptive heapsort to the best implementations of known efficient sorting algorithms (heapsort [1], splaysort [16], and introsort [17]). Our experimental settings, measurements, and outcomes are reported in Section 4.

Relaxed heaps [18] were introduced as an alternative to Fibonacci heaps to support *decrease* in $O(1)$ time. The basic idea, applied by Driscoll et al. to binomial queues [5, 19], is to permit some nodes to violate heap order, i.e. the element stored at a potential violation node may be smaller than the element stored at its parent. Driscoll et al. described two forms of relaxed heaps: 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 employed by run-relaxed heaps.

It is natural to require *find-min* to be a constant-time operation. To achieve that, the other operations are responsible for updating a pointer to the minimum after each modification of 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. Depending on the actual needs, our relaxed weak heaps can provide either logarithmic-time or constant-time *find-min*.

An array-based weak heap can be made addressable by storing the elements indirectly and associating with each element a back pointer to its position in the array. Another possibility is to make the data structure fully pointer-based. We rely on the latter approach. In Section 5, we incorporate the *decrease* operation to a pointer-based weak heap and show how to implement it in $O(1)$ time. Asymptotically, the performance of the developed priority queues—called *relaxed weak heaps*—matches that of relaxed heaps. We also work towards optimizing the bounds on the number of element comparisons needed to perform the underlying operations; in particular, the worst-case number of element comparisons performed is at most two (three if *find-min* takes constant time) per *insert* and *decrease*, and at most $2\lceil \lg n \rceil$ per *delete* and *delete-min*. The core difference between our relaxed weak heaps and the relaxed heaps of [18] is that the latter rely on multi-way trees [5] while ours use binary trees. In comparison to relaxed heaps, our key improvements are twofold. First, we immigrate the transformations of [18] into the binary-tree setting. Second, to further improve the bound on the number of element comparisons per *delete* and *delete-min*, we use a single binary tree instead of a forest of binomial trees and show how to perform the operations accordingly.

We show that relaxed weak heaps can compete to, and even improve over, all existing priority queues in network-optimization applications. 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 for Fibonacci heaps is $2m +$

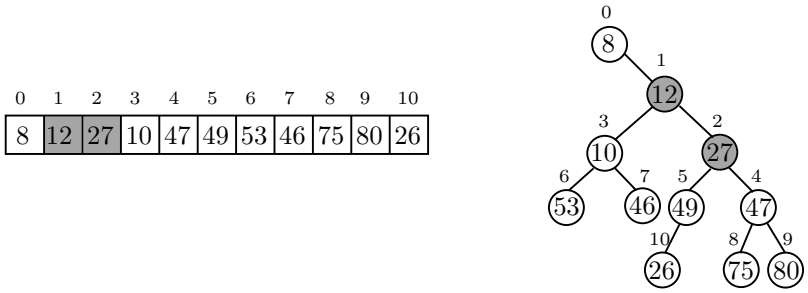

Figure 1: A weak heap of size 11: the array representation and the corresponding tree representation; the nodes for which the reverse bits are set are highlighted.

$2.89n \lg n$ element comparisons, and for other priority queues even higher. In Section 6, we back up the theoretical findings by experiments. We compare our priority queues to their natural competitors when computing the single-source shortest paths using Dijkstra's algorithm [2]. Experimental results confirm that our implementations of weak heaps and relaxed weak heaps can compete with highly-tuned implementations of binary, Fibonacci, and pairing heaps.

2. Weak Heaps

A *weak heap* [7] is a binary tree, where each node stores an element. A weak heap is obtained by loosening the requirements of a binary heap [1]. The root has no left child, and the leaves are found at the last two levels only; every node at any other level has two children. The height of a weak heap that has n elements is therefore $\lceil \lg n \rceil + 1$. The *weak-heap ordering* enforces that the element stored at a node is less than or equal to every element in the right subtree of that node. In the computer representation illustrated in Figure 1, besides the element array a, an array r of *reverse bits* is used. If the heap needs to be fully dynamic, both of these arrays must be resizable. We use a_i to refer to either the element at index i of array a or to a node in the corresponding tree structure. A weak heap is laid out such that, for a_i, the index of its left child is $2i + r_i$, the index of its right child is $2i + 1 - r_i$, and (assuming $i \neq 0$) the index of its parent is $\lfloor i/2 \rfloor$. Using the fact that the indices of the two children of a_i are exchanged when flipping r_i, subtrees can be swapped by setting $r_i \leftarrow 1 - r_i$.

Next we describe all the basic weak-heap operations in detail. A summary in pseudo-code is given in the accompanying figures (see Figures 2–8).

The *distinguished ancestor* of a_j, $j \neq 0$, is the parent of a_j if a_j is a right child, and the distinguished ancestor of the parent of a_j if a_j is a left child. We use *d-ancestor*(j) to denote the index of such ancestor. The weak-heap ordering enforces that no element is smaller than that at its distinguished ancestor.

The subroutine *join* conceptually combines two weak heaps into one weak heap conditioned on the following settings. Let a_i and a_j be two elements in a weak heap such that a_i is less than or equal to every element in the left subtree of a_j. Conceptually, a_j and its right subtree form a weak heap, while a_i and the left subtree of a_j form another weak heap. (Note that a_i can be any node except a descendant of a_j.) If $a_j < a_i$, the two elements are swapped and r_j is

procedure: *d-ancestor*
input: j: index
while $(j \ \& \ 1) = r_{\lfloor j/2 \rfloor}$
  $j \leftarrow \lfloor j/2 \rfloor$
return $\lfloor j/2 \rfloor$

Figure 2: Finding the distinguished ancestor in a weak heap.

flipped. As a result, a_j will not be larger than any of the elements in its right subtree, and a_i will not be larger than any of the elements in the subtree rooted at a_j. All in all, *join* requires $O(1)$ time and involves one element comparison.

procedure: *join*
input: i, j: indices
if $a_j < a_i$
  swap(a_i, a_j)
  $r_j \leftarrow 1 - r_j$
  return *false*
return *true*

Figure 3: Joining two weak heaps.

A weak heap of size n can be constructed using $n-1$ element comparisons by performing $n-1$ calls to the *join* subroutine. Although the call to *d-ancestor* may take more than a constant time, the $n-1$ calls involve $O(n)$ work altogether. The reason is that, among these $n-1$ calls, for every positive integer $\ell \leq \lceil \lg n \rceil$, at most $\lceil n/2^\ell \rceil$ calls require $O(\ell)$ work each.

procedure: *construct*
input: a: array of n elements; r: array of n bits
for $i \in \{0, 1, \ldots, n-1\}$
  $r_i \leftarrow 0$
for $j \in \{n-1, n-2, \ldots, 1\}$
  $i \leftarrow$ *d-ancestor*(j)
  join(i, j)

Figure 4: Constructing a weak heap.

The subroutine *sift-up*(j) is used to reestablish the weak-heap ordering between the element e, initially at location j, and those at the ancestors of a_j. Starting from location j, while e is not at the root and is smaller than the element at its distinguished ancestor, we swap the two elements, flip the bit of the node that previously contained e, and repeat from the new location of e.

To *insert* an element e, we first add e to the next available array entry making it a leaf in the heap. If this leaf is the only child of its parent, we make it a left child by updating the reverse bit at the parent. (This saves one

```
procedure: sift-up
input: j: index
while j ≠ 0
|  i ← d-ancestor(j)
|  if join(i, j)
|  |  break
|  j ← i
```

Figure 5: Remedying the weak-heap ordering on the path from a_j upwards.

unnecessary element comparison.) To reestablish the weak-heap ordering, we call the *sift-up* subroutine starting from the location of e. It follows that *insert* requires $O(\lg n)$ time and involves at most $\lceil \lg n \rceil$ element comparisons.

```
procedure: insert
input: a: array of n elements; r: array of n bits; e: element
a_n ← e
r_n ← 0
if (n & 1) = 0
|  r_⌊n/2⌋ ← 0
sift-up(n)
++n
```

Figure 6: Inserting an element into a weak heap.

The subroutine *sift-down*(j) is used to reestablish the weak-heap ordering between the element at location j and those in the right subtree of a_j. Starting from the right child of a_j, the last node on the left spine of the right subtree of a_j is identified; this is done by repeatedly visiting left children until reaching a node that has no left child. The path from this node to the right child of a_j is traversed upwards, and *join* operations are repeatedly performed between a_j and the nodes along this path. The correctness of the *sift-down* follows from the fact that, after each *join*, the element at location j is less than or equal to every element in the left subtree of the node considered in the next *join*.

To perform *delete-min*, the element stored at the root of the weak heap is replaced with that stored at the last occupied array entry. To restore the weak-heap ordering, a *sift-down* is called for the new root. Thus, *delete-min* requires $O(\lg n)$ time and involves at most $\lceil \lg n \rceil$ element comparisons.

In addition to sorting, weak heaps can be used in the following applications:

1. By building a weak heap and finding the smallest element on the left spine of the right subtree of the root, we optimally find both the smallest and second-smallest elements using $n + \lceil \lg n \rceil - 2$ element comparisons.
2. By building a weak heap and finding the maximum element among the leaves (and if n is odd, the node that has one child), we optimally find both the smallest and largest elements using $n + \lceil n/2 \rceil - 2$ element comparisons.

```
procedure: sift-down
input: j: index
k ← 2j + 1 − r_j
if k ≥ n
  | return
while 2k + r_k < n
  | k ← 2k + r_k
while k ≠ j
  | join(j, k)
  | k ← ⌊k/2⌋
```

Figure 7: Remedying the weak-heap ordering from a_j downwards.

```
procedure: delete-min
input: a: array of n elements; r: array of n bits
--n
a_0 ← a_n
if n > 1
  | sift-down(0)
```

Figure 8: Deleting the minimum of a weak heap.

3. Using a weak heap, we can slightly improve the bound for k-way merging of n elements over the $n\lceil \lg k \rceil + k - 1$ bound achieved when using a tournament tree. This can be done with at most $n\lceil \lg k \rceil + 0.086k - 1$ element comparisons as follows. Building a weak heap of size k requires $k - 1$ element comparisons, processing the first $n - k$ *delete-min* operations requires at most $(n - k)\lceil \lg k \rceil$ element comparisons, and processing the last k *delete-min* operations requires at most $\sum_{i=1}^{k} \lceil \lg i \rceil \leq k \lg k - 0.914k$ element comparisons (the last inequality has been proven in [14]).

3. Weak Heaps with Bulk Insertions

The cost of *insert* can be improved to an amortized constant. The key idea is to use a *buffer* that supports constant-time insertion. The buffer can be implemented as a separate resizable array, or as an extension of the element array a. Additionally, a pointer to the minimum element in the buffer is maintained. The maximum size of the buffer is set to $\lceil \lg n \rceil + 2$, where n is the total number of elements stored. A new element is inserted into the buffer as long as its size is below the threshold. Once the threshold is reached, a *bulk insertion* is performed by moving all the elements of the buffer to the weak heap. For the *delete-min* operation, the minimum of the buffer is compared with the minimum of the weak heap, and accordingly the operation is performed either in the buffer or in the weak heap. Deleting the minimum of the buffer is done by removing the minimum and scanning the buffer to determine the new minimum. Matching the

```
procedure: bulk-insert
input: b: array of k elements
right ← n + k − 1
left ← max{n, ⌊right/2⌋ + 1}
while k > 0
|  −−k
|  a_n ← b_k
|  r_n ← 0
|  ++n
while right > left + 1
|  left ← ⌊left/2⌋
|  right ← ⌊right/2⌋
|  for j ∈ {left, left + 1, ..., right}
|  |  sift-down(j)
j ← 0
if right ≠ 0
|  j ← d-ancestor(right)
|  sift-down(j)
if left ≠ 0
|  i ← d-ancestor(left)
|  sift-down(i)
|  sift-up(i)
sift-up(j)
```

Figure 9: Bulk insertion from a buffer b having k elements.

bounds for the weak heap, deleting the minimum of the buffer requires $O(\lg n)$ time and involves at most $\lceil \lg n \rceil$ element comparisons. Thus, *delete-min* involves at most $\lceil \lg n \rceil + 1$ element comparisons.

Let us consider how to perform a bulk insertion in $O(\lg n)$ time (see Figure 9). The elements of the buffer are first moved to the first vacant locations of the element array a (unless the buffer was implemented as an extension of a). The weak-heap ordering is then reestablished bottom-up level-by-level. Starting with the parents of the new nodes, for each node we perform a *sift-down* operation to restore the weak-heap ordering between the element at this node and those in its right subtree. We then consider the parents of these nodes on the next upper level, restoring the weak-heap ordering up to this level. This is repeated until the number of nodes that we need to deal with at a level is two. At this point, we have to switch to a more efficient strategy; otherwise, the amortized cost per *insert* would be logarithmic (due to the cost of repeated *sift-down* operations)! In the second phase, we reestablish the weak-heap ordering on the two paths from these two nodes upwards. To do that we identify the distinguished ancestor for each of the two nodes, then perform a *sift-down* operation followed by a *sift-up* operation starting at each of the two distinguished ancestors.

The correctness of the bulk-insertion follows since, after considering the ℓ-th

level, the value at each node up to level ℓ is less than or equal to the value at every node of its right subtree. Consider a $join(i, j)$ operation that is performed by the *sift-up* subroutine in the second phase of the procedure. At this point, the value of a_i is less than or equal to every element in the left subtree of a_j; this ensures the validity of the precondition for the *join* operations. After the *sift-up* operations, the weak-heap ordering must have been restored everywhere.

The intuition behind the constant amortized cost per inserted element, as illustrated in the upcoming theorem, is as follows. The number of nodes that need to be considered almost halves from a level to the next higher level. In the meantime, the amount of work needed for a *sift-down* only increases linearly with the level number. The total work done when there are only two nodes to be considered is obviously logarithmic (two *sift-down* and two *sift-up* operations). Because the number of elements inserted is logarithmic, this last cost is amortized as a constant per element.

Theorem 1. *For a weak heap with bulk insertions, the running time of insert is* $O(1)$ *in the amortized sense. The number of element comparisons performed per insert is* $5 + o(1)$.

Proof. Let $k = \lceil \lg n \rceil + 2$ be the number of elements moved from the buffer to the weak heap by the bulk-insertion procedure.

We separately consider two phases of the procedure. The first phase comprises the process of performing *sift-down* operations for the nodes at the levels with more than two involved nodes. The total number of those nodes at the ℓ-th last level is at most $\lfloor (k-2)/2^{\ell-1} \rfloor + 2$. Here we use the fact that the number of parents of a contiguous block of b elements in the array representing a weak heap is at most $\lfloor (b-2)/2 \rfloor + 2$. For a node at the ℓ-th last level, a *sift-down* operation requires $\ell - 1$ element comparisons and $O(\ell)$ work. It follows that the number of element comparisons performed in the first phase is at most $\sum_{j=1}^{\lg k} (j/2^j \cdot k + 2j) < 2k + o(k) = 2\lceil \lg n \rceil + o(\lg n)$. The running time is proportional to this quantity; that is $O(\lg n)$.

The second phase comprises two *sift-down* and two *sift-up* operations. The number of element comparisons performed in a *sift-down* and a *sift-up* operation starting from the same node adds up to $\lceil \lg n \rceil$. It follows that the number of element comparisons performed in the second phase is $2\lceil \lg n \rceil$. Once again the running time is proportional to this quantity; that is $O(\lg n)$.

When $k = \lceil \lg n \rceil + 2$, the number of element comparisons is less than $4\lceil \lg n \rceil + o(\lg n)$; this accounts for about four comparisons per element in the amortized sense. Due to the bulk insertion and the comparison between every inserted element and the minimum of the buffer, *insert* involves about five element comparisons on an average. □

While preserving the $\lceil \lg n \rceil + 1$ bound on the number of element comparisons performed by *delete-min*, the amortized number of element comparisons per *insert* can be reduced to $4 + o(1)$ as follows. Instead of one buffer, we use k buffers each of size k. We keep the minimum of each buffer at its first location, and we maintain a pointer to the overall minimum of the buffers. An inserted

element is stored in a buffer as long as one has an empty slot. After the insertion, the minimum of this buffer as well as the overall minimum of the buffers are adjusted if necessary; this requires at most two element comparisons. If all the buffers are full, we apply a bulk insertion to the k^2 elements in the buffers in the same manner as before; this accounts for $(2 + o(1))k^2 + 2\lceil \lg n \rceil$ element comparisons in total. For the *delete-min* operation, the overall minimum of the buffers is compared with the minimum of the weak heap. If the minimum is in a buffer, after the deletion we find the new minimum within this buffer and then the new overall minimum of the buffers; this involves at most $2k - 3$ element comparisons. If the minimum is in the weak heap, the deletion is performed in the way explained earlier. Setting $k = \lceil (\lg n)/2 \rceil$, the claimed bounds hold.

4. Application: Adaptive Sorting

To examine the applicability of the variant of the weak-heap data structure with a buffer, we used it in the implementation of the adaptive-heapsort algorithm [15]. In this section we describe the basic version of adaptive heapsort, summarize the analysis of its performance, and discuss the settings and outcomes of our performance tests. In these tests we measured the actual running time of the programs and the number of element comparisons performed.

4.1. Adaptive heapsort

The algorithm begins by building a *Cartesian tree* [20] for the input $X = \langle x_0, x_1, \ldots, x_{n-1} \rangle$. The root of the tree stores $x_k = \min\{x_0, x_1, \ldots, x_{n-1}\}$, the left subtree of the root is a Cartesian tree for $\langle x_0, \ldots, x_{k-1} \rangle$, and the right subtree of the root is a Cartesian tree for $\langle x_{k+1}, \ldots, x_{n-1} \rangle$. Such a tree can be built in $O(n)$ time [21] by scanning the input in order and inserting each element x_i into the existing tree using the previous insertion point as a hint from where to start the insertion process. More precisely, the bottommost node on the *right spine* of the tree (the path from the root to the rightmost leaf) is used to start from, and the nodes along this path are traversed bottom up until a node with an element x_j that is less than or equal to x_i is found. In such case, the right subtree of the node of x_j is made the left subtree of the node of x_i, and the node of x_i is made the right child of the node of x_j. If x_i is smaller than all the elements on the right spine, the whole tree is made the left subtree of the node of x_i. This procedure requires at most $2n - 3$ element comparisons [15].

The algorithm proceeds by moving the smallest element at the root of the Cartesian tree into a priority queue. The algorithm then continues by repeatedly outputting and deleting the minimum from the priority queue. After each deletion, the elements at the children of the Cartesian-tree node corresponding to the deleted element are inserted into the priority queue. As for the priority-queue operations, n *insert* and n *delete-min* operations are performed. But the heap will be small if the input sequence has a high amount of existing order. The algorithm is summarized in pseudo-code form in Figure 10.

The following improvement to the algorithm [15] is both theoretically and practically effective; even though, in this paper, it only affects the constant in the

```
procedure: adaptive-heapsort
input: x: array of n elements
C.construct()
hint ← 0
for i ∈ {0, 1, ..., n − 1}
|  hint ← C.insert(x_i, hint)
Q.construct()
Q.insert(C.find-min())
for j ∈ {0, 1, ..., n − 1}
|  x_j ← Q.delete-min()
|  Let Y be the (at most two) children x_j has in C
|  for y ∈ Y
|  |  Q.insert(y)
```

Figure 10: Adaptive heapsort in pseudo-code; $\mathcal{C}$ is the Cartesian tree used and $\mathcal{Q}$ the priority queue used. $\mathcal{C}$ stores copies of elements; $\mathcal{Q}$ stores references to $\mathcal{C}$ and the priorities are the values of the elements referred to.

linear term. Since at least $\lfloor n/2 \rfloor$ of the *delete-min* operations are immediately followed by an *insert* operation (deleting a node that is not a leaf of the Cartesian tree must be followed by an insertion), every such *delete-min* can be combined with the following *insert*. This can be implemented by replacing the minimum of the priority queue with the new element and thereafter reestablishing the heap ordering. Accordingly, the cost for half of the insertions will be saved.

Let $Inv(X)$ be the number of inversions in X; that is the number of pairs of elements that are in the wrong order, i.e. $Inv(X) = |\{(i,j) \mid 1 \leq i < j \leq n \text{ and } x_i > x_j\}|$ [22, Section 5.1.1]. Adaptive heapsort is designed to be asymptotically optimal with respect to several measures of disorder, including the measure Inv. The worst-case running time of the algorithm is $O(n \lg(1 + Inv(X)/n) + n)$ [15]. For a constant β, the number of element comparisons performed is at most $\beta n \lg(1 + Inv(X)/n) + O(n)$. Levcopoulous and Petersson suggested using a binary heap [1], which results in $\beta = 3$ (can be improved to $\beta = 2.5$ by combining *delete-min* and *insert* whenever possible). By using a binomial queue [5], we get $\beta = 2$. By using a weak heap [7], we get $\beta = 2$ (can be improved to $\beta = 1.5$ by combining *delete-min* and *insert*). By using the complicated multipartite priority queue [23], we indeed achieve $\beta = 1$. By using our variant of weak heaps that supports insertions at amortized constant cost, we also get $\beta = 1$. The purpose of our experiments is to examine whether we can achieve constant-factor-optimality and still ensure practicality!

In addition to the priority queue, the storage required by the algorithm is $2n$ extra pointers for the Cartesian tree. (We need not keep parent pointers since, during the construction, we can temporarily revert each right-child pointer on the right spine of the tree to point to the parent.) We also need to store the n elements inside the nodes of the Cartesian tree, either directly or indirectly.

4.2. Implementations considered

Our implementation of adaptive heapsort using a weak heap was array-based. Each entry of the array representing the Cartesian tree stored a copy of an element and two references to other entries in the tree. The arrays representing the weak heap and the buffer stored references to the Cartesian tree, and a separate array was used for the reverse bits. The amount of extra space used per element was three references, a copy of the element, and one bit. Dynamic memory allocation was avoided by preallocating all arrays from the stack. Users should be aware that, due to the large space requirements, the algorithm has a restricted utility depending on the amount of memory available.

To select suitable competitors, we consulted some earlier research papers concerning the practical performance of inversion-optimal sorting algorithms [16, 24, 25]. Based on this survey, we concluded that splaysort performs well in practice. In addition, the implementation of Moffat et al. [16] is highly tuned, practically efficient, and publicly available. Consequently, we selected their implementation of splaysort as our primary competitor. In the aforementioned experimental papers, splaysort has been reported to perform better than other tree-based algorithms (e.g. AVL-sort [26]), cache-oblivious algorithms (e.g. greedysort [27]), and partition-based algorithms (e.g. splitsort [28]).

When considering comparison-based sorting, one should not ignore quicksort [29]. Introsort [17] is a highly tuned variant of quicksort that is known to be fast in practice. It is based on half-recursive median-of-three quicksort, it coarsens the base case by leaving small subproblems unsorted, it calls insertionsort to finalize the sorting process, and it calls heapsort if the recursion depth becomes too large. Using the middle element as a candidate for the pivot, and using insertionsort at the back end make introsort adaptive with respect to the number of element comparisons (though not optimally adaptive with respect to any known measure of disorder). In addition, quicksort and its variants are known to be optimally adaptive with respect to the number of element swaps performed [30]. For these reasons, we selected the standard-library implementation of introsort shipped with our C++ compiler as our second competitor. Since an implementation of heapsort was readily available in the standard library, we used heapsort as the third competitor.

The length of a program could be a valid reflection to its simplicity, even though the lines-of-code[3] (LOC) metric does not exactly capture the intellectual challenge of creating the programming artifact in hand. Although the LOC metric may be considered questionable, we found it interesting to compare the code complexity of the sorting programs considered. Therefore, we extracted the code for introsort and heapsort from the standard library implementation (`gcc` version 4.5.2) and made LOC measurements. We did the same measurements for the splaysort code and our adaptive-heapsort code. The counts are given in Table 1. The adaptive-heapsort program is the longest, but it is not significantly

[3]Note that, in the LOC counts reported in this paper, we do not count comment lines and lines with a single parenthesis. We also count long statements as single lines.

longer than the other programs.

Table 1: Approximative LOC counts for various sorting programs.

Program	**LOC**
Heapsort	71
Introsort	191
Splaysort	172
Adaptive heapsort	268

4.3. Experiments

In the performance tests[4], the results of which are discussed here (see Figure 11), we used 4-byte integers as input data. The results were similar for different input sizes; for the reported experiments the number of elements was fixed to 10^8. We ensured that all the input elements were distinct. Integer data was sufficient to back up our theoretical analysis. However, for other types of input data, the number of element comparisons performed may have a more significant influence on the running time.

To generate the input data, we used *controlled shuffling* as in [24]. We started with a sorted sequence of the integers from 1 to n, and performed two types of perturbations; we call the sequences resulting from these two phases *local* and *global shuffles*. For local shuffles, the sorted sequence was broken into $\lceil n/m \rceil$ consecutive blocks each containing m elements (except possibly the last block), and the elements of each block were randomly permuted. For global shuffles, the sequence produced by the first phase was broken into m consecutive blocks each containing $\lceil n/m \rceil$ elements (except possibly the last block). From each block one element was selected at random, and these elements were randomly permuted. A small value of m means that the sequence is sorted or almost sorted, and a large value of m means that the sequence is random. Given a parameter m, this shuffling results in a sequence with expected $\Theta(n \cdot m)$ inversions. Since the resulting sequence is a permutation of the integers from 1 to n, the number of inversions could be easily calculated as $\sum_{i=1}^{n} |x_i - i|/2$.

The experiments showed that our implementation of adaptive heapsort performs a low number of element comparisons. The observed numbers are in alignment with our analytical results. When the number of inversions was small, splaysort performed fewer element comparisons than adaptive heapsort; when the number of inversions was large, splaysort performed a few more element comparisons than adaptive heapsort. In all our experiments, introsort was a bad performer with respect to the number of element comparisons; it showed

[4]All the experiments discussed throughout the paper were carried out 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 memory, 256 KB L2 cache memory, 8 MB (shared) L3 cache memory, and 12 GB main memory. All programs were compiled using GNU C++ compiler (`gcc` version 4.3.3 with option `-O3`).

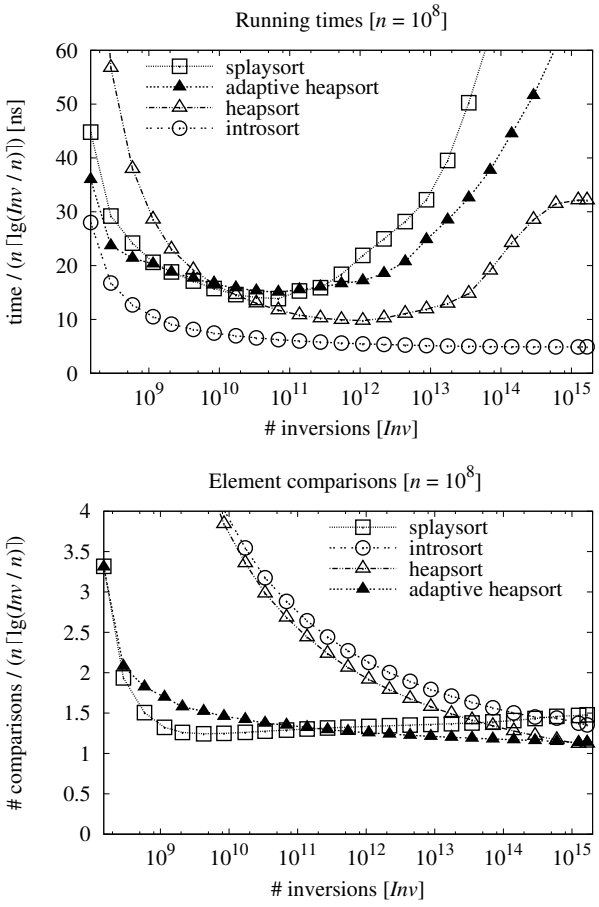

Figure 11: The CPU time used and the number of element comparisons performed by different sorting algorithms; $n = 10^8$. The observed values have been divided by $n \lceil \lg (Inv/n) \rceil$.

very little adaptivity and came last in the competition. Heapsort behaved stably but almost as badly as introsort; it performed about $n \lg n$ element comparisons independent of the amount of disorder in the input.

As to the running time, adaptive heapsort was faster than splaysort for almost all experiments. For random data, splaysort performed worst, and adaptive heapsort could be up to a factor of 15 slower than introsort, whereas heapsort was only a factor of 2–6 slower than introsort. In most experiments, introsort was the fastest sorting method; it was only beaten by adaptive heapsort when the number of inversions was very small (less than n).

Noting that the relative behaviour of the implemented algorithms is not consistent when comparing the two plots of Figure 11, it is evident that the number of element comparisons cannot be used to predict the actual running

time. On contemporary computers, if the elements are small objects and element comparisons are cheap operations, cache effects would have more significant influence on the running time; this is in particular true for large problem sizes.

5. Relaxed Weak Heaps

In our treatment, we realize relaxed weak heaps using pointers. In analogy with relaxed heaps [18], we allow some nodes to violate the weak-heap ordering. A *marked node* is potentially weak-heap-order violating; the root is always non-marked. The structure that keeps track of the marked nodes is called a *mark registry*. It must be possible to efficiently add a new marked node, remove a given marked node, and reduce the number of marked nodes if it is too big.

There are two differences for our treatment when compared to the standard priority-queue implementations. First, every node stores its *depth*, not its rank; the depth of the root is zero. Second, we have to keep track of the leaves to make it possible to expand and contract the heap following the heap operations: 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 (because a node at any other level has two children). 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.

The key ingredient is a set of transformations used to reduce the number of marked nodes. Each transformation involves a constant number of structural changes. The primitive transformations are visualized in a pictorial form in Figure 12. 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 producing 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 depth. The cleaning transformation does not require element comparisons, each of the parent and sibling transformations involves one element comparison, and the pair transformation involves two element comparisons.

Let λ be the number of marked nodes. For a relaxed weak heap of size n, we settle $\lambda \leq \lceil \lg n \rceil - 1$. Since the height of the heap is $\lceil \lg n \rceil + 1$ and the root is never marked, there will be, in addition to the root level, at least one more level where there are no marked nodes. For run-relaxed weak heaps, we adopt a lazy strategy where one marked node is removed at a time. For rank-relaxed weak heaps, we adopt an eager strategy where the transformations are employed until they cannot be applied anymore.

Accompanying each λ-reducing (parent, singleton, or run) transformation, 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 are still

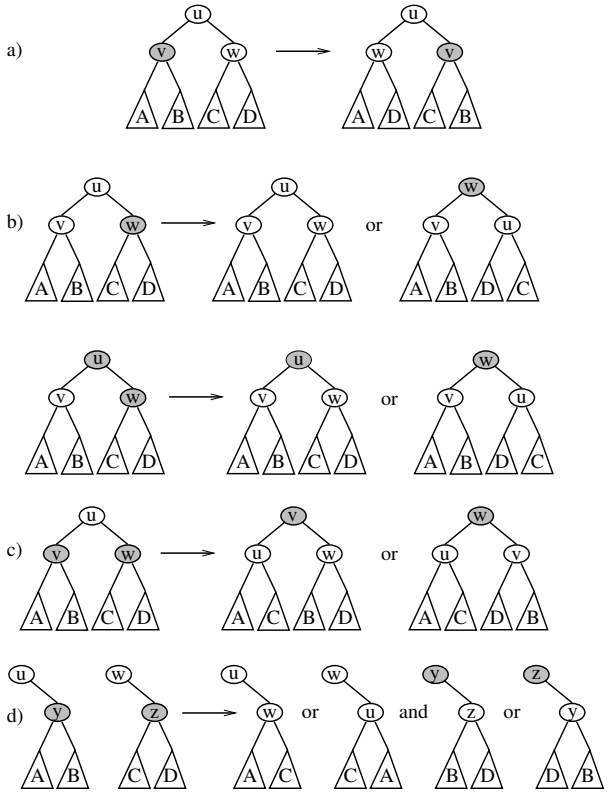

Figure 12: Primitive transformations: a) cleaning transformation at v, b) parent transformation at w, c) sibling transformation at v, and d) pair transformation for y and z. Grey nodes are marked.

at the last two levels, and the two lists of the leaf registry need not be altered. However, when a transformation moves a node from the last level to the second-to-last level and vice versa, the leaf registry must be updated accordingly.

5.1. Run-relaxed weak heaps

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 followed 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 [18]. The run leaders are kept in a *run list*. All singletons are kept in a *singleton table*, which is a resizable array accessed by depth. Each entry of the singleton table corresponds to a depth; pointers to singletons having this depth are kept

in a list of colliding markings. For each entry of the singleton table, the list of which has more than one singleton, a counterpart is kept in a *singleton-pair list*. When a singleton is marked, a new entry pointing to this node is added to the corresponding list of colliding markings. To facilitate the removal of a specified marked node, each node stores a back pointer to such an entry once created. The worst-case time of these operations is $O(1)$.

There are two compound transformations that can be used to reduce λ: 1) the *singleton transformation* is used for combining colliding singletons, and 2) the *run transformation* is used for making runs shorter. We show next how the primitive transformations are employed in these compound transformations.

Run transformation. When dealing with the marked nodes, the basic idea is to let the markings bubble upwards until two marked nodes have the same depth. 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. If we are not yet done, 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; in accordance, apply the sibling transformation at r. In total, at most two element comparisons are necessary: one may be performed by the parent transformation and another by the sibling transformation.

- q **is a right child (zig-zag case).** Perform a split at p, and thereafter apply the parent transformation at r. Then, join the resulting tree and that rooted at q. Two element comparisons are done: one by the join and another by the parent transformation.

Singleton transformation. If λ exceeds the threshold and the child of the root is not marked and there are no runs, at least two singletons must have the same depth. Since there are no runs, the parents of these two singletons are marked only if the singletons 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. 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. If one or both of q and s are left children, neither their parents nor their siblings are marked, apply the cleaning transformation on them to ensure that both are right children of their respective parents. Finally, apply the pair

transformation for q and s. In all cases, at most two element comparisons are performed.

The rationale behind the transformations is that, when there are more than $\lceil \lg n \rceil - 1$ marked nodes, there is either a run of marked nodes for which a run transformation is possible, one pair of singletons that root two subtrees of the same depth for which a singleton transformation is possible, or a node and its right child are both marked for which a parent transformation is possible.

Next we show how operations are performed for a run-relaxed weak heap:

find-min. The root and all marked nodes are examined, and the minimum is localized. All singletons are found by scanning the singleton table and following the lists of singletons of the same depth whenever necessary. All run leaders are found by scanning the run list, and all run members are traversed by starting from a leader and following left-child pointers until a non-marked node is reached.

decrease. After making the element replacement, the affected node is marked and an occurrence is inserted into the mark registry. If λ exceeds the threshold, a λ-reducing transformation is performed.

insert. The new node, say x, is added as a leaf at the last level. To do that, we pick a node, say y, from the list at the second-to-last level (currently having at most one child). We add x as a child of y, 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. If λ exceeds the threshold, a λ-reducing transformation is performed.

delete. Assume that we are deleting a node x. Let y be the last node on the left spine of the right subtree 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 the subheaps rooted at 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 put in its place. Since the attached node is possibly violating, it is marked. Because of the possible increase in the number of marked nodes and the reduction in the number of elements, the operation is followed by at most two λ-reducing transformations as appropriate. The correctness of this procedure follows from the correctness of *join*. 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 one less than its earlier depth. The depths of all other nodes remain unchanged.

delete-min. The minimum is either at one of the marked nodes or at the root. After the minimum node, say x, is localized, it is removed as described in *delete*. Observe that, if the minimum is at a marked node, the final marking will not increase the number of marked nodes, and if the minimum is at the root, there is no need to mark the new root. In accordance, at most one λ-reducing transformation may be performed. A further optimization can be done; we either borrow the node y if it is a leaf at the last level, or we borrow an arbitrary leaf z as before.

The next theorem summarizes the running times and number of element comparisons performed by run-relaxed weak heaps.

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

Proof. The *decrease* operation may be followed by a λ-reducing transformation, which requires at most two element comparisons. In addition, we only need to mark the node and adjust the mark registry. It follows that the worst-case time of *decrease* is $O(1)$.

The *insert* operation is performed by appending the new node as a leaf, which can be done in constant time using the leaf registry. The same actions as *decrease* are then performed on the inserted node. It follows that the worst-case time of *insert* is $O(1)$, involving at most two element comparisons.

The *delete-min* operation involves minimum finding, repeated join operations, and a possible λ-reducing transformation. Since the number of marked nodes is at most $\lceil \lg n \rceil - 1$, the number of minimum candidates including the root is $\lceil \lg n \rceil$. It follows that the worst-case time of minimum finding is $O(\lg n)$, involving at most $\lceil \lg n \rceil - 1$ element comparisons. The number of joins, and accordingly the number of element comparisons, performed to restore the weak-heap ordering is at most $\lceil \lg n \rceil - 1$. It follows that, in total, the worst-case time of *delete-min* is $O(\lg n)$, involving at most $2\lceil \lg n \rceil$ element comparisons. □

5.2. Rank-relaxed weak heaps

To remove markings eagerly, we enforce the following stronger invariants:

1. There exists at most one marked node per level.
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 the *join*.

Assume that the invariants are valid, and consider what to do when a node is marked. If the right child of that node was 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 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 depth exists.

Let q be the most-recently marked node, and let $parent(q)$ denote its parent. The propagation procedure has several cases:

- 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 procedure 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 s of the same depth as q, apply the pair transformation to q and s, and repeat 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 described earlier, and repeat the procedure 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 implementation for the mark registry is simple and space-economical. It is a resizable array storing at each entry a pointer to one marked node of that particular depth, 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-heap node uses three pointers, and a word storing both the depth and a 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 themselves. The amount of extra words can be reduced to $3n+O(\lg n)$ by storing the parent-child relationships cyclically [19].

Consider a typical network-optimization algorithm. For a priority-queue implementation, this involves the execution of n *insert*, m *decrease*, and n *delete-min* operations. Using run-relaxed weak heaps, we can show that such sequence can be executed in $O(m + n \lg n)$ time using at most $2m + 2n \lg n$ element comparisons. The bound on the number of element comparisons can be improved to $2m + 1.5n \lg n$ when using rank-relaxed weak heaps and applying the following improvement to the *delete-min* operation.

- If $\lambda < \frac{1}{2}\lceil \lg n \rceil$, perform the *delete-min* operation as above.

- If $\lambda \geq \frac{1}{2}\lceil \lg n \rceil$, perform 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 3. *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 and 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 λ-reducing transformation removes at least one marking, the total number of reductions is at most $m + n$. Since each λ-reducing transformation uses 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 minimum finding is less than $\frac{1}{2}\lceil \lg n \rceil$. Then the total number of element comparisons accounted for this 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 mark-removal procedure involves at most one parent transformation for each of these levels to lift a making one level up. Since a parent transformation involves 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 $\frac{1}{2}\lceil \lg n \rceil$. As the minimum is guaranteed to be at the root after the mark removals, there are no element comparisons involved in minimum finding. Then the total number of element comparisons accounted for this 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 $n, n-1, \ldots, 1$; a worst case resulting when n *insert* operations are followed by n *delete-min* operations [7]. In accordance, 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 again use the inequality $\sum_{i=1}^{n} \lceil \lg i \rceil \leq n \lg n - 0.914n$ [14]. Together with the $2m + 2n$ element comparisons required by the λ-reducing transformations, we perform a total of at most $2m + 1.5n \lg n$ element comparisons for the whole sequence of operations. □

```
procedure: shortest-path-distances
input: G: directed, weighted graph; s: vertex
(V, E) ← G
for v ∈ V
|  state_v ← unlabelled
distance_s ← 0
state_s ← labelled
Q ← construct()
Q.insert(s)
while Q.size() ≠ 0
|  v ← Q.delete-min()
|  state_v ← scanned
|  for e ∈ out(v)
|  |  (v, w) ← e
|  |  if state_w = scanned
|  |  |  continue
|  |  else if state_w = unlabelled
|  |  |  distance_w ← distance_v + length(e)
|  |  |  state_w ← labelled
|  |  |  Q.insert(w)
|  |  else if distance_v + length(e) < distance_w
|  |  |  Q.decrease(w, distance_v + length(e))
```

Figure 13: Dijkstra's algorithm in pseudo-code; $\mathcal{V}$ is the set of vertices and $\mathcal{E}$ the set of edges in the input graph $\mathcal{G}$, and $\mathcal{Q}$ is the priority queue used. We assume that, for each vertex v, $out(v)$ is the set of its outgoing edges and, for each edge e, $length(e)$ denotes its length. $\mathcal{Q}$ stores references to vertices and the priorities are the tentative shortest distances from the source s computed so far.

6. Application: Computing Shortest Paths

In most textbooks on algorithms and data structures, the computation of shortest paths in a directed graph is given as an example where an addressable priority queue can be used. We consider the single-source variant of the problem, where we are given a directed graph $\mathcal{G}$ with non-negative edge lengths and a source vertex s, and the task is to find a shortest path from s to every other vertex in $\mathcal{G}$. We use this application to examine the practical significance of the theory developed. For a broader experimental study, we refer to [31].

6.1. Dijkstra's algorithm

The single-source shortest-paths problem can be solved using the classical algorithm proposed by Dijkstra [2]. This algorithm is applicable when the graph has no negative edge lengths. For the sake of simplicity, we consider only how to compute the shortest-path distances since the extension to compute a shortest-path tree that summarizes the set of shortest paths is straightforward.

As a starting point, we used the description given in [32] (for a slightly modified version, see Figure 13). The algorithm keeps the vertices in three disjoint sets: *unlabelled*, *labelled*, and *scanned*. A *state* flag is associated with each vertex to indicate in which set this vertex is. Initially, only the source is labelled and other vertices are unlabelled. At each iteration, a labelled vertex with the shortest tentative distance from the source is moved to the scanned set. A vertex that has been scanned will never be considered again. The vertices adjacent to a vertex just scanned are labelled, unless they have been scanned or are labelled already. To find a labelled vertex with the minimum tentative distance from the source, the labelled vertices are kept in a priority queue. When an unlabelled vertex becomes labelled, it must be inserted into the priority queue, and when the tentative distance of a labelled vertex becomes shorter, its value must be decreased in the priority queue.

6.2. Implementations considered

By looking at the pseudo-code, there are two important decisions to make: 1) how to represent a graph and 2) how to represent a priority queue. Our experiments showed that the representation of the input graph is more critical, so we consider that issue first. Thereafter, we discuss the priority-queue implementations considered by us.

As a standard, a graph is represented using adjacency lists. However, as pointed out by the developers of LEDA [33] 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, considering the case when the underlying priority queue is a Fibonacci heap, each node should also store 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 of memory in total.

When this engineered graph representation was used with a Fibonacci heap in the implementation of Dijkstra's algorithm, for sparse random graphs, the resulting algorithm was about a factor of two faster compared to the implementation relying on the graph structures available in LEDA. The main problem was the interconnection between the two data structures. Somehow, it was necessary to indicate the location of a vertex inside the priority queue; 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 engineered graph representation in all our further experiments.

To test the effect of the priority-queue implementation in this application, we implemented three data structures: weak heap, run-relaxed weak heap, and rank-relaxed weak heap as described in the previous sections. All our implementations have been made part of the CPH STL (`www.cphstl.dk`). For comparison purposes, we also considered two implementations from LEDA [33]: 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. The priority queues from LEDA turned out to be slower than their counterparts in the CPH STL, and hence were excluded from further consideration. Also, the array-based weak-heap implementation, created for an earlier study [34], was outperformed by a small margin by the new pointer-based weak-heap implementation, and was accordingly excluded from further consideration.

This left us with three competing priority-queue implementations to the three weak-heap variants; below we briefly comment on each of them separately.

Binary heap [1]**.** Our array-embedded binary heap implementation employs a bottom-up heapifier with handles to retain referential integrity.

Fibonacci heap [6]**.** Our implementation is extremely 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 [35]**.** Our implementation is a reconstruction of the no-aux, two-pass approach used in LEDA.

When developing the programs, we used the code written for the experiments reported in [34] as the starting point. Our goal was to simplify the code base in order to make maintenance simpler. Therefore, we completely rewrote the code base. In particular, we aimed that the data structures use the same set of transformations. We built a component framework, characterized below, that could instantiate all pointer-based weak-heap variants:

Priority-queue engine. When developing the engine that provided the basic priority-queue functionality, we applied policy-based design by allowing the engine accept several type parameters: the type of elements, comparison function, nodes, transformations, mark registry, and leaf registry.

Mark registries. Three mark registries were written: the naive registry that immediately removes a marking by repeatedly visiting the distinguished ancestors (weak heap), the lazy registry that removes a marking only when absolutely necessary (run-relaxed weak heap), and the eager registry that removes as many markings as possible after the creation of a marking (rank-relaxed weak heap). To speed up the grouping of nodes of the same depth, we associated a static array of size 64 (i.e. we assumed that $n < 2^{64}$) for the advanced mark-registry implementations. To find the

first occupied depth 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).

To give a big picture of the code complexity of the different priority-queue implementations in the CPH STL, we list the LOC counts for some of the data structures in Table 2. There are several central pieces that are shared between the data structures, like the priority-queue engine, but the LOC count of these pieces is included in the total amounts for all data structures that use them.

Table 2: Approximative LOC counts for various addressable priority queues in the CPH STL.

Priority queue	**LOC**
Weak heap	570
Run-relaxed weak heap	1 016
Rank-relaxed weak heap	897
Array-based binary heap	205
Fibonacci heap	296
Pairing heap	204

6.3. Experiments

We tested the developed programs for randomly generated graphs. For this purpose, we used the tools available in LEDA [33, Chapter 6]. We varied the edge density of the input graphs from $m = 4n$ (*sparse graphs*), to $m = 2n \lg n$ (*moderate graphs*), up to $m = n^{1.5}$ (*dense graphs*). The results of the experiments are reported in Figures 14–16.

Let us consider the outcome for dense graphs first (see Figure 16). A rather unexpected result for large dense graphs is that the number of element comparisons seems to converge to about 0.5 per edge, independent of the data structure chosen; this reflects the inner workings of Dijkstra's algorithm for random graphs [36]. For dense graphs, in spite of the large number of edges scanned, the number of priority-queue updates is rather small.

For more pragmatic graphs with $O(n)$ edges (see Figure 14) or $O(n \lg n)$ edges (see Figure 15), as expected, weak heaps are best when considering the number of element comparisons, while array-based binary heaps with a bottom-up heapifier finish second. This illustrates that—similar to sorting—due to half ordering, weak heaps show an advantage in the number of element comparisons performed compared to binary heaps (even when using a refined heapifying policy for binary heaps). It is interesting that the simple data structures with a logarithmic worst-case performance are better than any of the advanced data structures that provide constant (amortized or worst-case) guarantees for *insert* and *decrease*. Rank-relaxed weak heaps perform fewer element comparisons than pairing and Fibonacci heaps for sparse and moderate graphs. Run-relaxed weak heaps come behind rank-relaxed weak heaps in the number of element

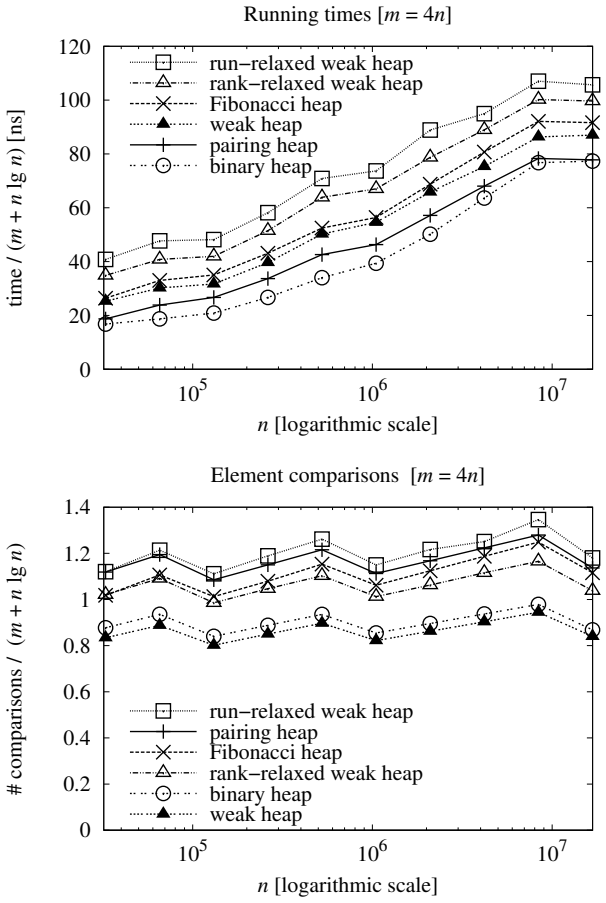

Figure 14: Performance of Dijkstra's algorithm for random graphs with $m = 4n$ using different addressable priority queues. The observed values have been divided by $m + n \lg n$.

comparisons performed. With respect to running time, binary heaps beat pairing heaps for smaller graphs, while staying behind for large graphs. Between weak and Fibonacci heaps there is an intense competition with no clear-cut winner. Both relaxed weak-heap structures come little behind; rank-relaxed weak heaps again beating run-relaxed weak heaps.

To sum up, there is not much diversity in the performance of different priority queues. However, for small problem instances, both versions of relaxed weak heaps have an overhead (see Figure 16); we attribute this fact to initializing the mark registry. Binary heaps are hard to beat in practice, as the *sift-up* loop is tight and only requires a few memory accesses and arithmetic operations. Moreover, in a typical case, only a constant number of iterations are necessary per *sift-up*. Pairing heaps are fastest for large problem instances. The number

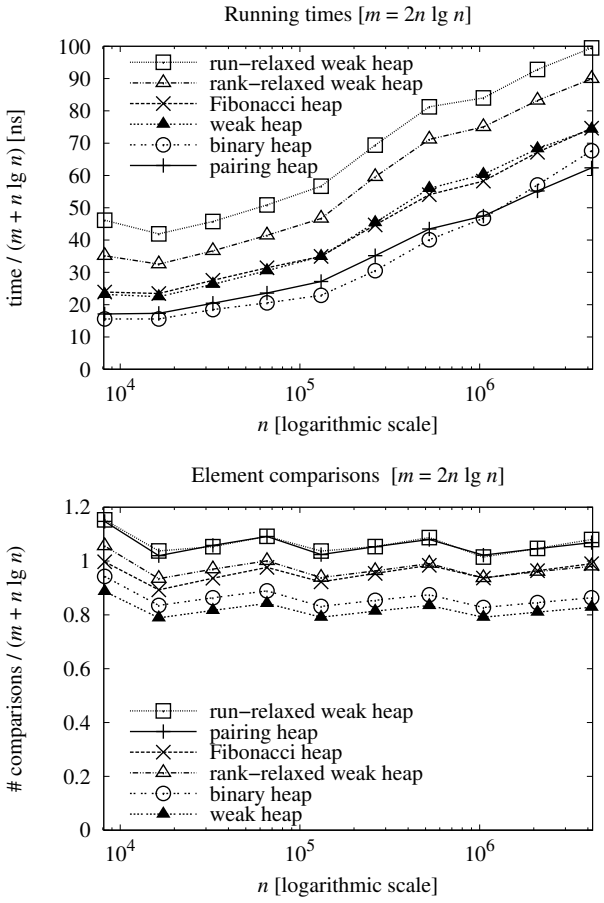

Figure 15: Performance of Dijkstra's algorithm for random graphs with $m = 2n \lg n$ using different addressable priority queues. The observed values have been divided by $m + n \lg n$.

of element comparisons, however, was smallest for weak heaps. Rank-relaxed weak heaps performed better than run-relaxed weak heaps and require less element comparisons than pairing or Fibonacci heaps. Fibonacci heaps are faster than relaxed weak-heap structures, but they lose to binary-heap and weak-heap structures.

7. Comments and Remarks

The weak heap is an interesting and intriguing data structure that, in our opinion, should be covered in textbooks on data structures and algorithms. After the introduction of the data structure, up to our work, the development of the theory concerning weak heaps was not well established. If you are looking

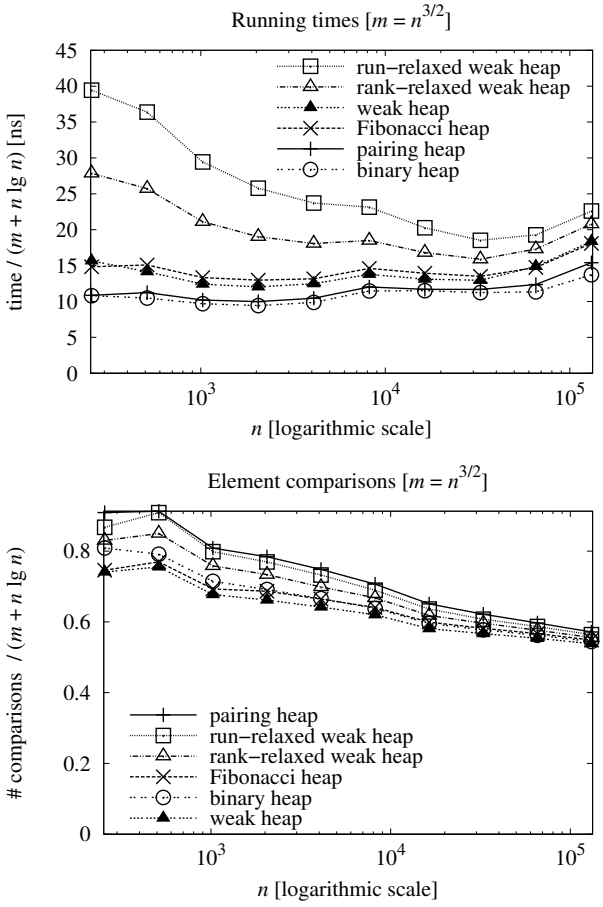

Figure 16: Performance of Dijkstra's algorithm for random graphs with $m = n^{1.5}$ using different addressable priority queues. The observed values have been divided by $m + n \lg n$.

for a priority queue that provides the best bounds with respect to the number of element comparisons performed in the worst case, the answer for your inquiry is a weak heap. However, one should be warned that the practical performance of the data structure depends on the application and computing environment.

We studied the constant-factor-optimality and practicality of adaptive heapsort when implemented with a weak heap. The question arises whether the constant factor for the linear term in the number of element comparisons can be improved; that is, how close we can get to the information-theoretic lower bound up to low-order terms. Even though our implementation of adaptive heapsort outperformed the state-of-the-art implementation of splaysort, the C++ standard-library introsort was faster for most inputs of integer data. Despite decades of research on adaptive sorting, there is still a gap between the theory

and the actual computing practice.

Even though our version of adaptive heapsort is constant-factor-optimal with respect to the number of element comparisons performed for several measures of disorder, the high number of cache misses is not on our side. Compared to earlier implementations of adaptive heapsort, a buffer increased the locality of memory references and thus reduced the number of cache misses incurred. Still, introsort has considerably better cache behaviour. It remains unanswered whether constant-factor-optimality with respect to the number of element comparisons can be achieved side by side to cache efficiency.

Another drawback of adaptive heapsort is the extra space required by the Cartesian tree. In introsort the elements are kept in the input array and sorting is carried out in-place. Overheads attributable to pointer manipulations, and a high memory footprint in general, deteriorate the performance of any implementation of adaptive heapsort. This is in particular true when the amount of disorder is high. The question arises whether the memory efficiency of adaptive heapsort can be improved without sacrificing the constant-factor-optimality with respect to the number of element comparisons.

We also considered the complexity of handling an operation sequence that appears in typical network-optimization algorithms. 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. Although it is possible to achieve $n \lg n + 3n \lg \lg n + O(m + n)$ element comparisons for this operation sequence using the priority queue developed in [37], the constant in the $O(\cdot)$ term for m and n is bigger than 2. It is open if the bound of $2m + n \lg n + o(n \lg n)$ element comparisons can be achieved for our reference sequence of operations.

When implementing Dijkstra's algorithm, our experiments indicated that the improvement achieved is basically analytical. We once more note that there is a gap between the theoretical worst-case bounds and the actual performance encountered in practice. In the test scenarios considered, for priority queues that support *decrease* operation at worst-case logarithmic cost (like binary and weak heaps), the number of element comparisons performed was consistently smaller than the observed values for relaxed weak heaps. One should question whether the theoretical worst-case bounds can indeed be used to predict the effectiveness of a data structure on real-world data.

8. Conclusion

The weak heap is theoretically superior over other priority queues when considering the number of element comparisons performed in the worst case. In practice, the story is most likely different. Other performance measures, like the amount of storage used and the number of cache misses incurred, could be more relevant depending on the application in hand. In addition, the theoretical worst-case bounds are likely to be deceiving indicators for the actual average-case practical performance.

References

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

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

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

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, The MIT Press, 3rd edition, 2009.

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

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

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

[8] B. Haeupler, S. Sen, R. Tarjan, Rank-pairing heaps, in: Proceedings of the 17th European Symposium on Algorithms, volume 5757 of *Lecture Notes in Computer Science*, Springer-Verlag, 2009, pp. 659–670.

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

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

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

[12] J. Katajainen, The ultimate heapsort, in: Proceedings of the Computing: the 4th Australasian Theory Symposium, volume 20 of *Australian Computer Science Communications*, Springer-Verlag Singapore Pte. Ltd., 1998, pp. 87–95.

[13] C. J. H. McDiarmid, B. A. Reed, Building heaps fast, Journal of Algorithms 10 (1989) 352–365.

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

[15] C. Levcopoulos, O. Petersson, Adaptive heapsort, Journal of Algorithms 14 (1993) 395–413.

[16] A. Moffat, G. Eddy, O. Petersson, Splaysort: Fast, versatile, practical, Software—Practice and Experience 126 (1996) 781–797.

[17] D. R. Musser, Introspective sorting and selection algorithms, Software—Practice and Experience 27 (1997) 983–993.

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

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

[20] J. Vuillemin, A unifying look at data structures, Communications of the ACM 23 (1980) 229–239.

[21] H. N. Gabow, J. L. Bentley, R. E. Tarjan, Scaling and related techniques for geometry problems, in: Proceedings of the 16th Annual ACM Symposium on Theory of Computing, ACM, 1984, pp. 135–143.

[22] D. E. Knuth, Sorting and Searching, volume 3 of *The Art of Computer Programming*, Addison Wesley Longman, 2nd edition, 1998.

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

[24] A. Elmasry, A. Hammad, Inversion-sensitive sorting algorithms in practice, ACM Journal of Experimental Algorithmics 13 (2009) Article 1.11.

[25] R. Saikkonen, E. Soisalon-Soininen, Bulk-insertion sort: Towards composite measures of presortedness, in: Proceedings of the 8th International Symposium on Experimental Algorithms, volume 5526 of *Lecture Notes in Computer Science*, Springer-Verlag, 2009, pp. 269–280.

[26] A. Elmasry, Adaptive sorting with AVL trees, in: Exploring New Frontiers of Theoretical Informatics, volume 155 of *IFIP Advances in Information and Communication Technology*, Springer, 2004, pp. 315–324.

[27] G. S. Brodal, R. Fagerberg, G. Moruz, Cache-aware and cache-oblivious adaptive sorting, in: Proceedings of the 32nd International Colloquium on Automata, Languages and Programming, volume 3580 of *Lecture Notes in Computer Science*, Springer-Verlag, 2005, pp. 576–588.

[28] C. Levcopoulos, O. Petersson, Splitsort: An adaptive sorting algorithm, Information Processing Letters 39 (1991) 205–211.

[29] C. A. R. Hoare, Quicksort, The Computer Journal 5 (1962) 10–16.

[30] G. S. Brodal, R. Fagerberg, G. Moruz, On the adaptiveness of Quicksort, ACM Journal of Experimental Algorithmics 12 (2008) Article 3.2.

[31] B. V. Cherkassky, A. V. Goldberg, T. Radzik, Shortest paths algorithms: Theory and experimental evaluation, Mathematical Programming 73 (1996) 129–174.

[32] R. E. Tarjan, Data Structures and Network Algorithms, SIAM, 1983.

[33] K. Mehlhorn, S. Näher, The LEDA Platform of Combinatorial and Geometric Computing, Cambridge University Press, 1999.

[34] A. Bruun, S. Edelkamp, J. Katajainen, J. Rasmussen, Policy-based benchmarking of weak heaps and their relatives, in: Proceedings of the 9th International Symposium on Experimental Algorithms, volume 6049 of *Lecture Notes in Computer Science*, Springer-Verlag, 2010, pp. 424–435.

[35] J. T. Stasko, J. S. Vitter, Pairing heaps: Experiments and analysis, Communications of the ACM 30 (1987) 234–249.

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

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