

Two-Tier Relaxed Heaps^{*}

Amr Elmasry¹, Claus Jensen², and Jyrki Katajainen²

¹ Department of Computer Engineering and Systems
Alexandria University, Alexandria, Egypt

² Department of Computing, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark

Abstract. We introduce an adaptation of run-relaxed heaps which provides efficient heap operations with respect to the number of element comparisons performed. Our data structure guarantees the worst-case cost of $O(1)$ for *find-min*, *insert*, and *decrease*; and the worst-case cost of $O(\lg n)$ with at most $\lg n + 3 \lg \lg n + O(1)$ element comparisons for *delete*, improving the bound of $3 \lg n + O(1)$ on the number of element comparisons known for run-relaxed heaps. Here, n denotes the number of elements stored prior to the operation in question, and $\lg n$ equals $\max\{1, \log_2 n\}$.

1 Introduction

In this paper we study (min-)heaps which support the following set of operations:

find-min(H). Return the location of a minimum element held in heap H .

insert(H, e). Insert element e into heap H and return the location of e in H .

delete(H, p). Remove the element at location p from heap H .

decrease(H, p, e). Replace the element at location p in heap H with element e , which must be no greater than the element earlier located at p .

Observe that *delete-min*(H), which removes the current minimum of heap H , can be accomplished by invoking *find-min* and thereafter *delete* with the location returned by *find-min*. In the heaps studied, the location abstraction is realized by storing elements in nodes and passing pointers to these nodes.

The research reported in this paper is a continuation of our earlier work aiming to reduce the number of element comparisons performed in heap operations. In [5] (conference version) and [7] (journal version), we described how the comparison complexity of heap operations can be improved using a multi-component data structure which is maintained by moving nodes from one component to another. In a technical report [6], we were able to add *decrease* having the worst-case cost of $O(1)$ to the operation repertoire. Unfortunately, the resulting data structure is complicated. In this paper we make the data structure simpler

^{*} Partially supported by the Danish Natural Science Research Council under contracts 21-02-0501 (project Practical data structures and algorithms) and 272-05-0272 (project Generic programming—algorithms and tools).

and more elegant by utilizing the connection between number systems and data structures (see, for example, [14]).

For the data structures considered our basic requirement is that the worst-case cost of *find-min*, *insert*, and *decrease* is $O(1)$. Given this constraint, our goal is to reduce the number of element comparisons involved in *delete*. Binary heaps [17] are to be excluded based on the fact that $\lg \lg n \pm O(1)$ element comparisons are necessary and sufficient for inserting an element into a heap of size n [11]. Also, pairing heaps [9] are excluded because they cannot guarantee *decrease* at a cost of $O(1)$ [8]. There exist several heaps that achieve a cost of $O(1)$ for *find-min*, *insert*, and *decrease*; and a cost of $O(\lg n)$ for *delete*. Fibonacci heaps [10] and thin heaps [13] achieve these bounds in the amortized sense. Run-relaxed heaps [4], fat heaps [12, 13], and the meldable heaps described in [1] achieve these bounds in the worst case.

For all of the aforementioned heaps guaranteeing a cost of $O(1)$ for *insert*, $2 \lg n - O(1)$ is a lower bound on the number of element comparisons performed by *delete*, and this is true even in the amortized sense (for binomial heaps [16], on which many of the above data structures are based, this is proved in [6, 7]). Run-relaxed heaps have a worst-case upper bound of $3 \lg n + O(1)$ on the number of element comparisons performed by *delete* (see Section 2). For fat heaps the corresponding bound is $4 \log_3 n + O(1) \approx 2.53 \lg n + O(1)$, and for meldable heaps the bound is higher.

In this paper we present a new adaptation of run-relaxed heaps. In Section 2, we give a brief review of the basic operations defined on run-relaxed heaps. In Section 3, we discuss the connection between number systems and data structures; among other things, we show that it is advantageous to use a zeroless representation of a run-relaxed heap which guarantees that any non-empty heap always contains at least one binomial tree of size one. In Section 4, we describe our data structure, called a *two-tier relaxed heap*, and prove that it guarantees the worst-case cost of $O(1)$ for *find-min*, *insert*, and *decrease*, and the worst-case cost of $O(\lg n)$ with at most $\lg n + 3 \lg \lg n + O(1)$ element comparisons for *delete*.

2 Run-relaxed heaps

Since we use run-relaxed heaps as the basic building blocks of the two-tier relaxed heaps, we recall the details of run-relaxed heaps in this section. However, we still assume that the reader is familiar with the original paper by Driscoll et al. [4], where the data structure was introduced.

A *binomial tree* [16] is a rooted, ordered tree defined recursively as follows: A binomial tree of rank 0 is a single node; for $r > 0$, a binomial tree of rank r consists of the root and its r binomial subtrees of ranks $0, 1, \dots, r-1$ connected to the root in that order. We denote the root of the subtree of rank 0 the *smallest child* and the root of the subtree of rank $r-1$ the *largest child*. The size of a binomial tree is always a power of two, and the *rank* of a tree of size 2^r is r .

Each node of a binomial tree stores an element drawn from a totally ordered set. Binomial trees are maintained *heap-ordered* meaning that the element stored

at a node is no greater than the elements stored at the children of that node. Two heap-ordered trees of the same rank can be linked together by making the root that stores the non-smaller element the largest child of the other root. We refer to this as a *join*. A *split* is the inverse of a join, where the subtree rooted at the largest child of the root is unlinked from the given binomial tree. A join involves a single comparison, and both a join and a split have a cost of $O(1)$.

A *relaxed binomial tree* [4] is an almost heap-ordered binomial tree where some nodes are denoted *active*, indicating that the element stored at that node may be smaller than the element stored at the parent of that node. Nodes are made active by *decrease*, even though no heap-order violation is introduced, and remain active until the potential heap-order violation is explicitly removed. From the definition, it follows that a root cannot be active. A *singleton* is an active node whose immediate siblings are not active. A *run* is a maximal sequence of two or more active nodes that are consecutive siblings.

Let τ denote the number of trees in any collection of relaxed binomial trees, and let λ denote the number of active nodes in the *entire* collection of trees. A *run-relaxed heap* is a collection of relaxed binomial trees where $\tau \leq \lfloor \lg n \rfloor + 2$ and $\lambda \leq \lfloor \lg n \rfloor$, n denoting the number of elements stored.

To keep track of the active nodes, a *run-singleton structure* is maintained as described in [4]. All singletons are kept in a *singleton table*, which is a resizable array accessed by rank. In particular, this table must be implemented in such a way that growing and shrinking at the tail is possible at the worst-case cost of $O(1)$, which is achievable, for example, by doubling, halving, and incremental copying. Each entry of the singleton table corresponds to a rank; pointers to singletons having this rank are kept in a list. For each entry of the singleton table that has more than one singleton of the same rank a counterpart is kept in a *pair list*. The last active node of each run is kept in a *run list*. All lists are doubly linked, and each active node should have a pointer to its occurrence in a list (if any). The bookkeeping details are quite straightforward so we will not repeat them here, but refer to [4]. The fundamental operations supported are an addition of a new active node, a removal of a given active node, and a removal of at least one arbitrary active node if λ is larger than $\lfloor \lg n \rfloor$. The cost of each of these operations is $O(1)$ in the worst case.

As to the transformations needed for reducing the number of active nodes, we again refer to the original description given in [4]. The rationale behind the transformations is that, when there are more than $\lfloor \lg n \rfloor$ active nodes, there is at least one pair of singletons that root a subtree of the same rank, or there is a run of two or more neighbouring active nodes. In that case, it is possible to apply the transformations—a constant number of singleton transformations or run transformations—to reduce the number of active nodes by at least one. The cost of performing any of the transformations is $O(1)$ in the worst case. Hereafter one application of the transformations together with all necessary changes to the run-singleton structure is referred to as a λ -*reduction*.

To keep track of the trees in a run-relaxed heap, the roots are doubly linked together in a *root list*. Each tree is represented as a normal binomial tree [3],

but to support the transformations used for reducing the number of active nodes each node stores an additional pointer. That is, a node contains sibling pointers, a child pointer, a parent pointer, a rank, and a pointer to its occurrence in the run-singleton structure. The occurrence pointer of every non-active node has the value null; for a node that is active and in a run, but not the last in the run, the pointer is set to point to a fixed sentinel. To support our two-tier relaxed heap, each node should store yet another pointer to its counterpart held at the upper store (see Section 4), and vice versa.

Let us now consider how the heap operations are implemented. A reader familiar with the original paper by Driscoll et al. [4] should be aware that we have made modifications to the implementation of the heap operations to adapt them for our purposes.

A minimum element is stored at one of the roots or at one of the active nodes. To facilitate a fast *find-min*, a pointer to the node storing a minimum element is maintained. When such a pointer is available, *find-min* can be accomplished at the worst-case cost of $O(1)$.

An insertion is performed in the same way as in a worst-case efficient binomial heap [6, 7]. To obtain the worst-case cost of $O(1)$ for *insert*, all the necessary joins cannot be performed at once. Instead, one join is done in connection with each insertion, and the execution of any remaining joins is delayed for forthcoming *insert* operations. A way of facilitating this is to maintain a logarithmic number of pointers to unfinished joins on a stack. In one *join step*, the pointer at the top of the stack is popped, the two roots are removed from the root list, the corresponding trees are joined, and the root of the resulting tree is put in the place of the two. If there exists another tree of the same rank as the resulting tree, a pointer indicating this pair is pushed onto the stack. In *insert* a join step is executed, if necessary, and a new node is added to the root list. If the given element is smaller than the current minimum, the pointer indicating the location of a minimum element is updated. If there exists another tree of rank 0, a pointer to this pair of trees is pushed onto the stack. When one join is done in connection with every *insert*, the ongoing joins are disjoint and there is always space for new elements (for a formal proof, see [2, p. 53 ff.]). To summarize, *insert* has the worst-case cost of $O(1)$ and requires at most two element comparisons. An alternative way of achieving these bounds is described in [4].

There exists at most two trees of any given rank in a relaxed binomial heap. In fact, a tighter analysis shows that the number of trees is bounded by $\lfloor \lg n \rfloor + 2$. Namely, it can be shown that *insert* (as well as *delete*) maintains the invariant that between any two ranks holding two trees there is a rank holding no tree (see [2, p. 53 ff.] or [12]).

In *delete* we rely on the same borrowing technique as in [4]: the root of a tree of the smallest rank is borrowed to fill in the hole created by the node being removed. To free a node that can be borrowed, a tree of the smallest rank is repeatedly split, if necessary, until the split results in a tree of rank 0. In one *split step*, if x denotes the root of a tree of the smallest rank and y its largest

child, the tree rooted at x is split, and if y is active, it is made non-active and its occurrence is removed from the run-singleton structure.

Deletion has two cases depending on whether one of the roots or one of the internal nodes is to be removed. Let z denote the node being deleted, and assume that z is a root. If the tree rooted at z has rank 0, then z is removed and no other structural changes are done. Otherwise, the tree rooted at z is repeatedly split and, when the tree rooted at z has rank 0, z is removed. In each split step all active children of z are retained active, but they are temporarily removed from the run-singleton structure (since the structure of runs may change). Thereafter, the freed tree of rank 0 (the node borrowed) and the subtrees rooted at the children of z are repeatedly joined by processing the trees in increasing order of rank. Finally, the active nodes temporarily removed are added back to the run-singleton structure. The resulting tree replaces the tree rooted at z in the root list. It would be possible to handle the tree used for borrowing and the tree rooted at z symmetrically, with respect to the treatment of the active nodes, but when *delete* is embedded into our two-tier relaxed heap it would be too expensive to remove all active children of z in a single *delete*. To complete the operation, all roots and active nodes are scanned to update the pointer indicating the location of a minimum element. Singletons are found by scanning through all lists in the singleton table. Runs are found by accessing their last node via the run list and following the sibling pointers until a non-active node is reached.

The computational cost of deleting a root is dominated by the repeated splits, the repeated joins, and the scan over all minimum candidates. In each of these steps a logarithmic number of nodes is visited, so their total cost is $O(\lg n)$. Splits as well as updates to the run-singleton structure do not involve any element comparisons. In total, joins may involve at most $\lfloor \lg n \rfloor$ element comparisons. Even though a tree of the smallest rank is split, after the joins the number of trees is at most $\lfloor \lg n \rfloor + 2$. If the number of active nodes is larger than $\lfloor \lg(n-1) \rfloor$ (the size of the heap is now one smaller), a single λ -reduction is performed which involves $O(1)$ element comparisons. To find the minimum of $2\lfloor \lg n \rfloor + 2$ elements, at most $2\lfloor \lg n \rfloor + 1$ element comparisons are to be done. To summarize, this form of *delete* performs at most $3 \lg n + O(1)$ element comparisons.

Assume now that the node z being deleted is an internal node, and let x be the node borrowed. Also in this case the tree rooted at z is repeatedly split, and after removing z the tree of rank 0 rooted at x and the subtrees of the children of z are repeatedly joined. The resulting tree is put in the place of the subtree rooted earlier at z . If z was active and contained the current minimum, the pointer to the location of a minimum element is updated. If x is the root of the resulting subtree, node x is made active. Finally, the number of active nodes is reduced, if necessary, by performing a λ -reduction once or twice (once because one new node may become active and possibly once more because of the decrement of n , since the difference between $\lfloor \lg n \rfloor$ and $\lfloor \lg(n-1) \rfloor$ can be one).

Similar to the case of deleting a root, the deletion of an internal node has the worst-case cost of $O(\lg n)$. If z did not contain the current minimum, only at most $\lg n + O(1)$ element comparisons are done; at most $\lfloor \lg n \rfloor$ due to joins

and $O(1)$ due to λ -reductions. However, if z contained the current minimum, at most $2\lfloor \lg n \rfloor + 1$ additional element comparisons may be necessary. That is, the total number of element comparisons performed is bounded by $3\lg n + O(1)$. To sum up, each *delete* has the worst-case cost of $O(\lg n)$ and requires at most $3\lg n + O(1)$ element comparisons.

In *decrease*, after making the element replacement, the corresponding node is made active, an occurrence is inserted into the run-singleton structure, and a single λ -reduction is performed if the number of active nodes is larger than $\lfloor \lg n \rfloor$. Moreover, if the given element is smaller than the current minimum, the pointer indicating the location of a minimum element is corrected to point to the active node. All these actions have the worst-case cost of $O(1)$.

3 Number systems and data structures

The way our two-tier framework works (see Section 4) suggests that the run-relaxed heaps are to be modified before they can be used. One of the main operations required for our framework is the ability to borrow a node from the structure at a cost of $O(1)$, such that this borrowing would only produce $O(1)$ new roots in the root list. Accordingly, we introduce an operation *borrow* which fulfils this requirement. We rely on the observation that, in a run-relaxed heap, there is a close connection between the sizes of the relaxed binomial trees of the heap and the number representation of the current size of the heap denoted by n . Three different number representations are relevant:

Binary representation:

$$n = \sum_{i=0}^{\lfloor \lg n \rfloor} d_i 2^i, \text{ where } d_i \in \{0, 1\} \text{ for all } i \in \{0, \dots, \lfloor \lg n \rfloor\}.$$

Redundant representation:

$$n = \sum_{i=0}^{\lfloor \lg n \rfloor} d_i 2^i, \text{ where } d_i \in \{0, 1, 2\} \text{ for all } i \in \{0, \dots, \lfloor \lg n \rfloor\}.$$

Zeroless representation:

$$n = \sum_{i=0}^k d_i 2^i, \text{ where } k \in \{-1, 0, \dots, \lfloor \lg n \rfloor\} \text{ and } d_i \in \{1, 2, 3, 4\} \text{ for all } i \in \{0, \dots, k\}.$$

For each such representation, the heap contains d_i relaxed binomial trees of size 2^i , appearing in the root list in increasing order of rank. Now *insert* can be realized elegantly by imitating increments in the underlying number system. The worst-case efficiency of *insert* is directly related to how far a carry has to be propagated. If the binary representation is used as in [3], *insert* has the worst-case cost of $\Theta(\lg n)$. Both the redundant and zeroless representations can reduce the worst-case cost of *insert* to $O(1)$; the zeroless representation can also support *borrow* at the worst-case cost of $O(1)$.

In Section 2, the redundant representation was used. For the zeroless representation, two crucial changes are made. First, the relaxed binomial trees of the

same rank are maintained in sorted order according to the elements stored at the roots. The significant consequence of this ordering is that *delete* has to only consider one root per rank when finding a minimum element stored at the roots. Second, every carry (digit 4 which corresponds to four consecutive relaxed binomial trees of the same rank) and every borrow (digit 1) are kept on a stack in rank order. When a carry/borrow stack is available, increments and decrements can be performed as follows [2, p. 56]:

- 1) Fix the topmost carry or borrow if the stack is not empty.
- 2) Add or subtract one as desired.
- 3) If the least significant digit becomes 4 or 1, push a pointer to this carry or borrow onto the stack.

Let x be a digit in the used number system. To fix a carry, $x4$ is converted to $(x + 1)2$, after which the top of the stack is popped. Analogously, to fix a borrow, $x1$ is converted to $(x - 1)3$ and the top of the stack is popped. If a fix creates a new carry or borrow, an appropriate pointer is pushed onto the stack. In terms of relaxed binomial trees, fixing a carry means that a join is made, which produces a relaxed binomial tree whose rank is one higher; and fixing a borrow means that a split is made, which produces two relaxed binomial trees whose rank is one lower.

To summarize, *insert* is carried out by doing at most one join or one split depending on the contents of the carry/borrow stack, and injecting a new node as a relaxed binomial tree of rank 0 into the root list. Correspondingly, after a join or split, if any, *borrow* ejects one relaxed binomial tree from the root list. Due to the zeroless representation, we can be sure that the rank of every ejected relaxed binomial tree is 0, i.e. it is a single node. However, if the ejected node contains the current minimum and the heap is not empty, another node is borrowed, and the first is inserted back into the data structure. The correctness of *insert* and *borrow* is proved in [2, p. 56 ff.] (see also [12]) by showing that both increments and decrements maintain the representation *regular*, i.e. between any two digits equal to 4 there is a digit other than 3, and between any two digits equal to 1 there is a digit other than 2. (In [2] digits $\{-1, 0, 1, 2\}$ are used, but this is equivalent to our use of $\{1, 2, 3, 4\}$.) Clearly, *insert* and *borrow* can be accomplished at the worst-case cost of $O(1)$.

4 Two-tier relaxed heaps

The two-tier relaxed heap is composed of two components, the *lower store* and the *upper store*. The lower store stores the actual elements of the heap. The reason for introducing the upper store is to avoid the scan over all minimum candidates when updating the pointer to the location of a minimum element; pointers to minimum candidates are kept in a heap instead. Actually, both components are realized as run-relaxed heaps modified to use the zeroless representation as discussed in Section 3.

Upper-store operations. The upper store is a modified run-relaxed heap storing pointers to *all* roots of the trees held in the lower store, pointers to all active nodes held in the lower store, and pointers to some earlier roots and active nodes. In addition to *find-min*, *insert*, *delete*, and *decrease*, which are realized as described earlier (however *delete* could also use *borrow*), it should be possible to mark nodes to be deleted and to unmark nodes if they reappear at the upper store before being deleted. Lazy deletions are necessary at the upper store when, at the lower store, a join is done or an active node is made non-active by a λ -reduction. In both situations, a normal upper-store deletion would be too expensive. The algorithms maintain the following invariant: for each marked node whose pointer refers to a node y in the lower store, in the same tree there is another node x such that the element stored at x is no greater than the element stored at y .

To provide worst-case efficient lazy deletions, we adopt the global-rebuilding technique from [15]. When the number of unmarked nodes becomes equal to $m_0/2$, where m_0 is the current size of the upper store, we start building a new upper store. The work is distributed over the forthcoming $m_0/4$ upper-store operations. In spite of the reorganization, both the old structure and the new structure are kept operational and used in parallel. All new nodes are inserted into the new structure, and all old nodes being deleted are removed from their respective structures. Since the old structure does not handle any insertions, it can be emptied using *borrow*. In connection with each of the next at most $m_0/4$ upper-store operations, four nodes are borrowed from the old structure; if a node is unmarked, it is inserted into the new structure; otherwise, it is released and in its counterpart in the lower store the pointer to the upper store is given the value null. When the old structure becomes empty, it is dismissed and thereafter the new structure is used alone. During the $m_0/4$ operations at most $m_0/4$ nodes can be deleted or marked to be deleted, and since there were $m_0/2$ unmarked nodes in the beginning, at least half of the nodes are unmarked in the new structure. Therefore, at any point in time, we are constructing at most one new structure. We emphasize that each node can only exist in one structure and whole nodes are moved from one structure to the other, so that pointers from the outside remain valid.

Given that the cost of each *borrow* and *insert* is $O(1)$, the reorganization only adds an additional cost of $O(1)$ to all upper-store operations. A *find-min* may need to consult both the old and the new upper stores, but its worst-case cost is still $O(1)$. The cost of marking and unmarking is clearly $O(1)$. If m denotes the total number of unmarked nodes currently stored, at any point during the rebuilding process, the total number of nodes stored is $\Theta(m)$, and all the time during this process $m_0 = \Theta(m)$. Therefore, since in both structures *delete* is handled normally, except that it may take part in reorganizations, it has the worst-case cost of $O(\lg m)$ and requires $3 \lg m + O(1)$ element comparisons.

Let n be the number of elements in the lower store. The number of trees in the lower store is at most $4(\lfloor \lg n \rfloor + 1)$, and the number of active nodes is at most $\lfloor \lg n \rfloor$. At all times at most a constant fraction of the nodes stored at

the upper store can be marked to be deleted. Hence, the number of pointers is $O(\lg n)$. That is, at the upper store the worst-case cost of *delete* is $O(\lg \lg n)$, including at most $3 \lg \lg n + O(1)$ element comparisons.

Lower-store operations. The lower store is a modified run-relaxed heap storing all the elements. Minimum finding relies on the upper store; an overall minimum element is either in one of the roots or in one of the active nodes held in the lower store. The counterparts of the minimum candidates are stored at the upper-store, so communication between the lower store and the upper store is necessary each time a root or an active node is added or removed, but not when an active node is made into a root.

In addition to the modifications described in Section 3, *insert* requires three further modifications in places where communication between the lower store and upper store is necessary. First, in each join the counterpart of the root of the loser tree must be lazily deleted from the upper store. Second, in each split a counterpart of the largest child of the given root must be inserted into the upper store, if it is not there already. Third, after inserting a new node its counterpart must be added to the upper store. After these modifications, the worst-case cost of *insert* is still $O(1)$.

Deletion is done as described in Section 2, but now borrowing is done by invoking *borrow* instead of repeated splitting. As a consequence of *borrow*, a lazy deletion or an insertion may be necessary at the upper store. As a consequence of *delete*, a removal of a root or an active node will invoke *delete* at the upper store, and an insertion of a new root or an active node will invoke *insert* at the upper store. A λ -reduction may invoke one or two lazy deletions (a λ -reduction can make up to two active nodes non-active) and at most one insertion at the upper store. In total, lazy deletions and insertions have the worst-case cost of $O(1)$. Also *borrow* has the worst-case cost of $O(1)$. At most one real upper-store deletion will be necessary, which has the worst-case cost of $O(\lg \lg n)$ and includes $3 \lg \lg n + O(1)$ element comparisons. Therefore, *delete* has the worst-case cost of $O(\lg n)$ and performs at most $\lg n + 3 \lg \lg n + O(1)$ element comparisons.

In *decrease*, three modifications are necessary. First, each time a new active node is created, *insert* has to be invoked at the upper store. Second, each time an active node is removed by a λ -reduction, the counterpart must be lazily deleted from the upper store. Third, when the node whose value is to be decreased is a root or an active node, *decrease* has to be invoked at the upper store as well. If due to a λ -reduction an active node is made into a root, no change at the upper store is required. After these modifications, the worst-case cost of *decrease* is still $O(1)$.

The following theorem summarizes the main result of the paper.

Theorem 1. *Let n be the number of elements of the heap prior to each operation. A two-tier relaxed heap guarantees the worst-case cost of $O(1)$ for *find-min*, *insert*, and *decrease*; and the worst-case cost of $O(\lg n)$ including at most $\lg n + 3 \lg \lg n + O(1)$ element comparisons for *delete*.*

We conclude the paper with two remarks. 1) It is relatively easy to extend the data structure to support *meld* at the worst-case cost of $O(\min \{\lg m, \lg n\})$,

where m and n are the number of elements of the two melded heaps. 2) It is an open problem whether it is possible or not to achieve a bound of $\lg n + O(1)$ element comparisons for *delete*, when fast *decrease* is to be supported. Note that the worst-case bound of $\lg n + O(1)$ is achievable [6, 7], when *decrease* is not supported.

References

- [1] G.S. Brodal. Worst-case efficient priority queues. *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM/SIAM (1996), 52–58.
- [2] M.J. Clancy and D.E. Knuth. A programming and problem-solving seminar. Technical Report STAN-CS-77-606, Department of Computer Science, Stanford University (1977).
- [3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*, 2nd Edition. The MIT Press (2001).
- [4] J.R. Driscoll, H.N. Gabow, R. Shrairman, and R.E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM* **31** (1988), 1343–1354.
- [5] A. Elmasry. Layered heaps. *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science* **3111**, Springer-Verlag (2004), 212–222.
- [6] A. Elmasry, C. Jensen, and J. Katajainen. A framework for speeding up priority-queue operations. CPH STL Report 2004-3. Department of Computing, University of Copenhagen (2004). Available at <http://cphstl.dk>.
- [7] A. Elmasry, C. Jensen, and J. Katajainen. Multipartite priority queues. Submitted for publication (2004).
- [8] M.L. Fredman. On the efficiency of pairing heaps and related data structures. *Journal of the ACM* **46** (1999), 473–501.
- [9] M.L. Fredman, R. Sedgwick, D.D. Sleator, and R.E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica* **1** (1986), 111–129.
- [10] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM* **34** (1987), 596–615.
- [11] G.H. Gonnet and J.I. Munro. Heaps on heaps. *SIAM Journal on Computing* **15** (1986), 964–971.
- [12] H. Kaplan, N. Shafir, and R.E. Tarjan. Meldable heaps and Boolean union-find. *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, ACM (2002), 573–582.
- [13] H. Kaplan and R.E. Tarjan. New heap data structures. Technical Report TR-597-99, Department of Computer Science, Princeton University (1999).
- [14] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press (1998).
- [15] M.H. Overmars and J. van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Information Processing Letters* **12** (1981), 168–173.
- [16] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM* **21** (1978), 309–315.
- [17] J.W.J. Williams. Algorithm 232: Heapsort. *Communications of the ACM* **7** (1964), 347–348.