

Multipartite Priority Queues

AMR ELMASRY

Alexandria University

and

CLAUS JENSEN and JYRKI KATAJAINEN

University of Copenhagen

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 $2 \log n + O(1)$ 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 $\log_2(\max\{2, n\})$. As an immediate application of the priority queue developed, we obtain a sorting algorithm 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.

Categories and Subject Descriptors: 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

Additional Key Words and Phrases: Priority queues, heaps, 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 have an $O(1)$ cost for insert, with an attempt to reduce the number of element comparisons involved in delete-min. Binary heaps [Williams 1964] 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 [Gonnet and Munro 1986]. Gonnet and Munro (corrected by Carlsson [1991]) 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

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

A. Elmasry, Department of Computer Engineering and Systems, Faculty of Engineering, Alexandria University, Alexandria, Egypt; C. Jensen and J. Katajainen, Department of Computing, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen East, Denmark.

© 2008 ACM. This is the authors’ version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version will be published in ACM Transactions on Algorithms.

binomial queues [Brown 1978; Vuillemin 1978] (called binomial heaps in [Cormen et al. 2001]) and pairing heaps [Fredman et al. 1986; Iacono 2000]. The same efficiency can be achieved in the worst case with a special implementation of a binomial queue (see, for example, [Carlsson et al. 1988]). For binomial queues, guaranteeing a cost of $O(1)$ per insert, $2 \log n - O(1)$ is a lower bound and $2 \log n + O(1)$ an upper bound on the number of element comparisons performed by delete-min and delete. In Section 2, we review how binomial trees are employed in binomial queues and prove the above-mentioned lower and upper bounds for binomial queues.

In Section 3, we present our framework for structuring priority queues. We apply the framework in two different ways to reduce the number of element comparisons performed in priority-queue operations. 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. This is an improvement over the $\log n + O(\log \log n)$ bound presented in the conference version of this paper [Elmasry 2004]. In Section 6, we show as an application of the framework that, by using a multipartite binomial queue in adaptive heapsort [Levcopoulos and Petersson 1993], a sorting algorithm is obtained that is optimally adaptive with respect to the *inversions* 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. In Section 7, we conclude by discussing which other data structures could be used in our framework as a substitute for binomial trees.

2. BINOMIAL QUEUES

A *binomial tree* [Brown 1978; Schönhage et al. 1976; Vuillemin 1978] 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, \dots , $r - 1$ 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 operation, every node should have space for a parent pointer. The parent pointer is set only if the node is the youngest child of its parent, otherwise its value is null. To distinguish the root from the other nodes, its parent pointer is set to point to a fixed sentinel.

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

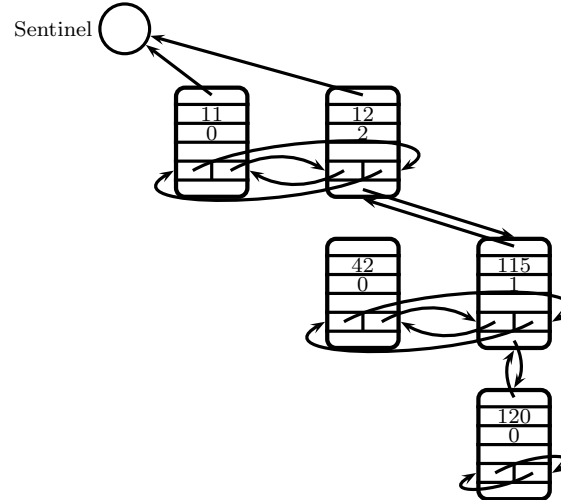


Fig. 1. A binomial queue storing integers $\{11, 12, 42, 115, 120\}$. Each node of a binomial tree has space for seven fields: parent pointer, element, rank, extra pointer (needed later on), older-sibling pointer, younger-sibling pointer, and youngest-child pointer. The non-standard features are the component sentinel and parent pointers that are only valid for the youngest children.

via the youngest child). It should be pointed out that with respect to the parent pointers our representation is non-standard. 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 4.1. 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 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 (see Figure 1). 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 heap-ordered trees can be linked together by making the root of the tree that stores the larger element the youngest child of the other root. Later on, we refer to this as a *join*. If the two joined trees are binomial trees of the same rank r , the resulting tree is a binomial tree of rank $r + 1$. 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 $O(1)$.

The operations for a binomial queue 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.

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 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.

For a binomial queue storing n elements, the worst-case cost per find-min is $O(1)$ and that per insert, delete-min, and delete 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 show 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 requires $\lfloor \log n \rfloor$ 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 achieve 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 is 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 is always space for new elements (for a similar treatment, see [Carlsson et al. 1988] or [Clancy and Knuth 1977, p. 53 ff.]). Analogously with an observation

made in [Carlsson et al. 1988], the size of the stack can be reduced dramatically if two joins 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 (for a similar proof, see [Clancy and Knuth 1977; Kaplan et al. 2002]). 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 same worst-case bounds, two element comparisons per insert and $2 \log n + O(1)$ element comparisons per delete-min/delete, is described in [Driscoll et al. 1988].

3. THE 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.

This is achieved by implementing the original queue as a binomial queue, while having an upper store forming another priority queue that only contains pointers to the elements stored at the roots of the binomial trees of the original queue. The minimum element referred to by this upper store is, therefore, an overall minimum element. The size of the upper store is $O(\log n)$ and every delete-min operation requires $O(\log \log n)$ comparisons for this queue. The challenge is how to maintain the upper store and how to efficiently implement the priority-queue operations on the lower store (original queue) to reduce the work to be done at the upper store, achieving the claimed bounds. If the delete-min operation is implemented the same way as that of the standard binomial queues, there would be a logarithmic number of new roots that need to be inserted at the upper store. Hence, a new implementation of the delete-min operation, that does not alter the current roots of the trees, is introduced. To realize this idea, we compose a priority queue using three components, which themselves are priority queues:

- (1) The *lower store* is a priority queue which stores at least half of the n elements. This store is implemented as a collection of separate *structures* storing the elements in individual *compartments*. 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 operations, *structure borrowing* should be supported in which a structure or part of a structure is 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 references to the m structures held in the lower store and uses the current minimum of each structure as the priority. The purpose of the upper store is to provide fast access to an overall minimum element stored at the lower store. The requirement is that *find-min* and *insert* 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 is readjusted to have the same size 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. Moreover, compartment borrowing should have a cost of $O(1)$.

To get from a compartment in the lower store to its counterpart in the upper store and vice versa, the corresponding compartments are linked together by pointers. 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 it is held, and that this information can be easily 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*, and by reservoir *find-min*. The smaller of these two elements is returned.

$I.insert(e)$. The 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, a compartment with the associated 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.

$I.delete(x)$. If the reservoir is empty, it is refilled using structure borrowing from the lower store. The extra information, associated with the structure in which 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 the lower-store and upper-store *insert* operations, 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 will be analysed after the actual realization of the components is detailed.

4. TWO-TIER BINOMIAL QUEUES

Our first realization of the framework uses binomial queues, in the form described in Section 2, as the basic structure. Because of the close interrelation between the upper store and the lower store during the execution of different operations, we call the data structure *two-tier binomial queue*. Its components and their implementations 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 of the lower store, but the upper store may also store pointers to earlier roots that are currently either in the reservoir or internal nodes in the lower store.
- (3) The reservoir is a single tree, which is binomial at the time of its creation.

The fields of the nodes stored at the lower store and the reservoir are identical, 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 component 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 roots, and we need to insert a pointer to each of these roots into the upper store. Possibly, some of the new subtrees have to be joined with the existing trees, which again may cascade a large 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 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 4.1. *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 x in T_2 , except the root, D_x increases by one because the only ancestor of x that gets a new younger sibling is the child of the new root. The claim follows from the induction assumption. \square

LEMMA 4.2. *Starting with a binomial tree of rank r in the reservoir, for any node x , D_x never gets larger than $r + 1$ during the life-span of this tree.*

PROOF. By Lemma 4.1, 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 this operation.

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 x in the subtree rooted at the i th oldest child (or simply the i th subtree), $i \in \{1, \dots, j\}$, D_x decreases by $j - i$. Except for the root, a join increases D_x by one for every other node x in the subtrees involved. Therefore, since a node x in the i th subtree is involved in $j - i + 1$ joins (except the

oldest subtree that is involved in $j - 1$ 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 the operation. \square

COROLLARY 4.3. *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. Let y be the root of the tree held in the reservoir. Let d_y denote the number of children of y , and x the oldest child of y . Clearly, $D_x = d_y + 1$. By Lemma 4.2, $D_x \leq r + 1$ all the time, and thus, $d_y \leq r$. \square

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 4.3 the number of children of the root is always smaller than or equal to r , and by Lemma 4.2 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 $O(\log \log n)$. 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 some of the nodes held in the other two components. In addition to the standard priority-queue operations, it has to support lazy deletions where nodes are marked to be deleted instead of being removed immediately (the use of lazy deletions and actual deletions will be explained in Section 4.3). An unmarked node points to a root of a tree stored at the lower store, and every root has a counterpart in the upper store. A marked node points to either an internal node held in the lower store or a node held in the reservoir. Therefore, a node in the upper store is marked when its counterpart becomes a non-root in the lower store. It should also be possible to unmark a node when the node pointed to by the stored pointer becomes a root.

To provide worst-case efficient lazy deletions, we use the global-rebuilding technique adopted from [Overmars and van Leeuwen 1981]. When the number of unmarked nodes becomes $m_0/2$, 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. 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 a constant additive term to the cost of all upper-store operations. A find-min operation may need to consult both the old and the new upper stores; its worst-case cost is still $O(1)$. The actual cost of marking and unmarking is clearly $O(1)$. If m denotes the number of unmarked nodes currently stored, the total number of nodes stored is $\Theta(m)$. Therefore, each delete-min and delete operation has the worst-case cost of $O(\log m)$ and performs at most $2 \log m + O(1)$ element comparisons.

4.3 Lower-store operations

A find-min operation invokes upper-store find-min and then follows the received pointer to the root storing a minimum element. Clearly, its worst-case cost is $O(1)$. An insert operation is accomplished, in a worst-case efficient manner, as described in Section 2. Once a new node is inserted in the lower store a corresponding node is inserted in the upper store. 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 that of each lazy deletion is also $O(1)$. Since each insert 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, and a node from the reservoir is borrowed and, seen as a tree of rank 0, repeatedly joined with the subtrees of the removed root. This results in a new binomial tree of the same size 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 (using an actual, not a lazy, deletion). However, if the pointer to the new 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 $O(1)$. 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 $\log n + O(\log \log n)$. Since in a binomial tree of size n the length of any D_x -path is never longer than $\lfloor \log n \rfloor + 1$, node swapping has the worst-case cost of $O(\log n)$, but it 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 our priority-queue operations can be summed up as follows:

THEOREM 4.4. *Let n be the number of elements stored in the data structure prior to each 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 $\log n + \log \log n + \dots + O(\log^{(k)} n)$, 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*. Instead of three components, the new data structure consists of five components (for an illustration, see Figure 2): main store, insert buffer, floating tree, upper store, and reservoir. The first three components replace the earlier lower store. The insert buffer is used for handling all insertions, the main store is used for storing the main bulk of elements, and the floating tree, if any, is used when moving elements from the insert buffer to the main store so the insert buffer remains small compared to the main store. The upper store facilitates fast access to a minimum element in the main store. Finally, the reservoir provides nodes when deletions are to be carried out in any of the first three components.

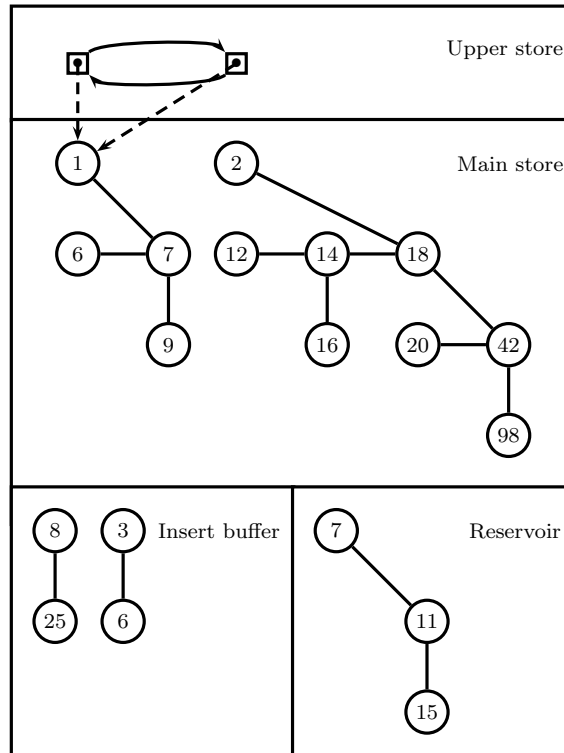


Fig. 2. A multipartite binomial queue storing 19 integers. Prefix-minimum pointers are indicated with dashed lines. No floating tree exists. Binomial trees are drawn in a schematic form (for details, see Figure 1).

The *main store* is maintained as a standard binomial queue containing at most one binomial tree of each rank. The root storing the current minimum is indicated by a pointer from the upper store. The *insert buffer* is also a binomial queue, but it is maintained in a worst-case efficient manner. A separate pointer indicates the root containing the current minimum in the insert buffer. The *floating tree*, if any, is a binomial tree having the same rank as the smallest tree in the main store.

At any point, let h denote the rank of the largest tree in the main store. We impose the following *rank invariant*: the ranks of all the trees in the insert buffer are between 0 and $\lceil h/2 \rceil$, and the ranks of all the trees in the main store are between $\lceil h/2 \rceil$ and h .

The *upper store* is implemented as a circular, doubly-linked list; having one node corresponding to each tree held in the main store. Each such node contains a pointer to the root of the tree which stores a minimum element among the elements stored in the trees of a lower rank, including the corresponding tree itself. We call these pointers the *prefix-minimum pointers*. The technique of prefix-minimum pointers has earlier been used, for example, in [Elmasry 2002; 2003a; 2003b].

A *reservoir* is still in use and all the reservoir operations are performed as previously described, but to refill it, a tree is borrowed from the insert buffer (if there are

more than one tree, the tree whose root stores the current minimum should not be borrowed), and if the insert buffer is empty, a tree is borrowed from the main store other than the tree of the highest rank. One possibility is to borrow the tree of the second highest rank from the main store, and update the prefix-minimum pointer of the tree of the highest rank. If there is only one tree in the two components—insert buffer and main store—altogether, we split that tree by cutting off the subtree rooted at the youngest child of the root, and move this subtree to the reservoir. From this and our earlier analysis (see Section 4.3), it follows that the worst-case cost of tree borrowing is $O(1)$.

All insert operations are directed to the insert buffer and executed in a manner similar to insertion in a worst-case efficient binomial queue. The only difference is when a tree of rank $\lceil h/2 \rceil + 1$ is produced in the insert buffer as a result of a join following an insertion. In such a case, the just created tree is split into two halves and the half rooted at the youngest child becomes a floating tree; the other half stays in the insert buffer. That is, tree borrowing is accomplished in a similar manner as in a reservoir refill at the worst-case cost of $O(1)$ (see Section 4.3). Note that this form of tree borrowing retains our rank invariant and does not affect the correctness of the minimum pointer maintained in the insert buffer.

If the rank of the smallest tree in the main store is larger than $\lceil h/2 \rceil$, the floating tree ceases to exist and is moved to the main store. Otherwise, the floating tree is united with the main store using the standard union algorithm (see, for example, [Cormen et al. 2001, Chapter 19]). However, the uniting process is executed incrementally by distributing the work over the forthcoming $h - \lceil h/2 \rceil + 1$ modifying operations—insertions or deletions. In connection with every modifying operation, one join is performed by uniting the floating tree with the smallest tree of the main store. That is, joins make the floating tree larger. Once there remains no other tree of the same rank as the floating tree, the uniting process is complete, and the floating tree becomes part of the main store. The position of the latest embedded floating tree is recalled to support the main-store find-min operation.

After moving the floating tree to the main store, the references of the prefix-minimum pointers may be incorrect so these have to be fixed, again incrementally, one pointer per operation starting from pointers corresponding to the trees of low ranks. The facts that there should have been at least $\lceil h/2 \rceil + 1$ insertions performed since another floating tree would have been moved from the insert buffer (as indicated by Lemma 5.1 below) and that at most $\lfloor h/2 \rfloor + 1$ element comparisons are required to finish the joins and fix the prefix-minimum pointers ensure that each uniting process will be finished before a new floating tree will be generated. In other words, there will be at most one floating tree at a time. It follows that an insert operation will have a worst-case cost of $O(1)$.

LEMMA 5.1. *After a floating tree is generated, at least $\lceil h/2 \rceil + 1$ insertions must be performed before another floating tree is generated.*

PROOF. Consider the insert buffer before the insertion that caused a floating tree to be generated. Because of the way the joins are performed (preference is given to joins involving small trees), there must be two trees of rank $\lceil h/2 \rceil$ and at most one tree of any other smaller rank. After the insertion, one of the two trees of rank $\lceil h/2 \rceil$ becomes the floating tree. For the same scenario to be produced again, at

least another $\lceil h/2 \rceil$ joins must be executed, starting by joining two trees of rank 0, then two of rank 1, and so on until producing another tree of rank $\lceil h/2 \rceil$. Since we are performing one join per insertion, the number of necessary insertions is at least $\lceil h/2 \rceil$. The next insertion would result in producing another floating tree. It is straightforward to verify that any other scenario will require more insertions to produce the second floating tree. It is also worth mentioning that tree borrowing may only increase the number of necessary joins to produce a floating tree. \square

An upper-store find-min operation provides a minimum element in the main store. This is done by comparing the element stored at the node pointed to by the prefix-minimum pointer associated with the root of the tree of the highest rank and the element stored at the root of the latest floating tree moved to the main store. The current minimum of the main store must be in one of these two positions. In a find-min operation, the four components storing elements—main store, reservoir, floating tree, and insert buffer—need to be consulted. Therefore, the worst-case cost of a find-min operation is $O(1)$.

Deletion from the reservoir is implemented as before with at most $\log n + O(1)$ element comparisons. Deletion from the insert buffer is performed in the same way as that of the worst-case efficient binomial queue. Because of the rank invariant, the largest rank of a tree in the insert buffer is at most $\lceil h/2 \rceil$, each insert-buffer deletion only requires at most $2\lceil h/2 \rceil + O(1)$ element comparisons, which is at most $h + O(1)$ element comparisons. The key idea of the deletion operation for the main store is to balance the work done in the main store and the upper store. After using a worst-case cost of $O(r)$ and at most $r + O(1)$ element comparisons to readjust a tree of rank r in the main store using node borrowing and repeated joins (as discussed in Section 4.3), only $\log n - r + O(1)$ element comparisons are used for the maintenance of the upper store. To delete a pointer corresponding to a tree of rank r from the upper store, the pointer in question is found by a sequential scan and thereafter removed, and the prefix-minimum pointers are updated for all trees of 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 root of a tree of rank r is changed (which can happen because of node borrowing), the prefix-minimum pointers can be updated in a similar manner. To summarize, the worst-case cost of main-store delete-min and delete operations is $O(\log n)$ with at most $\log n + O(1)$ element comparisons.

Each delete-min operation performs the following four tasks: (1) refill the reservoir if necessary, (2) execute one step of the incremental uniting process if necessary, (3) determine in which component an overall minimum element is stored, and (4) invoke the corresponding delete-min operation provided for that component. According to our earlier analysis, even with the overheads caused by the first three tasks, 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.

In a delete operation, the root is consulted to determine which of the delete operations provided for the components storing elements should be invoked. In each of the components, excluding the upper store, the deletion strategy relying on node borrowing used in the earlier lower store is applied (see Section 4.3). 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.

To show that the rank invariant is maintained, two observations are important. First, the only way for the highest rank of a tree in the main store, h , to increase is when the floating tree is joined with such a tree. This can only happen when there are no other trees in the main store which ensures that, even if $\lceil h/2 \rceil$ may increase, there is no need to move trees from the main store to the insert buffer to maintain the rank invariant. Second, the only way for h to decrease is when a subtree of the tree of rank h is moved to the reservoir by a tree borrowing operation. The way in which tree borrowing is implemented ensures that this can only happen when the tree of rank h in the main store is the only tree in the whole data structure. Therefore, even if $\lceil h/2 \rceil$ may decrease, there is no need to move trees from the insert buffer to the main store to maintain the rank invariant. To conclude, we have proved the following theorem.

THEOREM 5.2. *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 [Estivill-Castro and Wood 1992] or [Moffat and Petersson 1992]). In this section we consider adaptive heapsort, introduced by Levkopoulos and Petersson [1993], which is one of the simplest adaptive sorting algorithms. As in [Levcopoulos and Petersson 1993], 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 $X = \langle x_1, \dots, x_n \rangle$, the corresponding *Cartesian tree* [Vuillemin 1980] is a binary tree whose root stores element $x_i = \min \{x_1, \dots, x_n\}$, the left subtree of the root is the Cartesian tree for sequence $\langle x_1, \dots, x_{i-1} \rangle$, and the right subtree is the Cartesian tree for sequence $\langle x_{i+1}, \dots, x_n \rangle$. 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 returned and 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 [Gabow et al. 1984] 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. Levkopoulos and Petersson [1993] showed that, when element x_i is deleted, the number of elements in the priority queue is no

greater than $\lfloor |Cross(x_i)|/2 \rfloor + 2$, where

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

Levcopoulos and Petersson [1993, Corollary 20] showed that adaptive heapsort is optimally adaptive with respect to *Osc*, *Inv*, and several other measures of disorder. For a sequence $X = \langle x_1, x_2, \dots, x_n \rangle$ of length n , the measures *Osc* and *Inv* are defined as follows:

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

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

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 [Levcopoulos and Petersson 1993].

Implicitly, Levcopoulos and Petersson showed (see also an earlier version of their paper published in [Petersson 1990]) 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 at most $n \log(Osc(X)/n) + O(n)$ element comparisons when sorting a sequence X of length n . 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 based on insertion sort or mergesort [Elmasry and Fredman 2003].

7. CONCLUDING REMARKS

We provided a general framework for improving the efficiency of priority-queue operations with respect to the number of comparisons performed. Essentially, we showed that it is possible to get below the $2 \log n$ barrier on the number of comparisons performed per delete-min and delete, while keeping the cost of find-min and insert constant. From the information-theoretic lower bound for sorting, it follows that the worst-case efficiency of insert and delete-min cannot be improved much.

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, it is not strictly necessary to support so efficient node borrowing. It would be enough if this operation had the worst-case cost of $O(\log n)$, with no more than

$O(1)$ element comparisons. Our priority queues could be modified, 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, a rank, and an element. Assuming that a pointer and an integer can be stored each 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 of the younger sibling is reused as a parent pointer as in [Kaplan and Tarjan 1999], the space bound could be improved to $4n + O(\log n)$. Observe that after this change only weak node borrowing can be supported. In order to support lazy deletion, one extra pointer per node is needed, so a two-tier binomial queue requires additional $n + O(\log n)$ words of storage.

A navigation pile, proposed by Katajainen and Vitale [2003], 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 aforementioned modification relying on weak node borrowing, pointer-based navigation piles could substitute 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. [Tarjan 1983, 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 additional pointer to keep external references valid, the space overhead would be $4n + O(\log n)$ words.

By bucketing a group of elements into one node, the space overhead of any of the above-mentioned data structures can be improved to $(1 + \epsilon)n$ words, for any fixed real number $\epsilon > 0$. If $(\log n)$ -bucketing is used, as proposed in [Driscoll et al. 1988], the comparison complexity of delete/delete-min will increase by an additional $\log n$ factor, but with $O(1)$ -size buckets this increase can be reduced to an additive constant. However, this optimization is dangerous since it makes element moves necessary and one may lose the validity of external references. To avoid this complication, the bucketing technique has to be combined with the handle technique [Cormen et al. 2001, Section 6.5], whereby the space overhead becomes $(2 + \epsilon)n$ words. Further improvements to the space complexity of various data structures, including priority queues, are suggested in [Brönnimann et al. 2007].

The above space bounds should be compared to the bound achievable for a dynamic binary heap which can be realized using $\Theta(\sqrt{n})$ extra space [Brodnik et al. 1999; Katajainen and Mortensen 2001]. However, a dynamic binary heap does not keep external references valid and, thus, cannot support delete 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 which are not moved. After 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 $O(1)$.

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.

ACKNOWLEDGMENTS

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

- BRODNIK, A., CARLSSON, S., DEMAINE, E. D., MUNRO, J. I., AND SEDGEWICK, R. 1999. Resizable arrays in optimal time and space. In *Proceedings of the 6th International Workshop on Algorithms and Data Structures*. Lecture Notes in Computer Science, vol. 1663. Springer-Verlag, Berlin/Heidelberg, 37–48.
- BRÖNNIMANN, H., KATAJAINEN, J., AND MORIN, P. 2007. Putting your data structure on a diet. CPH STL Report 2007-1, Department of Computing, University of Copenhagen, Copenhagen.
- BROWN, M. R. 1978. Implementation and analysis of binomial queue algorithms. *SIAM Journal on Computing* 7, 3, 298–319.
- CARLSSON, S. 1991. An optimal algorithm for deleting the root of a heap. *Information Processing Letters* 37, 2, 117–120.
- CARLSSON, S., MUNRO, J. I., AND POBLETE, P. V. 1988. An implicit binomial queue with constant insertion time. In *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory*. Lecture Notes in Computer Science, vol. 318. Springer-Verlag, Berlin/Heidelberg, 1–13.
- CLANCY, M. J. AND KNUTH, D. E. 1977. A programming and problem-solving seminar. Technical Report STAN-CS-77-606, Department of Computer Science, Stanford University, Stanford.
- CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms*, 2nd ed. The MIT Press, Cambridge.
- DRISCOLL, J. R., GABOW, H. N., SHRAIRMAN, R., AND TARJAN, R. E. 1988. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM* 31, 11, 1343–1354.
- ELMASRY, A. 2002. Priority queues, pairing, and adaptive sorting. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*. Lecture Notes in Computer Science, vol. 2380. Springer-Verlag, Berlin/Heidelberg, 183–194.
- ELMASRY, A. 2003a. Distribution-sensitive binomial queues. In *Proceedings of the 8th International Workshop on Algorithms and Data Structures*. Lecture Notes in Computer Science, vol. 2748. Springer-Verlag, Berlin/Heidelberg, 103–113.
- ELMASRY, A. 2003b. Three sorting algorithms using priority queues. In *Proceedings of the 14th International Symposium on Algorithms and Computation*. Lecture Notes in Computer Science, vol. 2906. Springer-Verlag, Berlin/Heidelberg, 209–220.
- ELMASRY, A. 2004. Layered heaps. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory*. Lecture Notes in Computer Science, vol. 3111. Springer-Verlag, Berlin/Heidelberg, 212–222.
- ELMASRY, A. AND FREDMAN, M. L. 2003. Adaptive sorting and the information theoretic lower bound. In *Proceedings of the 20th Annual Symposium on Theoretical Aspects of Computer Science*. Lecture Notes in Computer Science, vol. 2607. Springer-Verlag, Berlin/Heidelberg, 654–662.
- ESTIVILL-CASTRO, V. AND WOOD, D. 1992. A survey of adaptive sorting algorithms. *ACM Computing Surveys* 24, 4, 441–476.
- ACM Transactions on Algorithms, Vol. , No. , 2008.

- FREDMAN, M. L., SEDGEWICK, R., SLEATOR, D. D., AND TARJAN, R. E. 1986. The pairing heap: A new form of self-adjusting heap. *Algorithmica* 1, 1, 111–129.
- GABOW, H. N., BENTLEY, J. L., AND TARJAN, R. E. 1984. Scaling and related techniques for geometry problems. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*. ACM, New York, 135–143.
- GONNET, G. H. AND MUNRO, J. I. 1986. Heaps on heaps. *SIAM Journal on Computing* 15, 4, 964–971.
- IACONO, J. 2000. Improved upper bounds for pairing heaps. In *Proceedings of the 7th Scandinavian Workshop on Algorithm Theory*. Lecture Notes in Computer Science, vol. 1851. Springer-Verlag, Berlin/Heidelberg, 32–45.
- KAPLAN, H., SHAFRIR, N., AND TARJAN, R. E. 2002. Meldable heaps and Boolean union-find. In *Proceedings of the 34th ACM Symposium on Theory of Computing*. ACM, New York, 573–582.
- KAPLAN, H. AND TARJAN, R. E. 1999. New heap data structures. Technical Report TR-597-99, Department of Computer Science, Princeton University, Princeton.
- KATAJAINEN, J. AND MORTENSEN, B. B. 2001. Experiences with the design and implementation of space-efficient dequeues. In *Proceedings of the 5th Workshop on Algorithm Engineering*. Lecture Notes in Computer Science, vol. 2141. Springer-Verlag, Berlin/Heidelberg, 39–50.
- KATAJAINEN, J. AND VITALE, F. 2003. Navigation piles with applications to sorting, priority queues, and priority dequeues. *Nordic Journal of Computing* 10, 3, 238–262.
- LEVCOPOULOS, C. AND PETERSSON, O. 1993. Adaptive heapsort. *Journal of Algorithms* 14, 3, 395–413.
- MOFFAT, A. AND PETERSSON, O. 1992. An overview of adaptive sorting. *Australian Computer Journal* 24, 2, 70–77.
- OVERMARS, M. H. AND VAN LEEUWEN, J. 1981. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Information Processing Letters* 12, 4, 168–173.
- PETERSSON, O. 1990. Adaptive sorting. Ph.D. thesis, Department of Computer Science, Lund University, Lund.
- SCHÖNHAGE, A., PATERSON, M., AND PIPPENGER, N. 1976. Finding the median. *Journal of Computer and System Sciences* 13, 2, 184–199.
- TARJAN, R. E. 1983. *Data Structures and Network Algorithms*. SIAM, Philadelphia.
- VUILLEMIN, J. 1978. A data structure for manipulating priority queues. *Communications of the ACM* 21, 4, 309–315.
- VUILLEMIN, J. 1980. A unifying look at data structures. *Communications of the ACM* 23, 4, 229–239.
- WILLIAMS, J. W. J. 1964. Algorithm 232: Heapsort. *Communications of the ACM* 7, 6, 347–348.

Received ; revised ; accepted