On the Power of Structural Violations in Priority Queues

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

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

Abstract. We give a priority queue which guarantees the worst-case cost of $\Theta(1)$ per minimum finding, insertion and decrease (often called decrease-key), and the worst-case cost of $\Theta(\lg n)$ with at most $\lg n + O(\sqrt{\lg n})$ element comparisons per minimum deletion and 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. The motivation comes from a recent paper by Kaplan and Tarjan, where they asked whether these two apparently different notions of a violation are equivalent in power.

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

Keywords. Data structures, priority queues, 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 supposed to support the standard set of operations: *find-min*, *insert*, *decrease*, *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, [4]).

There are two ways of relaxing a binomial queue [1, 13] to support *decrease* at a cost of $O(1)$. In run-relaxed heaps [5] 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 [8]

---

and thin heaps [10] structural violations are allowed. A *structural violation* means that a node has lost one or more of its children. Kaplan and Tarjan [10] 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 equal since fat heaps can be implemented using both types of violations [10]. 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 [7]. 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 [8] and thin heaps [10], 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 $\log_\Phi n + O(1)$ using the two-tier framework described in [6, 7] ($\log_\Phi n \approx 1.44 \lg n$). For run-relaxed heaps [5] this bound is $3 \lg n + O(1)$ in the worst case (as analysed in [7]), and the bound can be improved to $\lg n + O(\lg \lg n)$ using the two-tier framework [7]. For fat heaps [9, 10] the corresponding bounds without and with the two-tier framework are $4 \log_3 n + O(1)$ and $2 \log_3 n + O(1)$, 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 $2 \lg n + O(\sqrt{\lg n})$.

2. Two-tier pruned binomial queues

We use relaxed binomial trees [5] 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 children. Let τ denote the number of trees in any collection of trees, and let λ denote the number of missing children in these trees, i.e. 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.

Following the guidelines given in [6, 7], 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 original queue 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 regular counters to be used for maintaining the references to the trees in a pruned binomial queue. Second, we recall the details of pruned binomial queues, but we still assume that the reader is familiar with the original paper on run-relaxed heaps [5], 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 nonnegative 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 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 an abstract form, in a pruned binomial queue a regular counter is maintained under these operations. The counter indicates the total number of elements stored in all binomial trees. Brodal [2] described a data structure, called a *guide*, that can be used to implement the required 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. [7]) 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 functioning of a general guide.

To represent a counter, a resizable array can be 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 [3, 11]). We let each modifying operation maintain a cursor to the last entry in use and initiate a reorganization whenever necessary. In our application, the ith entry of a guide stores a list of up to two references to nodes of the height i. That is, the number of nonnull references corresponds to digit d_i. Even if it would be possible to store the value of a counter more compactly, it is because of these references a resizable array is needed.

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. 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 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, immediately 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, still 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.

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; each node stores an element, a rank, 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 [4]), where two subtrees of the same rank are linked together.

To technically handle the lost children, we use *phantom nodes* as place-holders 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 rank field indicates the rank of the lost subtree and its child pointer points to the node itself to distinguish it from the normal 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 are joined, both having the same rank, one phantom node is discarded and the other becomes the result of the join after the rank is increased by one. If a phantom node becomes a root, it is simply discarded.

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, ..., $k - 1$. The worst that can happen is that the phantom nodes are used to cover the subtrees of the highest rank. The remaining n elements can cover one node each. Therefore, the highest rank k cannot be larger than $2\lceil \lg n \rceil + 1$. $\square$

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

Proof. The basic idea of the proof is to consider a (sub)tree of rank $k + 2$ (k to be determined), and to replace some of the actual subtrees with phantom nodes such that:

1. The number of the subtrees covered by phantom nodes is $\lceil \lg n \rceil + 1$.
2. The number of nonphantom nodes remaining is at most n.
3. The value of k is maximized.

To maximize k, the children of the root of the chosen tree should be non-phantom nodes. Moreover, we should use the phantom nodes to cover the largest $j + 2$ subtrees of the children of the root, $2^j \leq n < 2^{j+1}$. 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 $1 + 2 + 3 + ... + h \leq j + 2$. Clearly, $h = \Theta(\sqrt{j})$. The number of nodes covered by phantom nodes in order to maximize k culminates to $\sum_{i=1}^{h} i2^{k-i+1} = 2^{k+2} - h2^{k-h+1} - 2^{k-h+2}$. The whole tree contains 2^{k+2} nodes in total. Real nodes can cover at most n of them. Now, the tree can only exist if $h2^{k-h+1} + 2^{k-h+2} \leq n$. When

$k \geq \lfloor \lg n \rfloor + h$ the number of the remaining 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 binomial queue (called a run-relaxed heap in [5]) is a collection of almost heap-order 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 one-to-one correspondence between the active nodes in a run-relaxed binomial queue and the phantom nodes in a pruned binomial queue. Therefore, many of the techniques used for the manipulation of run-relaxed binomial queues 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 [5], 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 binomial queues given in [5]. 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 there is a run of two or more neighbouring phantom nodes. When this is the case, it is possible to apply at least one of the transformations — 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, one 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 $\lambda > \lceil \lg n \rceil + 1$, one of 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 rank. 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 can cover at most $2^{r-1} - 1$ nodes. The subtree rooted at node x has a capacity for at least 2^{r+1} elements, and at most $2^r + 2^{r-1} - 1$ of the elements in the subtrees of ranks 0, 1, …, r can be covered by phantom 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. Thus, the presumption must be wrong. □

2.3 Upper-store operations

Let us now consider how the priority-queue operations are implemented in the upper store. A reader familiar with run-relaxed binomial queues should be aware that we have made some minor modifications to *find-min*, *delete-min*, and *delete* to adapt them for our two-tier framework. 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.

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 easily be 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 $O(1)$.

In *delete-min*, a phantom node is repeatedly joined with the subtrees of the removed 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. In the tree guide the old root is replaced by the root of the tree created by the joins. All roots are scanned through to update the pointer pointing to the minimum node. If at the end the number of phantom nodes is too large, a λ-reduction is performed once or twice. 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*. Otherwise, the subtrees of the deleted 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 $O(\lg m)$ and performs at most $2\lg m + O(\sqrt{\lg m})$ item comparisons.

A *decrease* is performed in a similar manner as in [5]. After making the update, it is checked whether the heap order is violated or not. If there is no violation, the operation is complete. Otherwise, the subtree of the given node is detached and replaced with a phantom node. The tree cut off is added to the tree guide as a new tree. If the new item is smaller than that stored at the minimum node, the pointer to the minimum node is updated to point to the updated node instead. Lastly, it may be necessary to perform a λ-reduction once.

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 result of an insertion, or a λ-reduction that involves the root of a tree is performed. 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 [12]. When the number of unmarked nodes becomes equal to $m_0/2$, where m_0 is the current size of the upper store, we start building a new upper store. The work is distributed over the forthcoming $m_0/4$ upper-store operations. In spite of the reorganization, both the old structure and the new structure are kept operational and used in parallel. All new nodes are inserted into the new structure, and all old nodes being deleted are removed from their respective structures.

In one *rebuilding step*, if there is no tree of rank 0, a tree of the smallest rank is split into two halves; also if there is only one tree of rank 0 that contains the current minimum, but that is not the only tree left in the old structure, a tree of the second smallest rank is split into two halves; otherwise, a tree of rank 0 — other than a tree of rank 0 which contains the current minimum — is removed from the old structure and, if not marked to be deleted, inserted into the new structure; otherwise, it is released and in its counterpart in the lower store the pointer to the upper store is given the value null. There can simultaneously be three trees of rank 0. This is done in order to keep the pointer to the location of the current minimum valid. If after a split the root of the other half is a phantom node, it is simply discarded and its occurrence is removed from the run-singleton structure. With this strategy, a tree of size m_0 can be emptied by performing $2m_0 - 1$ rebuilding steps.

In connection with each of the next at most $m_0/4$ upper-store operations,

eight 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 rebuilding only adds an additive term $O(1)$ to the cost of all upper-store operations.

Each *find-min* has to consult both the old and the new upper stores, but its worst-case cost is still $O(1)$. The actual cost of marking and unmarking is clearly $O(1)$, even if they take part in reorganizations. If m_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 a reorganization $m_0 = \Theta(m_u)$. According to our earlier analysis, in the old structure the efficiency of *delete-min* and *delete* depends on the original size m_0. In the new structure their efficiency depends on its current size. Therefore, since *delete-min* and *delete* are handled normally, except that they may take part in reorganizations, each of them has the worst-case cost of $O(\lg m)$ and performs at most $2 \lg m + O(\sqrt{\lg m})$ 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 the upper store. An insertion is also performed in the same way as in the upper store, but in connection with each join (if 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. Minimum deletion and deletion are also similar, but the pointer to the old root must be deleted from the upper store and a pointer to the new root must 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. Also, *decrease* is otherwise identical to that provided by the upper store, but now the insertions and λ-reductions may generate lazy deletions. Furthermore, if *decrease* involves a root, the operation is propagated to the upper store.

Because lazy deletions can be performed at a cost of $O(1)$ in the upper-store, lazy deletions do not affect the resource bounds in the lower store. The

main difference 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 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 with an additive factor $O(\lg \lg n)$.

The following theorem summarizes the main result of this paper.

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. Conclusions

With the new priority queue, where we only allow structural violations and not heap-order violations, we were able to achieve a worst-case bound of $\lg n + O(\sqrt{\lg n})$ element comparisons per minimum deletion and deletion. On the other hand, in a companion paper [7], where we only allow heap-order violations and not structural violations, we were able to achieve a worst-case bound of $\lg n + O(\lg \lg n)$ element comparisons per minimum deletion and deletion. Though, we were able to achieve better bounds with the latter approach, for the best realizations in both categories the difference was only in the lower-order terms. It is still interesting whether the two types of 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 minimum deletion and deletion, when we allow the decrease operation. Note that the worst-case bound of $\lg n + O(1)$ is achieved in [7], when the decrease operation is not allowed.

References

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

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

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

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*, 2nd Edition. The MIT Press (2001).

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

[6] A. Elmasry. Layered heaps. *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science* **3111**. Springer-Verlag (2004), 212–222.

[7] A. Elmasry, C. Jensen, and J. Katajainen. A framework for speeding up priority-queue operations. CPH STL Report 2004-3. Department of Computing, University of Copenhagen (2004).

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

[9] H. Kaplan, N. Shafrir, and R. E. Tarjan. Meldable heaps and boolean union-find. *Proceedings of 34th Annual ACM Symposium on Theory of Computing*. ACM (2002), 573–582.

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

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

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

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

Appendix

In this appendix a pictoral description of the transformations applied in a λ-reduction is given. Due to space limitations the pictures will not be included in the conference proceedings. 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.
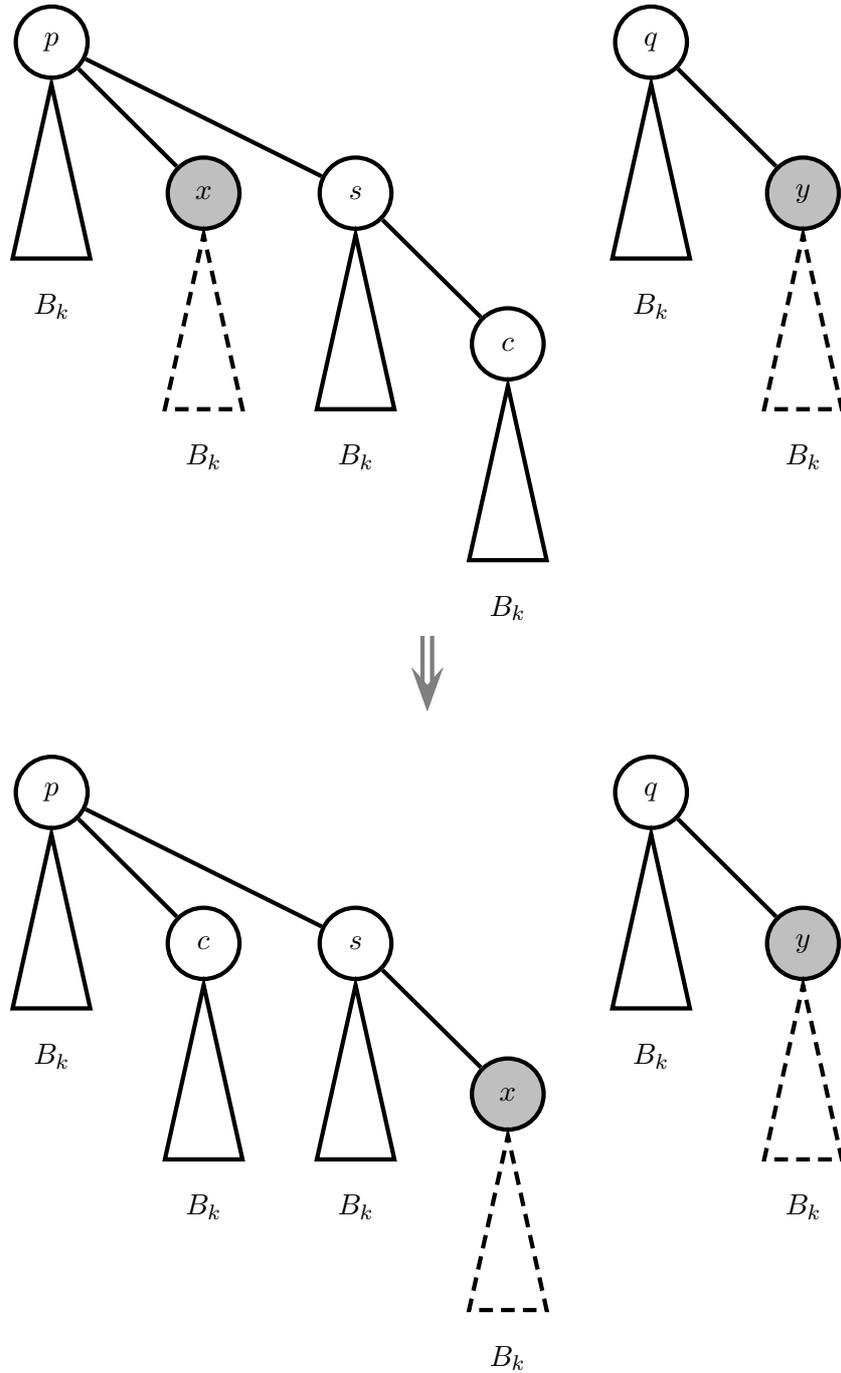
Singleton transformation I. Both x and y are the last children of their parents p and q, respectively. Assume that $element[p] \not> element[q]$. The case $element[p] > element[q]$ is symmetric. Observe that this transformation works even if x and/or y are part of a run.
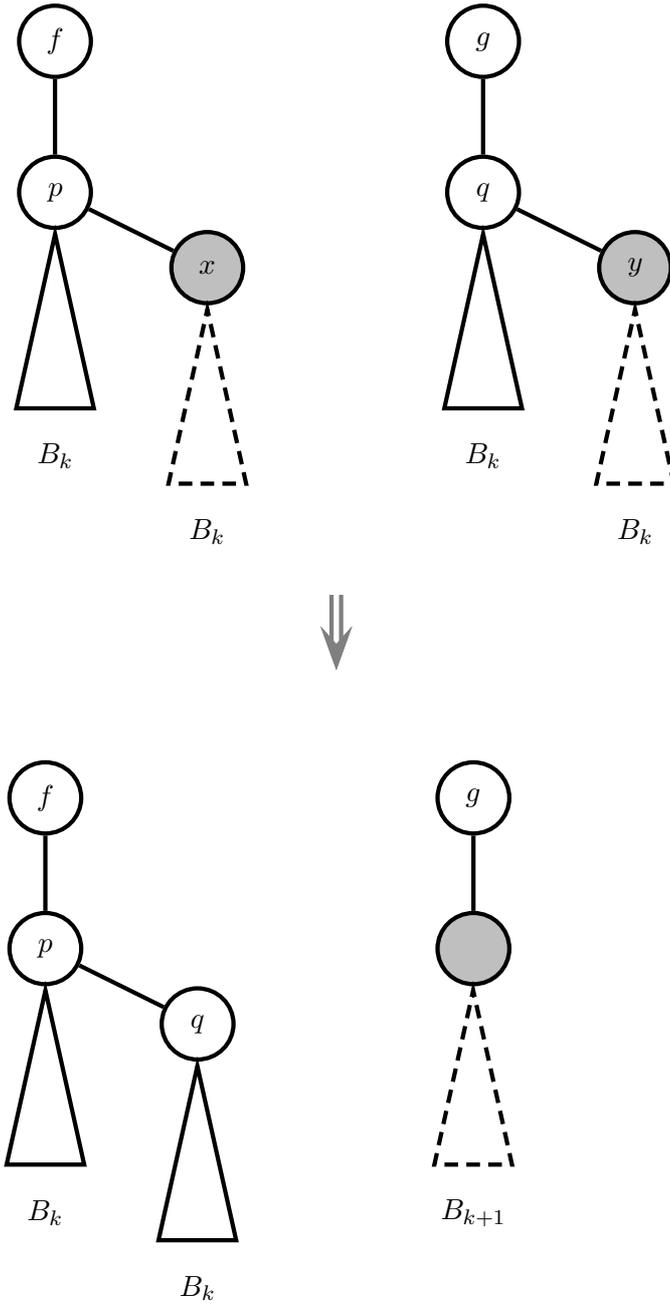
Singleton transformation II. The parent of y is the right sibling of x.
The case where the parent of x is the right sibling of y is symmetric.
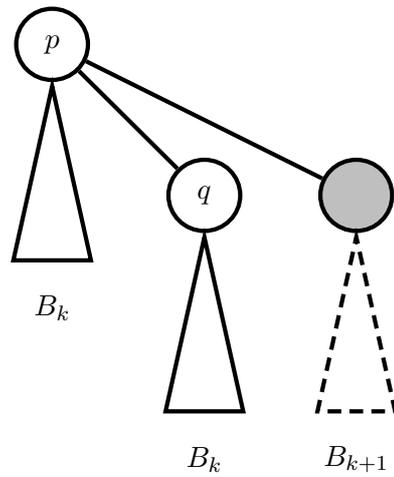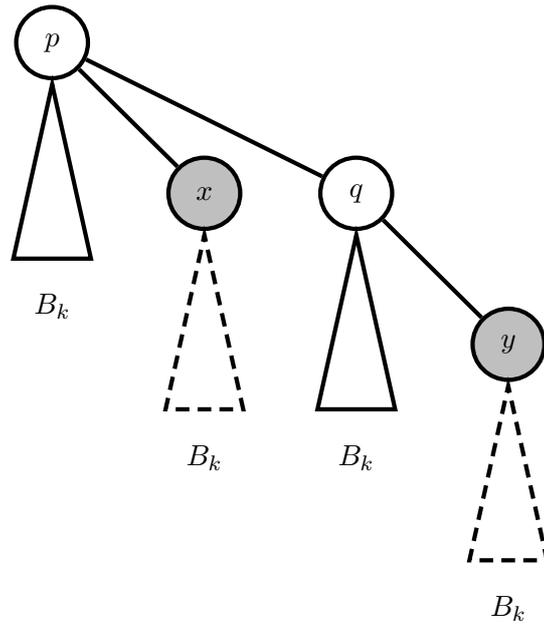
Singleton transformation III. Only y is the last child of its parent, but not x; and the last child of the right sibling of x is not a phantom node. The case where the roles of x and y are interchanged is symmetric.
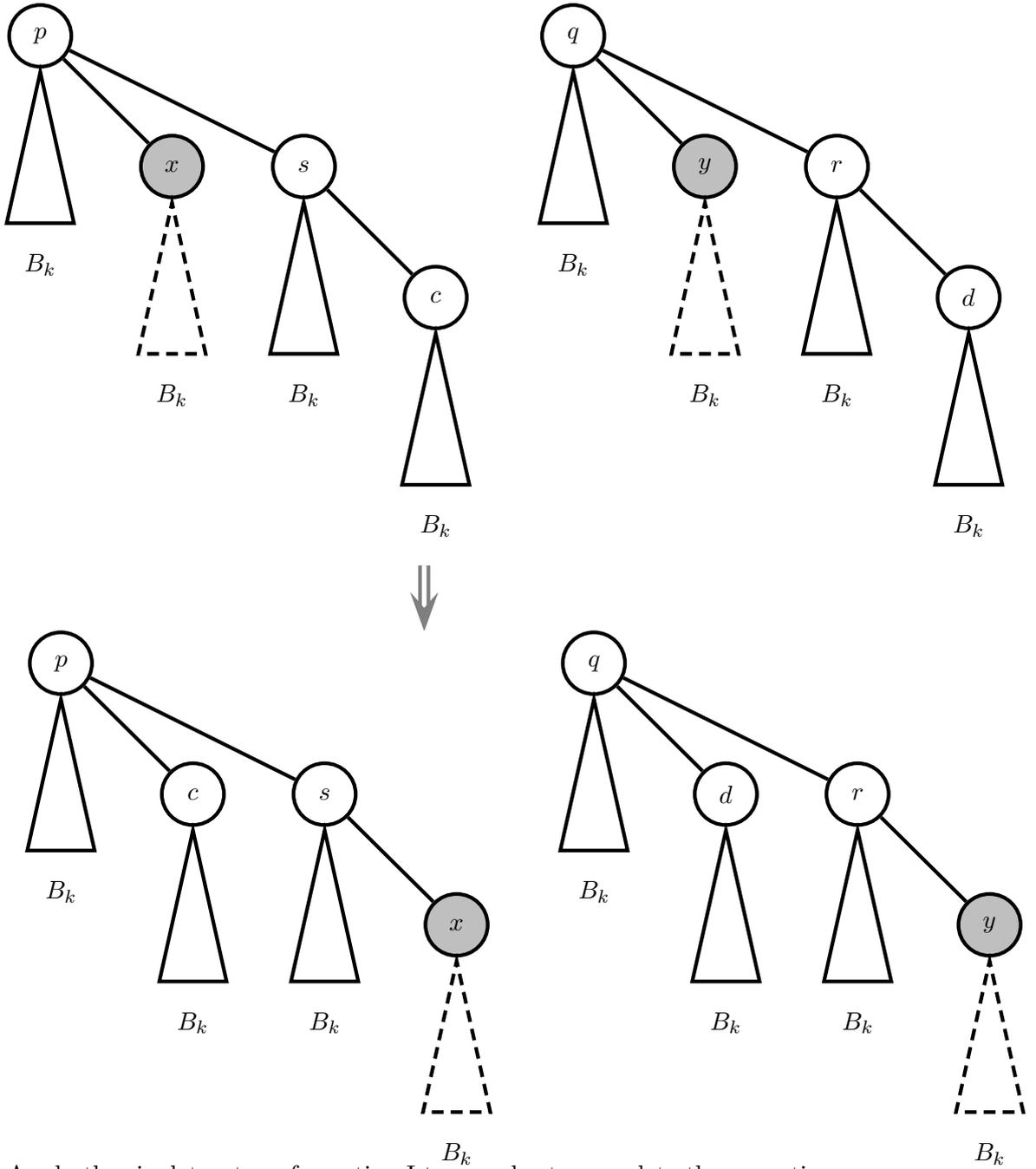

Apply the singleton transformation I to x and y to complete the operation.

Singleton transformation IV. Neither x nor y are the last children of
their respective parents, and the last child of the right sibling of x is a
phantom node (different from y). The case where the last child of the
right sibling of y is a phantom node is similar.
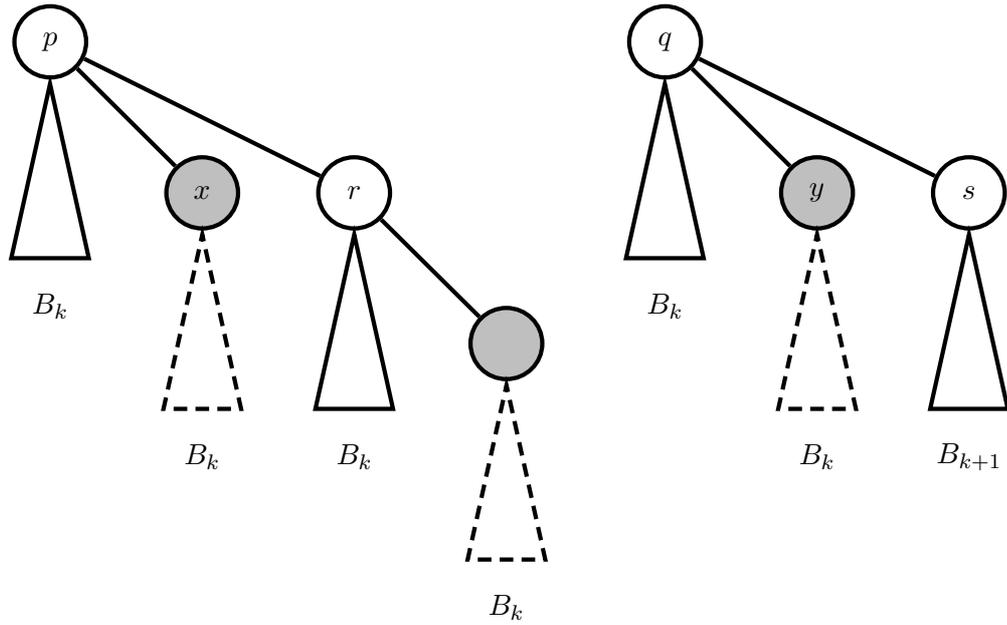
Apply the singleton transformation II to x and the last child of its sibling r
to complete the operation.

Singleton transformation V. Neither x nor y are the last children of their respective parents, and the last children of the right siblings of both x and y are not phantom nodes.

Apply the singleton transformation I to x and y to complete the operation.

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.
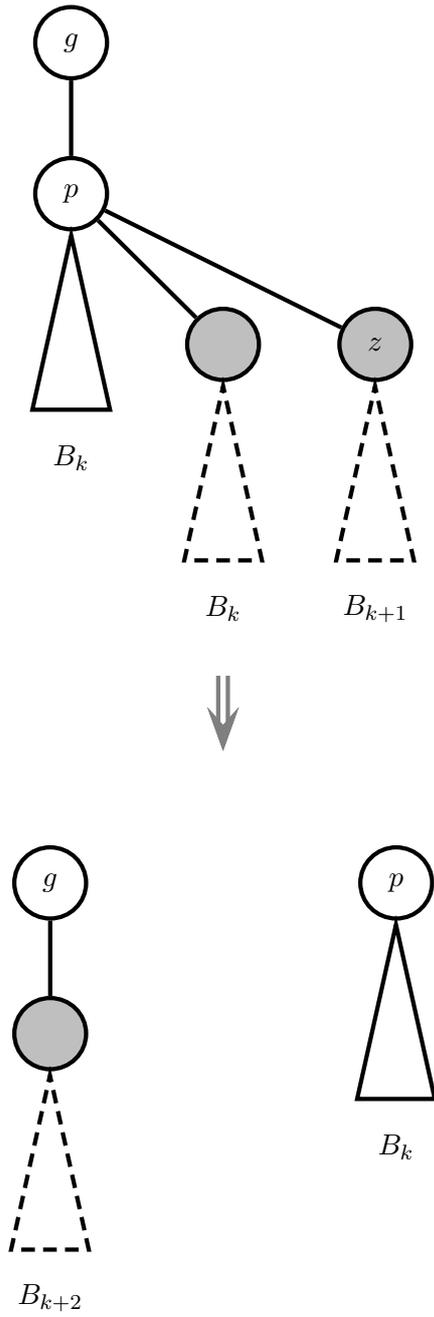
Run transformation II. The given node z is not a last child. The two last children of the right sibling of z may also be phantom nodes.