A Framework for Speeding Up Priority-Queue Operations

Amr Elmasry[1,*] Claus Jensen[2,†] Jyrki Katajainen[2,†]

[1] *Computer Science Department, Alexandria University*
Alexandria, Egypt
[2] *Department of Computing, University of Copenhagen*
Universitetsparken 1, 2100 Copenhagen East, Denmark

Abstract. We introduce a framework for reducing the number of element comparisons performed in priority-queue operations. In particular, we give a priority queue which guarantees the worst-case cost of $O(1)$ per minimum finding and insertion, and the worst-case cost of $O(\log n)$ with at most $\log n + O(1)$ element comparisons per minimum deletion and deletion, improving the bound of $2 \log n + O(1)$ on the number of element comparisons known for binomial queues. Here, n denotes the number of elements stored in the data structure prior to the operation in question, and $\log n$ equals $\max\{1, \log_2 n\}$. We also give a priority queue that provides, in addition to the above-mentioned methods, the priority-decrease (or decrease-key) method. This priority queue achieves the worst-case cost of $O(1)$ per minimum finding, insertion, and priority decrease; and the worst-case cost of $O(\log n)$ with at most $\log n + O(\log \log n)$ element comparisons per minimum deletion and deletion.

CR Classification. E.1 [Data Structures]: Lists, stacks, and queues; E.2 [Data Storage Representations]: Linked representations; F.2.2 [Analysis of Algorithms and Problem Complexity]: sorting and searching

Keywords. Data structures, priority queues, heaps, algorithms, sorting, meticulous analysis, constant factors

1. Introduction

One of the major research issues in the field of theoretical computer science is the comparison complexity of computational problems. In this paper, we consider priority queues (called heaps in some texts) that guarantee a cost of $O(1)$ for insert, with an attempt to reduce the number of element comparisons involved in delete-min. Binary heaps [29] are therefore excluded, following from the fact that $\log \log n \pm O(1)$ element comparisons are necessary and sufficient for inserting an element into a heap of size n

*Corresponding author. E-mail address: elmasry@alexeng.edu.eg.

†Partially supported by the Danish Natural Science Research Council under contract 21-02-0501 (project "Practical data structures and algorithms").

[15]. Gonnet and Munro [15] (corrected by Carlsson [4]) also showed that $\log n + \log^* n \pm O(1)$ element comparisons are necessary and sufficient for deleting a minimum element from a binary heap.

In the literature several priority queues have been proposed that achieve a cost of $O(1)$ per find-min and insert, and a cost of $O(\log n)$ per delete-min and delete. Examples of priority queues that achieve these bounds, in the amortized sense, are binomial queues [1, 27] and pairing heaps [12, 17, 25]. The same efficiency can be achieved in the worst case with a special implementation of a binomial queue (see, for example, [5] or [8, Section 3]). If the decrease (often called decrease-key) method is to be supported, Fibonacci heaps [13] and thin heaps [18] achieve, in the amortized sense, a cost of $O(1)$ per find-min, insert, and decrease; and a cost of $O(\log n)$ per delete-min and delete. Run-relaxed heaps [8], fat heaps [18], and meldable priority queues described in [2] achieve these bounds in the worst case.

For all the aforementioned priority queues guaranteeing a cost of $O(1)$ per insert, $2\log n - O(1)$ is a lower bound, even in the amortized sense, on the number of element comparisons performed by delete-min and delete (a derivation of this lower bound for binomial queues is given in Section 2). The upper bound of $2\log n + O(1)$ is known to hold for binomial queues in the worst case (see Section 2).

In all our data structures we use various forms of binomial trees as the basic building blocks. Therefore, in Section 2 we review how binomial trees are employed in binomial queues (called binomial heaps in [7]). In Section 3, we present our two-tier framework for structuring priority queues. We apply the framework in three different ways to reduce the number of element comparisons performed in priority-queue operations. In Section 8, we discuss which other data structures could be used in our framework as a substitute for binomial trees.

The results of this paper are as follows. In Section 4, we give a structure, called a *two-tier binomial queue*, that guarantees the worst-case cost of $O(1)$ per find-min and insert, and the worst-case cost of $O(\log n)$ with at most $\log n + O(\log\log n)$ element comparisons per delete-min and delete. In Section 5, we describe a refined priority queue, called a *multipartite binomial queue*, by which the better bound of at most $\log n + O(1)$ element comparisons per delete-min and delete is achieved. In Section 6, we show as an application of the framework that, by using a multipartite binomial queue in adaptive heapsort [21], a sorting algorithm is obtained that is optimally adaptive with respect to the inversion measure of disorder, and that sorts a sequence having n elements and I inversions with at most $n\log(I/n) + O(n)$ element comparisons. This is the first priority-queue-based sorting algorithm having these properties. Both in Section 5 and in Section 6 the results presented are stronger than those presented in the conference version of this paper [9].

In Section 7, we present a priority queue, called a *multipartite relaxed binomial queue*, that provides the decrease method in addition to the above-mentioned methods. The data structure is built upon run-relaxed binomial

queues (called run-relaxed heaps in [8]). A multipartite relaxed binomial queue guarantees the worst-case cost of $O(1)$ per insert, find-min, and decrease; and the worst-case cost of $O(\log n)$ with at most $\log n + O(\log \log n)$ element comparisons per delete-min and delete. During the course of this work we perceived an interesting taxonomy between different building blocks that can be used in our framework. In the conference version of this paper [9], it was outlined that with structures similar to thin binomial trees [18] a priority queue is obtained that guarantees, in the amortized sense, a cost of $O(1)$ per insert, find-min, and decrease; and a cost of $O(\log n)$ with at most $1.44 \log n + O(\log \log n)$ element comparisons per delete-min and delete. With fat binomial trees [18] the same costs can be achieved in the worst case and the number of element comparisons performed per delete-min and delete can be reduced to $1.27 \log n + O(\log \log n)$. Finally, with relaxed binomial trees [8] the same worst-case bounds are achieved, except that the constant factor in the logarithm term in the bound on the number of element comparisons performed by delete-min and delete can be reduced to one.

2. Binomial queues

In a generic form, a *priority queue* is a data structure which depends on four type parameters: $\mathcal{E}$, $\mathcal{C}$, $\mathcal{F}$, and $\mathcal{A}$. $\mathcal{E}$ is the type of the *elements* manipulated; $\mathcal{C}$ is the type of the *compartments* used for storing the elements, one element per compartment; and $\mathcal{F}$ is the type of the *ordering function* used in element comparisons. A compartment may also contain satellite data, like references to other compartments. We assume that the elements can only be moved and compared, both operations having a cost of $O(1)$. Furthermore, we assume that it is possible to get any datum stored at a compartment at a cost of $O(1)$. $\mathcal{A}$ is the type of the *allocator* which provides methods for allocating new compartments and deallocating old compartments. We omit the details concerning memory management, and simply assume that both allocation and deallocation have a cost of $O(1)$.

Any priority queue $Q\,\langle \mathcal{E}, \mathcal{C}, \mathcal{F}, \mathcal{A} \rangle$ should provide the following methods:

$\mathcal{E}$ **find-min().** Return a minimum element stored in Q. The minimum is taken with respect to $\mathcal{F}$. **Requirement.** The data structure is not empty. The element returned is passed by reference.

$\mathcal{C}$ **insert(**$\mathcal{E}$ e**).** Insert element e into Q and return its compartment for later use. **Requirement.** There is space available to accomplish this operation. Both e and the returned compartment are passed by reference.

void delete-min(). Remove a minimum element and its compartment from Q. **Requirement.** The data structure is not empty.

void delete($\mathcal{C}$ x**).** Remove both the element stored at compartment x and compartment x from Q. **Requirement.** x is a valid compartment. x is passed by reference.

Another method that may be considered is:

void decrease($\mathcal{C}$ x, $\mathcal{E}$ e). Replace the element stored at compartment x with element e. **Requirement.** x is a valid compartment. e is no greater than the old element stored at x. Both x and e are passed by reference.

Some additional methods — like a constructor, a destructor, and a set of methods for examining the number of elements stored in Q — are necessary to make the data structure useful, but these are computationally less interesting and therefore not considered here.

We would like to point out that, after inserting an element, the reference to the compartment where it is stored should remain the same so that possible later references made by delete and decrease operations are valid. In some sources this problem is not acknowledged, meaning that the proposed algorithms are actually incorrect. Our solution to this potential problem is simple: we do not move the elements after they have been inserted into the data structure. For other solutions, we refer to a longer discussion in [16].

In a tree its *nodes* are used as compartments for storing the elements. A *binomial tree* [1, 24, 27] 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 comprises the root and its r binomial subtrees of rank 0, 1, ..., $r - 1$ in this order. We call the root of the subtree of rank 0 the *oldest child* and the root of the subtree of rank $r - 1$ the *youngest child*. It follows directly from the definition that the size of a binomial tree is always a power of two, and that the *rank* of a tree of size 2^r is r.

A binomial tree can be implemented using the *child-sibling representation*, where every node has three pointers, one pointing to its youngest child, one to its closest younger sibling, and one to its closest older sibling. The children of a node are kept in a circular, doubly-linked list, called the *child list*, so one of the sibling pointers of the youngest child points to the oldest child, and vice versa. Unused child pointers have the value null. In addition, each node should store the rank of the maximal subtree rooted at it. To facilitate the delete method, every node should have space for a parent pointer, but the parent pointer is set only if the node is the youngest child of its parent. To distinguish the root from the other nodes, its parent pointer is set to point to a fixed sentinel; for other nodes the parent pointer points to another node or has the value null.

The children of a node can be sequentially accessed by traversing the child list from the youngest to the oldest, or vice versa if the oldest child is first accessed via the youngest child. It should be pointed out that with respect to the parent pointers our representation is nonstandard. An argument, why one parent pointer per child list is enough and why we can afford to visit all younger siblings of a node to get to its parent, is given in Lemma 1. In our representation each node has a constant number of pointers pointing to it, and it knows from which nodes those pointers come. Because of this, it is possible to detach any node by updating only a constant number of pointers.

In its standard form, a *binomial queue* is a forest of binomial trees with at most one tree of any given rank. In addition, the trees are kept *heap ordered*,

i.e. the element stored at every node is no greater than the elements stored at the children of that node. The sibling pointers of the roots are reused to keep the trees in a circular, doubly-linked list, called the *root list*, where the binomial trees appear in increasing order of rank.

Two binomial trees of the same rank can be linked together by making the root of the tree that stores the greater element the youngest child of the other root. Later on, we refer to this as a *join*. A *split* is the inverse of a join, where the subtree rooted at the youngest child of the root is unlinked from the given tree. A join involves a single element comparison, and both a join and a split have a cost of $O(1)$.

Let B be a binomial queue. The priority-queue operations for B can be implemented as follows:

B.**find-min().** The root storing a minimum element is accessed and that element is returned. The other operations are given the obligation to maintain a pointer to the location of the current minimum.

B.**insert(**e**).** A new node storing element e is constructed and then added to the forest as a tree of rank 0. If this results in two trees of rank 0, successive joins are performed until no two trees have the same rank. Furthermore, the pointer to the location of the current minimum is updated if necessary.

B.**delete-min().** The root storing an overall minimum element is removed, thus leaving all the subtrees of that node as independent trees. In the set of trees containing the new trees and the previous trees held in the binomial queue, all trees of equal ranks are joined until no two trees of the same rank remain. The root storing a new minimum element is then found by scanning the current roots and the pointer to the location of the current minimum is updated accordingly.

B.**delete(**x**).** The binomial tree containing node x is traversed upwards starting from x, the current node is swapped with its parent, and this is repeated until the root of the tree is reached. Note carefully that nodes are swapped by detaching them from their corresponding child lists and attaching them back in each others place. Since whole nodes are swapped, pointers to the nodes from the outside remain valid. Lastly, the root is removed as in a delete-min operation.

B.**decrease(**x, e**).** The element stored at node x is replaced with element e and node x is repeatedly swapped with its parent until the heap order is reestablished. Also, the pointer to the location of the current minimum is updated if necessary.

For a binomial queue storing n elements, the worst-case cost per find-min is $O(1)$ and that per insert, delete-min, delete, and decrease is $O(\log n)$. The amortized bound on the number of element comparisons is two per insert and $2\log n + O(1)$ per delete-min. To see that the bound is tight for delete-min (and delete), consider a binomial queue of size n which is one less than a power of two, an operation sequence which consists of pairs of delete-min and insert, and a situation where the element to be deleted is always stored

at the root of the tree of the largest rank. Every delete-min operation in such a sequence needs $\lfloor \log n \rfloor$ element comparisons for joining the trees of equal ranks and $\lfloor \log n \rfloor$ element comparisons for finding the root that stores a new minimum element.

To get the worst-case cost of $O(1)$ for an insert operation, all the necessary joins cannot be performed at once. Instead, a constant number of joins can be done in connection with each insertion, and the execution of the other joins can be delayed for forthcoming insert operations. To facilitate this, a logarithmic number of pointers to joins in process is maintained on a stack. More closely, each pointer points to a root in the root list; the rank of the tree pointed to should be the same as the rank of its neighbour. 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. Thereby a preference is given for joins involving small trees.

In an insert operation a new node is created and 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 to point to the newly created node. If there exists another tree of rank 0, a pointer to this pair of trees is pushed on the stack. After this a constant number of join steps is executed. If one join is done in connection with every insert operation, the on-going joins are already disjoint and there are always space for new elements (for a similar treatment, see [5] or [6, p. 53 ff.]). Analogously with an observation made in [5], the size of the stack can be reduced dramatically if two join steps are executed in connection with every insert operation, instead of one.

Since there are at most two trees of any given rank, the number of element comparisons performed by a delete-min and delete operation is never larger than $3 \log n$. In fact, a tighter analysis shows that the number of trees is bounded by $\lfloor \log n \rfloor + 1$. The argument is that insert, delete-min, and delete operations can be shown to maintain the invariant that any rank occupying two trees is preceded by a rank occupying no tree, possibly having a sequence of ranks occupying one tree in between. That is, the number of element comparisons is only at most $2 \log n + O(1)$ per delete-min and delete. An alternative way of achieving the worst-case bounds, two element comparisons per insert and $2 \log n + O(1)$ element comparisons per delete-min/delete, is described in [8, Section 3].

3. Two-tier framework

For the binomial queues there are two major tasks that contribute to the multiplicative factor of two in the bound on the number of element comparisons for delete-min. The first is the join of trees of equal ranks, and the second is the maintenance of the pointer to the location of a minimum

element. The key idea of our framework is to reduce the number of element comparisons involved in finding a new minimum element after the joins.

To realize this idea we compose a priority queue using the following three components, which themselves are priority queues:

1. The *lower store* is a priority queue which stores at least half of all of the n elements. This store is implemented as a collection of separate structures, the size of each of which is an exact power of two. Each element is stored only once, and there is no relation between elements held in different structures. A special requirement for delete-min and delete is that they only modify one of the structures and at the same time retain the size of that structure. In addition to the normal priority-queue methods, *structure borrowing* should be supported in which an arbitrary structure can be released from the lower store (and moved to the reservoir if this becomes empty). As to the complexity requirements, find-min and insert should have a cost of $O(1)$, and delete-min and delete a cost of $O(\log n)$. Moreover, structure borrowing should have a cost of $O(1)$.
2. The *upper store* is a priority queue which stores pointers to the m structures in the lower store, each giving one minimum candidate. In pointer comparisons, the candidates referred to are compared. The main purpose of the upper store is to provide fast access to an overall minimum element in the lower store. The requirement is that find-min and insert should have a cost of $O(1)$, and delete-min and delete a cost of $O(\log m)$.
3. The *reservoir* is a special priority queue which supports find-min, delete-min, and delete, but not insert. It contains the elements that are not in the lower store. Whenever a compartment together with the associated element is deleted from the lower store, as a result of a delete-min or delete operation, a *compartment* is *borrowed* from the reservoir. Using this borrowed compartment, the structure that lost a compartment can be readjusted to gain the same properties as before the deletion. Again, find-min should have a cost of $O(1)$, and delete-min and delete a cost of $O(\log n)$, where n is the number of *all* elements stored. In other words, the cost need only be logarithmic in the size of the reservoir at the time when the reservoir is refilled by borrowing a structure from the lower store. Moreover, compartment borrowing should have a cost of $O(1)$.

To get from the lower store to the upper store and from the upper store to the lower store, we assume that each structure in the lower store is linked to the corresponding pointer in the upper store, and vice versa. Moreover, to distinguish whether a compartment is in the reservoir or not, we assume that each structure has extra information indicating the component in which the structure is held, and that this information can easily be reached from each compartment.

Let I be an implementation-independent framework interface for a priority queue. Using the priority-queue operations provided for the components, the

priority-queue operations for I can be realized as follows:

I.**find-min().** A minimum element is either in the lower store or in the reservoir, so it can be found by lower-store find-min — which relies on upper-store find-min — or by reservoir find-min. The smaller of these two elements is returned.

I.**insert(e).** The given element e is inserted into the lower store using lower-store insert, which may invoke the operations provided for the upper store.

I.**delete-min().** First, if the reservoir is empty, a group of elements is moved from the lower store to the reservoir using structure borrowing. Second, lower-store find-min and reservoir find-min are invoked to determine in which component an overall minimum element lies. Depending on the outcome, lower-store delete-min or reservoir delete-min is invoked. If an element is to be removed from the lower store, another element is borrowed from the reservoir to retain the size of the modified lower-store structure. Depending on the changes made in the lower store, it may be necessary to update the upper store as well.

I.**delete(x).** It is first made sure that the reservoir is not empty; if it is, it is refilled by borrowing a structure from the lower store. The extra information, associated with the structure in which the given compartment x is stored, is accessed. If the compartment is in the reservoir, reservoir delete is invoked; otherwise, lower-store delete is invoked. In lower-store delete, a compartment is borrowed from the reservoir to retain the size of the modified structure. If necessary, the upper store is updated as well.

Assume now that the given complexity requirements are fulfilled. Since a lower-store find-min operation and a reservoir find-min operation have a cost of $O(1)$, a find-min operation has a cost of $O(1)$. The efficiency of an insert operation is directly related to that of a lower-store insert operation, i.e. the cost is $O(1)$. In a delete-min operation the cost of the find-min and insert operations invoked is only $O(1)$. Also, compartment borrowing and structure borrowing have a cost of $O(1)$. Let n denote the number of elements stored, and let $D_\ell(n)$, $D_u(n)$, and $D_r(n)$ be the functions expressing the complexity of lower-store delete-min, upper-store delete-min, and reservoir delete-min, respectively. Hence, the complexity of a delete-min operation is bounded above by $\max\{D_\ell(n) + D_u(n), D_r(n)\} + O(1)$. As to the efficiency of a delete operation, there is a similar dependency on the efficiency of lower-store delete, upper-store delete, and reservoir delete. The number of element comparisons performed can be analysed after the actual realization of the components is detailed.

4. Two-tier binomial queues

In our first realization of the framework we use binomial trees as the basic structures, and utilize binomial queues in the form described in Section 2.

Therefore, we call the data structure *two-tier binomial queue*. Its components are the following:

1. The lower store is implemented as a binomial queue storing the major part of the elements.
2. The upper store is implemented as another binomial queue that stores pointers to the roots of the binomial trees held in the lower store, but the upper store may also store pointers to earlier roots of the lower store that are currently either in the reservoir or inner nodes in the lower store.
3. The reservoir consists of a single tree, which is binomial at the time of its creation.

The form of the nodes is identical in the lower store and the reservoir, and each node is linked to the corresponding node in the upper store; if no counterpart in the upper store exists, the link has the value null. Also, we use the convention that the parent pointer of the root of the reservoir points to a reservoir sentinel, whereas for the trees held in the lower store the parent pointers of the roots point to a lower-store sentinel. This way we can easily distinguish the origin of a root. Instead of compartments and structures, nodes and subtrees are borrowed by exchanging references to these objects. We refer to these operations as *node borrowing* and *tree borrowing*.

If there are n elements in total, the size of the upper store is $O(\log n)$. Therefore, at the upper store, delete-min and delete require $O(\log \log n)$ element comparisons. The challenge is to maintain the upper store and to implement the priority-queue operations for the lower store such that the work done in the upper store is reduced. If in the lower store the removal of a root is implemented in the standard way, there might be a logarithmic number of new subtrees that need to be inserted into the upper store. Possibly, some of the new subtrees have to be joined with the existing trees, which again may cascade a high number of deletions to the upper store. Hence, as required, a new implementation of the removal of a root is introduced that alters only one of the lower-store trees.

Next, we show how different priority-queue operations may be handled. We describe and analyse the operations for the reservoir, the upper store, and the lower store in this order.

4.1 Reservoir operations

To borrow a node from the tree of the reservoir, the oldest child of the root is detached (or the root itself if it does not have any children), making the children of the detached node the oldest children of the root in the same order. Due to the circularity of the child list, the oldest child and its neighbouring nodes can be accessed by following a few pointers. So the oldest child can be detached from the child list at a cost of $O(1)$. Similarly, two child lists can be appended at a cost of $O(1)$. To sum up, the total cost of node borrowing is $O(1)$.

A find-min operation simply returns the element stored at the root of the tree held in the reservoir. That is, the worst-case cost of a find-min operation is $O(1)$.

In a delete-min operation, the root of the tree of the reservoir is removed and the subtrees rooted at its children are *repeatedly joined* by processing the children of the root from the oldest to the youngest. In other words, every subtree is joined with the tree which results from the joins of the subtrees rooted at the older children. In a delete operation, the given node is repeatedly swapped with its parent until the root is reached, the root is removed, and the subtrees of the removed root are repeatedly joined. In both delete-min and delete, when the removed node has a counterpart in the upper store, the counterpart is deleted as well.

For the analysis, the invariants proved in the following lemmas are crucial. For a node x in a rooted tree, let A_x be the set of ancestors of x, including x itself; let C_x be the number of all the siblings of x that are younger than x, including x; and let D_x be $\sum_{y \in A_x} C_y$.

Lemma 1. *For any node x in a binomial tree of rank r, $D_x \leq r + 1$.*

Proof. The proof is by induction. Clearly, the claim is true for a tree consisting of a single node. Assume that the claim is true for two trees T_1 and T_2 of rank $r - 1$. Without loss of generality, assume that the root of T_2 becomes the root after T_1 and T_2 are joined together. For every node x in T_1, D_x increases by one due to the new root. For every node y in T_2, except the root, D_y increases by one because the only ancestor of y that gets a new younger sibling is the child of the new root. Now the claim follows from the induction assumption. □

Lemma 2. *Consider any node x of the tree held in the reservoir. Starting with a binomial tree of rank r, D_x never gets larger than $r + 1$ during the life-span of this tree.*

Proof. By Lemma 1, the initial tree fulfils the claim. Node borrowing modifies the tree in the reservoir by removing the oldest child of the root and moving all its children one level up. For every node x in any of the subtrees rooted at the children of the oldest child of the root, D_x will decrease by one. For all other nodes the value remains the same. Hence, if the claim was true before borrowing, it must also be true after the modifications.

Each delete-min and delete operation removes the root of the tree in the reservoir and repeatedly joins the resulting subtrees. Due to the removal of the root, for every node x, D_x decreases by one. Moreover, since the subtrees are made separate, if there are j subtrees in all, for any node y in the subtree rooted at the ith oldest child (or simply the ith subtree), $i \in \{1, \ldots, j\}$, D_y decreases by $j - i$. A join increases D_x by one for every node x in the subtrees involved, except that the value remains the same for the root. Therefore, since a node x in the ith subtree is involved in $j - i + 1$ joins, D_x may increase at most by $j - i + 1$. To sum up, for every node x, D_x may only decrease or stay the same. Hence, if the claim was true before the root removal, it must also be valid after all the modifications. □

Corollary 3. *During the life-span of the tree held in the reservoir, starting with a binomial tree of rank r, the root of the tree has at most r children.*

Proof. For a node x, let d_x denote the number of children of x. Let y be the root of the tree held in the reservoir and z the oldest child of y. Clearly, $D_z = d_y + 1$. By Lemma 2, $D_z \leq r + 1$ all the time, and thus, $d_y \leq r$. □

The complexity of a delete-min and delete operation is directly related to the number of children of the root, and the complexity of a delete operation is also related to the length of the D_x-path for the node x being deleted. If the rank of the tree in the reservoir was initially r, by Corollary 3 the number of children of the root is always smaller than or equal to r, and by Lemma 2 the length of the D_x-path is bounded by r. During the life-span of the tree held in the reservoir, there is another binomial tree in the lower store whose rank is at least r (see Section 4.3). Thus, if n denotes the number of elements stored, $r < \log n$. The update of the upper store, if at all necessary, has an extra cost of $O(\log \log n)$, including $O(\log \log n)$ element comparisons. Hence, the worst-case cost of a delete-min and delete operation is $O(\log n)$ and the number of element comparisons performed is at most $\log n + O(\log \log n)$.

4.2 Upper-store operations

The upper store is a worst-case efficient binomial queue storing pointers to the nodes held in the other two components. In addition to the standard priority-queue methods, it supports lazy deletions where nodes are marked to be deleted instead of being removed immediately. It should also be possible to unmark a node if the node pointed to by the stored pointer becomes a root later on. The invariant maintained by the algorithms is that for each marked node, whose pointer refers to a node y in the lower store or in the reservoir, 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 use the global-rebuilding technique adopted from [23]. 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 need to handle insertions, the trees there can be emptied as in the reservoir by detaching the oldest child of the root in question, or the root itself if it does not have any children. If there are several trees left, if possible, a tree whose root does not contain the current minimum is selected as the target of each detachment.

In connection with each of the next at most $m_0/4$ upper-store operations, four nodes are detached 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.

Since the cost of each detachment and insertion is $O(1)$, the reorganization only adds an additive term $O(1)$ to the cost of all upper-store operations. A find-min operation, which is a normal binomial-queue operation, may need to consult both the old and the new upper stores, so its worst-case cost is still $O(1)$. The actual cost of marking and unmarking is clearly $O(1)$, even if they take part in reorganizations. If m denotes the total number of unmarked nodes currently stored, at any point in time, the total number of nodes stored is $\Theta(m)$, and during a reorganization $m_0 = \Theta(m)$. According to our earlier analysis, in the old structure the efficiency of delete-min and delete operations depends on the original size m_0. In the new structure their efficiency depends on the current size m. Therefore, since delete-min and delete operations are handled normally, except that they may take part in reorganizations, each of them has the worst-case cost of $O(\log m)$ and may perform at most $2 \log m + O(1)$ element comparisons.

4.3 Lower-store operations

A find-min operation simply invokes upper-store find-min and then follows the received pointer to the root storing a minimum element. Clearly, the worst-case cost of a find-min operation is $O(1)$.

An insert operation is accomplished, in a worst-case efficient manner, as described in Section 2. As a result of joins, some roots of the trees in the lower store are linked to other roots, so the corresponding pointers should be deleted from the upper store. Instead of using upper-store delete, lazy deletion is applied. The worst-case cost of each join is $O(1)$ and the worst-case cost of each lazy deletion is also $O(1)$. Since each insert operation only performs a constant number of joins and lazy deletions, its worst-case cost is $O(1)$.

Prior to each delete-min and delete operation, it is checked whether a reservoir refill is necessary. If the reservoir is empty, a tree of the highest rank is taken from the lower store. If the tree is of rank 0, it is moved to the reservoir and the corresponding pointer is deleted from the upper store. This special case when $n = 1$ can be handled at a cost of $O(1)$. In the normal case, the tree taken is split into two halves, and the subtree rooted at the youngest child is moved to the reservoir. The other half is kept in the lower store. However, if after the split the lower store contains another tree of the

same rank as the remaining half, the two trees are joined and the pointer to the root of the loser tree is to be deleted from the upper store. Again, lazy deletion is applied. A join has a cost of $O(1)$ and involves one element comparison. As shown, each lazy deletion has a cost of $O(1)$, including also some element comparisons. That is, the total cost of tree borrowing is $O(1)$.

In a delete-min operation, after a possible reservoir refill the root storing a minimum element is removed, a node from the reservoir is borrowed, and the borrowed node — seen as a tree of rank 0 — is repeatedly joined with the subtrees of the removed root. This results in a new binomial tree with the same structure as before the deletion. In the upper store, a pointer to the new root of the resulting tree is inserted and the pointer to the old root is deleted. However, if the pointer to the root already exists in the upper store, the upper-store node containing that pointer is simply unmarked. In a delete operation, after a possible reservoir refill the given node is swapped to the root as in a delete operation for a binomial queue, after which the root is deleted as in a delete-min operation.

As analysed earlier, tree borrowing and node borrowing have the worst-case cost of $O(1)$. Also, the removal of a root has the worst-case cost of $O(1)$. The at most $\lfloor \log n \rfloor$ joins executed have the worst-case cost of $O(\log n)$, and the number of element comparisons performed is at most $\log n$. The upper-store update has an additional cost of $O(\log \log n)$, including $O(\log \log n)$ element comparisons. To summarize, the worst-case cost of a delete-min operation is $O(\log n)$ and the number of element comparisons performed is at most $\log n + O(\log \log n)$. As to a delete operation, since in a binomial tree of size n the length of any D_x-path is never longer than $\log n$, node swapping has the worst-case cost of $O(\log n)$, but involves no element comparisons. Therefore, the complexity of a delete operation is the same as that of a delete-min operation.

4.4 Summing up the results

Using the components described and the complexity bounds derived, the efficiency of the priority-queue operations supported by the framework interface can be summed up as follows:

Theorem 4. *Let n be the number of elements stored in the data structure prior to each priority-queue operation. A two-tier binomial queue guarantees the worst-case cost of $O(1)$ per find-min and insert, and the worst-case cost of $O(\log n)$ with at most $\log n + O(\log \log n)$ element comparisons per delete-min and delete.*

The bound on the number of element comparisons for delete-min and delete can be further reduced. Instead of having two levels of priority queues, we can have several levels. At each level, except the highest one, delete-min and delete operations are carried out as in our earlier lower store relying on a reservoir; and at each level, except the lowest one, lazy deletions are carried out as in our earlier upper store. Except for the highest level, the constant factor in the logarithm term expressing the number of element comparisons

performed per delete-min or delete is one. Therefore the total number of element comparisons performed in all levels is at most $\log n + \log\log n + \ldots + O(\log^{(k)} n)$, where $\log^{(k)}$ denotes the logarithm function applied k times and k is a constant representing the number of levels. An insertion of a new element would result in a constant number of insertions and lazy deletions per level. Hence, the number of levels should be a fixed constant to achieve a constant cost for insertions.

5. Multipartite binomial queues

In this section we present a refinement of a two-tier binomial queue, called a *multipartite binomial queue*. To refine the previous construction, the following modifications are significant:

1. The lower store is divided into three components: main store, insert buffer, and floating tree. The *main store* is maintained as the lower store in our earlier construction. However, all insert operations are directed to the *insert buffer* which is a binomial queue maintained in a worst-case efficient manner. When the insert buffer becomes too large, a subtree is cut off from one of its trees and used as a cutting for the *floating tree*. The floating tree is *incrementally united* with the existing trees in the main store in connection with each modifying operation. That is, joins make the floating tree larger, and when uniting is complete, the floating tree becomes part of the main store. At any point in time, we ensure that the size of the insert buffer is logarithmic in the total number of elements stored. We also ensure that uniting will be finished before it will be necessary to create a new floating tree.
2. The upper store is implemented as a circular, doubly-linked list; there is one node per tree held in the main store. Each node contains a pointer to the root of the tree which stores a minimum element among the elements stored in the trees having a lower rank, including the tree itself. We call these pointers the *prefix-minimum pointers*.
3. A reservoir is still in use and all the reservoir operations are performed as previously described, but to refill it a subtree is borrowed from the insert buffer and if the insert buffer is empty from the main store. Borrowing is never necessary from the floating tree since during the whole uniting process the insert buffer will be large enough to service the refills which may be needed.

The nodes in the main store, insert buffer, floating tree, and reservoir may be moved from one component to another, so the form of the nodes must be identical in all four components.

In the improved construction the key idea is to balance the work done in the main store and the upper store. After using $r + O(1)$ element comparisons to readjust a tree of rank r in the main store, only $\log n - r + O(1)$ element comparisons are used for the maintenance of the upper store. Another important idea is to unite the floating tree to the main store such that

all the involved components, the main store, insert buffer, floating tree, and reservoir, are fully operational during the uniting operation.

5.1 Description of the components

An upper-store find-min operation provides a minimum element in the main store by following the prefix-minimum pointer for the tree of the highest rank. Thus, a find-min operation has the worst-case cost of $O(1)$. To delete a pointer corresponding to a tree of rank r from the upper store, the node in question is found by a sequential scan and thereafter removed, and the prefix-minimum pointers are updated for all trees having a rank higher than r. The total cost is proportional to $\log n$ and one element comparison per higher-rank tree is necessary, meaning at most $\log n - r + O(1)$ element comparisons. When the element stored at the root of a tree of rank r is changed, the prefix-minimum pointers can be updated in a similar manner. The complexity of such a change is the same as that of a delete operation. To insert a pointer corresponding to a tree of rank r, as done in the uniting process, a sequential scan has to be done to find the correct insertion point. It may happen that there already exists a tree of the same rank. Therefore, these trees must be joined and this join may propagate to all the higher ranks. In addition, the prefix-minimum pointers must be updated, but this can be done simultaneously with the joins, if at all necessary, so that each of the higher-rank trees is considered only once. Hence, the worst-case cost of an insert operation is proportional to $\log n$, and at most $2(\log n - r) + O(1)$ element comparisons are performed.

The main store is a binomial queue that is maintained as our earlier lower store, except that the main store and the upper store interact in another way. Tree borrowing is done almost as before. First, the tree of the highest rank is taken from the main store, the tree is split, one half of it is moved to the reservoir, and the other half is kept in the main store; the latter half is then joined with another tree of the same rank if there is any. Second, the prefix-minimum pointers for the trees of the two highest ranks are updated. The total cost of all these modifications is $O(1)$. From this and our earlier analysis, it follows that the worst-case cost of tree borrowing is $O(1)$.

Since no normal insertions are done in the main store, no lazy deletions are forwarded to the upper store. Because of node borrowing, only one tree need to be modified in a delete-min or delete operation. If the rank of the modified tree is r, both main-store operations have the worst-case cost of $O(r)$ and require at most $r + O(1)$ element comparisons. After the adjustment in the main store, the prefix-minimum pointers need to be updated in the upper store for all trees having a rank higher than or equal to r. This has an additional cost proportional to $\log n$, and $\log n - r + O(1)$ element comparisons may be necessary. To summarize, the worst-case cost of a delete-min and delete operation is $O(\log n)$ and never more than $\log n + O(1)$ element comparisons are performed.

The insert buffer is implemented as a worst-case efficient binomial queue.

Hence, an insert operation has the worst-case cost of $O(1)$. To reduce the size of the insert buffer or to refill the reservoir, tree borrowing is used as for the main store. Observe that after tree borrowing no change to the pointer indicating the location of the current minimum of the insert buffer is necessary. As will be shown, the invariants maintained guarantee that the insert buffer will never become larger than $c_1 \log n + c_2$ for some constants c_1 and c_2. Therefore, a delete-min and delete operation has the worst-case cost of $O(\log \log n)$ and performs at most $2 \log(c_1 \log n + c_2) + O(1)$ element comparisons, which is bounded by $\log n + O(1)$ for all $n \geq 0$.

The floating tree is maintained as the trees in our earlier lower store. The main point is that delete-min and delete operations should retain the size of this tree. Therefore, when a root is removed, a node from the reservoir is borrowed. Our earlier analysis implies that the worst-case cost of a delete-min and delete operation is $O(\log n)$, and the number of element comparisons performed is at most $\log n + O(1)$.

5.2 Interactions between the components

We let the priority-queue operations change the data structure in *phases*. Let n_0 denote the total number of elements at the beginning of a phase. All operations are made aware of the current phase using n_0, $\lfloor \log n_0 \rfloor$, and a single counter. To avoid the usage of the whole-number logarithm function, $\lfloor \log n_0 \rfloor$ can be calculated by maintaining the interval $[2^k \,..\, 2^{k+1})$ in which n_0 lies. When n_0 moves outside the interval, the logarithm and the interval are updated accordingly.

When performing the priority-queue operations the following invariants are maintained:

1. In a phase, exactly $\lfloor \log n_0 \rfloor$ *modifying* operations — insert, delete-min, or delete — are executed.
2. The number of elements in the floating tree is no smaller than $\log n_0$ if the tree exists, i.e. at least $\log n_0$ elements are extracted from the insert buffer if an extraction is done.
3. At the beginning of the phase, the insert buffer contains no more than $\max\{24, 9 \log n_0\}$ elements, i.e. the insert buffer never gets too large.
4. At the beginning of the phase, there is no floating tree, i.e. the floating tree from the previous phase, if any, has been successfully united to the main store.

The first invariant is forced by the protocol used for handling the priority-queue operations. Initially, the other invariants are valid since all the components are empty. The first and third invariants imply that, for all $n \geq 0$, the insert buffer never gets larger than $c_1 \log n + c_2$ for some constants c_1 and c_2.

At the beginning of a phase, the first modifying operation executes a *preprocessing step* prior to its actual task in order to make the insert buffer smaller, if necessary. If the size of the insert buffer is larger than $8 \log n_0$, half of a tree of the highest rank is borrowed and used to form a new floating

tree. On the other hand, if the size of the insert buffer is smaller than or equal to $8 \log n_0$, no changes are made to the insert buffer and no floating tree is created. Next we analyse the consequences of the preprocessing step.

Let us assume that, when a tree is borrowed in the preprocessing step, in the insert buffer a tree of the highest rank is of size 2^k. This tree would be the smallest possible if, for all $i \in \{0, 1, \ldots, k\}$, there existed two trees of size 2^i. Then the number of elements stored in the insert buffer would be $2^{k+2} - 2$. Since in the insert buffer there are at least $\lceil 8 \log n_0 \rceil - 1$ elements, $\lceil 8 \log n_0 \rceil - 1 \leq 2^{k+2} - 2$, from which it follows that $\lceil \log n_0 \rceil \leq 2^{k-1}$. Since the size of the borrowed tree is 2^{k-1}, the size of the floating tree must be at least $\lceil \log n_0 \rceil$ at the time of its creation. During the uniting process, the floating tree can only become larger, so the second invariant is established.

Assume that a phase involves i_0 insertions, $0 \leq i_0 \leq \lfloor \log n_0 \rfloor$, and d_0 deletions, $0 \leq d_0 \leq \lfloor \log n_0 \rfloor$. Let b_0 denote the number of elements stored in the insert buffer at the beginning of the phase, and n_1 the total number of elements at the end of the phase. To analyse the effect of the preprocessing step on the size of the insert buffer, we consider five cases:

Case 1. $n_0 \leq 24$. Since $b_0 \leq n_0$, $b_0 < 8 \log n_0$ for all $0 \leq b_0 \leq n_0 \leq 24$. Thus, $b_0 + \lfloor \log n_0 \rfloor < 9 \log n_0$. If $n_1 > n_0$, we are done. Otherwise, if $n_1 \leq n_0$, then $n_1 \leq 24$ and hence the size of the insert buffer is less than 24 at the end of the phase.

Case 2. $n_0 > 24$, $i_0 \geq d_0$, and $b_0 \leq 8 \log n_0$. Since $i_0 \geq d_0$, $n_1 \geq n_0$. If $b_0 \leq 8 \log n_0$, the size of the insert buffer must be bounded by $9 \log n_0 \leq 9 \log n_1$ at the end of the phase.

Case 3. $n_0 > 24$, $i_0 \geq d_0$, and $b_0 > 8 \log n_0$. Since $b_0 > 8 \log n_0$, at least $\log n_0$ elements must have been extracted from the insert buffer in the preprocessing step. Therefore, at the end of the phase the insert buffer cannot be larger than b_0. Since $i_0 \geq d_0$, it must be that $n_1 \geq n_0$. So if $b_0 \leq 9 \log n_0$ at the beginning of the phase, the size of the insert buffer must be bounded by $9 \log n_1$ at the end of the phase.

Case 4. $n_0 > 24$, $i_0 < d_0$, and $b_0 \leq 8 \log n_0$. Since $i_0 < d_0$, at most half of the modifying operations have been insertions. Moreover, it must be true that $n_1 \geq n_0 - \lfloor \log n_0 \rfloor$. Since $8.5 \log n_0 \leq 9 \log(n_0 - \lfloor \log n_0 \rfloor)$ for all $n_0 > 24$, the insert buffer cannot be larger than $9 \log n_1$ at the end of the phase. The above-mentioned inequality is easy to verify for n_0 larger than 2^{18}. We used a computer to verify it for all integers in the range $\{25, 26, \ldots, 2^{18}\}$.

Case 5. $n_0 > 24$, $i_0 < d_0$, and $b_0 > 8 \log n_0$. Again, since $i_0 < d_0$, at most half of the modifying operations involved in the phase have been insertions. Since $b_0 > 8 \log n_0$, at least $\log n_0$ elements must have been extracted from the insert buffer in the preprocessing step. Thus, the insert buffer cannot be larger than $b_0 - \log n_0 + (1/2) \lfloor \log n_0 \rfloor$ at the end of the phase. This means that its size must be bounded by $8.5 \log n_0$. As in Case 4, $n_1 \geq n_0 - \lfloor \log n_0 \rfloor$. So by the same argument as in Case 4, the input buffer cannot be larger than $9 \log n_1$ at the end of the phase.

In particular, note that reservoir refills only make the insert buffer smaller, so these cannot cause any harm. In conclusion, if the insert buffer was not larger than $\max\{24, 9\log n_0\}$ at the beginning of the phase, it cannot be larger than $\max\{24, 9\log n_1\}$ at the end of the phase. Thus, the third invariant is established.

Basically, to unite the floating tree and the main store a normal insert operation for binomial queues is executed, except that the insertion starts from a rank higher than 0. In the worst case, uniting may involve logarithmically many joins so it is done incrementally. This means that the prefix-minimum pointers are not necessarily valid for trees whose rank is higher than the rank of the tree up to which the uniting process has advanced. To solve the problem, each find-min operation should consult two trees; one referred to by the prefix-minimum pointer for the tree up to which the uniting process has advanced and the other referred to by the prefix-minimum pointer for the tree of the highest rank.

The main store can have at most $\lceil \log(n_0 - \lfloor 8\log n_0 \rfloor) \rceil$ trees when a new floating tree is created, so this is the highest rank before uniting. Of course, the trees having a rank lower than the rank of the given tree can be skipped. Since $\lceil \log(n_0 - \lfloor 8\log n_0 \rfloor) \rceil \leq \lfloor \log n_0 \rfloor$ for all positive n_0, at most one tree need to be visited in connection with each modifying operation. At each visit, one join step is executed and the corresponding prefix-minimum pointer updated. At this speed, the uniting process will be finished before the end of the phase is reached, which establishes the fourth invariant.

Now that we have proved the correctness of the invariants, we can analyse the efficiency of the priority-queue operations. The overhead caused by the phase management and the preprocessing step is only $O(1)$ per modifying operation. Also, incremental uniting will only increase the cost of modifying operations by an additive constant.

In a find-min operation, the four components storing elements — main store, reservoir, floating tree, and insert buffer — need to be consulted. Since all these components support a find-min operation at the worst-case cost of $O(1)$, the worst-case cost of a find-min operation is $O(1)$. Insertions only involve the insert buffer so, from the bound derived for worst-case efficient binomial queues and the fact that the extra overhead per insert is $O(1)$, the worst-case cost of $O(1)$ directly follows.

Each delete-min operation refills the reservoir if necessary, determines in which component an overall minimum element is stored, and thereafter invokes the corresponding delete-min operation provided for that component. According to our earlier analysis, each of the components storing elements supports a delete-min operation at the worst-case cost of $O(\log n)$, including at most $\log n + O(1)$ element comparisons. Even with other overheads, the bounds are the same.

In a delete operation, the root is consulted to determine which of the delete operations provided for the components storing elements should be invoked. The traversal to the root has the worst-case cost of $O(\log n)$, but even with this and other overheads, a delete operation has the worst-case

cost of $O(\log n)$ and performs at most $\log n + O(1)$ element comparisons as shown earlier.

To conclude, we have proved the following theorem.

Theorem 5. *Let n be the number of elements stored in the data structure prior to each priority-queue operation. A multipartite binomial queue guarantees the worst-case cost of $O(1)$ per find-min and insert, and the worst-case cost of $O(\log n)$ with at most $\log n + O(1)$ element comparisons per delete-min and delete.*

6. Application: adaptive heapsort

A sorting algorithm is adaptive if it can sort all input sequences and performs particularly well for sequences having a high degree of existing order. The cost consumed is allowed to increase with the amount of disorder in the input. In the literature many adaptive sorting algorithms have been proposed and many measures of disorder considered (for a survey, see [11] or [22]). In this section we consider adaptive heapsort, introduced by Levcopoulos and Petersson [21], which is one of the simplest adaptive sorting algorithms. As in [21], we assume that all input elements are distinct.

At the commencement of adaptive heapsort a Cartesian tree is built from the input sequence. Given a sequence $X = \langle x_1, \ldots, x_n \rangle$, the corresponding *Cartesian tree* [28] is a binary tree whose root stores element $x_i = \min\{x_1, \ldots, x_n\}$, the left subtree of the root is the Cartesian tree for sequence $\langle x_1, \ldots, x_{i-1} \rangle$, and the right subtree is the Cartesian tree for sequence $\langle x_{i+1}, \ldots, x_n \rangle$. After building the Cartesian tree, a priority queue is initialized by inserting the element stored at the root of the Cartesian tree into it. In each of the following n iterations, a minimum element stored in the priority queue is output and thereafter deleted, the elements stored at the children of the node that contained the deleted element are retrieved from the Cartesian tree, and the retrieved elements are inserted into the priority queue.

The total cost of the algorithm is dominated by the cost of the n insertions and n minimum deletions; the cost involved in building [14] and querying the Cartesian tree is linear. The basic idea of the algorithm is that only those elements that can be the minimum of the remaining elements are kept in the priority queue, not all elements. Levcopoulos and Petersson [21] showed that, when element x_i is deleted, the number of elements in the priority queue is no greater than $\lfloor |\mathit{Cross}(x_i)|/2 \rfloor + 2$, where

$$\mathit{Cross}(x_i) = \{ j \mid j \in \{1, \ldots, n\} \text{ and } \min\{x_j, x_{j+1}\} < x_i < \max\{x_j, x_{j+1}\} \}.$$

Levcopoulos and Petersson [21, Corollary 20] showed that adaptive heapsort is optimally adaptive with respect to *Osc*, *Inv*, and several other measures of disorder. For a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ of length n, the

measures *Osc* and *Inv* are defined as follows:

$$Osc(X) = \sum_{i=1}^{n} |Cross(x_i)|$$

$$Inv(X) = |\{(i,j) \mid i \in \{1,2,\ldots,n-1\},\ j \in \{i+1,\ldots,n\}, \text{ and } x_i > x_j\}|.$$

The optimality with respect to the *Inv* measure, which measures the number of pairs of elements that are in wrong order, follows from the fact that $Osc(X) \leq 4Inv(X)$ for any sequence X [21].

Implicitly, Levcopoulos and Petersson showed that using an advanced implementation of binary-heap operations the cost of adaptive heapsort is proportional to

$$\sum_{i=1}^{n} (\log |Cross(x_i)| + 2 \log\log |Cross(x_i)|) + O(n)$$

and that this is an upper bound on the number of element comparisons performed. Using a multipartite binomial queue, instead of a binary heap, we get rid of the log log term and achieve the bound

$$\sum_{i=1}^{n} \log |Cross(x_i)| + O(n).$$

Because the geometric mean is never larger than the arithmetic mean, it follows that our version is optimally adaptive with respect to the measure *Osc*, and performs no more than $n \log (Osc(X)/n) + O(n)$ element comparisons when sorting a sequence X of length n. From this, the bounds for the measure *Inv* immediately follow: the cost is $O(n \log (Inv(X)/n))$ and the number of element comparisons performed is $n \log (Inv(X)/n) + O(n)$. Other adaptive sorting algorithms that guarantee the same bounds are either based on insertionsort or mergesort [10].

7. Multipartite relaxed binomial queues

In this section, our main goal is to extend the repertoire of priority-queue methods to include the decrease method. There are two alternative ways of relaxing the definition of a binomial queue to support a fast decrease operation. In run-relaxed heaps [8], heap-order violations are allowed and a separate structure is maintained to keep track of all violations. In Fibonacci heaps [13] and thin heaps [18], structural violations are allowed; in general, some nodes may have lost some of their children. We tried both approaches; with the former approach we were able to achieve better bounds even though for the best realizations in both categories the difference was only in the lower-order terms.

We use relaxed binomial trees, as defined in [8], as our basic building blocks. Our third priority queue has multiple components and the interactions between the components are similar to those in a multipartite binomial

queue. The main difference is that there is no separate reservoir, but the insert buffer is kept nonempty so it can support node borrowing. Since all components are implemented as run-relaxed binomial queues [8] with some minor variations, we call the resulting data structure a *multipartite relaxed binomial queue*. We describe the data structure in three parts. First, we recall the details of run-relaxed binomial queues, but we still assume that the reader is familiar with the original paper by Driscoll et al. [8], where the data structure was introduced. Second, we show how the upper store is maintained. Third, we explain how the insert buffer and the main store are organized.

The following theorem summarizes the main result of this section.

Theorem 6. *Let n be the number of elements stored in the data structure prior to each priority-queue operation. A multipartite relaxed binomial queue guarantees the worst-case cost of $O(1)$ per find-min, insert, and decrease, and the worst-case cost of $O(\log n)$ with at most $\log n + O(\log \log n)$ element comparisons per delete-min and delete.*

7.1 Run-relaxed binomial queues

A *relaxed binomial tree* [8] is an almost heap-ordered binomial tree where some nodes are denoted to be *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 a decrease operation if the replaced element causes a heap-order violation between the accessed node and its parent. Even though a later priority-queue operation may repair the heap-order violation, the node can still be active. A node remains active until the heap-order violation is explicitly removed. From the definition, it directly 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 these trees, i.e. in the *entire* collection of trees. A *run-relaxed binomial queue* (called a run-relaxed heap in [8]) is a collection of relaxed binomial trees where $\tau \leq \lfloor \log n \rfloor + 1$ and $\lambda \leq \lfloor \log n \rfloor$, n denoting the number of elements stored.

To keep track of the trees in a run-relaxed binomial queue, the roots are doubly linked together as in a binomial queue. To keep track of the active nodes, a *run-singleton structure* is maintained as described in [8]. 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 (see also [3, 19]). Singletons of the same rank are kept in a list. Each entry of the singleton table has a counterpart in a *pair list* if there are more than one singleton of that rank. The youngest 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 [8]. 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 $\lfloor \log n \rfloor$. 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 [8]. The rationale behind the transformations is that, when there are more than $\lfloor \log n \rfloor$ active nodes, there must be at least one pair of active nodes 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 at least one of the transformations — 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. Later on, one application of the transformations together with all necessary changes to the run-singleton structure is referred to as a *λ-reduction*.

Each tree in a run-relaxed binomial queue can be represented in the same way as a normal binomial tree, but to support the transformations used for reducing the number of active nodes some additional data need to be stored at the nodes. In addition to sibling pointers, a child pointer, and a rank, each node should contain a pointer to its parent and a pointer to its occurrence in the run-singleton structure. The occurrence pointer of every nonactive 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 run sentinel; and for all other nodes the pointer gives the occurrence in the run-singleton structure. To support our framework, each node should store yet another pointer to its counterpart in the upper store, and vice versa.

Let us now consider how the priority-queue methods can be implemented. A reader familiar with the original paper by Driscoll et al. [8] should be aware that we have made some minor modifications to the find-min, insert, delete-min, and delete methods to adapt them for our purposes.

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

An insert operation is performed in the same way as in a worst-case efficient binomial queue. As pointed out in Section 2, even if some of the joins are delayed, there can never be more than $\lfloor \log n \rfloor + 1$ trees. From our earlier analysis, it follows that an insert operation has the worst-case cost of $O(1)$ and requires at most two element comparisons.

In delete-min and delete operations, we rely on the same borrowing technique as in [8]: 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 youngest child, the tree rooted at x is split, and if y is active, it is made nonactive and its occurrence is removed from the run-singleton structure. Note that this splitting does not have any effect on the pointer indicating the location of a minimum element, since no nodes are removed.

A delete-min operation has two cases depending on whether one of the roots or one of the active nodes is to be removed. Similarly, a delete operation has two cases depending on whether the given node is a root or not. Next, we consider the two forms of deletions, deletion of a root and deletion of one of the inner nodes, separately.

Let z denote the node being deleted, and assume that z is a root. If the tree rooted at z has rank 0, z is simply 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. Compared to above, each split step is modified such that 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 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 treating the active nodes, but when the delete-min/delete method is embedded into our two-tier framework, it would be too expensive to remove all active children of z in the course of a single delete-min/delete operation.

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 the youngest nodes via the run list and for each such node by following the sibling pointers until a nonactive node is reached.

The computational cost of a delete-min/delete operation, when a root is being deleted, 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 the total cost of these operations is $O(\log n)$. Splits as well as updates to the run-singleton structure do not involve any element comparisons. In total, joins may involve at most $\lfloor \log n \rfloor$ element comparisons. Even though a tree of the smallest rank is split, after the joins the number of trees is at most $\lfloor \log n \rfloor + 1$. Since no new active nodes are created, the number of active nodes is still at most $\lfloor \log n \rfloor$. To find the minimum of $2\lfloor \log n \rfloor + 1$ elements, at most $2\lfloor \log n \rfloor$ element comparisons are to be done. To summarize, this form of a delete-min/delete operation performs at most $3 \log n$ element comparisons.

Assume now that the node z being deleted is an inner 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 operation is completed by updating the pointer to the location of a minimum element. If x is the root of the resulting subtree and a heap-order violation is introduced, node x is made active and the number of active nodes is reduced, if necessary, by performing a λ-reduction once or twice.

Similar to the case of deleting a root, this case has the worst-case cost of $O(\log n)$. If z did not contain the current minimum, only at most $\lfloor \log n \rfloor + O(1)$ element comparisons are done; at most $\lfloor \log n \rfloor$ due to joins and $O(1)$ due to λ-reductions. However, if z contained the current minimum, at most $2\lfloor \log n \rfloor$ additional element comparisons may be necessary. That is, the total number of element comparisons performed is bounded by $3 \log n + O(1)$. To sum up, each delete-min/delete operation has the worst-case cost of $O(\log n)$ and requires at most $3 \log n + O(1)$ element comparisons.

A decrease operation is performed as in [8]. After making the element replacement, it is checked whether the replacement causes a heap-order violation between the given node and its parent. If there is no violation, the operation is complete. Otherwise, the given 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 $\lfloor \log n \rfloor$. 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 given node. All these modifications have the worst-case cost of $O(1)$.

7.2 Upper-store operations

The upper store contains pointers to the roots of the trees held in the insert buffer and in the main store, pointers to all active nodes in the insert buffer and in the main store, and pointers to some earlier roots and active nodes. The number of trees in the insert buffer is at most $\log \log n + O(1)$, the number of trees in the main store is at most $\lfloor \log n \rfloor + 1$, and the number of active nodes is at most $\lfloor \log n \rfloor$. The last property follows from the fact that the insert buffer and the main store share the same run-singleton structure. At any given point in time only a constant fraction of the nodes in the upper store can be marked to be deleted. Hence, the number of pointers is $O(\log n)$.

The upper store is implemented as a run-relaxed binomial queue. In addition to the priority-queue methods find-min, insert, delete-min, delete, and decrease, which are realized as described earlier, 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, in the insert buffer or in the main store, a join is done or an active node is made nonactive by a λ-reduction. In both situations, a normal upper-store deletion would be too expensive.

As in Section 4, global rebuilding will be used to get rid of the marked

nodes when there are too many of them, but for three reasons our earlier procedure is not applicable for run-relaxed binomial queues:

1. Due to parent pointers the oldest child of a root cannot necessarily be detached at a cost of $O(1)$, since the parent pointers of the children of the detached node must be updated as well.
2. The transformations used for reducing the number of active nodes require that the rank of a node and that of its sibling are consecutive. To keep the old data structure operational, the binomial structure of the trees should not be broken.
3. The transformations might constantly swap subtrees of the same size, so it would be difficult to assure that all nodes have been visited if a simple tree traversal was done incrementally.

Our solution to these problems is repeated splitting. In one *rebuilding step*, if there is no tree of rank 0, a tree of the smallest rank is split into two halves; also if there is only one tree of rank 0 that contains the current minimum, but that is not the only tree left in the old structure, a tree of the smallest rank is split into two halves; otherwise, a tree of rank 0 — other than a tree of rank 0 which contains the current minimum — is removed from the old structure and, if not marked to be deleted, inserted into the new structure. That is, there can simultaneously be three trees of rank 0. This is done in order to keep the pointer to the location of the current minimum valid. As in a delete-min/delete operation, in one split step the youngest child of the root is made nonactive if it is active and its occurrence is removed from the run-singleton structure. With this strategy, a tree of size m can be emptied by performing $2m - 1$ rebuilding steps. Observe also that this strategy is in harmony with the strategy used in delete-min/delete operations; in the old structure the splits made by these operations will only speed up the rebuilding process.

Assume that there are m_0 pointers in the upper store when rebuilding is initiated, and assume that $m_0/2$ of them are marked to be deleted. Rebuilding is done piecewise over the forthcoming $m_0/4$ upper-store operations. More precisely, in connection with each of the following $m_0/4$ upper-store operations eight rebuilding steps are executed. At this speed, even with intermixed upper-store operations, the whole old structure will be empty before it will be necessary to rebuild the new structure.

A tree of rank 0, which does not contain the current minimum or is the only tree left, can be detached from the old run-relaxed binomial queue at a cost of $O(1)$. Similarly, a node can be inserted into the new run-relaxed binomial queue at a cost of $O(1)$. A marked node can also be released and its counterpart updated at a cost of $O(1)$. Also, a split step has the worst-case cost of $O(1)$. From these observations and our earlier analysis, it follows that rebuilding only adds an additive term $O(1)$ to the cost of all upper-store operations.

7.3 Insert-buffer and main-store operations

The elements are stored in the insert buffer and in the main store. Both components are implemented as run-relaxed binomial queues, but they have a common run-singleton structure. In the main store there can only be one tree per rank, except perhaps a single rank that may have two trees. All insertions are directed to the insert buffer, which also provides the nodes borrowed by insert-buffer and main-store deletions. 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, stored either in the insert buffer or in the main store. The counterparts of the minimum candidates are stored in the upper-store, so communication between the components storing elements and the upper store is necessary each time when a root or an active node is added or removed, but not when an active node is made a root.

As in a multipartite binomial queue, the priority-queue operations are executed in phases. If n_0 denotes the number of elements at the beginning of a phase, in one phase $\lfloor \log n_0 \rfloor$ modifying operations are carried out. Compared to our earlier construction, the only difference is one additional invariant:

5. At the beginning of the phase, if $n_0 > 0$, the input buffer contains at least $\lfloor \log n_0 \rfloor$ elements, i.e. borrowing is always possible even if all modifying operations in the phase were deletions.

Consider now node borrowing, and assume that the new invariant can be maintained. Since the insert buffer never becomes empty, it has always at least one tree and a tree of the smallest rank can be repeatedly split as prior to a run-relaxed-binomial-queue deletion, after which there is a free tree of rank 0 that can be borrowed. In each split step, if the youngest child of the root of the tree being split is active, it is removed from the run-singleton structure, but it need not be removed from the upper store. However, if the youngest child is not active, its counterpart has to be inserted into the upper store or unmarked if already present in the upper store. Since the size of the insert buffer is bounded by $c_1 \log n + c_2$ for some constants c_1 and c_2, the total cost of all splits is $O(\log \log n)$; and because of the upper-store operations $O(\log \log n)$ element comparisons may be necessary.

Let b_0 denote the size of the insert buffer at the beginning of a phase. To maintain the new invariant, we modify the preprocessing step such that, if $b_0 \leq 2\lfloor \log n_0 \rfloor$, an *incremental separating process* is initiated, the purpose of which is to move a small tree from the main store to the insert buffer. In this process the trees in the main store are visited one by one, starting from the tree of the highest rank, until the smallest tree, the size of which is larger than $2\lfloor \log n_0 \rfloor$, is found. Thereafter, this tree is repeatedly split until a tree is obtained whose size is between $2\lfloor \log n_0 \rfloor$ and $4\lfloor \log n_0 \rfloor$. This work is distributed such that one modifying operation handles one rank. The last operation in such an operation sequence moves a tree of the required size to the insert buffer in its proper place. In the insert buffer there can be at most a constant number of trees that have a higher rank, so this addition has a constant cost, i.e. it is not too expensive for a single modifying operation.

If no tree of size $2\lfloor \log n_0 \rfloor$ or larger exists, which is possible when $n_0 \leq 24$, a single operation moves all trees from the main store to the insert buffer and performs all necessary joins.

Even if all modifying operations in a phase were insertions, at the end of the phase the size of the insert buffer would be bounded above by $2\lfloor \log n_0 \rfloor + 4\lfloor \log n_0 \rfloor + \lfloor \log n_0 \rfloor \leq 7\lfloor \log n_0 \rfloor$ or, if $n_0 \leq 24$, by $24 + \lfloor \log n_0 \rfloor$, i.e. the new tree cannot make the insert buffer too large. If all modifying operations were deletions, the size of the insert buffer would be bounded below by $\lfloor \log n_0 \rfloor + 2\lfloor \log n_0 \rfloor - \lfloor \log n_0 \rfloor \geq 2\lfloor \log n_0 \rfloor$, i.e. the insert buffer cannot become too small either. Note that there can only be one active uniting process, which is initiated if $b_0 \geq 8\lfloor \log n_0 \rfloor$, or one active separating process, which is initiated if $b_0 \leq 2\lfloor \log n_0 \rfloor$, but not both at the same time. When in an active uniting process a join is done, a lazy deletion is necessary at the upper store; and when in an active separating process a split is done, an insertion may be necessary at the upper store. Hence, these incremental processes can only increase the cost of modifying operations by an additive constant.

An insert operation for a run-relaxed binomial queue requires two modifications in places where communication between the insert buffer and upper store is necessary. First, after the creation of a new node its counterpart must be added to the upper store. Second, in each join the counterpart of the loser tree must be lazily deleted from the upper store. Even after these modifications, the worst-case cost of an insert operation is $O(1)$.

In a decrease operation, three modifications will be necessary. First, each time when a new active node is created, an insert operation has to be done at the upper store. Second, each time when an active node is removed, the counterpart must be deleted from the upper store, which can be done lazily in a λ-reduction. Third, when the node accessed is a root or an active node, a decrease operation has to be invoked at the upper store. If an active node is made into a root, no change at the upper store is required. Even after these modifications, the worst-case cost of a decrease operation is $O(1)$.

A delete-min/delete operation always begins with an invocation of the procedure that frees a tree of rank 0 to be used for filling in the hole created by the node being deleted. The two forms of deletions are done otherwise as described for a run-relaxed binomial queue, but now the update of the pointer to the location of a minimum element can be avoided. A removal of a root or an active node will invoke a delete operation at the upper store, and an insertion of a new root or an active node will invoke an insert operation at the upper store. A λ-reduction may invoke one or two lazy deletions and at most one insertion at the upper store. These lazy deletions and insertions have the worst-case cost of $O(1)$. Node borrowing has the worst-case cost of $O(\log \log n)$, including $O(\log \log n)$ element comparisons. Only at most one real upper-store deletion will be necessary, which has the worst-case cost of $O(\log \log n)$ and includes $O(\log \log n)$ element comparisons. Therefore, as in the original form, a delete-min/delete operation has the worst-case cost of $O(\log n)$, but now the number of element comparisons performed is at most $\log n + O(\log \log n)$.

8. Concluding remarks

We provided a general framework for improving the efficiency of priority-queue operations with respect to the number of element comparisons performed. Essentially, we showed that it is possible to get below the $2 \log n$ barrier on the number of element comparisons performed per delete-min and delete, while keeping the cost of find-min and insert constant. We showed that this is possible even when a decrease operation is to be supported at the worst-case cost of $O(1)$. From the information-theoretic lower bound for sorting, it follows that the worst-case efficiency of insert and delete-min cannot be improved much. However, if the worst-case cost of find-min, insert, and decrease is required to be $O(1)$, we do not know whether the worst-case bound of $\log n + O(\log \log n)$ on the number of element comparisons performed per delete-min and delete could be improved.

The primitives, on which our framework relies, are tree joining, tree splitting, lazy deleting, and node borrowing; all of which have the worst-case cost of $O(1)$. However, as already indicated in Section 7, it is not strictly necessary to support so efficient node borrowing. It would be enough if this operation had the worst-case cost of $O(\log n)$, but included no more than $O(1)$ element comparisons. All our priority queues could be implemented, without affecting the complexity bounds derived, to use this weak version of node borrowing.

We used binomial trees as the basic building blocks in our priority queues. The main drawback of binomial trees is their high space consumption. Each node should store four pointers and a rank, in addition to the elements themselves. Assuming that a pointer and an integer can be stored in one word, a multipartite binomial queue uses $5n + O(\log n)$ words, in addition to the n elements. However, if the child list is doubly linked, but not circular, and if the unused pointer to the younger sibling is reused as a parent pointer as in [18], weak node borrowing can still be supported, keeping the efficiency of all other fundamental primitives the same. Therefore, if the above-mentioned modification relying on weak node borrowing is used, the space bound could be improved to $4n + O(\log n)$. In order to support lazy deleting, one extra pointer per node is needed, so a two-tier binomial queue requires additional $n + O(\log n)$ words of storage. A multipartite relaxed binomial queue needs even more space, $7n + O(\log n)$ words. As proposed in [8], the space requirement could be reduced by letting each node store a resizable array of pointers to its children.

These space bounds should be compared to the bound achievable for a dynamic binary heap which can be realized using $\Theta(\sqrt{n})$ extra space [3, 19]. However, a dynamic binary heap does not keep external references valid and, therefore, cannot support delete or decrease operations. To keep external references valid, a heap could store pointers to the elements instead, and the elements could point back to the respective nodes in the heap. Each time a pointer in the heap is moved, the corresponding pointer from the element to the heap should be updated as well. The references from the outside can refer

to the elements themselves which are not moved. With this modification, the space consumption would be $2n + O(\sqrt{n})$ words. Recall, however, that a binary heap cannot support insertions at a cost of $O(1)$.

A navigation pile, proposed by Katajainen and Vitale [20], supports weak node borrowing (cf. the second-ancestor technique described in the original paper). All external references can be kept valid if the compartments of the elements are kept fixed, the leaves store pointers to the elements, and the elements point back to the leaves. Furthermore, if pointers are used for expressing parent-child relationships, tree joining and tree splitting become easy. With the above-mentioned modification relying on weak node borrowing, pointer-based navigation piles could substitute for binomial trees in our framework. A navigation pile is a binary tree and, thus, three parent-child pointers per node are required. With the standard trick (see, e.g. [26, Section 4.1]), where the parent and children pointers are made circular, only two pointers per node are needed to indicate parent-child relationships. Taking into account the single pointer stored at each branch and the two additional pointers to keep external references valid, the total space consumption would be $5n + O(\log n)$ words.

It would be interesting to see which data structure performs best in practice when external references to compartments inside the data structure are to be supported. In particular, which data structure should be used when developing an industry-strength priority queue for a program library. It is too early to make any firm conclusions whether our framework would be useful for such a task. To unravel the practical utility of our framework, further investigations would be necessary.

Acknowledgements

We thank Fabio Vitale for reporting the observation, which he made independently of us, that prefix-minimum pointers can be used to speed up delete-min operations for navigation piles.

References

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

[2] G.S. Brodal. Worst-case efficient priority queues. *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms*, ACM/SIAM (1996), 52–58.

[3] A. Brodnik, S. Carlsson, E.D. Demaine, J.I. Munro, and R. Sedgewick. Resizable arrays in optimal time and space. *Proceedings of the 6th International Workshop on Algorithms and Data Structures*, *Lecture Notes in Computer Science* **1663**, Springer-Verlag (1999), 37–48.

[4] S. Carlsson. An optimal algorithm for deleting the root of a heap. *Information Processing Letters* **37** (1991), 117–120.

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

[6] 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).
[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*, 2nd Edition. The MIT Press (2001).
[8] J.R. Driscoll, H.N. Gabow, R. Shrairman, and R.E. Tarjan. Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM* **31** (1988), 1343–1354.
[9] A. Elmasry. Layered heaps. *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory*, *Lecture Notes in Computer Science* **3111**, Springer-Verlag (2004), 212–222.
[10] A. Elmasry and M.L. Fredman. Adaptive sorting and the information theoretic lower bound. *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science*, *Lecture Notes in Computer Science* **2607**, Springer-Verlag (2003), 654–662.
[11] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys* **24** (1992), 441–476.
[12] M.L. Fredman, R. Sedgewick, D.D. Sleator and R.E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica* **1** (1986), 111–129.
[13] 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.
[14] H.N. Gabow, J.L. Bentley, and R.E. Tarjan. Scaling and related techniques for geometry problems. *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, ACM (1984), 135–143.
[15] G.H. Gonnet and J.I. Munro. Heaps on heaps. *SIAM Journal on Computing* **15** (1986), 964–971.
[16] T. Hagerup and R. Raman. An efficient quasidictionary. *Proceedings of the 8th Scandinavian Workshop on Algorithm Theory*, *Lecture Notes in Computer Science* **2368**, Springer-Verlag (2002), 1–18.
[17] J. Iacono. Improved upper bounds for pairing heaps. *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*, *Lecture Notes in Computer Science* **1851**, Springer-Verlag (2000), 32–45.
[18] H. Kaplan and R.E. Tarjan. New heap data structures. Technical Report TR-597-99, Department of Computer Science, Princeton University (1999).
[19] J. Katajainen and B.B. Mortensen. Experiences with the design and implementation of space-efficient deques. *Proceedings of the 5th Workshop on Algorithm Engineering*, *Lecture Notes in Computer Science* **2141**, Springer-Verlag (2001), 39–50.
[20] J. Katajainen and F. Vitale. Navigation piles with applications to sorting, priority queues, and priority deques. *Nordic Journal of Computing* **10** (2003), 238–262.
[21] C. Levcopoulos and O. Petersson. Adaptive Heapsort. *Journal of Algorithms* **14** (1993), 395–413.
[22] A. Moffat and O. Petersson. An overview of adaptive sorting. *Australian Computer Journal* **24** (1992), 70–77.
[23] 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.
[24] A. Schönhage, M. Paterson, and N. Pippenger. Finding the median. *Journal of Computer and Systems Sciences* **13** (1976), 184–199.
[25] J.T. Stasko and J.S. Vitter. Pairing heaps: experiments and analysis. *Communications of the ACM* **30** (1987), 234–249.
[26] R.E. Tarjan. *Data Structures and Network Algorithms*. SIAM (1983).
[27] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM* **21** (1978), 309–315.
[28] J. Vuillemin. A unifying look at data structures. *Communications of the ACM* **23** (1980), 229–239.
[29] J.W.J. Williams. Algorithm 232: Heapsort. *Communications of the ACM* **7** (1964),

347–348.