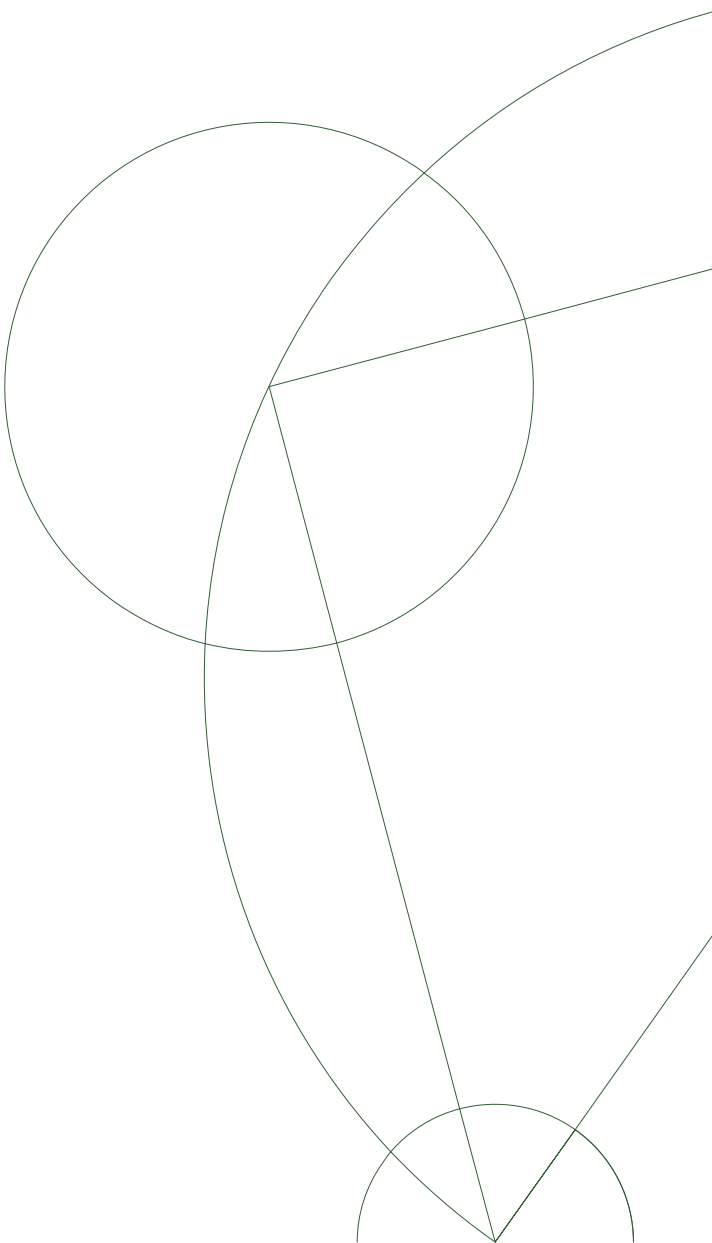
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Ph. D. Thesis

Claus Jensen

Improving the efficiency of priority-queue structures

Using data-structural transformations and number systems



Advisors: Jyrki Katajainen and Amr Elmasry

Submitted: July 2012

Abstract

In this thesis we investigate the comparison complexity of operations used in the manipulation of worst-case efficient data structures. The focus of the study is on the design and analysis of priority queues and double-ended priority queues. A priority queue is a data structure that stores a collection of elements and supports the operations *find-min*, *insert*, *extract*, *decrease*, *delete*, and *meld*; a double-ended priority queue also supports the operation *find-max*.

The worst-case efficiency of the priority queues and double-ended priority queues is improved using data-structural transformations and number systems. The research has been concentrated on improving the leading constant in the bound expressing the worst-case comparison complexity of the *delete* operation while obtaining a constant cost for a subset of the other operations. Our main contributions are:

- We devise a priority queue that for *find-min*, *insert*, and *delete* has a comparison-complexity bound that is optimal up to the constant additive terms, while keeping the worst-case cost of *find-min* and *insert* constant.
- We introduce a priority queue that for *delete* has a comparison-complexity bound that is constant-factor optimal (i.e. the constant factor in the leading term is optimal), while keeping the worst-case cost of *find-min*, *insert*, and *decrease* constant.
- We describe two new data-structural transformations to construct double-ended priority queues from priority queues.
- We introduce three new number systems.

In total, we introduce seven priority queues, two double-ended priority queues, and three number systems.

Acknowledgements

I thank my advisor Jyrki Katajainen for introducing me to the world of scientific research, teaching me how to write research papers, and teaching me the importance of details. I am deeply grateful for the help and support he has given me. I also thank my other advisor Amr Elmasry for his help and great support.

Furthermore, I like to thank Torben Hagerup and his group for their hospitality during my visit at Institut für Informatik, Universität Augsburg.

Finally, I thank my family for their encouragement and continued support.

Contents

General introduction

In this thesis we study the efficiency of addressable priority-queue structures. The main focus of the study is on the reduction of the worst-case comparison complexity of the operations used in the manipulation of priority queues and double-ended priority queues.

The thesis consists of this general introduction and seven individual papers. In the introduction we review worst-case-efficient priority queues relevant to our study. Also we give a brief survey of number systems and we explain their connection to priority queues. We conclude the introduction by summarizing the results obtained in the individual papers.

1. Priority queues

A priority queue is a data structure that maintains a collection of elements from a totally ordered universe. For reasons of simplicity we will not distinguish the elements from their associated priorities. The following set of operations is supported by a minimum priority queue Q:

find-min(Q). Returns a reference to a node containing a minimum element of priority queue Q.

insert(Q, x). Inserts a node referenced by x into priority queue Q. It is assumed that the node has already been constructed to contain an element.

extract(Q). Extracts an unspecified node from priority queue Q, and returns a reference to that node. The *extract* operation is in some places called *borrow*.

delete-min(Q). Removes a minimum element and the node in which it is contained from priority queue Q.

delete(Q, x). Removes the node referenced by x, and the element it contains, from priority queue Q.

decrease(Q, x, e). Replaces the element at the node referenced by x with element e. It is assumed that e is not greater than the element earlier stored in the node.

meld(Q_1, Q_2). Creates a new priority queue containing all the elements held in the priority queues Q_1 and Q_2, and returns a reference to that priority queue. This operation destroys Q_1 and Q_2.

Observe that *extract* is a non-standard priority queue operation. However, the importance of using *extract* internally within priority queues will be demonstrated in many of the included papers. Furthermore, placing *extract* in the priority-queue interface makes it possible to use this operation in other

data structures, the advantage of which will be demonstrated in connection with double-ended priority queues.

A double-ended priority queue Q supports the following operations in addition to the above-mentioned operations:

find-max(Q). Returns a reference to a node containing a maximum element of priority queue Q.

delete-max(Q). Removes a maximum element and the node in which it is contained from priority queue Q.

Throughout the introduction we use m and n to denote the number of elements stored in the manipulated data structures prior to an operation and $\lg n$ as a shorthand for $\log_2(\max\{2, n\})$.

Priority queues are important in many application areas like, for example, networks, simulation, compression, and sorting. Priority queues are used in well-known algorithms like Dijkstra's single-source shortest-paths algorithm (see for example [9, Chapter 24]) and heapsort (see [9, Chapter 6]). Priority queues with good worst-case bounds can be used for managing limited resources like bandwidth in network routers and for managing events in discrete event simulations.

In the following we review a relevant selection of worst-case-efficient priority queues introduced by others. Observe that the mentioned comparison-complexity bounds are derived by us.

Constant-cost find-min. The binary-heap data structure introduced by Williams [25] in 1964 is one of the most well-known priority queues and also one of the earliest. For the binary heap by Williams having size n, the worst-case cost of *insert* and *delete-min* is $O(\lg n)$ and the comparison complexity for *insert* is $\lg n + 1$ and that for *delete-min* $2 \lg n$. Gonnet and Munro [15] has shown that $\log \log n \pm O(1)$ element comparisons are necessary and sufficient for inserting a element into a binary heap and that $\log n + \log^* n \pm O(1)$ element comparisons are necessary and sufficient for deleting a minimum element from a binary heap.

A binary heap is a nearly complete (or left-complete) binary tree where each node contains an element. In a minimum heap, the priority-queue operations maintain the nodes in *heap order*, i.e. for a node having at least one child, the element stored at that node should not be greater than an element stored at any child of that node. A binary heap can be represented using an array where the nodes are stored in breath-first order.

For a more extensive description of the basic concepts related to binary heaps (see for example [9, Chapter 6]). The heaps described by Johnson [16] generalize binary heaps to d-ary heaps, which are d-ary trees for $d > 2$.

Constant-cost insert. Binomial queues improve the cost of *insert* to $O(1)$; for the original version of binomial queues [24] this bound is only valid in the amortized sense, but it was quickly observed that the bound could also be achieved in the worst case [5]. Later several other worst-case-efficient variants of binomial queues have been developed (see, for example [6] or [10]). For binomial queues, guaranteeing *insert* at the worst-case constant

cost, $2 \lg n - O(1)$ is a lower bound and $2 \lg n + O(1)$ an upper bound on the number of element comparisons performed by *delete*, see [12].

The basic components of binomial queues are heap-ordered binomial trees, see Figure 1. For a positive integer r, the *rank* of a binomial tree can be defined as follows: A binomial tree of rank 0 is a single node; for ranks higher than 0, a binomial tree of rank r consists of the root and its r subtrees of rank 0, 1, ..., $r - 1$. The size of a binomial tree is always a power of two, and the rank of a tree of size 2^r is r.

In a standard representation of a binomial tree a node contains an element, a rank, a parent pointer, a child pointer, and two sibling pointers.

If two heap-ordered binomial trees have the same rank, they can be linked together by making the root that stores the non-smaller element a child of the other root. We refer to this as a *join*. A *split* is the inverse of a join. A join involves a single element comparison, and both a join and a split have the worst-case cost of $O(1)$.

A binomial queue of size n is a forest of $O(\lg n)$ binomial trees. To keep the number of trees bounded, different strategies can be used; one strategy is to use number systems (see Section 2), another is to define an upper limit on the number of trees τ. For instance, the binomial queue used in a run-relaxed heap [10] maintains the invariant $\tau \leq \lfloor \lg n \rfloor + 1$. When inserting a new node into a binomial queue the node is treated as a tree of rank 0. Whether this tree is joined with another tree is governed by the strategy chosen to maintain the number of trees logarithmic.

Constant-cost decrease. Run-relaxed heaps [10] achieve the worst-case cost of $O(1)$ for *insert* and *decrease*. For *delete*, run-relaxed heaps achieve the same asymptotical bound in the worst-case sense as binomial queues and the comparison complexity of *delete* is $3 \lg n + O(1)$ given that *find-min* has the worst-case cost of $O(1)$.

Similar to binomial queues the basic components of run-relaxed heaps are binomial trees. Apart from the differences in the way the forest of trees is maintained, the main difference between binomial queues and run-relaxed heaps is that run-relaxed heaps allow a number of nodes to violate the heap order. These heap-order violations (potential violations) can be introduced

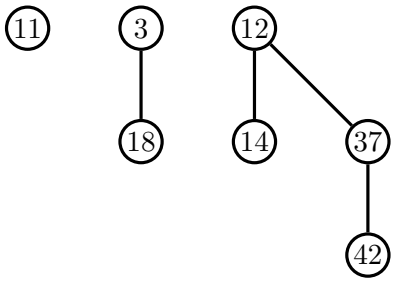


Figure 1. A binomial queue storing 7 integers. The binomial trees are drawn in schematic form.

by the *decrease* operation and at most $O(\lg n)$ violations are allowed to appear in the data structure.

To ensure that the number of heap-order violations do not become too large, violation-reducing transformations are used. The use of these transformations ensures that *decrease* can be supported at the worst-case cost of $O(1)$. The idea behind the transformations is that they perform some incremental work, if necessary. This incremental work guarantees that nodes which violate the heap order are moved upward in the trees.

Constant-cost meld. If the worst-case cost of $O(1)$ for *insert* and *meld* is required the meldable priority queues described in [2] and [4] achieve these bounds. The bootstrapped priority queues described in [4] achieve the worst-case cost of $O(1)$ for *meld* by using a priority queue that supports *meld* at the worst-case cost of $O(\lg n)$. A bootstrapped priority queue is a priority queue that is represented as a pair. Each pair in the bootstrapped structure contains an element and a priority queue, and each priority queue contains pairs. This way the bootstrapped priority queue is a priority queue which can recursively contain other priority queues. Using this kind of recursion, a bootstrapped priority queue transforms *meld* into a special case of *insert*. For the bootstrapped priority queues described in [4], the comparison complexity for *delete-min* is at least $4 \lg n - O(1)$ and *delete* is not supported.

One of the reasons the meldable priority queues in [2] can achieve the worst-case cost of $O(1)$ for *meld* is by maintaining a structure consisting of one tree instead of a forest of trees. For a positive integer r, the *rank* of a meldable tree can be defined as follows: A tree of rank 0 is a single node; for ranks higher than 0, a tree of rank r consists of a node having at least one and at most three subtrees of rank $0, 1, \ldots, r-1$. However, one subtree of the node can have a rank that is non-smaller than r. The rank of the tree containing the root is 0. Given a node in any meldable tree, the number of subtrees having the same rank is maintained by emulating a zeroless number system (see Section 2). The zeroless number system ensures that the number of subtrees having the same rank is within the range $\{1, 2, 3\}$, which again guarantees that the sizes of the trees are exponential with respect to their ranks. The subtrees maintained using the zeroless number system all have ranks which are smaller than the rank of the tree of their parent. Allowing one subtree to have a rank that is non-smaller than the rank of the tree of its parent facilitates the achievement of the worst-case cost of $O(1)$ for *meld*. The comparison complexity for *delete* and *delete-min* of the meldable priority queues described in [2] is $7 \lg n + O(1)$ (the bound is derived in [14]).

2. Number systems

The connection between number systems and data structures is well-known and number systems have been utilized in the design of worst-case efficient data structures for a long time. This connection was, as far as we know,

first made explicit in the seminar notes by Clancy and Knuth [8]. In this section we give a brief survey of how number systems can be used in the design of priority queues.

The following notation and definitions are valid throughout this section. In a positional number system represented by its digits and their corresponding weights, a *numeral representation* is a string of digits $\langle d_0, d_1, \ldots, d_{\ell-1} \rangle$, ℓ being the length of the representation. Here d_0 is the least significant digit and $d_{\ell-1} \neq 0$ the most significant digit. In order of appearance the least significant digit is the *lowest* digit of the string and the most significant digit the *highest* digit of the string. Let $d = \langle d_0, d_1, \ldots, d_{\ell-1} \rangle$; the decimal value of d, denoted $value(d)$, is $\sum_{i=0}^{\ell-1} d_i \cdot w_i$, where w_i is the weight corresponding to d_i. In a b-ary numeral representation, $w_i = b^i$. A numeral representation can have constraints which limits the form a string can take. One way to express these constraints is to use the syntax of regular expressions (see, for example, [1, Section 3.3]).

We will here define a *number system* as a numeral representation together with the corresponding operations on the numbers. A number system can allow the following standard operations:

increment(d, i)**:** Assert that $i \in \{0, 1, \ldots, \ell - 1\}$. Perform $\texttt{++}d_i$ resulting in d', i.e. $value(d') = value(d) + w_i$. Transform d' to a form that fulfils the given constraints, if necessary.

decrement(d, i)**:** Assert that $i \in \{0, 1, \ldots, \ell - 2\}$. Perform $\texttt{--}d_i$ resulting in d', i.e. $value(d') = value(d) - w_i$. Transform d' to a form that fulfils the given constraints, if necessary.

add(d, d')**:** Construct a string d'' fulfilling the given constraints such that $value(d'') = value(d) + value(d')$.

For *increment* the special case where only the least significant digit is operated on usually suffices for the task at hand. Normally, if only this special case has to be handled, the realization of the number system is simpler than that in the general case where an arbitrary digit d_i, $i \in \{0, 1, \ldots, \ell - 1\}$, can be increased. The same applies to *decrement*.

The binary number system is well-known within computer science. The digit set of a *binary representation* is $d_i \in \{0, 1\}$ and their corresponding weight is $w_i = 2^i$, $value(d)$ is $\sum_{i=0}^{\ell} d_i \cdot w_i$, and every string has the form $(0 \mid 1)^*$. If the connection between the binary number system and the number of trees in a collection of trees (for example in a binomial queue) is used, inserting into a priority queue can be realized elegantly by emulating an *increment* in the associated number system. Using this connection the worst-case efficiency of *insert* in a binomial queue is directly related to how far a carry is propagated in the corresponding numeral representation. Using a representation where only the digits 0 and 1 are allowed, *insert* has the worst-case cost of $\Theta(\lg n)$. The reasons for the high worst-case cost is that an *increment* can result in a propagating carry, likewise a *decrement* immediately after an *increment* can result in a propagating borrow. The consequence of using the binary number system in connection with binomial queues is therefore that inserting a node can result in a sequence of tree joins

and removing a node can result in a sequence of tree splits. Therefore, an alternating sequence of insertion and removal of nodes results in a sequence of priority-queue operations all having a $\Theta(\lg n)$ cost.

Using an extra (redundant) digit in the representation can solve the problem of expensive alternating *increment* and *decrement* operations. Having a *redundant representation* makes it possible to have $value(d) = value(d')$ using two different strings. This way, the usage of digits 0, 1, and 2 will make it possible to avoid hitting the same string of digits when increasing and decreasing a number, thereby avoiding an alternating sequence of carries and borrows. Even though an alternating sequence of expensive operations can be avoided by the use of a redundant representation a single operation can still have the worst-case cost of $\Theta(\lg n)$.

Constant-cost increment. A regular redundant representation can be used to avoid the $\Theta(\lg n)$ cost of a single *increment*. A *regular representation* also uses the digit set $d_i \in \{0, 1, 2\}$. However, it also maintains the constraint that every digit 2 is preceded by at least one digit 0; this definition was used in [22]. Using the syntax of regular expressions every regular string has the form $(0 \mid 1 \mid 01^*2)^*$. A substring of the form 01^*2 is called a *block*. Alternatively, the regular representation could also be defined using a more loose constraint stating that between every two 2's there is a 0 or more formally stated if $d_i = 2$ and $d_k = 2$ and $i < k$, then $d_j = 0$ for some $j \in \{i + 1, \ldots, k - 1\}$; this definition was used in [8]. The regularity constraints guarantee that an *increment* can be performed at the worst-case cost of $O(1)$; for a proof of this claim, see [8]. Therefore, if the regular representation is used in connection with binomial queues, the *insert* operation can be performed at the worst-case cost of $O(1)$.

In a regular representation every digit 2 corresponds to a delayed carry and maintaining the regularity of a string can be done using a fix. A *fix* sets $d_i \leftarrow d_i - 2$ and $d_{i+1} \leftarrow d_{i+1} + 1$. Maintaining the regularity of a string in connection with every $increment(d, i)$ is done by fixing the closest delayed carry higher than i in the *carry sequence* (sequence of carries ordered by the digit index i of the digits which the carries are associated with).

The delayed carries can be maintained in first-in last-out (FILO) order, when *increment* only has to be supported at the least significant digit. As a consequence of this order the carry with the lowest digit is accessed first. Using a *carry stack*, increments can be performed as follows (here the strong constraint is used).

1) Fix the first carry if any.
2) Add one.
3) If the least significant digit becomes 2, fix this 2, and if the fix creates a new carry, add this to the carry stack.

The increment of the least significant digit only requires a constant number of digit changes. Therefore, the corresponding *insert* operation in a binomial queue can be performed at the worst-case cost of $O(1)$.

Constant-cost arbitrary increment. The regular number system can also be used to support an *increment* at an arbitrary digit in the string.

An arbitrary *increment* can be performed at the worst-case cost of $O(1)$ as follows (here the loose constraint is used).

1. If $d_i = 0$ or $d_i = 1$ is part of a block, increase d_i by one and fix the 2 ending the block.
2. Otherwise, increase d_i by one and fix d_i.

One way to realize this stronger type of regular number system is to use the guide data structure described in [3] to handle the delayed carries and their corresponding blocks at $O(1)$ worst-case cost. The idea of the guide data structure is, that given a digit position, it can identify and remove the block where this digit is positioned.

Using the fact that an arbitrary *increment* is supported, the *add* operation can be realized as follows (again the loose constraint is used). Given two regular strings of lengths k and ℓ, assuming without loss of generality that $k \leq \ell$, process the digits of the shorter string one digit at a time and update the digits of the longer string accordingly. Let d_i denote the digit at position i in the shorter string and d'_i the digit in the longer string.

- $d_i = 0$: Do nothing.
- $d_i = 1$:
  - If $d'_i = 0$ or $d'_i = 1$ is part of a block, increase d'_i by one and fix the 2 ending the block.
  - Otherwise, increase d'_i by one and fix d'_i.
- $d_i = 2$: Carry out the previous case twice.

It follows from the definition of the loose regularity constraint that the sum of the digits of a string having length k is at most $k + 1$. Therefore, if k is the length of the shorter string in an addition, at most $k + 1$ fixes are performed. Therefore, for two binomial queues of sizes m and n, $m \leq n$, the corresponding *meld* operation involves at most $\lg m + 2$ element comparisons and has the worst-case cost of $O(\lg m)$.

Constant-cost increment and decrement. The above-mentioned regular representation cannot be used, when both *increment* and *decrement* have to be supported at the worst-case cost of $O(1)$. However, using a digit set of size four (see [8, p. 56 ff.] and [18]) in a regular number system, where every digit d_i has the corresponding weight $w_i = 2^i$, the constant worst-case cost for both *increment* and *decrement* can be obtained. The digit set used can be $d_i \in \{0, 1, 2, 3\}$ or $d_i \in \{1, 2, 3, 4\}$. If the digit set $d_i \in \{1, 2, 3, 4\}$ are used the representation is zeroless. A *zeroless regular representation* having the digit set $d_i \in \{1, 2, 3, 4\}$ has the property that between any two digits equal to 4 there is a digit other than 3, and between any two digits equal to 1 there is a digit other than 2, except when one of the digits equal to 1 is the most significant digit.

In a zeroless regular representation having the digit set $d_i \in \{1, 2, 3, 4\}$, every digit 4 corresponds to a delayed carry and every digit 1 to a delayed borrow. The regularity of a string under operation can be maintained as follows. In connection with every *increment*(d, i) or *decrement*(d, i), process the closest delayed carry or borrow higher than i in the carry/borrow

sequence by performing a fix. A fix processes a delayed carry by setting $d_i \leftarrow d_i - 2$ and $d_{i+1} \leftarrow d_{i+1} + 1$ and a delayed borrow by setting $d_i \leftarrow d_i + 2$ and $d_{i+1} \leftarrow d_{i+1} - 1$.

The delayed carries and borrows can be maintained in FILO order, when *increment* and *decrement* only have to be supported at the least significant digit. As a consequence, the carry/borrow with the lowest digit is accessed first. Using a *carry/borrow stack*, *increment* and *decrement* can be performed as follows:

1) Fix the first carry or borrow if any.
2) Add or subtract one as desired.
3) If the least significant digit becomes 4 or 1, fix this digit, and if the fix creates a new carry or borrow, add this to the carry/borrow stack.

Both *increment* and *decrement* of the least significant digit only requires a constant number of digit changes. Therefore, the corresponding *insert* and *extract* operations in a binomial queue can be performed at the worst-case cost of $O(1)$. The correctness of the worst-case bounds of *increment* and *decrement* can be proved by showing that both *increment* and *decrement* maintain the representation regular [13].

Other number systems and data structures. A zeroless regular numeral representation with the digit set $d_i \in \{1, 2, 3\}$ having the corresponding weight $w_i = 2^i$ and where every string has the form $(1 \mid 2 \mid 12^*3)^*$ was used by Brodal in [2]. Observe that these meldable priority queues were reviewed in Section 1.

The use of the above-mentioned number systems is not only connected to binomial queues but can also be used in connection with other priority queue structures like selection trees [19, Section 5.2.3 and 5.4.1], complete binary leaf trees [17], and pennants [23].

For some priority queues other types of number systems may be a more natural choice. For example, a skew-binary number system [21] can be used to obtain a constant amount of digit changes per *increment* and *decrement* operation. In the *skew-binary numeral representation* $d_i \in \{0, 1, 2\}$ and the corresponding weight is $w_i = 2^{i+1} - 1$. The skew-binary numeral representation is a redundant representation whereas the canonical skew-binary numeral representation [21] has an unique representation. In a *canonical skew-binary representation* there is the following constraint on the digits: If $d_j = 2$, then $d_i = 0$ for all $i \in \{0, 1, \ldots, j - 1\}$. For an example of a priority queue that uses the canonical skew-binary number system, see [4].

3. Outline of this thesis

Here we summarize the results obtained in the individual papers, see also Table 1 and Table 2.

1. Amr Elmasry, Claus Jensen, and Jyrki Katajainen, Two skew-binary numeral systems and one application, *Theory of Computing Systems*, vol. 50, no. 1 (2012), 185–211.

Table 1. The worst-case comparison complexity for the operations of the designed priority queues; m and n are the sizes of the data structures and $m < n$. The symbol – means that the worst-case bound of the operation is not derived in the paper.

Paper	*find-min*	*insert*	*extract*	*decrease*	*meld*	*delete*
1	0^a	$O(1)$	$O(1)$	–	$O(\lg^2 m)$	$3\lg n + O(1)$
2	0^b	$O(1)$	$0^{\ c}$	–	–	$\lg n + O(1)$
3	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(\lg m)$	$\lg n + O(\lg \lg n)$
4	0^b	$O(1)$	$O(1)$	$O(1)$	–	$\lg n + O(\lg \lg n)$
5	0^d	$O(1)$	–	–	$O(1)$	$3\lg n + O(1)$
6	0^d	$O(1)$	–	–	$O(1)$	$2\lg n + O(1)$

[a] A global minimum pointer is used, therefore *find-min* can be performed without element comparisons.

[b] The comparison complexity proved in the paper is $O(1)$. However, using a global minimum pointer we can reduce the comparison complexity to 0.

[c] Nodes are extracted using the split operation, therefore no element comparisons are performed.

[d] A minimum element is kept in a special node, therefore *find-min* can be performed without element comparisons.

2. Amr Elmasry, Claus Jensen, and Jyrki Katajainen, Multipartite priority queues, *ACM Transactions on Algorithms*, vol. 5, no. 1 (2008), article 14.
3. Amr Elmasry, Claus Jensen, and Jyrki Katajainen, Two-tier relaxed heaps, *Acta Informatica*, vol. 45, no. 3 (2008), 193–210.
4. Amr Elmasry, Claus Jensen, and Jyrki Katajainen, On the power of structural violations in priority queues, *Proceedings of the 13th Computing: The Australasian Theory Symposium*, Conferences in Research and Practice in Information Technology 65, Australian Computer Society, Inc. (2007), 45–53.
5. Claus Jensen, A note on meldable heaps relying on data-structural bootstrapping, CPH STL Report 2009-2, Department of Computer Science, University of Copenhagen (2009).
6. Amr Elmasry, Claus Jensen, and Jyrki Katajainen, Strictly-regular number system and data structures, *Proceedings of the 12th Scandinavian Symposium and Workshops on Algorithm Theory*, Lecture Notes in Computer Science 6139, Springer-Verlag (2010), 26–37.
7. Amr Elmasry, Claus Jensen, and Jyrki Katajainen, Two new methods for constructing double-ended priority queues from priority queues, *Computing*, vol. 83, no. 4 (2008), 193–204.

In paper 1, we use two skew-binary number systems and a forest of pointer-based binary heaps to derive two different data structures that support *find-min* and *insert* at the worst-case cost of $O(1)$, and *delete* at logarithmic cost.

The first skew-binary number system, which we call the magical skew system, has a digit set of size five, skew weights, and an unconventional carry-propagation mechanism. The magical skew system supports *increment* and

decrement at a constant worst-case cost and each operation involves at most four digit changes. Using the magical skew system in combination with pointer-based binary heaps we obtain a data structure that supports *find-min* and *insert* at the worst-case cost of $O(1)$, and *delete* at logarithmic worst-case cost using at most $6 \lg n$ element comparisons.

The second skew-binary number system, which we call the regular skew system, has a digit set of size three and skew weights. The regular skew system supports *increment* and *decrement* at constant worst-case cost and each operation involves at most two digit changes. The system supports *add* at the worst-case cost of $O(\lg^2 k)$, where k is the length of the shorter representation, and the operation involves at most $O(k)$ digit changes. We use the regular skew system together with pointer-based binary heaps to create a data structure that supports *find-min* and *insert* at the worst-case cost of $O(1)$; *delete* at worst-case logarithmic cost using at most $3 \lg n + O(1)$ element comparisons; and *meld* at the worst-case cost of $O(\lg^2 m)$, m denoting the number of elements stored in the smaller priority queue. This priority queue improves all bounds of the above-mentioned priority queue.

The two priority queues both use *extract* in *delete* to preserve the structure of the tree from which a node is deleted. This use of *extract* to keep the structure intact is also used in papers 2, 3, 4, and 7.

Observe that if all priority queues performing *find-min* and *insert* at the worst-case cost of $O(1)$ and *delete* at logarithmic worst-case cost are considered we do not improve the best known comparison-complexity bounds for *delete*. However, for binary heaps we do push the limit as it has been shown that the array-based implementation of a binary heap [25] has $\Omega(\lg \lg n)$ as a lower bound on the worst-case complexity of *insert* [15].

In paper 2, we present multipartite priority queues that for *find-min*, *insert*, and *delete* has a comparison-complexity bound that is optimal[1] up to the constant additive terms, and perform *find-min* and *insert* at the worst-case cost of $O(1)$. Using multipartite priority queues we produce an adaptive sorting algorithm that is constant-factor optimal (i.e. the constant factor in the leading term is optimal) for several measures of disorder.

Multipartite priority queues consist of the following components which through their interaction facilitate the achieved bounds:

1. The main store is a binomial queue that holds most of the elements.
2. The upper store is the component that maintains the order among the roots of the binomial queue in the main store. The order is maintained using prefix-minimum pointers. For a given rank a prefix-minimum pointer points to the root which holds the minimum element among the roots with equal or smaller rank.
3. The insert buffer is a binomial queue which handles all insert operations.

[1] Given the comparison-based lower bound for sorting (see [9, Chapter 9]), it follows that *delete* has to perform at least $\lg n - O(1)$ element comparisons, if *find-min* and *insert* only perform $O(1)$ element comparisons.

4. The reservoir is a tree from which nodes are extracted.
5. The floating tree is created when the insert buffer becomes too large. The tree is merged into the main store incrementally.

We use the extraction of an element from the reservoir to preserve the structure of the tree from which a node is deleted. Using extraction based *delete* in combination with the prefix-minimum pointers of the upper store make it possible to obtain the comparison-complexity bound of $\lg n + O(1)$ for *delete*. The insert buffer and the floating tree are used to avoid updating the prefix-minimum pointers every time an insert operation is performed.

For priority queues performing *find-min* and *insert* at the worst-case cost of $O(1)$ and *delete* at logarithmic worst-case cost the best known comparison-complexity bounds for *delete* were $2\lg n + O(1)$ for minimum-remembering run-relaxed heaps [10] and $\lg n + O(\lg \lg n)$ for layered heaps [11]. Multipartite priority queues improve the comparison-complexity bound for *delete* to $\lg n + O(1)$.

Paper 2 is a journal version of the conference paper on layered heaps [11] written by Amr Elmasry. Observe that the data structure has been heavily redesigned and that the parts that are still used (the parts that gave improved worst-case bounds) were developed in a collaboration between the authors (see the acknowledgement note in the conference paper [11]).

In paper 3, we present two-tier relaxed heaps for which the comparison complexity of *delete* is constant-factor optimal, and which perform *find-min*, *insert*, and *decrease* at the worst-case cost of $O(1)$.

Two-tier relaxed heaps consist of two modified run-relaxed heaps, one holding the elements and another holding pointers to the minimum candidates. To support *insert* and *extract* at worst-case constant cost, a zeroless regular number system using a digit set of size four is utilized. The heap holding pointers to the minimum candidates uses lazy deletions (marking). Incremental global rebuilding is used to remove the markings when the number of markings becomes to large.

For priority queues performing *find-min*, *insert*, and *decrease* at the worst-case cost of $O(1)$ and *delete* at logarithmic worst-case cost the best known comparison-complexity bounds for *delete* were $3\lg n + O(1)$ for minimum-remembering run-relaxed heaps [10]. Two-tier relaxed heaps improve this bound to $\lg n + 3\lg \lg n + O(1)$.

In paper 4, we present a priority queue called a pruned binomial queue that for *delete* has a comparison-complexity bound that is constant-factor optimal, and perform *find-min*, *insert*, and *decrease* at the worst-case cost of $O(1)$. Furthermore, we show that using structural violations it is possible to obtain worst-case comparison-complexity bounds comparable to those obtained using heap-order violations in two-tier relaxed heaps.

The pruned binomial queues use the same two-tier structure as the two-tier relaxed heaps in paper 3. In the pruned binomial queues having the best comparison-complexity bound for *delete*, we emulate heap-order violations using a shadow structure. When a violation occurs we replace the violating

Table 2. The worst-case comparison complexity for the operations of the designed double-ended priority queues; m and n are the sizes of the data structures and $m < n$. The symbol – means that the worst-case bound of the operation is not derived in the paper.

Paper	*find-min*/ *find-max*	*insert*	*extract*	*meld*	*delete*
7	$O(1)$	$O(1)$	$O(1)$	–	$\lg n + O(1)$
7	$O(1)$	$O(1)$	$O(1)$	$O(\lg m)$	$\lg n + O(\lg \lg n)$

node with a placeholder node and move the node and its subtree to the shadow structure.

In paper 5, we present a meldable priority queue that uses data-structural bootstrapping to obtain a good comparison-complexity bound for *delete*, while achieving a constant worst-case cost for *find-min*, *insert*, and *meld*.

The data structure is created by bootstrapping a binomial queue that supports *insert* at constant worst-case cost and *meld* at logarithmic worst-case cost. The structure obtains a $3 \lg n + O(1)$ comparison-complexity bound for *delete* improving the bound of $7 \lg n + O(1)$ achieved by Brodal for his meldable priority queues (the bound of Brodal's meldable priority queues is derived in [14]). The bound of $3 \lg n + O(1)$ for *delete* is improved in paper 6 to $2 \lg n + O(1)$. However, the priority queue in paper 6 is more involved and mainly of theoretical interest.

In paper 6, we present a new number system, which we call the strictly-regular system. The system supports the operations: *increment*, *decrement*, *cut*, *concatenate*, and *add* having a digit set of size three. The strictly-regular system is superior to both the regular number system which with a digit set of size three only supports *increment* and a regular number system which with a digit set of size four can support both *increment* and *decrement*.

We demonstrate the potential of the number system by using it to modify Brodal's meldable priority queues obtaining a comparison-complexity bound of $2 \lg n + O(1)$ for *delete*, improving the original bound of $7 \lg n + O(1)$, while maintaining the constant cost of *find-min*, *insert*, and *meld*.

In paper 7, we describe two data-structural transformations to construct double-ended priority queues from priority queues. Using the first transformation we obtain a double-ended priority queue that for *find-min*, *insert*, and *delete* has a comparison-complexity bound that is optimal up to the constant additive terms, and perform *find-min*, *find-max*, and *insert* at the worst-case cost of $O(1)$. Using the second transformation we obtain a double-ended priority queue that supports *find-min*, *find-max*, and *insert* at the worst-case cost of $O(1)$; *meld* at the worst-case cost of $O(\min\{\lg m, \lg n\})$; and that for *delete* has a comparison-complexity bound that is constant-factor optimal. The two data-structural transformations are general transformations, but to obtain the above-mentioned bounds we use the priority queues developed in papers 1 and 2.

In the first transformation we use a special pivot element to partition the elements of the double-ended priority queue into three collections (main-

tained as priority queues) containing the elements smaller than, equal to, and larger then the pivot element. Using this partition of the elements we can delete an element only touching one priority queue. To maintain the partitioning balanced we rebuild the data structure after a linear number of operations. The rebuilding is done incrementally to obtain the derived worst-case bounds. In the second transformation we use the total correspondence approach (see [7]) and the fact that the underlying priority queue supports *decrease*. Utilizing the fact that *decrease* in two-tier relaxed heaps (paper 3) has a constant worst-case cost, we obtain a double-ended priority queue *delete* operation which comparison-complexity bound is dominated by the priority queue *delete* operation. Using the previously best known transformations to construct double-ended priority queues from priority queues [7, 20] the worst-case bound for *delete* becomes at least twice the bound of the *delete* operation of the underlying priority queues.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools*, 2nd Edition, Pearson Education, Inc. (2007).

[2] G. S. Brodal, Fast meldable priority queues, *Proceedings of the 4th International Workshop on Algorithms and Data Structures*, *Lecture Notes in Computer Science* **955**, Springer-Verlag (1995), 282–290.

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

[4] G. S. Brodal and C. Okasaki, Optimal purely functional priority queues, *Journal of Functional Programming* **6**, 6 (1996), 839–857.

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

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

[7] K.-R. Chong and S. Sahni, Correspondence-based data structures for double-ended priority queues, *The ACM Journal of Experimental Algorithmics* **5** (2000), article 2.

[8] M. J. Clancy and D. E. Knuth, A programming and problem-solving seminar, Technical Report STAN-CS-77-606, Department of Computer Science, Stanford University (1977).

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

[10] 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**, 11 (1988), 1343–1354.

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

[12] A. Elmasry, C. Jensen, and J. Katajainen, Multipartite priority queues, *ACM Transactions on Algorithms* **5**, 1 (2008), Acticle 14.

[13] A. Elmasry, C. Jensen, and J. Katajainen, Two-tier relaxed heaps, *Acta Informatica* **45**, 3 (2008), 193–210.

[14] A. Elmasry, C. Jensen, and J. Katajainen, Strictly-regular number system and data structures, *Proceedings of the 12th Scandinavian Symposium and Workshops on Algorithm Theory*, *Lecture Notes in Computer Science* **6139**, Springer-Verlag (2010), 26–37.

[15] G. H. Gonnet and J. I. Munro, Heaps on heaps, *SIAM Journal on Computing* **15**, 4 (1986), 964–971.
[16] D. B. Johnson, Priority queues with update and finding minimum spanning trees, *Information Processing Letters* **4**, 3 (1975), 53–57.
[17] A. Kaldewaij and V. J. Dielissen, Leaf trees, *Science of Computer Programming* **26**, 1–3 (1996), 149–165.
[18] H. Kaplan, N. Shafrir, and R. E. Tarjan, Meldable heaps and Boolean union-find, *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, ACM (2002), 573–582.
[19] D. E. Knuth, *Sorting and Searching*, *The Art of Computer Programming* **3**, 2nd Edition, Addison Wesley Longman (1998).
[20] C. Makris, A. Tsakalidis, and K. Tsichlas, Reflected min-max heaps, *Information Processing Letters* **86**, 4 (2003), 209–214.
[21] E. W. Myers, An applicative random-access stack, *Information Processing Letters* **17**, 5 (1983), 241–248.
[22] C. Okasaki, *Purely Functional Data Structures*, Cambridge University Press (1998).
[23] J.-R. Sack and T. Strothotte, A characterization of heaps and its applications, *Information and Computation* **86**, 1 (1990), 69–86.
[24] J. Vuillemin, A data structure for manipulating priority queues, *Communications of the ACM* **21**, 4 (1978), 309–315.
[25] J. W. J. Williams, Algorithm 232: Heapsort, *Communications of the ACM* **7**, 6 (1964), 347–348.

Two Skew-Binary Numeral Systems and One Application*

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

[1] Department of Computer Science, University of Copenhagen, Denmark
[2] The Royal Library, Copenhagen, Denmark

Abstract. We introduce two numeral systems, the magical skew system and the regular skew system, and contribute to their theory development. For both systems, increments and decrements are supported using a constant number of digit changes per operation. Moreover, for the regular skew system, the operation of adding two numbers is supported efficiently. Our basic message is that some data-structural problems are better formulated at the level of a numeral system. The relationship between number representations and data representations, as well as operations on them, can be utilized for an elegant description and a clean analysis of algorithms. In many cases, a pure mathematical treatment may also be interesting in its own right. As an application of numeral systems to data structures, we consider how to implement a priority queue as a forest of pointer-based binary heaps. Some of the number-representation features that influence the efficiency of the priority-queue operations include weighting of digits, carry-propagation and borrowing mechanisms.

Keywords. Numeral systems, data structures, priority queues, binary heaps

1 Introduction

The interrelationship between numeral systems and data structures is efficacious. As far as we know, the issue was first discussed in the paper by Vuillemin on

A preliminary version of this paper entitled "The magic of a number system" [1] was presented at the 5th International Conference on Fun with Algorithms held on Ischia Island in June 2010.

A. Elmasry was supported by the Alexander von Humboldt Foundation and the VELUX Foundation. J. Katajainen was partially supported by the Danish Natural Science Research Council under contract 09-060411 (project "Generic programming—algorithms and tools").

binomial queues [2] and the seminar notes by Clancy and Knuth [3]. However, in many write-ups this connection has not been made explicit.

In a positional numeral system, a string $\langle d_0, d_1, \ldots, d_{\ell-1} \rangle$ of *digits* d_i, $i \in \{0, 1, \ldots, \ell - 1\}$, is used to represent an integer, ℓ being the length of the representation. By convention, d_0 is the least-significant digit, $d_{\ell-1}$ the most-significant digit, and $d_{\ell-1} \neq 0$. If w_i is the *weight* of d_i, the string represents the value $\sum_{i=0}^{\ell-1} d_i w_i$. In *binary systems* $w_i = 2^i$, and in *skew binary systems* $w_i = 2^{i+1} - 1$. In the *standard binary system* $d_i \in \{0, 1\}$, in a *redundant binary system* $d_i \in \{0, 1, 2\}$, and in the *zeroless variants* $d_i \neq 0$. Other numeral systems include the *regular system* [3], which is a redundant binary system where $d_i \in \{0, 1, 2\}$ conditional on that between every two 2's there is at least one 0; and the *canonical skew system* [4], which is a skew binary system where $d_i \in \{0, 1\}$ except that the first non-zero digit may be 2. These numeral systems and some others, together with their applications to data structures, are discussed in [5, Chapter 9].

The key operations used in the manipulation of number representations include a modification of a specified digit in a number, e.g. by increasing or decreasing a digit by one, and an addition of two numbers. Sometimes, it may also be relevant to support the cutting of a number in two numbers and the concatenation of two numbers. For all operations, the resulting representations must still obey the rules governing the numeral system. An important measure of efficiency is the number of digit changes made by each operation.

As an application, we consider *addressable* and *meldable priority queues*, which store one element per node, and support the following operations:

find-min(Q)**.** Return a handle (pointer) to the node with a minimum element in priority queue Q.

insert(Q, x)**.** Insert node x, already storing an element, into priority queue Q.

borrow(Q)**.** Remove an unspecified node from priority queue Q, and return a handle to that node.

delete-min(Q)**.** Remove a node from priority queue Q with a minimum element, and return a handle to that node.

delete(Q, x)**.** Remove node x from priority queue Q.

meld(Q_1, Q_2)**.** Create and return a new priority queue that contains all the nodes of priority queues Q_1 and Q_2. This operation destroys Q_1 and Q_2.

Since *find-min* is easily accomplished at $O(1)$ worst-case cost by maintaining a pointer to the node storing the current minimum, *delete-min* can be implemented at the same asymptotic worst-case cost as *delete* by calling *delete* with the handle returned by *find-min*. Hence, from now on, we shall concentrate on the operations *insert*, *borrow*, *delete*, and *meld*.

Employing the developed numeral systems, we describe two priority-queue realizations, both implemented as a forest of pointer-based binary heaps, which support *insert* at $O(1)$ worst-case cost and *delete* at $O(\lg n)$ worst-case cost, n denoting the number of elements stored in the data structure prior to the operation, and $\lg n$ being a shorthand for $\log_2(\max\{2, n\})$. In contrast, for the array-based implementation of a binary heap [6], $\Omega(\lg \lg n)$ is known to be a

Table 1. The numeral systems employed in some priority queues and their effect on the complexity of *insert*. All the mentioned structures support *find-min* at $O(1)$ worst-case cost and *delete* at $O(\lg n)$ worst-case cost, where n is the size of the data structure.

Digit set	**Forest of binomial trees**	**Forest of pennants**	**Forest of perfect binary heaps**
$\{0,1\}$	$O(\lg n)$ worst case $O(1)$ amortized [2]	$O(\lg^2 n)$ worst case [8]	$O(\lg^2 n)$ worst case [folklore]
$\{0,1\}$, there may be one 2	$O(1)$ worst case [9]	$O(\lg n)$ worst case [10]	$O(\lg n)$ worst case [§6][a] $O(1)$ amortized [11, 12]
$\{0,1,2\}$	$O(1)$ worst case [13]	$O(\lg n)$ worst case [10] $O(1)$ worst case [8]	$O(1)$ worst case [§6][a]
$\{1,2,3,4\}$	$O(1)$ worst case [14][a]		
$\{0,1,2,3,4\}$			$O(1)$ worst case [§4]

[a] *borrow* has $O(1)$ worst-case cost.

lower bound on the worst-case complexity of *insert* [7]. Our second realization supports *borrow* at $O(1)$ worst-case cost and *meld* at $O(\lg^2 m)$ worst-case cost, m denoting the number of elements stored in the smaller priority queue. We summarize relevant results related to the present study in Table 1.

A binomial queue is a forest of heap-ordered binomial trees [2]. If the queue stores n elements and the binary representation of n contains a 1-bit at position i, $i \in \{0, 1, \ldots, \lfloor \lg n \rfloor\}$, the queue contains a tree of size 2^i. In the binary numeral system, an addition of two 1-bits at position i results in a 1-bit carry to position $i+1$. Correspondingly, in a binomial queue two trees of size 2^i are linked resulting in a tree of size 2^{i+1}. For binomial trees, this linking is possible at $O(1)$ worst-case cost. Since *insert* corresponds to an increment of an integer, *insert* would have logarithmic worst-case cost due to the propagation of carries. Instead of relying on the binary system, some of the other specialized variants could be used to avoid cascading carries. That way, a binomial queue can support *insert* at $O(1)$ worst-case cost [9, 13, 14]. A binomial queue based on a zeroless system, where $d_i \in \{1, 2, 3, 4\}$, also supports *borrow* at $O(1)$ worst-case cost [14].

An approach similar to that used for binomial queues has also been considered for binary heaps. The components in this case are either *perfect binary heaps* [11, 12] or *pennants* [10]. A perfect binary heap is a heap-ordered complete binary tree, and accordingly is of size $2^{i+1} - 1$ where $i \geq 0$. A pennant is a heap-ordered tree whose root has one subtree that is a complete binary tree, and accordingly is of size 2^i where $i \geq 0$. In contrast to binomial trees, the worst-case cost of linking two pennants of the same size is logarithmic, not constant. To link two perfect binary heaps of the same size, we even need to have an additional node, and if this node is arbitrarily chosen, the worst-case cost per link is also logarithmic. When perfect binary heaps are used, it is natural to rely on a skew binary system. Because of the linking cost, this approach achieves $O(\lg n)$ worst-case cost [10] and $O(1)$ amortized cost per *insert* [11, 12]. By implementing the

linkings incrementally with the upcoming operations, $O(1)$ worst-case cost per *insert* is achievable for pennants [8].

The remainder of this paper is organized as follows. In Section 2, we review the canonical skew system introduced in [4]. Due to its conventional carry-propagation mechanism, in our application a digit change may involve a costly linking. We propose two ways of avoiding this computational bottleneck. First, in Section 3, we introduce the magical skew system that uses five symbols, skew weights, and an unconventional carry-propagation mechanism. It may look mysterious why this numeral system works as effectively as it does; to answer this question, we use an automaton to model the behaviour of the numeral system. We discuss the application of the magical skew system to priority queues in Section 4. Second, in Section 5, we modify the regular system discussed in [3] to get a better fit for our application. Specifically, we use skew binary weights, and support incremental digit changes (this idea was implicit in [8]) making the incremental execution of linkings possible without breaking the invariants of this numeral system. We discuss the application of the regular skew system to priority queues in Section 6. To conclude, in Section 7, we briefly summarize the results proved and the issues left open.

2 Canonical Skew System: A Warm-Up

In this section, we review the *canonical skew system* introduced in [4]. A positive integer n is represented as a string $\langle d_0, d_1, \ldots, d_{\ell-1} \rangle$ of digits, least-significant digit first, such that

- $d_i \in \{0, 1, 2\}$ for all $i \in \{0, 1, \ldots, \ell - 1\}$, and $d_{\ell-1} \neq 0$,
- if $d_j = 2$, then $d_i = 0$ for all $i \in \{0, 1, \ldots, j - 1\}$,
- $w_i = 2^{i+1} - 1$ for all $i \in \{0, 1, \ldots, \ell - 1\}$, and
- the value of n is $\sum_{i=0}^{\ell-1} d_i w_i$.

In other words, every string has at most one 2; if a 2 exists, it is the first non-zero digit. In this system, every number is represented uniquely [4].

From the perspective of the related applications (see, for example, [4, 9]), it is important that such number representations efficiently support increments, decrements, and additions. Here, we shall only consider increments and decrements; additions can be realized by converting the numbers to binary form, relying on ordinary binary addition, and converting the result back to skew binary form.

In a computer realization, we would rely on a *sparse representation* [5, Section 9.1], which keeps all non-zero digits together with their positions in a singly-linked list. Only one-way linking is necessary since both the increment and decrement operations access the string of digits from the front, by adding new digits or removing old digits. The main reason for using a sparse representation is to have an immediate access to the first non-zero digit. Consequently, the overall cost of the operations will be proportional to the number of non-zero digits accessed.

In this system, increments are performed as follows:

Algorithm *increment*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle)$

```
1:  let d_j be the first non-zero digit, if it exists
2:  if d_j exists and d_j = 2
3:      d_j ← 0
4:      increase d_{j+1} by 1
5:  else
6:      increase d_0 by 1
```

Clearly, this procedure increases the value of the represented number by one. Also, by a straightforward case analysis, it can be shown that the procedure maintains the representation canonical.

Decrements are equally simple:

Algorithm *decrement*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle)$

```
1:  assert ⟨d_0, d_1, ..., d_{ℓ-1}⟩ is not empty
2:  let d_j be the first non-zero digit
3:  decrease d_j by 1
4:  if j ≠ 0
5:      d_{j-1} ← 2
```

As a result of the operation, the value of the represented number is reduced by $2^{j+1}-1$ and increased by $2(2^j-1)$, so the total change is -1 as it should be. It is again easy to check that this operation maintains the representation canonical.

The efficiency of the operations and the representation can be summarized as follows.

Theorem 1 ([4]). *The canonical skew system supports increments and decrements at $O(1)$ worst-case cost, and each operation involves at most two digit changes. The amount of space needed for representing a positive integer n while supporting these modifications is $O(\lg n)$.*

Remark 1. With respect to the number of digit changes made, the canonical skew system is very efficient. However, in an application domain, the data structure relying on this system is not necessarily efficient since each digit change may require a costly data-structural operation (cf. Section 6). □

3 Magical Skew System

In this section, we introduce a new skew-binary numeral system, which uses an unconventional carry-propagation mechanism. Because of the tricky correctness proof, we call the system *magical*. In this system, we represent a positive integer n as a string $\langle d_0, d_1, \ldots, d_{\ell-1}\rangle$ of digits, least-significant digit first, such that

– $d_i \in \{0, 1, 2, 3, 4\}$ for all $i \in \{0, 1, \ldots, \ell - 1\}$, and $d_{\ell-1} \neq 0$,
– $w_i = 2^{i+1} - 1$ for all $i \in \{0, 1, \ldots, \ell - 1\}$, and
– the value of n is $\sum_{i=0}^{\ell-1} d_i w_i$.

Remark 2. In general, a skew binary system that uses five symbols is redundant, i.e. there is possibly more than one representation for the same integer. However, the operational definition of the magical skew system and the way the operations are performed guarantee a unique representation for any integer. □

We define two operations on strings of digits: An increment increases the corresponding value by one, and a decrement decreases the value by one. We define the possible representations of numbers operationally: The empty string ε represents the integer 0, and a positive integer n is represented by the string obtained by starting from ε and performing the increment operation n times. Hence, to compute the representation of n, a naive algorithm performing n increments has $O(n)$ cost since each increment involves a constant amount of work.

We say that a string $\langle d_0, d_1, \ldots, d_{\ell-1} \rangle$ of digits is *valid* if $d_i \in \{0, 1, 2, 3, 4\}$ for each $i \in \{0, 1, \ldots, \ell - 1\}$. The most interesting part of our correctness proofs is to show that increments and decrements retain the strings valid.

In a computer realization, we use a doubly-linked list to record the digits of a string. This list should grow when the string gets a new non-zero last digit and shrink when the last digit becomes 0. Such a representation is called *dense* [5, Section 9.1].

Remark 3. The number of digits in the string representing a positive integer may only change by one per operation. □

A digit is said to be *high* if it is either 3 or 4, and *low* if it is either 0 or 1. For efficiency reasons, we also maintain two singly-linked lists to record the positions of high and low digits, respectively.

Remark 4. The lists recording the positions of high and low digits should be updated after every operation. Only the first two items per list may change per operation. Accordingly, it is sufficient to implement both lists singly-linked. □

3.1 Increments

Assume that d_j is high. A *fix* for d_j is performed as follows:

Algorithm *fix*$(\langle d_0, d_1, \ldots, d_{\ell-1} \rangle, j)$

```
1: assert d_j is high
2: decrease d_j by 3
3: increase d_{j+1} by 1
4: if j ≠ 0
5:     increase d_{j−1} by 2
```

Remark 5. Since $w_0 = 1$, $w_1 = 3$, and $3w_i = w_{i+1} + 2w_{i-1}$ for $i \geq 1$, a fix does not change the value of the represented number. □

The following pseudo-code summarizes the actions to increment a number.

Algorithm *increment*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle)$

1: increase d_0 by 1
2: let d_j be the first digit where $d_j \in \{3, 4\}$, if it exists
3: **if** d_j exists
4: $\quad$ *fix*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, j)$

Remark 6. The integers from 1 to 30 are represented by the following strings in our system: 1, 2, 01, 11, 21, 02, 12, 22, 03, 301, 111, 211, 021, 121, 221, 031, 302, 112, 212, 022, 122, 222, 032, 303, 113, 2301, 0401, 3111, 1211, 2211. □

Remark 7. If we do not insist on performing the fix at the first high digit, the representation may become invalid. For example, starting from 22222, which is valid, two increments will subsequently give 03222 and 30322. If we now repeatedly fix the second 3 in connection with the forthcoming increments, after three more increments we will end up at 622201. □

As for the correctness, we need to show that by starting from 0 and applying any number of increments, in the produced string every digit satisfies $d_i \in \{0, 1, 2, 3, 4\}$. When fixing d_j, although we increase d_{j-1} by 2, no violation is possible as d_{j-1} was at most 2 before the increment. So, a violation would only be possible if, before the increment, d_0 or d_{j+1} was 4.

To show that our algorithms are correct, we use some notions of the theory of automata and formal languages. We use d^* to denote the string that contains zero or more repetitions of the digit d. Let $\mathcal{S} = \{S_1, S_2, \ldots\}$ and $\mathcal{T} = \{T_1, T_2, \ldots\}$ be two sets of strings of digits. We use $\mathcal{S} \mid \mathcal{T}$ to denote the set containing all the strings in $\mathcal{S}$ and $\mathcal{T}$. We write $\mathcal{S} \subseteq \mathcal{T}$ if for every $S_i \in \mathcal{S}$ there exists $T_j \in \mathcal{T}$ such that $S_i = T_j$, and we write $\mathcal{S} = \mathcal{T}$ if $\mathcal{S} \subseteq \mathcal{T}$ and $\mathcal{T} \subseteq \mathcal{S}$.

We also write $S \xrightarrow{+} T$ indicating that the string T results by applying an increment operation to S, and we write $\mathcal{S} \xrightarrow{+} \mathcal{T}$ if for each $S_i \in \mathcal{S}$ there exists $T_j \in \mathcal{T}$ such that $S_i \xrightarrow{+} T_j$. Furthermore, we write $\overline{S}$ for a string that results from S by increasing its first digit by one without performing a fix, and $\overline{\mathcal{S}}$ for $\{\overline{S_1}, \overline{S_2}, \ldots\}$. To capture the intricate structure of allowable strings, we define the following rewriting rules, each specifying a set of strings.

$$\tau \stackrel{\text{def}}{=} 2^* \mid 2^*1 \tag{1}$$

$$\alpha \stackrel{\text{def}}{=} 2^*1\gamma \tag{2}$$

$$\beta \stackrel{\text{def}}{=} 2^* \mid 2^*1\tau \mid 2^*3\psi \tag{3}$$

$$\gamma \stackrel{\text{def}}{=} 1 \mid 2\tau \mid 3\beta \mid 4\psi \tag{4}$$

$$\psi \stackrel{\text{def}}{=} 0\gamma \mid 1\alpha \tag{5}$$

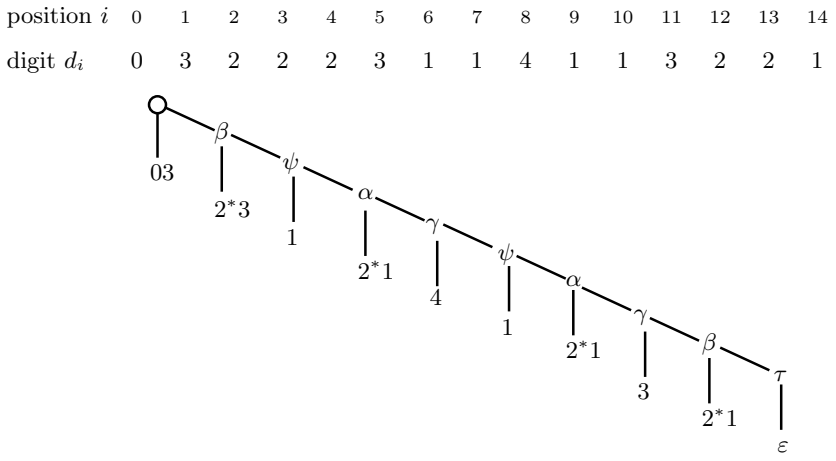


Fig. 1. The unique representation of the integer 100 000 in the magical skew system and its syntactic structure in the form of a parse tree.

Remark 8. The aforementioned rewriting rules can be viewed as production rules of an extended context-free grammar, i.e. one that allows arbitrary regular expressions in the definition of production rules. Actually, it is not difficult to convert these rules into a strictly right-regular form. As an illustration, the syntactic structure of a string in the magical skew system is given in Fig. 1. □

The definitions of the rewriting rules immediately imply that

$$\begin{aligned}
\tau &= \varepsilon \mid 1 \mid 2\tau \\
\overline{\beta} &= 1 \mid 32^* \mid 2\tau \mid 32^*1\tau \mid 32^*3\psi \mid 4\psi \\
&= 1 \mid 2\tau \mid 3\big(2^* \mid 2^*1\tau \mid 2^*3\psi\big) \mid 4\psi \\
&= 1 \mid 2\tau \mid 3\beta \mid 4\psi \\
&= \gamma \\
\overline{\psi} &= 1\gamma \mid 2\alpha \\
&= 1\gamma \mid 22^*1\gamma \\
&= \alpha
\end{aligned}$$

Next, we show that the strings representing the positive integers in the magical skew system can be classified into a number of equivalence classes, that we call *states*. Every increment is equivalent to a transition whose current and next states are uniquely determined by the current string. Since this state space is closed under the transitions, and each contains a set of strings of digits drawn from the set $\{0, 1, 2, 3, 4\}$, the correctness of the increment operation follows.

Define the eleven states: 12α, 22β, 03β, 30γ, 11γ, 23ψ, 04ψ, 31α, 21τ, 02τ, and 12τ. We show that the following are the only possible transitions when performing increments. This implies that the numbers in our system must be represented by one of these states.

1. $\underline{12\alpha \xrightarrow{+} 22\beta}$
   $12\alpha = 122^*1\gamma = 122^*1\big(1 \mid 2\tau \mid 3\beta \mid 4\psi\big) \xrightarrow{+} 222^*1\tau \mid 222^*3\big(0\overline{\beta} \mid 1\overline{\psi}\big) = 222^*1\tau \mid 222^*3\big(0\gamma \mid 1\alpha\big) = 22\big(2^*1\tau \mid 2^*3\psi\big) \subseteq 22\beta$
2. $\underline{22\beta \xrightarrow{+} 03\beta}$
   Obvious.
3. $\underline{03\beta \xrightarrow{+} 30\gamma}$
   $03\beta \xrightarrow{+} 30\overline{\beta} = 30\gamma$
4. $\underline{30\gamma \xrightarrow{+} 11\gamma}$
   Obvious.
5. $\underline{11\gamma \xrightarrow{+} 21\tau \mid 23\psi}$
   $11\gamma = 11\big(1 \mid 2\tau \mid 3\beta \mid 4\psi\big) \xrightarrow{+} 21\tau \mid 23\big(0\overline{\beta} \mid 1\overline{\psi}\big) = 21\tau \mid 23\psi$
6. $\underline{23\psi \xrightarrow{+} 04\psi}$
   Obvious.
7. $\underline{04\psi \xrightarrow{+} 31\alpha}$
   $04\psi \xrightarrow{+} 31\overline{\psi} = 31\alpha$
8. $\underline{31\alpha \xrightarrow{+} 12\alpha}$
   Obvious.
9. $\underline{21\tau \xrightarrow{+} 02\tau}$
   Obvious.
10. $\underline{02\tau \xrightarrow{+} 12\tau}$
    Obvious.
11. $\underline{12\tau \xrightarrow{+} 22\beta}$
    $12\tau \xrightarrow{+} 22\tau \subseteq 22\beta$

Remark 9. The strings representing the integers from 1 to 4 in the magical skew system are $1, 2, 01, 11$. These strings are obviously valid, though not in the form of any of the defined states. However, the string 21, which represents the integer 5, is of the form 21τ. So, we may assume that the initial string is 21 and the initial state is 21τ. □

3.2 Decrements

Our objective is to implement decrements as the reverse of increments. Given a string representing a number, we can efficiently identify the current state.

Remark 10. The first two digits are enough to distinguish between all the states except the states 12τ and 12α. To distinguish 12τ from 12α, we need to examine the first items of the lists recording the positions of high and low digits. If the string contains a high digit, then we conclude that the current state is 12α. Otherwise, we check the second item of the list recording the positions of low digits, and compare that to the position of the last digit. If only one low digit exists or if the second low digit is the last digit, the string is of the form 122^* or 122^*1; that is 12τ. On the other hand, if the second low digit is not the last digit, the string is of the form 122^*11, 122^*122^*, or 122^*122^*1; that is 12α. □

Assume that d_j is low. We define an *unfix* as the reverse of a fix:

Algorithm *unfix*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, j)$

assert d_j is low **and** $j \neq \ell - 1$
increase d_j by 3
decrease d_{j+1} by 1
if $j \neq 0$
    decrease d_{j-1} by 2

The following pseudo-code summarizes the actions to decrement a number:

Algorithm *decrement*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle)$

assert the value of $\langle d_0, d_1, \ldots, d_{\ell-1}\rangle$ is larger than 5
case the current state is in
    $\{12\alpha, 03\beta, 11\gamma, 04\psi, 02\tau\}$: *unfix*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, 0)$
    $\{30\gamma, 31\alpha\}$: *unfix*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, 1)$
    $\{23\psi\}$: *unfix*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, 2)$
    $\{22\beta\}$: let d_j be the first digit where $d_j = 3$, **if** d_j exists
                unfix$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, j + 1)$
    $\{12\tau, 21\tau\}$: do nothing
decrease d_0 by 1

Remark 11. If the string is of the form 22β, with the first high digit $d_j = 3$, then d_{j+1} is a low digit (cf. rewriting rules (3) and (5)). By unfixing this low digit, we get the exact reverse of the increment process. □

We write $T \xrightarrow{-} S$ indicating that the string S results by applying a decrement to T, and we write $\mathcal{T} \xrightarrow{-} \mathcal{S}$ if for each $T_i \in \mathcal{T}$ there exists $S_j \in \mathcal{S}$ such that $T_i \xrightarrow{-} S_j$. Furthermore, $\underline{T}$ stands for a string that results from T by decreasing its first digit by one, and $\underline{\mathcal{T}}$ for $\{\underline{T_1}, \underline{T_2}, \ldots\}$. We then have

$$\underline{\gamma} = \beta$$

$$\underline{\alpha} = \psi$$

We show that the following are the only possible transitions when performing decrements. This implies that the numbers in our system must be represented by one of our states.

1. $\underline{22\beta \xrightarrow{-} 12\tau \mid 12\alpha}$
   $22\beta = 22\big(2^* \mid 2^*1\tau \mid 2^*3\psi\big) \xrightarrow{-} 122^* \mid 122^*1\tau \mid 122^*1\big(3\underline{\gamma} \mid 4\underline{\alpha}\big)$
   $= 122^* \mid 122^*1\big(\varepsilon \mid 1 \mid 2\tau\big) \mid 122^*1\big(3\beta \mid 4\psi\big)$
   $= 12\big(2^* \mid 2^*1\big) \mid 122^*1\big(1 \mid 2\tau \mid 3\beta \mid 4\psi\big)$
   $= 12\tau \mid 122^*1\gamma = 12\tau \mid 12\alpha$

2. $\underline{12\alpha \xrightarrow{-} 31\alpha}$
   Obvious.
3. $\underline{31\alpha \xrightarrow{-} 04\psi}$
   $31\alpha \xrightarrow{-} 04\underline{\alpha} = 04\psi$
4. $\underline{04\psi \xrightarrow{-} 23\psi}$
   Obvious.
5. $\underline{23\psi \xrightarrow{-} 11\gamma}$
   $23\psi = 23\big(0\gamma \mid 1\alpha\big) \xrightarrow{-} 11\big(3\underline{\gamma} \mid 4\underline{\alpha}\big) = 11\big(3\beta \mid 4\psi\big) \subseteq 11\gamma$
6. $\underline{11\gamma \xrightarrow{-} 30\gamma}$
   Obvious.
7. $\underline{30\gamma \xrightarrow{-} 03\beta}$
   $30\gamma \xrightarrow{-} 03\underline{\gamma} = 03\beta$
8. $\underline{03\beta \xrightarrow{-} 22\beta}$
   Obvious.
9. $\underline{12\tau \xrightarrow{-} 02\tau}$
   Obvious.
10. $\underline{02\tau \xrightarrow{-} 21\tau}$
    Obvious.
11. $\underline{21\tau \setminus \{21\} \xrightarrow{-} 11\gamma}$
    $21\tau \setminus \{21\} = 21\big(\varepsilon \mid 1 \mid 2\tau\big) \setminus \{21\} = 21\big(1 \mid 2\tau\big) \xrightarrow{-} 11\big(1 \mid 2\tau\big) \subseteq 11\gamma$

Remark 12. For the above state transitions, the proof does not consider a decrement on the five strings from 21 down to 1. □

3.3 Properties

The following lemma directly follows from the state definitions.

Lemma 1. *Define a* block *to be a maximal substring where none of its digits is high, except its last digit. Define the* tail *to be the substring of digits following all the blocks in the representation of a number.*

- *The body of a block ending with* 4 *is either* 0 *or of the form* 12^*1.
- *The body of a block ending with* 3 *is either* 0 *or of the form* 12^*1 *or* 2^*.
- *Each* 4, 23 *and* 33 *is followed by either* 0 *or* 1.
- *There can be at most one* 0 *in the tail, which must then be its first digit.*

The next lemma provides a bound on the number of digits in any string.

Lemma 2. *For any positive integer* $n \neq 3$*, the number of digits in the string representing* n *in the magical skew system is at most* $\lg n$.

Proof. Inspecting all the state definitions and strings for small integers, the sum of the digits for any of our strings is at least 2, except for the strings 1 and 01. We also note that $d_{\ell-1} \neq 0$. It follows that, for all other strings, either $d_{\ell-1} > 1$ or $d_{\ell-1} = 1$ and $d_j \neq 0$ for some $j \neq \ell - 1$. Accordingly, we have $n \geq 2^\ell$ implying that $\ell \leq \lg n$. □

The following lemma bounds the average of the digits in any of our strings to be at most 2.

Lemma 3. *If $\langle d_0, d_1, \ldots, d_{\ell-1}\rangle$ is a representation of a positive integer in the magical skew system, then $\sum_{i=0}^{\ell-1} d_i \leq 2\ell$. If ℓ' denotes the number of the digits constituting the blocks of the representation, then $2\ell' - 1 \leq \sum_{i=0}^{\ell'-1} d_i \leq 2\ell'$.*

Proof. We prove the second part of the lemma, which implies the first part using the fact that any digit in the tail is at most 2. First, we show by induction on the length of the strings that the sum of the digits of a substring of the form $\alpha, \beta, \gamma, \psi$ is respectively $\sum_\alpha = 2\ell_\alpha, \sum_\beta = 2\ell_\beta, \sum_\gamma = 2\ell_\gamma + 1, \sum_\psi = 2\ell_\psi - 1$, where $\ell_\alpha, \ell_\beta, \ell_\gamma, \ell_\psi$ are the lengths of the corresponding substrings when ignoring the trailing digits that are not in a block. The base case is for the substring solely consisting of the digit 3, which is a type-γ substring with $\ell_\gamma = 1$ and $\sum_\gamma = 3$. From rewriting rule (2), $\sum_\alpha = 2(\ell_\alpha - \ell_\gamma - 1) + 1 + \sum_\gamma = 2(\ell_\alpha - \ell_\gamma - 1) + 1 + 2\ell_\gamma + 1 = 2\ell_\alpha$. From rewriting rule (3), $\sum_\beta = 2(\ell_\beta - \ell_\psi - 1) + 3 + \sum_\psi = 2(\ell_\beta - \ell_\psi - 1) + 3 + 2\ell_\psi - 1 = 2\ell_\beta$. From rewriting rule (4), $\sum_\gamma = 3 + \sum_\beta = 3 + 2\ell_\beta = 3 + 2(\ell_\gamma - 1) = 2\ell_\gamma + 1$. Alternatively, $\sum_\gamma = 4 + \sum_\psi = 4 + 2\ell_\psi - 1 = 4 + 2(\ell_\gamma - 1) - 1 = 2\ell_\gamma + 1$. From rewriting rule (5), $\sum_\psi = \sum_\gamma = 2\ell_\gamma + 1 = 2(\ell_\psi - 1) + 1 = 2\ell_\psi - 1$. Alternatively, $\sum_\psi = 1 + \sum_\alpha = 1 + 2\ell_\alpha = 1 + 2(\ell_\psi - 1) = 2\ell_\psi - 1$. The induction step is accordingly complete, and the above bounds follow.

Consider the substring that constitutes the blocks of the representation. Let ℓ' be the length of that substring. Since any sequence of blocks can be represented in one of the forms: 12α, 22β, 03β, 30γ, 11γ, 23ψ, 04ψ, 31α (excluding the tail). It follows that $\ell_\alpha, \ell_\beta, \ell_\gamma, \ell_\psi = \ell' - 2$. A case analysis implies that $\sum_{i=0}^{\ell'-1} d_i$ either equals $2\ell' - 1$ or $2\ell'$ for all cases. □

As a consequence to the previous two lemmas, we get the following corollary.

Corollary 1. *The sum of the digits in the string representing a positive integer n in the magical skew system is at most $2 \lg n$.*

The efficiency of the operations and the representation can be summarized as follows.

Theorem 2. *The magical skew system supports increments and decrements at $O(1)$ worst-case cost, and each operation involves at most four digit changes. The amount of space needed for representing a positive integer n while supporting these modifications is $O(\lg n)$.*

4 Application: A Worst-Case-Efficient Priority Queue

A binary heap [6] is a *heap-ordered* binary tree where the element associated with a node is not greater than that associated with its children. A *perfect binary heap* of height h is a complete binary tree storing $2^h - 1$ elements for an integer $h \geq 1$. Our heaps are pointer-based; each node has pointers to its parent and children.

As in a binomial queue, which is an ordered collection of heap-ordered binomial trees, in our binary-heap version, we maintain an ordered collection of perfect binary heaps. A similar approach has been used in several earlier publications [10–12]. The key difference between our approach and the earlier approaches is the numeral system in use; here we rely on our magical skew system. Assuming that the number of elements being stored is n and that $\langle d_0, d_1, \ldots, d_{\ell-1} \rangle$ is the representation of n in the numeral system, we maintain the invariant that the number of perfect binary heaps of size $2^{i+1} - 1$ is d_i.

To keep track of the perfect binary heaps, we maintain an auxiliary structure resembling the magical skew system. As before, we maintain a doubly-linked list of digits and two singly-linked lists recording the positions of the high digits and the low digits, respectively. In addition to these lists, we associate each digit with a list of items where each item stores a pointer to the root of a perfect binary heap of that particular size. To facilitate fast *find-min*, we maintain a pointer to a root that is associated with a minimum element.

The basic toolbox for manipulating binary heaps is given in most textbooks on algorithms and data structures (see, for example, [15, Chapter 6]). We need the function *siftdown* to reestablish the heap order when the element associated with a node is made larger, and the function *siftup* to reestablish the heap order when the element associated with a node is made smaller. Both operations are known to have logarithmic cost in the worst case; *siftdown* performs at most $2 \lg n$ and *siftup* at most $\lg n$ element comparisons. Note that in *siftdown* and *siftup* we move whole nodes not elements. In effect, the handles to nodes will always remain valid, and *delete* operations can be executed without problems.

A fix is emulated by involving three perfect binary heaps of the same height h, determining which root is associated with the smallest element, making this node the new root of a perfect binary heap of height $h+1$, and making the roots of the other two perfect binary heaps the children of this new root. The old subtrees of the detached root become perfect binary heaps of height $h-1$. That is, starting with three heaps of height h, one heap of height $h+1$ and two heaps of height $h-1$ are created; this corresponds to the digit changes resulting from a fix in the numeral system. After performing the fix on the heaps, the respective changes have to be made in the auxiliary structure (lists of roots, digits, and positions of high and low digits). The emulation of an unfix is a reverse of these actions. Compared to a fix, the only new ingredient is that, when the root of a perfect binary heap of height $h + 1$ is made the root of the two perfect binary heaps of height $h - 1$, *siftdown* is necessary. Otherwise, it cannot be guaranteed that the heap order is valid for the composed tree. Hence, a fix can be emulated at $O(1)$ worst-case cost, whereas an unfix has $O(\lg n)$ worst-case cost involving at most $2 \lg n$ element comparisons.

In *insert*, a node that is a perfect binary heap of size one is first added to the collection. Thereafter, the other actions specified for an increment in the numeral system are emulated. In effect, if a high or low digit is created, the corresponding list is updated. The location of the desired fix can be easily determined by accessing the first item in the list recording the positions of high

digits. If the element in the inserted node is smaller than the current minimum, the minimum pointer is updated to point to the new node. The worst-case cost of *insert* is $O(1)$ and it involves at most three element comparisons (one to compare the new element with the minimum and two when performing a fix).

When deleting a node, it is important that we avoid any disturbance to the numeral system and do not change the sizes of the heaps. Hence, we implement *delete* by borrowing a node and using it to replace the deleted node in the associated perfect binary heap. This approach guarantees that the numeral system only has to support decrements of the least-significant digit. First, *borrow* performs an unfix, and thereafter it removes a perfect binary heap of size one from the data structure. Due to the cost of the unfix, the worst-case cost of *borrow* is $O(\lg n)$ and it involves at most $2 \lg n$ element comparisons.

By the aid of *borrow*, it is straightforward to implement *delete*. Assuming that the borrowed node is different from the node to be deleted, the replacement is done, and *siftdown* or *siftup* is executed depending on the element associated with the replacement node. Because of this process, the root of the underlying perfect binary heap may change. If this happens, we have to scan through the roots of the heaps and update the pointer, in the item referring to this root, to point to the new root instead. A deletion may also invalidate the minimum pointer. If this happens, we have to scan all roots to determine the current overall minimum and update the minimum pointer to point to this root. The worst-case cost of all these operations is $O(\lg n)$. In total, the number of element comparisons performed is at most $6 \lg n$; *borrow* requires at most $2 \lg n$, *siftdown* (as well as *siftup*) requires at most $2 \lg n$, and the scan over all roots requires at most $2 \lg n$ element comparisons.

Remark 13. For perfect binary heaps, as for binomial trees [16], the parent-child relationships can be represented using two pointers per node instead of three. In accordance, the amount of extra space can be reduced from $3n + O(\lg n)$ words to $2n + O(\lg n)$ words. □

The above discussion can be summarized as follows.

Theorem 3. *A priority queue that maintains an ordered collection of perfect binary heaps guarantees $O(1)$ worst-case cost per insert and $O(\lg n)$ worst-case cost per delete, where n is the number of elements stored. This data structure requires $2n + O(\lg n)$ words of memory, in addition to the elements themselves.*

Remark 14. Our focus has been on good worst-case performance. Alternatively, more efficient amortized performance is achievable through lazy execution [17]. The idea is to maintain the roots of the perfect binary heaps in a doubly-linked list, keep a pointer to the root associated with the current minimum, and let *delete* do most of the work. In *insert*, a new node is added to the list of roots and the minimum pointer is updated if necessary. In *meld*, the two root lists are concatenated and the minimum pointer is set to point to the smaller of the two minima. In *borrow*, a root not pointed to by the minimum pointer is borrowed and its children (if any) are added to the root list. If there is only one tree, its

root is borrowed, its two children are moved to the root list, and the minimum pointer is updated to point to the child whose associated element is smaller.

Clearly, the worst-case cost of *insert*, *borrow*, and *meld* is $O(1)$. In *delete*, a *consolidation* is done: the perfect binary heaps of the same size are gathered together using a temporary array indexed by height and thereafter a fix—as described in Section 3—is performed as many times as possible. It is not difficult to show that the cost of *delete* is $\Theta(n)$ in the worst case.

Nevertheless, the amortized costs are: $O(1)$ per *insert* and *meld*, and $O(\lg n)$ per *borrow* and *delete*, where n is the size of the considered priority queue. To establish these bounds, we use a potential function that is the sum of the heights of the heaps currently in the priority queue. The key observation is that a fix decreases the potential by 1, *insert* increases it by 1, *borrow* increases it by $O(\lg n)$, *meld* does not change the total potential, and *delete* involves $O(\lg n)$ work that is not compensated by a potential reduction. □

5 Regular Skew System

In this section, we reconsider the regular system discussed in [3]. The advantage of the regular system is that it efficiently supports increments, decrements, and additions. We modify this system to handle expensive digit changes. When this modified regular system is used in our application, the performance of the priority queue developed matches, or even outperforms, that of the related priority queues presented in [8, 10].

Recall that in the regular system a positive integer n is represented as a string $\langle d_0, d_1, \ldots, d_{\ell-1} \rangle$ of digits, where

- $d_i \in \{0, 1, 2\}$ for all $i \in \{0, 1, \ldots, \ell - 1\}$, and $d_{\ell-1} \neq 0$, and
- if $d_i = 2$ and $d_k = 2$ and $i < k$, then $d_j = 0$ for some $j \in \{i + 1, \ldots, k - 1\}$ *(regularity condition)*.

Traditionally, the regular system employs perfect weights, i.e. the weight of digit d_i is 2^i. We, however, adapt the system to skew weights. Therefore, as in the other skew binary systems, we have

- $w_i = 2^{i+1} - 1$ for all $i \in \{0, 1, \ldots, \ell - 1\}$, and
- the value of n is $\sum_{i=0}^{\ell-1} d_i w_i$.

Moreover, we associate a variable b_i with every digit d_i. This variable expresses the amount of work left until the change at that digit is completed.

In a computer realization, we rely on a sparse representation that keeps all non-zero digits together with their positions and variables in a doubly-linked list in sorted order according to the positions. In addition, we maintain a doubly-linked list for the positions of the positive b_i's and another for the positions of the 2's.

5.1 Increments

To perform an increment, it would be natural to use the surplus unit supplied by the increment for transferring the first $d_j = 2$ (if it exists) to d_{j+1}, by setting $d_j \leftarrow 0$ and increasing d_{j+1} by one. Such a *transfer* is exactly what is done in the canonical skew system. One complication that arises in our application is that the transfer of d_j may have $O(j)$ worst-case cost. To handle expensive digit changes, the key idea is to delay the work to be done in an expensive digit change and leave the delayed work for the upcoming operations (increments, decrements, and additions). Hence, we perform digit changes incrementally and keep track of the digits that are under an incremental change. Because of these incremental changes, we have to allow more than one 2 in the representation.

We divide the work to be done in an expensive digit change into discrete units, called *bricks*. When a digit d_k is under an incremental change, i.e. when $b_k > 0$, d_k is said to form a *wall* of b_k bricks. Decreasing b_k by one means that we perform a constant amount of work in order to bring the incremental change at d_k forward, which can be viewed as removing a brick from that wall.

The increment algorithm works as follows. Let d_j be the first 2 for which $b_j = 0$. If j is smaller than the position of the first wall, we transfer d_j to d_{j+1}; a value of $j + 1$ bricks is associated with b_{j+1}, and one brick is immediately removed from d_{j+1} leaving j bricks to be removed by the upcoming operations. Otherwise, d_0 is increased by one, and one brick is removed from the first wall if any. For this case, since $d_0 \in \{0, 1\}$ before the increment, $d_0 \in \{1, 2\}$ after the increment. For both cases, the value of the number is increased by one.

Algorithm *transfer*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, j)$

1: **assert** $d_j = 2$
2: $d_j \leftarrow 0$
3: increase d_{j+1} by 1
4: $b_{j+1} \leftarrow j + 1$

Remark 15. Since $w_{i+1} = 2w_i + 1$, a transfer increases the value of the represented number by one. □

Algorithm *increment*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle)$

1: let d_k be the first digit for which $b_k > 0$, if it exists
2: let d_j be the first 2 for which $b_j = 0$, if it exists
3: **if** d_j exists **and** (d_k does not exist **or** $j < k$)
4: *transfer*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, j)$
5: reduce b_{j+1} by 1
6: **else**
7: increase d_0 by 1
8: **if** d_k exists
9: reduce b_k by 1

position i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
critical 0					d_1		d_3		d_5			d_7				
digit d_i	2	0	1	0	$\boxed{1}$	0	$\boxed{1}$	0	$\boxed{1}$	1	0	$\boxed{2}$	1	1	0	1
variable b_i					1		4		6			8				

Fig. 2. The representation obtained by performing 100 000 increments in the regular skew system starting from 0. All the walls are drawn in boxes.

Remark 16. Starting from 0 and performing 30 increments, the strings generated are: 1, 2, 01, 11, 21, 02, 00$\boxed{1}$, 101, 201, 011, 111, 211, 021, 00$\boxed{2}$, 102, 100$\boxed{1}$, 200$\boxed{1}$, 010$\boxed{1}$, 1101, 2101, 0201, 00$\boxed{1}$1, 1011, 2011, 0111, 1111, 2111, 0211, 00$\boxed{2}$1, and 1021. All walls are drawn in boxes. □

Remark 17. In the algorithm described in [8], several units of leftover work are to be performed per increment. Interestingly, in our case, it is enough to perform only one unit of leftover work per operation. This stems from the fact that we use skew weights instead of perfect weights. □

To prove the correctness of our algorithms, we have to rely on stronger regularity conditions than those of the standard regular system. Our first invariant states that every 2 is immediately preceded by a 0, while it is possible to have $d_0 = 2$. The second invariant indicates that between every two 2's there are at least two 0's. The third invariant states that every wall is immediately preceded by a 0. The fourth invariant indicates that if a 2 is followed by a wall, there are at least two 0's between them. The fifth invariant indicates that every wall has at least two preceding 0's. Assume that d_k is a wall. Let d_j be a 0, such that $j < k-1$ and $d_{j'} \neq 0$ for all $j' \in \{j+1, \ldots, k-2\}$. In other words, d_j is the 0 with the highest position that precedes the 0 at d_{k-1}. By the fifth invariant, such a 0 always exists. We call this 0 the *critical* 0 for the wall d_k. The sixth invariant states that the number of leftover bricks at a wall is not larger than $j+1$, where j is the position of the critical 0 for that wall, and even not larger than j if this critical 0 is immediately preceded by a substring of the form 21^*.

More formally, the invariants maintained are:

(1) If $d_j = 2$ and $j > 0$, then $d_{j-1} = 0$.
(2) If $d_j = 2$, $d_k = 2$, and $j < k$, then $d_{j'} = 0$ for some $j' \in \{j+1, \ldots, k-2\}$.
(3) If d_k is a wall, then $d_{k-1} = 0$.
(4) If $d_j = 2$, d_k is a wall, and $j < k$, then $d_{j'} = 0$ for some $j' \in \{j+1, \ldots, k-2\}$.
(5) If d_k is a wall, then $d_i = 0$ for some $i \in \{0, \ldots, k-2\}$.
(6) Let d_j be the critical 0 for a wall d_k. Then
    (i) $b_k \leq j+1$; moreover
    (ii) $b_k \leq j$, if $d_i = 2$ and $d_{i'} = 1$ for all $i' \in \{i+1, \ldots, j-1\}$.

Remark 18. Our intuition about the wall-breaking process is the following (for a snapshot of this dynamic process, see Fig. 2):

- The first wall is *visible* for an increment if all the digits preceding that wall are 0's and 1's. When a wall is visible, this particular increment will reduce the number of bricks at that wall by 1.
- The configuration of the digits preceding a wall affects the number of bricks remaining at that wall. In particular, the position of the critical 0, and whether the critical 0 is immediately preceded by a substring of the form 21^* or not, is used to bound the height of the wall. In accordance, this prevents that two walls become adjacent.
- During the counting process, there will be enough configurations for every wall being visible for increments. Specifically, when—or even before—the critical 0 for the first wall disappears (all the digits preceding the first wall, except its immediately preceding 0, are 1's or 2's), there are no bricks left at this wall, i.e. the wall has been dismantled. □

The aforementioned stronger regularity conditions imply that in any configuration a digit is either $0, 1$, or 2, and that no walls become adjacent. In addition, our algorithms will never transfer a digit that is a 2 if it also forms a wall, i.e. no walls are involved in creating other walls. Next, we show that the invariants are retained after every increment if they were fulfilled before the operation; the correctness of *increment* accordingly follows.

Consider the case where a digit d_j is the first digit that equals 2, is not a wall, precedes any wall, and $j \neq 0$. In such a case, a transfer is initiated at d_j. Using invariant (1), $d_{j-1} = 0$. After the operation, $d_j = 0$ and d_{j+1} becomes a wall with $b_{j+1} = j$. At this point, d_{j-1} is the critical 0 for the wall d_{j+1}; this fulfils all the invariants for this wall. A special case is when $j = 0$. For such a case, the created wall d_1 is immediately dismantled. Consider a digit $d_k = 2$ that follows d_j. Using invariants (1) and (2), before the increment, there are two 0's, one at d_{k-1} and the other has a position in $\{j + 1, \ldots, k - 2\}$. Whether d_{j+1} was 0 or 1 before the operation, it is straightforward to verify that all the invariants are fulfilled for d_k after the operation. The interesting case is when a digit $d_k \in \{1, 2\}$ is a wall and d_{j+1} was the critical 0 for this wall before the operation. For such a case, using invariant (6.ii) as $d_j = 2$, we have $b_k \leq j + 1$. After the operation, the critical 0 for the wall d_k becomes d_j. However, the validity of the invariants still hold as the weaker bound of invariant (6.i) applies and is fulfilled. It is straightforward to verify that all the other invariants are also retained.

Consider the case where d_k is the first wall and that it precedes any 2. In such a case, one brick is removed from the wall d_k and d_0 is increased by 1. Using invariants (3) and (5), $d_{k-1} = 0$ and $d_j = 0$ for some $j \in \{0, \ldots, k - 2\}$. An interesting case is when, before the increment, d_i was 1 for all $i \in \{0, \ldots, j - 1\}$, and d_j was the critical 0 for the wall d_k. For such a case, using invariant (6.i), $b_k \leq j + 1$. After the operation, as d_0 becomes 2, the stronger bound of invariant (6.ii) applies and is fulfilled following the brick removal from d_k, i.e. $b_k \leq j$. Another interesting case is when, before the increment, d_0 was the critical 0 for the wall d_k. For such a case, using invariant (6.i), $b_k \leq 1$. After the operation,

d_0 becomes 1, and as a result only one 0 precedes d_k. Fortunately, one brick must have been removed from the wall d_k, dismantling the wall and saving our invariants just on time. It is straightforward to verify that, even in the trivial cases omitted, the invariants are also retained.

5.2 Decrements

In the regular skew system, we carry out decrements in the same way as in the canonical skew system. To ensure that the amount of work left at the walls satisfies our invariants, if the first non-zero digit was a wall, the wall is moved to the preceding position and a brick is removed from it. The details are specified in the following pseudo-code:

Algorithm *decrement*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle)$

```
1: assert ⟨d_0, d_1, ..., d_{ℓ-1}⟩ is not empty
2: let d_j be the first non-zero digit
3: if b_j > 0
4:    reduce b_j by 1
5: decrease d_j by 1
6: if j ≠ 0
7:    d_{j-1} ← 2;   b_{j-1} ← b_j;   b_j ← 0
```

Let d_j be the first non-zero digit before a decrement. First, consider the case where $j = 0$, i.e. d_0 was 1 or 2 before the operation. After the operation, d_0 is decreased by one. It is straightforward to verify that all the invariants are retained after the operation. Next, assume that $j \neq 0$. Consider the case where d_j was a wall before the operation. By invariant (6), $b_j \leq j - 1$ was valid before the operation. Due to the brick removal, $b_{j-1} \leq j - 2$ after the operation, and invariant (6) is retained. Consider the case where d_j was equal to 1 before the operation, and d_k was the following wall whose critical 0 was d_{j-1}. (The case where the critical 0 for d_k comes after d_j is trivial.) Using invariant (6.i), $b_k \leq j$ before the operation. After the operation, d_j becomes the critical 0 for the wall d_k, and d_{j-1} becomes 2. In accordance, invariant (6.ii) applies for b_k and is fulfilled. Alternatively, if d_j was equal to 2 before the operation, the critical 0 for the following wall d_k must come after d_j, and invariant (6) trivially holds after the operation. It is also clear that, whether d_j was a wall or not, and whether it was equal to 1 or 2, all the other invariants are retained.

5.3 Increments at Arbitrary Positions

Given a string $\langle d_0, d_1, \ldots, d_{\ell-1}\rangle$, our objective is to efficiently increase an arbitrary digit d_i by one. Compared to a normal increment, this general increment involves more new cases which make the algorithm a bit more complicated. Also,

the worst-case complexity of the operation becomes higher, namely $O(i)$, even though we still solve the special case $i = 0$ at $O(1)$ worst-case cost.

A crucial difference between *increment* and *arbitrary-increment* is that, if $i > 0$, we have no surplus unit to transfer a digit forward. Accordingly, we always have to add one to digit d_i. To preserve regularity, we may need to perform up to two digit transfers. To create the surplus needed for digit transfers, we perform two decrement operations, and use the resulting surplus for such transfers. Since we do not know in advance the number of the transfers that we perform, at the end, we perform one *increment* for each surplus we have saved but not used. The following corrective actions are necessary to reestablish our invariants.

If d_i is a wall, we dismantle this wall by removing all (at most $i - 1$) of its bricks. If $d_i = 2$, we transfer d_i one position forward. Hereafter, we can assume that $d_i \in \{0, 1\}$. At this point, let d_j be the first 2 for which $j > i$ and $b_j = 0$. It is time now that we increase d_i by one. If after increasing d_i its value becomes 2, we transfer the created 2 one position forward. Note that at most one of these aforementioned two transfers is executed. If d_{i+1} is a wall, we dismantle this wall by removing all (at most i) of its bricks. This wall is either an original wall even before the operation or has just been created by transferring a 2 from d_i forward. At this point, let d_k be the first wall for which $k > i$. Afterwards, we perform the following steps that are quite similar to the increment algorithm of Section 5.1. If j is smaller than k, we transfer the 2 at d_j one position forward, and remove the minimum of $i + 2$ and all the bricks from d_{j+1}. If k is smaller than j, we remove the minimum of $i + 1$ and all the bricks from d_k.

To sum up, at most a constant number of digits are changed in each increment. To find the walls and 2's operated on, it may be necessary to scan the linked lists, starting from the beginning, to the desired positions. To get there, at most $O(i)$ list items have to be visited. Also, the number of bricks removed at each step is $O(i)$. We conclude that the complexity of the algorithm is $O(i)$ in the worst case. This completes the description of the algorithm. All actions are summarized in the enclosed pseudo-code.

To prove the correctness of the algorithm, we shall show that all the invariants hold for every wall and for every 2 when considering the preceding digits for each. For the purpose of our analyses, we split the pseudo-code into *phase I* constituting lines 5 to 16, and *phase II* constituting lines 17 to 23. Consider the effect of executing phase I: only d_i and d_{i+1} may change, d_i will neither be a 2 nor a wall, and d_{i+1} will not be a wall. There are three cases which result in d_{i+1} being a 2 after phase I.

- d_i was 1 and d_{i+1} was 1: After phase I, d_i becomes a 0.
- d_i was 2 and d_{i+1} was 1: After phase I, d_i becomes a 1. In such case, $j = i+1$.
- d_i was 0 and d_{i+1} was 2: After phase I, d_i becomes a 1. In such case, $j = i+1$.

For the first case, the invariants are directly fulfilled for d_{i+1} after phase I. For the last two cases, we have to wait until phase II, when the 2 at d_{i+1} is transferred to d_{i+2}, for the invariants to hold.

In general, if $j > i + 1$, phase II of the algorithm is pretty similar to the increment algorithm of Section 5.1. The only differences are the positions of the

Algorithm *arbitrary-increment*$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, i)$

assert the value of $\langle d_0, d_1, \ldots, d_{\ell-1}\rangle$ is larger than 2
repeat 2 times
    decrement$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle)$
surplus $\leftarrow 2$
if $b_i > 0$
    reduce b_i to 0
if $d_i = 2$
    transfer$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, i)$
    decrease *surplus* by 1
let d_j be the first 2 for which $j > i$ and $b_j = 0$, if it exists
increase d_i by 1
if $d_i = 2$
    transfer$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, i)$
    decrease *surplus* by 1
if $b_{i+1} > 0$
    reduce b_{i+1} to 0
let d_k be the first digit for which $k > i$ and $b_k > 0$, if it exists
if d_j exists **and** (d_k does not exist **or** $j < k$)
    transfer$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle, j)$
    decrease *surplus* by 1
    reduce b_{j+1} by $\min\{b_{j+1}, i+2\}$
else if d_k exists
    reduce b_k by $\min\{b_k, i+1\}$
repeat *surplus* times
    increment$(\langle d_0, d_1, \ldots, d_{\ell-1}\rangle)$

critical 0's, and the number of bricks we remove from the walls. The same proof of Section 5.1 implies the validity of the invariants for all the 2's. What is left to be shown is that invariant (6) is also satisfied for all the walls.

Consider the first wall d_k for which $i < k < j$. If the critical 0 for d_k was following d_{i+1}, then it does not change after phase I, and hence all the invariants hold for d_k. If the critical 0 for d_k was preceding d_{i+1}, then $b_k \leq i+1$. This means that by removing $i+1$ bricks this wall is dismantled. If the critical 0 for d_k was at d_{i+1} and d_i was a 2, then $b_k \leq i+1$, and the wall is also dismantled. If the critical 0 for d_k was at d_{i+1} and d_i was not a 2, then the critical 0 for d_k becomes at d_i or is still at d_{i+1}. Either way, invariant (6) still holds.

Back to the case where $j = i+1$. When the 2 at d_{i+1} is transferred in phase II, it results in a wall at d_{i+2}. However, we instantaneously dismantle this wall by removing its $i+2$ bricks. Alternatively, if $i+1 < j < k$, then d_{j-1} or d_{j-2} is the critical 0 for the resulting wall at d_{j+1}. In this case, b_{j+1} is initiated with $j+1$, which is followed by removing $i+2$ bricks (we only need to remove one or two bricks in this case). Accordingly, invariant (6) still holds.

5.4 Additions

When an increment at any position can be carried out efficiently, a simple way of implementing *addition* is to apply *arbitrary-increment* repeatedly. For every digit d_i of the shorter string, starting from the least-significant digit, increment the ith digit of the longer string d_i times. The correctness of this approach directly follows from the correctness of *arbitrary-increment*.

Remark 19. An addition will destroy the longer of the two input strings. □

Algorithm *addition*($\langle d_0, d_1, \ldots, d_{k-1}\rangle$, $\langle e_0, e_1, \ldots, e_{\ell-1}\rangle$)

```
1: assert k ≤ ℓ
2: for i ∈ {0, 1, ..., k − 1}
3:    repeat d_i times
4:       if b_i > 0
5:          reduce b_i to 0
6:       arbitrary-increment(⟨e_0, e_1, ..., e_{ℓ−1}⟩, i)
7: return ⟨e_0, e_1, ..., e_{ℓ−1}⟩
```

Assume that the representations of the given numbers are of length k and ℓ, and that $k \leq \ell$. By Lemma 4 (see Section 5.5), the sum of the digits in the shorter representation is at most $k + 1$. So, an addition of the two numbers involves at most $k+1$ arbitrary increments. Each such increment only changes a constant number of digits. Hence, the total number of digit changes performed is proportional to the length of the shorter representation. At position i, both the wall dismantling and the arbitrary increment have $O(i)$ worst-case cost. Thus, the worst-case cost of *addition* is $O(k^2)$.

5.5 Properties

The following lemma directly follows from the invariants maintained by the operations.

Lemma 4. *Let $\langle d_0, d_1, \ldots, d_{\ell-1}\rangle$ be a string of digits in the regular skew system. The sum of the digits is at most $\ell + 1$, i.e. $\sum_{i=0}^{\ell-1} d_i \leq \ell + 1$.*

We can summarize the results proved for the regular skew system as follows.

Theorem 4. *The regular skew system supports increments and decrements at $O(1)$ worst-case cost, and additions at $O(k^2)$ worst-case cost, where k denotes the length of the shorter of the two strings added. The number of digit changes involved is at most two for increments and decrements, and $O(k)$ for additions. The amount of space needed for representing a positive integer n while supporting these modifications is $O(\lg n)$.*

6 Application Revisited

As earlier, we implement a priority queue as a forest of perfect binary heaps, and use a numeral system to control the number and size of the heaps in the structure. When the canonical skew system is in use, *insert* has logarithmic worst-case cost, whereas *borrow* has $O(1)$ worst-case cost. For the magical skew system, the opposite is true: *insert* has $O(1)$ worst-case cost and *borrow* has logarithmic worst-case cost. In both cases, the reason for the logarithmic behaviour is that, even though only a constant number of digit changes is made per operation, it may be necessary to perform a *siftdown* operation for a large heap, and such operation can be costly. In order to avoid this problem, we rely on the regular skew system which allows incremental execution of *siftdown* operations.

In our implementation the basic components are: perfect binary heaps, an auxiliary structure resembling the regular skew system keeping track of the heaps, and a minimum pointer identifying the location of the current minimum. The auxiliary structure comprises the data recording the digits, the positions of the walls and 2's, and the heights of the walls. The digit at position i denotes the number of perfect binary heaps of size $2^{i+1} - 1$. Each digit should be associated with a list of items referring to heaps of that particular size; every such list has at most two items, and each item stores a pointer to the root of a heap. In a brick removal, a *siftdown* process is advanced one level down a heap, or nothing is done if the heap order has already been reestablished. To facilitate incremental execution of *siftdown* operations, each of the items associated with a digit also stores a pointer to the node up to which the corresponding *siftdown* has proceeded; this pointer may be null indicating that there is no ongoing *siftdown* process. When this pointer is available, the process can be easily advanced, level by level, until the bottom level is reached or the heap order is reestablished. In addition, the path traversed by every ongoing *siftdown* is memorized by storing the path traversed so far in binary form; we call this a *bit trace*. A 0-bit (1-bit) at the index i of a bit trace reflects that the process continued in the direction of the left (right) child at level i.

Remark 20. A suitable realization of a bit trace would be a single computer word. Using logical bitwise operations and bit shifts, the bit at a specified index can be easily fetched and updated at $O(1)$ worst-case cost. This accounts for $O(\lg n)$ words to be stored, at most one per heap. □

In *insert*, the actions specified for an increment in the numeral system are emulated. There are two possibilities to consider. The first possibility is that the given node is used to combine two heaps, by making it the root of the combined heap and making the roots of the two heaps the children of that root. A *siftdown* process is initiated at the new root and advanced one level downwards. This maintains the invariant that the root of any heap contains a minimum element among those in that heap. The second possibility is that the given node is inserted into the structure as a heap of size one. For such a case, the *siftdown* process at the first wall (if any) is advanced one level downwards.

When a *siftdown* is advanced, the direction to which the process proceeds is recorded in the bit trace, by updating the bit corresponding to the current level. Finally, the minimum pointer is updated if necessary. The worst-case cost of *insert* is $O(1)$, and at most three element comparisons are performed (two in *siftdown* and one to check whether the minimum pointer is up to date or not).

In *borrow*, the actions specified for a decrement in the numeral system are emulated. The numeral system dictates that the smallest heap is selected for splitting. If the first non-zero digit is a 2 that forms a wall, of the two candidates, the heap that has an ongoing *siftdown* process is selected. First, the ongoing *siftdown* (if any) in the selected heap is advanced one level downwards. Normally, the selected heap is split by detaching and returning its root, and moving the heaps rooted at the children of the root (if any) to the preceding position. A special case is when the root of the selected heap is associated with the minimum of the priority queue. In such a case, this root is swapped either with another root or with one of its two children, which is then detached and returned. A removal of the root, or a change in one of its children, will not affect the *siftdown* process possibly being in progress further down in the heap. Lastly, if the selected heap had an ongoing *siftdown*, it should be possible to identify which of the two heaps owns that process after the split. Here we need the bit trace. The existing bit trace is updated to correspond to the root removal and is assigned to the new owner. Clearly, the worst-case cost of *borrow* is $O(1)$, and at most two element comparisons are performed.

In accordance, *delete* can use this kind of borrowing and reestablish the heap order by sifting the replacement node up or down. Naturally, the minimum pointer has to be updated if the current minimum is being deleted. Compared to our earlier solution, the only difference is that the affected heap may have an incremental *siftdown* process in progress. We let *delete* finish this process before handling the actual deletion. As before, the worst-case cost of *delete* is $O(\lg n)$. By Lemma 4, the number of perfect binary heaps is bounded by $\lg n + O(1)$. Therefore, in the worst case, *delete* involves at most $5 \lg n + O(1)$ element comparisons as a result of performing at most two *siftdown* operations and one possible scan over the roots, or one *siftup* and one possible *siftdown*.

Remark 21. It is possible to reduce the number of element comparisons performed by *delete* to at most $3 \lg n + O(1)$. The worst-case scenario occurs when a *siftdown* operation is to be performed, while another *siftdown* is in progress. In this case, the ongoing *siftdown* is first completed, but the second *siftdown* is performed only partially, leaving the wall with the same height as before the operation. Then, the number of element comparisons performed is at most $3 \lg n + O(1)$; the two *siftdown* operations combined account for $2 \lg n + O(1)$, and the scan over the roots accounts for $\lg n + O(1)$. □

In *meld*, the actions specified for an addition in the numeral system are emulated. A transfer of two perfect binary heaps forward, using a surplus node borrowed at the beginning of the addition, is done in the same manner as in *insert*. Similarly, *siftdown* processes can be advanced as in *insert*. The only difference

is that now several bricks are processed, instead of just one. The performance of *meld* directly follows from the performance of the addition operation. Given two priority queues of sizes m and n, $m \leq n$, the worst-case cost of *meld* is $O(\lg^2 m)$.

Remark 22. The space optimization mentioned in Remark 13 is also applicable here. Hence, the amount of extra space can be reduced to $2n+O(\lg n)$ words. □

The efficiency of the operations and the representation can be summarized as follows.

Theorem 5. *A meldable priority queue that maintains an ordered collection of perfect binary heaps guarantees $O(1)$ worst-case cost per insert and borrow, $O(\lg n)$ worst-case cost per delete, and $O(\lg^2 m)$ worst-case cost per meld, where n is the number of elements in the priority queue manipulated by delete and m is the number of elements in the smaller of the two priority queues manipulated by meld. A priority queue storing n elements requires $2n + O(\lg n)$ words, in addition to the space used by the elements themselves.*

7 Conclusions

All the numeral systems discussed in this paper—canonical skew, magical skew, and regular skew—support increments and decrements at $O(1)$ worst-case cost, and involve at most a constant number of digit changes per operation. In spite of this, the efficiency of the data structures using the numeral systems varied depending on the carry-propagation and borrowing mechanisms used. Hence, to understand the actual behaviour in our application, a deeper investigation into the properties of the numeral systems was necessary.

For the magical skew system, when proving the correctness of our algorithms, we used an automaton. The state space of the modelling automaton was small, and the states can be seen as forming an orbit. Increments were proved to move along the states around this orbit. The system turned out to be extremely sensitive. Therefore, we decided to implement decrements as the reverse of increments. In the conference version of this paper [1], we relied on a general undo-logging technique to implement decrements as the reverse of increments. This, however, has the drawback that the amount of space used is proportional to the value of the underlying number, which is exponential in the length of the representation. In this version of the paper, we showed that the reversal approach works even if the amount of space available is proportional to the length of the representation.

One drawback of the magical skew system is that it does not support additions efficiently. Simply, it is not known how to efficiently get from an arbitrary configuration back to one of the configurations on the orbit. The invariants enforced by the canonical skew and regular skew systems are more relaxed, so additions can be efficiently realized. The regular skew system supports additions even more efficiently than the canonical skew system ($O(k)$ versus $O(\ell)$ digit changes, where k and ℓ are the lengths of the added strings and $k \leq \ell$).

By implementing a priority queue as a forest of perfect binary heaps, we provided two data structures that perform *insert* at $O(1)$ worst-case cost and *delete* at $O(\lg n)$ worst-case cost. There are several other ways of achieving the same bounds; for example, one could use specialized data structures like a forest of pennants [8] or a binomial queue [9, 13, 14]. The priority queue obtained using the regular skew system supports *borrow* at $O(1)$ worst-case cost and *meld* at $O(\lg^2 m)$ worst-case cost, where m is the number of elements stored in the smaller of the two melded priority queues.

We conclude the paper with some open problems.

1. The array-based implementation of a binary heap supports *delete* with at most $2 \lg n$ element comparisons (or even less [7]). To perform *insert* at $O(1)$ worst-case cost, we increased the number of element comparisons performed by *delete* to $6 \lg n$ (Section 4) or $3 \lg n$ (Section 6). How to reduce the number of element comparisons performed by *delete* without making other operations asymptotically slower?
2. We showed that all the following operations: *insert*, *meld*, and *delete* could be performed at optimal asymptotic cost in the amortized sense. However, our best data structure only supports *meld* at $O(\lg^2 m)$ worst-case cost, where m is the number of elements stored in the smaller of the two melded priority queues. In [9], the technique of data-structural bootstrapping was used to speed up *meld* for (skew) binomial queues. In our case, the worst-case cost of *meld* is still too large to apply this technique successfully. Can the worst-case performance of *meld* be improved?
3. All known approaches to implement fast *decrease*, which decreases the value stored at a given node, are based on heap-ordered binomial-tree-like structures. Is it really so that binary heaps are not competitive in this respect?
4. So far, we have not attempted to provide any evidence for the practical relevance of the findings reported in this paper. How efficient are the described data structures in practice?
5. We hope that our approach for designing numeral systems and related data structures is relevant in other contexts. Are there other applications that could be tackled with the developed numeral systems?

Acknowledgment

We thank the anonymous referee who motivated us to extend this version of the paper beyond the frontiers of the conference version.

References

1. Elmasry, A., Jensen, C., Katajainen, J.: The magic of a number system. In: Proceedings of the 5th International Conference on Fun with Algorithms. Volume 6099 of Lecture Notes in Computer Science, Springer-Verlag (2010) 156–165
2. Vuillemin, J.: A data structure for manipulating priority queues. Communications of the ACM **21**(4) (1978) 309–315

3. Clancy, M., Knuth, D.: A programming and problem-solving seminar. Technical Report STAN-CS-77-606, Department of Computer Science, Stanford University (1977)
4. Myers, E.W.: An applicative random-access stack. Information Processing Letters **17**(5) (1983) 241–248
5. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1998)
6. Williams, J.W.J.: Algorithm 232: Heapsort. Communications of the ACM **7**(6) (1964) 347–348
7. Gonnet, G.H., Munro, J.I.: Heaps on heaps. SIAM Journal on Computing **15**(4) (1986) 964–971
8. Carlsson, S., Munro, J.I., Poblete, P.V.: An implicit binomial queue with constant insertion time. In: Proceedings of the 1st Scandinavian Workshop on Algorithm Theory. Volume 318 of Lecture Notes in Computer Science, Springer-Verlag (1988) 1–13
9. Brodal, G.S., Okasaki, C.: Optimal purely functional priority queues. Journal of Functional Programming **6**(6) (1996) 839–857
10. Sack, J.R., Strothotte, T.: A characterization of heaps and its applications. Information and Computation **86**(1) (1990) 69–86
11. Bansal, S., Sreekanth, S., Gupta, P.: M-heap: A modified heap data structure. International Journal of Foundations of Computer Science **14**(3) (2003) 491–502
12. Harvey, N.J.A., Zatloukal, K.: The post-order heap. In: Proceedings of the 3rd International Conference on Fun with Algorithms. (2004)
13. Elmasry, A., Jensen, C., Katajainen, J.: Multipartite priority queues. ACM Transactions on Algorithms **5**(1) (2008) 14:1–14:19
14. Elmasry, A., Jensen, C., Katajainen, J.: Two-tier relaxed heaps. Acta Informatica **45**(3) (2008) 193–210
15. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. 3nd edn. The MIT Press, Cambridge (2009)
16. Brown, M.R.: Implementation and analysis of binomial queue algorithms. SIAM Journal on Computing **7**(3) (1978) 298–319
17. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM **34**(3) (1987) 596–615

Multipartite Priority Queues

AMR ELMASRY
Max-Planck Institut für Informatik
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 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 article, 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.

The work of the authors was 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 was also supported by an Alexander von Humboldt Fellowship.
A. Elmasry, Max-Planck Institut für Informatik, Saarbrücken, Germany; C. Jensen and J. Katajainen, Department of Computing, University of Copenhagen, Universitetsparken 1, 2100 Copenhagen East, Denmark.

Examples of priority queues that achieve these bounds, in the amortized sense, are 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 aforementioned 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 article [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], we obtain a sorting algorithm 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, ..., $r-1$ 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

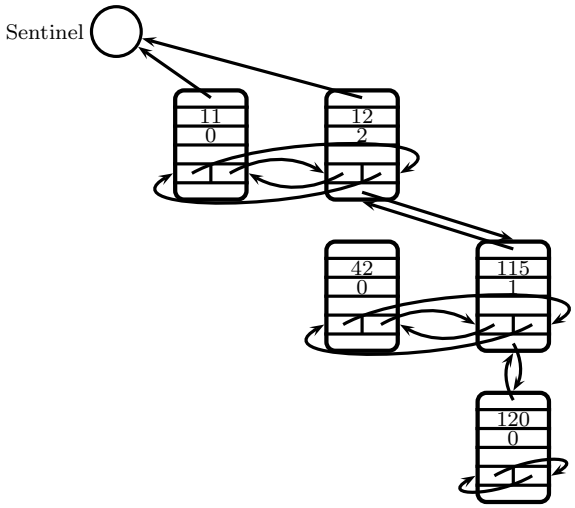


Fig. 1. A binomial queue storing integers $\{11, 12, 42, 115, 120\}$. Each node of a binomial tree has space for seven fields: parent pointer, element, rank, extra pointer (needed later on), older-sibling pointer, younger-sibling pointer, and youngest-child pointer. The nonstandard features are the component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

from youngest to oldest (or vice versa, if the oldest child is first accessed via the youngest child). It should be pointed out that with respect to the parent pointers our representation is nonstandard. An argument for 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 these 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*, that is 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 rank 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 largest rank. Every delete-min operation in such a sequence requires $\lfloor \log n \rfloor$ element comparisons for joining the trees of equal rank 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 neighbor. 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 onto 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]). In other words, 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 rank, 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 those 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 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, described next.

(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 as well retain the size of this 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 those 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, either 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, that is, the cost of an insert operation 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 analyzed 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 lower store during the execution of different operations, we call the data structure a *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 analyze 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 neighboring 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. Thus, 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 oldest to youngest. In other words, every subtree is joined with that tree resulting from the joins of the subtrees rooted at 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. □

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 that is a descendant 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 ith oldest child (or simply the ith subtree), $i \in \{1, \ldots, j\}$, 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 ith 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. □

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 $d_y \leq r$. □

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, then $r < \log n$. 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 to a node held in the reservoir. Therefore, a node in the upper store is marked when its counterpart becomes a nonroot 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 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 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 worst-case cost $O(\log m)$ 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 a root storing the 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 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 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 where $n = 1$ 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. Thus, 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. Then seen as a tree of rank 0, this node is 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 this 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 analyzed 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 worst-case cost $O(\log n)$, but involves no element comparisons. Therefore, the complexity of a delete operation is the same as that of a delete-min operation.

4.4 Summing up the results

Using the components described and the complexity bounds derived, the efficiency of 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. At each level except the highest, 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, lazy deletions are carried out as in our earlier upper store. Except for the highest level, the constant factor in the logarithm term expressing the number of element comparisons performed per delete-min or delete is one. Therefore, the total number of element comparisons performed in all levels is at most $\log n + \log \log n + \ldots + O(\log^{(k)} n)$, where $\log^{(k)}$ denotes the logarithm function applied k times and k is a constant representing the number of levels. An insertion of a new element would result in a constant number of insertions and lazy deletions per level. Hence, the number of levels should be a fixed constant to achieve a constant cost for insertions.

5. MULTIPARTITE BINOMIAL QUEUES

In this section we present a refinement of a two-tier binomial queue, called a *multipartite binomial queue*. 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

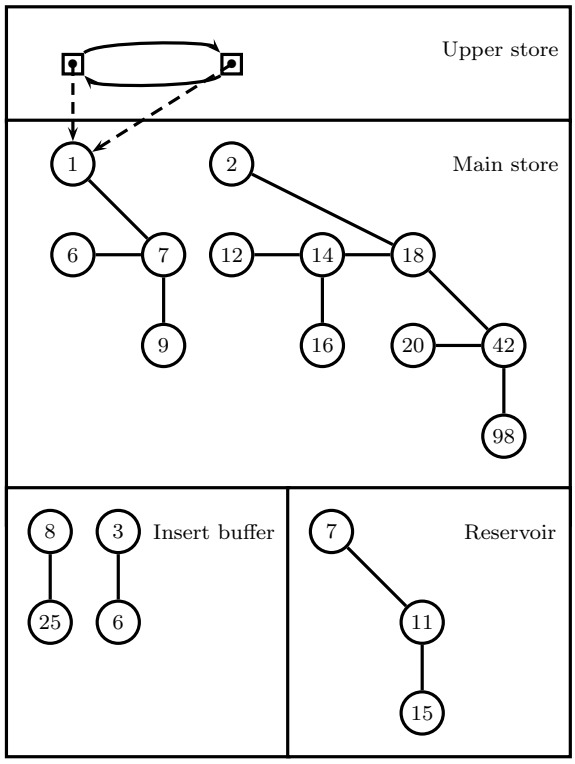


Fig. 2. A multipartite binomial queue storing 19 integers. Prefix-minimum pointers are indicated with dashed lines. No floating tree exists. Binomial trees are drawn in a schematic form (for details, see Figure 1).

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.

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 trees in the insert buffer are between 0 and $\lceil h/2 \rceil$, and the ranks of all 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 a tree that stores the minimum element among those 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 is more than one tree, that 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. 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 (i.e., insert buffer and main store) altogether, we split this 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. 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 validity 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 either 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 trees of lower rank. 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 to follow) 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. □

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 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, namely 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 $\lceil h/2 \rceil$, and each insert-buffer deletion only requires at most $2\lceil h/2 \rceil + O(1)$, 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 maintaining 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 for all trees of higher rank are updated. 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 this 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, explained Section 4.3, is applied. The traversal to the root has worst-case cost $O(\log n)$, but even with this and other overheads, a delete operation has worst-case cost $O(\log n)$ 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 perform 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 Levcopoulos 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 $X = \langle x_1, \ldots, x_n \rangle$, the corresponding *Cartesian tree* [Vuillemin 1980] is a binary tree whose root stores element $x_i = \min\{x_1, \ldots, x_n\}$, the left subtree of the root is the Cartesian tree for sequence $\langle x_1, \ldots, x_{i-1} \rangle$, and the right subtree is the Cartesian tree for sequence $\langle x_{i+1}, \ldots, x_n \rangle$. After building the Cartesian tree, a priority queue is initialized by inserting the element stored at the root of the Cartesian tree into it. In each of the following n iterations, a minimum element stored in the priority queue is 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. Levcopoulos 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

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

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 $X = \langle x_1, x_2, \ldots, x_n \rangle$ of length n, the measures *Osc* and *Inv* are defined as

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

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

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

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

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 $n \log\big(Osc(X)/n\big) + O(n)$ element comparisons when sorting a sequence X of length n. The bounds for the measure *Inv* immediately follow: The cost is $O\big(n \log\big(Inv(X)/n\big)\big)$ and the number of element comparisons performed is $n \log\big(Inv(X)/n\big) + O(n)$. Other adaptive sorting algorithms that guarantee the same bounds are based on insertionsort or mergesort [Elmasry and Fredman 2003; 2008].

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 node borrowing with this efficiency. It would be enough if this operation had worst-case cost $O(\log n)$, including 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; 2008], 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 (refer to 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 for binomial trees in our framework. A navigation pile is a binary tree, and thus three parent-child pointers per node are required. With the standard trick (see, for example [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 aforesaid 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 it may be possible to 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 aforementioned 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 heap should be updated as well. The references from the outside can refer to those 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, it remains to be experimented 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 F. 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.

Elmasry, A. and Fredman, M. L. 2008. Adaptive sorting: An information theoretic perspective. *Acta Informatica 25,* 1, 33–42.

Estivill-Castro, V. and Wood, D. 1992. A survey of adaptive sorting algorithms. *ACM Computing Surveys 24,* 4, 441–476.

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.

Kaplan, H. and Tarjan, R. E. 2008. Thin heaps, thick heaps. *ACM Transactions on Algorithms 4,* 1, Acticle 3.

Katajainen, J. and Mortensen, B. B. 2001. Experiences with the design and implementation of space-efficient deques. 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 deques. *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

Acta Informatica manuscript No.
(will be inserted by the editor)

Amr Elmasry · Claus Jensen · Jyrki Katajainen

Two-tier relaxed heaps

Received: date / Revised: date

Abstract We introduce a data structure which provides efficient heap operations with respect to the number of element comparisons performed. Let n denote the size of the heap being manipulated. Our data structure guarantees the worst-case cost of $O(1)$ for finding the minimum, inserting an element, extracting an (unspecified) element, and replacing an element with a smaller element; and the worst-case cost of $O(\lg n)$ with at most $\lg n + 3 \lg \lg n + O(1)$ element comparisons for deleting an element. We thereby improve the comparison complexity of heap operations known for run-relaxed heaps and other worst-case efficient heaps. Furthermore, our data structure supports melding of two heaps of size m and n at the worst-case cost of $O(\min\{\lg m, \lg n\})$.

A preliminary version of this paper [8] was presented at the 17th International Symposium on Algorithms and Computation held in Kolkata in December 2006.

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).

Amr Elmasry
Department of Computer Engineering and Systems
Alexandria University, Alexandria, Egypt

Claus Jensen
Department of Computing, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark

Jyrki Katajainen
Department of Computing, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark

1 Introduction

In this paper we study (min-)heaps which store a collection of elements and support the following operations:

find-min(H): Return the location of a minimum element held in heap H.
insert(H, p): Insert the element at location p into heap H.
extract(H): Extract an unspecified element from heap H and return the location of that element.
decrease(H, p, e): Replace the element at location p in heap H with element e, which must be no greater than the element earlier located at p.
delete(H, p): Remove the element at location p from heap H.
meld(H_1, H_2): Create a new heap containing all the elements held in heaps H_1 and H_2, and return that heap. This operation destroys H_1 and H_2.

Observe that *delete-min*(H), which removes the current minimum of heap H, can be accomplished by invoking *find-min* and thereafter *delete* with the location returned by *find-min*. In the heaps studied, the location abstraction is realized by storing elements in nodes, moving nodes around inside a data structure by updating pointers (without changing positions of the nodes), and passing pointers to these nodes. (For more information about the location abstraction, see e.g. [14, Section 2.4.4].) Please note that *extract* is a non-standard operation, not supported by earlier data structures, but in our experience [6,7,9] it is useful in data-structural transformations.

The research reported in this paper is a continuation of our earlier work aiming to reduce the number of element comparisons performed in heap operations. In [9], we described how the comparison complexity of heap operations can be improved using a multi-component data structure which is maintained by moving nodes from one component to another. In a technical report [6], we were able to add *decrease* having the worst-case cost of $O(1)$ to the operation repertoire. Unfortunately, the resulting data structure was complicated. In this paper we make the data structure simpler and more elegant by utilizing the connection between number systems and data structures (see, for example, [18]).

It should be emphasized that we make no claims about the practical utility of the data structures considered; the main motivation for our study is theoretical. More precisely, we are interested in the comparison complexity of heap operations. We assume the availability of the standard set of primitive operations, including memory allocation and memory deallocation functions, as defined, for example, in the C programming language [17]. Since the running time of an algorithm may vary depending on the duration of individual primitive operations executed, we use the term *cost* to denote the sum of the total number of primitive operations, element constructions, element destructions, and element comparisons executed.

For the data structures considered our basic requirement is that the worst-case cost of *find-min*, *insert*, and *decrease* is $O(1)$. Given this constraint, our goal is to reduce the number of element comparisons involved in *delete*, and to keep the data structure meldable. Binary heaps [21] are to be excluded based on the fact that $\lg \lg n - O(1)$ element comparisons are necessary for inserting an element into a heap of size n [13]. (Here, n denotes the number of

elements stored in the data structure prior to the operation in question, and $\lg n$ equals $\log_2(\max\{2, n\})$.) Also, pairing heaps [11] are excluded because they cannot guarantee *decrease* at a cost of $O(1)$ [10]. There exist several heaps that achieve a cost of $O(1)$ for *find-min*, *insert*, and *decrease*; and a cost of $O(\lg n)$ for *delete*. Fibonacci heaps [12] and thin heaps [16] achieve these bounds in the amortized sense. Run-relaxed heaps [5], fat heaps [15,16], and the meldable heaps described in [1] achieve these bounds in the worst case. When these data structures are extended to support *meld*, in addition to the other operations, the performance of *meld* is as follows: Fibonacci heaps and thin heaps achieve the amortized cost of $O(1)$, run-relaxed heaps and fat heaps can achieve the worst-case cost of $O(\min\{\lg m, \lg n\})$, and meldable priority queues achieve the worst-case cost of $O(1)$, m and n being the sizes of the data structures melded.

For all the aforementioned heaps guaranteeing a cost of $O(1)$ for *find-min* and *insert*, the number of element comparisons performed by *delete* is at least $2\lg n - O(1)$, and this is true even in the amortized sense (for binomial heaps [20], on which many of the above data structures are based, this is proved in [6,9]). Run-relaxed heaps have the worst-case upper bound of $3\lg n + O(1)$ on the number of element comparisons performed by *delete* (see Section 3). According to our analysis, the corresponding bound for fat heaps is $4\log_3 n + O(1) \approx 2.53\lg n + O(1)$.

Like a binomial heap [20] (see also [4, Chapter 19]), a run-relaxed heap [5] is composed of a logarithmic number of binomial trees, but in these trees there can be a logarithmic number of heap-order violations. In this paper we present a new adaptation of run-relaxed heaps, called a *two-tier relaxed heap*. In Section 2, we start by discussing the connection between number systems and data structures; among other things, we show that it is advantageous to use a zeroless representation of a binomial heap which guarantees that a non-empty heap always contains at least one binomial tree of size one. In Section 3, we give a brief review of the implementation of heap operations on run-relaxed heaps. In Section 4, we describe our data structure and prove that it guarantees the worst-case cost of $O(1)$ for *find-min*, *insert*, *extract*, and *decrease*; the worst-case cost of $O(\lg n)$ with at most $\lg n + 3\lg\lg n + O(1)$ element comparisons for *delete*; and the worst-case cost of $O(\min\{\lg m, \lg n\})$ for *meld*. In Section 5, we conclude the paper with a few final remarks.

2 Number systems and binomial heaps

In order to perform *delete* efficiently, it is important that an unspecified node can be extracted from a heap so that no more than O(1) structural changes (node removals or pointer updates) are made to the data structure. In this section we describe how to realize *extract* (and *insert*). We rely on the observation that, in a binomial heap (and in a run-relaxed heap), there is a close connection between the sizes of the binomial trees of the heap and the number representation of the current size of the heap.

A *binomial tree* [20] is a rooted, ordered tree defined recursively as follows: A binomial tree of rank 0 is a single node; for $r > 0$, a binomial tree of rank r consists of the root and its r binomial subtrees of ranks 0, 1, ..., $r - 1$

connected to the root in that order. We denote the root of the subtree of rank 0 the *smallest child* and the root of the subtree of rank $r-1$ the *largest child*. The size of a binomial tree is always a power of two, and the rank of a tree of size 2^r is r.

Each node of a binomial tree stores an element drawn from a totally ordered set. Binomial trees are maintained *heap-ordered* meaning that the element stored at a node is no greater than the elements stored at the children of that node. As any multiary tree, a binomial tree can be implemented using the *child-sibling representation*. At each node, in addition to an element, space is reserved for a rank and four pointers: a child pointer, a parent pointer, and two sibling pointers. The children of a node are kept in a doubly-linked list, called the *child list*, and the child pointer points to the largest child. If a node has no children or sibling, the corresponding pointer has the value *null*. As for parent pointers, we want to emphasize that our representation is non-standard. If a node is a root, its parent pointer points to the node itself; and if a node is the largest child of its parent, the parent can be accessed via the parent pointer. For all other nodes the parent pointer has the value *null*. The advantage of maintaining parent pointers only for the largest children is that each node has a constant in-degree and it stores pointers back to all referring nodes, which enables the detachment of a node at a cost of $O(1)$.

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

A *binomial heap* is composed of at most a logarithmic number of binomial trees, the roots of which are kept in a doubly-linked list, called the *root list*, in increasing rank order. For binomial heaps, four different number representations are relevant:

Binary representation:
$n = \sum_{i=0}^{\lfloor \lg n \rfloor} d_i 2^i$, where $d_i \in \{0,1\}$ for all $i \in \{0,1,\ldots,\lfloor \lg n \rfloor\}$.

Redundant representation:
$n = \sum_{i=0}^{\lfloor \lg n \rfloor} d_i 2^i$, where $d_i \in \{0,1,2\}$ for all $i \in \{0,1,\ldots,\lfloor \lg n \rfloor\}$.

Skew representation:
$n = \sum_{i=0}^{\lfloor \lg(n+1) \rfloor} d_i(2^{i+1}-1)$, where every $d_i \in \{0,1\}$, except that the lowest non-zero digit d_i may also be 2.

Zeroless representation:
$n = \sum_{i=0}^{k} d_i 2^i$, where $k \in \{-1,0,\ldots,\lfloor \lg n \rfloor\}$ and $d_i \in \{1,2,3,4\}$ for all $i \in \{0,\ldots,k\}$. If $k=-1$, this means that the heap is empty.

For each representation, the heap of size n contains d_i binomial trees of size 2^i, or d_i skew binomial trees of size $2^{i+1} - 1$ (for the definition of a skew binomial tree, see [2]).

Now *insert* can be realized elegantly by imitating increments in the underlying number system. The worst-case efficiency of *insert* is directly related to how far a carry has to be propagated. If the binary representation is used as in [4, Chapter 19], *insert* has the worst-case cost of $\Theta(\lg n)$. If the redundant representation is used as, for example, in [6,9], *insert* has the worst-case cost of $O(1)$. And as shown in [2], the same holds true if the skew representation is used. The overall strategy is to perform at most one join in connection with each insertion, and execute delayed joins (if any) in connection with forthcoming *insert* operations.

Using the zeroless representation (see [3, p. 56 ff.] and [15]), both *insert* and *extract* can be supported at the worst-case cost of $O(1)$. To achieve this, pointers to every carry (digit 4 which corresponds to four consecutive binomial trees of the same rank) and every borrow (digit 1) are kept on a stack in rank order with the smallest rank on top. When a *carry/borrow stack* is available, increments and decrements can be performed as follows:

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

To fix a carry, the digit 4 is changed to 2 and the next higher digit is increased by 1. Analogously, to fix a borrow, the digit 1 is changed to 3 and the next higher digit is decreased by 1. After every such change, the top of the carry/borrow stack is popped and, if a fix creates a new carry or borrow, an appropriate pointer is pushed onto the stack. Observe that carries and borrows at the highest order position require special handling. If the most significant digit is 4, fixing it produces 1 at the new highest order position, but a pointer to this digit is *not* pushed onto the carry/borrow stack. A transition from 2 to 1 is also possible, but 1 at the highest order position is never considered to be a borrow. In terms of binomial trees, fixing a carry means that a join is made, which produces a binomial tree whose rank is one higher; and fixing a borrow means that a split is made, which produces two binomial trees whose rank is one lower.

A sequence of digits $\langle d_i, d_{i+1}, \ldots, d_j \rangle$, each drawn from the set $\{1, 2, 3, 4\}$, is called a 4-*block* (1-*block*) if $d_i = 4$ ($d_i = 1$), $d_j = 4$ ($d_j = 1$), and any other digit between d_i and d_j is different from 4 (1). A *regular* zeroless representation satisfies the property that in any 4-block among the digits between the two endpoints there is a digit other than 3, and in any 1-block among the digits between the two endpoints there is a digit other than 2, except when the 1-block ends with the most significant digit. The correctness of *insert* and *extract* can be proved by showing that both operations maintain the representation regular (for a similar treatment, see [3, p. 56 ff.] or [15]). Since our algorithms are different from those discussed in the earlier sources, their correctness is proved in the following lemma.

Lemma 1 *Each insert and extract keeps the zeroless representation regular.*

Table 1 Changes made to the representation at the different steps. We use "–" to indicate that the digit does not exist and "ⓧ" that x is the most significant digit.

Step	Before d_j	Before d_{j+1}	After d_j	After d_{j+1}	Remark
Fix carry d_j	④	–	2	1	
	4	ⓧ, $x \in \{1,2,3\}$	2	$x+1$	
	4	1	2	2	
	4	2	2	3	
	4	3	2	4	Case I
	4	4 or ④			Not possible
Fix borrow d_j	1	①	3	–	
	1	ⓧ, $x \in \{2,3,4\}$	3	$x-1$	
	1	1			Not possible
	1	2	3	1	Case II
	1	3	3	2	
	1	4	3	3	
Increase d_0 (i.e. $j=0$)	–	–	1	–	
	ⓧ, $x \in \{1,2,3\}$	–	$x+1$	–	
	1	x or ⓧ			Not possible
	2	x or ⓧ	3	x	
	3	x or ⓧ	4	x	Case III
	4 or ④	– or x or ⓧ			Not possible
Decrease d_0 (i.e. $j=0$)	①	–	–	–	
	ⓧ, $x \in \{2,3\}$	–	$x-1$	–	
	1	x or ⓧ			Not possible
	2	x or ⓧ	1	x	Case IV
	3	x or ⓧ	2	x	
	4 or ④	– or x or ⓧ			Not possible

Proof The condition is that the representation is regular before each operation, and the task is to show that the same holds true after the operation. Table 1 depicts the changes made to the representation at the different steps of the *insert* and *extract* algorithms. The cases given without a remark are trivially seen to keep the representation regular because either no new blocks are introduced or some blocks just cease to exist. Because of the regularity of the initial representation, or because of the fix performed just before an increment or a decrement, some of the cases are impossible. It is straightforward to verify that the regularity is preserved in the remaining cases.

Case I: The first digit of the 4-block (if any) starting with d_j is moved to the next higher position. By the regularity of the initial representation, the 4-block must contain a digit other than 3 since d_{j+1} was 3.

Case II: This is similar to Case I, but now the first digit of a 1-block (if any) is moved to the next higher position. By the regularity condition, one of the digits in the old 1-block makes the new 1-block valid.

Case III: Before the increment, d_1 (x) cannot be 4 because a fix would have removed this carry. If $d_1 \in \{1,2\}$, it makes the 4-block created (if any) valid. The case $d_1 = 3$ is a bit more involved. If before the increment no fixing was necessary, the new value of d_0 (4) does not start a 4-block and the representation remains regular. Otherwise, some digit d_j, $j \geq 0$, has

just been fixed. If $d_j = 4$ before the fix, $d_j = 2$ after the fix, which makes the 4-block (if any) starting with d_0 valid. If $d_j = 1$ and $d_{j+1} \in \{2, 3\}$ before the fix, $d_{j+1} \in \{1, 2\}$ after the fix, which makes the 4-block (if any) starting with d_0 valid. If $d_j = 1$ and $d_{j+1} = 4$ before the fix, and if d_{j+1} did not start a 4-block, d_0 does not start a 4-block and the representation remains regular. Finally, if $d_j = 1$ and $d_{j+1} = 4$ before the fix, and if d_{j+1} started a 4-block, this 4-block contained a digit other than 3 making the 4-block starting with d_0 valid. That is, the representation remains regular because of the fix performed just prior to the increment.

Case IV: This is symmetric to Case III, but now a 1-block may be created. The regularity is shown to be preserved as above. □

To sum up, *insert* is carried out by doing at most one join or split depending on the contents of the carry/borrow stack, and injecting a new node as a binomial tree of rank 0 into the root list. Correspondingly, after a join or split, if any, *extract* ejects one binomial tree from the root list. Due to the regular zeroless representation, we can be sure that the rank of every ejected binomial tree is 0, i.e. it is a single node. However, if the ejected node contains the current minimum and the heap is not empty, another node is extracted, and the node extracted first is inserted back into the data structure. Clearly, *insert* and *extract* are accomplished at the worst-case cost of $O(1)$.

3 Run-relaxed heaps

Since we use modified run-relaxed heaps as the basic building blocks of two-tier relaxed heaps, we describe the details of run-relaxed heaps, including our modifications, in this section. However, we still assume that the reader is familiar with the original paper by Driscoll et al. [5].

A *relaxed binomial tree* [5] is an almost heap-ordered binomial tree where some nodes are denoted *active*, indicating that the element stored at that node may be smaller than the element stored at the parent of that node. Nodes are made active by *decrease* regardless of whether a heap-order violation is actually introduced. A touched node remains active until the potential heap-order violation is explicitly removed. From the definition, it directly follows that a root cannot be active. A *singleton* is an active node whose immediate siblings are not active. A *run* is a maximal sequence of two or more active nodes that are consecutive siblings.

A *run-relaxed heap* is a collection of relaxed binomial trees. Let τ denote the number of trees in such a collection and λ the number of active nodes in the entire collection. In the original version of a run-relaxed heap [5], an invariant is maintained that $\tau \leq \lfloor \lg n \rfloor + 1$ and $\lambda \leq \lfloor \lg n \rfloor$ after every heap operation, where n denotes the number of elements stored. For run-relaxed heaps relying on different number representations almost the same bounds can be shown to hold.

Lemma 2 *In a run-relaxed heap of size n, relying on the binary, redundant, or zeroless representation, the rank of a tree can never be higher than $\lfloor \lg n \rfloor$.*

Proof Let the highest rank be r. None of the trees can be larger than the size of the whole heap, i.e. $2^r \leq n$. Thus, since r is an integer, $r \leq \lfloor \lg n \rfloor$. □

Corollary 1 *Let τ denote the number of relaxed binomial trees in a run-relaxed heap of size n relying on the regular zeroless representation. It must be that $\tau \leq 3\lfloor \lg n \rfloor + O(1)$.*

Proof Since there are at most 4 trees at each rank, by the previous lemma $4(\lfloor \lg n \rfloor + 1)$ would be a trivial upper bound for the number of trees. The tighter bound follows since in the regular zeroless representation in every 4-block there is a digit other than 3. □

To keep track of the active nodes, a *run-singleton structure* is maintained as described in [5]. All singletons are kept in a *singleton table*, which is a resizable array accessed by rank. In particular, this table must be implemented in such a way that growing and shrinking at the tail is possible at the worst-case cost of $O(1)$, which is achievable, for example, by doubling, halving, and incremental copying. Each entry of the singleton table corresponds to a rank; pointers to singletons rooting a tree of this rank are kept in a list. For each entry of the singleton table that has more than one singleton, a counterpart is kept in a *pair list*. The last active node of each run is kept in a *run list*. All lists are doubly linked, and each active node should have a pointer to its occurrence in a list (if any). The bookkeeping details are quite straightforward so we will not repeat them here. The fundamental operations supported are an addition of a new active node, a removal of a given active node, and a removal of at least one active node if λ becomes too large. The cost of each of these operations is $O(1)$ in the worst case.

Two types of transformations are used when removing active nodes: *singleton transformations* are used for combining singletons and *run transformations* are used for making runs shorter. A pictorial description of the transformations is given in the appendix. Compared to the original transformations given in [5], our new contribution is that the transformations can be made to work even if the parent pointers are only maintained for the largest children. The transformations give us a mechanism to bound λ from above.

Lemma 3 *Let λ denote the number of active nodes in a run-relaxed heap of size n relying on the binary, redundant, or zeroless representation. If $\lambda > \lfloor \lg n \rfloor$, the transformations can be applied to reduce λ by at least one.*

Proof Presume that $\lambda > \lfloor \lg n \rfloor$. If none of the run transformations or the singleton transformations applies, there must be at least $\lfloor \lg n \rfloor + 1$ singletons rooting subtrees of different ranks. This is impossible, and the presumption must be wrong, because of the following two facts:

1. The highest rank can be at most $\lfloor \lg n \rfloor$ by the previous lemma.
2. The root of a tree of rank $\lfloor \lg n \rfloor$ cannot be active. □

The rationale behind the transformations is that, when there are more than $\lfloor \lg n \rfloor$ active nodes, there must be at least one pair of singletons that root a subtree of the same rank, or there is a run of two or more neighbouring

active nodes. In both cases, it is possible to apply the transformations—a constant number of singleton transformations or a constant number of run transformations—to reduce the number of active nodes by at least one. The worst-case cost of performing any of the transformations is $O(1)$. One application of the transformations together with all necessary changes to the run-singleton structure is referred to as a *λ-reduction*.

To keep track of the trees in our modification of a run-relaxed heap, a root list and a carry/borrow stack are maintained and the representation is kept zeroless as described in Section 2. Each relaxed binomial tree is represented as a binomial tree, but to support the transformations used for reducing the number of active nodes each node stores an additional pointer to its occurrence in the run-singleton structure. The occurrence pointer of every non-active node has the value null; for a node that is active and in a run, but not the last in the run, the pointer is set to point to a fixed sentinel. To support our two-tier relaxed heap, each node should store yet another pointer to its counterpart held at the upper store, and vice versa (see Section 4).

In the root list, the relaxed binomial trees of the same rank are maintained in sorted order according to the elements stored at the roots. Recall that there are at most four trees per rank. Since in our data structure each heap operation only modifies a constant number of trees, the cost of maintaining the sorted order within each rank is $O(1)$ per operation. The significant consequence of this ordering is that *delete* only has to consider one root per rank when finding a minimum element stored at the roots.

Let us now consider how the heap operations are implemented. A reader familiar with the original paper by Driscoll et al. [5] should be aware that we have made modifications to their implementations of heap operations to adapt them for our purposes.

A minimum element is stored at one of the roots or at one of the active nodes. To facilitate a fast *find-min*, a pointer to the node storing a minimum element is maintained. When such a pointer is available, *find-min* can be accomplished at the worst-case cost of $O(1)$. Observe that in [5] such a pointer was not maintained resulting in *find-min* having logarithmic cost.

An insertion is performed as described in Section 2. Additionally, if the inserted element is smaller than the current minimum, the pointer indicating the location of a minimum element is updated. On the basis of this, *insert* has the worst-case cost of $O(1)$. In [5] a completely different method for achieving the same bound was described.

An extraction is performed as described in Section 2, so *extract* has the worst-case cost of $O(1)$. In [5] a borrowing technique was described which could be used to implement *extract* having logarithmic cost.

In *decrease*, after making the element replacement, the corresponding node is made active, an occurrence is inserted into the run-singleton structure, and a single λ-reduction is performed if the number of active nodes is larger than $\lfloor \lg n \rfloor$. Moreover, if the given element is smaller than the current minimum, the pointer indicating the location of a minimum element is corrected to point to the decreased node. All these actions have the worst-case cost of $O(1)$. Compared to [5], we make the decreased node unconditionally

active, since the decreased node does not necessarily have a parent pointer facilitating a fast check whether a heap-order violation is introduced or not.

In *delete* we want to make as few structural changes as possible so we rely on *extract* to get a replacement for the deleted node. Let x be the node extracted and z the node to be deleted. (Recall that x only contains the current minimum if the heap is of size one.) Deletion has two cases depending on whether one of the roots or one of the internal nodes is to be removed.

Assume that z is a root. If x and z are the same node, that node is already removed and no other structural changes are necessary. Otherwise, the tree rooted at z is repeatedly split until the tree rooted at z has rank 0, and then z is removed. In these splits all active children of z are retained active, but they are temporarily removed from the run-singleton structure (since the structure of runs may change). After this the tree of rank 0 rooted at the extracted node x and the subtrees rooted at the children of z are repeatedly joined by processing the trees in increasing order of rank. The active nodes temporarily removed are added back to the run-singleton structure. The resulting tree replaces the tree rooted at z in the root list. If z contained the current minimum, all roots containing the smallest element of each rank and active nodes are scanned to update the pointer indicating the location of a minimum element. Singletons are found by scanning through all lists in the singleton table. Runs are found by accessing their last node via the run list and following the sibling pointers until a non-active node is reached. It would be possible to remove all active children of z, but when *delete* is embedded into our two-tier relaxed heap, it would be too expensive to remove many active nodes in the course of a single *delete* operation.

The computational cost of deleting a root is dominated by the repeated splits, the repeated joins, and the scan of all minimum candidates. In each of these steps a logarithmic number of nodes is visited, so the total cost is $O(\lg n)$. Splits and the actions on the run-singleton structure do not involve any element comparisons. In total, joins may involve at most $\lfloor \lg n \rfloor$ element comparisons. If the number of active nodes is larger than $\lfloor \lg(n-1) \rfloor$ (the size of the heap is now one smaller), a single λ-reduction is performed involving $O(1)$ element comparisons. To find the minimum of $2\lfloor \lg n \rfloor + 1$ elements, at most $2\lfloor \lg n \rfloor$ element comparisons are needed. To summarize, this form of *delete* performs at most $3 \lg n + O(1)$ element comparisons.

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

Similar to the case of deleting a root, the deletion of an internal node has the worst-case cost of $O(\lg n)$. If z did not contain the current minimum,

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

In [5] a description of *meld* was not given, but it is relatively easy to accomplish *meld* in a manner similar to the one described in Section 4.

4 Two-tier relaxed heaps

A two-tier relaxed heap is composed of two components: the *lower store* and the *upper store* (see Figure 1). The lower store stores the actual elements of the heap. The reason for introducing the upper store is to avoid the scan over all minimum candidates when updating the pointer to the location of a minimum element; pointers to minimum candidates are kept in a heap instead. Actually, both components are realized as run-relaxed heaps modified to use the zeroless representation as discussed in Section 3.

4.1 Upper-store operations

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

To provide worst-case efficient lazy deletions, we adopt the global-rebuilding technique from [19]. 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 insertions and extractions are handled by the new upper store, and all decreases, deletions, markings, and unmarkings by the respective upper stores. The heap operations are realized as described earlier with the following exceptions. In *delete* the replacement node is taken from the new upper store. In *decrease* a complication is that one should know the component in which the given node lies. (This issue was discussed in depth in [15].) To facilitate this, each node includes a bit indicating its component. This information is enough to access the right run-singleton structure. Each time a node is moved from the

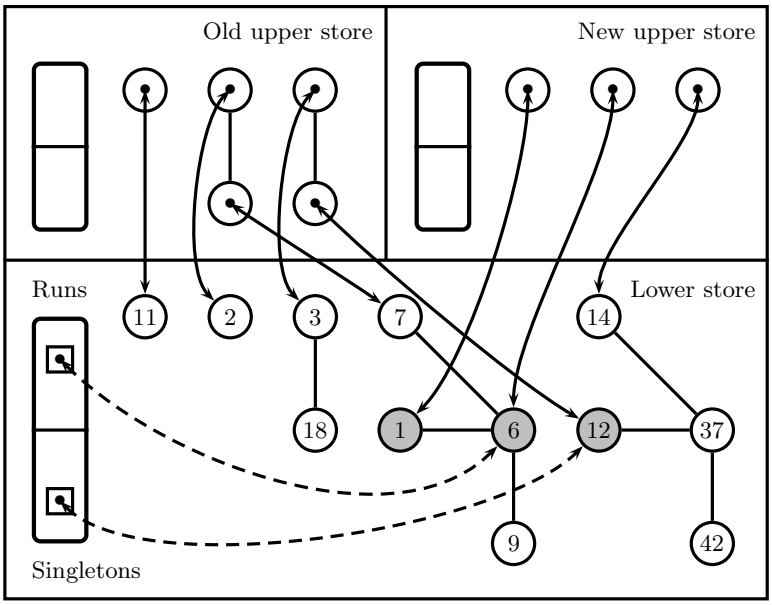


Fig. 1 A two-tier relaxed heap storing 12 integers. The relaxed binomial trees are drawn in schematic form and active nodes are drawn in grey. At the upper store a global rebuilding is in progress.

old upper store to the new upper store and vice versa, its component bit is simply flipped.

After initiating a reorganization, in connection with each of the next at most $m_0/4$ upper-store operations, four nodes are extracted from the old structure; if the node under consideration is unmarked, it is inserted into the new structure; otherwise, it is released and at its counterpart in the lower store the pointer to the upper store is given the value null. Since the old upper store does not handle any insertions, each heap operation decreases its size by four. When the old structure becomes empty, it is dismissed and the new structure is used alone. During the $m_0/4$ heap operations at most $m_0/4$ nodes are extracted, 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. Thus, at any point in time, we are constructing at most one new structure. We emphasize that each node only exists in one structure and that nodes are moved around by updating pointers, so pointers from the outside remain valid.

Given that the cost of each *extract* and *insert* is $O(1)$, the reorganization only incurs an additive cost of $O(1)$ to all upper-store operations. A *find-min* has to consult both the old and the new upper stores, but its worst-case cost is still $O(1)$. The cost of marking and unmarking is clearly $O(1)$. If m denotes the total number of unmarked nodes currently stored, at any point during the rebuilding process the total number of nodes stored is $\Theta(m)$. Therefore,

since in both structures *delete* is handled normally, except that it may take part in reorganizations, it has the worst-case cost of $O(\lg m)$ and requires at most $3 \lg m + O(1)$ element comparisons.

Let n be the number of elements in the lower store. The number of trees in the lower store is at most $3\lfloor \lg n \rfloor + O(1)$, and the number of active nodes is at most $\lfloor \lg n \rfloor$. At all times at most a constant fraction of the nodes stored at the upper store can be marked to be deleted. Hence, the number of pointers is $O(\lg n)$. That is, at the upper store the worst-case cost of *delete* is $O(\lg \lg n)$, including at most $3 \lg \lg n + O(1)$ element comparisons.

4.2 Lower-store operations

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

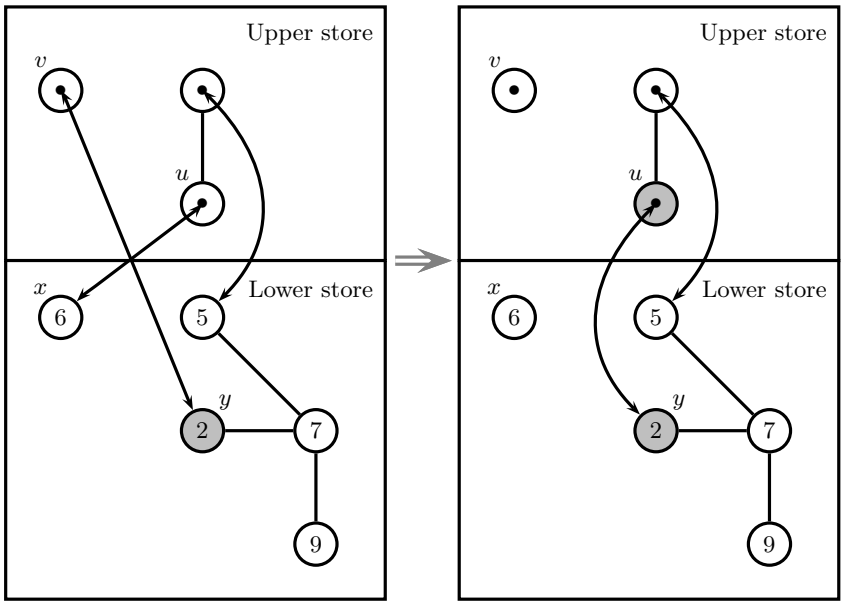
An insertion is carried out as described in Section 3, but *insert* requires three modifications in places where communication between the lower store and upper store is necessary. First, in each join the counterpart of the loser tree must be lazily deleted from the upper store. Second, in each split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

In comparison with *decrease* described in Section 3, three modifications are necessary. First, each time a new active node is created, *insert* has to be invoked at the upper store. Second, each time an active node is removed by a λ-reduction, the counterpart must be lazily deleted from the upper store. Third, when the node whose value is to be decreased is a root or an active node, *decrease* has to be invoked at the upper store as well. If an active node becomes a root due to a λ-reduction, no change at the upper store is required. After these modifications, the worst-case cost of *decrease* is still $O(1)$.

An extraction is done as described in Section 3, but again the upper store must be synchronized accordingly. Let x be the node extracted from the lower store and let u be its counterpart at the upper store. Since the deletion of u may be too expensive, an *extract* is performed at the upper store as well. Let v be the node extracted from the (new) upper store and let y be its counterpart at the lower store. If we are so lucky that u and v are the same node, we are done; u is released and x is returned. Otherwise, if u and v are not the same node, there are two cases depending on the contents of nodes x and y (for an illustration, see Figure 2).

Case I: If the element stored at x is no smaller than that stored at y, the pointer at u is set to point to y (instead of x) and *decrease* is invoked on u at the upper store. At the end, v is released and x is returned.

Case I: If the element stored at x is no smaller than that stored at y, the pointer at u is set to point to y and *decrease* is invoked on u at the upper store.



Case II: If the element stored at x is smaller than that stored at y, the nodes x and y are swapped, and *decrease* is invoked on x at the lower store.

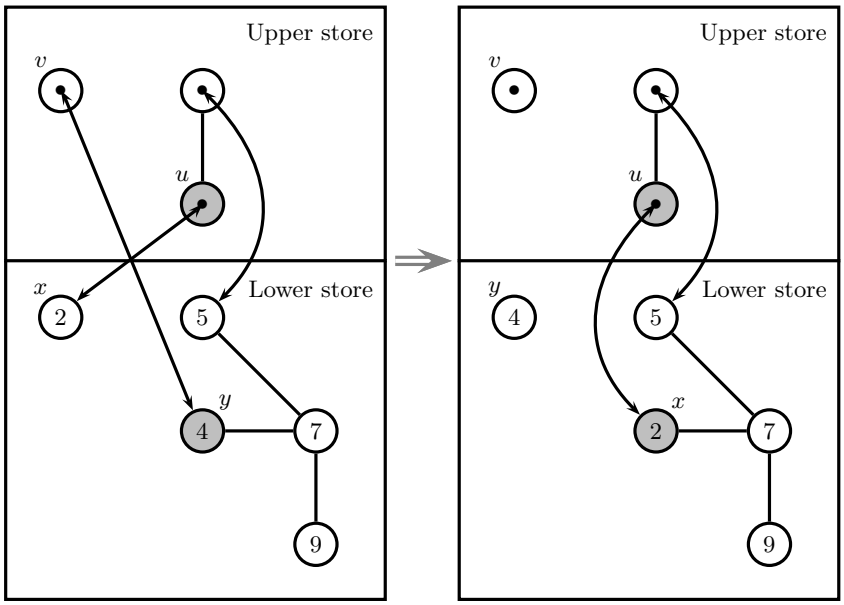


Fig. 2 Illustration of the syncronization between the upper store and the lower store when extracting an element from a two-tier relaxed heap. Only the trees worked with are drawn.

Case II: If the element stored at x is smaller than that stored at y, the nodes x and y are swapped, and *decrease* is invoked on x at the lower store. Since u still points to x, no change is necessary at the upper store, other than those caused by *decrease*. In this case, v is released and y is returned.

When swapping x and y no elements are moved, but the nodes are detached from and reattached into the lower store by updating pointers, which retains the integrity of all references. Also, even if u and v can come from a different component, *decrease* operations are handled correctly since at the upper store the nodes know the component in which they lie. Since *decrease* has the worst-case cost of $O(1)$ both at the upper and the lower store, *extract* has a total cost of $O(1)$.

Deletion is done as described in Section 3, except that scanning for a new minimum can be omitted because minimum finding is handled by the upper store. When extracting a node its counterpart is kept at the upper store and no *decrease* operation is necessary, since the extracted node is put back into the lower store. As a consequence of *extract*, lazy deletions and/or insertions may be necessary at the upper store. As a consequence of *delete*, a removal of a root or an active node will invoke *delete* at the upper store, and an insertion of a new root or a new active node will invoke *insert* at the upper store. A λ-reduction may invoke one or two lazy deletions (a λ-reduction can make up to two active nodes non-active) and at most one insertion at the upper store. In total, lazy deletions and insertions have the worst-case cost of $O(1)$. Also *extract* has the worst-case cost of $O(1)$. At most one real upper-store deletion will be necessary, which has the worst-case cost of $O(\lg \lg n)$ and includes at most $3 \lg \lg n + O(1)$ element comparisons. Joins performed at the lower store incur at most $\lfloor \lg n \rfloor$ element comparisons, whereas the scan of a new minimum is avoided saving up to $2\lfloor \lg n \rfloor$ element comparisons. Therefore, *delete* has the worst-case cost of $O(\lg n)$ and performs at most $\lg n + 3 \lg \lg n + O(1)$ element comparisons.

4.3 Operations on a two-tier relaxed heap

In a two-tier relaxed heap the heap operations are realized by invoking the operations available at its components: *find-min* invokes that provided for the upper store; and *insert*, *extract*, *decrease*, and *delete* invoke the corresponding operations provided for the lower store. Next we take a closer look at *meld* which heavily relies on the fact that the sizes of both the upper store and the run-singleton structure of the lower store are logarithmically bounded in the number of elements stored.

Consider a *meld* operation where two two-tier relaxed heaps, H_1 and H_2, are to be melded. Without loss of generality, we assume that the maximum rank r of H_1 is smaller than or equal to the maximum rank of H_2. At first the carry/borrow stack of H_2 is repeatedly popped and carries/borrows are fixed until the index of the topmost carry/borrow is larger than r or the stack becomes empty. Thereafter the roots of H_1 are inserted into H_2 in order of increasing rank. For each rank, the trees are joined until two or three trees are

left and carries are propagated forward. When rank r is reached, a fix-add-push step is performed at rank $r+1$ for every carry produced at rank r. The counterparts of the roots and active nodes in the upper store of H_1 are added to the upper store of H_2. The runs of H_1 are traversed and the active nodes in these runs are inserted into the run-singleton structure of H_2. The singletons of H_1 are traversed and the active nodes are inserted into the run-singleton structure of H_2. The operation is completed by performing λ-reductions in H_2 until the number of active nodes is under the permitted bound. Finally, the upper store and run-singleton structure of H_1 are destroyed.

It can easily be shown by induction that during melding at each rank the incoming carry can be at most 5. The carry/borrow stack is popped at most $O(r)$ times, and each fix has a cost of $O(1)$. At most $3r + O(1)$ new trees are added to the lower store of H_2, and at most $O(r)$ joins are performed. When melding H_1 into H_2, $O(r)$ insertions into and lazy deletions from the upper store of H_2 may be necessary. The cost of each of these operations is $O(1)$. Each of the at most r insertions into the run-singleton structure has a cost of $O(1)$. Clearly, the number of λ-reductions performed is bounded by $O(r)$, and each has a cost of $O(1)$. Finally, at most $O(r)$ nodes have to be visited in order to destroy the nodes in the upper store and run-singleton structure of H_1, for a cost of $O(1)$ per node. When the respective sizes of H_1 and H_2 are m and n, it must be that r is $O(\min\{\lg m, \lg n\})$. Therefore, the worst-case cost of *meld* is $O(\min\{\lg m, \lg n\})$.

The following theorem summarizes the main result of the paper.

Theorem 1 *Let n be the number of elements in the heap prior to each operation. A two-tier relaxed heap guarantees the worst-case cost of $O(1)$ for find-min, insert, extract, and decrease; and the worst-case cost of $O(\lg n)$ with at most $\lg n + 3\lg\lg n + O(1)$ element comparisons for delete. The worst-case cost is $O(\min\{\lg m, \lg n\})$ for meld, where m and n are the number of elements in the two heaps melded.*

5 Concluding remarks

We described an adaptation of run-relaxed heaps which provides heap operations that are almost optimal with respect to the number of element comparisons performed. It is possible to use other heap structures than run-relaxed heaps as the building blocks in our two-tier framework. The data structure could be simplified by substituting run-relaxed heaps with less complicated structures like thin heaps [16] or fat heaps [15,16]. However, this would have a consequence on the comparison complexity of *delete*. For example, for fat heaps the worst-case bound on the number of element comparisons performed by *delete* becomes $2\log_3 n + o(\lg n) \approx 1.27\lg n + o(\lg n)$.

The two-tier approach used in this paper can be extended quite easily to three or more tiers. Using k levels of heaps, for a fixed constant $k > 2$, the number of element comparisons performed by *delete* would reduce to $\lg n + \lg\lg n + \ldots + O(\lg^{(k)} n)$, where $\lg^{(k)}$ denotes the logarithm function applied k times. For a description on how this can be realized, see [6,9].

As to the comparison complexity of heap operations, three questions are left open.

1. Is it possible or not to achieve a bound of $\lg n + O(1)$ element comparisons per *delete* when efficient *decrease* is to be supported? Note that the worst-case bound of $\lg n + O(1)$ is achievable [6,9] if *decrease* is allowed to have logarithmic cost.
2. Can global rebuilding performed at the upper store be eliminated so that the constant number of element comparisons performed by *insert*, *extract*, and *decrease* is reduced?
3. What is the lowest number of element comparisons performed by *delete* under the constraint that all other heap operations, including *meld*, are required to have the worst-case cost of $O(1)$?

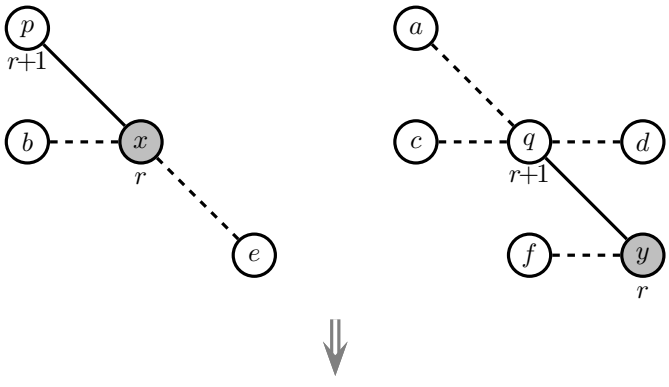
Significant simplifications are called for to obtain a practical data structure that supports *find-min*, *insert*, *extract*, *decrease*, *delete*, and *meld* at optimal or near-optimal worst-case cost. We leave the challenge of devising such a data structure for the reader.

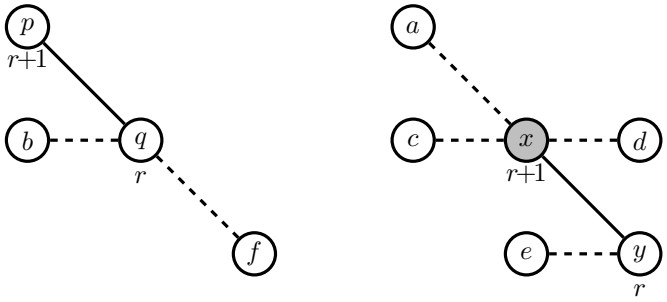
Acknowledgements We thank the anonymous referees for their constructive comments that improved the presentation of this paper.

Appendix

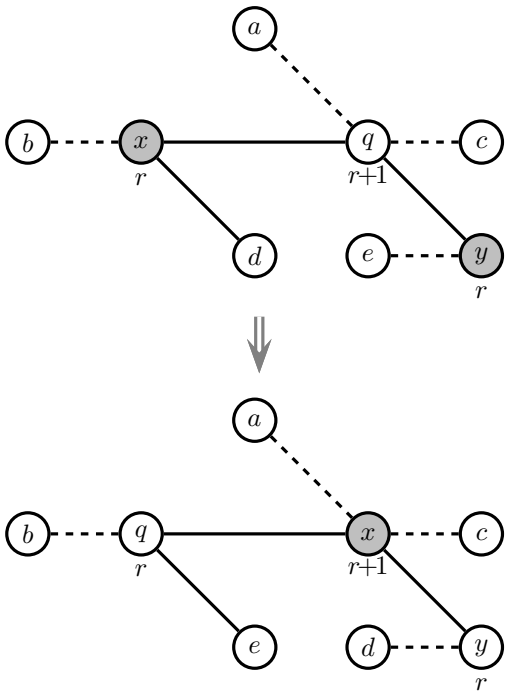
In this appendix a pictorial description of the transformations applied in a λ-reduction is given. In a singleton transformation two singletons x and y are given, and in a run transformation the last active node z of a run is given. In the following only the relevant nodes for each transformation are drawn, all active nodes are drawn in grey, and $element[p]$ denotes the element stored at node p. Dashed lines are connections to nodes that may not exist.

Singleton transformation I: Both x and y are the last children of their parents p and q, respectively. Assume that $element[p] \leq element[q]$ and that $element[x] \leq element[y]$. (There are three other cases which are similar.) Note that this transformation works even if x and/or y are part of a run.

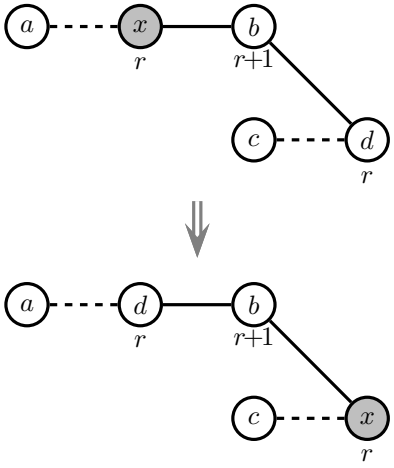



p
r+1
b
q
r
f
a
c
x
r+1
d
e
y
r

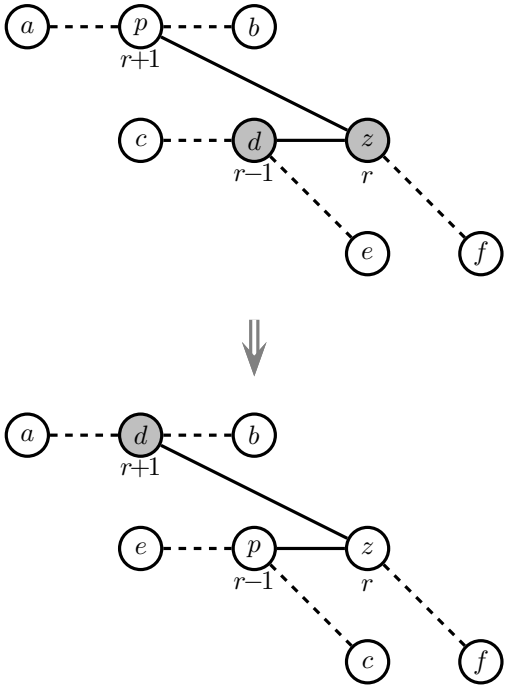
Singleton transformation II: y is the last child of its parent q and q is the right sibling of x. (The case where the parent of x is the right sibling of y is symmetric.) Assume that $element[x] \leq element[y]$. (The case where $element[x] > element[y]$ is similar.)



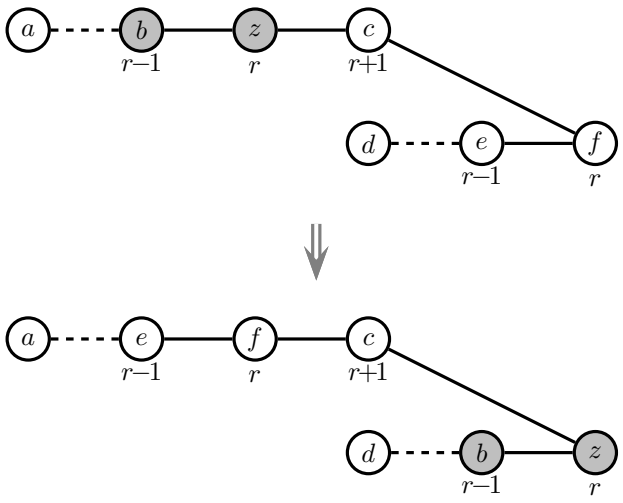
Singleton transformation III: x (or y) is not the last child of its parent, and the last child of its right sibling is not active.



Run transformation I: z is the last child of its parent. Assume that $element[d] \leq element[p]$ and $element[d] \leq element[z]$. (There are three other cases which are similar.)



Run transformation II: z is not the last child of its parent. For the right sibling of z, one or both of its two largest children may also be active.



References

1. Brodal, G.S.: Worst-case efficient priority queues. In: Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 52–58. ACM/SIAM, New York/Philadelphia (1996)
2. Brodal, G.S., Okasaki, C.: Optimal purely functional priority queues. Journal of Functional Programming **6**(6), 839–857 (1996)
3. Clancy, M.J., Knuth, D.E.: A programming and problem-solving seminar. Technical Report STAN-CS-77-606, Department of Computer Science, Stanford University (1977)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. The MIT Press, Cambridge (2001)
5. Driscoll, J.R., Gabow, H.N., Shrairman, R., Tarjan, R.E.: Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. Communications of the ACM **31**(11), 1343–1354 (1988)
6. Elmasry, A., Jensen, C., Katajainen, J.: A framework for speeding up priority-queue operations. CPH STL Report 2004-3, Department of Computing, University of Copenhagen (2004). Available at `http://cphstl.dk`
7. Elmasry, A., Jensen, C., Katajainen, J.: Two new methods for transforming priority queues into double-ended priority queues. CPH STL Report 2006-9, Department of Computing, University of Copenhagen (2006). Available at `http://cphstl.dk`
8. Elmasry, A., Jensen, C., Katajainen, J.: Two-tier relaxed heaps. In: Proceedings of the 17th International Symposium on Algorithms and Computation, *Lecture Notes in Computer Science*, vol. 4288, pp. 308–317. Springer-Verlag, Berlin/Heidelberg (2006)
9. Elmasry, A., Jensen, C., Katajainen, J.: Multipartite priority queues. ACM Transactions on Algorithms (to appear)
10. Fredman, M.L.: On the efficiency of pairing heaps and related data structures. Journal of the ACM **46**(4), 473–501 (1999)
11. Fredman, M.L., Sedgewick, R., Sleator, D.D., Tarjan, R.E.: The pairing heap: A new form of self-adjusting heap. Algorithmica **1**(1), 111–129 (1986)
12. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM **34**(3), 596–615 (1987)
13. Gonnet, G.H., Munro, J.I.: Heaps on heaps. SIAM Journal on Computing **15**(4), 964–971 (1986)
14. Goodrich, M.T., Tamassia, R.: Algorithm Design: Foundations, Analysis, and Internet Examples. John Wiley & Sons, Inc., New York (2002)
15. Kaplan, H., Shafrir, N., Tarjan, R.E.: Meldable heaps and Boolean union-find. In: Proceedings of the 34th Annual ACM Symposium on Theory of Computing, pp. 573–582. ACM, New York (2002)
16. Kaplan, H., Tarjan, R.E.: New heap data structures. Technical Report TR-597-99, Department of Computer Science, Princeton University (1999)
17. Kernighan, B.W., Ritchie, D.M.: The C Programming Language, 2nd edn. Prentice Hall PTR, Englewood Cliffs (1988)
18. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press, Cambridge (1998)
19. Overmars, M.H., van Leeuwen, J.: Worst-case optimal insertion and deletion methods for decomposable searching problems. Information Processing Letters **12**(4), 168–173 (1981)
20. Vuillemin, J.: A data structure for manipulating priority queues. Communications of the ACM **21**(4), 309–315 (1978)
21. Williams, J.W.J.: Algorithm 232: Heapsort. Communications of the ACM **7**(6), 347–348 (1964)

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

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).

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, $\ldots$, 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

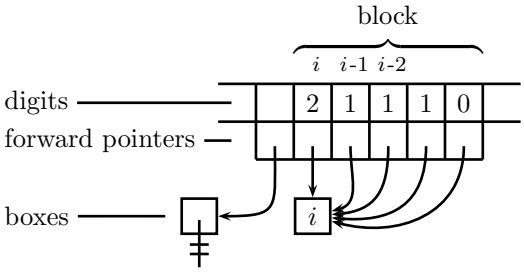


Figure 1: Illustration of a guide.

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, $\ldots$, 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 $1 + 2 + 3 + ... + h \leq j + 1$. Clearly, $h = \Theta(\sqrt{j})$. In order to maximize k, the number of nodes covered by missing subtrees culminates to $\sum_{i=1}^{h} i 2^{k-i+1} = 2^{k+2} - h2^{k-h+1} - 2^{k-h+2}$. 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 $h2^{k-h+1} + 2^{k-h+2} \leq n$. 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, ..., 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. □

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

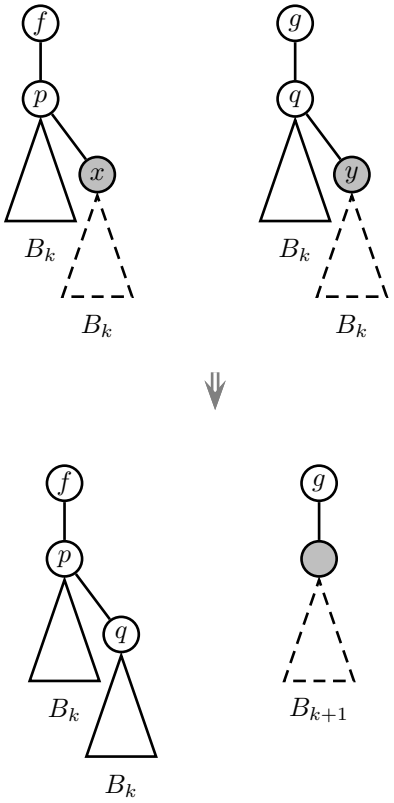
We gave two priority queues that support *decrease* and rely on structural violations. For the first priority queue, we allow structural violations in a straightforward manner. This priority queue achieves the worst-case bound of $\lg n + O(\sqrt{\lg n})$ element comparisons per deletion. For the second 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.

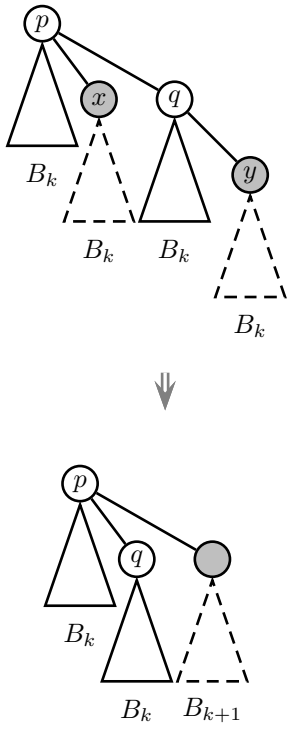
Appendix

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.

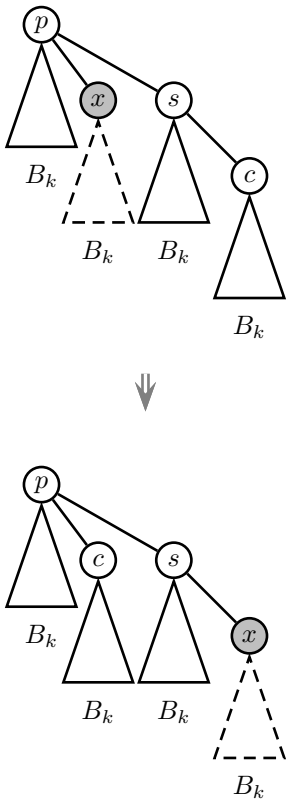
Singleton transformation I Both x and y are the last children of their parents p and q, respectively. Name the nodes such that *element*$[p] \not> $ *element*$[q]$. Observe that this transformation works even if x and/or y are part of a run.



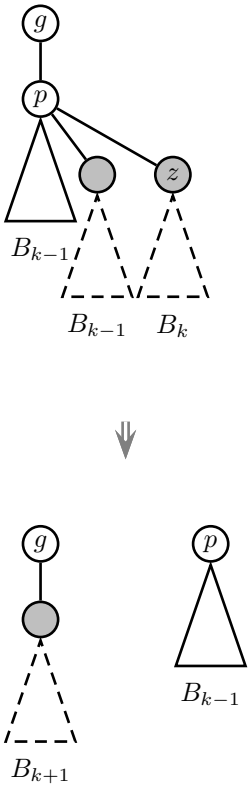
Singleton transformation II The parent of y is the right sibling of x, and y is the last child of its parent.



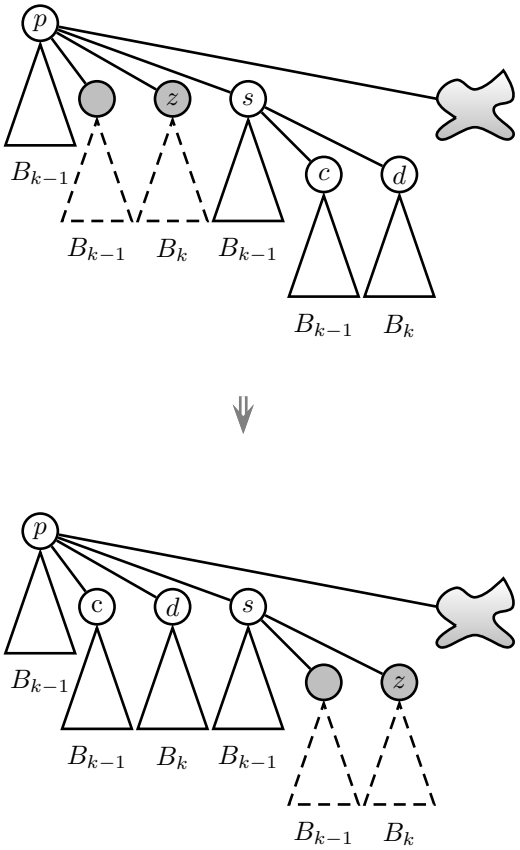
Singleton transformation III The given node x is not the last child of its parent and the last child of the right sibling of x is not a phantom node.



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.

A note on meldable heaps relying on data-structural bootstrapping*

Claus Jensen

The Royal Library, Postbox 2149, 1016 Copenhagen K, Denmark

Abstract. We introduce a meldable heap which guarantees the worst-case cost of $O(1)$ for *find-min*, *insert*, and *meld* with at most 0, 3, and 3 element comparisons for the respective operations; and the worst-case cost of $O(\lg n)$ with at most $3 \lg n + O(1)$ element comparisons for *delete*. Our data structure is asymptotically optimal and nearly constant-factor optimal with respect to the comparison complexity of all the meldable-heap operations. Furthermore, the data structure is also simple and elegant.

1. Introduction

A (min-)heap is a data structure which stores a collection of elements and supports the following operations:

find-min(Q): Return the reference of the node containing a minimum element held in heap Q.

insert(Q, p): Insert the node (storing an element) pointed to by p into heap Q.

delete(p): Remove the node pointed to by p from the heap in which it resides.

meld(Q_1, Q_2): Move the nodes from one heap into the other and return a reference to that heap.

It is easy to see that the deletion of the minimum element can be accomplished by invoking *find-min*, followed by *delete* given the reference returned by *find-min*. Throughout this paper we use m and n to denote the number of elements stored in a data structure prior to an operation and $\lg n$ as a shorthand for $\log_2(\max\{2, n\})$.

As our model of computation, we use the *word RAM*. It would be possible, with some modifications, to realize our data structure on a pointer machine. However, when using a word-RAM model the data structure and the operations are simpler and more elegant. The term *cost* is used to denote the sum of instructions and element comparisons performed.

We want to obtain a data structure where the operations *find-min*, *insert*, and *meld* have worst-case constant cost, and using this as a base we want to minimize the comparison complexity of *delete*. In the amortized sense a cost

*Partially supported by the Danish Natural Science Research Council under contract 09-060411 (project Generic programming—algorithms and tools).

CPH STL Report 2009-2, March 2009. Revised May 2011.

Table 1. The worst-case comparison complexity of heap operations for selected data structures. For all structures *find-min*, *insert*, and *meld* have $O(1)$ worst-case cost. Observe that the constant factors are not proved in some of the original sources, but are derived by us.

Source	*delete-min*	*delete*
[1]	$7 \lg n + O(1)$	$7 \lg n + O(1)$
[2]	$4 \lg n + O(1)$	not supported
this paper	$3 \lg n + O(1)$	$3 \lg n + O(1)$
[9]	$2.5 \lg n + O(1)$	$2.5 \lg n + O(1)$
[8]	$2 \lg n + O(1)$	$2 \lg n + O(1)$

of $O(1)$ for *find-min*, *insert*, and *meld*; and a cost of $O(\lg n)$ for *delete* is achieved by binomial heaps[1], Fibonacci heaps [10], thin heaps [12], and thick heaps (one-step heaps) [12, 11]. The same worst-case bounds are achieved by the meldable heaps described in [1], [2], and [8], see Table 1.

Given the comparison-based lower bound for sorting, it follows that *delete* has to perform at least $\lg n - O(1)$ element comparisons, if *find-min* and *insert* only perform $O(1)$ element comparisons. Furthermore, in [1] it is shown that if *meld* can be performed at the worst-case cost of $o(n)$ then *delete* cannot be performed at the worst-case cost of $o(\lg n)$.

In this paper we present a meldable heap which is a binomial heap transformed using data-structural bootstrapping. We obtain an good bound for the worst-case comparison complexity of *delete* given that the heap operations *find-min*, *insert*, and *meld* have a worst-case constant cost. Observe that there exist other data structures [8, 9] that obtain slightly better worst-case comparison complexity for *delete*. However, these data structures are advanced and mainly of theoretical interest. We reuse some of the techniques described in [2], [14], and [5], but our focus is on good constant factors and a simple data structure. In Section 2, we describe a modified binomial heap that supports *insert* at the worst-case cost of $O(1)$ and *meld* at the worst-case cost of $O(\lg n)$. In Section 3, we describe how to transform a binomial heap supporting *meld* at the worst-case cost of $O(\lg n)$ into a heap that supports melding at the worst-case cost of $O(1)$ and we prove the worst-case cost of $O(1)$ for *find-min*, *insert*, and *meld*; and the worst-case cost of $O(\lg n)$ with at most $3 \lg n + O(1)$ element comparisons for *delete*. In Section 4, we conclude the paper with some final remarks.

2. Binomial heaps

In this section, we give a description of binomial heaps. However, be aware that the binomial heaps described here are different from the standard binomial heaps described in most textbooks.

A *binomial tree* [15] is an ordered tree defined recursively as follows: A binomial tree of rank 0 is a single node; for ranks higher than 0, a binomial

[1] This result is folklore; it can be achieved by applying lazy melding to binomial heaps [15] as in Fibonacci heaps [10].

tree of rank r consists of the root and its r subtrees of rank 0, 1, ..., $r-1$ connected to the root in that order. Given a root of a tree, we denote the root of the subtree of rank 0 the *smallest child* and the root of the subtree of rank $r-1$ the *largest child*. The size of a binomial tree is always a power of two, and the rank of a tree of size 2^r is r.

A node in a binomial tree contains an element, a rank, a parent pointer, a child pointer, and two sibling pointers. The child pointer of each node points to its largest child if it exists. The children of a node are kept in a *child list*, which is implemented as a doubly-linked list using the sibling pointers. If a node is a root, the parent pointer points to the heap in which the node resides. Furthermore, the right-sibling pointer of a root points to the root itself.

The binomial trees used by us are *heap ordered* which means that the element stored at a node is no greater than the elements stored at the children of that node. If two heap-ordered binomial trees have the same rank, they can be linked together by making the root that stores the non-smaller element the largest child of the other root. We call this linking of trees a *join*. A join involves a single element comparison and has the worst-case cost of $O(1)$.

The *binomial heap* is a collection of a logarithmic number of binomial trees storing n elements. To obtain a strict bound on the number of trees τ, we maintain the invariant $\tau \leq \lfloor \lg n \rfloor + 1$. A similar approach is used for run-relaxed heaps [5].

Lemma 1. *In a binomial heap of size n, the rank of a tree can never be higher than $\lfloor \lg n \rfloor$.*

Proof. Let the highest rank be k. None of the trees can be larger than the size of the whole heap, i.e. $2^k \leq n$. Since k is an integer, $k \leq \lfloor \lg n \rfloor$. □ □

The roots of the trees in the binomial heap are maintained using a data structure, adopted from [5], which we call a *root store*. The central structure is a resizable array which supports growing and shrinking at the tail at worst-case constant cost; this can be obtained, for example, by doubling, halving, and incremental copying. Each entry in this resizable array corresponds to a rank. For each entry (rank) there exists a *rank list* (doubly-linked) containing pointers to the roots of trees having this rank. Each root uses its left-sibling pointer as a *back pointer* to its corresponding item in the rank list. For each entry of the resizable array that has more than one root, a pointer to this entry is kept in a *pair list* (doubly-linked). An entry of the resizable array contains a pointer to the beginning of a *rank list* and a pointer to an item in the pair list if one exist.

The root of a tree of arbitrary rank is inserted into the root store using *add*. This operation inserts an item pointing to the given root into the corresponding rank list, sets the back pointer of the root to point to that item, and updates the pair list if necessary. If $\tau > \lfloor \lg n \rfloor + 1$, *add* uses the pair list to find two trees of the same rank which it removes from the rank list. It then performs a join of the two trees followed by an insert of the

new tree into the corresponding rank list, after which it updates the pair list if necessary. A tree is removed from the root store using *subtract*, which removes the item in the rank list referred to by the back pointer of the given root and updates the pair list if necessary. Both *add* and *subtract* have the worst-case cost of $O(1)$. In connection with each *add* it may be necessary to perform a single element comparison.

In the following, the binomial-heap operations are described and their worst-case bounds are analysed.

The candidates for a minimum element are stored at the roots of the binomial trees. A fast *find-min* is obtained by maintaining a *minimum pointer* to the node containing a minimum element. Thus *find-min* can be accomplished at the worst-case cost of $O(1)$ using no element comparisons.

When inserting an element, the node containing the element is treated as a tree with rank 0 and added to the root store using *add*. The element of the inserted node and the element of the node pointed to by the minimum pointer are compared and the minimum pointer is updated if necessary. From our earlier analysis, *add* has the worst-case cost of $O(1)$ and at most one element comparison is performed. Checking the minimum pointer has the worst-case cost of $O(1)$ and at most one element comparison is performed. Therefore *insert* has the the worst-case cost of $O(1)$ and at most two element comparisons are performed.

A *delete* is performed in the following way. A root is deleted by subtracting it from the root store, unlinking its subroots and adding them to the root store. If the node to be deleted is not a root, the node is swapped with its parent until it becomes a root, the old root is subtracted from the root store, and the subroots of the new root are unlinked and added to the root store. Furthermore, a scan of all roots is performed and the minimum pointer is updated if necessary. The operation of swapping a node until it becomes a root has the worst-case cost of $O(\lg n)$. A *subtract* has the worst-case cost of $O(1)$. Both the unlinking and addition of a subroot have the worst-case cost of $O(1)$ and at most one element comparison is performed in connection with each addition. By Lemma 1, a root can have at most $\lg n$ subroots which means that at most $\lg n$ element comparisons are performed when adding the roots to the root store. A scan of all roots has the worst-case cost of $O(\lg n)$ and requires at most $\lg n$ element comparisons. To sum up, each *delete* has the worst-case cost of $O(\lg n)$ and requires at most $2 \lg n + O(1)$ element comparisons.

In *meld*, two binomial heaps Q_1 and Q_2 are to be melded. Without loss of generality, we assume that the number of trees in Q_1 is smaller than or equal to that in Q_2. The root store of Q_1 is emptied using *subtract*; after each *subtract* the removed tree is added to the root store of Q_2 using *add*. The elements of the nodes pointed to by the minimum pointer of Q_1 and Q_2 are compared and the minimum pointer of Q_2 is updated if necessary. When all trees have been removed from Q_1, the root store of Q_1 is released. When the respective sizes of Q_1 and Q_2 are m and n, the number of trees moved is at most $\min\{\lg m, \lg n\} + 1$. In connection with every *add* at most one element

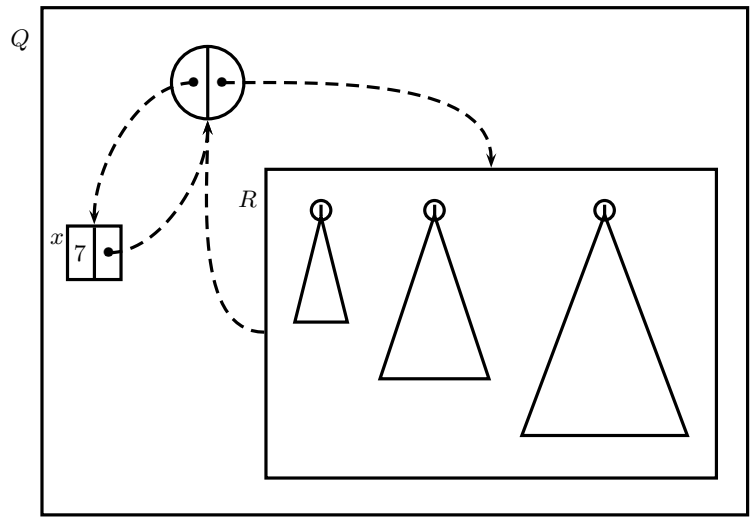


Figure 1. A simplified view of the transformed heap.

comparison is performed, and both *add* and *subtract* have the worst-case cost of $O(1)$. Finding the minimum pointer has the worst-case cost of $O(1)$ and at most one element comparison is performed. Therefore, the worst-case cost of *meld* is $O(\min\{\lg m, \lg n\})$ and at most $\min\{\lg m, \lg n\} + O(1)$ element comparisons are performed.

3. Transformation of heaps

We now describe how the binomial heap can be modified to support the *meld* operation at a constant cost using a transformation called *data-structural bootstrapping* [3, 4, 2, 14]. We use the structural abstraction technique of data-structural bootstrapping which bootstrap efficient data structures from less efficient data structures. Observe that in the heap context bootstrapping results in a structure where heaps contain heaps.

The transformed heap is represented by a binomial heap (as described in Section 2) containing one node. This node stores a pair which contains a reference to a node storing an element and a reference to a heap storing pairs. There is a back reference from the element node to the pair and from the heap to the pair. The back reference indicates that the pair is the *owner* of the element node and the heap. The element of a pair is no greater than any element stored within the heap of that pair. Furthermore, the order of the pairs within a heap is based on the elements associated with the pairs.

In the following the heap operations are described; Figure 2 describes the

```
find-min(Q):
  if Q = ∅
    return nil
  (x, R) ← minimum[Q]
  return x

insert(Q, x):
  p ← construct pair
  if Q = ∅
    p ← (x, nil)
    owner[x] ← p
    insert(Q, p)
    minimum[Q] ← (x, nil)
    return
  (y, R) ← q ← minimum[Q]
  if R = nil
    R ← construct heap
    owner[R] ← (y, R)
  if element[x] < element[y]
    p ← (y, nil)
    owner[y] ← p
    insert(R, p)
    owner[x] ← q
    minimum[Q] ← (x, R)
  else
    p ← (x, nil)
    owner[x] ← p
    insert(R, p)
    minimum[Q] ← (y, R)

delete(x):
  (x, R) ← p ← owner[x]
  Q ← parent[root(p)]
  if owner[Q] = nil and R = ∅
    delete(p)
    owner[x] ← nil
  elseif owner[Q] = nil and R ≠ ∅
    (z, S) ← q ← find-min(R)
    delete(q)
    T ← meld(R, S)
    p ← (z, T)
    owner[z] ← p
    owner[T] ← p
  else
    delete(p)
    (z, Q) ← q ← owner[Q]
    T ← meld(Q, R)
    q ← (z, T)
    owner[T] ← q

meld(Q, R):
  if Q = ∅
    return R
  if R = ∅
    return Q
  (x, S) ← minimum[Q]
  (y, T) ← minimum[R]
  if element[x] < element[y]
    p ← (y, T)
    insert(S, p)
    minimum[Q] ← (x, S)
  else
    p ← (x, S)
    insert(T, p)
    minimum[Q] ← (y, T)
  return Q
```

Figure 2. This pseudo code implements our heap operations. We use property maps in our interaction with objects, so an attribute is accessed using the attribute name followed by the name of the object in square brackets. The *construct* operation creates a new object. Given a node, the *root* operation returns the root of the tree in which the given node resides. Observe that we rely on automatic garbage collection.

same operations using pseudo code.

Let Q refer to the transformed heap given to the user. A minimum element is found within the pair (x, R) of Q where the element referenced by x is no greater than any element within R and thereby within the whole transformed heap. To reach x, a constant number of pointers has to be followed and therefore *find-min*(Q) can be accomplished at the worst-case cost of $O(1)$ using no element comparisons.

In *insert*(Q, x), if the element referred to by x is smaller than the current minimum element referred to by y in the pair (y, R) of Q, x replaces y in this

pair and a new pair containing y and a reference to an empty heap is inserted into R. Otherwise, a pair containing x and an empty heap is inserted into R. On the basis of the above description and our earlier analysis, *insert*(Q, x) has the worst-case cost of $O(1)$ and at most three element comparisons are performed, one when checking for a possible new minimum and two when inserting a pair into R.

In *delete*(x), let x be a reference to the element to be deleted, (x, R) the pair containing x, and Q the heap containing this pair. Let us next consider the three cases of *delete*.

Case 1: Q is not owned by a pair and R is empty. The pair (x, R) is destroyed.

Case 2: Q is not owned by a pair and R is non-empty. A *find-min*(R) is performed. Let (z, S) be the pair returned by *find-min*(R). Now this pair is deleted from the heap R, the heaps R and S are melded using *meld*(R, S), after which z and a reference to the melded heap forms the new pair replacing the pair (x, R).

Case 3: Q is owned by a pair. Let the pair owning Q be (z, Q). The pair (x, R) is deleted from Q, the heaps Q and R are melded using *meld*(Q, R), after which z and a reference to the melded heap forms the new pair replacing the pair (z, Q).

The cost of *delete*(x) is dominated by the cost of the binomial-heap operations *find-min*, *delete*, and *meld*. Therefore, *delete*(x) has the worst-case cost of $O(\lg n)$ and the number of element comparisons performed is $3 \lg n + O(1)$, of which $O(1)$ are performed by *find-min*, $2 \lg n + O(1)$ by *delete*, and $\lg n + O(1)$ by *meld*.

In *meld*(Q, R), the new minimum element is found by comparing the (minimum) element of the two pairs representing the two bootstrapped heaps. The pair referring to the non-smaller element is inserted into the heap of the other pair. The *meld*(Q, R) operation has the worst-case cost of $O(1)$ and at most three element comparisons are performed, one when determining the new minimum and two during *insert*.

The following theorem summarizes the main result of the paper.

Theorem 1. *Let n be the number of elements in the heap prior to each operation. Our heap guarantees the worst-case cost of $O(1)$ for find-min, insert, and meld with at most 0, 3, and 3 element comparisons for the respective operations; and the worst-case cost of $O(\lg n)$ with at most $3 \lg n + O(1)$ element comparisons for delete.*

4. Concluding remarks

We have described how data-structural bootstrapping can be used to transform a binomial heap supporting the *meld* operation at the worst-case cost of $O(\lg n)$ into a heap that supports melding at the worst-case cost of $O(1)$, and with a *delete* operation having the worst-case cost of $O(\lg n)$ and performing at most $3 \lg n + O(1)$ element comparisons. Binomial heaps have

only been used as an example and other data structures like weak queues [6] and navigation piles [13] could, with some modifications, have been used instead. Other implementations of binomial heaps could also have been used; for example, the binomial heap described in [7]. To obtain the same comparison bounds as our heap implementation using other data structures as the basic component, it is important that the data structure used can perform the three operations *find-min*, *delete*, and *meld* using at most $3 \lg n + O(1)$ element comparisons in total, as these operations are the essential part of the *delete* operation in the bootstrapped heap.

Looking at the comparison complexity of heap operations, the following open questions still remain.

1. Is it possible to achieve a bound of $\lg n + O(1)$ element comparisons per *delete* when *meld* is required to have a constant cost? Note that, if *meld* is allowed to have a logarithmic cost, the worst-case bound of $\lg n + O(1)$ is achievable using the approach described in [7].
2. Given that *decrease* is added to the collection of operations and defined as follows:

   decrease(p, v): Replace the element stored at the node pointed to by p with element v, given that the new element is no greater than the old element.

   What would be the lowest possible number of element comparisons performed by *delete* Assuming that all other operations were required to have the worst-case cost of $O(1)$?

References

[1] G. S. Brodal, Fast meldable priority queues, *Proceedings of the 4th International Workshop on Algorithms and Data Structures*, Springer-Verlag (1995), 282–290.

[2] G. S. Brodal and C. Okasaki, Optimal purely functional priority queues, *Journal of Functional Programming* **6**, 6 (1996), 839–857.

[3] A. L. Buchsbaum, Data-structural bootstrapping and catenable deques, Ph. D. Thesis, Department of Computer Science, Princeton University, Princeton (1993).

[4] A. L. Buchsbaum, R. Sundar, and R. E. Tarjan, Data structural bootstrapping, linear path compression, and catenable heap ordered double ended queues, *SIAM Journal on Computing* **24**, 6 (1995), 1190–1206.

[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**, 11 (1988), 1343–1354.

[6] A. Elmasry, C. Jensen, and J. Katajainen, Relaxed weak queues: An alternative to run-relaxed heaps, CPH STL Report 2005-2, Department of Computer Science, University of Copenhagen (2005). (Available at `http://cphstl.dk`.)

[7] A. Elmasry, C. Jensen, and J. Katajainen, Multipartite priority queues, *ACM Transactions on Algorithms* **5**, 1 (2008), Acticle 14.

[8] A. Elmasry, C. Jensen, and J. Katajainen, Strictly-regular number system and data structures, *Proceedings of the 12th Scandinavian Workshop on Algorithm Theory*, *Lecture Notes in Computer Science* **6139**, Springer-Verlag (2010), 26–37.

[9] A. Elmasry, C. Jensen, and J. Katajainen, Fast meldable heaps via data-structural transformations, (Submitted for publication.) (2011)

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

[11] P. Høyer, A general technique for implemention of efficient priority queues, *Proceedings of the 3rd Israel Symposium on the Theory of Computing Systems*, IEEE (1995), 57–66.
[12] H. Kaplan and R. E. Tarjan, Thin heaps, thick heaps, *ACM Transactions on Algorithms* **4**, 1 (2008), Acticle 3.
[13] J. Katajainen and F. Vitale, Navigation piles with applications to sorting, priority queues, and priority deques, *Nordic Journal of Computing* **10** (2003), 238–262.
[14] C. Okasaki, *Purely Functional Data Structures*, Cambridge University Press (1998).
[15] J. Vuillemin, A data structure for manipulating priority queues, *Communications of the ACM* **21**, 4 (1978), 309–315.

Strictly-Regular Number System and Data Structures*

Amr Elmasry[1], Claus Jensen[2], and Jyrki Katajainen[3]

[1] Max-Planck Institut für Informatik, Saarbrücken, Germany
[2] The Royal Library, Copenhagen, Denmark
[3] Department of Computer Science, University of Copenhagen, Denmark

Abstract. We introduce a new number system that we call the strictly-regular system, which efficiently supports the operations: digit-increment, digit-decrement, cut, concatenate, and add. Compared to other number systems, the strictly-regular system has distinguishable properties. It is superior to the regular system for its efficient support to decrements, and superior to the extended-regular system for being more compact by using three symbols instead of four. To demonstrate the applicability of the new number system, we modify Brodal's meldable priority queues making deletion require at most $2 \lg n + O(1)$ element comparisons (improving the bound from $7 \lg n + O(1)$) while maintaining the efficiency and the asymptotic time bounds for all operations.

1 Introduction

Number systems are powerful tools of the trade when designing worst-case-efficient data structures. As far as we know, their usage was first discussed in the seminar notes by Clancy and Knuth [1]. Early examples of data structures relying on number systems include finger search trees [2] and binomial queues [3]. For a survey, see [4, Chapter 9]. The problem with the normal binary number representation is that a single increment or decrement may change all the digits in the original representation. In the corresponding data structure, this may give rise to many changes that would result in weak worst-case performance.

The characteristics of a positional number system $\mathcal{N}$ are determined by the constraints imposed on the digits and the weights corresponding to them. Let $rep(d, \mathcal{N}) = \langle d_0, d_1, \ldots, d_{r-1} \rangle$ be the sequence of digits representing a positive integer d in $\mathcal{N}$. (An empty sequence can be used to represent zero.) By convention, d_0 is the least-significant digit and $d_{r-1} \neq 0$ is the most-significant digit.

The work of the authors was partially supported by the Danish Natural Science Research Council under contract 09-060411 (project "Generic programming—algorithms and tools"). A. Elmasry was supported by the Alexander von Humboldt Foundation and the VELUX Foundation.

The value of d in $\mathcal{N}$ is $val(d,\mathcal{N}) = \sum_{i=0}^{r-1} d_i \cdot w_i$, where w_i is the weight corresponding to d_i. As a shorthand, we write $rep(d)$ for $rep(d,\mathcal{N})$ and $val(d)$ for $val(d,\mathcal{N})$. In a redundant number system, it is possible to have $val(d) = val(d')$ while $rep(d) \neq rep(d')$. In a b-ary number system, $w_i = b^i$.

A sequence of digits is said to be *valid* in $\mathcal{N}$ if all the constraints imposed by $\mathcal{N}$ are satisfied. Let d and d' be two numbers where $rep(d) = \langle d_0, d_1, \ldots, d_{r-1}\rangle$ and $rep(d') = \langle d'_0, d'_1, \ldots, d'_{r'-1}\rangle$ are valid. The following operations are defined.

increment(d,i)**:** Assert that $i \in \{0,1,\ldots,r\}$. Perform $\texttt{++}d_i$ resulting in d', i.e. $val(d') = val(d) + w_i$. Make d' valid without changing its value.

decrement(d,i)**:** Assert that $i \in \{0,1,\ldots,r-1\}$. Perform $\texttt{--}d_i$ resulting in d', i.e. $val(d') = val(d) - w_i$. Make d' valid without changing its value.

cut(d,i)**:** Cut $rep(d)$ into two valid sequences having the same value as the numbers corresponding to $\langle d_0, d_1, \ldots, d_{i-1}\rangle$ and $\langle d_i, d_{i+1}, \ldots, d_{r-1}\rangle$.

concatenate(d,d')**:** Concatenate $rep(d)$ and $rep(d')$ into one valid sequence that has the same value as $\langle d_0, d_1, \ldots, d_{r-1}, d'_0, d'_1, \ldots, d'_{r'-1}\rangle$.

add(d,d')**:** Construct a valid sequence d'' such that $val(d'') = val(d) + val(d')$.

One should think that a corresponding data structure contains d_i components of rank i, where the meaning of rank is application specific. A component of rank i has size $s_i \leq w_i$. If $s_i = w_i$, we see the component as perfect. In general, the size of a structure corresponding to a sequence of digits need not be unique.

The regular system [1], called the segmented system in [4], comprises the digits $\{0,1,2\}$ with the constraint that every 2 is preceded by a 0 possibly having any number of 1's in between. Using the syntax for regular expressions (see, for example, [5, Section 3.3]), every regular sequence is of the form $\big(0 \mid 1 \mid 01^*2\big)^*$. The regular system allows for the increment of any digit with $O(1)$ digit changes [1, 6], a fact that can be used to modify binomial queues to accomplish *insert* at $O(1)$ worst-case cost. Brodal [7] used a zeroless variant of the regular system, comprising the digits $\{1,2,3\}$, to ensure that the sizes of his trees are exponential with respect to their ranks. For further examples of structures that use the regular system, see [8, 9]. To be able to perform decrements with $O(1)$ digit changes, an extension was proposed in [1, 6]. Such an extended-regular system comprises the digits $\{0,1,2,3\}$ with the constraint that every 3 is preceded by a 0 or 1 possibly having any number of 2's in between, and that every 0 is preceded by a 2 or 3 possibly having any number of 1's in between. For examples of structures that use the extended-regular system, see [6, 10, 11].

In this paper, we introduce a number system that we call the strictly-regular system. It uses the digits $\{0,1,2\}$ and allows for both increments and decrements with $O(1)$ digit changes. The strictly-regular system contains less redundancy and is more compact, achieving better constant factors while supporting a larger repertoire of operations. We expect the new system to be useful in several other contexts in addition to the applications we mention here.

Utilizing the strictly-regular system, we introduce the strictly-regular trees. Such trees provide efficient support for adding a new subtree to the root, detaching an existing one, cutting and concatenating lists of children. We show that

Table 1: Known results on the worst-case comparison complexity of priority-queue operations when *decrease* is not considered and *find-min* has $O(1)$ cost. Here n and m denote the sizes of priority queues.

Source	*insert*	*delete*	*meld*
[12]	$O(1)$	$\lg n + O(1)$	–
[11]	$O(1)$	$\lg n + O(\lg \lg n)$	$O(\lg(\min\{n, m\}))$
[7] (see Section 3.1)	$O(1)$	$7 \lg n + O(1)$	$O(1)$
[13]	$O(1)$	$3 \lg n + O(1)$	$O(1)$
this paper	$O(1)$	$2 \lg n + O(1)$	$O(1)$

the number of children of any node in a strictly-regular tree is bounded by $\lg n$, where n is the number of descendants of such node.

A *priority queue* is a fundamental data structure which stores a dynamic collection of elements and efficiently supports the operations *find-min*, *insert*, and *delete*. A *meldable* priority queue also supports the operation *meld* efficiently. As a principal application of our number system, we implement an efficient meldable priority queue. Our best upper bound is $2 \lg n + O(1)$ element comparisons per *delete*, which is achieved by modifying the priority queue described in [7]. Table 1 summarizes the related known results.

The paper is organized as follows. We introduce the number system in Section 2, study the application to meldable priority queues in Section 3, and discuss the applicability of the number system to other data structures in Section 4.

2 The Number System

Similar to the redundant binary system, in our system any digit d_i must be 0, 1, or 2. We call 0 and 2 *extreme digits*. We say that the representation is *strictly regular* if the sequence from the least-significant to the most-significant digit is of the form $\left(1^+ \mid 01^*2\right)^* \left(\varepsilon \mid 01^+\right)$. In other words, such a sequence is a combination of zero or more interleaved 1^+ and 01^*2 blocks, which may be followed by at most one 01^+ block. We use $w_i = 2^i$, implying that the weighted value of a 2 at position i is equivalent to that of a 1 at position $i + 1$.

2.1 Properties

An important property that distinguishes our number system from other systems is what we call the *compactness* property, which is defined in the next lemma.

Lemma 1. *For any strictly-regular sequence,* $\sum_{i=0}^{r-1} d_i$ *is either* $r - 1$ *or* r.

Proof. The sum of the digits in a 01^*2 block or a 1^* block equals the number of digits in the block, and the sum of the digits in the possibly trailing 01^+ block is one less than the number of digits in that block. □

Note that the sum of digits $\sum_{i=0}^{r-1} d_i$ for a positive integer in the regular system is between 1 and r; in the zeroless system, where $d_i \in \{1, 2, \ldots h\}$, the sum of digits is between r and $h \cdot r$; and in the zeroless regular representation, where $d_i \in \{1, 2, 3\}$ [7], the sum of digits is between r and $2r$.

An important property, essential for designing data structures with exponential size in terms of their rank, is what we call the *exponentiality* property. Assume $s_i \geq \theta^i/c$ and $s_0 = 1$, for fixed real constants $\theta > 1$ and $c > 0$. A number system has such property if for each valid sequence $\sum_{i=0}^{r-1} d_i \cdot s_i \geq \theta^r/c - 1$ holds.

Lemma 2. *For the strictly-regular system, the exponentiality property holds by setting $\theta = c = \Phi$, where Φ is the golden ratio.*

Proof. Consider a sequence of digits in a strictly-regular representation, and think about $d_i = 2$ as two 1's at position i. It is straightforward to verify that there exists a distinct 1 whose position is at least i, for every i from 0 to $r - 2$. In other words, we have $\sum_{i=0}^{r-1} d_i \cdot s_i \geq \sum_{i=0}^{r-2} s_i$. Substituting with $s_i \geq \Phi^{i-1}$ and $s_0 = 1$, we obtain $\sum_{i=0}^{r-1} d_i \cdot s_i \geq 1 + \sum_{i=0}^{r-3} \Phi^i \geq \Phi^{r-1} - 1$. □

The exponentiality property holds for any zeroless system by setting $\theta = 2$ and $c = 1$. The property also holds for any θ when $d_{r-1} \geq \theta$; this idea was used in [8], by imposing $d_{r-1} \geq 2$, to ensure that the size of a tree of rank r is at least 2^r. On the other hand, the property does not hold for the regular system.

2.2 Operations

It is convenient to use the following subroutines that change two digits but not the value of the underlying number.

fix-carry(d, i)**:** Assert that $d_i \geq 2$. Perform $d_i \leftarrow d_i - 2$ and $d_{i+1} \leftarrow d_{i+1} + 1$.
fix-borrow(d, i)**:** Assert that $d_i \leq 1$. Perform $d_{i+1} \leftarrow d_{i+1} - 1$ and $d_i \leftarrow d_i + 2$.

Temporarily, a digit can become a 3 due to $\texttt{++}d_i$ or *fix-borrow*, but we always eliminate such a violation before completing the operations. We demonstrate in Algorithm *increment* (*decrement*) how to implement the operation in question with at most one *fix-carry* (*fix-borrow*), which implies Theorem 1. The correctness of the algorithms follows from the case analysis of Table 2.

Theorem 1. *Given a strictly-regular representation of d, increment(d, i) and decrement(d, i) incur at most three digit changes.*

Algorithm *increment*(d, i)

1: $\texttt{++}d_i$
2: Let d_b be the first extreme digit before d_i, $d_b \in \{0, 2, \textit{undefined}\}$
3: Let d_a be the first extreme digit after d_i, $d_a \in \{0, 2, \textit{undefined}\}$
4: **if** $d_i = 3$ **or** $(d_i = 2$ **and** $d_b \neq 0)$
5: *fix-carry*(d, i)
6: **else if** $d_a = 2$
7: *fix-carry*(d, a)

Algorithm *decrement*(d, i)

1: Let d_b be the first extreme digit before d_i, $d_b \in \{0, 2, \textit{undefined}\}$
2: Let d_a be the first extreme digit after d_i, $d_a \in \{0, 2, \textit{undefined}\}$
3: **if** $d_i = 0$ **or** ($d_i = 1$ **and** $d_b = 0$ **and** $i \neq r - 1$)
4: *fix-borrow*(d, i)
5: **else if** $d_a = 0$
6: *fix-borrow*(d, a)
7: $--d_i$

By maintaining pointers to all extreme digits in a circular doubly-linked list, the extreme digits are readily available when increments and decrements are carried out at either end of a sequence.

Corollary 1. *Let $\langle d_0, d_1, \ldots, d_{r-1}\rangle$ be a strictly-regular representation of d. If such sequence is implemented as two circular doubly-linked lists, one storing all the digits and another all extreme digits, any of the operations increment$(d, 0)$, increment$(d, r-1)$, increment(d, r), decrement$(d, 0)$, and decrement$(d, r-1)$ can be executed at $O(1)$ worst-case cost.*

Theorem 2. *Let $\langle d_0, d_1, \ldots, d_{r-1}\rangle$ and $\langle d'_0, d'_1, \ldots, d'_{r'-1}\rangle$ be strictly-regular representations of d and d'. The operations cut(d, i) and concatenate(d, d') can be executed with $O(1)$ digit changes. Assuming without loss of generality that $r \leq r'$, add(d, d') can be executed at $O(r)$ worst-case cost including at most r carries.*

Proof. Consider the two sequences resulting from a cut. The first sequence is strictly regular and requires no changes. The second sequence may have a preceding 1^*2 block followed by a strictly-regular subsequence. In such case, we perform a *fix-carry* on the 2 ending such block to reestablish strict regularity. A catenation requires a fix only if *rep*(d) ends with a 01^+ block and *rep*(d') is not equal to 1^+. In such case, we perform a *fix-borrow* on the first 0 of *rep*(d'). An addition is implemented by adding the digits of one sequence to the other starting from the least-significant digit, simultaneously updating the pointers to the extreme digits in the other sequence, while maintaining strict regularity. Since each increment propagates at most one *fix-carry*, the bounds follow. □

2.3 Strictly-Regular Trees

We recursively define a *strictly-regular tree* such that every subtree is as well a strictly-regular tree. For every node x in such a tree

- the rank, in brief *rank*(x), is equal to the number of the children of x;
- the *cardinality sequence*, in which entry i records the number of children of rank i, is strictly regular.

The next lemma directly follows from the definitions and Lemma 1.

Table 2: d_i is displayed in bold. d_a is the first extreme digit after d_i, k is a positive integer, α denotes any combination of 1^+ and 01^*2 blocks, and ω any combination of 1^+ and 01^*2 blocks followed by at most one 01^+ block.

(a) Case analysis for *increment*(d, i).

Initial configuration	Action	Final configuration
$\alpha 01^*\mathbf{2}$	$d_i \leftarrow 3$; *fix-carry*(d, i)	$\alpha 01^*\mathbf{1}1$
$\alpha 01^*\mathbf{2}1^k\omega$	$d_i \leftarrow 3$; *fix-carry*(d, i)	$\alpha 01^*\mathbf{1}21^{k-1}\omega$
$\alpha 01^*\mathbf{2}01^*2\omega$	$d_i \leftarrow 3$; *fix-carry*(d, i)	$\alpha 01^*\mathbf{1}11^*2\omega$
$\alpha 01^*\mathbf{2}01^k$	$d_i \leftarrow 3$; *fix-carry*(d, i)	$\alpha 01^*\mathbf{1}11^k$
$\alpha\mathbf{1}$	$d_i \leftarrow 2$; *fix-carry*(d, i)	$\alpha\mathbf{0}1$
$\alpha\mathbf{1}1^k\omega$	$d_i \leftarrow 2$; *fix-carry*(d, i)	$\alpha\mathbf{0}21^{k-1}\omega$
$\alpha\mathbf{1}01^*2\omega$	$d_i \leftarrow 2$; *fix-carry*(d, i)	$\alpha\mathbf{0}11^*2\omega$
$\alpha\mathbf{1}01^k$	$d_i \leftarrow 2$; *fix-carry*(d, i)	$\alpha\mathbf{0}11^k$
$\alpha 01^*\mathbf{1}1^*2$	$d_i \leftarrow 2$; *fix-carry*(d, a)	$\alpha 01^*\mathbf{2}1^*01$
$\alpha 01^*\mathbf{1}1^*21^k\omega$	$d_i \leftarrow 2$; *fix-carry*(d, a)	$\alpha 01^*\mathbf{2}1^*021^{k-1}\omega$
$\alpha 01^*\mathbf{1}1^*201^*2\omega$	$d_i \leftarrow 2$; *fix-carry*(d, a)	$\alpha 01^*\mathbf{2}1^*011^*2\omega$
$\alpha 01^*\mathbf{1}1^*201^k$	$d_i \leftarrow 2$; *fix-carry*(d, a)	$\alpha 01^*\mathbf{2}1^*011^k$
$\alpha\mathbf{0}1^*2$	$d_i \leftarrow 1$; *fix-carry*(d, a)	$\alpha\mathbf{1}1^*01$
$\alpha\mathbf{0}1^*21^k\omega$	$d_i \leftarrow 1$; *fix-carry*(d, a)	$\alpha\mathbf{1}1^*021^{k-1}\omega$
$\alpha\mathbf{0}1^*201^*2\omega$	$d_i \leftarrow 1$; *fix-carry*(d, a)	$\alpha\mathbf{1}1^*011^*2\omega$
$\alpha\mathbf{0}1^*201^k$	$d_i \leftarrow 1$; *fix-carry*(d, a)	$\alpha\mathbf{1}1^*011^k$
$\alpha 01^*\mathbf{1}1^*$	$d_i \leftarrow 2$	$\alpha 01^*\mathbf{2}1^*$
$\omega\mathbf{0}$	$d_i \leftarrow 1$	$\omega\mathbf{1}$
$\alpha\mathbf{0}1^k$	$d_i \leftarrow 1$	$\alpha\mathbf{1}1^k$

(b) Case analysis for *decrement*(d, i).

Initial configuration	Action	Final configuration
$\alpha\mathbf{0}2\omega$	*fix-borrow*(d, i); $d_i \leftarrow 1$	$\alpha\mathbf{1}1\omega$
$\alpha\mathbf{0}1^k2\omega$	*fix-borrow*(d, i); $d_i \leftarrow 1$	$\alpha\mathbf{1}01^{k-1}2\omega$
$\alpha\mathbf{0}1^k$	*fix-borrow*(d, i); $d_i \leftarrow 1$	$\alpha\mathbf{1}01^{k-1}$
$\alpha 01^*\mathbf{1}2\omega$	*fix-borrow*(d, i); $d_i \leftarrow 2$	$\alpha 01^*\mathbf{2}1\omega$
$\alpha 01^*\mathbf{1}1^k2\omega$	*fix-borrow*(d, i); $d_i \leftarrow 2$	$\alpha 01^*\mathbf{2}01^{k-1}2\omega$
$\alpha 01^*\mathbf{1}1^k$	*fix-borrow*(d, i); $d_i \leftarrow 2$	$\alpha 01^*\mathbf{2}01^{k-1}$
$\alpha\mathbf{1}1^*02\omega$	*fix-borrow*(d, a); $d_i \leftarrow 0$	$\alpha\mathbf{0}1^*21\omega$
$\alpha\mathbf{1}1^*01^k2\omega$	*fix-borrow*(d, a); $d_i \leftarrow 0$	$\alpha\mathbf{0}1^*201^{k-1}2\omega$
$\alpha\mathbf{1}1^*01^k$	*fix-borrow*(d, a); $d_i \leftarrow 0$	$\alpha\mathbf{0}1^*201^{k-1}$
$\alpha 01^*\mathbf{2}1^*02\omega$	*fix-borrow*(d, a); $d_i \leftarrow 1$	$\alpha 01^*\mathbf{1}1^*21\omega$
$\alpha 01^*\mathbf{2}1^*01^k2\omega$	*fix-borrow*(d, a); $d_i \leftarrow 1$	$\alpha 01^*\mathbf{1}1^*201^{k-1}2\omega$
$\alpha 01^*\mathbf{2}1^*01^k$	*fix-borrow*(d, a) ; $d_i \leftarrow 1$	$\alpha 01^*\mathbf{1}1^*201^{k-1}$
$\alpha\mathbf{1}1^*$	$d_i \leftarrow 0$	$\alpha\mathbf{0}1^*$
$\alpha 01^*\mathbf{1}$	$d_i \leftarrow 0$	$\alpha 01^*$
$\alpha 01^*\mathbf{2}1^*$	$d_i \leftarrow 1$	$\alpha 01^*\mathbf{1}1^*$

Lemma 3. *Let $\langle d_0, d_1, \ldots d_{r-1}\rangle$ be the cardinality sequence of a node x in a strictly-regular tree. If the last block of this sequence is a 01^+ block, then $rank(x) = r-1$; otherwise, $rank(x) = r$.*

The next lemma illustrates the exponentiality property for such trees.

Lemma 4. *A strictly-regular tree of rank r has at least 2^r nodes.*

Proof. The proof is by induction. The claim is clearly true for nodes of rank 0. Assume the hypothesis is true for all the subtrees of a node x with rank r. Let y be the child of x with the largest rank. From Lemma 3, if the last block of the cardinality sequence of x is a 01^+ block, then $rank(x) = rank(y)$. Using induction, the number of nodes of y's subtree is at least 2^r, and the lemma follows. Otherwise, the cardinality sequence of x only contains 01^*2 and 1^+ blocks. We conclude that there exists a distinct subtree of x whose rank is at least i, for every i from 0 to $r-1$. Again using induction, the size of the tree rooted at x must be at least $1 + \sum_{i=0}^{r-1} 2^i = 2^r$. □

The operations that we would like to efficiently support include: adding a subtree whose root has rank at most r to the children of x; detaching a subtree from the children of x; splitting the sequence of the children of x, those having the highest ranks and the others; and concatenating a strictly-regular subsequence of trees, whose smallest rank equals r, to the children of x.

In accordance, we need to support implementations corresponding to the subroutines *fix-carry* and *fix-borrow*. For these, we use *link* and *unlink*.

link(T_1, T_2)**:** Assert that the roots of T_1 and T_2 have the same rank. Make one root the child of the other, and increase the rank of the surviving root by 1.

unlink(T)**:** Detach a child with the largest rank from the root of tree T. If T has rank r, the resulting two trees will have ranks either $r-1$, $r-1$ or $r-1$, r.

Subroutine *fix-carry*(d, i), which converts two consecutive digits $d_i = 2$ and $d_{i+1} = q$ to $0, q+1$ is realizable by subroutine *link*. Subroutine *fix-borrow*(d, i), which converts two consecutive digits $d_i = 0$ and $d_{i+1} = q$ to $2, q-1$ is realizable by subroutine *unlink* that results in two trees of equal rank. However, unlinking a tree of rank r may result in one tree of rank $r-1$ and another of rank r. In such case, a *fix-borrow* corresponds to converting the two digits $0, q$ to $1, q$. For this scenario, as for Table 2(b), it is also easy to show that all the cases following a decrement lead to a strictly-regular sequence. We leave the details for the reader to verify.

3 Application: Meldable Priority Queues

Our motivation is to investigate the worst-case bound for the number of element comparisons performed by *delete* under the assumption that *find-min*, *insert*, and *meld* have $O(1)$ worst-case cost. From the comparison-based lower bound for sorting, we know that if *find-min* and *insert* only involve $O(1)$ element comparisons, *delete* has to perform at least $\lg n - O(1)$ element comparisons, where n is the number of elements stored prior to the operation.

3.1 Brodal's Meldable Priority Queues

Our development is based on the priority queue presented in [7]. In this section, we describe this data structure. We also analyse the constant factor in the bound on the number of element comparisons performed by *delete*, since the original analysis was only asymptotic.

The construction in [7] is based on two key ideas. First, *insert* is supported at $O(1)$ worst-case cost. Second, *meld* is reduced to *insert* by allowing a priority queue to store other priority queues inside it. To make this possible, the whole data structure is a tree having two types of nodes: $\boxdot$-nodes (read: *square* or *type-I nodes*) and $\odot$-nodes (read: *circle* or *type-II nodes*). Each node stores a locator to an element, which is a representative of the descendants of the node; the representative has the smallest element among those of its descendants.

Each node has a non-negative integer *rank*. A node of rank 0 has no $\odot$-children. For an integer $r > 0$, the $\odot$-children of a node of rank r have ranks from 0 to $r - 1$. Each node can have at most one $\boxdot$-child and that child can be of arbitrary rank. The number of $\odot$-children is restricted to be at least one and at most three per rank. More precisely, the *regularity constraint* posed is that the cardinality sequence is of the form $\big(1 \mid 2 \mid 12^*3\big)^*$. This regular number system allows for increasing the least significant digit at $O(1)$ worst-case cost. In addition, because of the zeroless property, the size of a subtree of rank r is at least 2^r and the number of children of its root is at most $2r$. The rank of the root is required to be zero. So, if the tree holds more than one element, the other elements are held in the subtree rooted at the $\boxdot$-child of the root.

To represent such multi-way tree, the standard child-sibling representation can be used. Each node stores its rank as an integer, its type as a Boolean, a pointer to its parent, a pointer to its sibling, and a pointer to its $\odot$-child having the highest rank. The children of a node are kept in a circular singly-linked list containing the $\odot$-children in rank order and the $\boxdot$-child after the $\odot$-child of the highest rank; the $\boxdot$-child is further connected to the $\odot$-child of rank 0. Additionally, each node stores a pointer to a linked list, which holds pointers to the first $\odot$-node in every group of three consecutive nodes of the same rank corresponding to a 3 in the cardinality sequence.

A basic subroutine used in the manipulation of these trees is *link*. For node u, let $\mathit{element}(u)$ denote the element associated with u. Let u and v be two nodes of the same rank such that $\mathit{element}(u) \leq \mathit{element}(v)$. Now, *link* makes v a $\odot$-child of u. This increases the rank of u by one. Note that *link* has $O(1)$ worst-case cost and performs one element comparison.

The minimum element is readily found by accessing the root of the tree, so *find-min* is easily accomplished at $O(1)$ worst-case cost.

When inserting a new element, a node is created. The new element and those associated with the root and its $\boxdot$-child are compared; the two smallest among the three are associated with the root and its $\boxdot$-child, and the largest is associated with the created node. Hereafter, the new node is added as a $\odot$-child of rank 0 to the $\boxdot$-child of the root. Since the cardinality sequence of that node

was regular before the insertion, only $O(1)$ structural changes are necessary to restore the regularity constraint. That is, *insert* has $O(1)$ worst-case cost.

To meld two trees, the elements associated with the root and its $\boxdot$-child are taken from both trees and these four elements are sorted. The largest element is associated with a $\boxdot$-child of the root of one tree. Let T be that tree, and let S be the other tree. The two smallest elements are then associated with the root of S and its $\boxdot$-child. Accordingly, the other two elements are associated with the root of T and its $\boxdot$-child. Subsequently, T is added as a rank-0 $\odot$-child to the $\boxdot$-child of the root of S. So, also *meld* has $O(1)$ worst-case cost.

When deleting an element, the corresponding node is located and made the current node. If the current node is the root, the element associated with the $\boxdot$-child of the root is swapped with that associated with the root, and the $\boxdot$-child of the root is made the current node. On the other hand, if the current node is a $\odot$-node, the elements associated with the current node and its parent are swapped until a $\boxdot$-node is reached. Therefore, both cases reduce to a situation where a $\boxdot$-node is to be removed.

Assume that we are removing a $\boxdot$-node z. The actual removal involves finding a node that holds the smallest element among the elements associated with the children of z (call this node x), and finding a node that has the highest rank among the children of x and z (call this node y). To reestablish the regularity constraint, z is removed, x is promoted into its place, y is detached from its children, and all the children previously under x and y, plus y itself, are moved under x. This is done by performing repeated linkings until the number of nodes of the same rank is one or two. The rank of x is updated accordingly.

In the whole deletion process $O(\lg n)$ nodes are handled and $O(1)$ work is done per node, so the total cost of *delete* is $O(\lg n)$. To analyse the number of element comparisons performed, we point out that a node with rank r can have up to $2r$ $\odot$-children (not $3r$ as stated in [7]). Hence, finding the smallest element associated with a node requires up to $2\lg n + O(1)$ element comparisons, and reducing the number of children from $6\lg n + O(1)$ to $\lg n + O(1)$ involves $5\lg n + O(1)$ element comparisons (each *link* requires one). To see that this bound is possible, consider the addition of four numbers 1, 1232^k, 2222^k, and 1232^k (where the least significant digits are listed first), which gives $1211^{k+1}2$.

Our discussion so far can be summarized as follows.

Theorem 3. *Brodal's meldable priority queue, as described in [7], supports find-min, insert, and meld at $O(1)$ worst-case cost, and delete at $O(\lg n)$ worst-case cost including at most $7\lg n + O(1)$ element comparisons.*

3.2 Our Improvement

Consider a simple mixed scheme, in which the number system used for the children of $\odot$-nodes is perfect, following the pattern 1^*, and that used for the children of $\boxdot$-nodes is regular. This implies that the $\odot$-nodes form binomial trees [3]. After this modification, the bounds for *insert* and *meld* remain the same if we rely on the delayed melding strategy. However, since each node has at most

$\lg n + O(1)$ children, the bound for *delete* would be better than that reported in Theorem 3. Such an implementation of *delete* has three bottlenecks: finding the minimum, executing a delayed *meld*, and adding the $\odot$-children of a $\boxdot$-node to another node. In this mixed system, each of these three procedures requires at most $\lg n + O(1)$ element comparisons. Accordingly, *delete* involves at most $3 \lg n + O(1)$ element comparisons. Still, the question is how to do better!

The major change we make is to use the strictly-regular system instead of the zeroless regular system. We carry out *find-min*, *insert*, and *meld* similar to [7]. We use subroutine *merge* to combine two trees. Let y and y' be the roots of these trees, and let r and r' be their respective ranks where $r \leq r'$. We show how to *merge* the two trees at $O(r)$ worst-case cost using $O(1)$ element comparisons. For this, we have to locate the nodes representing the extreme digits closest to r in the cardinality sequence of y'. Consequently, by Theorems 1 and 2, a cut or an increment at that rank is done at $O(1)$ worst-case cost. If $element(y') \leq element(y)$, add y as a $\odot$-child of y', update the rank of y' and stop. Otherwise, cut the $\odot$-children of y' at r. Let the two resulting sublists be C and D, C containing the nodes of lower rank. Then, concatenate the lists representing the sequence of the $\odot$-children of y and the sequence D. We regard y' together with the $\odot$-children in C and y''s earlier $\boxdot$-child as one tree whose root y' is a $\odot$-node. Finally, place this tree under y and update the rank of y.

Now we show how to improve *delete*. If the node to be deleted is the root, we swap the elements associated with the root and its $\boxdot$-child, and let that $\boxdot$-node be the node z to be deleted. If the node to be deleted is a $\odot$-node, we repeatedly swap the elements associated with this node and its parent until the current node is a $\boxdot$-node (Case 1) or the rank of the current node is the same as that of its parent (Case 2). When the process stops, the current node z is to be deleted.

Case 1: z is a $\boxdot$-node. Let x denote the node that contains the smallest element among the children of z (if any). We remove z, lift x into its place, and make x into a $\boxdot$-node. Next, we move all the other $\odot$-children of z under x by performing an addition operation, and update the rank of x. Since z and x may each have had a $\boxdot$-child, there may be two $\boxdot$-children around. In such case, *merge* such two subtrees and make the root of the resulting tree the $\boxdot$-child of x.

Case 2: z is a $\odot$-node. Let p be the parent of z. We remove z and move its $\odot$-children to p by performing an addition operation. As $rank(p) = rank(z)$ before the addition, $rank(p) = rank(z)$ or $rank(z) + 1$ after the addition. If $rank(p) = rank(z) + 1$, to ensure that $rank(p)$ remains the same as before the operation, we detach the child of p that has the highest rank and *merge* the subtree rooted at it with the subtrees rooted at the $\boxdot$-children of p and z (there could be up to two such subtrees), and make the root of the resulting tree the $\boxdot$-child of p.

Let r be the maximum rank of a node in the tree under consideration. Climbing up the tree to locate a node z has $O(r)$ cost, since after every step the new current node has a larger rank. In Case 1, a $\boxdot$-node is deleted at $O(r)$ cost involving at most r element comparisons when finding its smallest child. In Cases

1 and 2, the addition of the $\odot$-children of two nodes has $O(r)$ cost and requires at most r element comparisons. Additionally, applying the *merge* operation on two trees (Case 1) or three trees (Case 2) has $O(r)$ cost and requires $O(1)$ element comparisons. Thus, the total cost is $O(r)$ and at most $2r + O(1)$ element comparisons are performed. Using Lemma 4, $r \leq \lg n$, and the claim follows.

In summary, our data structure improves the original data structure in two ways. First, by Lemma 4, the new system reduces the maximum number of children a node can have from $2 \lg n$ to $\lg n$. Second, the new system breaks the bottleneck resulting from delayed melding, since two subtrees can be merged with $O(1)$ element comparisons. The above discussion implies the following theorem.

Theorem 4. *Let n denote the number of elements stored in the data structure prior to a deletion. There exists a priority queue that supports find-min, insert, and meld at $O(1)$ worst-case cost, and delete at $O(\lg n)$ worst-case cost including at most $2 \lg n + O(1)$ element comparisons.*

4 Other Applications

Historically, it is interesting to note that in early papers a number system supporting increments and decrements of an arbitrary digit was constructed by putting two regular systems back to back, i.e. $d_i \in \{0, 1, 2, 3, 4, 5\}$. It is relatively easy to prove the correctness of this system. This approach was used in [14] for constructing catenable deques, in [9] for constructing catenable finger search trees, and in [8] for constructing meldable priority queues. (In [8], $d_i \in \{2, 3, 4, 5, 6, 7\}$ is imposed, since an extra constraint that $d_i \geq 2$ was required to facilitate the violation reductions and to guarantee the exponentiality property.) Later on, it was realized that the extended-regular system, $d_i \in \{0, 1, 2, 3\}$, could be utilized for the same purpose (see, for example, [6]). The strictly-regular system may be employed in applications where these more extensive number systems have been used earlier. This replacement, when possible, would have two important consequences:

1. The underlying data structures become simpler.
2. The operations supported may become a constant factor faster.

While surveying papers that presented potential applications to the new number system, we found that, even though our number system may be applied, there were situations where other approaches would be more favourable. For example, the relaxed heap described in [11] relies on the zeroless extended-regular system to support increments and decrements. Naturally, the strictly-regular system could be used instead, and this would reduce the number of trees that have to be maintained. However, the approach of using a two-tier structure as described in [11] makes the reduction in the number of trees insignificant since the amount of work done is proportional to the logarithm of the number of trees. Also, a fat heap [6] uses the extended-regular binary system for keeping track of the potential violation nodes and the extended-regular ternary system for keeping

track of the trees in the structure. However, we discovered that a priority queue with the same functionality and efficiency can be implemented with simpler tools without using number systems at all. The reader is warned: number systems are powerful tools but they should not be applied haphazardly.

Up till now we have ignored the cost of accessing the extreme digits in the vicinity of a given digit. When dealing with the regular or the extended-regular systems this can be done at $O(1)$ cost by using the guides described in [8]. In contrary, for our number system, accessing the extreme digits in the vicinity of any digit does not seem to be doable at $O(1)$ cost. However, the special case of accessing the first and last extreme digits is soluble at $O(1)$ cost.

In some applications, like fat heaps [6] and the priority queues described in [8], the underlying number system is ternary. We have not found a satisfactory solution to extend the strictly-regular system to handle ternary numbers efficiently; it is an open question whether such an extension exists.

References

1. Clancy, M., Knuth, D.: A programming and problem-solving seminar. Technical Report STAN-CS-77-606, Dept. of Computer Science, Stanford University (1977)
2. Guibas, L.J., McCreight, E.M., Plass, M.F., Roberts, J.R.: A new representation for linear lists. In: Proceedings of the 9th Annual ACM Symposium on Theory of Computing, ACM (1977) 49–60
3. Vuillemin, J.: A data structure for manipulating priority queues. Communications of the ACM **21**(4) (1978) 309–315
4. Okasaki, C.: Purely Functional Data Structures. Cambridge University Press (1998)
5. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, & Tools. 2nd edn. Pearson Education, Inc. (2007)
6. Kaplan, H., Shafrir, N., Tarjan, R.E.: Meldable heaps and Boolean union-find. In: Proceedings of the 34th Annual ACM Symposium on Theory of Computing, ACM (2002) 573–582
7. Brodal, G.S.: Fast meldable priority queues. In: Proceedings of the 4th International Workshop on Algorithms and Data Structures. Volume 955 of Lecture Notes in Computer Science., Springer-Verlag (1995) 282–290
8. Brodal, G.S.: Worst-case efficient priority queues. In: Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms, ACM/SIAM (1996) 52–58
9. Kaplan, H., Tarjan, R.E.: Purely functional representations of catenable sorted lists. In: Proceedings of the 28th Annual ACM Symposium on Theory of Computing, ACM (1996) 202–211
10. Elmasry, A.: A priority queue with the working-set property. International Journal of Foundations of Computer Science **17**(6) (2006) 1455–1465
11. Elmasry, A., Jensen, C., Katajainen, J.: Two-tier relaxed heaps. Acta Informatica **45**(3) (2008) 193–210
12. Elmasry, A., Jensen, C., Katajainen, J.: Multipartite priority queues. ACM Transactions on Algorithms **5**(1) (2008) Article 14
13. Jensen, C.: A note on meldable heaps relying on data-structural bootstrapping. CPH STL Report 2009-2, Department of Computer Science, University of Copenhagen (2009) Available at http://cphstl.dk.

14. Kaplan, H., Tarjan, R.E.: Persistent lists with catenation via recursive slow-down. In: Proceedings of the 27th Annual ACM Symposium on Theory of Computing, ACM (1995) 93–102

Two new methods for constructing double-ended priority queues from priority queues*

Amr Elmasry† Claus Jensen‡

Jyrki Katajainen‡

† Max-Planck Institut für Informatik
Saarbrücken, Germany

‡ Datalogisk Institut, Københavns Universitet
Universitetsparken 1, 2100 København Ø, Denmark

Abstract

We introduce two data-structural transformations to construct double-ended priority queues from priority queues. To apply our transformations the priority queues exploited must support the extraction of an unspecified element, in addition to the standard priority-queue operations. With the first transformation we obtain a double-ended priority queue which guarantees the worst-case cost of $O(1)$ for *find-min*, *find-max*, *insert*, *extract*; and the worst-case cost of $O(\lg n)$ with at most $\lg n+O(1)$ element comparisons for *delete*. With the second transformation we get a meldable double-ended priority queue which guarantees the worst-case cost of $O(1)$ for *find-min*, *find-max*, *insert*, *extract*; the worst-case cost of $O(\lg n)$ with at most $\lg n+O(\lg \lg n)$ element comparisons for *delete*; and the worst-case cost of $O(\min\{\lg m, \lg n\})$ for *meld*. Here, m and n denote the number of elements stored in the data structures prior to the operation in question.

AMS Classification. 68P05, 68P10, 68W40, 68Q25

Keywords. Data structures, priority queues, double-ended priority queues, min-max priority queues, priority deques, meticulous analysis, comparison complexity

The work of the authors was 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 was supported by Alexander von Humboldt fellowship.

1 Introduction

In this paper, we study efficient realizations of data structures that can be used to maintain a collection of double-ended priority queues. The fundamental operations to be supported include *find-min*, *find-max*, *insert*, and *delete*. For single-ended priority queues only *find-min* or *find-max* is supported. A (double-ended) priority queue is called *meldable* if it also supports operation *meld*. Even though the data-structural transformations to be presented are fully general, our main focus is on the comparison complexity of double-ended priority-queue operations. Throughout this paper we use m and n to denote the number of elements stored in a data structure prior to an operation and $\lg n$ as a shorthand for $\log_2(\max\{2, n\})$. Many data structures [2, 3, 5, 7, 8, 9, 10, 11, 17, 18, 19, 20, 21, 23, 24, 25] have been proposed for the realization of a double-ended priority queue, but none of them achieve $\lg n + o(\lg n)$ element comparisons per *delete*, if *find-min*, *find-max*, and *insert* must have the worst-case cost of $O(1)$.

We use the *word RAM* as our model of computation as defined in [15]. We assume the availability of instructions found in contemporary computers, including built-in functions for allocating and freeing memory. We use the term *cost* to denote the sum of instructions, element constructions, element destructions, and element comparisons performed.

When defining a (double-ended) priority queue, we use the locator abstraction discussed in [14]. A *locator* is a mechanism for maintaining the association between an element and its current position in a data structure. A locator follows its element even if the element changes its position inside the data structure.

Our goal is to develop realizations of a double-ended priority queue that support the following operations:

find-min(Q) (*find-max*(Q)). Return a locator to an element that, of all elements in double-ended priority queue Q, has a minimum (maximum) value. If Q is empty, return a *null* locator.

insert(Q, p). Add an element with locator p to double-ended priority queue Q.

extract(Q). Extract an *unspecified* element from double-ended priority queue Q and return a locator to that element. If Q is empty, return a *null* locator.

delete(Q, p). Remove the element with locator p from double-ended priority queue Q (without destroying the element).

Of these operations, *extract* is non-standard, but we are confident that it is useful, for example, for data-structural transformations. The following operations may also be provided.

meld(Q, R). Move all elements from double-ended priority queues Q and R to a new double-ended priority queue S, destroy Q and R, and return S.

decrease(Q, p, x) (*increase*(Q, p, x)). Replace the element with locator p by element x, which should not be greater (smaller) than the old element.

Any double-ended priority queue can be used for sorting (say, a set of size n). So if *find-min* (*find-max*) and *insert* have a cost of $O(1)$, *delete* must perform at least $\lg n - O(1)$ element comparisons in the worst case in the decision-tree model. Similarly, as observed in [21], if *find-min* (*find-max*) and *insert* have a cost of $O(1)$, *increase* (*decrease*) must perform at least $\lg n - O(1)$ element comparisons in the worst case. Recall, however, that single-ended priority queues can support *find-min*, *insert*, and *decrease* (or *find-max*, *insert*, and *increase*) at the worst-case cost of $O(1)$ (see, for example, [6]).

1.1 Previous approaches

Most realizations of a (meldable) double-ended priority queue—but not all—use two priority queues, minimum priority queue Q_{min} and maximum priority queue Q_{max}, that contain the minimum and maximum candidates, respectively. The approaches to guarantee that a minimum element is in Q_{min} and a maximum element in Q_{max} can be classified into three main categories [10]: dual correspondence, total correspondence, and leaf correspondence. The correspondence between two elements can be maintained implicitly, as done in many implicit or space-efficient data structures, or explicitly relying on pointers.

In the *dual correspondence* approach a copy of each element is kept both in Q_{min} and Q_{max}, and *clone pointers* are maintained between the corresponding copies. Using this approach Brodal [5] showed that *find-min*, *find-max*, *insert*, and *meld* can be realized at the worst-case cost of $O(1)$, and *delete* at the worst-case cost of $O(\lg n)$. Asymptotically, Brodal's double-ended priority queue is optimal with respect to all operations. However, as pointed out by Cho and Sahni [9], Brodal's double-ended priority queue uses almost twice as much space as his single-ended priority queue, and the leading constant in the bound on the complexity of *delete* is high (according to our analysis the number of element comparisons performed in the worst case is at least $4 \lg n - O(1)$ for the priority queue and $8 \lg n - O(1)$ for the double-ended priority queue).

In the *total correspondence* approach, both Q_{min} and Q_{max} contain $\lfloor n/2 \rfloor$ elements and, if n is odd, one element is kept outside these structures. Every element x in Q_{min} has a *twin* y in Q_{max}, x is no greater than y, and there is a *twin pointer* from x to y and vice versa. Both Chong and Sahni [10] and Makris et al. [21] showed that with this approach the space efficiency of Brodal's data structure can be improved. Now the elements are stored only once so the amount of extra space used is nearly cut in half. The results reported in [10, 21] are rephrased in Table 1 (on p. 123).

A third possibility is to employ the *leaf correspondence* approach, where only the elements stored at the leaves of the data structures used for realizing Q_{min} and Q_{max} have their corresponding twins. This approach is less general and requires that some type of tree is used to represent the two priority queues. Chong and Sahni [10] showed that Brodal's data structure could be customized to rely on the leaf correspondence as well, but the worst-case complexity of *delete* is still about twice as high as that in the original priority queue.

In addition to these general transformations, several ad-hoc modifications of

existing priority queues have been proposed. These modifications inherit their properties, like the operation repertoire and the space requirements, directly from the modified priority queue. Most notably, many of the double-ended priority queues proposed do not support general *delete*, *meld*, nor *insert* at the worst-case cost of $O(1)$. Even when such priority queues can be modified to provide *insert* at the worst-case cost of $O(1)$, as shown by Alstrup et al. [1], *delete* would perform $\Theta(\lg n)$ additional element comparisons as a result.

1.2 Efficient priority queues

Our data-structural transformations are general, but to obtain our best results we rely on our earlier work on efficient priority queues [12, 13]. Our main goal in these two earlier papers was to reduce the number of element comparisons performed by *delete* without sacrificing the asymptotic bounds for the other supported operations. In this paper, we use these priority queues as building blocks to achieve the same goal for double-ended priority queues.

Both our data-structural transformations require that the priority queues used support *extract*, which extracts an unspecified element from the given priority queue and returns a locator to that element. This operation is used for moving elements from one priority queue to another and for reconstructing a priority queue incrementally. Many existing priority queues can be easily extended to support *extract*. When this is not immediately possible, the borrowing technique presented in [12, 13] may be employed.

The performance of the priority queues described in [12, 13] is summarized in the following lemmas.

Lemma 1 [12] *There exists a priority queue that supports find-min, insert, and extract at the worst-case cost of* $O(1)$*; and delete at the worst-case cost of* $O(\lg n)$ *including at most* $\lg n + O(1)$ *element comparisons.*

Lemma 2 [13] *There exists a meldable priority queue that supports find-min, insert, extract, and decrease at the worst-case cost of* $O(1)$*; delete at the worst-case cost of* $O(\lg n)$ *including at most* $\lg n + O(\lg \lg n)$ *element comparisons; and meld at the worst-case cost of* $O(\min\{\lg m, \lg n\})$.

1.3 Our results

In this paper, we present two transformations that show how priority queues can be employed to obtain double-ended priority queues. With our first transformation we obtain a data structure for which all fundamental operations are nearly optimal with respect to the number of element comparisons performed. With our second transformation we obtain a data structure that also supports *meld*.

In our first transformation we divide the elements into three collections containing elements smaller than, equal to, and greater than a partitioning element. It turns out to be cheaper to maintain a single partitioning element than

Table 1: Complexity of general transformations from priority queues to double-ended priority queues. Here C^n_{op} denotes the worst-case cost of double-ended priority-queue operation *op* for a given problem size n, and c^n_{op} the corresponding cost of the priority-queue operation *op*. Throughout the paper, we assume that functions c^n_{op} are non-decreasing and smooth, i.e. that for non-negative integers m and n, $m \leq n \leq 2m$, $c^m_{op} \leq c^n_{op} \leq O(1) \cdot c^m_{op}$. Naturally, if $c^n_{\mathit{find\text{-}max}} = c^n_{\mathit{find\text{-}min}}$, then $C^n_{\mathit{find\text{-}max}} = C^n_{\mathit{find\text{-}min}}$.

Reference / Complexity	[10, 21]	This paper, Section 2	This paper, Section 3
$C^n_{\mathit{find\text{-}min}}$	$c^{n/2}_{\mathit{find\text{-}min}} + O(1)$	$2 \cdot c^n_{\mathit{find\text{-}min}} + O(1)$	$c^{n/2}_{\mathit{find\text{-}min}} + O(1)$
C^n_{insert}	$2 \cdot c^{n/2}_{\mathit{insert}} + O(1)$	$O(1) \cdot c^n_{\mathit{extract}} + O(1) \cdot c^n_{\mathit{insert}} + O(1)$	$2 \cdot c^{n/2}_{\mathit{insert}} + O(1)$
C^n_{extract}	Not supported	$O(1) \cdot c^n_{\mathit{extract}} + O(1) \cdot c^n_{\mathit{insert}} + O(1)$	$2 \cdot c^{n/2}_{\mathit{extract}} + 2 \cdot c^{n/2}_{\mathit{decrease}} + O(1)$
C^n_{delete}	$2 \cdot c^{n/2}_{\mathit{delete}} + 2 \cdot c^{n/2}_{\mathit{insert}} + O(1)$	$c^n_{\mathit{delete}} + O(1) \cdot c^n_{\mathit{extract}} + O(1) \cdot c^n_{\mathit{insert}} + O(1)$	$c^{n/2}_{\mathit{delete}} + c^{n/2}_{\mathit{extract}} + c^{n/2}_{\mathit{insert}} + 2 \cdot c^{n/2}_{\mathit{decrease}} + O(1)$
$C^{m,n}_{\mathit{meld}}$	$2 \cdot c^{\lceil m/2 \rceil, n/2}_{\mathit{meld}} + 2 \cdot c^{m/2}_{\mathit{insert}} + O(1)$	Not supported	$2 \cdot c^{\lceil m/2 \rceil, n/2}_{\mathit{meld}} + 2 \cdot c^{m/2}_{\mathit{insert}} + O(1)$
extra space	$n + O(1)$ words	$(4/3)n$ elements $O(\lg n)$ words $3n$ bits	$n + O(1)$ words n bits

to maintain many twin relationships as done in the correspondence-based approaches. When developing this transformation we were inspired by the priority queue described in [22], where a related partitioning scheme is used. The way we implement partitioning allows efficient deamortization; in accordance our bounds are worst-case rather than amortized in contrast to the bounds derived in [22]. Our second transformation combines the total correspondence approach with an efficient priority queue supporting *decrease*. This seems to be a new application of priority queues supporting fast *decrease*.

The complexity bounds attained are summarized in Table 1. The main difference between the earlier results and our results is that the leading constant in the cost of *delete* is reduced from two to one, provided that the priority queues used support *insert*, *extract*, and *decrease* at the worst-case cost of O(1). By constructing double-ended priority queues from the priority queues mentioned in Lemmas 1 and 2, respectively, we get the following theorems.

Theorem 1 *There exists a double-ended priority queue that supports find-min, find-max, insert, and extract at the worst-case cost of* $O(1)$*; and delete at the worst-case cost of* $O(\lg n)$ *including at most* $\lg n + O(1)$ *element comparisons.*

Theorem 2 *There exists a meldable double-ended priority queue that supports find-min, find-max, insert, and extract at the worst-case cost of $O(1)$; delete at the worst-case cost of $O(\lg n)$ including at most $\lg n + O(\lg \lg n)$ element comparisons; and meld at the worst-case cost of $O(\min\{\lg m, \lg n\})$.*

2 Pivot-based partitioning

In this section we show how a double-ended priority queue, call it Q, can be constructed with the help of three priority queues Q_{min}, Q_{mid}, and Q_{max}. The basic idea is to maintain a special *pivot* element and use it to partition the elements held in Q into three candidate collections: Q_{min} holding the elements smaller than *pivot*, Q_{mid} those equal to *pivot*, and Q_{max} those larger than *pivot*. Note that, even if the priority queues are meldable, the resulting double-ended priority queue cannot provide *meld* efficiently.

To illustrate the general idea, let us first consider a realization that guarantees good amortized performance for all modifying operations (*insert*, *extract*, and *delete*). We divide the execution of the operations into phases. Each phase consists of $\max\{1, \lfloor n_0/2 \rfloor\}$ modifying operations, if at the beginning of a phase the data structure stored n_0 elements (initially, $n_0 = 0$). At the end of each phase, a restructuring is done by partitioning the elements using the median element as *pivot*. That way we ensure that at any given time—except if there are no elements in Q—the minimum (maximum) element is in Q_{min} (Q_{max}) or, if it is empty, in Q_{mid}. Thus all operations of a phase can be performed correctly.

Now it is straightforward to carry out the double-ended priority-queue operations by relying on the priority-queue operations.

find-min(Q) (*find-max*(Q)). If Q_{min} (Q_{max}) is non-empty, the minimum (maximum) of Q_{min} (Q_{max}) is returned; otherwise, the minimum of Q_{mid} is returned.

insert(Q, p). If the element with locator p is smaller than, equal to, or greater than *pivot*, the element is inserted into Q_{min}, Q_{mid}, or Q_{max}, respectively.

extract(Q). If Q_{min} is non-empty, an element is extracted from Q_{min}; otherwise, an element is extracted from Q_{mid}.

delete(Q, p). Depending on in which component Q_{min}, Q_{mid}, or Q_{max} the element with locator p is stored, the element is removed from that component. To implement this efficiently, we assume that each node of the priority queues is augmented by an extra field that gives the name of the component in which that node is stored.

A more detailed description of the operations on Q is given in Figure 1.

After each modifying operation it is checked whether the end of a phase is reached, and if this is the case, a partitioning is carried out. To perform the partitioning efficiently, all the current elements are copied to a temporary array

```
find-min(Q): // find-max(Q) is similar
  if size(Q_min) = 0
    return find-min(Q_mid)
  return find-min(Q_min)
insert(Q, p):
  if p.element() < pivot
    insert(Q_min, p)
  else if pivot < p.element()
    insert(Q_max, p)
  else
    insert(Q_mid, p)
  count++
  if count ≥ ⌊n_0/2⌋
    reorganize(Q)
extract(Q):
  p ← extract(Q_min)
  if p = null
    p ← extract(Q_mid)
  count++
  if count ≥ ⌊n_0/2⌋
    reorganize(Q)
  return p
delete(Q, p):
  R ← component(Q, p)
  delete(R, p)
  count++
  if count ≥ ⌊n_0/2⌋
    reorganize(Q)
```

```
reorganize(Q):
  n_0 ← size(Q)
  count ← 0
  construct an empty priority queue P
  allocate an element array A of size n_0
  i ← 1
  for R ∈ {Q_min, Q_mid, Q_max}
    for j ∈ {1, ..., size(R)}
      p ← extract(R)
      A[i++] ← p.element()
      insert(P, p)
  pivot ← selection(A[1: n_0], ⌈n_0/2⌉)
  destroy A
  for i ∈ {1, ..., n_0}
    p ← extract(P)
    if p.element() < pivot
      insert(Q_min, p)
    else if pivot < p.element()
      insert(Q_max, p)
    else
      insert(Q_mid, p)
```

Figure 1: This pseudo-code implements the amortized scheme. The subroutine $component(Q, p)$ is assumed to return the component of Q in which the element with locator p is stored.

A. This is done by employing *extract* to repeatedly remove elements from Q. Each element is copied to A and temporarily inserted into another data structure P for later use. We chose to implement P as a priority queue to reuse the same structure of the nodes as Q. A linear-time selection algorithm [4] is then used to set *pivot* to the value of the median element in array A. Actually, we rely on a space-efficient variant of the standard prune-and-search algorithm described in [16, Section 3.6]. For an input of size n, the extra space used by this variant is $O(\lg n)$ words. After partitioning, A is destroyed, and Q is reconstructed by repeatedly re-extracting the elements from the temporary structure P and inserting them into Q (using the new *pivot*).

Assuming that priority-queue operations *insert* and *extract* have a cost of $O(1)$, the restructuring done at the end of a phase has the worst-case cost of $O(n_0)$. So a single modifying operation can be expensive, but when the reorganization work is amortized over the $\max\{1, \lfloor n_0/2 \rfloor\}$ operations executed in a phase, the amortized cost is only $O(1)$ per modifying operation.

Next we consider how we can get rid of the amortization. In our deamortization strategy, each phase consists of $\max\{1, \lfloor n_0/4 \rfloor\}$ modifying operations.

We maintain the following *size invariant*: If at the beginning of a phase there are n_0 elements in total, the size of Q_{min} (Q_{max}) plus the size of Q_{mid} is at least $\max\{1, \lfloor n_0/4 \rfloor\}$. This guarantees that the minimum (maximum) element is in Q_{min} (Q_{max}) or, if it is empty, in Q_{mid}. Throughout each phase, three subphases are performed in sequence; each subphase consists of about equally many modifying operations.

In the first subphase, the n_0 elements in Q at the beginning of the phase are to be incrementally copied to A. To facilitate this, we employ a supplementary data structure P that has the same form as Q and is composed of three components P_{min}, P_{mid}, and P_{max}. Accompanying each modifying operation, an adequate number of elements is copied from Q to A. This is accomplished by extracting an element from Q, copying the element to A, and inserting the node into P using the same *pivot* as in Q. An *insert* directly adds the given element to P without copying it to A, and a *delete* copies the deleted element to A only if that element is in Q. At the end of this subphase, A stores copies of all the elements that were in Q at the beginning of the phase, and all the elements that should be in the double-ended priority queue are now in P leaving Q empty.

In the second subphase the median of the elements of A is found. That is, the selection is based on the contents of Q at the beginning of the phase. In this subphase the modifying operations are executed normally, except that they are performed on P, while they incrementally take part in the selection process. Each modifying operation takes its own share of the work such that the whole selection process is finished before reaching the end of this subphase.

The third subphase is reserved for clean-up. Each modifying operation carries out its share of the work such that the whole clean-up process is finished before the end of the phase. First, A is destroyed by gradually destructing the elements copied and freeing the space after that. Second, all elements held in P are moved to Q. When moving the elements, *extract* and *insert* are used, and the median found in the second subphase is used as the partitioning element.

As to *find-min*, the current minimum can be found from one of the priority queues Q_{min} (Q_{mid} if Q_{min} is empty) or P_{min} (P_{mid} if P_{min} is empty). Hence, *find-min* (similarly for *find-max*) can still be carried out efficiently.

Even if the median found is exact for A, it is only an approximate median for the whole collection at the end of a phase. Since after freezing the contents of A at most $\max\{1, \lfloor n_0/4 \rfloor\}$ elements are added to or removed from the data structure, it is easy to verify that the size invariant holds for the next phase.

Let us now analyse the space requirements of the deamortized construction. Let n denote the present size of the double-ended priority queue, and n_0 the size of A. The worst case is when all the operations performed during the phase are deletions, and hence n may be equal to $3n_0/4$. That is, the array can never have more than $4n/3$ elements. In addition to array A, the selection routine requires a work area of $O(\lg n)$ words and each node has to store 3 bits to record the component (Q_{min}, Q_{mid}, Q_{max}, P_{min}, P_{mid}, or P_{max}) the node is in.

For the deamortized scheme, it is straightforward to derive the bounds given in Table 1. By combining the results derived for the transformation (Table 1) with the bounds known for the priority queues (Lemma 1), the bounds of The-

orem 1 are obtained.

3 Total correspondence

In this section we describe how an efficient meldable priority queue supporting *extract* and *decrease* can be utilized in the realization of a meldable double-ended priority queue Q. We use Q_{min} to denote the minimum priority queue of Q (supporting *find-min* and *decrease*) and Q_{max} the maximum priority queue of Q (supporting *find-max* and *increase*). We rely on the total correspondence approach, where each of Q_{min} and Q_{max} contains $\lfloor n/2 \rfloor$ elements, with the possibility of having one element outside these structures. To perform *delete* efficiently, instead of using two priority-queue *delete* operations as in [10, 21], we use only one *delete* and employ *extract* that may be followed by *decrease* and *increase*.

Each element stored in Q_{min} has a *twin* in Q_{max}. To maintain the twin relationships and to access twins fast, we assume that each node of the priority queues allows an attachment of one more pointer, a *twin pointer*. The element that has no twin is called a *singleton*. Using these concepts, the double-ended priority-queue operations on Q can be performed as follows (a more detailed description of the operations is given in Figure 2):

find-min(Q) (*find-max*(Q)). If Q_{min} (Q_{max}) is empty or the singleton of Q is smaller (greater) than the minimum (maximum) element of Q_{min} (Q_{max}), the singleton is returned; otherwise, the minimum (maximum) of Q_{min} (Q_{max}) is returned.

insert(Q, p). If Q has no singleton, the element with locator p is made the singleton of Q and nothing else is done. If Q has a singleton and the given element is smaller than the singleton, the element is inserted into Q_{min} and the singleton is inserted into Q_{max}; otherwise, the element is inserted into Q_{max} and the singleton is inserted into Q_{min}. Finally, the element and the singleton are made twins of each other.

extract(Q). If Q has a singleton, it is extracted and nothing else is done. If Q has no singleton, an element is extracted from Q_{min}. If Q_{min} was non-empty, an element is also extracted from Q_{max}. The element extracted from Q_{max} is made the new singleton, and the element extracted from Q_{min} is returned. Before this the twins of the two extracted elements are made twins of each other and their positions are swapped if necessary, and the order in Q_{min} and Q_{max} is restored by decreasing and increasing the swapped elements (if any).

delete(Q, p). If the element to be deleted is the singleton of Q, the singleton is removed and nothing else is done. If Q has a singleton, the element with locator p is removed from its component, the singleton is inserted into that component, and the singleton and the twin of the removed element are

```
find-min(Q): // find-max(Q) is similar
  s ← singleton(Q)
  t ← find-min(Q_min)
  if t = null or
    (s ≠ null and s.element() < t.element())
    return s
  return t
insert(Q, p):
  s ← singleton(Q)
  if s = null
    make-singleton(Q, p)
    return
  if p.element() < s.element()
    insert(Q_min, p)
    insert(Q_max, s)
  else
    insert(Q_min, s)
    insert(Q_max, p)
  make-twins(p, s)
  make-singleton(Q, null)
extract(Q):
  s ← singleton(Q)
  if s ≠ null
    make-singleton(Q, null)
    return s
  else
    p ← extract(Q_min)
    if p ≠ null
      q ← extract(Q_max)
      r ← twin(p)
      t ← twin(q)
      make-twins(r, t)
      swap-twins-if-necessary(Q, t)
      make-singleton(Q, q)
    return p
swap-twins-if-necessary(Q, s): // s in Q_min
  t ← twin(s)
  if t.element() < s.element()
    swap(s, t)
    decrease(Q_min, t, t.element())
    increase(Q_max, s, s.element())
delete(Q, p):
  s ← singleton(Q)
  if p = s
    make-singleton(Q, null)
    return
  P ← component(Q, p)
  t ← twin(p)
  T ← component(Q, t)
  if s = null
    q ← extract(T)
    s ← twin(q)
    make-singleton(Q, q)
  else
    insert(P, s)
    make-singleton(Q, null)
  make-twins(s, t)
  if P = Q_min
    swap-twins-if-necessary(Q, s)
  else
    swap-twins-if-necessary(Q, t)
  delete(P, p)
meld(Q, R):
  q ← singleton(Q)
  r ← singleton(R)
  if q ≠ null and r = null
    make-singleton(S, q)
  else if q = null and r ≠ null
    make-singleton(S, r)
  else
    make-singleton(S, null)
    if q ≠ null
      if q.element() < r.element()
        insert(Q_min, q)
        insert(Q_max, r)
      else
        insert(Q_min, r)
        insert(Q_max, q)
      make-twins(q, r)
  S_min ← meld(Q_min, R_min)
  S_max ← meld(Q_max, R_max)
  return S
```

Figure 2: This pseudo-code implements our second data-structural transformation. The subroutines used are assumed to have the following effects: *twin*(p) returns a locator to the twin of the element with locator p; *make-twins*(p, q) assigns the twin pointers between the elements with locators p and q; *singleton*(Q) returns a locator to the singleton of Q; *make-singleton*(Q, p) makes the element with locator p the singleton of Q and sets the twin pointer of p to *null*; *swap*(p, q) puts the element with locator p in place of the element with locator q, and vice versa; and *component*(Q, p) returns the component of Q, in which the element with locator p is stored. One way of implementing *component* is to attach to each node of Q a bit indicating whether that node is in Q_{min} or Q_{max}, and let *insert* update these bits.

made twins of each other. As in *extract* the two twins are swapped and the order in Q_{min} and Q_{max} is restored if necessary. On the other hand, if Q has no singleton, the element with locator p is removed, another element is extracted from the component of its twin, the extracted element is made the new singleton, and, if necessary, the twin of the extracted element and the twin of the removed element are swapped and the order in Q_{min} and Q_{max} is restored as above.

meld(Q, R). Let S denote the outcome of this operation. Without loss of generality, assume that the size of Q is smaller than or equal to that of R. If Q and R together have exactly one singleton, this element becomes the singleton of S. If they have two singletons, these are compared, the non-greater is inserted into Q_{min}, the non-smaller is inserted into Q_{max}, and the inserted elements are made twins of each other. After these preparations, Q_{min} and R_{min} are melded to become S_{min}, and Q_{max} and R_{max} are melded to become S_{max}.

It is straightforward to verify that the above implementation achieves the bounds of Table 1. By combining the results of Table 1 with the bounds known for the priority queues (Lemma 2), the bounds of Theorem 2 are obtained.

4 Conclusions

We conclude the paper with four open problems, the solution of which would improve the results presented in this paper.

1. One drawback of our first transformation is the extra space used for elements, and the extra element constructions and destructions performed when copying elements. The reason for copying elements instead of pointers is that some elements may be deleted during the selection process. It would be interesting to know whether the selection problem could be solved at linear cost when the input is allowed to be modified during the computation.

2. Our realization of a double-ended priority queue using the priority queues introduced in [12] works on a pointer machine, but the meldable version using the priority queues introduced in [13] relies on the capabilities of a RAM. This is in contrast with Brodal's data structure [5] which works on a pointer machine. Therefore, it is natural to ask whether random access could be avoided.

3. To obtain *meld* having the worst-case cost of $O(1)$, the price paid by Brodal [6] is a more expensive *delete*. It is unknown whether *meld* could be implemented at the worst-case cost of $O(1)$ such that at most $\lg n + o(\lg n)$ element comparisons are performed per *delete*.

4. If for a meldable double-ended priority queue *meld* is allowed to have the worst-case cost of $O(\min\{\lg m, \lg n\})$, it is still relevant to ask whether

delete can be accomplished at logarithmic cost with at most $\lg n + O(1)$ element comparisons.

References

[1] Alstrup, S., Husfeld, T., Rauhe, T., Thorup M.: Black box for constant-time insertion in priority queues. ACM Transactions on Algorithms **1**, 102–106 (2005).

[2] Arvind, A., Pandu Rangan, C.: Symmetric min-max heap: A simpler data structure for double-ended priority queue. Information Processing Letters **69**, 197–199 (1999).

[3] Atkinson, M. D., Sack, J.-R., Santoro, N., Strothotte, T.: Min-max heaps and generalized priority queues. Communications of the ACM **29**, 996–1000 (1986).

[4] Blum, M., Floyd, R. W., Pratt, V., Rivest, R. L., Tarjan, R. E.: Time bounds for selection. Journal of Computer and System Sciences **7**, 448–461 (1973).

[5] Brodal, G. S.: Fast meldable priority queues. Proceedings of the 4th International Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science **955**, Springer-Verlag, Berlin/Heidelberg (1995), 282–290.

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

[7] Carlsson, S.: The deap—A double-ended heap to implement double-ended priority queues. Information Processing Letters **26**, 33–36 (1987).

[8] Chang, S. C., Du, M. W.: Diamond deque: A simple data structure for priority deques. Information Processing Letters **46**, 231–237 (1993).

[9] Cho, S., Sahni, S.: Mergeable double-ended priority queues. International Journal of Foundations of Computer Science **10**, 1–18 (1999).

[10] Chong, K.-R., Sahni, S.: Correspondence-based data structures for double-ended priority queues. The ACM Journal of Experimental Algorithmics **5**, article 2 (2000).

[11] Ding, Y., Weiss, M. A.: The relaxed min-max heap: A mergeable double-ended priority queue. Acta Informatica **30**, 215–231 (1993).

[12] Elmasry, A., Jensen, C., Katajainen, J.: Multipartite priority queues. ACM Transactions on Algorithms (to appear).

[13] Elmasry, A., Jensen, C., Katajainen, J.: Two-tier relaxed heaps. Acta Informatica **45**, 193–210 (2008).

[14] Goodrich, M. T., Tamassia, R.: Algorithm Design: Foundations, Analysis, and Internet Examples. John Wiley & Sons, Inc., Hoboken (2002).

[15] Hagerup, T.: Sorting and searching on the word RAM. Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science **1373**, Springer-Verlag, Berlin/Heidelberg (1998), 366–398.

[16] Horowitz, E., Sahni, S., Rajasekaran, S.: Computer Algorithms/C++. Computer Science Press, New York (1998).

[17] Høyer, P.: A general technique for implementation of efficient priority queues. Proceedings of the 3rd Israel Symposium on the Theory of Computing and Systems, IEEE, Los Alamitos (1995), 57–66.

[18] Katajainen, J., Vitale, F.: Navigation piles with applications to sorting, priority queues, and priority deques. Nordic Journal of Computing **10**, 238–262 (2003).

[19] Khoong, C. M., Leong, H. W.: Double-ended binomial queues. Proceedings of the 4th International Symposium on Algorithms and Computation, Lecture Notes in Computer Science **762**, Springer-Verlag, Berlin/Heidelberg (1993), 128–137.

[20] van Leeuwen, J., Wood, D.: Interval heaps. The Computer Journal **36**, 209–216 (1993).

[21] Makris, C., Tsakalidis, A., Tsichlas, K.: Reflected min-max heaps. Information Processing Letters **86**, 209–214 (2003).

[22] Mortensen, C. W., Pettie, S.: The complexity of implicit and space-efficient priority queues. Proceedings of the 9th Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science **3608**, Springer-Verlag, Berlin/Heidelberg (2005), 49–60.

[23] Nath, S. K., Chowdhury, R. A., Kaykobad, M.: Min-max fine heaps. arXiv.org e-Print archive article cs.DS/0007043 (2000). Available at `http://arxiv.org`.

[24] Olariu, S., Overstreet, C. M., Wen, Z.: A mergeable double-ended priority queue. The Computer Journal **34**, 423–427 (1991).

[25] Rahman, M. Z., Chowdhury, R. A., Kaykobad, M.: Improvements in double ended priority queues. International Journal of Computer Mathematics **80**, 1121–1129 (2003).