On the Power of Structural Violations in Priority Queues

Amr Elmasry[1]   Claus Jensen[2]   Jyrki Katajainen[2]

[1] Department of Computer Engineering and Systems, Alexandria University
Alexandria, Egypt
[2] Department of Computing, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark

Abstract

We give a priority queue that guarantees the worst-case cost of $\Theta(1)$ per minimum finding, insertion, and decrease; and the worst-case cost of $\Theta(\lg n)$ with at most $\lg n + O(\sqrt{\lg n})$ element comparisons per deletion. Here, n denotes the number of elements stored in the data structure prior to the operation in question, and $\lg n$ is a shorthand for $\max\{1, \log_2 n\}$. In contrast to a run-relaxed heap, which allows heap-order violations, our priority queue relies on structural violations. By mimicking a priority queue that allows heap-order violations with one that only allows structural violations, we improve the bound on the number of element comparisons per deletion to $\lg n + O(\lg \lg n)$.

Keywords: Data structures, priority queues, binomial queues, relaxed heaps, meticulous analysis, constant factors

1 Introduction

In this paper we study priority queues that are efficient in the worst-case sense. A *priority queue* is a data structure that stores a dynamic collection of elements and supports the standard set of operations for the manipulation of these elements: *find-min*, *insert*, *decrease*[-key], *delete-min*, and *delete*. We will not repeat the basic definitions concerning priority queues, but refer to any textbook on data structures and algorithms [see, for instance, (Cormen, Leiserson, Rivest & Stein 2001)].

There are two ways of relaxing a binomial queue (Brown 1978, Vuillemin 1978) to support *decrease* at a cost of $O(1)$. In run-relaxed heaps (Driscoll, Gabow, Shrairman & Tarjan 1988) heap-order violations are allowed. In a min-heap, a *heap-order violation* means that a node stores an element that is smaller than the element stored at its parent. A separate structure is maintained to keep track of all such violations. In Fibonacci heaps (Fredman & Tarjan 1987) and thin heaps (Kaplan & Tarjan 1999) structural violations are allowed. A *structural violation* means that a node has lost one or more of its subtrees. Kaplan & Tarjan

(1999) posed the question whether these two apparently different notions of a violation are equivalent in power.

Asymptotically, the computational power of the two approaches is known to be equivalent since fat heaps can be implemented using both types of violations (Kaplan & Tarjan 1999). To facilitate a more detailed comparison of data structures, it is natural to consider the number of element comparisons performed by different priority-queue operations since often these determine the computational costs when maintaining priority queues. A framework for reducing the number of element comparisons performed by *delete-min* and *delete* is introduced in a companion paper (Elmasry, Jensen & Katajainen 2004) [see also (Elmasry, Jensen & Katajainen 2006)]. The results presented in that paper are complemented in the present paper.

Let n denote the number of elements stored in the data structure prior to the operation in question. For both Fibonacci heaps (Fredman & Tarjan 1987) and thin heaps (Kaplan & Tarjan 1999), the bound on the number of element comparisons performed by *delete-min* and *delete* is $2\log_\Phi n + O(1)$ in the amortized sense, where Φ is the golden ratio. This bound can be reduced to $\log_\Phi n + O(\lg \lg n)$ using the two-tier framework described in (Elmasry 2004, Elmasry et al. 2004) ($\log_\Phi n \approx 1.44 \lg n$). For run-relaxed heaps (Driscoll et al. 1988) this bound is $3 \lg n + O(1)$ in the worst case [as analysed in (Elmasry et al. 2004, Elmasry et al. 2006)], and the bound can be improved to $\lg n + O(\lg \lg n)$ using the two-tier framework (Elmasry et al. 2004, Elmasry et al. 2006). For fat heaps (Kaplan, Shafrir, & Tarjan 2002, Kaplan & Tarjan 1999) the corresponding bounds without and with the two-tier framework are $4 \log_3 n + O(1)$ and $2 \log_3 n + O(\lg \lg n)$, respectively ($2 \log_3 n \approx 1.27 \lg n$).

In this paper we introduce a new priority-queue structure, named a *two-tier pruned binomial queue*, which supports all the standard priority-queue operations at the asymptotic optimal cost: *find-min*, *insert*, and *decrease* at the worst-case cost of $\Theta(1)$; and *delete-min* and *delete* at the worst-case cost of $\Theta(\lg n)$. We only allow structural violations, and not heap-order violations, to the binomial-queue structure. We are able to prove the worst-case bound of $\lg n + O(\sqrt{\lg n})$ on the number of element comparisons performed by *delete-min* and *delete*. Without the two-tier framework the number of element comparisons would be bounded above by $2 \lg n + O(\sqrt{\lg n})$.

In a two-tier pruned binomial queue, structural violations are applied in a straightforward way, but the analysis implies some room for improvement. In an attempt to answer the question posed by Kaplan and Tarjan, we show that the notion of structural violations is as powerful as that of heap-order violations in the case of relaxed heaps. Accordingly, we improve the bound on the number of element comparisons per

delete-min and *delete* to $\lg n + O(\lg \lg n)$. This is done by mimicking a two-tier relaxed heap described in (Elmasry et al. 2006) with a pruned version that only allows structural violations.

2 Two-tier pruned binomial queues

We use relaxed binomial trees (Driscoll et al. 1988) that rely on structural violations instead of heap-order violations as our basic building blocks. The trees, which we call *pruned binomial trees*, are heap-ordered and binomial, but a node does not necessarily have all its subtrees. Let τ denote the number of trees in any collection of trees, and let λ denote the number of missing subtrees in the *entire* collection of trees. A *pruned binomial queue* is a collection of pruned binomial trees where at all times both τ and λ are logarithmic in the number of elements stored.

Analogously to binomial trees, the *rank* of a pruned binomial tree is defined to be the same as the *degree* of its root, which is equal to the number of real children plus the number of lost children. For a pruned binomial tree, we let its *capacity* denote the number of nodes stored in a corresponding binomial tree where no subtrees are missing. The *total capacity* of a pruned binomial queue is the sum of the capacities of its trees. In a pruned binomial queue, there is a close connection between the capacities of the pruned binomial trees stored and the number representation of the total capacity. If the number representing the total capacity consists of digits d_0, d_1, up to d_{k-1}, the data structure stores d_i pruned binomial trees of capacity 2^i for each $i \in \{0, 1, \ldots, k-1\}$. In the number system used by us, digits d_i are allowed to be 0, 1, or 2. In an abstract form, a data structure that keeps track of the trees can be seen as a counter representing a number in this redundant number system. To allow increments and decrements at any digit at constant cost, we use a regular counter discussed, for example, in (Brodal 1996, Kaplan et al. 2002).

Following the guidelines given in (Elmasry 2004, Elmasry et al. 2004), our data structure has two main components, an *upper store* and a *lower store*, and both are implemented as pruned binomial queues with some minor variations. Our objective is to implement the priority queue storing the elements as the lower store, while having an upper store forming another priority queue that only contains pointers to the elements stored at the roots of the trees of the original queue. The minimum indicated by the upper store is, therefore, an overall minimum element.

We describe the data structure in four parts. First, we review the internals of a regular counter to be used for maintaining the references to the trees in a pruned binomial queue. Second, we give the details of a pruned binomial queue, but we still assume that the reader is familiar with a run-relaxed heap (Driscoll et al. 1988), from which many of the ideas are borrowed. Third, we show how the upper store of a two-tier pruned binomial queue is implemented. Fourth, we describe how a pruned binomial queue held in the upper store has to be modified so that it can be used in the lower store.

2.1 Guides for maintaining regular counters

Let d be a non-negative integer. In a *redundant binary system*, d is represented as a sequence of digits d_0, d_1, ..., d_{k-1} such that $d = \sum_{i=0}^{k-1} d_i \cdot 2^i$, where d_0 is the least significant digit, d_{k-1} the most significant digit, and $d_i \in \{0, 1, 2\}$ for all $i \in \{0, 1, \ldots, k-1\}$. The redundant binary representation of d is said to be *regular* if any digit 2 is preceded by a digit 0, possibly having a sequence of 1's in between. A digit sequence
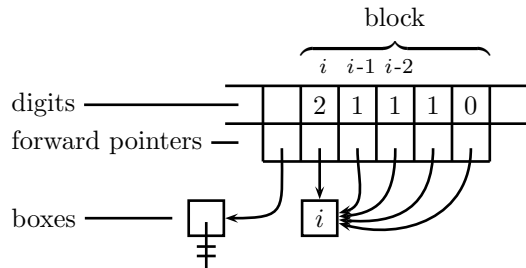

Figure 1: Illustration of a guide.

of the form $01^\alpha 2$, where $\alpha \in \{0, 1, \ldots, k-2\}$, is called a *block*. That is, every digit 2 must be part of a block, but there can be digits, 0's and 1's, that are not part of a block. The digit 2 that ends a block is called the *leader* of that block.

Assuming that the representation of d in the redundant binary system is d_0, d_1, ..., d_{k-1}, the following operations should be supported efficiently:

Fix up d_i **if** $d_i = 2$**.** Propagate the carry to the next digit, i.e. carry out the assignments $d_i \leftarrow 0$; $d_{i+1} \leftarrow d_{i+1} + 1$.

Increase d_i **by one if** $d_i \in \{0, 1\}$**.** Calculate $d + 2^i$.

Decrease d_i **by one if** $d_i \in \{1, 2\}$**.** Calculate $d - 2^i$.

Note that, if $d_i = 0$, a decrement need not be supported. Also, if $d_i = 2$, an increment can be done by fixing up d_i before increasing it.

In a pruned binomial queue, the data structure keeping track of the pruned binomial trees stored can be seen as a regular counter maintained under these operations. Brodal (1996) described a data structure, called a *guide*, that can be used to implement a regular counter such that the worst-case cost of each of the operations is $O(1)$. In a worst-case efficient binomial queue [see, e.g. (Elmasry et al. 2004)] the root list can be seen to represent a regular counter that only allows increments at the digit d_0. In such case, a stack is used as a guide. A general guide is needed to make it possible to increase or decrease any digit at the worst-case cost of $O(1)$. Next we briefly review the functionality of a general guide.

To represent a counter, a resizable array is used. In particular, a guide 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 (Brodnik, Carlsson, Demaine, Munro & Sedgewick 1999, Katajainen & Mortensen 2001)]. We let each priority-queue operation maintain a pointer to the last entry in use and initiate reorganization whenever necessary. In our application, the ith entry of a guide stores a list of up to two references to nodes of degree i. That is, the number of non-null references corresponds to digit d_i.

In addition to a list of nodes, the ith entry stores a *forward pointer* which points to the next leader d_j, $j > i$, if d_i is part of a block. To make it possible to destroy a block at a cost of $O(1)$, forward pointers are made indirect: for each digit its forward pointer points to a *box* that contains the index of the corresponding leader. All members of a block must point to the same box. Furthermore, a box can be *grounded* meaning that a digit pointing to it is no longer part of a block. The data structure is illustrated in Figure 1. Initially, a counter must have the value zero, which can be represented by a single 0 letting the forward pointer point to a grounded box.

Let us now consider how the counter operations can be realized.

Fix up d_i. There are three cases depending on the state of d_{i+1}. If $d_{i+1} = 0$ and d_{i+1} is not part of a block, assign $d_{i+1} \leftarrow 1$ and ground the box associated with d_i. If $d_{i+1} = 0$ and d_{i+1} is part of a block, assign $d_{i+1} \leftarrow 1$, ground the box associated with d_i, and extend the following block to include d_i as its first member. If $d_{i+1} = 1$, ground the box associated with d_i and start a new block having two members d_i and d_{i+1}.

Increase d_i by one if $d_i = 0$. If d_i is not part of a block, increase d_i by one. If d_i is part of a block, fix up the leader of that block. This will destroy the block, so after this d_i can be increased by one, keeping it outside a block.

Increase d_i by one if $d_i = 1$. If d_i is not part of a block, increase d_i by one and immediately afterwards fix up d_i. If d_i is part of a block, fix up the leader of that block, increase d_i by one, and fix up d_i. Both cases can create a new block of length two.

Decrease d_i by one if $d_i = 1$. If d_i is not part of a block, decrease d_i by one. If d_i is part of a block, fix up the leader of that block, which destroys the block, and thereafter decrease d_i by one.

Decrease d_i by one if $d_i = 2$. Ground the box associated with d_i and assign $d_i \leftarrow 1$.

By routine inspection, one can see that all these modifications keep the counter regular. Also, in the worst case at most two 2's need to be fixed up per increment and decrement.

2.2 Pruned binomial queues

A pruned binomial tree can be represented in the same way as a normal binomial tree [see, e.g. (Cormen et al. 2001)]; each node stores an element, a degree, a parent pointer, a child pointer, and two sibling pointers. To support the two-tier framework, the nodes should store yet another pointer to link a node in the lower store to its counterpart in the upper store, and vice versa. The basic tool used in our algorithms is a *join* procedure [called the binomial-link procedure in (Cormen et al. 2001)], where two subtrees of the same rank are linked together. The inverse of a join is called a *split*.

As a result of *decrease*, a node may loose one of its subtrees. To technically handle the lost subtrees, we use *phantom nodes* as placeholders for the subtrees cut off. A phantom node can be treated as if it stores an extremely large element ∞. A phantom node has the same associated information as the other nodes; its degree field indicates the rank of the lost subtree and its child pointer points to the node itself to distinguish it from real nodes. A *run* is a maximal sequence of two or more neighbouring phantom nodes. A *singleton* is a phantom node that is not in a run. When two pruned subtrees rooted at phantom nodes of the same degree are joined, one phantom node is released and the other becomes the result of the join and its degree is increased by one. If a phantom node becomes a root, it is simply released.

Formally, a *pruned binomial queue* is defined as follows. It is a collection of pruned binomial trees where the number of phantom nodes is no larger than $\lceil \lg n \rceil + 1$, n being the number of elements stored, and the total capacity of all trees is maintained as a regular counter. The following properties of a pruned binomial queue, which follow from the definition, are important for our analysis.

Lemma 1 *In a pruned binomial queue storing n elements, the rank of a tree can never be higher than $2 \lg n + O(1)$.*

Proof: Let the highest rank be k. The root of a tree of rank k has subtrees of rank $0, 1, \ldots, k-1$. In the worst-case scenario the $\lceil \lg n \rceil + 1$ phantom nodes are used as placeholders for the subtrees of the highest rank. The n elements occupy one node each, taking up a total of at most $\lfloor \lg n \rfloor + 1$ subtrees. Thus, the highest rank k cannot be larger than $2 \lg n + O(1)$. □

Lemma 2 *In a pruned binomial queue storing n elements, a node can never have more than $\lg n + O(\sqrt{\lg n})$ real children.*

Proof: The basic idea of the proof is to consider a tree whose root has $k + 2$ real children (k to be determined), and to replace some of its actual subtrees with phantom nodes such that:

- The number of the subtrees rooted at a phantom node is $\lceil \lg n \rceil + 1$.

- The number of real nodes is at most n.

- The value of k is maximized.

To maximize k, the children of the root of the chosen tree should be real nodes. Moreover, we should use the phantom nodes as placeholders for the largest $j+1$ subtrees of the children of the root, $2^{j-1} < n \leq 2^j$, i.e. $j = \lceil \lg n \rceil$. The largest such subtrees are: one binomial tree of rank k, two of rank $k - 1$, three of rank $k - 2$, and so forth.

Let h be the largest integer satisfying $1 + 2 + 3 + \ldots + h \leq j + 1$. Clearly, $h = \Theta(\sqrt{j})$. In order to maximize k, the number of nodes covered by missing subtrees culminates to $\sum_{i=1}^{h} i 2^{k-i+1} = 2^{k+2} - h 2^{k-h+1} - 2^{k-h+2}$. The total capacity of the whole tree is 2^{k+2} nodes, and of these at most n can be real nodes. Now the tree can only exist if $h 2^{k-h+1} + 2^{k-h+2} \leq n$. When $k \geq \lg n + h$, the number of the real nodes is larger than n, which means that such tree cannot exist. □

Lemma 3 *A pruned binomial queue storing n elements can never contain more than $\lg n + O(\sqrt{\lg n})$ trees.*

Proof: The proof is similar to that of Lemma 2. □

A *run-relaxed heap* (Driscoll et al. 1988) is a collection of almost heap-ordered binomial trees where there may be at most $\lfloor \lg n \rfloor$ heap-order violations between a node and its parent. A node is called *active* if it may be the case that the element stored at that node is smaller than the element stored at the parent of that node. There is a close correspondence between active nodes in a run-relaxed heap and phantom nodes in a pruned binomial queue. Therefore, many of the techniques used for the manipulation of run-relaxed heaps can be reused for the manipulation of pruned binomial queues.

To keep track of the trees in a pruned binomial queue, references to them are held in a *tree guide*, in which each tree appears under its respective rank. To keep track of the phantom nodes, a *run-singleton structure* is maintained as described in (Driscoll et al. 1988), so we will not repeat the bookkeeping details here. The fundamental operations supported by the run-singleton structure are an addition of a new phantom node, a removal of a given phantom node, and a removal of at least one arbitrary phantom node. The cost of all these operations is $O(1)$ in the worst case.

To support the transformations used for reducing the number of phantom nodes, when there are too many of them, each phantom node should have space for a pointer to the corresponding object, if any, in the run-singleton structure. A pictorial description of the transformations needed is given in the appendix. For further details, we refer to the description of the corresponding transformations for run-relaxed heaps given in (Driscoll et al. 1988). The rationale behind the transformations is that, when there are more than $\lceil \lg n \rceil + 1$ phantom nodes, there must be at least one pair of phantom nodes that root a subtree of the same rank, or a run of two or more neighbouring phantom nodes. When this is the case, it is possible to apply the transformations—a constant number of singleton transformations or run transformations—to reduce the number of phantom nodes by at least one. The cost of performing any of the transformations is $O(1)$ in the worst case. Later on, an application of the transformations together will all necessary changes to the run-singleton structure is called a λ-*reduction*.

The fact that the number of phantom nodes can be kept logarithmic in the number of elements stored is shown in the following lemma.

Lemma 4 *Let λ denote the number of phantom nodes. If $\lambda > \lceil \lg n \rceil + 1$, the transformations can be applied to reduce λ by at least one.*

Proof: The proof is by contradiction. Let us make the presumption that $\lambda \geq \lceil \lg n \rceil + 2$ and that none of the transformations applies. Since none of the singleton transformations applies, none of the singletons have the same degree. Hence, there must be a phantom node rooting a subtree whose rank r is at least $\lambda - 1$. A root cannot be a phantom node, so there must be a real node x that has this phantom node as its child. Since none of the run transformations applies, there are no runs. Hence, the sibling of the phantom node must be a real node; the subtree rooted at this real node is of rank $r - 1$. For all $i \in \{0, 1, \ldots, r - 2\}$, there is at most one phantom node rooting a subtree of that rank. These missing subtrees can cover at most $2^{r-1} - 1$ nodes. The total capacity of the subtree rooted at node x is 2^{r+1} nodes, and the missing subtrees of ranks $0, 1, \ldots, r$ can cover at most $2^r + 2^{r-1} - 1$ of the nodes. Hence, the subtree rooted at node x must store at least $2^{r+1} - 2^r - 2^{r-1} + 1 = 2^{r-1} + 1$ elements. If $\lambda \geq \lceil \lg n \rceil + 2$, this accounts for at least $2^{\lceil \lg n \rceil} + 1$ elements, which is impossible since there are only n elements. $\qquad\square$

2.3   Upper-store operations

The lower store contains elements and the upper store contains pointers to the roots of the trees in the lower store, as well as possibly pointers to some former roots lazily deleted. The number of pointers held in the upper store is never larger than $2 \lg n + O(\sqrt{\lg n})$. For the sake of clarity, we use m to denote the size of the upper store, and we call the pointers manipulated *items*. Of course, in item comparisons the elements stored at the roots pointed to in the lower store are compared. Let us now consider how the priority-queue operations are implemented in the upper store.

To facilitate a fast *find-min*, a pointer to the node storing the current minimum is maintained. When such a pointer is available, *find-min* can be easily accomplished at a cost of $O(1)$.

In *insert*, a new node is created, the given item is placed into this node, and the least significant digit of the tree guide is increased to get the new tree of rank 0 into the structure. If the given item is smaller than the current minimum, the pointer indicating the

location of the current minimum is updated to point to the newly created node. Clearly, the worst-case cost of *insert* is $O(1)$.

A *decrease* is performed by reusing some of the techniques described in (Driscoll et al. 1988). First, the item at the given node is replaced. Second, if the given node is not a root, the subtree rooted at that node is detached, a phantom node is put as its placeholder, and the detached subtree is added to the tree guide as a new tree. Third, if the new item is smaller than the current minimum, the pointer to the location of the current minimum is updated to point to the given node instead. At last, a λ-reduction is performed, if necessary. The cost of all this work is $O(1)$ in the worst case.

In *delete-min*, there are two cases depending on whether the degree of the root to be deleted is 0 or not.

Case 1 The root to be deleted has degree 0. In this case the root is released, the least significant digit of the tree guide is decreased to reflect this change, and a λ-reduction is performed once (since the difference between $\lceil \lg n \rceil + 1$ and $\lceil \lg(n-1) \rceil + 1$ can be one).

Case 2 The root to be deleted has degree greater than 0. In this case the root is released and a phantom node is repeatedly joined with the subtrees of the released root. More specifically, the phantom node is joined with the subtree of rank 0, the resulting tree is then joined with the next subtree of rank 1, and so on until the resulting tree is joined with the subtree of the highest rank. If before a join a subtree is rooted at a phantom node, the phantom node is temporarily removed from the run-singleton structure, and added back again after the join. This is necessary since the structure of runs may be changed by the joins. In the tree guide a reference to the old root is replaced by a reference to the root of the tree created by the joins. If after these modifications the number of phantom nodes is too large, a λ-reduction is performed once or twice (once because of the potential difference between $\lceil \lg n \rceil + 1$ and $\lceil \lg(n-1) \rceil + 1$, and once more because of the new phantom node introduced).

After both cases, all roots are scanned through to update the pointer pointing to the location of the current minimum.

The computational cost of *delete-min* is dominated by the joins and the scan, both having a cost of $O(\lg m)$. Everything else has a cost of $O(1)$. By Lemma 2, repeated joins may involve $\lg m + O(\sqrt{\lg m})$ item comparisons, and by Lemma 3, a scan visits at most $\lg m + O(\sqrt{\lg m})$ trees, so the total number of item comparisons is at most $2 \lg m + O(\sqrt{\lg m})$.

If the given node is a root, *delete* is similar to *delete-min*. If the given node is not a root, the subtree rooted at that node is detached and the node is released. The subtrees of the released node are repeatedly joined with a phantom node as above, after which the detached subtree is replaced by the resulting tree. Due to the new phantom node, at most two λ-reductions may be necessary to get the number of phantom nodes below the threshold. As *delete-min*, *delete* has the worst-case cost of $O(\lg m)$ and performs at most $2 \lg m + O(\sqrt{\lg m})$ item comparisons.

In addition to the above operations, 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 lower store, a join is done as a consequence of an insertion, or a λ-reduction is performed

that involves the root of a tree. In both situations, a normal upper-store deletion would be too expensive.

To support lazy deletions efficiently, we adopt the global-rebuilding technique described in (Overmars & van Leeuwen 1981). 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 (*modifying operations* including insertions, decreases, deletions, markings, and unmarkings). In spite of reorganization, both the old structure and the new structure are kept operational and used in parallel. New nodes are inserted into the new structure, and old nodes being deleted are removed from their respective structures.

In addition to the tree guide, which is used as normally, we maintain a separate *buffer* that can contain up to two trees of rank 0. Initially, the buffer is empty. It is quite easy to extend the priority-queue operations to handle these extra trees of rank 0. Depending on the state of the buffer and the guide, every *rebuilding step* does the following:

Case 1 a) In the buffer or in the guide there is a tree of rank 0 (i.e. a node) that does not contain the current minimum or b) there is only one node left in the old structure. In both cases that node is removed from the old structure. If the node is not marked to be deleted, it is inserted into the new structure. Otherwise, the node is released and, in its counterpart in the lower store, the pointer to the upper store is given the value null.

Case 2 a) In the buffer or in the guide there is no tree of rank 0 or b) there is only one tree of rank 0 that contains the current minimum, but it is not the only tree left in the old structure. In both cases the tree of rank 0 (if any) is moved from the guide to the buffer, if it is not there already, and thereafter in the guide a tree of the smallest rank is split into two halves. If after the split the root of the lifted half is a phantom node, it is released and its occurrence is removed from the run-singleton structure. Also, if after the split the guide contains two trees of rank 0, one of them is moved to the buffer.

There can simultaneously be three trees of rank 0, two in the buffer and one in the guide. This is done in order to keep the pointer to the location of the current minimum valid during reorganization.

It is crucial for the correctness of priority-queue operations that the guide is kept regular all the time. It is straightforward to see that this is the case. If the least significant digit of the guide is non-zero, it can never be part of a block. Thus, a decrement does not involve any joins and an increment can involve at most one join. Additionally, observe that when splitting a tree of the smallest rank the corresponding decrement at the guide can be done without any joins. (If $d_i = 1$ and d_i is part of a block, the block can just be made one shorter. A new block of length two is created unless a tree is moved to the buffer.)

With this strategy, a tree of size m_0 can be emptied by performing at most $c \cdot m_0$ rebuilding steps, for a positive integer c, provided that reorganization is spread over at most $\lceil d \cdot m_0 \rceil$ modifying operations, for a non-negative real number d. The following lemma shows that, for $d = 1/4$ and for any $m_0 > 0$, $c = 4$ will be a valid choice.

Lemma 5 *To empty a pruned binomial queue storing n elements, at most $2n + \lceil \lg n \rceil + 2N$ rebuilding steps have to be performed, provided that reorganization is spread over N modifying operations.*

Proof: Let us perceive the given pruned binomial queue as a graph having k nodes and ℓ edges, each connecting a node to its parent. Since the given data structure is a forest of trees, the graph has at most $k - 1$ edges. In the beginning, the pruned binomial queue has n real nodes and at most $\lceil \lg n \rceil + 1$ phantom nodes. Therefore, for the corresponding graph, $k \leq n + \lceil \lg n \rceil + 1$ and $\ell \leq n + \lceil \lg n \rceil$. We let n and ℓ vary during reorganization, and note that the process terminates when $n = 0$ and $\ell = 0$.

To see that each rebuilding step makes progress, observe that at each step either a real node is removed, meaning that n becomes one smaller, or a tree is split, meaning that ℓ becomes one smaller. That is, to ensure progress it is important that the associated decrements at the tree guide do no involve any joins. Hence, after at most $2n + \lceil \lg n \rceil$ rebuilding steps the data structure must be empty, provided that no other operations are executed. However, the data structure allows priority-queue operations, including markings and unmarkings, to be executed simultaneously with reorganization, but only operations creating new real nodes (*insert*) or modifying the linkage of nodes (*insert*, *decrease*, *delete-min*, and *delete*) can interfere with reorganization.

Of the modifying operations, only *insert* creates new real nodes. When the least significant digit of the tree guide is increased, at most one join will be necessary. That is, *insert* can increase both n and ℓ by one. A *decrease* may introduce a new phantom node, but an old edge is reused when connecting this phantom node to the structure. When the detached subtree is made into a separate tree, an increment at the tree guide may involve up to two joins, meaning that ℓ is increased by at most two. A deletion may introduce a new phantom node in place of the removed node, and the linkage between nodes may change, but the total number of edges remains the same or becomes smaller due to joins involving missing subtrees. It may happen that the node to be deleted roots a tree of rank 0, but in this case no joins are necessary in connection with a decrement done at the tree guide. The removal of a real node is just advantageous for reorganization. After *decrease*, *delete-min*, or *delete*, one or two λ-reductions may be done, but these will reduce the number of phantom nodes and will not increase the number of edges. (For run transformation I—see the appendix—an increment at the tree guide may involve up to two joins, but this is compensated for the two edges discarded.) □

In connection with each of the next at most $m_0/4$ upper-store operations, $4 \cdot c$ rebuilding steps are to be executed. 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.

A tree of rank 0, which does not contain the current minimum or is the only tree left, can be detached from the old pruned binomial queue at a cost of $O(1)$. Similarly, a node can be inserted into the new pruned 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 has the worst-case cost of $O(1)$. From these observations, it follows that reorganization only increases the cost of all modifying operations by an additive term of $O(1)$.

Each *find-min* has to consult both the old struc-

ture and the new structure, but its worst-case cost is still $O(1)$. The cost of markings and unmarkings is clearly $O(1)$, even if they take part in reorganization. If m_u denotes the total number of unmarked nodes currently stored, at any point in time, the total number of nodes stored is $\Theta(m_u)$, and during reorganization $m_0 = \Theta(m_u)$. In both structures, the efficiency of *delete-min* and *delete* depends on their current sizes which must be $O(m_u)$. Since *delete-min* and *delete* are handled normally, except that they may take part in reorganization, each of them has the worst-case cost of $O(\lg m_u)$ and performs at most $2 \lg m_u + O(\sqrt{\lg m_u})$ item comparisons.

2.4 Lower-store operations

Since the lower store is also a pruned binomial queue, most parts of the algorithms are similar to those already described for the upper store. In the lower store, *find-min* relies on *find-min* provided by the upper store. An insertion is performed in the same way as in the upper store, but a counterpart of the new root is also inserted into the upper store. In connection with each join (which may be necessary when an entry in the tree guide is increased) the pointer pointing to the root of the loser tree is lazily deleted from the upper store. Also, *decrease* is otherwise identical to that provided by the upper store, but the insertion of the cut subtree and the λ-reduction may generate lazy deletions at the upper store. Additionally, it may be necessary to insert a counterpart for the cut subtree into the upper store. If *decrease* involves a root, this operation is propagated to the upper store as well. Minimum deletion and deletion are also similar to the operations provided by the upper store, but the pointer to the old root might be deleted from the upper store and a pointer to the new root might be added to the upper store. In a λ-reduction, it may be necessary to move a tree in the tree guide, which may involve joins that again generate lazy deletions. In connection with *decrease*, *delete-min*, and *delete*, it is not always necessary to insert a counterpart of the created root into the upper store, because the counterpart exists but is marked. In this case, the counterpart is unmarked and *decrease* is invoked at the upper store if unmarking was caused by *decrease*.

Because at the upper store at most $O(1)$ insertions, decreases, markings, and unmarkings are done per lower-store operation, and because each of these operations can be carried out at the worst-case cost of $O(1)$, these upper-store operations do not affect the resource bounds in the lower store, except by an additive term of $O(1)$. The main advantage of the upper store is that both in *delete-min* and *delete* the scan of the roots is avoided. Instead, an old pointer is possibly removed from the upper store and a new pointer is possibly inserted into the upper store. By Lemma 3, the lower store holds at most $\lg n + O(\sqrt{\lg n})$ trees, and because of global rebuilding the number of pointers held in the upper store can be doubled. Therefore, the size of the upper store is bounded by $2 \lg n + O(\sqrt{\lg n})$. The upper-store operations increase the cost of *delete-min* and *delete* in the lower store by an additive term of $O(\lg \lg n)$.

The following theorem summarizes the result of this section.

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

3 Mimicking heap-order violations

The analysis of the two-tier pruned binomial queues reveals (cf. the proof of Lemma 2) that phantom nodes can root a missing subtree that is too large compared to the number of elements stored. In a run-relaxed heap, which relies on heap-order violations, this is avoided by keeping the trees binomial at all times. In this section we show that a run-relaxed heap (Driscoll et al. 1988) and a two-tier relaxed heap (Elmasry et al. 2006) can be mimicked by another priority queue that only allows structural violations. The key observation enabling this mimicry is that a relaxed heap would allow two subtrees of the same rank that are rooted at violation nodes to be exchanged without affecting the cost and correctness of priority-queue operations.

Let Q be a priority queue that has a binomial structure and relies on heap-order violations. We mimic Q with another priority queue Q' which relies on structural violations. A crucial difference between Q and Q' is that, if in Q a subtree is rooted at a violation node, in Q' the corresponding subtree is detached from its parent and the place of the root of the detached subtree is taken by a phantom node. All cut subtrees are maintained in a *shadow structure* that consists of a resizable array where the rth entry stores a pointer to a list of cut subtrees of rank r. While performing different priority-queue operations, we maintain an invariant that the number of phantom nodes of degree r is the same as the number of trees of rank r in the shadow structure. Otherwise, Q' has the same components as Q:

- The *main structure* contains the trees whose roots are not violation nodes.

- The *upper store* consists of a single pointer or another priority queue storing pointers to nodes held in the main structure and the shadow structure.

- The *run-singleton structure* stores references to phantom nodes held in the main structure or in the shadow structure. That is, the run-singleton structure is shared by the two other structures.

In general, all priority-queue operations are executed as for a pruned binomial queue, but now we ensure that the shadow invariant is maintained. When two missing subtrees of rank r—represented by phantom nodes of degree r—are joined, one of the phantom nodes is released, the degree of the other phantom node is increased by one, and in the shadow structure two trees of rank r are joined. When a phantom node becomes a root, the phantom node is released, a tree of the same rank is taken from the shadow structure and moved to the main structure, and the root of the moved tree is given the place of the phantom node. If a phantom node is involved in a join with a tree rooted at a real node, the phantom node becomes a child of that real node, and no changes are made in the shadow structure. To relate a tree held in the shadow structure with the run-singleton structure, we start from a phantom node and locate a tree of the same rank in the shadow structure using the resizable array. Clearly, the overhead of maintaining and accessing the shadow structure is a constant per operation.

Because *insert* only involves the trees held in the main structure, it is not necessary to consider the trees held in the shadow structure. Also, *find-min* is straightforward since it operates with the pointer(s) available at the upper store without making any changes to the data structure. If *decrease* involves a root held either in the main structure or in the shadow

structure, the change is propagated to the upper store. Otherwise, a subtree is cut off, a phantom node is put in the place of the root of the cut subtree, the cut subtree is moved to the appropriate list of the resizable array in the shadow structure, and the upper store is updated accordingly.

Compared to a pruned binomial queue, a new ingredient is an operation *borrow* which allows us to remove an arbitrary real node at a logarithmic cost from a run-relaxed heap (Driscoll et al. 1988) and at a constant cost from its adaptation relying on the zeroless number representation (Elmasry et al. 2006). In a pruned binomial queue, *borrow* can be implemented in an analogous manner, but instead of a guide we use an implementation of a regular counter, described in (Kaplan et al. 2002), which is suited for the zeroless number representation. In particular, in connection with a deletion it is not necessary to replace the deleted node with a phantom node, but a real node can be borrowed instead. This is important since a phantom node used by a deletion would not have a counterpart in the shadow structure. In *delete*, if the borrowed node becomes the root of the new subtree and a potential violation is introduced, the subtree is cut off and moved to the appropriate list of the resizable array in the shadow structure.

When the above description is combined with the analysis of a two-tier relaxed heap given in (Elmasry et al. 2006), we get the following theorem.

Theorem 2 *Let n be the number of elements stored in the data structure prior to each priority-queue operation. There exists a priority queue that only relies on structural violations and guarantees the worst-case cost of $O(1)$ per find-min, insert, and decrease; and the worst-case cost of $O(\lg n)$ with at most $\lg n + O(\lg \lg n)$ element comparisons per delete-min and delete.*

4 Conclusions

We gave two priority queues that support *decrease* and rely on structural violations. For the first priority queue, we allow structural violations in a straightforward manner. This priority queue achieves the worst-case bound of $\lg n + O(\sqrt{\lg n})$ element comparisons per deletion. For the second priority queue, we only allow structural violations in a weaker manner by keeping an implicit relation between the cut subtrees and the holes left after the cuts. This priority queue achieves $\lg n + O(\lg \lg n)$ element comparisons per deletion.

Though we were able to achieve better bounds with the latter approach, the difference was only in the lower-order terms. It is still interesting whether the two types of violations, heap-order violations and structural violations, are in a one-to-one correspondence or not. Another interesting question is whether it is possible or not to achieve a bound of $\lg n + O(1)$ element comparisons per deletion, when we allow *decrease*. Note that the worst-case bound of $\lg n + O(1)$ is achieved in (Elmasry et al. 2004), when *decrease* is not allowed.

Appendix

In this appendix, a pictorial description of the transformations applied in a λ-reduction is given. In a singleton transformation two singletons x and y are given, and in a run transformation the last phantom node z of a run is given. In the following only the relevant nodes for each transformation are drawn, all phantom nodes are drawn in grey, and $element[p]$ denotes the element stored at node p.

Singleton transformation I Both x and y are the last children of their parents p and q, respectively. Name the nodes such that $element[p] \not\succ element[q]$. Observe that this transformation works even if x and/or y are part of a run.
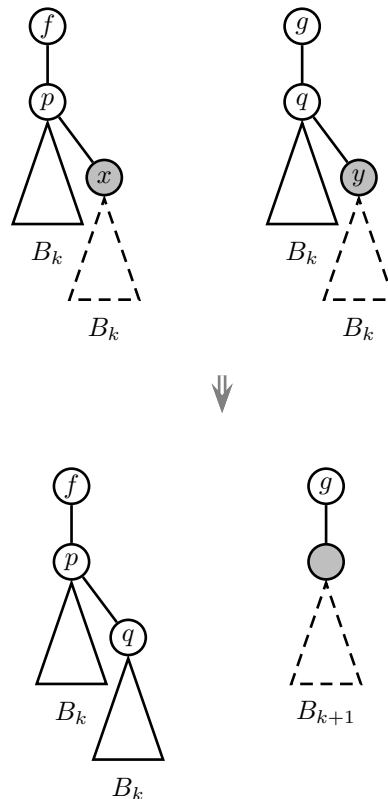

Singleton transformation II The parent of y is the right sibling of x, and y is the last child of its parent.
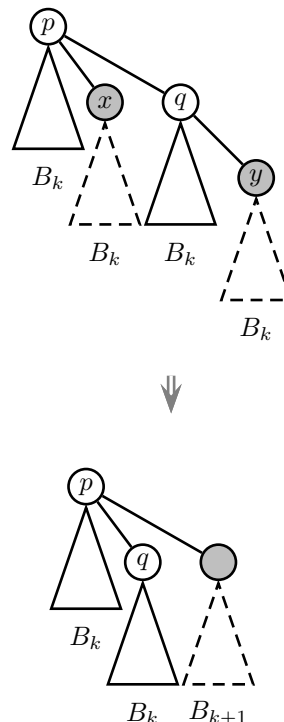
Singleton transformation III The given node x is not the last child of its parent and the last child of the right sibling of x is not a phantom node.
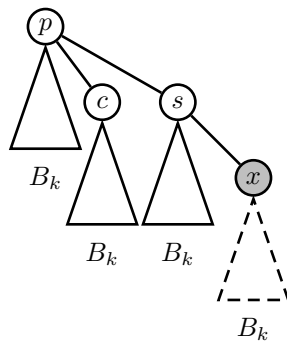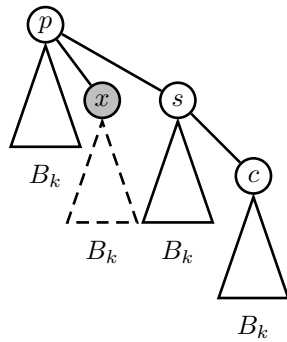
Run transformation I The given node z is the last child of its parent. After the transformation the earlier subtree rooted at the parent of z is seen as a separate tree.
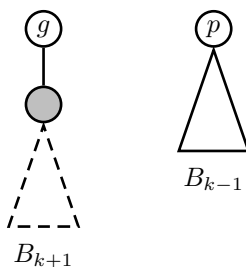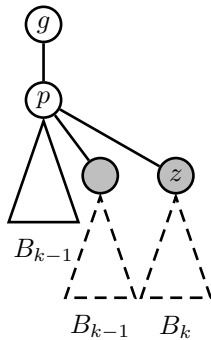
Run transformation II The given node z is not a last child. This transformation works even if some children of the right sibling of z are phantom nodes.
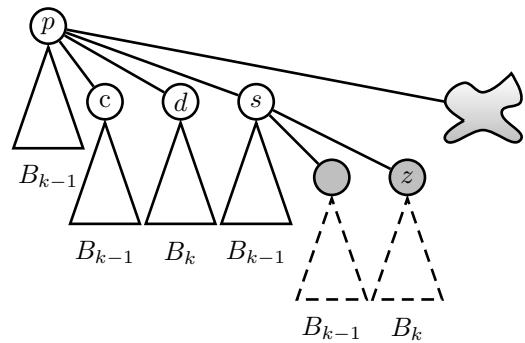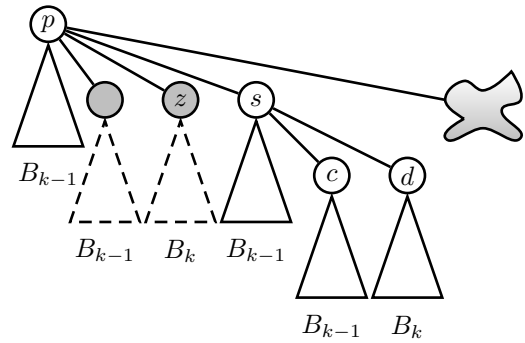
References

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

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

Brown, M. R. (1978), 'Implementation and analysis of binomial queue algorithms', *SIAM Journal on Computing* **7**(3), 298–319.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2001), *Introduction to Algorithms*, 2nd edn, The MIT Press.

Driscoll, J. R., Gabow, H. N., Shrairman, R. & 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. (2004), Layered heaps, *in* 'Proceedings of the 9th Scandinavian Workshop on Algorithm Theory', Vol. 3111 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 212–222.

Elmasry, A., Jensen, C. & Katajainen, J. (2004), A framework for speeding up priority-queue operations, CPH STL Report 2004-3, Department of Computing, University of Copenhagen. Available at `http://cphstl.dk`.

Elmasry, A., Jensen, C. & Katajainen, J. (2006), Two-tier relaxed heaps, *in* 'Proceedings of the 17th International Symposium on Algorithms and Computation', Vol. 4288 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 308–317.

Fredman, M. L. & Tarjan, R. E. (1987), 'Fibonacci heaps and their uses in improved network optimization algorithms', *Journal of the ACM* **34**(3), 596–615.

Kaplan, H., Shafrir, N., & Tarjan, R. E. (2002), Meldable heaps and Boolean union-find, *in* 'Proceedings of the 34th Annual ACM Symposium on Theory of Computing', ACM, pp. 573–582.

Kaplan, H. & Tarjan, R. E. (1999), New heap data structures, Technical Report TR-597-99, Department of Computer Science, Princeton University.

Katajainen, J. & Mortensen, B. B. (2001), Experiences with the design and implementation of space-efficient deques, *in* 'Proceedings of the 5th International Workshop on Algorithm Engineering', Vol. 2141 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 39–50.

Overmars, M. H. & van Leeuwen, J. (1981), 'Worst-case optimal insertion and deletion methods for decomposable searching problems', *Information Processing Letters* **12**(4), 168–173.

Vuillemin, J. (1978), 'A data structure for manipulating priority queues', *Communications of the ACM* **21**(4), 309–315.