SIAM J. COMPUT.
Vol. 40, No. 6, pp. 1463–1485

RANK-PAIRING HEAPS*

BERNHARD HAEUPLER†, SIDDHARTHA SEN‡, AND ROBERT E. TARJAN§

Abstract. We introduce the *rank-pairing heap*, an implementation of heaps that combines the asymptotic efficiency of Fibonacci heaps with much of the simplicity of pairing heaps. Other heap implementations that match the bounds of Fibonacci heaps do so by maintaining a balance condition on the trees representing the heap. In contrast to these structures but like pairing heaps, our trees can evolve to have arbitrary (unbalanced) structure. Also like pairing heaps, our structure requires at most one cut and no other restructuring per key decrease, in the worst case: the only changes that can cascade during a key decrease are changes in node ranks. Although our data structure is simple, its analysis is not.

Key words. algorithm, data structure, heap, priority queue, amortized analysis

AMS subject classifications. 68P05, 68P10, 68Q25

DOI. 10.1137/100785351

1. Introduction. A *meldable heap* (henceforth just a *heap*) is a data structure consisting of a set of items, each with a distinct real-valued key, that supports the following operations:

- *make-heap*: return a new, empty heap.
- *insert*(x, H): insert item x, with predefined key, into heap H.
- *find-min*(H): return the item in heap H of minimum key.
- *delete-min*(H): if heap H is not empty, delete from H the item of minimum key.
- *meld*(H_1, H_2): return a heap containing all the items in disjoint heaps H_1 and H_2, destroying H_1 and H_2.

Some applications of heaps need either or both of the following additional operations:

- *decrease-key*(x, Δ, H): decrease the key of item x in heap H by amount $\Delta > 0$, assuming that H is the unique heap containing x.
- *delete*(x, H): delete item x from heap H, assuming that H is the unique heap containing x.

We can allow equal keys by breaking ties using any total order of the items. We allow only binary comparisons of keys, and we study the amortized efficiency [34] of heap operations. To obtain a bound on amortized efficiency, we assign to each configuration of the data structure a nonnegative *potential*, initially zero. We define

*Received by the editors February 8, 2010; accepted for publication (in revised form) July 4, 2011; published electronically November 15, 2011. A preliminary version of some of these results appeared in a conference paper [19].

http://www.siam.org/journals/sicomp/40-6/78535.html

†Department of Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139 (haeupler@mit.edu).

‡Department of Computer Science, Princeton University, Princeton, NJ 08544 (sssix@cs.princeton.edu). This author's research was partially supported by NSF grants CCF-0830676 and CCF-0832797 and US-Israel Binational Science Foundation grant 2006204. The information contained herein does not necessarily reflect the opinion or policy of the federal government and no official endorsement should be inferred.

§Department of Computer Science, Princeton University, Princeton, NJ 08544 (ret@cs.princeton.edu), and HP Labs, Palo Alto, CA 94304. This author's research was partially supported by NSF grants CCF-0830676 and CCF-0832797 and US-Israel Binational Science Foundation grant 2006204. The information contained herein does not necessarily reflect the opinion or policy of the federal government and no official endorsement should be inferred.

the *amortized time* of an operation to be its actual time plus the change in potential it causes. Then, for any sequence of operations, the sum of the actual times is at most the sum of the amortized times.

Since n numbers can be sorted by doing n insertions into an initially empty heap followed by n delete-min operations, the classical $\Omega(n \log n)$ lower bound [26, p. 183] on the number of binary comparisons needed for sorting implies that either insertion or minimum deletion must take $\Omega(\log n)$ amortized time, where n is the number of items currently in the heap, which for simplicity in stating bounds we assume is at least two. We investigate simple data structures such that minimum deletion (or deletion of an arbitrary item if this operation is supported) takes $O(\log n)$ amortized time, and each of the other supported heap operations takes $O(1)$ amortized time. These bounds match the lower bound. (The logarithmic lower bound can be beaten in a model of computation that allows more powerful operations on keys than binary comparisons. See, e.g., [16, 20, 35, 36].)

Many heap implementations have been proposed over the years. We mention only those directly related to our work. The *binomial queue* of Vuillemin [37] supports all the heap operations in $O(\log n)$ worst-case time per operation. This structure performs quite well in practice [4]. Fredman and Tarjan [15] invented the *Fibonacci heap* specifically to support key decrease operations in $O(1)$ time, which allows efficient implementation of Dijkstra's shortest path algorithm [6, 15], Edmonds' minimum branching algorithm [9, 17], and certain minimum spanning tree algorithms [15, 17]. Fibonacci heaps support deletion of the minimum or of an arbitrary item in $O(\log n)$ amortized time and the other heap operations in $O(1)$ amortized time.

Several years after the introduction of Fibonacci heaps, Fredman et al. [14] introduced a related self-adjusting heap implementation, the *pairing heap*. Pairing heaps support all the heap operations in $O(\log n)$ amortized time. Fibonacci heaps do not perform well in practice, but pairing heaps do [27, 28]. Fredman et al. [14] conjectured that pairing heaps have the same amortized efficiency as Fibonacci heaps, in particular an $O(1)$ amortized time bound for key decrease. Despite empirical evidence supporting the conjecture [27, 31], Fredman [13] showed that it is not true: pairing heaps and related data structures that do not store subtree size information require $\Omega(\log \log n)$ amortized time per key decrease if the other operations are allowed only $O(\log n)$ amortized time. Whether pairing heaps meet this bound is open; the best upper bound on the amortized time per key decrease is $O(2^{2\sqrt{\lg \lg n}})$ [30][1] if the other operations are only allowed $O(\log n)$ amortized time.

These results motivated work to improve Fibonacci heaps and pairing heaps. Some of this work obtained better bounds, but at the cost of making the data structure more complicated. In particular, the bounds of Fibonacci heaps can be made worst-case. Run-relaxed heaps [7] achieve the bounds except for melding, which takes $O(\log n)$ time. Fat heaps [23] and trinomial heaps [32] are similar structures that achieve the same bounds somewhat more simply. (Although no melding algorithm is given in [32], it is easy to obtain one that runs in $O(\log n)$ time.) Two-tier relaxed heaps [12] are more complicated but achieve these bounds with two improvements: minimum deletion takes $\lg n + O(\log \log n)$ comparisons, thus achieving the best possible constant factor on comparisons; and melding takes $O(\log n')$ time, where n' is the number of items in the smaller of the heaps to be melded. The data structures of Brodal [2] and of Brodal and Okasaki [3] achieve the bounds of Fibonacci heaps except for key decrease, which takes $O(\log n)$ time in the worst case. A very compli-

[1] We denote by lg the base-two logarithm.

cated data structure of Brodal [2] achieves all the bounds in the worst case. Working in another direction, Elmasry [10] proposed an alternative to pairing heaps that does not store subtree size information but takes $O(\log \log n)$ amortized time for a key decrease, matching Fredman's lower bound. (Fredman's bound does not in fact apply to Elmasry's data structure because the structure does not satisfy certain technical restrictions in the bound.)

Working in yet a third direction, several authors proposed data structures with the same amortized efficiency as Fibonacci heaps but intended to be simpler. This is our goal as well. Peterson [29] gave a structure based on AVL trees. Høyer [21] gave several structures, including ones based on red-black trees, AVL trees, and a, b-trees. Takaoka [33] gave a structure based on 2, 3-trees. Høyer's simplest structure is one he calls a *one-step heap*. Kaplan and Tarjan [24] filled a lacuna in Høyer's presentation of one-step heaps and gave a related structure, the *thin heap*. Independently of our own work but concurrently, Elmasry [11] developed *violation heaps* and Chan [5] *quake heaps*. Violation heaps are similar to our heaps in that they avoid the cascading cuts of Fibonacci heaps, but they use a different form of cut that moves two subtrees instead of one, and they use a somewhat complicated rank-update method. Quake heaps use large-scale rebuilding, triggered by a subtree becoming sufficiently unbalanced.

Except for violation heaps, all of these structures have in common that the trees representing the heap have some kind of balance property that the heap operations must maintain. As a result, a key decrease can trigger an arbitrary amount of restructuring. Our main insight is the surprising fact that no balance condition is necessary: all that is needed is a way to control the size of the subtrees that are combined. Our new data structure, the *rank-pairing heap* or *rp-heap*, needs at most one (standard) cut and no other restructuring per key decrease. As in all the cited structures other than pairing heaps, we store a rank for each node. Ranks give lower bounds (but not upper bounds) on subtree sizes. Only trees whose roots have equal rank are combined. After a key decrease, rank changes (decreases) can cascade up the tree. But since there is only one cut per key decrease, appropriate sequences of key decreases can cause the trees representing the heap to evolve to have arbitrary, even completely unbalanced structure. Rank-pairing heaps have the same amortized efficiency as Fibonacci heaps and are, at least in our view, the simplest such structure so far proposed. (As noted above, violation heaps are similar to rp-heaps but use a more complicated form of cut and a more complicated rank-update rule.) Although rp-heaps are simple, their analysis is not.

2. Organization. The remainder of our paper consists of seven sections. We first review the common basis of binomial queues, Fibonacci heaps, pairing heaps, and all the related structures. Each of these structures can be viewed as a set of single-elimination tournaments. Section 3 discusses such tournaments and ways of representing them. Section 4 explores the design space of binomial queues. It presents three lazy versions of binomial queues: one old, *multipass*; and two new, *one-pass* and *one-tree*. Section 5 extends multipass binomial queues to support key decrease and arbitrary deletion, giving us the *rank-pairing heap*. The same extension applies to one-pass binomial queues. There are two types of rp-heaps, type 1 and type 2, which differ only in the rule obeyed by the node ranks. Type 2 obeys a weaker rank rule, which makes it slightly more complicated but simplifies its analysis and yields mostly smaller constant factors. Section 6 analyzes the amortized efficiency of both types. Section 7 presents a one-tree version of rp-heaps. The modification that makes the data structure into a single tree applies to either type 1 or type

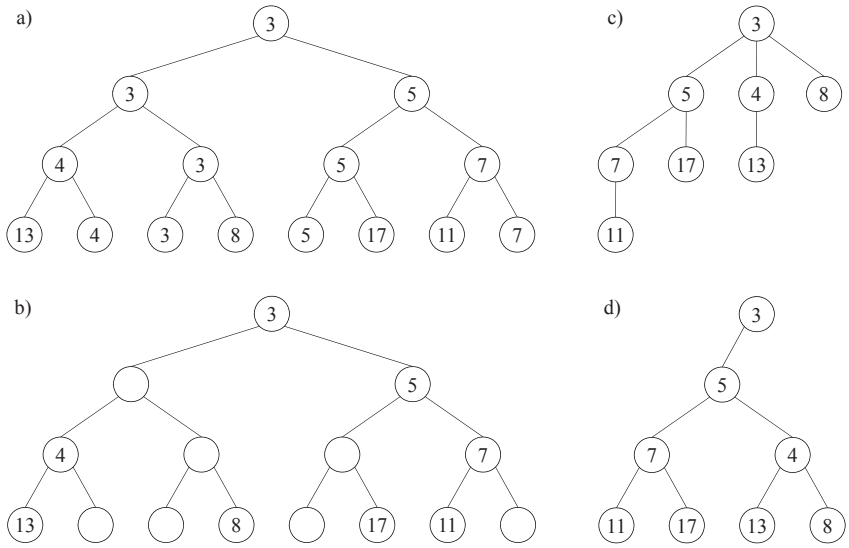

FIG. 3.1. *Four representations of a tournament:* (a) *full,* (b) *half-full,* (c) *heap-ordered, and* (d) *half-ordered.*

2 rp-heaps, and preserves the efficiency results of section 6. Section 8 shows that some even simpler ways of implementing key decrease do not yield the bounds of Fibonacci heaps. Section 9 gives our conclusions and mentions some open problems. A preliminary version of some of this work appeared in a conference paper [19].

3. Tournaments. The basis of many of the heap implementations mentioned in the introduction, as well as of our own, is the (single-elimination) tournament. A *tournament* is either empty, or consists of a single item, the *winner*, or is formed from two item-disjoint tournaments by *linking* them. To link two tournaments, combine their sets of items and compare the keys of their winners. The item of smaller key is the *winner* of the link and of the tournament formed by the link; the item of larger key is the *loser* of the link. Building an n-item tournament takes $n-1$ comparisons; the winner of the tournament is the item of minimum key.

There are (at least) four equivalent representations of a tournament. (See Figure 3.1.) The *full representation* is a full binary tree with one leaf per item and one nonleaf per link. Each nonleaf contains the winner of the corresponding link. Thus the nodes containing a given item form a path in the tree, consisting of a leaf and the nonleaves corresponding to the links won by the item. The tree is *heap-ordered*: the item in a node has minimum key among the items in the descendants of the node.

We obtain the *half-full representation* from the full representation by removing every item from all but the highest node containing it. This representation is a binary heap-ordered tree in which the root is full, each parent has one full and one empty child, and each item occurs in one (full) node.

Both the full and the half-full representation use $2n-1$ nodes to represent an n-item tournament. The *heap-ordered representation* uses only n nodes. It is an ordered tree in which the items are the nodes and the children of an item are those that lost comparisons to it, most-recent comparison first. The tree is heap-ordered but not in general binary. Many of the heap implementations mentioned in the introduction were originally presented in the heap-ordered representation.

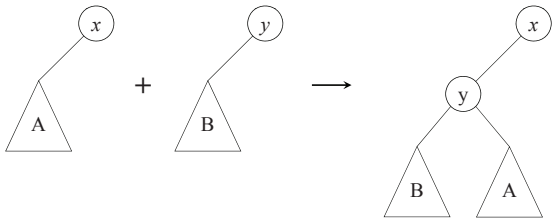

FIG. 3.2. *A link of two half trees with roots x and y, x having smaller key.*

We obtain the *half-ordered representation* from the heap-ordered representation by using the binary tree representation [25, pp. 332–346] of a tree: the left child of an item in the half-ordered representation is its first child in the heap-ordered representation, and the right child of an item in the half-ordered representation is its next sibling in the heap-ordered representation. The resulting tree is a *half tree*: a binary tree whose root has a missing right subtree. The half tree is *half-ordered*: the key of any item is less than that of all items in its left subtree. In a half-ordered half tree we define the *ordered ancestor* of a node x other than the root to be the parent of the nearest ancestor of x that is a left child. This is exactly the parent of x in the heap-ordered representation.

The half-ordered representation appears in the original paper on pairing heaps [14]. Peterson [29] and Dutton [8] each independently reinvented it, unfortunately swapping left and right. We shall use the half-ordered representation in its original form, which is consistent with Knuth's description [25]. Henceforth all our trees are binary and half-ordered. The half-ordered representation has two advantages over the heap-ordered representation: it is closer to an actual implementation, and it unifies the treatment of key decrease. All four representations of tournaments are fully equivalent, however, and all of our results, as well as all previous ones, can be presented in any of them, if one does an appropriate mapping of pointers.

We represent a binary tree by storing with each node x pointers to its left and right children, left(x) and right(x), respectively. The *right spine* of a node in a binary tree is the path from the node through right children to a missing node. Linking takes the following form on half trees (see Figure 3.2): compare the keys of the roots. If x and y are the roots of smaller and larger key, respectively, detach the old left subtree of x and make it the right subtree of y; then make the tree rooted at y the new left subtree of x. A link takes O(1) time.

4. Lazy binomial queues. When translated into the half-ordered representation, many of the heap implementations mentioned in the introduction, and ours, are extensions of the following generic implementation. A heap consists of a set of half trees whose nodes are the items in the heap, represented by a singly-linked circular list of the tree roots. Access to the root list is by a pointer to the root of minimum key, which we call the *min-root*. Do the various heap operations, excluding key decrease and arbitrary deletion, as follows. To find the minimum in a heap, return the min-root. To make a heap, create an empty list of roots. To insert an item, make it a one-node half tree, insert it into the list of roots after the min-root, and make it the min-root if its key is smaller than that of the old min-root. To meld two heaps, catenate their lists of roots, and make the old min-root of smaller key the min-root of the new list. To delete the minimum, disassemble the half tree rooted at the min-root x, as follows. Let y be the left child of x. Delete x, and cut each edge on the right

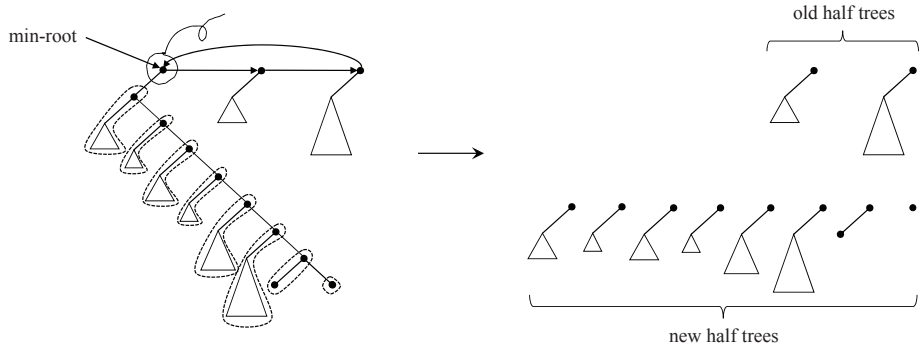

FIG. 4.1. *Tree disassembly during a minimum deletion. Each node on the right spine of the left child of the min-root becomes the root of a new half tree.*

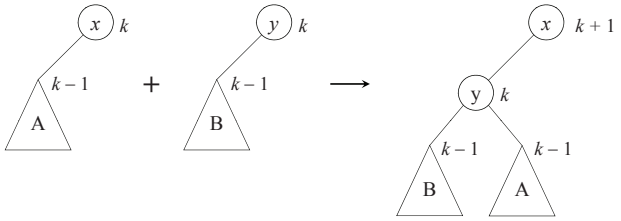

FIG. 4.2. *A link of two half trees with roots x and y, x having smaller key. Ranks are to the right of nodes.*

spine of y. This makes each node on the right spine of y the root of a new half tree, containing itself and its left subtree. (See Figure 4.1.) Add the new half trees to the remaining half trees. Find the root of minimum key, and make it the min-root. Additionally, after each heap operation, do zero or more links of half trees to reduce their number. With this implementation, the nodes that lost links to a given node x are exactly those on the right spine of the left child of x.

This data structure is efficient only if the links are done carefully. In pairing heaps [14], of which there are several forms, all the links are of half trees whose roots are adjacent in the list of roots. This method is not efficient: Fredman [13] showed that to obtain the bounds of Fibonacci heaps it is necessary (subject to certain technical requirements of the proof) to do many links of half trees of related sizes. Except for pairing heaps, all previous versions of this data structure use nonnegative *node ranks* as a proxy for size. The simplest way to use ranks is as follows. Let the rank of a half tree be the rank of its root. Give a newly inserted item a rank of zero. Link two half trees only if they are of equal rank; after the link, increase the rank of the winning root by one; do not change the loser's rank. (See Figure 4.2.)

If all links are done this way, every half tree ever in a heap is *perfect*: it consists of a root whose left subtree is a perfect binary tree, each child has rank one less than that of its parent, and the tree contains 2^k nodes, where k is its rank. Thus the maximum rank is at most $\lg n$. The resulting data structure is the *binomial queue*, so-called because in the heap-ordered representation the number of nodes of depth d in a tree of rank r is the binomial coefficient $\binom{r}{d}$. (See Figure 4.3.)

In the original version of binomial queues [37], links are done eagerly to maintain the invariant that a heap contains at most one root per rank. This gives an $O(\log n)$

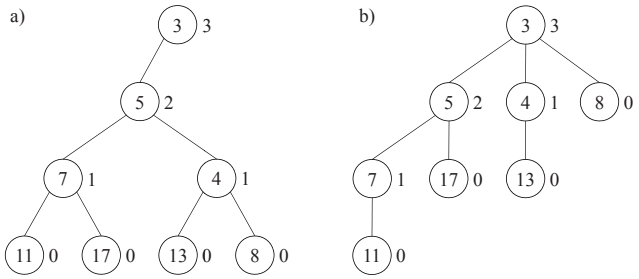

FIG. 4.3. *The* (a) *half-ordered and* (b) *heap-ordered representations of a tree in a binomial queue. Ranks are to the right of nodes.*

worst-case time bound for insert, meld, and delete-min. Doing links lazily, specifically only during minimum deletions, gives better amortized efficiency. One method, used in Fibonacci heaps and all the other similar structures, is to do as many links as possible after a minimum deletion, leaving at most one root per rank. This method, which we call *multipass linking*, works for us as well. A lazier alternative, which we call *one-pass linking*, also works: after a minimum deletion, form a maximum number of pairs of half trees of equal rank, and link these pairs but no others. This linking method resembles the one used in the lazy variant of pairing heaps [14]. We call a binomial queue with multipass linking a *multipass binomial queue* and one with one-pass liking a *one-pass binomial queue*.

To implement one-pass or multipass linking, maintain a set of buckets, one per rank. During a minimum deletion, process the half trees, beginning with those formed by the disassembly and finishing with the remaining ones. To process a half tree, add it to the bucket for its rank if this bucket is empty. If not, link the half tree with the half tree in the bucket, leaving the bucket empty; add the new half tree to the list of trees in the new heap if linking is one-pass, or to the bucket of one higher rank if the linking is multipass. (See Figure 4.4.) Throughout the processing, keep track of the nonempty buckets. Once all the half trees have been processed, add any half tree still in a bucket to the list of trees in the new heap, leaving all the buckets empty.

Although this implementation of linking seems to require an array, it is easy to implement it in a pointer model, as observed by Fredman and Tarjan [15]: Construct a doubly-linked list of *rank nodes*, one for each possible rank, from zero to the maximum rank. For each i, rank node i points to rank nodes $i+1$ and $i-1$, and to the bucket for rank i. Each node points to the rank node for its rank.

To analyze one-pass and multipass binomial queues, we define the potential of a heap to be twice the number of half trees.

THEOREM 4.1. *The amortized time for an operation on a one-pass or multipass binomial queue is* $O(1)$ *for a make-heap, find-min, insert, or meld, and* $O(\log n)$ *for a delete-min.*

Proof. A make-heap, find-min, insert, or meld takes O(1) actual time. Of these operations, only an insert increases the potential, by two. Thus each of these operations takes O(1) amortized time. Consider a minimum deletion. The following argument applies to both one-pass and multipass linking. Disassembling the half tree rooted at the node of minimum key increases the number of half trees by at most $\lg n$ and hence the potential by at most $2 \lg n$. Let h be the number of half trees after the disassembly. The entire minimum deletion takes h key comparisons and $O(h+1)$ time. Scale this time to be at most $h+O(1)$. (This is equivalent to multiplying the

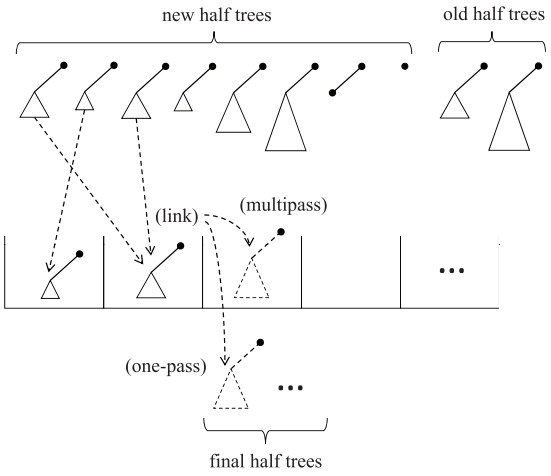

FIG. 4.4. *Linking during a minimum deletion using a set of buckets, one per rank. The new half trees from the disassembly are processed first, followed by the remaining half trees. After a link, the resulting half tree is added to the output list of the new heap if linking is one-pass, or to the bucket of one higher rank if linking is multipass.*

potential by a constant factor.) Each link after the disassembly reduces the potential by two. At most $\lg n + 1$ half trees do not participate in a link, so there are at least $(h - \lg n - 1)/2$ links. The minimum deletion thus increases the potential by at most $\lg n - h + 1$, giving an amortized time of $O(\log n)$ and at most $3 \lg n$ amortized key comparisons. □

We have included one-pass linking here to illustrate the range of efficient binomial queue implementations. One-pass linking is lazier but produces longer root lists than multipass linking. This slows down minimum deletion. We can make this observation quantitative: with a potential function equal to the number of half trees, an argument such as that in the proof of Theorem 4.1 shows that multipass linking does at most $2 \lg n$ amortized key comparisons per minimum deletion, fewer than one-pass linking by a factor of 3/2.

A method that may be better than both one-pass and multipass linking is to maintain the heap as a single half tree, avoiding the overhead of root lists entirely. Such a representation encodes all key comparisons in the data structure, whereas in a multitree representation the key comparisons done to maintain the min-root are not encoded in the structure, and their outcomes are lost after each minimum deletion. We conclude this section by presenting a one-tree version of binomial queues.

Maintaining the heap as a single half tree requires linking half trees of different ranks. We call such a link *unfair*; we call a link of two half trees of equal rank *fair*. After an unfair link, leave the rank of both the winner and the loser unchanged. Unfair links do not adversely affect the amortized efficiency of the data structure, if one does as few of them as possible.

A *one-tree binomial queue* consists of a single half tree. To find the minimum, return the root. To make a heap, create an empty half tree. To insert a new item, make it into a one-node half tree of rank zero and link it with the existing half tree. To meld two heaps, link their half trees. To delete the minimum, delete the root and disassemble the half tree into one half tree rooted at each node on the right spine of the old left child of the deleted root. Process the new half trees by

doing multipass linking. Once all half trees are in buckets, remove them from the buckets and link them in any order (by fair or unfair links) until only one half tree remains.

LEMMA 4.2. *A one-tree binomial queue of rank k contains at least 2^k nodes. Hence $k \leq \lg n$.*

Proof. We prove the lemma by induction on the number of links and half-tree disassemblies. A new one-node half tree has rank zero and satisfies the lemma. A fair link combines two half trees of equal rank, say k, into one half tree of rank $k+1$. By the induction hypothesis each component half tree contains at least 2^k nodes, so the combined half tree contains at least 2^{k+1} nodes and satisfies the lemma. An unfair link combines two half trees of different ranks, say j and k with $j < k$, into one half tree of rank at most k. By the induction hypothesis, the component half tree of rank k contains at least 2^k nodes, so the combined half tree satisfies the lemma. A half-tree disassembly undoes all the links won by the root and deletes the root. Since the resulting half trees satisfied the lemma when they were created, they satisfy the lemma after the disassembly. □

To analyze one-tree binomial queues, we define the potential of a node to be zero if it is the loser of a fair link or one otherwise (it is a root or the loser of an unfair link); we define the potential of a heap to be the sum of the potentials of its nodes. A minimum deletion undoes all the links won by the node deleted. The loser of each such link becomes a root and is no longer a loser; thus its potential increases by one if the link was fair and does not change if the link was unfair.

THEOREM 4.3. *The amortized time for an operation on a one-tree binomial queue is $O(1)$ for a make-heap, find-min, insert, or meld, and $O(\log n)$ for a delete-min.*

Proof. The analysis of find-min, make-heap, insert, and meld is just like that of multipass binomial queues except that each insert or meld does a link. Each such link takes O(1) time and does not increase the potential. Consider a minimum deletion. Disassembling the half tree undoes all the links won by the root. This increases the potential by one for each fair link won by the root. Each such link increased the rank of the root when it took place. By Lemma 4.2 there were at most $\lg n$ such links, so the disassembly increases the potential by at most $\lg n$. Let h be the number of half trees after the disassembly. The entire minimum deletion takes $h-1$ key comparisons and $O(h+1)$ time. Scale this time to be at most $h+O(1)$. Each fair link after the disassembly reduces the potential by one; each unfair link does not change it. There are at most $\lg n$ unfair links, so there are at least $h - \lg n - 1$ fair ones. Hence the minimum deletion increases the potential by at most $2\lg n - h + 1$, giving an amortized time of $O(\log n)$ and at most $2\lg n$ amortized key comparisons. □

5. Rank-pairing heaps. Our main goal is to implement key decrease so that it takes O(1) amortized time. Once key decrease is supported, one can delete an arbitrary item by decreasing its key to $-\infty$ and doing a minimum deletion. A parameter of both key decrease and arbitrary deletion is the heap containing the given item. If the application does not provide this information and melds occur, one needs a separate disjoint set data structure to maintain the partition of items into heaps. With such a data structure, the time to find the heap containing a given item is small but not O(1) [22].

We shall extend multipass binomial queues to support key decrease. This extension also works for one-pass binomial queues. We call the resulting data structure the *rank-pairing heap*. We develop two types of rp-heaps: type 1, which is simpler but harder to analyze and has larger constant factors in the time bounds, and type 2, a

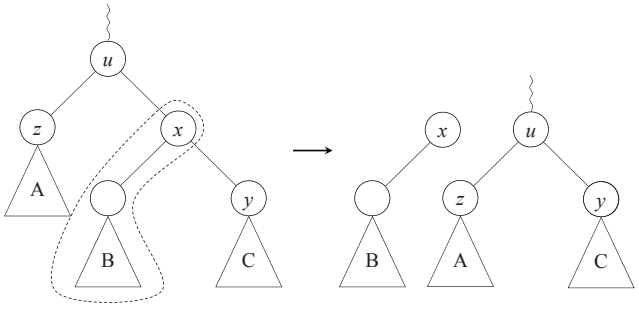

FIG. 5.1. *Restructuring during a key decrease.*

relaxed version that is easier to analyze and has smaller constant factors in the time bounds.

In order to implement key decrease, we add parent pointers to the half trees, so that there are three pointers per node instead of two. As observed by Fredman et al. [14], two pointers per node suffice: each node points to its left child, or to its right child if it has no left child, and to its right sibling, or to its parent if it has no right sibling. This alternative representation trades time for space. If a node has only one child, we view it as the left child if it has larger key or as the right child if not; this avoids the need for an extra bit to disambiguate these possibilities. This can convert right children into left children, but it affects neither the correctness nor the analysis of the data structure. The same idea applies to pairing heaps [14].

Once the data structure supports parental access, we can decrease the key of item x in heap H as follows. (See Figure 5.1.) Reduce the key of x. If x is not a root, then x may now violate half order. To restore half order, create a new half tree rooted at x by detaching the subtrees rooted at x and at $y = \text{right}(x)$, reattaching the subtree rooted at y in place of the original subtree rooted at x, and adding x to the list of roots. We call this a *cut* at x. Whether or not x was originally a root, make it the min-root if its key is now minimum.

Remark. If x is not originally a root, there is no way in our representation to test in O(1) time whether decreasing the key of x has violated half order: such a test requires access to the ordered ancestor of x. Thus we make x a root whether or not a violation occurs.

This implementation is correct, but it destroys the efficiency of the data structure, as we show in section 8: there are arbitrarily long sequences of operations that take $\Omega(n)$ time per operation. The trouble is that a key decrease can remove an arbitrary half tree, and a sequence of such removals can produce a half tree whose rank is $\omega(\log n)$. To preserve efficiency, we need a way to guarantee that the ranks remain $O(\log n)$, one that takes only O(1) amortized time per key decrease.

In Fibonacci heaps, the solution is to do a sequence of cuts after each key decrease, but only O(1) amortized per decrease. These cuts occur along a path of ancestors in the heap-ordered representation. Since the parent of a node x in the heap-ordered representation is its ordered ancestor in the half-ordered representation, implementation of this method requires an additional set of pointers, to ordered ancestors, which is one reason Fibonacci heaps do not perform well in practice.

There are many other ways to accomplish the same objective: adapt a known balanced tree structure, such as AVL trees [1] or red-black trees [18]; devise a new

balance rule, as in Høyer's one-step heaps [21] (thick heaps [24]), or thin heaps [24]; or do more-global rebuilding, as in quake heaps [5]. Another approach, used in relaxed heaps [7], is to allow violations of half order. Then a key decrease does not require immediate action; it just creates one more violation. To preserve efficiency, the set of violations must be controlled in some way, which requires periodic restructuring to reduce the set of violations.

All of these methods have one thing in common: they do extra restructuring to maintain some balance condition. Surprisingly, no such balance condition is needed: it suffices just to update ranks, in particular to decrease the ranks of certain ancestors of the node x whose key decreases. The only restructuring is the cut at x. A sequence of key decreases can create half trees of arbitrary structure, but ranks remain logarithmic, which preserves efficiency.

A little terminology helps the presentation. We denote by $p(x)$ and $r(x)$ the parent and rank of node x, respectively. We adopt the convention that the rank of a missing child is -1. If x is a child, its *rank difference* is $\Delta r(x) = r(p(x)) - r(x)$. A child of rank difference i is an *i-child*; a root whose left child is an i-child is an *i-node*; a nonroot whose children are an i-child and a j-child is an *i,j-node*. These definitions apply even if the left child of a root, or either or both children of a nonroot, are missing. The definition of an i,j-node does not distinguish between its left and right child. In a binomial queue other than the one-tree version, every root is a 1-node and every nonroot is a 1,1-node. We shall relax the second half of this invariant.

Our key observation is that each node has a number of descendants at least exponential in its rank even if we allow $0,i$-nodes for arbitrary i in addition to 1,1-nodes. With this in mind, we introduce the *type-1 rank rule*: every root is a 1-node and every child is a 1,1-node or a $0,i$-node for some $i > 0$ (possibly different for each node). A *type-1 rp-heap* is a set of heap-ordered half trees whose nodes have ranks that obey the type-1 rank rule.

Ranks give an exponential lower bound (but not an upper bound) on subtree sizes.

LEMMA 5.1. *In a type-1 rp-heap, every node of rank k has at least 2^k descendants including itself, at least $2^{k+1} - 1$ if it is a child. Hence $k \leq \lg n$.*

Proof. The second part of the lemma implies the first and third parts. We prove the second part by induction on the height of a node. A leaf has rank zero and satisfies the second part. Let x be a child of rank k whose children satisfy the second part. If x is a $0,i$-node, its 0-child has $2^{k+1} - 1$ descendants by the induction hypothesis; so does x. If x is a 1,1-node, x has $2(2^k - 1) + 1 = 2^{k+1} - 1$ descendants by the induction hypothesis. □

The operations make-heap, find-min, insert, meld, and delete-min are exactly the same on type-1 rp-heaps as on multipass binomial queues, except for one change in minimum deletion: during the half-tree disassembly, give each new root a rank that is one greater than that of its left child. Since every link is fair, each link preserves the rank rule: the loser becomes a 1,1-node.

To decrease the key of a node x, proceed as follows (see Figure 5.2): Reduce the key of x. If x is a root, make it the min-root if its key is now minimum. If x is not a root, detach the subtrees rooted at x and at its right child y, reattach the subtree rooted at y in place of the one rooted at x, and add x to the list of roots, making it the min-root if its key is minimum. Finish by restoring the rank rule: make the rank of x one greater than that of its left child; and, starting at the new parent of y, walk up through ancestors, reducing their ranks to obey the rank rule, until reaching the root or reaching a node whose rank needs no reduction. To do the rank reductions,

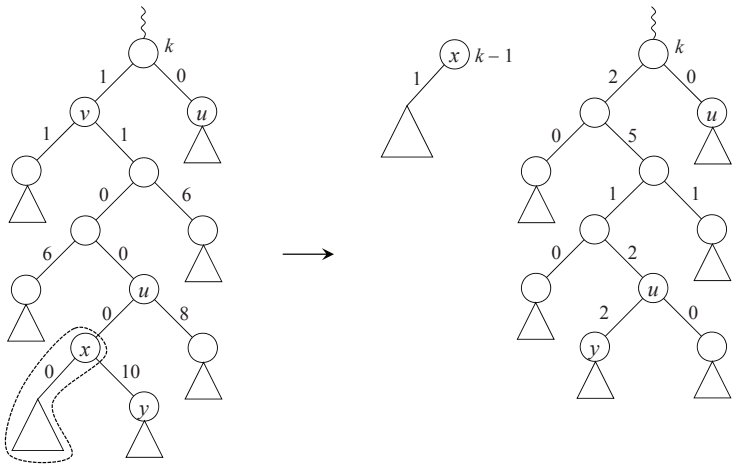

FIG. 5.2. *Key decrease in a type-*1 *rp-heap. Numbers on edges are rank differences. When* x *becomes a root, its rank becomes that of its left child plus* 1. *Each node on the path from* u *to* v *decreases in rank.*

let $u = p(y)$ and repeat the following step until it stops:

*Type-*1 *rank-reduction step*: If u is a root, set $r(u) = r(\text{left}(u))+1$ and stop. Otherwise, let v and w be the children of u. Let $k = \max\{r(v), r(w)\}$ if $r(v) \neq r(w)$, $k = r(v)+1$ if $r(v) = r(w)$. If $k \geq r(u)$, stop. Otherwise, let $r(u) = k$ and $u = p(u)$.

Remark. In a multipass rp-heap, $k \leq r(u)$ in every rank-reduction step. This is not true in the one-tree variant of rp-heaps that we develop in section 7.

LEMMA 5.2. *The rank-reduction process restores the rank rule.*

Proof. Since all rank differences are nonnegative, $r(y) \leq r(x)$ before the key decrease. If $r(y) < r(x)$, replacing x by y may cause $p(y)$, but only $p(y)$, to violate the rank rule. If u violates the rank rule before a rank-reduction step, the step reduces its rank to make it obey the rule. This may cause $p(u)$, but only $p(u)$, to violate the rule. An induction on the number of steps gives the lemma. □

Before analyzing type-1 rp-heaps, we introduce a relaxed version, obeying the *type-*2 *rank rule*: every root is a 1-node and every child is a 1,1-node, a 1,2-node, or a $0, i$-node for some $i > 1$ (possibly different for each node). A *type-*2 *rp-heap* is a set of heap-ordered half trees whose nodes have ranks that obey the type-2 rank rule.

The heap operations on type-2 rp-heaps are exactly the same as on type-1 rp-heaps except that the rank-reduction process restores the type-2 rule by using the following step in place of the type-1 step:

*Type-*2 *rank-reduction step*: If u is a root, set $r(u) = r(\text{left}(u))+1$ and stop. Otherwise, let v and w be the children of u. Let $k = \max\{r(v), r(w)\}$ if $|r(v) - r(w)| > 1$, $k = \max\{r(v), r(w)\} + 1$ if $|r(v) - r(w)| \leq 1$. If $k \geq r(u)$, stop. Otherwise, let $r(u) = k$ and $u = p(u)$.

Lemma 5.2 holds for type-2 rank reduction by the same proof. In either type of rank reduction, each successive rank decrease is by the same or a smaller amount, and in a multipass rp-heap $k \leq r(u)$ in every rank-reduction step, although this is not true in the variant we develop in section 7.

The rank bound for type-2 rp-heaps is bigger by a constant factor than that for type-1 heaps, but is the same as that for Fibonacci heaps. We denote by F_k the kth Fibonacci number, defined by the recurrence $F_0 = 0$, $F_1 = 1$, $F_k = F_{k-1} + F_{k-2}$ for $k > 1$. We denote by ϕ the golden ratio, $(1 + \sqrt{5})/2$.

LEMMA 5.3. *In a type-2 rp-heap, every node of rank k has at least $F_{k+2} \geq \phi^k$ descendants including itself, at least $F_{k+3} - 1$ if it is a child. Hence $k \leq \log_\phi n$.*

Proof. The second part of the lemma implies the first and third parts, given the known [25, p. 18] inequality $F_{k+2} \geq \phi^k$. We prove the second part by induction on the height of a node. A missing node satisfies the second part; so does a leaf. Let x be a child of rank k whose children satisfy the second part. If x is a $0, i$-node, then the 0-child of x has at least $F_{k+3} - 1$ descendants by the induction hypothesis; so does x. If x is a 1,1- or 1,2-node, then x has at least $F_{k+1} - 1 + F_{k+2} - 1 + 1 = F_{k+3} - 1$ descendants by the induction hypothesis, since $F_{k+1} \leq F_{k+2}$. □

6. Amortized efficiency of rank-pairing heaps. In this section we analyze the efficiency of rp-heaps. We begin by analyzing type-2 heaps, which is easier than analyzing type-1 heaps. We use a potential function argument. Choosing a potential function is more an art than a science. Our choice resulted from a process of trial and error, refinement and simplification: even for type-2 rp-heaps, the choice is a bit delicate. In hindsight, we can provide some intuition for our choice. Recall from section 5 that a missing child has rank -1 by convention. The nonexistent right child of a root is not regarded as missing. We need to amortize two kinds of O(1)-time steps: links, each of which converts a root into a 1,1-node, and rank-reduction steps. It is natural to try a potential function that is a sum of node potentials. If we assign one unit of potential to each root and zero to each 1,1-node, then each link reduces the potential by one. This pays for links. To pay for rank-reduction steps, we observe that if the rank of a nonroot node decreases by k, the rank difference of each of its two children decreases by k, and its own rank difference increases by k, so the sum of these three rank differences decreases by k. Thus if we assign a potential of $i + j$ to each i, j-child, then each rank-reduction step reduces the potential by at least one. This remains true if we assign a potential of $i + j + c$ to each i, j-child, where c is any constant. Choosing $c = -2$ gives each 1,1-node zero potential, consistent with the assignment needed to pay for links.

Unfortunately, there is one more constraint: the half-tree disassembly triggered by a minimum deletion can convert an arbitrarily large number of 0,2-nodes into roots, each of which needs one unit of potential. For each such node to have the needed potential before the disassembly, we must choose $c \geq -1$, which produces a seemingly unresolvable circularity. Fortunately, there is a way to break this circularity. As we prove below, at most one 1,1-node can decrease in rank during each key decrease. This allows us to reduce the potential of each 1,1-node by one, since the extra unit needed when it changes state can be charged to the corresponding key decrease. Thus we choose $c = -1$ but give each 1,1-node a potential of zero.

It remains to prove that this works. We assign a potential to every heap equal to the sum of its node potentials. The potential of a node is the sum of the rank differences of its children, minus one if it is a child but not a 1,1-node or minus two if it is a 1,1-node. This definition applies not only to nodes that obey the rank rule but also to the node that violates the rank rule in the middle of a key decrease. That is, the potential of a root with an i-child is i, the potential of a 1,1-node is 0, and the potential of an i, j-node other than a 1,1-node is $i + j - 1$.

THEOREM 6.1. *The amortized time for an operation on a type-2 rp-heap is $O(1)$*

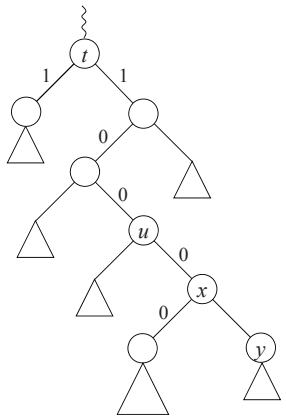

FIG. 6.1. *Rank reductions during a key decrease in a type-2 rp-heap. Node t is the bottommost $1,1$-node among those whose ranks decrease. Node y can have arbitrarily large rank difference. Nodes on the path from u to t can decrease in rank by an arbitrarily large amount, but t and any node above t can only decrease in rank by 1. Hence the next $1,1$-node above t cannot decrease in rank, but only become a $1,2$-node.*

for a make-heap, find-min, insert, meld, or decrease-key, and $O(\log n)$ for a delete-min.

Proof. A make-heap, find-min, or meld operation takes O(1) actual time and does not change the potential; an insertion takes O(1) time and increases the potential by one. Hence each of these operations takes O(1) amortized time. Consider a minimum deletion. Each new root created by the disassembly has the potential it needs (one unit) unless it was previously a 1,1-node. At most one 1,1-node can become a new root for each rank less than that of the deleted root. By Lemma 5.3 there are at most $\log_\phi n$ such 1,1-nodes. Thus the disassembly increases the potential by at most $\log_\phi n$. Let h be the number of half trees after the disassembly. The entire minimum deletion takes $h-1$ key comparisons and $O(h+1)$ time. Scale this time to be at most $h+O(1)$. Each link after the disassembly converts a root into a 1,1-node, which reduces the potential by one. At most $\log_\phi n + 1$ half trees remain after all the links, so there are at least $h - \log_\phi n - 1$ links. The amortized time of the minimum deletion is thus at most $\log_\phi n + h + O(1) - h + \log_\phi n + 1 = 2\log_\phi n + O(1)$, and the amortized number of key comparisons is at most $2\log_\phi n$.

The novel part of the analysis is that of key decrease. Consider decreasing the key of a node x. If x is a root, the key decrease takes O(1) actual time and does not change the potential. Suppose that x is not a root. Let y be the right child of x and u the parent of x. Consider the half tree containing x before the subtree rooted at x is detached. Let z be the last (topmost) nonroot node whose rank decreases as a result of the key decrease. There can be at most one 1,1-node on the path from x to z, including x and z. Indeed, let t be the lowest 1,1-node on this path. If $t = x$, then replacement of x as a child of u by y gives u a new child of rank difference one higher than that of its old child. If on the other hand $t \neq x$, t can decrease in rank by at most one. In either case, every subsequent rank reduction (after the replacement of x by y or after the rank reduction of t) is by exactly one. If a child of a 1,1-node decreases in rank by one, then the node becomes a 1,2-node, and the rank-reduction process stops. Thus the path from x to z cannot contain a second 1,1-node. See Figure 6.1.

Add one unit of potential to the 1,1-node, if any, on the path from x to z, and add two units to the potential of x. Now every nonroot node on the path from x to z has potential equal to the sum of the rank differences of its children minus one, and x has two extra units of potential. Detach the half tree rooted at x, replace it by the subtree rooted at y, and give to u all but one unit of the potential of x. Now x is a root, it has the one unit of potential it needs (its rank is now one greater than that of its left child), and every node on the path from u to z has potential equal to the sum of the rank differences of its children, minus one. Finally, do the successive rank reductions. Reducing the rank of a nonroot node by k reduces its potential by at least $2k$ ($2k+1$ if it becomes a 1,1-node) and increases the potential of its parent by k, unless the parent is a 1,1-node that becomes a 1,2-node. In this case the potential of the node decreases by two, the potential of its parent increases by two, and this is the last rank reduction. Decreasing the rank of the root by k decreases its potential by k without affecting the potential of any other node. Thus if we charge one unit of time per rank reduction, the amortized time for a rank reduction other than the last is at most zero, and that of the last is at most one. The amortized time for all the rank reductions is at most four (three units of added potential plus at most one for the last rank reduction). We conclude that the key decrease takes $O(1)$ amortized time. □

Now we analyze type-1 rp-heaps. In contrast to our analysis of type-2 rp-heaps, our analysis of type-1 rp-heaps requires a restriction on the links done during minimum deletion. During such a deletion, we partition the half trees into two kinds, *new* and *old*: those formed by the disassembly of the half tree rooted at the min-root are new; the others in the heap are old. We repeatedly link pairs of new half trees, making each tree resulting from such a link old. Once there are no two new half trees of the same rank, we do arbitrary links until no two half trees have the same rank. We call this *restricted multipass linking*. A simple way to implement such linking is to do one-pass linking of the new half trees and then do (standard) multipass linking of all the remaining half trees. We do not know if our analysis can be extended to arbitrary multipass linking.

We need a more complicated potential function than the one for type-2 rp-heaps. A single key decrease can convert an arbitrarily large number of 1,1-nodes into 0,1-nodes. If we assign a potential to each node based only on the rank differences of its children, we need the potential of roots to exceed that of 1,1-nodes, that of 1,1-nodes to exceed that of 0,1-nodes, and that of 0,1-nodes to be no less than that of roots, an impossibility. Still, key decreases have a limited effect on 1,1-nodes and 0,1-nodes, an effect that depends on the rank differences of their grandchildren. Consider a 1,1-node x with two 1,1-children. A decrease in the rank of x converts x into a 0,1-node whose 0-child is a 1,1-node. A 0,1-node whose 0-child is a 1,1-node cannot decrease in rank as a result of the rank reduction of one of its children: such a node stops the cascade of rank reductions triggered by a key decrease. This suggests a potential function that treats such nodes as special cases. To make this idea work, we need to distinguish between links that create 1,1-nodes with two 1,1-children and those that do not. We partition the nodes into two kinds, *good* and *bad*. A node is good if it is a root of rank zero (with a missing left child) or its left child is a 1,1-node; or it is a 1,1-node of rank zero (with missing children) or its children are 1,1-nodes; or it is a 0,1-node whose 0-child is a 1,1-node. All other nodes are bad. With this definition, the winner of a link is a good root; the loser of a link is a 1,1-node that is good if the two roots linked were both good, bad if at least one of the roots linked was bad.

As we show below, a key decrease can make at most one good node bad. This al-

lows us to give good nodes anomalously low potential, thereby avoiding the circularity problem mentioned above. There is one final difficulty, however. The tree disassembly triggered by a minimum deletion can produce new bad roots whose potential is too low. Such roots have enough potential to pay for links between two of them, but not for links between one of them and a good root. This is the reason for the restriction on linking.

It remains for us to work out the details of this idea. We define the potential of a heap to be the sum of its node potentials. We define the potential of a node to be the sum of the rank differences of its children, plus two if it is a root, minus one if it is good, or plus three if it is bad. This definition applies not only to nodes that obey the rank rule but to the node that violates the rank rule in the middle of a key decrease. That is, the potential of a root with an i-child is $i+1$ if the root is good, $i+5$ if the root is bad. The potential of a 1,1-node is 1 if it is good, 5 if it is bad. The potential of a 0,1-node is 0 if it is good, 4 if it is bad. The potential of an i,j-node that is not a 1,1-node or a 0,1-node is $i+j+3$: all such nodes are bad.

THEOREM 6.2. *The amortized time for an operation on a type-*1 *rp-heap with restricted multipass linking is* $O(1)$ *for a make-heap, find-min, insert, meld, or decrease-key, and* $O(\log n)$ *for a delete-min.*

Proof. A make-heap, find-min, or meld operation takes O(1) actual time and does not change the potential; an insertion takes O(1) time and increases the potential by 2, since the new root is good. Hence each of these operations takes O(1) amortized time.

Consider a minimum deletion. During the disassembly and the links of pairs of new half trees (those formed by the disassembly), we give each root of a new half tree a *temporary potential* of four whether it is good or bad, instead of its correct potential of 2 if it is good, 6 if it is bad. We claim that if we do this, the increase in potential caused by the disassembly is at most $4 \lg n + 4$. The only nodes whose potential can increase are the new roots. Each bad node has at least four units of potential, so if it becomes a root it has the four units of temporary potential that it needs. Consider a good node x that becomes a root. There are two cases. If x is a 1,1-node, or x is a 0,1-node whose right child is a 1-child, then we charge the four or fewer units of temporary potential needed by x as a new root to $r(x)$. If x is a 0,1-node whose right child is a 0-child, then we charge the four units needed by x as a new root to $r(y)$, where y is the highest bad node on the right spine of x. See Figure 6.2. If there is no such y, then we charge the four units needed by x to the minimum deletion. If y exists, then each node on the path from x to y is a good 1,1-node except for x and y. Since y is bad, $r(y)$ can be charged only once for such a 0,1-node, and it cannot be charged for a good 1,1-node or a 0,1-node whose right child is a 1-child. If y does not exist, then each node on the right spine of x except x is a good 1,1-node. Thus x is the only node that can produce a charge to the minimum deletion. This gives the claim.

Now consider the links of pairs of new half trees. Each such link converts a root into a 1,1-node and makes the remaining root good. Before the link, these nodes have potential eight (four plus four). We give the root remaining after the link its correct potential of two. After the link the remaining root and the new 1,1-node have potential at most seven (two plus five), so the link reduces the potential by at least one.

After all the links of pairs of new half trees are done, there is at most one new half tree per rank. We give each such half tree its correct potential: 2 if it is good, 6 if it is bad. This increases the potential by at most two units for each rank less

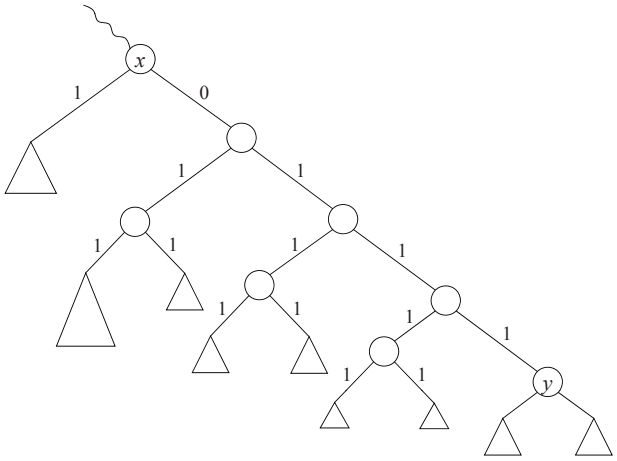

FIG. 6.2. *Charging rule for extra potential needed by roots. Node x needs additional potential when it becomes a root. This potential is charged to $r(y)$, the highest bad node on the right spine of x. Since all nodes above y on the right spine of x are good, all except x are good $1,1$-nodes. If $x = y$, so that $r(x)$ is charged, then the parent of x cannot be a good $1,1$-node, nor can x be the 0-child of a good $0,1$-node. Thus $r(x)$ can be charged only once.*

than the maximum rank, for a total of at most $2 \lg n$. (This bound on the potential increase is the reason for the restriction on linking.) Then we do the remaining links. Each such link reduces the potential by one: if at least one of the roots to be linked is bad, then the two roots have at least $6 + 2 = 8$ units of potential before the link and at most 2 (the winner) + 5 (the loser) = 7 after the link; if both roots are good, they have $2 + 2 = 4$ units before the link and $2 + 1 = 3$ after, since the loser is a good 1,1-node.

We conclude that the net increase in potential caused by the entire minimum deletion is at most $6 \lg n + 4$, minus one per link. Let h be the number of half trees after the disassembly. The entire minimum deletion takes $h - 1$ key comparisons and $O(h + 1)$ time. Scale this time to be at most h+O(1). At most $\lg n + 1$ half trees remain after all links, so there are at least $h - \lg n - 1$ links. The minimum deletion thus increases the potential by at most $7 \lg n - h + 1$, giving an amortized time of $O(\log n)$ and at most $7 \lg n + 5$ amortized key comparisons.

The analysis of a key decrease at a node x is just like that for type-2 heaps, except that we must show that the key decrease can make only O(1) nodes bad. A good 1,1-node cannot become bad; it can only become a good 0,1-node. A good 0,1-node cannot decrease in rank, so if it becomes bad it is the last node at which a rank-reduction step occurs. If x becomes a root, then it can only become bad if it were previously a good 0,1-node with a right 0-child, in which case no ranks change and x is the only node that becomes bad. For the root of the old half tree containing x to become bad, its left child must be a 1,1-node, and the old root is the only node that becomes bad. We conclude that the key decrease can make only one node bad. If we give the one node that becomes bad four units of potential before the rank-reduction process, then each reduction of a node rank reduces the potential by at least one, paying for the decrease. The key decrease can also create one new root, for a potential increase of two. Thus the key decrease takes O(1) amortized time. □

The worst-case time for a key decrease in an rp-heap of either type is $\Theta(n)$, as it is for Fibonacci heaps. We can reduce this to O(1) by delaying each key decrease

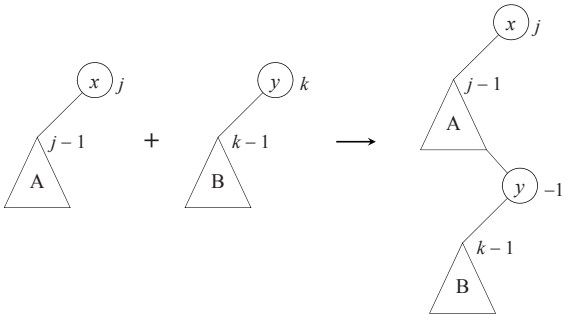

FIG. 7.1. *An unfair link.*

operation until the next minimum deletion. This requires maintaining the set of nodes that might have the minimum key; namely, all the roots and all nodes whose keys have decreased since the last minimum deletion.

7. One-tree rank-pairing heaps. It is natural to ask whether there is a one-tree version of rp-heaps. An ideal solution would handle unfair links (those in which the nodes to be linked have different ranks) in the same way as fair links, as in section 4: the loser of a fair link becomes the new left child of the winner. We have been unable to obtain a provably efficient version that does this. Instead, we offer here a one-tree version of rp-heaps that handles fair and unfair links differently. If x is a node, then the right spine of left(x) contains all the losers of links, with the newest loser the shallowest. That is, it is a stack of the losers to x. Make each such spine a deque (double-ended queue) instead of a stack. If y loses a fair link to x, push y onto the stack, making it the new left child of x; if y loses an unfair link to x, inject y into the bottom of the stack, making it the new deepest node on the right spine of left(x). (See Figure 7.1.) In addition, when y loses a fair link to x, set its rank to -1; do not change the rank of x. The -1 is merely a flag that prevents rank reduction from propagating through losers of unfair links. When the loser of an unfair link becomes a root again, either because its key has decreased or as a result of disassembly, set its rank equal to one greater than that of its left child.

Represent a heap by a single half tree. To insert an item into a heap, make it into a one-node half tree and link this half tree with the existing half tree, by a fair or unfair link as appropriate. To meld two heaps, link their half trees, by a fair or unfair link. To do a minimum deletion, disassemble the half tree, link the half trees formed by the disassembly by fair links until no two remaining trees have equal rank, and then link the remaining half trees by unfair links. If the heap is of type 1, do the fair links by the restricted multipass method (section 6). Do a key decrease as in section 5; once the rank-reduction process stops, if there are two trees, link them, by a fair or unfair link.

To implement this method efficiently, we need to change the pointer structure so that unfair links can be done in O(1) time. One way of doing this is to make each node with a left child point not to its left child but to the bottommost node on the right spine of its left child, and to make this bottommost node (which has no right child) point to the left child. (See Figure 7.2.) Other representations are possible. Which is best is a question for experiments to resolve.

The analysis of one-tree rp-heaps is like that of one-pass rp-heaps except that we must account for unfair links. There is one unfair link per insert, meld, and decrease-

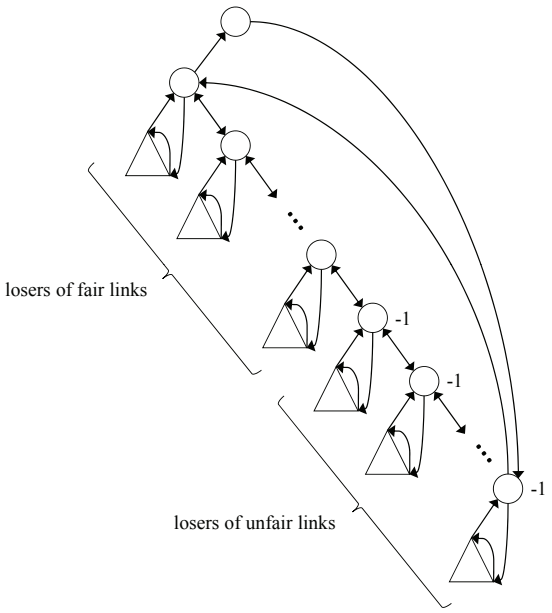

FIG. 7.2. *A one-tree rp-heap representation that uses three pointers per node. Unfair losers get a temporary rank of* -1 *to prevent rank reduction from propagating through them.*

key, and $O(\log n)$ per delete-min. We give losers of unfair links the same potential as roots: in effect, they are roots. It is then straightforward to extend the proofs of Theorems 6.1 and 6.2 to one-tree rp-heaps.

8. Can key decrease be made simpler? It is natural to ask whether there is an even simpler way to decrease keys while retaining the amortized efficiency of Fibonacci heaps. We give two answers: "no" and "maybe." We answer "no" by showing that two possible methods fail. The first method allows arbitrarily negative but bounded positive rank differences. With such a rank rule, the rank-reduction process following a key decrease need examine only ancestors of the node whose key decreases, not their siblings. Such a method can take $\Omega(\log n)$ time per key decrease, however, as the following example shows. Let b be the maximum allowed rank difference. Choose k arbitrarily. By means of a suitable sequence of insertions and minimum deletions, build a heap that contains a perfect half tree of each rank from 0 through $bk+1$. Let x be the root of the half tree of rank $bk+1$. Consider the right spine of left(x). Decrease the key of each node on this path whose rank is not divisible by b. Each such key decrease takes $O(1)$ time and does not violate the rank rule, so no ranks change. Now the path consists of $k+1$ nodes, each with rank difference b except the topmost. (See Figure 8.1.) Decrease the keys of these nodes, from smallest rank to largest. Each such key decrease will cause a cascade of rank reductions all the way to the topmost node on the path. The total time for these $k+1$ key decreases is $\Omega(k^2)$. After all the key decreases, the heap contains three perfect half trees of rank zero and two of each rank from 1 through bk. A minimum deletion (of one of the roots of rank zero) followed by an insertion makes the heap again into a set of perfect half trees, one of each rank from 0 through $bk+1$. Each execution of this cycle does $O(\log n)$ key decreases, one minimum deletion, and one insertion, and takes $\Omega(\log^2 n)$ time.

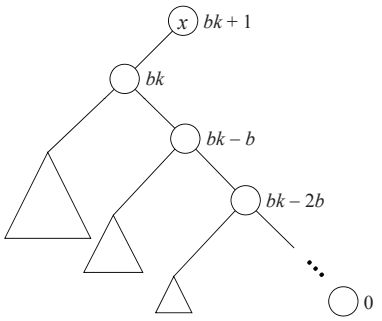

FIG. 8.1. *A half tree of rank* $bk+1$ *in the counterexample to the key decrease method that allows arbitrary negative rank differences but positive rank differences bounded by* b. *A sequence of* $k+1$ *key decreases on the right spine of left*(x), *from smallest rank to largest, requires* $\Omega(k^2)$ *total time.*

The second, even simpler method spends only $O(1)$ time worst-case on each key decrease, thus avoiding arbitrary cascading. In this case, by doing enough operations one can build a half tree of each possible rank, up to a rank that is $\omega(\log n)$. Once this is done, repeatedly doing an insertion followed by a minimum deletion (of the just-inserted item) will result in each minimum deletion taking $\omega(\log n)$ time. Here are the details. Suppose each key decrease changes the ranks of nodes at most d pointers away from the node whose key decreases, where d is fixed. Choose k arbitrarily. By means of a suitable sequence of insertions and minimum deletions, build a heap that contains a perfect half tree of each rank from 0 through k. On each node of distance $d+2$ or greater from the root, in decreasing order by distance, do a key decrease with $\Delta = \infty$ followed by a minimum deletion. No roots can be affected by any of these operations, so the heap still consists of one half tree of each rank, but each half tree contains at most 2^{d+1} nodes, so there are at least $n/2^{d+1}$ half trees. Now repeat the cycle of an insertion followed by a minimum deletion. Each such cycle takes $\Omega(n/2^{d+1})$ time. The choice of "$d+2$" in this construction guarantees that no key decrease can reach the child of a root, and hence cannot change the rank of a root (other than the node whose key decreases).

This construction works even if we add extra pointers to the half trees, as in Fibonacci heaps. Suppose we add ordered ancestor pointers to our half trees. Even for such an augmented structure, the latter construction gives a bad example, except that the size of a constructed half tree of rank k is $O(k^{d+1})$ instead of $O(2^{d+1})$, and each cycle of an insertion followed by a minimum deletion takes $\Omega(n^{1/(d+2)})$ time.

One limitation of this construction is that building the initial set of half trees takes a number of operations exponential in the size of the heap on which the repeated insertions and minimum deletions are done. Thus it is not a counterexample to the following question: is there a fixed d such that if each key decrease is followed by at most d rank-reduction steps (say of type 1), then the amortized time is $O(1)$ per insert, meld, and decrease-key, and $O(\log m)$ per delete-min, where m is the total number of insertions? A related question is whether Fibonacci heaps without cascading cuts have these bounds. We conjecture that the answer is yes for some positive d, perhaps even $d = 1$. The following counterexample shows that the answer is no for $d = 0$. That is, the answer is no for the method in which a key decrease changes no ranks except for the ranks of roots. For arbitrary k, build a half tree of each rank from 0 through k, each consisting of a root and a path of left children, by proceeding inductively as

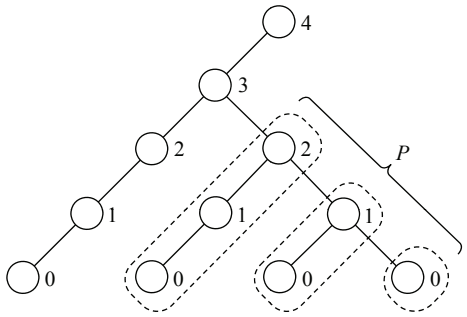

FIG. 8.2. *A half tree of rank $k = 4$ buildable in $O(k^3)$ operations if key decreases do not change ranks. Key decreases on the nodes of P detach the circled subtrees.*

follows. Given such half trees of ranks 0 through $k-1$, insert an item less than all those in the heap and then do k cycles, each consisting of an insertion followed by a minimum deletion that deletes the just-inserted item. The result will be one half tree of rank k consisting of the root, a path of left children descending from the root, a path P of right children descending from the left child of the root, and a path of left children descending from each node of P; every child has rank difference 1. (See Figure 8.2.) Do a rank reduction on each node of P. This produces a set of half trees of ranks 0 through k except for $k-1$, each a path. Repeat this process on the set of half trees up to rank $k-2$, resulting in a set of half trees of ranks 0 through k with $k-2$ missing. Continue in this way until only rank 0 is missing, and then do a single insertion. Now there is a half tree of each rank, 0 through k. The total number of heap operations required to increase the maximum rank from $k-1$ to k is $O(k^2)$, so in m heap operations one can build a set of half trees of each possible rank up to a rank that is $\Omega(m^{1/3})$. On the heap represented by this set of half trees, an insertion followed by a minimum deletion takes $\Omega(m^{1/3})$ time, and this pair of operations can be repeated any number of times.

9. Remarks. We have presented a new data structure, the rank-pairing heap, that combines the performance guarantees of Fibonacci heaps with simplicity approaching that of pairing heaps. Like pairing heaps, the trees in rp-heaps can have arbitrarily unbalanced structure; unlike pairing heaps, rp-heaps use ranks to guarantee efficiency. As Fredman [13] showed, subject to certain technical constraints, some information such as ranks must be stored to guarantee efficiency; our data structure obeys the constraints of Fredman's bound. Our results build on previous work by Peterson, Høyer, Kaplan and Tarjan, and others, and may be the natural conclusion of this work: we have shown that simpler methods of doing key decreases do not have the desired efficiency.

Type-1 rp-heaps, although simple, are not simple to analyze. Indeed, we were surprised to discover that type-1 rp-heaps have the same efficiency as Fibonacci heaps. This is less surprising for type-2 heaps. One can make the analysis even simpler by relaxing the data structure even more, specifically by disallowing 0,2-nodes but allowing 1,3-nodes. This gives the *type*-3 rp-heap: every nonroot is a 1,1-node, a 1,2-node, a 1,3-node, or a $0,i$-node for some $i > 2$. For this data structure, giving each root a potential of one and each nonroot a potential equal to minus two plus the sum of the rank differences of its children allows one to prove the analogue of Theorem 6.1. The trade-off is that the subtree size bound (the analogue of Lemma 5.3) is worse for

type 3 than for type 2. Type 2 seems to us to be the sweet spot of the design space: simple, not too hard to analyze, with small constant factors.

Our preliminary experiments suggest that rp-heaps are competitive in practice with pairing heaps. Much more thorough and careful experiments remain to be done to compare these structures and others. We leave this for the future.

Several interesting theoretical questions remain. Is there a simpler analysis of type-1 rp-heaps? Do type-1 rp-heaps still have the efficiency of Fibonacci heaps if the restriction on linking used in the analysis of section 6 is removed? More interestingly, can one obtain an $O(1)$ amortized time bound for insert, meld, and decrease-key and $O(\log m)$ for delete-min (where m is the total number of insertions) if only $O(1)$ rank changes are made after each key decrease? (See section 8.)

Acknowledgments. We thank Haim Kaplan and Uri Zwick for extensive discussions that helped to clarify the ideas in this paper, and for pointing out an error in our original analysis of type-1 rp-heaps.

REFERENCES

[1] G. M. Adel'son-Vel'skii and E. M. Landis, *An algorithm for the organization of information*, Sov. Math. Dokl., 3 (1962), pp. 1259–1262.

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

[3] G. S. Brodal and C. Okasaki, *Optimal purely functional priority queues*, J. Funct. Programming, 6 (1996), pp. 839–857.

[4] M. R. Brown, *Implementation and analysis of binomial queue algorithms*, SIAM J. Comput., 7 (1978), pp. 298–319.

[5] T. M. Chan, *Quake Heaps: A Simple Alternative to Fibonacci Heaps*, 2009; available online at http://www.cs.uwaterloo.ca/~tmchan/heap.ps.

[6] E. W. Dijkstra, *A note on two problems in connexion with graphs*, Numer. Math., 1 (1959), pp. 269–271.

[7] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, *Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation*, Comm. ACM, 31 (1988), pp. 1343–1354.

[8] R. D. Dutton, *The Weak-Heap Data Structure*, Technical report CS-TR-92-09, University of Central Florida, Orlando, FL, 1992.

[9] J. Edmonds, *Optimum branchings*, J. Res. Nat. Bur. Standards, B71 (1967), pp. 233–240.

[10] A. Elmasry, *Pairing heaps with* $O(\log \log n)$ *decrease cost*, in Proceedings of the 20th ACM-SIAM Symposium on Discrete Algorithms (SODA), 2009, pp. 471–476.

[11] A. Elmasry, *The violation heap: A relaxed Fibonacci-like heap*, in Proceedings of the 16th International Conference on Computing and Combinatorics (COCOON), 2010, pp. 479–488.

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

[13] M. L. Fredman, *On the efficiency of pairing heaps and related data structures*, J. ACM, 46 (1999), pp. 473–501.

[14] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan, *The pairing heap: A new form of self-adjusting heap*, Algorithmica, 1 (1986), pp. 111–129.

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

[16] M. L. Fredman and D. E. Willard, *Trans-dichotomous algorithms for minimum spanning trees and shortest paths*, J. Comput. System Sci., 48 (1994), pp. 533–551.

[17] H. N. Gabow, Z. Galil, T. H. Spencer, and R. E. Tarjan, *Efficient algorithms for finding minimum spanning trees in undirected and directed graphs*, Combinatorica, 6 (1986), pp. 109–122.

[18] L. J. Guibas and R. Sedgewick, *A dichromatic framework for balanced trees*, in Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science (FOCS), 1978, pp. 8–21.

[19] B. Haeupler, S. Sen, and R. E. Tarjan, *Rank-pairing heaps*, in Proceedings of the 17th European Symposium on Algorithms (ESA), 2009, pp. 659–670.

[20] Y. Han and M. Thorup, *Integer sorting in* $O(n\sqrt{\log\log n})$ *expected time and linear space*, in Proceedings of the 43rd IEEE Symposium on Foundations of Computer Science (FOCS), 2002, pp. 135–144.
[21] P. Høyer, *A general technique for implementation of efficient priority queues*, in Proceedings of the Third Israeli Symposium on the Theory of Computing and Systems (ISTCS), 1995, pp. 57–66.
[22] H. Kaplan, N. Shafrir, and R. E. Tarjan, *Meldable heaps and boolean union-find*, in Proceedings of the 34th ACM Symposium on Theory of Computing (STOC), 2002, pp. 573–582.
[23] H. Kaplan and R. E. Tarjan, *New Heap Data Structures*, Technical report TR-597-99, Department of Computer Science, Princeton University, Princeton, NJ, 1999.
[24] H. Kaplan and R. E. Tarjan, *Thin heaps, thick heaps*, ACM Trans. Algorithms, 4 (2008), pp. 1–14.
[25] D. E. Knuth, *The Art of Computer Programming, Volume* 1: *Fundamental Algorithms*, Addison-Wesley, Upper Saddle River, NJ, 1973.
[26] D. E. Knuth, *The Art of Computer Programming, Volume* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.
[27] A. M. Liao, *Three priority queue applications revisited*, Algorithmica, 7 (1992), pp. 415–427.
[28] B. M. E. Moret and H. D. Shapiro, *An empirical analysis of algorithms for constructing a minimum spanning tree*, in Proceedings of the 2nd Workshop on Algorithms and Data Structures (WADS), 1991, pp. 400–411.
[29] G. L. Peterson, *A Balanced Tree Scheme for Meldable Heaps with Updates*, Technical report GIT-ICS-87-23, School of Informatics and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1987.
[30] S. Pettie, *Towards a final analysis of pairing heaps*, in Proceedings of 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS), 2005, pp. 174–183.
[31] J. T. Stasko and J. S. Vitter, *Pairing heaps: Experiments and analysis*, Comm. ACM, 30 (1987), pp. 234–249.
[32] T. Takaoka, *Theory of trinomial heaps*, in Proceedings of the Sixth International Conference on Computing and Combinatorics (COCOON), 2000, pp. 362–372.
[33] T. Takaoka, *Theory of* 2-3 *heaps*, Discrete Appl. Math., 126 (2003), pp. 115–128.
[34] R. E. Tarjan, *Amortized computational complexity*, SIAM J. Alg. Disc. Meth., 6 (1985), pp. 306–318.
[35] M. Thorup, *Integer priority queues with decrease key in constant time and the single source shortest paths problem*, J. Comput. System Sci., 69 (2004), pp. 330–353.
[36] M. Thorup, *Equivalence between priority queues and sorting*, J. ACM, 54 (2007), 28.
[37] J. Vuillemin, *A data structure for manipulating priority queues*, Comm. ACM, 21 (1978), pp. 309–315.