RANK-PAIRING HEAPS

BERNHARD HAEUPLER[1] , SIDDHARTHA SEN[2,4] , AND ROBERT E. TARJAN[3,4]

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. Unlike all other heap implementations that match the bounds of Fibonacci heaps, our structure needs only one cut and no other structural changes per key decrease; the trees representing the heap can therefore evolve to have arbitrary structure. Although the data structure is simple, its analysis is not. Our initial experiments indicate that rank-pairing heaps perform almost as well as pairing heaps on typical input sequences and better on worst-case sequences.

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

AMS subject classifications.

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:

- $make\text{-}heap$: return a new, empty heap.
- $insert(x, H)$: insert item x, with predefined key, into heap H.
- $find\text{-}min(H)$: return the item in heap H of minimum key.
- $delete\text{-}min(H)$: 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:

- $decrease\text{-}key(x, \Delta, H)$: decrease the key of item x in heap H by amount $\Delta > 0$, assuming H is the unique heap containing x.
- $delete(x, H)$: delete item x from heap H, assuming 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 [32] of heap operations. To obtain a bound on amortized efficiency, we assign to each configuration of the data structure a non-negative *potential*, initially zero. We define 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 [25, p. 183] on the number of 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 $\mathrm{O}(\log n)$ amortized time, and each of the other supported heap operations takes $\mathrm{O}(1)$ amortized time. These bounds match the lower bound. (The logarithmic lower bound can be beaten in a more powerful model of computation that allows multiway branching [15, 19].)

Many heap implementations have been proposed over the years. We mention only those directly related to our work. The *binomial queue* of Vuillemin [34] supports all the heap

[1]Massachusetts Institute of Technology, haeupler@mit.edu.

[2]Princeton University, sssix@cs.princeton.edu.

[3]Princeton University and HP Labs, ret@cs.princeton.edu.

[4]Research at Princeton University 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.

operations in $O(\log n)$ worst-case time per operation. This structure performs quite well in practice [4]. Fredman and Tarjan [14] 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, 14], Edmonds' minimum branching algorithm [9, 16], and certain minimum spanning tree algorithms [14, 16]. 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. [13] 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 [26, 27]. One reason Fibonacci heaps perform poorly is that they need an extra pointer per node. Fredman et al. [13] 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 [26, 31], Fredman [12] 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][5] 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] and fat heaps [22] achieve the bounds except for melding, which takes $O(\log n)$ time worst-case, data structures of Brodal [2] and of Brodal and Okasaki [3] achieve the bounds except for key decrease, which takes $O(\log n)$ time in the worst case, and a very complicated data structure of Brodal [2] achieves all the bounds worst-case. Also, Elmasry [11] has 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, although Elmasry's data structure does not satisfy the technical restrictions of Fredman's bound.

Working in a different direction, several authors proposed data structures with the same amortized efficiency as Fibonacci heaps but intended to be simpler. Peterson [29] gave a structure based on AVL trees. Høyer [20] gave several structures, including ones based on red-black trees, AVL trees, and a, b-trees. Høyer's simplest structure is one he calls a *one-step heap*. Kaplan and Tarjan [23] 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 [10] developed *violation heaps* and Chan [5] *quake heaps*.

All these structures have in common that the trees representing the heap have some kind of balance property. As a result, a key decrease can trigger not only a cut of a subtree, but some additional tree restructuring. Our insight is that such restructuring is unnecessary: all that is needed is a way to control the size of trees that are combined. Our new data structure, the *rank-pairing heap*, does (at most) one cut and no other restructuring per key decrease. As in 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 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. Although rank-pairing heaps are simple, their analysis is not.

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

In our preliminary experiments, rank-pairing heaps perform almost as well as pairing heaps on typical input sequences and faster on worst-case sequences.

The remainder of our paper consists of eight sections. Sections 2 and 3 review the common basis of binomial queues, Fibonacci heaps, and all the related structures. Each of these data structures can be viewed as a set of single-elimination tournaments. Section 2 describes such tournaments and ways of representing them. Section 3 develops two variants of binomial queues, *one-pass binomial queues* and *one-tree binomial queues*. Section 4 extends one-pass binomial queues to support key decrease and arbitrary deletion, thereby obtaining the *rank-pairing heap* or *rp-heap*. We present two types of rp-heaps, type 1 and type 2, which differ only in the rule obeyed by ranks. Type 2 obeys a more relaxed rank rule, which makes it slightly more complicated but simplifies its analysis and yields mostly smaller constant factors. Section 5 analyzes the amortized efficiency of both types. Section 6 presents a one-tree version of rank-pairing heaps. The modification that makes the data structure into a single tree applies to either type 1 or type 2 rp-heaps, and preserves the efficiency results of Section 5. Section 7 shows that some even simpler ways of implementing key decrease have worse amortized efficiency. Section 8 describes our preliminary experiments comparing rank-pairing heaps with pairing heaps. Section 9 gives our conclusions and mentions some open problems. A preliminary version of some of this work appeared in a conference paper [18].

2. Tournaments. The basis 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 2.1.) The *full representation* is a full binary tree with one leaf per item and one non-leaf per link. Each non-leaf 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 non-leaves 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. Most of the heap implementations mentioned in the introduction were originally presented in the heap-ordered representation.

We obtain the *half-ordered representation* from the heap-ordered representation by using the binary tree representation [24, 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

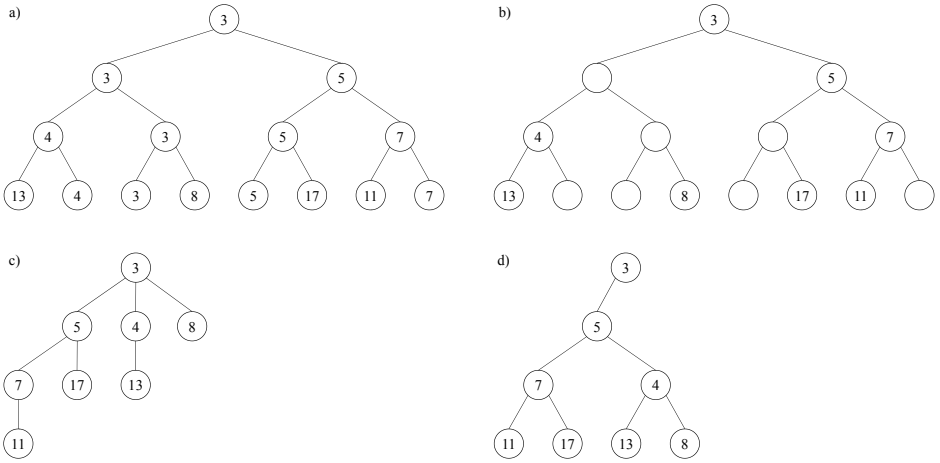

FIG. 2.1. *Four representations of a tournament: a) full, b) half-full, c) heap-ordered, and d) half-ordered.*

the root to be the parent of the nearest ancestor of x that is a left child. This is just the parent of x in the heap-ordered representation.

The half-ordered representation appears in the original paper on pairing heaps [13]. 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 [24]. 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. Henceforth all our binary trees are half-ordered. Linking takes the following form on half-trees (see Figure 2.2): 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.

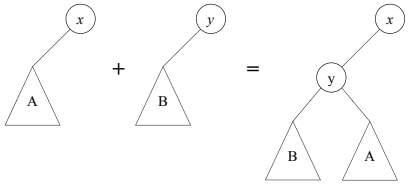

FIG. 2.2. *A link of two half-trees with roots x and y, x having smaller key.*

3. Two New Variants of Binomial Queues. The heap implementations mentioned in the introduction, and ours, are extensions of the following generic implementation. We represent a heap by a set of half-trees whose nodes are the items in the heap. We represent the set by a singly-linked circular list of the tree roots, with the root of minimum key first. Access to the list is via the first root.

We implement the various heap operations, excluding key decrease and arbitrary deletion, as follows. To find the minimum in a heap, return the first root. To make a heap, create an empty list of roots. To insert an item, make it a one-node half-tree and insert it into the list of roots, in first position if it has minimum key, second if not. To meld two heaps, catenate their lists of roots, making the root of minimum key first on the new list. To delete the minimum, disassemble the half-tree rooted at the first root x, as follows. Let y be the left child of x. Delete x and cut each edge on the right 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. Add the new half-trees to the remaining half-trees. Find the root of minimum key, and make it first on the root list. 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 only efficient if the links are done carefully. In pairing heaps [13], 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 [12] 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 non-negative *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. Only link two half-trees 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 3.1.

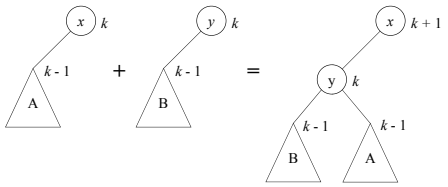

FIG. 3.1. *A link of two half-trees with roots x and y, x having smaller key. Ranks are to the right of nodes.*

If all links are done this way, every half-tree ever in a heap is *perfect*; namely, 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 height h in a tree of rank r is the binomial coefficient $\binom{r}{h}$. (See Figure 3.2.)

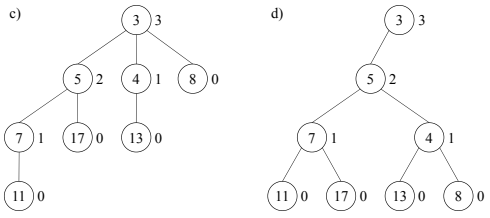

FIG. 3.2. *The c) heap-ordered and d) half-ordered representations of a tournament from Figure 2.1. Ranks are to the right of nodes.*

In the original version of binomial queues [34], links are done eagerly, to maintain the invariant that a heap contains at most one root per rank. This gives an $\mathrm{O}(\log n)$ worst-case

time bound for insertion, meld, and delete-min. Doing links 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 would work for us as well, but we prefer a lazier alternative, giving what we call the *one-pass binomial queue*: 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 [13].

To implement one-pass 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 and add the newly formed half-tree to an output list representing the updated heap, leaving the bucket empty. 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 output list, leaving all the buckets empty.

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

THEOREM 3.1. *The amortized time for an operation on a one-pass binomial queue is* $\mathrm{O}(1)$ *for a make-heap, find-min, insert, or meld, and* $\mathrm{O}(\log n)$ *for a delete-min.*

Proof. A make-heap, find-min, insert, or meld takes $\mathrm{O}(1)$ actual time. Of these operations, only an insert increases the potential, by one. Thus each of these operations takes $\mathrm{O}(1)$ amortized time. Consider a minimum deletion. Disassembling the half-tree rooted at the node of minimum key increases the number of half-trees and hence the potential by at most $\lg n$. Let h be the number of half-trees after the disassembly. The entire minimum deletion takes $\mathrm{O}(h+1)$ time. Scale this time to be at most $h/2+\mathrm{O}(1)$. (This is equivalent to multiplying the potential by a constant factor.) Each link after the disassembly reduces the potential by one. 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 $(3/2)\lg n - h/2 + 1/2$, giving an amortized time of $\mathrm{O}(\log n)$. □

Theorem 3.1 holds if we do arbitrary additional links of half-trees of equal rank during a minimum deletion, up to and including doing links until there is at most one half-tree per rank.

We conclude this section by presenting a one-tree version of binomial queues with the same amortized efficiency as the one-pass version. 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 loser's rank unchanged; either leave the winner's rank unchanged or, optionally, increase it to any value not greater than the loser's rank. (Thus if the loser's rank is less than the winner's, the winner's rank does not change.) Unfair links do not adversely affect the amortized efficiency of the data structure, if one does as few 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 as in the one-pass method, but add each tree formed by a link not to an output list but to the set of trees to be processed. Once all half-trees are in buckets, remove them from the buckets and link them in any order (by unfair links) until only one half-tree remains.

LEMMA 3.2. *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 3.3. *The amortized time for an operation on a one-tree binomial queue is* $\mathrm{O}(1)$ *for a make-heap, find-min, insert, or meld, and* $\mathrm{O}(\log n)$ *for a delete-min.*

Proof. The analysis of find-min, make-heap, insert, and meld is just like that of one-pass binomial queues except that each insert or meld does a link. Each such link takes $\mathrm{O}(1)$ time and does not increase the potential. Consider a minimum deletion. Disassembling the half-tree 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 3.2 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 $\mathrm{O}(h+1)$ time. Scale this time to be at most $h + \mathrm{O}(1)$. 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 $\mathrm{O}(\log n)$. □

4. Rank-Pairing Heaps. Our main goal is to implement key decrease so that it takes $\mathrm{O}(1)$ 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 $\mathrm{O}(1)$ [21].

We shall extend one-pass binomial queues to support key decrease. We call the resulting data structure the *rank-pairing heap*. We develop two types of rank-pairing heaps: type 1, which is simpler but harder to analyze and has larger constant factors in the time bounds, and type 2, a 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. [13], 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.

Once the data structure supports parental access, we can decrease the key of item x in heap h as follows. (See Figure 4.1.) Reduce the key of x. If x is not a root, 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 $y = right(x)$, reattaching the subtree rooted at y in place of the original subtree rooted at x, and adding x to the list of roots. Whether or not x was originally a root, make it the first root if its key is now minimum.

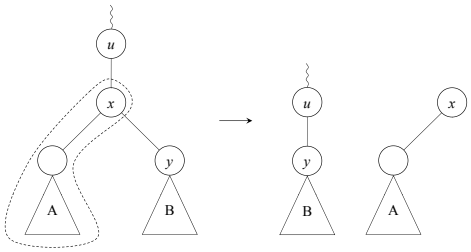

FIG. 4.1. *Restructuring during a key decrease.*

Remark: If x is not originally a root, there is no way in our representation to test in $\mathrm{O}(1)$ 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 7: 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 $\mathrm{O}(\log n)$, one that takes only $\mathrm{O}(1)$ amortized time per key decrease.

In Fibonacci heaps, the solution is to do a sequence of half-tree removals after each key decrease, but only $\mathrm{O}(1)$ amortized per decrease. These removals 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 [17]; devise a new balance rule, as in Høyer's one-step heaps [20] (thick heaps [23]), or thin heaps [23]; or do more-global rebuilding, as in violation heaps [10] and 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. Our insight is that 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 removal of the half-tree rooted 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 node 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 non-root 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 non-root, are missing. The definition of an i, j-node does

not distinguish between its left and right child. In a one-pass binomial queue, every root is a 1-node and every non-root is a 1,1-node. We shall relax the second half of this invariant.

Our key observation is that if a node has a 0-child, there is no harm in its other child having arbitrarily large rank difference. 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 rank-pairing heap (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 4.1. *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 one-pass 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 4.2): Reduce the key of x. If x is a root, make it the first 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, in first position if its key is minimum, second if not. 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, let $u = p(y)$ and repeat the following step until it stops:

Type-1 Rank-Decrease Step: If u is a root, set $r(u) = r(left(u)) + 1$ 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 rank-decrease step it cannot happen that $k > r(u)$, but this can happen in the one-tree variant of rp-heaps that we develop in Section 6.

LEMMA 4.2. *The rank-reduction process restores the rank rule.*

Proof. Since all rank differences are non-negative, $r(y) \leq r(x)$ 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-decrease step, the step decreases 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 rank-pairing heap (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-Decrease Step: If u is a root, set $r(u) = r(left(u)) + 1$ and stop. Otherwise,

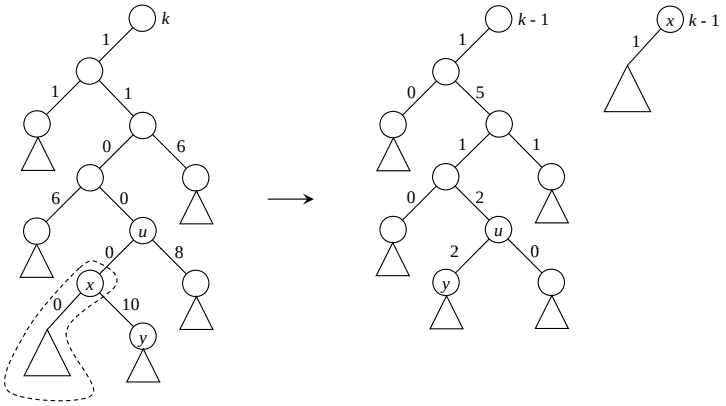

FIG. 4.2. *Key decrease in a type-1 rp-heap.*

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 4.2 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 it can never be the case that $k > r(u)$, although this can happen in the variant we develop in Section 6.

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 k^{th} 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 4.3. *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 [24, 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 $F_{k+1} \leq F_{k+2}$. □

5. 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. We assign a potential to every heap equal to the sum of its node potentials. The potential of a node is the sum of its *base potential* and its *extra potential*. The base potential of a node is the sum of the rank differences of its children, minus one. The extra potential of a node is one if it is a root, minus one if it is a 1,1-node, and zero otherwise. Thus the potential of a node is zero if it is a 1,1-node, one if it is a root, two if it is a 1,2-node, or $i - 1$ if it is a $0, i$-node.

THEOREM 5.1. *The amortized time for an operation on a type-2 rp-heap is $\mathrm{O}(1)$ for a make-heap, find-min, insert, meld, or decrease-key, and $\mathrm{O}(\log n)$ for a delete-min.*

Proof. A make-heap, find-min, or meld operation takes $\mathrm{O}(1)$ actual time and does not change the potential; an insertion takes $\mathrm{O}(1)$ time and increases the potential by one. Hence each of these operations takes $\mathrm{O}(1)$ 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 4.3 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 $O(h+1)$ time. Scale this time to be at most $h/2 + O(1)$. 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 do not participate in a link, so there are at least $(h - \log_\phi n - 1)/2$ links. The minimum deletion thus increases the potential by at most $O(\log n) - h/2$, giving an amortized time of $O(\log n)$.

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 x is not a root. Let $u_0 = left(x)$, let $u_1 = x$, and let $u_2, \ldots, u_k$ be such that $u_j = p(u_{j-1})$, u_j for $2 \leq j < k$ decreases in rank as a result of the key decrease, and u_k is either a root or does not decrease in rank. For $1 \leq j \leq k$ let v_j be the child of u_j other than u_{j-1}. Let r, r' denote ranks before and after the key decrease, respectively. The only nodes whose potential changes as a result of the key decrease are $u_1, \ldots, u_k$. Consider the sum of the base potentials of these nodes before and after the key decrease. The sum of the rank differences of $v_0, u_1, \ldots, u_{k-1}$ before the key decrease telescopes to $r(u_k) - r(v_0)$. The sum of the rank differences of $v_0, u_2, \ldots, u_{k-1}$ after the key decrease telescopes to $r'(u_k) - r'(v_0) \leq r(u_k) - r(v_0)$ since $r'(u_k) \leq r(u_k)$ and $r'(v_0) = r(v_0)$. Also, $\Delta r'(u_0) = 1 \leq \Delta r(u_0) + 1$, and $\Delta r'(v_j) \leq \Delta r(v_j) - 1$ for $2 \leq j < k$. It follows that the sum of the base potentials of $u_0, u_1, \ldots, u_k$ drops by at least $k - 3$ as a result of the key decrease. Consider the sum of the extra potentials of $u_1, \ldots, u_k$ before and after the key decrease. The following argument shows that at most two of these nodes can be 1,1-nodes before the key decrease. Let j be minimum such that u_j is a 1,1-node. Node u_j can decrease in rank by at most one; hence $u_{j'}$ for $j' > j$ can decrease in rank by at most one. If $u_{j'}$ for $j' > j$ is a 1,1-node, $u_{j'}$ becomes a 1,2-node as a result of $u_{j'-1}$ decreasing in rank by one, so $u_{j'}$ does not itself decrease in rank, and $j' = k$. We conclude that the sum of the extra potentials of $u_0, u_1, \ldots, u_k$ increases by at most three as a result of the key decrease: at most two 1,1-nodes become non-1,1-nodes, increasing the sum by two, and u_1 becomes a root, increasing the sum by one. If u_1 were originally a 1,1-node, its extra potential increases by two, since it is both an old 1,1-node and a new root. Combining results, we see that the key decrease causes the potential to drop by at least $k - 6$. Scale the time of a rank-decrease step so that it is at most one. Then the amortized time of the key decrease is $O(1)$. □

The analysis of type-1 rp-heaps is more complicated. We define the potential of a heap to be the sum of its node potentials, but we assign potentials to nodes based on their grandchildren, not just their children. We call a non-root node *good* if it is a root whose left child is a 1,1-node, or it and both of its children are 1,1-nodes, or it is a 0,1-node whose 0-child is a 1,1-node; otherwise, the node is *bad*. We define the potential of a leaf to be zero if it is a non-root or $3/2$ if it is a root. We define the potential of a non-leaf 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, plus three if it is bad. Thus its potential is zero if it is a good 0,1-node, one if it is a good 1,1-node, two if it is a good root, four if it is a bad 0,1-node, five if it is a bad 1,1-node, six if it is a bad root, and $i + 3$ if it is a $0, i$-node for $i > 1$.

Our analysis of minimum deletion in type-1 rp-heaps uses the fact that for each rank at most one half-tree of this rank formed by the disassembly is not linked to another half-tree formed by the disassembly. This fact follows from our linking method (see Section 3), which first links the new half-trees formed by the disassembly and then links remaining half-trees. Our analysis fails if links are done in arbitrary order. In contrast, our analysis of type-2

rp-heaps (above) is valid for any linking order.

THEOREM 5.2. *The amortized time for an operation on a type-1 rp-heap 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 $3/2$. Hence each of these operations takes $O(1)$ amortized time.

Consider a minimum deletion. During the disassembly and the links of half-trees formed by the disassembly, we give each non-leaf root of a tree formed by the disassembly a *temporary potential* of four whether it is good or bad. We claim that if we do this, the increase in potential caused by the disassembly is at most $4 \lg n$. Deletion of the root of minimum key reduces the potential by at least $3/2$. The only nodes whose potential can increase are the new roots. At most one leaf can become a root, increasing the potential by $3/2$. Such an increase is cancelled by the decrease resulting from deletion of the root of minimum key. Each bad node that becomes a root already has the four units of potential it needs as a new root. 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, we charge the four or fewer units of 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, we charge the four units needed by x as a new root to $r(y)$, where y is the node nearest x on the right spine of x that is not good. Since each node on the path from x to y except x is a 1,1-node, and each such node except y is good, $r(y)$ can only be charged 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. Thus the increase in potential caused by the disassembly is at most four for each rank less than the maximum rank, verifying the claim.

Now consider the links of new half-trees formed by the disassembly. Each such link of two leaves reduces the potential by one. Each such link of two non-leaves 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); after the link, they have potential at most seven. Thus the link reduces the potential by one.

After the links of new half-trees formed by the disassembly, there is at most one new half-tree per rank that has not been linked. We now give the root of each such half-tree its correct potential, two if it is good, six if it is bad. This increases the potential by at most two units for each rank less than the maximum rank, for a total of at most $2 \lg n$. Then we do the remaining links, each of which links a new half-tree and an old half-tree, or links two old half-trees. An extension of the argument in the previous paragraph shows that each such link reduces the potential by at least one. This is immediate if the roots of the half-trees to be linked are leaves, or if their potential totals at least eight. The remaining case is that both roots are good. Then their total potential is four before the link but three after it, since they both remain good.

We conclude that the net increase in potential caused by the entire minimum deletion is at most $6 \lg n$ minus one per link. Let h be the number of half-trees after the disassembly. The entire minimum deletion takes $O(h+1)$ time. Scale this time to be at most $h/2 + O(1)$. 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 $(13/2) \lg n - h/2 + 1/2$, giving an amortized time of $O(\log n)$.

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-decrease step occurs. If x becomes a root, it can only become bad if it was previously a good 0,1-node with a right 0-child, in

which case no ranks decrease 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, for a potential increase of at most four. It can also create a new root, for a potential increase of two. By the argument in the proof of Theorem 5.1, the key decrease reduces the potential by at least one for each node other than x whose rank decreases. Thus the key decrease takes $\mathrm{O}(1)$ 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 $\mathrm{O}(1)$ by delaying each key decrease operation until the next minimum deletion. This requires maintaining the set of nodes that might have the minimum key. This set includes all the roots and all nodes whose keys have decreased since the last minimum deletion. Making this change complicates the data structure, which may worsen its performance in practice.

6. One-Tree Rank-Pairing Heaps. Like binomial queues, rp-heaps retain their efficiency to within a constant factor if arbitrary additional fair links occur during minimum deletions. We can obtain a one-tree version of rp-heaps with the same efficiency if we allow unfair links as well. The construction, which consists of two steps, works for either type 1 or type 2.

The first step is to augment the half-tree data structure by maintaining, for each node x, the set of half-trees whose roots lost unfair links to x. We represent a heap by such an augmented half-tree. To do an unfair link between two half-trees, compare the keys of their roots and add the half-tree rooted at the loser to the set of half-trees of the winner. To insert an item into a heap, make the item into a one-node half-tree with an empty set of half-trees and link it with the half-tree representing the heap. To meld two heaps, link their half-trees. In each of these operations the link can be either fair or unfair; do it correspondingly. To delete the minimum of a heap, disassemble the half-tree representing the heap, add all the new half-trees to the set of half-trees whose roots lost unfair links to the deleted root, do fair links between half-trees of equal rank in this set until no two half-trees have equal rank, and then do unfair links to combine the remaining half-trees into a single half-tree. To decrease the key of a node x, proceed as in a one-pass rp-heap unless x is a root that lost an unfair link, in which case decrease the key of x and remove its half-tree from the set containing it. In either case finish the key decrease by linking the half-tree rooted at x with the half-tree representing the heap; the link can be either fair or unfair.

To implement this method we represent each set of half-trees by a doubly-linked list of their roots; we need double linking because deletions occur. Since a root has no parent and no right child, we can use the parent and right child pointers to store the list pointers, but each node needs one additional pointer, to the first root on its list. (See Figure 6.1.)

The second step eliminates the need for extra pointers and makes the data structure into a true one-tree representation. We change the representation as follows. For each node x, catenate the right spine of $left(x)$ with the root list of x. Now the right spine of $left(x)$ contains the nodes that lost fair links to x followed by those that lost unfair links to x. To do an unfair link in this representation, add the loser to the bottom of the right spine of the left child of the winner, rather than to the top. Also, give the loser of an unfair link a temporary rank of -1; this prevents rank reduction from propagating through such a node. When the loser of an unfair link becomes a root again, either because its key has decreased or as a result of disassembly, restore its correct rank, which is one greater than that of its left child.

With this change, a heap is just a half-tree. To do a minimum deletion, disassemble the half-tree, and link the half-trees formed by the disassembly until only one half-tree remains; do fair links until no more are possible, followed by unfair links. To do a key decrease,

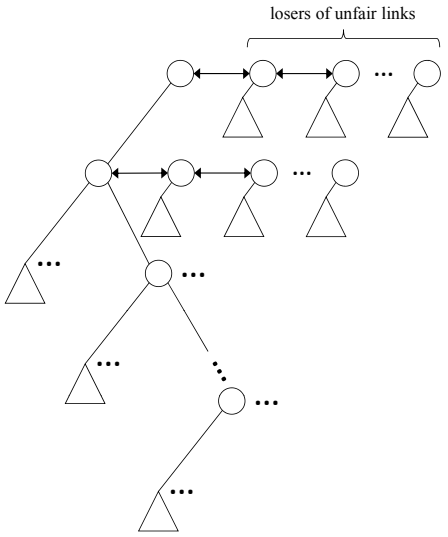

FIG. 6.1. *A one-tree rp-heap representation obtained by augmenting the half-tree data structure with a list of losers of unfair links for each node.*

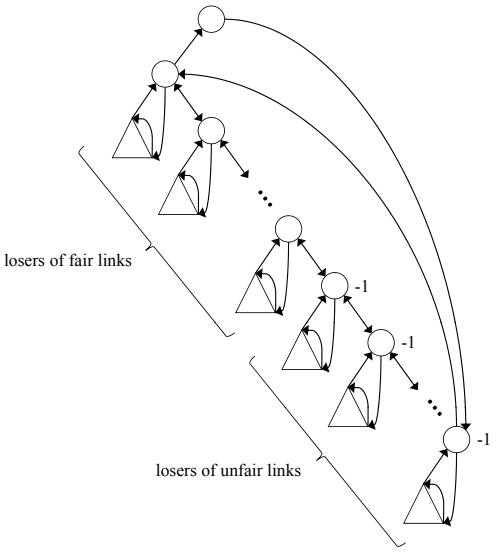

FIG. 6.2. *A one-tree rp-heap representation that uses three pointers per node. Unfair losers get a temporary rank of* -1 *to prevent rank reduction from propagating through them.*

proceed as in a one-pass rp-heap. Once the rank reduction process stops, if there are two half-trees, link them.

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 6.2.) 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 key decrease, and $O(\log n)$ per minimum deletion. 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 5.1 and 5.2 to one-tree rp-heaps.

7. 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-decrease 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 7.1.) Decrease the keys of these nodes, smallest rank to largest. Each such key decrease will cause a cascade of rank decreases 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.

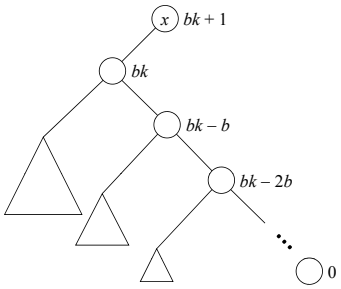

FIG. 7.1. *A half-tree of rank $bk + 1$ in the counterexample to the key decrease method that allows arbitrary negative rank differences but positive rank differences bounded by b. A sequence of $k + 1$ key decreases on the right spine of $left(x)$, from smallest rank to largest, requires $\Omega(k^2)$ 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 $\mathrm{O}(k^{d+1})$ instead of $\mathrm{O}(2^{d+1})$, and each cycle of an insertion followed by a minimum deletion takes $\Omega(n^{1/(d+2)})$ time.

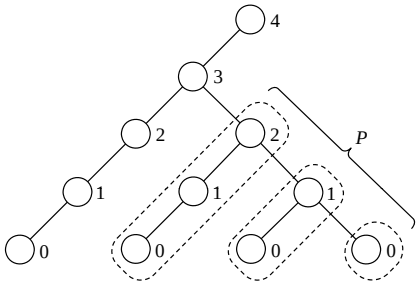

FIG. 7.2. *A half-tree of rank $k = 4$ buildable in $\mathrm{O}(k^3)$ operations if key decreases do not change ranks. Key decreases on the nodes of P detach the circled subtrees.*

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-decrease steps (say of type 1), then the amortized time is $\mathrm{O}(1)$ per insert, meld, and key decrease, and $\mathrm{O}(\log m)$ per deletion, 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 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 7.2.) Do a rank decrease 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 $\mathrm{O}(k^2)$, 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

TABLE 8.1
Performance of pairing heaps versus rp-heaps on typical input sequences. The number of vertices or items in the heap is n; the number of arcs is m. All results are in millions of updates.

Test	Parameters		P-heap	Type-2 rp-heap		Type-1 rp-heap	
	n	m		m-pass	1-pass	m-pass	1-pass
1. Dijkstra (max heap size)	8388609	25165824	339.40	346.38	323.05	352.59	325.07
2. Dijkstra (max key decrease)	65536	655360	5.06	5.49	4.27	5.66	4.27
3. Nagamochi-Ibaraki	32768	2684272	136.96	96.15	119.66	98.78	121.77
4. Sorting	100000	-	23.65	26.89	30.95	26.89	30.95
5. Two-way Dijkstra, E. USA	3598623	8778114	2371.04	2603.39	2878.15	2611.77	2896.85
6. Two-way Dijkstra, NYC	264346	733846	1070.66	1316.20	1430.82	1314.26	1425.03
7. Key decrease	262144	-	421.30	322.62	390.00	322.04	389.31
8. Key decrease	4096	-	4.32	3.32	3.91	3.40	3.95

by a minimum deletion takes $\Omega(m^{1/3})$ time, and this pair of operations can be repeated any number of times.

8. Experiments. In our preliminary experiments, rank-pairing heaps performed almost as well as pairing heaps on typical input sequences and faster on worst-case sequences. We compared rp-heaps to the standard two-pass version of pairing heaps on typical sequences and to the *auxiliary two-pass* version [31] on worst-case sequences; Stasko and Vitter [31] claimed that the latter outperforms other versions of pairing heaps on the class of worst-case sequences we used. We compared these two versions of pairing heaps to four versions of rp-heaps: type-1 and type-2, with as many links as possible done during minimum deletions ("multipass") and with only a single pass of links done ("one-pass"). We implemented all the methods in C, compiled with the `gcc -O2` option. We did all testing on a 2.13 Ghz Intel CPU running Windows Vista.

We performed three sets of experiments using typical input sequences (Table 8.1). Our performance measure was the total number of field updates done by each heap implementation. The first set of experiments used publicly available heap operation sequences [33], obtained by running Dijkstra's shortest paths algorithm [6] on graphs that maximize heap size or the number of key decreases; by running the Nagamochi-Ibaraki minimum-cut algorithm [28] on random graphs; and by running heapsort. The second set consisted of heap operation sequences obtained by running two-way Dijkstra between random pairs of nodes in real road networks. For the Eastern USA road network, we used 10 pairs of nodes, and for the New York City road network, we used 100 pairs. The third set tested key decreases by running $2n$ rounds of the following operations on an n-node heap, initially created by $n+1$ insertions followed by a minimum deletion: an insertion of a random key, followed by $\lg n - 1$ random key decreases, followed by a minimum deletion.

The results in Table 8.1 show that rp-heaps performed fewer field updates than pairing heaps on sequences with many key decreases (tests 2,3,7, and 8), and more updates on sequences with few key decreases. The multipass versions slightly outperformed the one-pass versions overall, with an average speedup of over 1.8% versus an average slowdown of under 5% relative to pairing heaps, respectively. The one-pass versions, while benefiting from smaller trees (and hence fewer rank updates), suffered from greater overhead during minimum deletions as a result of maintaining longer lists of half-trees.

To investigate the worst-case behavior of key decreases, we used Fredman's [12] version of an experiment of Stasko and Vitter [31], designed to show that pairing heaps require $\Omega(\log\log n)$ amortized time per key decrease. The experiment is identical to the third set of experiments in Table 8.1, but it uses information-theoretic heuristics in the heap operations, in two ways. First, the outcome of a link is determined by the sizes of the half-trees involved:

the root of the larger tree wins the link. Second, the candidates for key decreases are chosen to undo links of high efficiency, where the efficiency of a link is the ratio of the size of the half-tree that loses to the size of the half-tree that wins. These heuristics correspond to the bad sequences and adversarial behavior in Fredman's lower bound analysis [12]. The cost of a round is measured as the number of links done divided by $\lg n$. We modified this cost measure for rp-heaps to include, in addition to the number of links, the number of rank updates in key decreases and the number of unpaired half-trees in minimum deletions, all divided by $\lg n$. We ran the experiment on pairing heaps and type-2 multipass rp-heaps and measured the average round cost at regular intervals. The round cost for pairing heaps approaches 2.76 for $n = 2^{12}$ and 2.97 for $n = 2^{18}$, showing positive growth, as expected. The round cost for type-2 multipass rp-heaps approaches 2.36 for $n = 2^{12}$ and 2.32 for $n = 2^{18}$, showing slightly negative growth. In contrast, when we applied the same cost measure to the non-adversarial version of this experiment (tests 7 and 8 in Table 8.1, we could not distinguish between pairing heaps and type-2 multipass rp-heaps.

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. Our results build on previous work by Peterson, Høyer, and Kaplan and Tarjan, 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. In our preliminary experiments, rp-heaps are competitive with pairing heaps on typical input sequences and more efficient on worst-case sequences. Type-1 rp-heaps, although simple, are not simple to analyze. Indeed, we were surprised by our discovery that type-1 rp-heaps have the same efficiency as Fibonacci heaps; this is less surprising for type-2 rp-heaps.

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 5 is removed? More interestingly, can one obtain an $\mathrm{O}(1)$ amortized time bound for insert, meld, and key decrease and $\mathrm{O}(\log m)$ for minimum deletion (where m is the total number of insertions) if only $\mathrm{O}(1)$ rank changes are made after each key decrease? (See Section 7.)

Our paper is primarily theoretical, and our experiments with rank-pairing heaps are only preliminary. We hope to do a much more thorough investigation of their practical performance.

Acknowledgement. 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 SODA, 1996, pp. 52–58.
[3] GERTH STØLTING BRODAL AND CHRIS OKASAKI, *Optimal purely functional priority queues*, Journal of Functional Programming, 6 (1996), pp. 839–857.
[4] MARK R. BROWN, *Implementation and analysis of binomial queue algorithms*, SIAM J. Comput., (1978), pp. 298–319.
[5] TIMOTHY M. CHAN, *Quake heaps: a simple alternative to Fibonacci heaps*, 2009. 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. of the ACM, 31 (1988), pp. 1343–1354.

[8] R.D. DUTTON, *The weak-heap data structure*, tech. report, University of Central Florida, 1992.
[9] J. EDMONDS, *Optimum branchings*, J. Res. Nat. Bur. Standards, B71 (1967), pp. 233–240.
[10] AMR ELMASRY, *Violation heaps: A better substitute for Fibonacci heaps*, CoRR, abs/0812.2851 (2008).
[11] AMR ELMASRY, *Pairing heaps with* $\mathrm{O}(\log \log n)$ *decrease cost*, in SODA, 2009, pp. 471–476.
[12] MICHAEL L. FREDMAN, *On the efficiency of pairing heaps and related data structures*, J. ACM, 46 (1999), pp. 473–501.
[13] 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.
[14] MICHAEL L. FREDMAN AND ROBERT ENDRE TARJAN, *Fibonacci heaps and their uses in improved network optimization algorithms*, J. of the ACM, 34 (1987), pp. 596–615.
[15] MICHAEL L. FREDMAN AND DAN E. WILLARD, *Trans-dichotomous algorithms for minimum spanning trees and shortest paths*, J. Comput. Sys. Sci., (1994), pp. 533–551.
[16] HAROLD N. GABOW, ZVI GALIL, THOMAS H. SPENCER, AND ROBERT ENDRE TARJAN, *Efficient algorithms for finding minimum spanning trees in undirected and directed graphs*, Combinatorica, 6 (1986), pp. 109–122.
[17] LEO J. GUIBAS AND ROBERT SEDGEWICK, *A dichromatic framework for balanced trees*, in FOCS, 1978, pp. 8–21.
[18] BERNHARD HAEUPLER, SIDDHARTHA SEN, AND ROBERT ENDRE TARJAN, *Rank-pairing heaps*, in ESA, Amos Fiat and Peter Sanders, eds., 2009, pp. 659–670.
[19] Y. HAN AND M. THORUP, *Integer sorting in* $\mathrm{O}(n\sqrt{\log \log n})$ *expected time and linear space*, in FOCS, 2002, pp. 135–144.
[20] P. HØYER, *A general technique for implementation of efficient priority queues*, in ISTCS, 1995, pp. 57–66.
[21] HAIM KAPLAN, NIRA SHAFRIR, AND ROBERT E. TARJAN, *Meldable heaps and boolean union-find*, in STOC, 2002, pp. 573–582.
[22] H. KAPLAN AND R. E. TARJAN, *New heap data structures*, tech. report, Department of Computer Science, Princeton University, 1999.
[23] HAIM KAPLAN AND ROBERT ENDRE TARJAN, *Thin heaps, thick heaps*, ACM Trans. Alg., 4 (2008), pp. 1–14.
[24] DONALD E. KNUTH, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, 1973.
[25] ———, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, 1973.
[26] ANDREW M. LIAO, *Three priority queue applications revisited*, Algorithmica, 7 (1992), pp. 415–427.
[27] B.M.E. MORET AND H.D. SHAPIRO, *An empirical analysis of algorithms for constructing a minimum spanning tree*, WADS, (1991), pp. 400–411.
[28] HIROSHI NAGAMOCHI AND TOSHIHIDE IBARAKI, *Computing edge-connectivity in multigraphs and capacitated graphs*, J. Disc. Math., 5 (1992), pp. 54–66.
[29] G. L. PETERSON, *A balanced tree scheme for meldable heaps with updates*, Tech. Report GIT-ICS-87-23, School of Informatics and Computer Science, Georgia Institute of Technology, 1987.
[30] SETH PETTIE, *Towards a final analysis of pairing heaps*, in FOCS, 2005, pp. 174–183.
[31] JOHN T. STASKO AND JEFFREY SCOTT VITTER, *Pairing heaps: experiments and analysis*, Comm. ACM, 30 (1987), pp. 234–249.
[32] R. E. TARJAN, *Amortized computational complexity*, J. Alg. Disc. Methods, 6 (1985), pp. 306–318.
[33] VARIOUS, *The Fifth DIMACS Challenge—Priority Queue Tests*, 1996. http://www.cs.amherst.edu/ccm/challenge5/p_queue/index.html.
[34] JEAN VUILLEMIN, *A data structure for manipulating priority queues*, Comm. ACM, 21 (1978), pp. 309–315.