Rank-Pairing Heaps

Bernhard Haeupler[1], Siddhartha Sen[2,4], and Robert E. Tarjan[2,3,4]

[1] Massachusetts Institute of Technology, haeupler@mit.edu
[2] Princeton University, {sssix,ret}@cs.princeton.edu
[3] HP Laboratories, Palo Alto CA 94304

Abstract. We introduce the *rank-pairing heap*, a heap (priority queue) implementation 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 evolve to have arbitrary structure. 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.

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 and currently in no heap, 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 $\Delta > 0$.
- $delete(x, H)$: delete item x from heap H.

We break ties between equal keys using any total order of the items. We allow only binary comparisons of keys, and we study the amortized efficiency [26] of heap operations. 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.

[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.

Since a heap can be used to sort n numbers, the classical $\Omega(n \log n)$ lower bound [19, p. 183] on comparisons 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. For simplicity in stating bounds we assume $n \geq 2$. 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 heap operations takes $\mathrm{O}(1)$ amortized time. These bounds match the lower bound.

Many heap implementations have been proposed over the years. See e.g. [17]. We mention only those directly related to our work. The *binomial queue* of Vuillemin [28] supports all the heap operations in $\mathrm{O}(\log n)$ worst-case time per operation and performs well in practice [2]. Fredman and Tarjan [11, 12] invented the *Fibonacci heap* specifically to support key decrease operations in $\mathrm{O}(1)$ time, which allows efficient implementation of Dijkstra's shortest path algorithm [4, 12] and several other algorithms [6, 12, 13]. Fibonacci heaps support deletion of the minimum or of an arbitrary item in $\mathrm{O}(\log n)$ amortized time and the other heap operations in $\mathrm{O}(1)$ amortized time.

Several years later, Fredman et al. [10] introduced a self-adjusting heap implementation called the *pairing heap* that supports all the heap operations in $\mathrm{O}(\log n)$ amortized time. Fibonacci heaps do not perform well in practice [20, 21], but pairing heaps do [20, 21]. Fredman et al. conjectured that pairing heaps have the same amortized efficiency as Fibonacci heaps, but despite empirical evidence supporting the conjecture [20, 25], Fredman [9] 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. In contrast, the best known upper bound is $\mathrm{O}(2^{2\sqrt{\lg \lg n}})$ [24] amortized time [5].

These results motivated work to improve Fibonacci heaps and pairing heaps. Some of this work obtained better bounds but only by making the data structure more complicated. In particular, the bounds of Fibonacci heaps can be made worst-case: run-relaxed heaps [5] and fat heaps [16] achieve the bounds except for melding, which takes $\mathrm{O}(\log n)$ time worst-case, and a very complicated data structure of Brodal [1] achieves all the bounds worst-case. Also, Fredman's lower bound can be matched: very recently Elmasry [8] has proposed an alternative to pairing heaps that does not store subtree size information but takes $\mathrm{O}(\log \log n)$ amortized time for a key decrease.

Working in a different direction, several authors proposed data structures with the same amortized efficiency as Fibonacci heaps but intended to be simpler. Peterson [23] gave a structure based on AVL trees. Høyer [14] 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 [17] filled a lacuna in Høyer's presentation of this heap and gave a related structure, the *thin heap*. Independently of our own work but concurrently, Elmasry [7] developed *violation heaps* and Chan [3] *quake heaps*.

In all these structures, the trees representing the heap satisfy a balance property. As a result, a key decrease must in general do restructuring to restore balance. 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, allowing trees to evolve to have arbitrary structure. We store a rank for each node and only combine trees whose roots

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

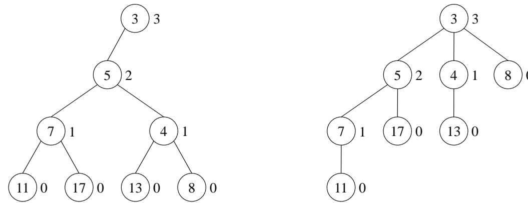

Fig. 1. A half-ordered half tree and the equivalent heap-ordered tree.

have equal rank. After a key decrease, rank changes (decreases) can propagate up the tree. 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. Our initial experiments suggest that rank-pairing heaps compare well with pairing heaps in practice.

The remainder of our paper has six sections. Section 2 describes a one-pass version of binomial queues on which we base our data structure. Section 3 extends this version to obtain two types of rank-pairing heaps, which we analyze in Section 4. Section 5 shows that some even simpler ways of implementing key decrease have worse amortized efficiency. Section 6 describes our initial experiments comparing rank-pairing heaps with pairing heaps. Section 7 concludes and mentions some open problems.

2 One-Pass Binomial Queues

We represent a heap by a set of half-ordered half trees [10, 18] whose nodes are the items in the heap. A *half tree* is a binary tree whose right subtree is missing. A *half-ordered* binary tree is a binary tree in which each node has a key, such that the key of a node is less than that of all nodes in its left subtree. Thus in a half-ordered half tree the root has minimum key. A half-ordered half tree is just the binary tree representation [18] of a heap-ordered tree, corresponding to the latter's implementation. (See Figure 1.)

We represent a half tree by storing with each node x pointers to $left(x)$ and $right(x)$, its left and right child, respectively. We represent a set of half trees 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. This representation allows us to find the minimum, insert a new half tree, or catenate two such lists in $O(1)$ time.

The basic operation on half trees is *linking*, which combines two half trees into one, in $O(1)$ time. To link two half trees with roots x and y, compare the keys of x and y. Assume x has smaller key; the other case is symmetric. Detach the left subtree of x and make it the right subtree of y. Then make the tree rooted at y the left subtree of x.

To make our data structure efficient, we restrict linking by using *ranks*. A *ranked half tree* is a half tree in which each node has an integer *rank*. The *rank* of a half tree is the rank of its root. We only link half trees of equal rank. After a link, we increase the rank of the new root by one. (See Figure 2.)

We implement the various heap operations 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 of rank zero and insert it into the list of roots, in first or second position depending on whether it has minimum key or not. To meld two

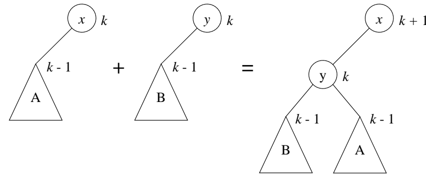

Fig. 2. A link of two half trees of rank k. Ranks are to the right of nodes.

heaps, catenate their lists of roots, making the root of minimum key first on the new list. Each of these operations takes $\mathrm{O}(1)$ time worst-case.

To delete the minimum, take apart the half tree rooted at the node of minimum key by deleting this node and walking down the path from its left (only) child through right children, making each node on the path together with its left subtree into a half tree. The set of half trees now consists of the original half trees, minus the one disassembled, plus the new ones formed by the disassembly. Group these half trees into a maximum number of pairs of equal rank and link the half trees in each pair. To do this, maintain a set of buckets, one per rank. Process each half tree in turn by putting it into 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 root of the new half tree to the list representing the updated heap, leaving the bucket empty. Once all the half trees have been processed, add the half trees remaining in buckets to the list of roots, leaving all the buckets empty.

This data structure is a one-pass version of binomial queues. Linking only half trees of equal rank guarantees that every child has rank exactly one less than that of its parent, and every half tree is *perfect*: its left subtree is a perfect binary tree. Thus a half tree of rank k contains exactly 2^k nodes, and the maximum rank is at most $\lg n$.

In the original version of binomial queues, there is never more than one tree per rank. Maintaining this invariant requires doing links during insertions and melds and doing additional links during minimum deletions, until no two half trees have equal rank. The original version has a worst-case time bound of $\mathrm{O}(1)$ for make-heap and $\mathrm{O}(\log n)$ for the other operations. Since we are interested in amortized efficiency, we prefer to avoid linking during insertions and melds and to link lazily during minimum deletions; extra links of equal-rank half trees done during minimum deletions do not affect our bounds.

To analyze one-pass binomial queues, we define the potential of a heap to be the number of half trees it contains. A make-heap, find-min, or meld operation takes $\mathrm{O}(1)$ time and does not change the potential. An insertion takes $\mathrm{O}(1)$ time and 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 trees and thus the potential by at most $\lg n$. Let h be the number of half trees after the disassembly but before the links. The total time for the minimum deletion is $\mathrm{O}(h+1)$, including the time to pair the trees by rank. There are at least $(h - \lg n - 1)/2$ links, reducing the potential by at least this amount. If we scale the running time so that it is at most $h/2 + \mathrm{O}(1)$, then the amortized time of the minimum deletion is $\mathrm{O}(\log n)$.

3 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. To find an item's heap takes $\omega(1)$ time [15].

We shall modify one-pass binomial queues to support key decrease, obtaining *rank-pairing heaps*, or *rp-heaps*. We make two changes to the data structure. First, we need to provide access from each node to its parent as well as to its children. We add parent pointers, resulting in three pointers per node; this can be reduced to two pointers if we are willing to trade time for space, as observed by Fredman et al. [10].

Second, we relax the constraint on ranks. Let $p(x)$ and $r(x)$ be the parent and rank of node x, respectively. If x is a child, the *rank difference* of x is $r(p(x)) - r(x)$. A node of rank difference i is an *i-child*; a node whose children have rank differences i and j is an *i, j-node*. The latter definition does not distinguish between left and right children. We adopt the convention that a missing child has rank -1.

We shall define two types of rank-pairing heaps. In both, every child of a root is a 1-child, and every leaf has rank zero and is thus a 1,1-node. In a *type-1 rank-pairing heap*, every child is a 1,1-node or a $0, i$-node for some $i > 0$. We call this the *rank rule*. Ranks give a lower bound but not an upper bound on subtree sizes.

Lemma 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.*

Proof. The second part of the lemma implies the first part. We prove the second part by induction on the height of a node. A leaf has rank zero and satisfies the second part. Consider a non-root x of rank k whose children satisfy the second part. If x is a $0, i$-node, by the induction hypothesis it has at least $2^{k+1} - 1$ descendants. If x is a 1,1-node, by the induction hypothesis it has at least $2(2^k - 1) + 1 = 2^{k+1} - 1$ descendants. □

We implement make-heap, find-min, insert, meld, and delete-min exactly as on one-pass binomial queues. Links preserve the rank rule: if a link makes a root into a child, the new child is a 1-child and a 1,1-node. (See Figure 2.)

We implement key decrease as follows. (See Figure 3.) To decrease the key of item x in rp-heap H by Δ, subtract Δ from the key of x. If x is a root, make it first on the root list if it now has minimum key, and stop. Otherwise, let $y = right(x)$; detach the subtrees rooted at x and y; reattach the subtree rooted at y in place of the original subtree rooted at x; add x to the list of roots, making it first if it has minimum key; set $r(x) = r(left(x)) + 1$. There may now be a violation of the rank rule at $p(y)$, whose new child, y, may have lower rank than x, the node it replaces. To restore the rank rule, let $u = p(y)$ and repeat the following step until it stops:

Decrease rank (*type 1*): If u is the root, set $r(u) = r(left(u)) + 1$ and stop. Otherwise, let v and w be the children of u. Let k equal $r(v)$ if $r(v) > r(w)$, $r(w)$ if $r(w) > r(v)$, or $r(w) + 1$ if $r(v) = r(w)$. If $k = r(u)$, stop. Otherwise, let $r(u) = k$ and $u = p(u)$.

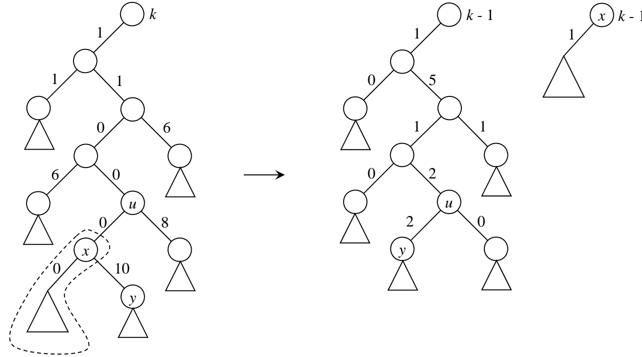

Fig. 3. Key decrease in a type-1 rp-heap.

If u breaks the rank rule, it obeys the rule after its rank decreases, but $p(u)$ may not. Rank decreases propagate up along a path through the tree until all nodes obey the rule. Each successive rank decrease is by the same or a smaller amount.

Before analyzing type-1 rp-heaps, we introduce a relaxed version. In a *type-2 rank-pairing heap*, every child is a 1,1-node, a 1-2-node, or a $0, i$-node for some $i > 1$. The implementations of the heap operations are identical to those of type-1 rp-heaps, except for key decrease, which is the same except that the rank decrease step becomes the following:

Decrease rank (*type 2*): If u is the root, set $r(u) = r(left(u)) + 1$ and stop. Otherwise, let v and w be the children of u. Let k equal $r(v)$ if $r(v) > r(w) + 1$, $r(w)$ if $r(w) > r(v) + 1$, or $\max\{r(v) + 1, r(w) + 1\}$ otherwise. If $k = r(u)$, stop. Otherwise, let $r(u) = k$ and $u = p(u)$.

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 $k > 1$, and by $\phi = (1 + \sqrt{5})/2$ the golden ratio.

Lemma 2. *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.*

Proof. The inequality $F_{k+2} \geq \phi^k$ is well-known [18, p. 18]. The second part of the lemma implies the first part. We prove the second part by induction on the height of a node. A leaf has rank zero and satisfies the second part of the lemma. A missing node also satisfies the second part. Consider a non-root x of rank k whose children satisfy the second part. If x is a $0, i$-node, by the induction hypothesis it has at least $F_{k+3} - 1$ descendants. If x is a 1,1-node, by the induction hypothesis it has at least $2(F_{k+2} - 1) + 1 \geq F_{k+3} - 1$ descendants. If x is a 1,2-node, by the induction hypothesis it has at least $F_{k+2} - 1 + F_{k+1} - 1 + 1 = F_{k+3} - 1$ descendants. □

The worst-case time for a key decrease in an rp-heap of either type is $\Theta(n)$, as in Fibonacci heaps. We can reduce this to $\mathrm{O}(1)$ by delaying key decreases until the next minimum deletion and maintaining the set of nodes that might have minimum key. This complicates the data structure, however, and may worsen its performance in practice.

4 Amortized Efficiency of Rank-Pairing Heaps

We begin by analyzing type-2 heaps, which is easier than analyzing type-1 heaps. We define the potential of a node to be the sum of the rank differences of its children, minus one if it is a 1,2- or $0, i$-node, minus two if it is a 1,1-node. That is, its potential 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. We define the potential of a heap to be the sum of the potentials of its nodes.

Theorem 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. The disassembly increases the potential by at most the number of 1,1-nodes that become roots. By Lemma 2 there are at most $\log_\phi n$ such conversions, so 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 $\mathrm{O}(h+1)$ time. Scale this time to be at most $h/2+\mathrm{O}(1)$. 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 decreases the potential by at least $h/2 - \mathrm{O}(\log n)$, which implies that its amortized time is $\mathrm{O}(\log n)$.

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 $\mathrm{O}(1)$ actual time and does not change the potential. Otherwise, let $u_0 = left(x)$, $u_1 = x$, and $u_2, \ldots, u_k$ be the successive nodes at which rank decrease steps take place, omitting the root if its rank decreases ($u_j = p(u_{j-1})$ for $2 \leq j \leq k$). 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 $u_1, \ldots, u_k$. At most two of these nodes are 1,1-nodes before the key decrease; if there is a second, it must be u_k, which must become a 1,2-node. The sum of the rank differences of $v_1, u_1, \ldots, u_{k-1}$ before the key decrease telescopes to $r(u_k) - r(v_1)$; the sum of the rank differences of $v_1, u_2, ..., u_{k-1}$ after the key decrease telescopes to $r'(u_k) - r'(v_1) \leq r(u_k) - r(v_1)$ since $r'(u_k) \leq r(u_k)$ and $r'(v_1) = r(v_1)$. The key decrease reduces the rank difference of each v_j, $2 \leq j < k$ by at least one. It follows that the key decrease reduces the potential by at least $k - 6$; the "-6" accounts for one additional root and up to two fewer 1,1-nodes. If we scale the time of a rank decrease step so that it is at most one, the amortized time of the key decrease is $\mathrm{O}(1)$. ☐

To analyze type-1 rp-heaps, we assign potentials based on the children of a node. We call a non-leaf 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 the potential of a 0,1-node is zero if it is good or four if it is bad, of a 1,1-node is one if it is good or five if it is bad, of a root is two if it is good or six if it is bad, and of a $0, i$-node

with $i > 1$ is $i + 3$. We define the potential of a heap to be the sum of the potentials of its nodes. If we restrict the links during a minimum deletion to preferentially pair the half trees produced by the disassembly, the following theorem holds:

Theorem 2. *The amortized time for an operation on a type-1 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 3/2. Hence each of these operations takes $\mathrm{O}(1)$ amortized time.

Our analysis of minimum deletion relies on the fact that for each rank at most one of the half trees formed by the disassembly is not linked to another such half tree. During the disassembly, we give each new root of rank one or more a potential of four whether it is good or bad. We give the correct potential (two if good, six if bad) to each root of a half tree formed by a link; and, once all half trees formed by the disassembly have been inserted into buckets, we give the correct potential to the root of each such half tree remaining in a bucket. This correction increases the potential by two for each such root that is bad. We charge these two units against the rank of the bad root.

Consider the effect of the disassembly on the potential. Deletion of the root 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. Each bad node that becomes a root already has the four units of potential it needs as a root. Consider a good node x that becomes a root. There are three cases. If x is a 1,1-node, it needs three additional units of potential as a root. We charge these to $r(x)$. If x is a 0,1-node whose right child is a 0-child, we walk down the path of right children from x until reaching a node y that is either bad or a leaf. Each node along the path must be a 1,1-node. If y is a bad 1,1-node, it has five units of potential, four of which it needs as a root, leaving one for x. We charge the remaining three needed by x to $r(y)$. If y is a leaf, it has rank zero; we charge the units needed by x (3/2 or 4) to rank zero. Node y is reached only from one such node x. If x is a 0,1-node whose left child is a 0-child, we charge the four units x needs as a root to $r(x)$. In this case x is the last new root of its rank; since x is good, its rank is not charged two units to correct the rank of a bad root. Each rank can only be charged once for a new root. The total increase in potential caused by the disassembly and correction of the potential is thus at most $5 \lg n$: each rank up to the maximum rank minus one can be charged two units for a correction and three for a new root, or zero for a correction and four for a new root.

Each link of two rank-0 roots reduces the potential by one. Each link of two roots of positive rank converts a root into a 1,1-node and makes the remaining root good. After the link, these two nodes have total potential at most seven. Before the link, they have potential at least eight (four plus four or at least two plus six), unless they were both good before the link, in which case the new 1,1-node is good after the link and the link reduces the total potential of the two nodes from four to three. Thus each link reduces the potential by one. 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 $h/2+\mathrm{O}(1)$. 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 decreases the potential by at least $h/2-(11/2)\lg n-1/2$, which implies that its amortized time is $\mathrm{O}(\log n)$.

The analysis of a key decrease at a node x is just like that for type-2 heaps, except we must show that the key decrease can make only $\mathrm{O}(1)$ 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, increasing the potential by at most four, and it can create a new root, increasing the potential by two. An argument like that in the proof of Theorem 1 shows that if there are k rank decrease steps, the potential drops by $k - \mathrm{O}(1)$. Thus the key decrease takes $\mathrm{O}(1)$ amortized time. □

5 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 describing two possible methods that fail. The first method allows arbitrarily negative but bounded positive rank differences. With such a rank rule, the rank decrease process need only examine the ancestors of the node whose key decreases, not their siblings. This method can take $\Omega(\log n)$ amortized time per key decrease, however. The second, even simpler method spends only $\mathrm{O}(1)$ time worst-case on each key decrease. By doing enough operations, however, one can build a half tree of each possible rank up to a rank that is $\omega(\log n)$. Then, repeatedly doing an insertion of minimum key followed by a minimum deletion will result in each minimum deletion taking $\omega(\log n)$ time. We omit the details of these counterexamples.

One limitation of the second construction is that building the initial trees takes a number of operations exponential in the size of the heap. Thus it is not a counterexample to the following question: is there a fixed d such that if each key decrease performs 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 the answer is no for $d = 0$; that is, if key decrease changes the rank only of the node whose key decreases. 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, 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 repetitions of an insertion of minimum key followed by a minimum deletion. 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 4.) Do a key decrease on each node of P. Now there is a collection of half trees of rank 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

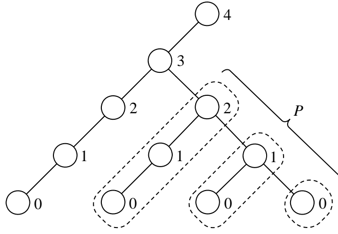

Fig. 4. A half tree of rank $k = 4$ buildable in $O(k^3)$ operations if key decreases do not change ranks. Key decreases on the right children detach the circled subtrees.

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 $\Omega(m^{1/3})$. Each successive cycle of an insertion followed by a minimum deletion takes $\Omega(m^{1/3})$ time.

6 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 [25] on worst-case sequences; Stasko and Vitter [25] claimed the latter outperforms other versions on the class of worst-case sequences we used. We compared them to four versions of rp-heaps: type-1 and type-2, performing as many links as possible during minimum deletion ("multipass") and performing only a single pass of links ("one-pass"). All heap implementations were written in C and compiled with the `gcc -O2` option; all tests were performed on a 2.13 Ghz Intel CPU running Windows Vista.

For typical input sequences, we performed three sets of experiments (Table 1). Our measure of performance is the total number of field updates performed by each heap implementation. The first set of experiments consists of publicly available heap operation sequences [27] obtained by running Dijkstra's shortest paths algorithm [4] on graphs that maximize heap size and the number of key decreases, respectively; the Nagamochi-Ibaraki minimum-cut algorithm [22] on a random graph; and heapsort. The second set consists of heap operation sequences obtained by running two-way Dijkstra between random pairs of nodes on real road networks. The third set tests key decreases by running $2n$ rounds of the following operations on a binomial heap of n nodes: an insertion, followed by $\lg n - 1$ key decreases, followed by a minimum deletion.

The results in Table 1 show that rp-heaps performed fewer field updates than pairing heaps on sequences with many key decreases (2,3,7,8), and more updates on sequences with few key decreases. The multipass versions 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), experienced greater overhead in maintaining longer lists of half trees during minimum deletions.

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 decreases)	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

Table 1. Performance of pairing heaps versus rp-heaps on typical input sequences. n is the number of vertices or items in the heap; m is the number of arcs. All results are in millions of updates.

To investigate the worst-case behavior of key decreases, we ran Fredman's [9] version of an experiment of Stasko and Vitter [25]. The experiment is identical to the third set of experiments in Table 1, but uses information-theoretic heuristics in the heap operations. In particular, the winner of a link is always the root of the larger tree, and the candidates for key decreases are chosen to detach links of high efficiency, where efficiency is defined as the ratio of the child and parent tree sizes prior to linking. (See [9].) The cost of a round is the number of links performed—plus, for rp-heaps, the number of rank updates in a key decrease and the number of unpaired half trees in a minimum deletion—divided by $\lg n$. The round cost for pairing heaps converges to 2.76 and 2.97 for $n = 2^{12}$ and $n = 2^{18}$, respectively, showing positive growth. The round cost for type-2 multipass rp-heaps converges to 2.36 and 2.32, showing slightly negative growth.

7 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 better on worst-case sequences. Type-1 rp-heaps, although simple, are not simple to analyze.

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 restrictions on linking in Section 4 are 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 a key decrease? (See Section 5.)

Acknowledgement

We thank Haim Kaplan and Uri Zwick for 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. Brodal. Worst-case efficient priority queues. In *SODA*, pages 52–58, 1996.
2. M. R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM J. Comput.*, pages 298–319, 1978.
3. T. M. Chan. Quake heaps: a simple alternative to Fibonacci heaps, 2009.
4. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1:269–271, 1959.
5. J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation. *Comm. ACM*, 31(11):1343–1354, 1988.
6. J. Edmonds. Optimum branchings. *J. Res. Nat. Bur. Standards*, B71:233–240, 1967.
7. A. Elmasry. Violation heaps: A better substitute for Fibonacci heaps. *CoRR*, 2008.
8. A. Elmasry. Pairing heaps with $\mathrm{O}(\log \log n)$ decrease cost. In *SODA*, pages 471–476, 2009.
9. M. L. Fredman. On the efficiency of pairing heaps and related data structures. *J. ACM*, 46(4):473–501, 1999.
10. M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.
11. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *FOCS*, pages 338–346, 24–26 1984.
12. M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.
13. 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(2):109–122, 1986.
14. P. Høyer. A general technique for implementation of efficient priority queues. In *ISTCS*, pages 57–66, 1995.
15. H. Kaplan, N. Shafrir, and R. E. Tarjan. Meldable heaps and boolean union-find. In *STOC*, pages 573–582, 2002.
16. H. Kaplan and R. E. Tarjan. New heap data structures. Technical Report TR-597-99, Princeton Univ., 1999.
17. H. Kaplan and R. E. Tarjan. Thin heaps, thick heaps. *ACM Trans. Alg.*, 4(1):1–14, 2008.
18. D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, 1973.
19. D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1973.
20. A. M. Liao. Three priority queue applications revisited. *Algorithmica*, 7:415–427, 1992.
21. B. Moret and H. Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. *WADS*, pages 400–411, 1991.
22. H. Nagamochi and T. Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *J. Disc. Math.*, 5(1):54–66, 1992.
23. G. L. Peterson. A balanced tree scheme for meldable heaps with updates. Technical Report GIT-ICS-87-23, Georgia Inst. of Tech., 1987.
24. S. Pettie. Towards a final analysis of pairing heaps. In *FOCS*, pages 174–183, 2005.
25. J. T. Stasko and J. S. Vitter. Pairing heaps: experiments and analysis. *Comm. ACM*, 30(3):234–249, 1987.
26. R. E. Tarjan. Amortized computational complexity. *J. Alg. Disc. Methods*, 6:306–318, 1985.
27. Various. The Fifth DIMACS Challenge—Priority Queue Tests, 1996.
28. J. Vuillemin. A data structure for manipulating priority queues. *Comm. ACM*, 21(4):309–315, 1978.