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 $0, 1, \ldots, r-1$

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:

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

Redundant representation:

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

Skew representation:

$$n = \sum_{i=0}^{\lfloor \lg(n+1) \rfloor} d_i (2^{i+1}-1), \text{ where every } d_i \in \{0,1\}, \text{ except that the lowest}$$

non-zero digit d_i may also be 2.

Zeroless representation:

$$n = \sum_{i=0}^{k} d_i 2^i, \text{ where } k \in \{-1,0,\ldots,\lfloor \lg n \rfloor\} \text{ and } d_i \in \{1,2,3,4\} \text{ for all}$$

$i \in \{0,\ldots,k\}$. 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 "$\textcircled{x}$" that x is the most significant digit.

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

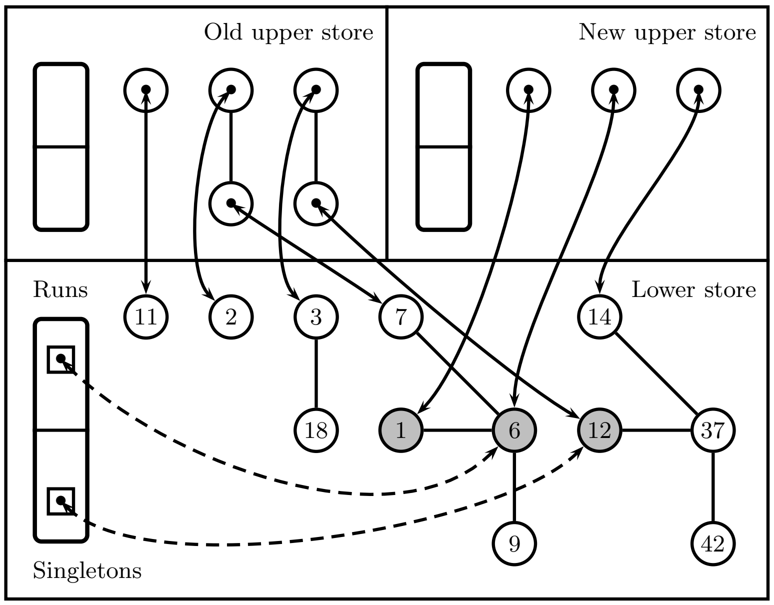


Fig. 1 A two-tier relaxed heap storing 12 integers. The relaxed binomial trees are drawn in schematic form and active nodes are drawn in grey. At the upper store a global rebuilding is in progress.

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.

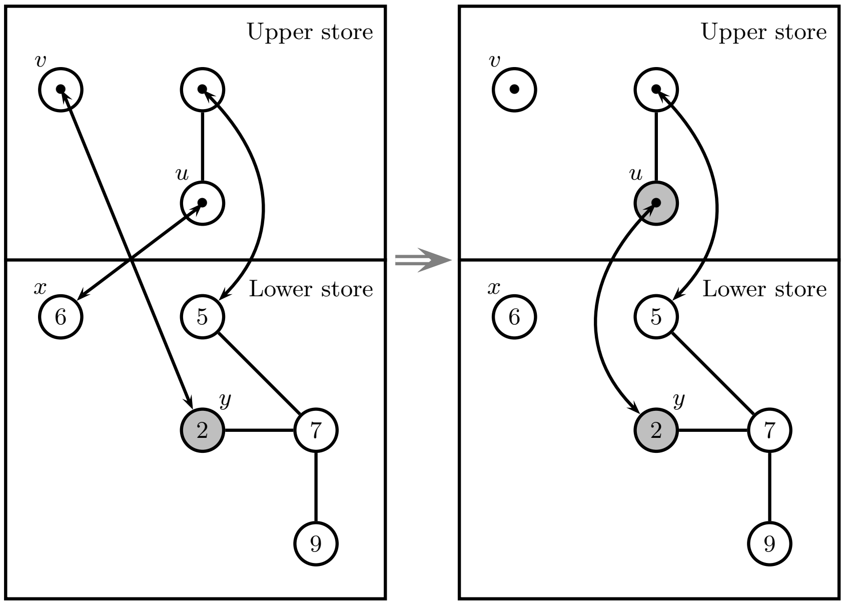
An insertion is carried out as described in Section 3, but *insert* requires three modifications in places where communication between the lower store and upper store is necessary. First, in each join the counterpart of the loser tree must be lazily deleted from the upper store. Second, in each split 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.

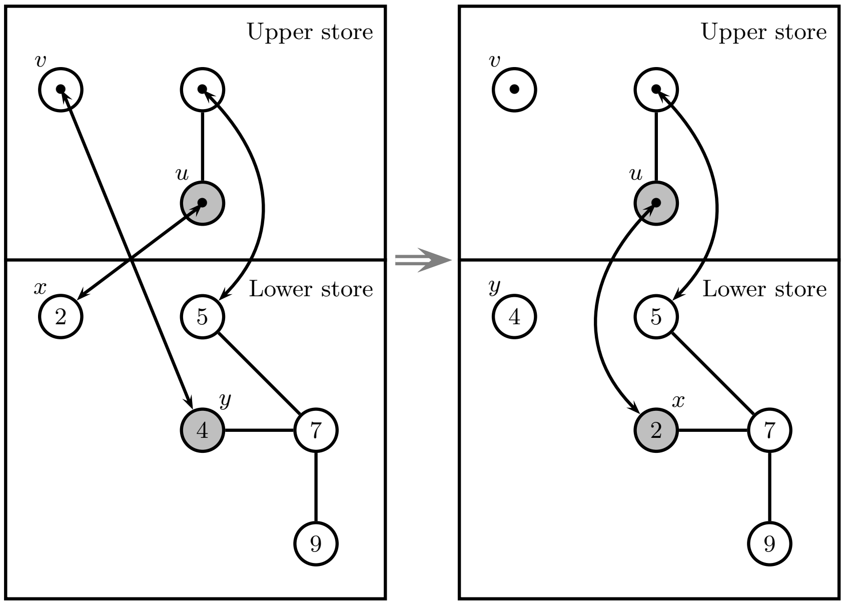


Fig. 2 Illustration of the syncronization between the upper store and the lower store when extracting an element from a two-tier relaxed heap. Only the trees worked with are drawn.

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 $O(1)$?

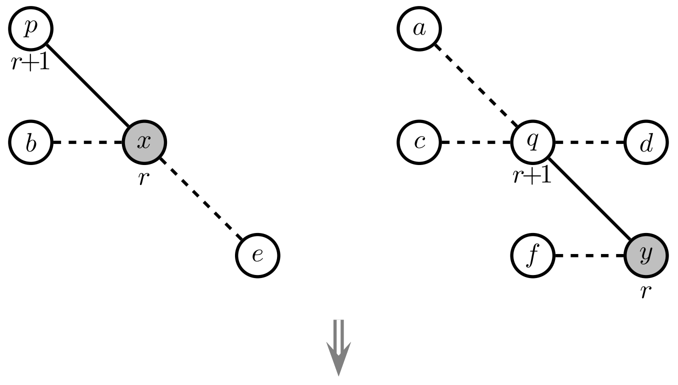
Significant simplifications are called for to obtain a practical data structure that supports *find-min*, *insert*, *extract*, *decrease*, *delete*, and *meld* at optimal or near-optimal worst-case cost. We leave the challenge of devising such a data structure for the reader.

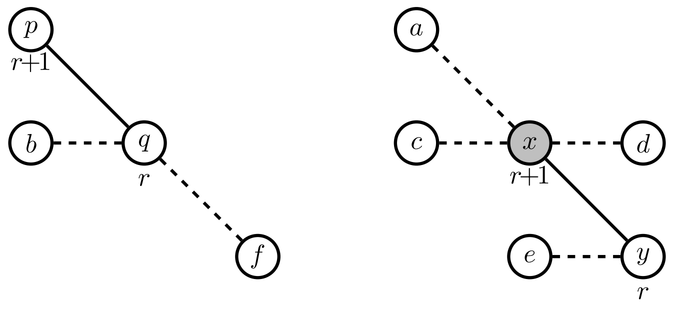
Acknowledgements We thank the anonymous referees for their constructive comments that improved the presentation of this paper.

Appendix

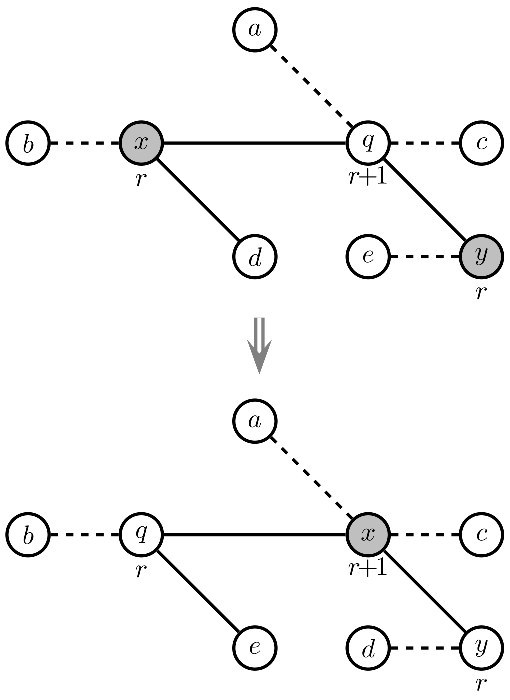
In this appendix a pictorial description of the transformations applied in a λ-reduction is given. In a singleton transformation two singletons x and y are given, and in a run transformation the last active node z of a run is given. In the following only the relevant nodes for each transformation are 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.

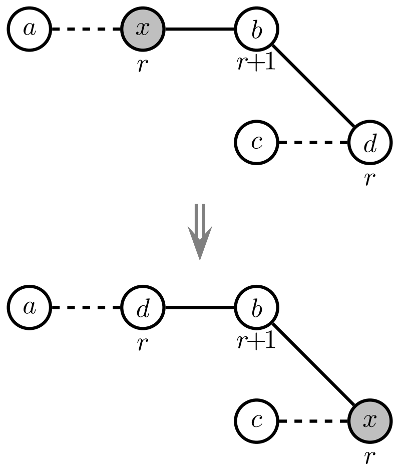



p
$r{+}1$
b
q
r
f
a
c
x
d
$r{+}1$
e
y
r

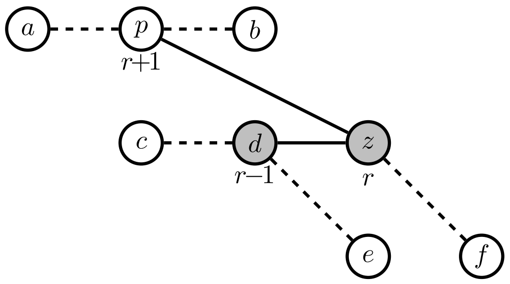
Singleton transformation II: y is the last child of its parent q and q is the right sibling of x. (The case where the parent of x is the right sibling of y is symmetric.) Assume that $element[x] \leq element[y]$. (The case where $element[x] > element[y]$ is similar.)

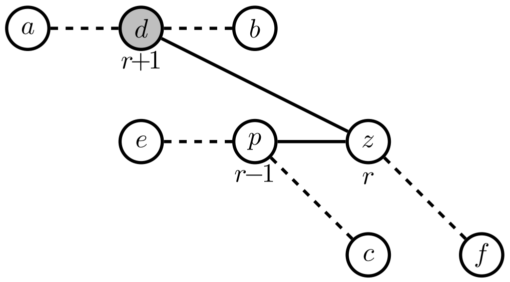


Singleton transformation III: x (or y) is not the last child of its parent, and the last child of its right sibling is not active.

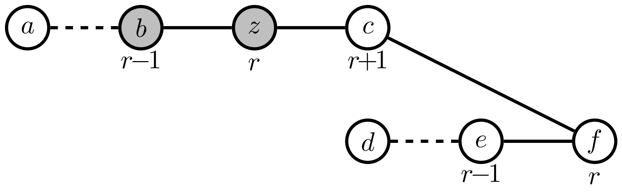


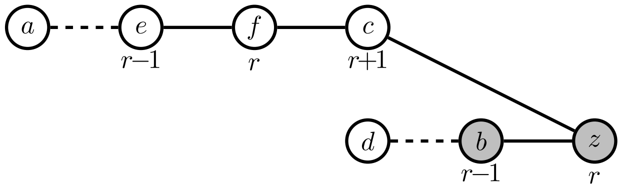
Run transformation I: z is the last child of its parent. Assume that $element[d] \le element[p]$ and $element[d] \le element[z]$. (There are three other cases which are similar.)





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)