New Heap Data Structures

Haim Kaplan* Robert E. Tarjan †

May 14, 1998

ABSTRACT

We describe two new heap data structures. Our first structure, called the *thin heap*; is closely related to Fibonacci heaps [7] and has the same asymptotic performance. Specifically, *make-heap*, *insert*, *find-min*, *decrease-key*, and *meld* take $O(1)$ amortized time, and *delete-min* and *delete* take $O(\log n)$ amortized time. The advantage of thin heaps over Fibonacci heaps is that the heap-ordered trees being maintained have more structure and require fewer fields per node, but the operations are as simple as for Fibonacci heaps. In a practical implementation, the time and space constant factors of thin heaps should thus be better than those of Fibonacci heaps.

The second structure, called the *fat heap*, achieves the same asymptotic performance as the run-relaxed heaps described by Driscoll et al. [6] and offers an alternative to it in all its applications. Specifically, *make-heap*, *insert*, *find-min*, and *decrease-key* take $O(1)$ worst-case time, and *delete*, *delete-min*, and *meld* take $O(\log n)$ worst-case time. Our data structure is much simpler than run-relaxed heaps.

There exists a well-known connection between binomial queue operations and arithmetic

*AT&T-Labs, 180 Park Ave, Florham Park, NJ 07932. hkl@research.att.com

†Department of Computer Science, Princeton University, Princeton, NJ 08544 and InterTrust Technologies Corporation, Sunnyvale, CA 94086. Research at Princeton University partially supported by the NSF, Grant No. CCR-9626862. ret@cs.princeton.edu

operations on binary numbers. Our fat heaps exploit this connection and achieve their performance by incorporating two redundant counters.

1 Introduction

A *heap* is an abstract data structure consisting of a set of *items*, each with a real-valued *key*, subject to the following operations:

make-heap: Return a new, empty heap.

insert(i, h): Insert a new item i with predefined key into heap h.

find-min(h): Return an item of minimum key in heap h. This operation does not change h.

delete-min(h): Delete an item of minimum key from heap h and return it.

meld(h_1, h_2): Return the heap formed by taking the union of the item-disjoint heaps h_1 and h_2. This operation destroys h_1 and h_2.

decrease-key(Δ, i, h): Decrease the key of item i in heap h by subtracting the nonnegative real number Δ. This operation assumes that the position of i in h is known.

delete(i, h): Delete item i from heap h. This operation assumes that the position of i in h is known.

Vuillemin [13] invented a class of heaps, called *binomial queues*, that support all the heap operations in $O(\log n)$ worst-case time. Subsequently, Fredman and Tarjan [7] invented *Fibonacci heaps*, which support *find-min*, *insert*, *decrease-key* and *meld* in $O(1)$ amortized time, and *delete-min* and *delete* in $O(\log n)$ amortized time. Fibonacci heaps allow an optimal implementation of Dijkstra's shortest path algorithm and are useful in many other algorithms (see [7]).

A Fibonacci heap is a collection of item-disjoint heap-ordered trees. The definition of a Fibonacci heap does not impose any explicit constraint on the structure of the trees; the only constraint is implicit in the way the trees are manipulated. In a Fibonacci heap each node contains a pointer to its parent and a pointer to one of its children. The children of a node are doubly linked in a circular list. Each node is either marked or unmarked.

Our first result is a new class of heaps that we call *thin heaps*. Thin heaps satisfy the same time bound for each heap operation as Fibonacci heaps. The main advantage that thin heaps offer is the elimination of parent pointers and mark fields from the nodes, without complicating any operation. Another advantage of thin heaps is that the trees being maintained have an explicit binomial structure. Therefore we expect thin heaps to have better time and space constant factors in practice than Fibonacci heaps.

Since the invention of Fibonacci heaps, some effort has been devoted to a search for a structure that achieves time bounds similar to those of Fibonacci heaps but worst-case rather than amortized. Heap structures with good worst-case bounds are important for real-time applications. They also play a role in the design of parallel algorithms in which each processor maintains a heap and they must all be synchronized.

Driscoll et al. [6] described *run-relaxed heaps*, which satisfy the same bounds as Fibonacci heaps in the worst-case for all operations but *meld*. Run-relaxed heaps do support *meld*, but in $O(\log n)$ worst-case time rather than the desired $O(1)$ time. Driscoll et al. [6] apply their relaxed heaps to obtain an optimal parallel algorithm for shortest paths on a directed graph with nonnegative edge lengths on an EREW PRAM with $p \leq m/(n \log n)$ processors, where $m \geq n \log n$ is the number of arcs in the graph and n is the number of vertices. Recently, Brodal [3] described the first heap structure with the same performance as Fibonacci heaps but in the worst-case for all operations including meld.

The main drawback of run-relaxed heaps and Brodal's heaps is their complexity – in contrast to the simplicity of binomial queues and Fibonacci heaps. A run-relaxed heap consists of a collection of heap-ordered binomial queues [13] with $O(\log n)$ violations in the heap order. An auxiliary structure keeps track of the violations. The implementation of *decrease-key* uses a rather complicated set of transformations to reduce the number of violations by one in $O(1)$ time. Brodal's structure is even more complicated. It maintains a heap-ordered pair of trees with at most a linear number of violations in the heap order. Each element x has two sets of violations associated with it, each of size at most $O(\log n)$. Several external data structures (essentially redundant counters) are used to maintain the children of the roots and the violations associated with the roots. The needed violation-reducing transformation imposes several technical invariants on the structure. This transformation also complicates the mechanism required to maintain the children of roots, since it may have to cut off such a child of a given rank and reincorporate it into the structure.

Our second data structure, which we call *fat heaps*, matches the performance of run-relaxed heaps but is much simpler. A fat heap is a collection of heap-ordered trees corresponding to the digits of a redundant ternary counter, together with a redundant binary counter that maintains a set of heap-order violations. We expect fat heaps to be efficient in practice, based on evidence that binomial heaps are efficient in practice [4]: fat heaps have better asymptotic behavior for *decrease-key* and *insert* with few additional complications.

In run-relaxed heaps as well as in Brodal's structure, a violation of rank r is a node of rank r that violates the heap order: instead of being greater, it is smaller than its parent. Fibonacci heaps also have an implicit notion of a rank r violation: a rank r child is missing from a node of rank greater than r. It is an interesting question whether these two apparently different notions of a violation are equivalent in power. A property of fat heaps is that they can be implemented using either of these notions of a violation.

We maintain a more structured violation list than in run-relaxed heaps. In the heap structure we describe, there can be at most two violations at any particular rank at any particular time. In contrast, a run-relaxed heap can have $O(\log n)$ violations at one particular rank; the only restriction is a total of $O(\log n)$ violations.

An essential mechanism of fat heaps is redundant binary and ternary counters. Redundant binary counters (base 2) were first described by Knuth and Clancy [5] and then used by Guibas et al. [8] to construct finger search trees. More recently, Kaplan and Tarjan used redundant binary counters to design purely functional data structures such as catenable deques [9] and fast catenable finger search trees [10]. Brodal [1, 3] used redundant binary counters to design fast meldable heaps, and Brodal and Okasaki [2] used redundant binary counters to construct purely functional heaps. The heap data structure described by Brodal in [3] and the finger search tree data structure of Kaplan and Tarjan [10] both exploit the possibility of incrementing any digit in the counter in constant worst-case time. For the other applications, a constant-time increment to the least-significant digit suffices. Brodal [3] uses arrays to represent the redundant binary counters, while Kaplan and Tarjan use lists [10]. With the addition of one extra digit, Kaplan and Tarjan's counter also supports constant-time decrement of any digit.

The reminder of this paper consists of three sections. In section 2 we present thin heaps, and in section 3 fat heaps. Section 4 contains some closing remarks.

2 Thin Heaps

A Fibonacci heap is a collection of item-disjoint heap-ordered trees. The definition of a Fibonacci heap imposes no explicit constraints on the structure of the trees; the only constraints are implicit in the way the trees are manipulated. In contrast, a binomial heap consists of a logarithmic number of item-disjoint heap-ordered binomial trees (to be defined below). In this section we show how to obtain a heap with the same behavior as Fibonacci heaps, but in which the trees are essentially binomial. The structure we present also requires fewer fields per node than Fibonacci heaps.

For any integer $k \geq 0$ we define a binomial tree of rank k, denoted by B_k, as follows. A *binomial tree of rank zero* is a single node. For any $k > 0$, a *binomial tree of rank k* is obtained by linking two binomial trees of rank $k-1$ so that the root of one becomes the leftmost child of the root of the other. The *rank of a node x* in a binomial tree is defined to be the rank of the binomial subtree rooted at x.

A *thin tree* is a binomial tree in which each nonroot and nonleaf node may optionally have lost its leftmost child. We offer the following more elaborate (and more precise) definition. A *thin tree* is a rooted, ordered tree in which each node x has a nonnegative integer *rank*, denoted by $rank(x)$, such that the following three conditions hold, where $degree(x)$ is the number of children of x:

(1) For any node x, $rank(x) = degree(x)$ or $rank(x) = degree(x) + 1$. In the latter case, we say x is *marked*; in the former, *unmarked*.

(2) The root is unmarked.

(3) For any node x, the ranks of the children of x, from rightmost to leftmost, are $0, 1, 2, \ldots, degree(x) - 1$.

Observe that if we link two thin trees whose roots are of equal rank, by making one root the new leftmost child of the other root and increasing the rank of the single remaining root by one, then the result is a thin tree.

Our heap structure, the *thin heap*, consists of a set of heap-ordered thin trees. By *heap-ordered* we mean that each node corresponds to an item, and each child node has a key no smaller than that of its parent. We maintain the roots of the thin trees in a doubly-linked list. We maintain a pointer to a root corresponding to an item of minimum key; we call this root the *minimum node* of the heap. We also maintain, for each node in the heap, a doubly-linked list of its children. In addition, each node points to its leftmost child, if it has one, and each leftmost child points to its parent instead of its (nonexistent) left sibling. Finally, each node stores its rank. Observe that we can tell whether a node x is marked by comparing its rank to that of its leftmost child y: x is marked if $rank(x) = rank(y) + 2$ (or $rank(x) = 1$ and x has no leftmost child). If x is marked we can unmark it merely by decreasing its rank by one (although this may cause a violation of one of the rank constraints at the parent of x). The representation proposed for thin heaps saves a pointer per node over the standard representation of Fibonacci heaps: in the latter every node has a pointer to its parent, as well as pointers to its left and right siblings and to its leftmost child.

We perform the various heap operations on thin heaps as follows. The implementation of *make-heap*, *find-min*, *meld*, and *delete-min* is exactly as for Fibonacci heaps. We repeat it here for completeness. To perform *make-heap*, we return a null pointer. To perform *find-min(h)*, we return the item in the minimum node of h. To perform $meld(h_1, h_2)$, we combine the root lists of h_1 and h_2 into a single list, and return as the minimum node of the new heap either the minimum node of h_1 or the minimum node of h_2, whichever has smaller key (breaking a tie arbitrarily).

To perform *delete-min*(h), we remove the minimum node, say x, from h. Then we combine the list of children of x with the list of roots of h other than x and repeat the following *linking step* until it no longer applies: find any two trees whose roots have equal rank and link them by making the root with larger key the new leftmost child of the root of smaller key (breaking a tie arbitrarily) and increasing the rank of the new root by one. Once there are no two trees with roots of equal rank, we form a list of the remaining roots, in the process finding a root of minimum key to serve as the minimum node of the new heap. To find links to perform in $O(1)$ time per link, we use a temporary array indexed by rank to store tree roots. See [7]. We can avoid the use of random-access memory and implement the entire algorithm on a pointer machine by having each node of rank k point to a global *rank node* for rank k, and having the rank node for k point to the rank nodes for $k+1$ and $k-1$. We leave the details as an exercise.

The implementations of *decrease-key* and *delete* differ from their implementations for Fibonacci heaps. In a Fibonacci heap, a *decrease-key* operation can cause a sequence of *cascading cuts*, in which successive nodes along an ancestral path are detached from their parents. In thin heaps, we perform a related but different sequence of cuts, possibly with some intervening rank reductions.

We perform *decrease-key*(Δ, i, h) as follows. Let x be the node containing i. Subtract Δ from the key of i. If x is not a tree root, let y be the left sibling of x, or the parent of x, if x has no left sibling. Delete x from the list of children containing it, making it a new tree root, and updating the minimum node if necessary. Set the rank of x to one greater than that of its leftmost child, or to zero if it has no children. This may create a violation of one of the rank rules at y. Specifically, if x was previously the right sibling of y, the new right sibling of y may have a rank two less than that of y, a violation of (3). We call this a *sibling violation*. If x was previously a child of y and y was previously marked, y will now have a rank three greater than that of its leftmost child, a violation of (1). We call this a *child violation*. We repeat the following step, which will repair the violation at y but may create a new violation at the left sibling or parent of y, until there is no violation:

<u>Case 1</u> (node y is a root): Set the rank of y to one greater than that of its leftmost child, or zero if it has no children. This case is terminal.

<u>Case 2</u> (sibling violation): Node y has rank one and no right sibling, or its right sibling has rank two less than that of y. Let z be the left sibling of y, or its parent if it has no left sibling.

<u>Case 2a</u>: Node y is unmarked. Remove the leftmost child of y, say w, and insert it after y in the list of children containing y, making it the new right sibling of y. Since y is unmarked, w will have rank one less than the rank of y, and this repairs the violation while marking y. This case is terminal.

<u>Case 2b</u>: Node y is marked. Decrease the rank of y. This will unmark y and repair the violation

at y, but may create a new violation at z. This case is possibly non-terminal.

Case 3 (child violation): Node y has rank three greater than that of its leftmost child, or it has rank two and no children. Let z be the left sibling of y, or its parent if it has no left sibling. Delete y from the list of children containing it, making it a new tree root. Set its rank to one greater than that of its leftmost child, or to zero if it has no children. Replace y by z. This case is possibly non-terminal.

Based on the discussion in the description of the algorithm, it is straightforward to verify that *decrease-key* produces a valid thin heap.

We perform *delete(i,h)* as follows. Let x be the node containing i. If x is the minimum node of h, proceed as in *delete-min*. Otherwise, let y be the left sibling of x, or the parent of x if x has no left sibling, or null if x is a root. Remove x from the list of children containing it, and combine the list of children of x with the list of tree roots, updating the rank of each new root to be one greater than that of its leftmost child. If y is not null, apply to y the same violation-repair process as in *decrease-key*.

The analysis of thin heaps is essentially the same as the analysis of Fibonacci heaps. It uses the *potential function* technique [12]. We assign to each possible collection of heaps an integer called the *potential* of the heaps. This number equals the number of trees plus twice the number of marked nodes. The *amortized time* of a heap operation is defined to be its actual running time plus the net increase in the potential caused by the operation. With this definition, the total actual time of a sequence of heap operations, starting with no heaps, is at most the sum of the amortized times of the operations [12].

Both the actual and the amortized times of *find-min*, *insert*, and *meld* are obviously $O(1)$: an insertion increases the number of trees by one; the other operations do not affect the number of trees. If we charge one unit of time for each linking step, then the amortized time of the *delete-min* operation is no greater than the maximum rank of any node in the heap, because each linking step reduces the potential by one. The following lemma shows that every thin binomial tree has size exponential in the rank of its root. This implies that the amortized time of *delete-min* on an n-item heap is $O(\log n)$.

Lemma 2.1 *A node of rank k in a thin tree has at least $F_k \geq \phi^{k-1}$ descendants, including itself, where F_k is the k^{th} Fibonacci number, defined by $F_0 = 1$, $F_1 = 1$, $F_k = F_{k-2} + F_{k-1}$ for $k \geq 2$, and $\phi = (1+\sqrt{5})/2$ is the golden ratio.* ■

Proof. Let T_k be the minimum possible number of descendants of a node of rank k in a thin tree. From properties (1) and (3) of ranks, we obtain the following recurrence: $T_0 = 1$, $T_1 = 1$, $T_k \geq 1 + \sum_{i=0}^{k-2} T_i$ for $k \geq 2$. The Fibonacci numbers satisfy the same recurrence with equality. It follows by induction that $T_k \geq F_k$ for all k. The inequality $F_k \geq \phi^{k-1}$ is well known [11]. ■

To bound the amortized time of *decrease-key* and *delete*, let us charge one unit of time for each iteration of the violation-repair step. Each non-terminal iteration either unmarks a node, reducing the potential by two, or unmarks a node and adds a new tree, reducing the potential by one. Thus each non-terminating iteration takes at most zero amortized time. The remainder of *decrease-key*, including the last iteration of the violation repair step, takes $O(1)$ amortized time, giving a total amortized time for *decrease-key* of $O(1)$. A delete operation that is not a *delete-min* on an n-item heap can increase the number of trees, and hence the potential, by $O(\log n)$, by removing a node x. Not counting the violation-repair process, the time required for such a delete is also $O(\log n)$. The total amortized time for *delete*, including the violation repair process, is thus $O(\log n)$.

Thus we obtain the following theorem:

Theorem 2.2 *If we begin with no thin heaps and perform an arbitrary sequence of heap operations, then the total time is at most the total amortized time, with each* delete-min *or* delete *on an n-item heap charged $O(\log n)$ amortized time and each other operation charged $O(1)$ amortized time.*

Remark. It may be more efficient to delay correcting the rank of new roots until links involving them are performed in *delete-min*. ■

3 Worst-case time bounds

In this section we describe fat heaps. Fat heaps are similar both to binomial heaps and to the thin heaps of the previous section. The main difference is in two redundant counters that are embedded in the structure in order to achieve constant worst-case time bounds. One counter controls insertions of trees into the forest, and the other controls violations to the heap order. Another difference is that our fundamental linking step combines three trees instead of just two, which makes violation reduction less complicated than in run-relaxed heaps.

We begin in Section 3.1 by describing a simple pointer-based counting mechanism that allows incrementing of any digit in worst-case constant time. Section 3.2 describes the heap structure and Section 3.3 the heap operations.

3.1 Redundant Counters

Two redundant counters form the heart of our heap structure. These counters are based on the redundant binary representation of Knuth and Clancy [5], extended to support incrementing of arbitrary digits as described by Brodal [3] and Kaplan and Tarjan [10]. For completeness, and since we need both a binary and a ternary counter, we briefly describe here a simple pointer-based implementation of a redundant b-ary counter (for $b \geq 2$) that supports incrementing an arbitrary digit in $O(1)$ time. For additional discussion consult the previously cited references.

A *b-ary redundant representation* (b-ary RR) of a non-negative integer x is a sequence of digits $d_n, \ldots, d_0$, with $d_i \in \{0, 1, \ldots, b\}$, such that $x = \sum_{i=0}^{n} d_i b^i$. We call d *regular* if, between any two digits equal to b, there is a digit other than $b-1$. A *fix* operation on a digit $d_i = b$ in a regular b-ary RR d increments d_{i+1} by 1 and sets d_i to 0, producing a new regular b-ary RR d' representing the same number as d.

To add 1 to digit d_i of a regular b-ary RR d, we proceed as follows: If $d_i = b$, fix d_i. If $d_i = b-1$ and the least significant digit d_j with $j > i$ and $d_j \neq b-1$ satisfies $d_j = b$, fix d_j. Add one to d_i. If $d_i = b$, fix d_i. It is straightforward to verify that this computation preserves regularity and adds b^i to the number represented by d.

To implement this scheme, we use a singly-linked list of digits, from least-significant to most-significant. In addition, each digit d_i equal to $b-1$ has a *forward pointer* indicating the least-significant digit d_j with $j > i$ and $d_j \neq b-1$, if this digit is a b; otherwise, the forward pointer of d_i indicates an arbitrary digit d_j with $j > i$ that can be either b or different from b. While incrementing $d_i = b-1$ if the forward pointer of d_i indicates a digit $d_j = b$ we fix d_j. If the least significant digit $d_{j'}$ with $j' > i$ and $d_{j'} \neq b-1$ satisfies $d_{j'} = b$, then by the definition of the forward pointers $j = j'$. Otherwise we fix an arbitrary digit which is not necessary to fix, but fixing it does no harm to the regularity of the representation.

With this representation, adding one to a digit as described above takes constant time. We update forward pointers as follows: when a digit d_{i+1} becomes $b-1$ because of a fix of digit d_i, set the forward pointer of d_{i+1} to indicate d_{i+2} if $d_{i+2} = b$, or equal to the forward pointer of d_{i+2} if $d_{i+2} = b-1$. When actually adding 1 to d_i in the middle of an increment operation, it is not necessary to update the forward pointer of d_i, even if d_i becomes $b-1$, because the possible fix of d_j guarantees that the forward pointer of d_i need not have any particular value.

This scheme can be extended to support constant-time decrementing of an arbitrary digit by adding an additional digit value $b+1$. We leave the details as an exercise.

3.2 Fat Heaps

In analogy with binomial trees, we define a *fat tree* F_k of rank k recursively, as follows. A fat tree F_0 consists of a single node. A fat tree F_k for $k > 0$ consists of three F_{k-1} trees linked by making the roots of two of them leftmost children of the root of the third.

A *fat heap* consists of a forest of almost heap-ordered fat trees, at most three per rank, with at most two violations of the heap order per rank. (A violation of rank k is a node of rank k whose key is less than that of its parent). We also impose regularity constraints on the trees of various ranks and the violations of various ranks, as described below. We maintain a pointer to a node containing a minimum key in the forest, called the *minimum node*. We maintain the property that the minimum node is a tree root; whenever this is not the case, we can restore this property by swapping the item in the minimum node with the item in an arbitrary root; this may eliminate a violation. For each node we maintain pointers to its leftmost child, to its left and right siblings, and to its parent. We also store its rank.

We organize the roots of the fat trees into a redundant ternary counter called the root counter. If there are t trees of rank k, the digit d_k of the counter has an implicit value of t. Instead of storing the value of d_k, we store a list of the trees of rank k. Incrementing d_k corresponds to adding a new tree of rank k to the forest. We maintain the invariant that the digit sequence of the counter is regular as defined in Section 3.1.

We organize the violations into a similar redundant binary counter called the *violation counter*, the k^{th} position of which is a list of nodes of rank k at which violations occur. Again we maintain the invariant that the digit sequence of the counter is regular. Incrementing the i^{th} digit of the violation counter corresponds to creating a new rank i violation.

We maintain an array indexed by rank, whose k^{th} position stores the list of trees corresponding to the k^{th} digit of the root counter. We maintain a similar array for accessing the violations counter. We also maintain forward pointers as described in Section 3.1. Since the lengths of the counters are functions of the current size of the heap these arrays must be extendible. (It is well-known that extendible arrays can be obtained from ordinary arrays by array doubling and incremental copying).

Remark. We can avoid the use of arrays and implement fat heaps on a pointer machine. In such an implementation we store in any rank i node a pointer to the i^{th} list in the violation counter, and a pointer to the i^{th} list in the root counter. In order to have a pointer to the i^{th} node of the violations counter when creating a new rank i node, the i^{th} node in the root counter stores a pointer to the i^{th} node in the violation counter. In this implementation the digits of the counters must be singly-linked from least-significant to most-significant as described in Section 3.1.

In the pointer model, we can implement all the heap operations except for *meld* essentially as described in the next section. Our implementation of *meld* works only in the RAM model. ■

3.3 Operations

To perform *make-heap*, we return pointers to an empty root counter and an empty violation counter. To perform *find-min*(h), we return the item in the minimum node of h. To perform *insert*(i, h), we put the new item i in the single node of a new F_0 tree, and insert this tree into the forest by incrementing the least-significant digit of the root counter. To increment a digit of the root counter, we may need to fix one or more digits of the root counter whose value is 3. To fix $d_i = 3$ we link the three rank-i trees in the tree list to form one rank-$(i+1)$ tree, by making the two roots with higher keys leftmost children of the root with the smallest key, breaking a tie arbitrarily.

We perform *decrease-key*(Δ, i, h) as follows. Let x be the node containing i. Subtract Δ from the key of i. If the new key of i is smaller than the minimum key of h we swap i with the minimum key. Let r be the rank of x. If x is a violating node, we add x as a new rank-r violation by incrementing the r^{th} digit d_r of the violation counter.

The increment of d_r may require fixing one or more digits of the violation counter. Such a fix, say of $d_i = 2$, corresponds to converting two rank-i violations into a rank-$(i+1)$ violation, which we do as follows. We arrange the two violations to have the same parent by swapping the subtree rooted at the violating node whose parent has smaller key with the subtree rooted at the rank-i sibling of the violating node whose parent has the larger key. It is easy to verify that no new violations are created by this swap.

Let y be the common parent of the two violating nodes after the swap. If the rank of y is $i+1$, let F be an F_i tree obtained from y by deleting the two violating nodes from its list of children, and let F' and F'' be the F_i trees rooted at the two violating nodes. Link F, F', and F'' to form an F_{i+1} tree whose root z is a node with smallest key among the roots of F, F' and F''. If y was not a tree root, replace it by z in the list of children of the old parent of y; and, if z turns out to be violating, increment d_{i+1}. Otherwise, replace the tree rooted at y by the new tree rooted at z in the forest. If, on the other hand, the rank of y is bigger than $i+1$, then, by the regularity condition on the violation counter, y must have a child w of rank $i+1$ that is not violating, and the two rank-i children of w must also be non-violating. Replace the two violating rank-i children of y by the two good rank-i children of w. Then link the two F_i trees rooted at the two violating nodes and the one rooted at w after cutting off its two rank-i children to form a new F_{i+1} tree. The root z of the newly formed F_{i+1} tree replaces w as a child of y. If z turns out to be violating, increment d_{i+1}.

We perform *delete-min*(h) as follows. Delete the subtree rooted at the minimum node from the forest. (It follows from the regularity constraint on the counters that we can decrease a nonzero digit in constant time without updating any forward pointers.) Discard the minimum node, and insert the trees rooted at its children one-by-one into the forest by incrementing the appropriate digit of the root counter. The new minimum key is either at the root of a tree in the forest or in a violating node. Search the roots of the trees in the forest and the violating nodes for the new minimum m'. If m' is in a violating node, swap it with the element stored in a root of a tree in the forest. The node that previously stored m' may no longer be a violating node, and if this happens decrease the corresponding digit in the violation counter. After the swap, the new minimum is at a root of a tree in the forest. Let this root be the new minimum node. We perform $delete(i, h)$ by performing $decrease\text{-}key(\infty, i, h)$ and then $delete\text{-}min(i, h)$.

We perform $meld(h_1, h_2)$ as follows. The first step is to fix all violations in the heap with smaller maximum rank (breaking a tie arbitrarily). Assume w.l.o.g. that this heap is h_2. Traverse the violation counter of h_2 from the least-significant digit to the most-significant digit, skipping digits with value 0. For the i^{th} digit $d_i \neq 0$ do a fix operation on the digit indicated by the forward pointer of d_i if this digit has value 2. Then, if $d_i = 2$, fix d_i. If $d_i = 1$, convert this rank-i violation into a rank-$(i+1)$ violation as in fix, using the rank-i sibling of the violating node instead of the (nonexistent) other rank-i violation.

Once h_2 does not contain any violations, insert the roots from the root counter of h_2 into the root counter of h_1 by incrementing the appropriate digits. If the minimum node of h_2 contains smaller key than that of h_1, set the new minimum node of h_1 to be the minimum node of h_2. Return the modified heap h_1 as the result of the meld.

Based on the implementation described above, it is straightforward to obtain the following result, whose proof we leave as an exercise.

Theorem 3.1 *Fat heaps support* find-min, insert *and* decrease-key *in $O(1)$ worst-case time, and* delete, delete-min, *and* meld *in $O(\log n)$ worst-case time.*

There is an alternative way to define violations in fat heaps, which makes them more closely resemble thin heaps. We can define a rank k violation to be a missing rank k node somewhere in a fat tree F_j with $k < j$. We allow between zero and two such missing nodes of each rank. The i^{th} node of the violation counter stores pointers to the parent and immediate siblings of every missing rank-i node. An increment of the i^{th} digit of the violation counter corresponds to the addition of a new rank-i missing subtree. A fix of the i^{th} digit of the counter corresponds to converting two missing rank-i nodes into a missing rank-$(i+1)$ node.

We can perform such a fix as follows. We arrange the two missing subtrees to have the same parent as in the original version of fix. If the parent y is of rank $i+1$ then we delete y from the list of children of its parent and insert y into the forest as an F_i tree by incrementing the i^{th} digit of the root counter. Otherwise, by the regularity condition on the violation counter, y has a rank-$(i+1)$ sibling z, and z has two rank-i children. We remove the rank-i children of z from the list of children of z, and add them as two rank-i children of y. We delete z from the list of children of y, creating a new rank-$(i+1)$ violation, and add the tree rooted at z to the forest.

4 Remarks

There is an alternative representation of the redundant counters described in Section 3.1. Instead of using a record per digit, we can save space by using one record for each maximal consecutive block of identical digits. We connect these records in a doubly-linked list. A record r stores the value of the digits in the block it represents, the index of the first digit in the block, and the number of digits in the block. It is straightforward to verify that one can perform an increment of any digit in constant time using this alternative representation (assuming a pointer to the record representing the interval containing the desired digit is given). An advantage of this representation is the ability to extend the counter by an arbitrary number of identical digits in constant time.

As mentioned in Section 1, Brodal in [3] gives a heap structure that is theoretically better than our fat heaps, since its time bound for *meld* is constant in the worst-case. Brodal's structure, however, is much more complicated than fat heaps. It is an intriguing open question whether one can achieve the time bounds of Fibonacci heaps in the worst case without a severe increase in the complexity of the data structure.

Our fat heaps require arrays to implement *meld* efficiently, as do run-relaxed heaps and Brodal's heaps. It is an open problem to match Brodal's time bounds for heaps on a pointer machine.

References

[1] G. S. Brodal. Fast meldable priority queues. In *Proceedings of the 4th International Workshop on Algorithms and data structures (WADS'95), LNCS 955*, pages 282–290. Springer, 1995.

[2] G. S. Brodal and C. Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6(6):839–858, 1996.

[3] G.S. Brodal. Worst case priority queues. In *Proc. 7th annual ACM-SIAM Symposium on Discrete Algorithms (SODA 96)*, pages 52–58. ACM Press, 1996.

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

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

[6] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communication of the ACM*, 31(11):1343–1354, 1988.

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

[8] L. J. Guibas, E. M. McCreight, M. F. Plass, and J. R. Roberts. A new representation for linear lists. In *Proc. of the 9th Annual ACM Symposium on Theory of computing*, pages 49–60. ACM Press, 1977.

[9] H. Kaplan and R. E. Tarjan. Persistent lists with catenation via recursive slow-down. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 93–102. ACM Press, 1995.

[10] H. Kaplan and R. E. Tarjan. Purely functional representations of catenable sorted lists. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing*, pages 202–211. ACM Press, 1996.

[11] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1973.

[12] R. E. Tarjan. Amortized computational complexity. *SIAM J. on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[13] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21:309–314, 1978.