Thin Heaps, Thick Heaps

Haim Kaplan* Robert E. Tarjan †

April 1, 2005

Abstract

The Fibonacci heap was devised to provide an especially efficient implementation of Dijksta's shortest path algorithm. Although asymptotically efficient, it does not seem to be as fast in practice as other heap implementations. We explore three heap implementations (two versions of *thin heaps* and one of *thick heaps*) that have the same amortized efficiency as Fibonacci heaps but need less space and promise better practical performance. Thick heaps were originally proposed by Høyer; we fill in a gap in his analysis.

Keywords and Phrases: data structure, priority queue, heap, Fibonacci heap, thin heap, thick heap.

*School of Computer Science, Tel Aviv University, Israel. Research supported in part by Israeli Science Foundation (ISF), Grant No. 548/00

†Department of Computer Science, Princeton University, Princeton, NJ 08544 and Office of Strategy and Technology, Hewlett Packard, Palo Alto CA 94304. Research at Princeton University partially supported by the Aladdin Project, NSF Grant No. CCR-9626862.

1 Introduction

A *heap* (or *priority queue*) is an abstract data structure consisting of a set of *items*, each with a real-valued *key*, supporting the following operations:

create: Return a new, empty heap.

insert(x, h): Return the heap formed by inserting item x, with predefined key, into heap h.

find-min(h): Return an item in heap h of minimum key.

delete-min(h): Return the heap formed by deleting the item *find-min*(h) from h.

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

decrease-key(d, x, h): Decrease the key of item x in heap h by subtracting the non-negative real number d.

delete(x, h): Return the heap formed by deleting item x from heap h.

We assume that any given item is in only one heap at a time, that the operations are allowed to destructively modify their input heaps, and that in the case of *decrease-key*(d, x, h) and *delete*(x, h), the position of item x in heap h is known. We denote by n the number of items in the heap or heaps involved in an operation.

Many implementations of heaps represent them as sets of item-disjoint heap-ordered trees. Vuillemin [16] (see also [3]) described such an implementation, called *binomial queues*, that supports each heap operation in $O(\log n)$ worst-case time. Subsequently, Fredman and Tarjan [8] presented a relaxation of binomial queues called *Fibonacci heaps* that supports *create*, *insert*, *find-min*, *meld*, and *decrease-key* in $O(1)$ amortized time, and *delete-min* and *delete* in $O(\log n)$ amortized time. Fibonacci heaps allow an efficient implementation of Dijkstra's shortest path algorithm and have other applications. See [8, 9].

In a Fibonacci heap, each node requires four pointers: one to its parent, one to its first (leftmost) child, and one each to its left and right siblings. In addition, each node requires an integer rank and a mark bit. Although theoretically efficient, there is some evidence that Fibonacci heaps do not perform quite as well as other heap structures in practice [4].

In this paper, we explore alternative heap implementations that have the same amortized bounds as Fibonacci heaps but that save space (one pointer and the mark bit per node) without complicating the heap operations. We expect our structures to have better time and space performance than Fibonacci heaps in practice.

The remainder of the paper contains four sections. Section 2 presents our first structure, *thin heaps*. Section 3 discusses an alternative way to perform *decrease-key* on thin heaps and then describes a related data structure, *thick heaps*. Thick heaps are equivalent to a structure presented by Høyer [10], called *one-step heaps*; we fill in a gap in Høyer's amortized analysis. Section 4 presents variations and extensions of these structures. Section 5 offers comments on other related work.

2 Thin heaps

We define binomial trees inductively, as follows. A binomial tree of rank 0 is a single node. For any positive integer k, a binomial tree of rank k consists of two binomial trees of rank $k-1$ linked so that the root of one becomes the first child of the root of the other. (See Figure 1.) A binomial tree of rank k contains exactly 2^k nodes, and its root has exactly k children.

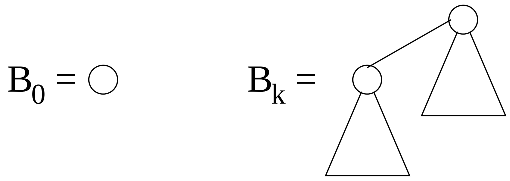

Figure 1: Binomial Trees.

Every tree in a Fibonacci heap is a binomial tree each of whose nodes may be missing one child (and the subtree rooted there). Similarly, every tree in a thin heap is a binomial tree, each of whose nodes may be missing its first child and the subtree rooted there. More formally, a *thin tree* is an ordered tree, each of whose nodes has a non-negative integer *rank*, with the following three properties.

(1) The children of a node with k children have ranks $k-1$, $k-2$, …, 0, from first to last.

(2) A node with k children has rank k or $k+1$. We call the node *normal* in the former case and *thin* in the latter case.

(3) The root is normal.

The *rank of a thin tree* is the rank of its root. A thin tree all of whose nodes are normal is a binomial tree, and vice-versa. Analogously to binomial trees, if we link two thin trees of equal rank by making the root of one tree the new first child of the root of the other and adding one to the rank of the single remaining root, then the result is a thin tree. Thin trees have the same size bound as the trees in Fibonacci heaps, given by the following lemma, whose proof is more direct than that for Fibonacci heaps.

Lemma 1. *A node with k children in a thin tree has at least $F_{k+1} \geq \phi^k$ 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. The inequality $F_{k+1} \geq \phi^k$ is well-known [12]. We prove by induction on k that a node with k children in a thin tree has at least F_{k+1} descendants. This is obvious for $k = 0, 1$. Suppose $k \geq 2$, let x be a node with k children in a thin tree, and consider the subtree rooted at x, which is itself a thin tree. Cutting the link between x and its first child and decreasing the rank of x by one results in two thin trees, one whose root x has $k-1$ children and the other whose root has $k-1$ or $k-2$ children, by (1) and (2). By the induction hypothesis, the original number of descendants of x is at least $F_k + \min\{F_k, F_{k-1}\} \geq F_{k+1}$. □

A *heap-ordered tree* is a rooted tree each of whose nodes has a real-valued *key*, satisfying *heap order*: no child has key smaller than that of its parent. A *thin heap* is a set of node-disjoint heap-ordered thin trees whose nodes are the heap items. Maintenance of the following data allows efficient performance of the heap operations on thin heaps:

(a) a singly-linked circular list of the tree roots, with a root of minimum key first;

(b) for each node, a doubly-linked list of its children;

(c) for each first child, its parent;

(d) for each node, its rank.

To maintain this data, an integer (the rank) and three pointers per node suffice: one to the first child; one to the right sibling, or next root if the node is a root; one to the left sibling, or parent if the node is a first child. This representation supports constant-time linking of two thin trees given their roots, as well as constant-time tests of whether a node is thin, whether a node is a root, and whether a node is a first child: a node is thin iff its rank is two more than that of its first child, a root iff its left sibling/parent pointer is null, and a first child iff it is the first child of the node indicated by its left sibling/parent pointer.

We perform *create*, *insert*, *find-min*, *meld*, *delete-min*, and *delete* on thin heaps exactly as on Fibonacci heaps. Specifically, to perform *create*, return a null pointer. To perform *insert*(x, h), make a new thin tree with the single node x, and insert x into the root list of h, in second or first position depending on whether its key is larger than that of the old first root or not. Return the modified heap. To perform *find-min*(h), return the first root of h. To perform *meld*(h_1, h_2), combine the root lists of h_1 and h_2 into a single list, whose first root is the first root of h_1 or the first root of h_2, whichever has smaller key, breaking a tie arbitrarily. Return the new heap.

To perform *delete-min*(h), remove from h the first root of h, say x, and make each child of x normal if it is not by changing its rank by one. Then 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: link two trees of equal rank by making the root of larger key the new first 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 of equal rank, form a list of the remaining roots, choosing a root of minimum key to be the first on the list. Return the modified heap. One can find links to perform in $O(1)$ time per link by using a temporary array indexed by rank to store tree roots. See [8]. Alternatively, one can avoid the use of random-access memory and implement *delete-min* (and all the other operations) 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 $k - 1$. During a *delete-min* each rank node points to a root of the corresponding rank, or to null.

Perform a *delete*(x, h) as *delete-min*(*decrease-key*(∞, x, h)).

The implementation of the last heap operation, *decrease-key*, differs from its implementation on Fibonacci heaps. A *decrease-key* on a Fibonacci heap can cause a sequence of *cascading cuts*, in which successive links along a path of ancestors are cut. A *decrease-key* on a thin tree can cause a sequence of cuts, but only of first children; between such cuts, rank reduction can occur, and the sequence can end with a restructuring step in which a child becomes a sibling of its parent.

Specifically, to perform *decrease-key*(d, x, h) on a thin heap, begin by subtracting d from the key of x. If x is a first child whose key remains no less than that of its parent, the *decrease-key* is done. Otherwise, if x is a root, complete the *decrease-key* by making x the first root of h if its key is now smaller than the previous minimum. If, on the other hand, x is not a root, let y be the left sibling of x, or the parent if it is a first child. Cut the link between x and its parent by removing x from the list of children containing it, making it a new tree root. Make x normal if it is not, and add x to the list of roots, in first or second position depending on whether its key is smallest or not.

Cutting at x may violate (1), (2), or (3) at y. Repeat the following *repair step*, which repairs the violation at y but may create a new violation at the left sibling or parent of y, until there is no new violation. (See Figure 2.) Then return the modified heap.

Repair Step

Case 1. Violation of (1): node y has rank two greater than that of its next sibling, or has rank 1 and no next sibling.

Case 1a. Node y is thin. Reduce the rank of y by one, repairing the violation and making y normal. Replace y by its left sibling, or by its parent if it is a first child, and check for a violation at the new y.

Case 1b. Node y is normal. Remove the first child of y, say w, and insert w after y in the list of children containing y. This makes node y thin but repairs the violation without creating a new violation.

Case 2. Violation of (2): node y has rank three greater than that of its first child, or has rank two and no children. Decrease the rank of y by two, repairing the violation and making y normal. Let z be the left sibling of y, or its parent if y is a first child. Remove y from the list of children containing it, and add y to the root list in the second position. Replace y by z and check for a violation at the new y.

Case 3. Violation of (3): node y is a thin root. Make y normal by decreasing its rank by one.

It is straightforward to verify that *decrease-key* produces a valid thin heap. Note that a node loses at most one child during a *decrease-key*, which means that a root, which is initially normal, can become thin and violate (3), but can never violate (2). This means that node y in Case 2 is always a nonroot.

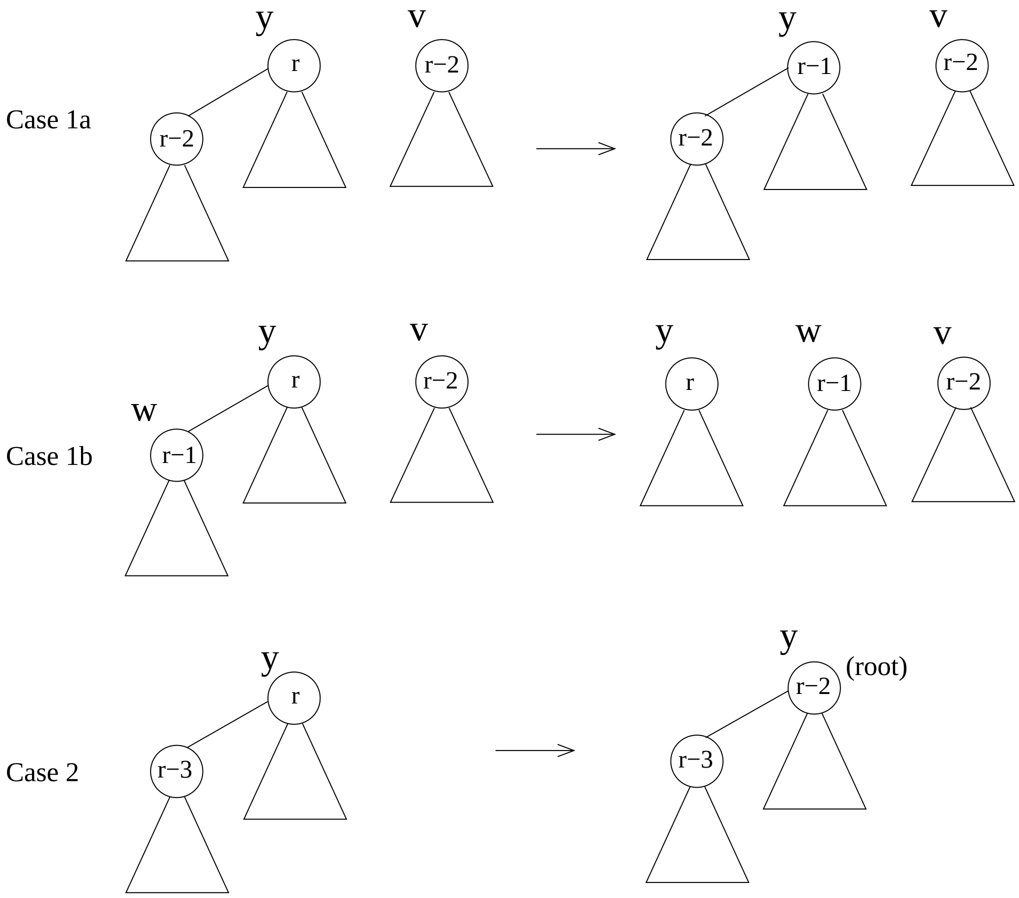

Figure 2: Nonroot repair step for thin heaps. In Cases 1a and 1b, node v is the right sibling (if it exists) of node y.

The amortized analysis of thin heaps is virtually the same as the analysis of Fibonacci heaps, with thin nodes in thin heaps taking the place of marked nodes in Fibonacci heaps [8]. We use the *potential function* technique [15]. Assign to each possible collection of node-disjoint thin heaps a non-negative integer *potential* equal to the number of trees plus twice the number of thin nodes. Define the *amortized time* of a heap operation to be its actual time plus the net increase in potential caused by the operation. Then the total actual time of an arbitrary sequence of heap operations, starting with no heaps, is at most the sum of the amortized times of the operations [15].

Both the actual and the amortized times of *find-min*, *insert*, and *meld* are $O(1)$: an insertion increases the number of trees and hence the potential by one; the other operations do not change the potential. If we charge one unit of time for each linking step, the amortized time of a *delete-min* operation is at most a constant times the maximum node rank, because each linking step reduces the potential by one and the remaining time spent in *delete-min* is bounded by a constant times the maximum rank. By Lemma 1, the maximum rank in an n-node thin heap is at most $\log_\phi n$, which means that the amortized time of *delete-min* is $O(\log n)$.

To analyze *decrease-key*, let us charge one unit of time for each repair step. Each repair step except the last makes a thin node normal and may create a new tree, for a potential drop of one. The initial cut and the last repair step take $O(1)$ time and increase the potential by at most three (at most one for the cut and two for the repair step), resulting in an overall amortized time of $O(1)$ for *decrease-key*. The amortized time of *delete* on an n-node heap is $O(\log n)$, since it is a *decrease-key* followed by a *delete-min*.

Thus we obtain the following theorem:

Theorem 2. *Beginning with no thin heaps and performing an arbitrary sequence of heap operations, if we charge each* delete-min *and* delete *on an n-node heap* $O(\log n)$ *amortized time and each other operation* $O(1)$ *amortized time, then the total time is at most the total amortized time.*

The worst-case time of a *decrease-key* operation on an n-node thin heap is $O(\log n)$, because each successive violation occurs at a node of higher rank. In contrast, a single *decrease-key* on an n-node Fibonacci heap can take $\Omega(n)$ time, as one can show by modifying the solution to exercise 20.4-1 on page 496 of [5].

3 From thin heaps to thick heaps

As on a Fibonacci heap, a *decrease-key* operation on a thin heap can in the worst case take more than one cut. By modifying Case 2 of *decrease-key*, however, we can avoid all but the first cut, the one at the node whose key decreases. Specifically, we replace Case 2 by the following variant (see Figure 3):

Case 2′. Violation of (2): node y has rank three greater than that of its first child, or has rank two and no children. Let w be the right sibling of node y.

Case 2′a. Node w is normal. Increase the rank of w by one, decrease the rank of y by one, and swap y and w in the list of children containing them. This makes both y and w thin and repairs the violation without creating a new violation.

Case 2′b. Node w is thin. Let z be the left sibling of y, or its parent if y has no left sibling. If y has key no less than that of w, remove y from the list of children containing it and make it the new first child of w; decrease the rank of y by two.

If, on the other hand, y has key less than that of w, remove w from the list of children and make it the new first child of y; decrease the ranks of both w and y by one. (In either case, both y and w become normal.) Finally, replace y by z and check for a violation at the new y.

As with the original implementation, it is straightforward to verify that this implementation of *decrease-key* produces a valid thin heap. The amortized analysis is similar, except that a smaller potential function suffices. Specifically, we define the potential of a collection of thin heaps to be the number of trees plus the number of thin nodes. The only non-terminating cases in *decrease-key*, Case 1a and Case 2′b, reduce the potential by one and two respectively. The rest of the analysis is the same as in Section 2. Thus we obtain Theorem 2 for thin heaps with the alternative *decrease-key* implementation.

We can also try to improve thin heaps by reducing the number of children of each node. We can in fact do this by replacing property (2) with the following alternative:

(2′) A node with k children has rank k or $k-1$. We call the node *normal* in the first case and *thick* in the latter case.

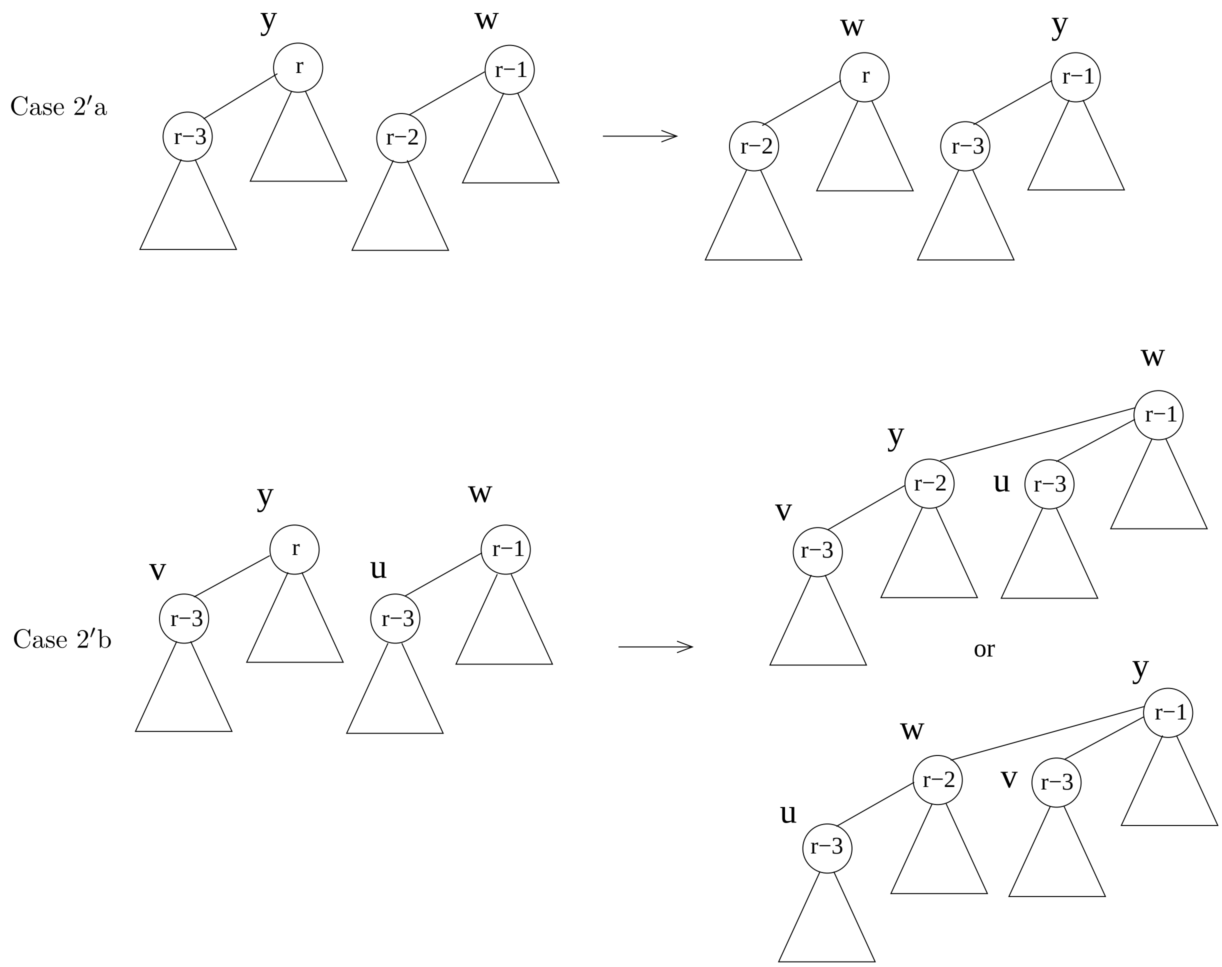

Figure 3: Alternative repair step for thin heaps. The outcome in Case 2′b depends on which of y and w has smaller key.

We call a tree with node ranks that satisfy (1), (2′), and (3) a *thick tree*. For thick trees the following improvement of Lemma 1 holds:

Lemma 3. *A node with k children in a thick tree has at least 2^k descendants including itself.*

Proof. We prove the lemma by induction on k. The lemma is clearly true for $k = 0$. Suppose $k \geq 1$, let x be a node with k children in a thick tree, and consider the subtree rooted at x, which is itself a thick tree. Cutting the link between x and its first child and decreasing the rank of x by one results in two thick trees, one whose root x has $k-1$ children and the other whose root, say y, has k or $k-1$ children. If y has k children we obtain a smaller thick tree whose root has $k-1$ children by cutting the link between y and its first child. Thus the original subtree rooted at x consists of at least two disjoint trees with roots having $k-1$ children, plus possibly additional nodes. By the induction hypothesis the original subtree rooted at x contains at least $2^{k-1} + 2^{k-1} = 2^k$ nodes. □

A *thick heap* is a set of node-disjoint heap-ordered thick trees whose nodes are the heap items. We can implement all the heap operations except *decrease-key* on thick heaps exactly as on thin heaps. The implementation of decrease-key is similar to the alternative implementation for thin heaps, with the analogs of Cases 1a and 1b, as well as those of Cases 2′a and 2′b, switched. Specifically, perform *decrease-key* on a thick heap exactly as on a thin heap, but use the following repair step (see Figure 4):

Repair Step for Thick Heaps

Case 1″. Violation of (1): node y has rank two greater than that of its next sibling, or has rank 1 and no next sibling.

- **Case 1″a.** Node y is normal. Reduce the rank of y by one, repairing the violation and making y thick. Replace y by its left sibling, or by its parent if it is a first child, and check for a violation at the new y.
- **Case 1″b.** Node y is thick. Remove the first child of y, say w, decrease the rank of y by one, and insert w before y in the list of children containing y. This leaves y thick and repairs the violation without creating a new violation.

Case 2″. Violation of (2′): node y has rank two greater than that of its first child, or has rank one and no children. Decrease the rank of y by one, making it normal. If y is a root, the repair is complete. Otherwise, let w be the right sibling of y and apply the appropriate one of the following two cases:

Case 2″a. Node w is thick. Increase the rank of w by one and swap y and w in the list of children containing them. This makes both y and w normal and repairs the violation without creating a new violation.

Case 2″b. Node w is normal. Let z be the left sibling of y, or its parent if it has no left sibling. If y has key no less than that of w, remove y from the list of children containing it and make it a new first child of w. (Node w becomes thick.) If, on the other hand, y has key less than that of w, remove w from the list of children containing it and make it a new first child of y. (Node y becomes thick.) Finally, replace y by z and check for a violation at the new y.

The correctness of *decrease-key* is easy to establish. Note that a violation of (3) can occur only if the violating root violates (2′) as well. Thus the analog of Case (3) is part of case 2″. More interesting is the amortized analysis of thick heaps. In thin heaps, thin nodes are bad in that they can cause expensive *decrease-key* operations. In thick heaps, normal (nonroot) nodes are bad in the same way; thick nodes are good. Our potential function for thick heaps is twice the number of trees plus the number of nonroot normal nodes. With this definition of the potential, the amortized time of a link (in a *delete-min*) is zero if we charge it an actual time of one; the link reduces the number of trees by one (decreases the potential by two) but increases the number of nonroot normal nodes by one (increases the potential by one). Each of the non-terminating cases in *decrease-key* (1″a and 2″b) converts a previously normal node to thick, decreasing the potential by one; hence the amortized time to apply the case is zero if we charge it an actual time of one. The remainder of the analysis is exactly as for thin heaps, and thus we obtain Theorem 2 for thick heaps.

As with thin heaps, the worst-case time of a *decrease-key* on an n-node thick heap is $O(\log n)$, because each successive repair step is on a node of higher rank.

Thick heaps are not new; they were invented by Høyer [10], who called them *one-step heaps*. Our presentation differs from Høyer's in four ways. First, Høyer represents one-step heaps (as well as alternative structures) as half-ordered binary trees, instead of as forests of heap-ordered rooted trees. The two representations are equivalent; the standard mapping [12] between rooted forests

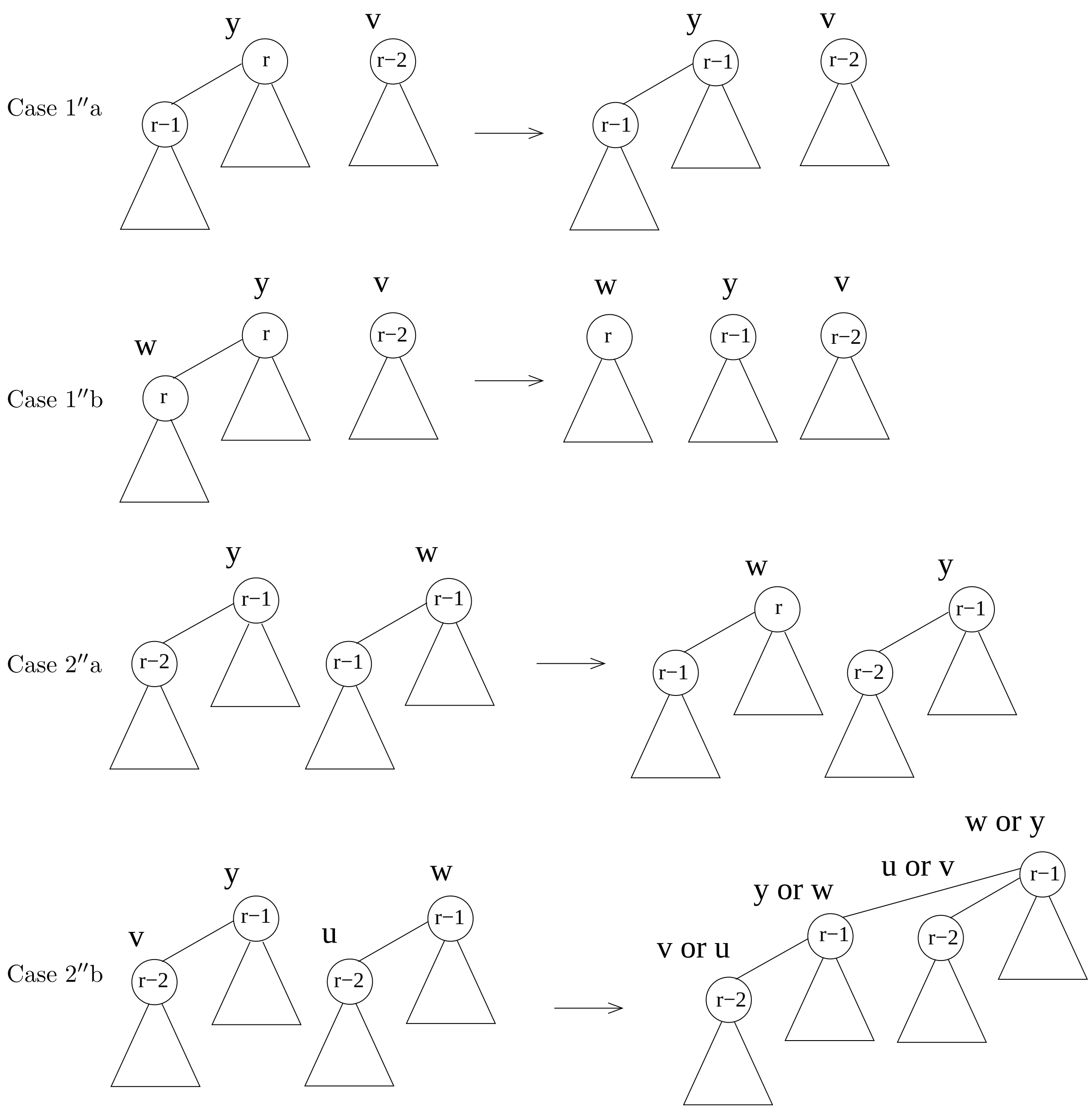

Figure 4: Repair step for thick heaps. In Cases 2″a and 2″b, node y is a nonroot and its rank has already been decreased by one. The outcome in case 2″b depends on which of y and w has smaller key.

and binary trees maps heap order to half order, as Høyer notes. The choice between the two is entirely a matter of conceptual comfort. Second, Høyer relaxes property (2′) to allow a node to have any rank no greater than its number of children. In fact, all the operations create only normal and thick nodes; Høyer's relaxation adds no generality. Third, Høyer adds to his data structure what he calls a *tree list*, which has the effect of reducing the worst-case time for a *delete-min* from $\Theta(n)$ to $O(\log n)$, while increasing the worst-case time for a *meld* from $O(1)$ to $\Theta(\log n)$. The tree list is a variant of an idea introduced by Driscoll et al. [6] and is related to the idea of a redudant counter, used later by Brodal [1, 2] and Kaplan et al. [11]. We discuss the use of a tree list briefly in Section 4. Finally, Høyer's amortized analysis of one-step heaps has a lacuna; he does not account for the creation of nonroot normal nodes by links. The analysis above fills this gap.

The implementation of *decrease-key* on thick heaps is analogous to the alternative implementation on thin heaps. It is natural to consider an implementation analogous to the original one on thin heaps. We obtain such an implementation by making the following substitution for case 2″:

Case 2‴. Violation of (2′): node y has rank two greater than that of its first child, or has rank two and no children. Decrease the rank of y by one, making y normal. If y is not a root, proceed exactly as in Case 2, removing y from its list of children, adding it to the root list, and checking for a new violation at the old left sibling of y, or at its old parent if it had no left sibling.

Perhaps surprisingly, thick heaps with this implementation of *decrease-key* are asymptotically less efficient. To obtain an expensive sequence of operations, do $2^k + 1$ insertions followed by one *delete-min*, resulting in a thick heap that is a B_k tree with all nodes normal. Then repeat the following sequence of operations any number of times: do k *decrease-key* operations on the deepest nodes of rank zero, followed by a single insert and a single *delete-min*. This sequence takes $\Theta(k^2)$ time and reproduces a B_k tree with all nodes normal. The amortized cost per *decrease-key* is $\Theta(k) = \Theta(\log n)$.

4 Variants and Extentions

All the variants and extensions discussed here apply equally to both versions of thin heaps and to thick heaps, and also to other heap structures presented by Høyer and similar structures as well.

We can reduce the number of pointers per node from three to two at the cost of a small constant factor in running time. Brown [3], pp. 306-308 describes how to do this for binomial queues; the same idea works for thin or thick heaps. For example, each node can point to its left sibling, or next root if it is a root; and to its parent if it is a first child, or to the first child of its left sibling if it is not a root and not a first child, or to the first child of its previous root if it is a root. (See Figure 5.)

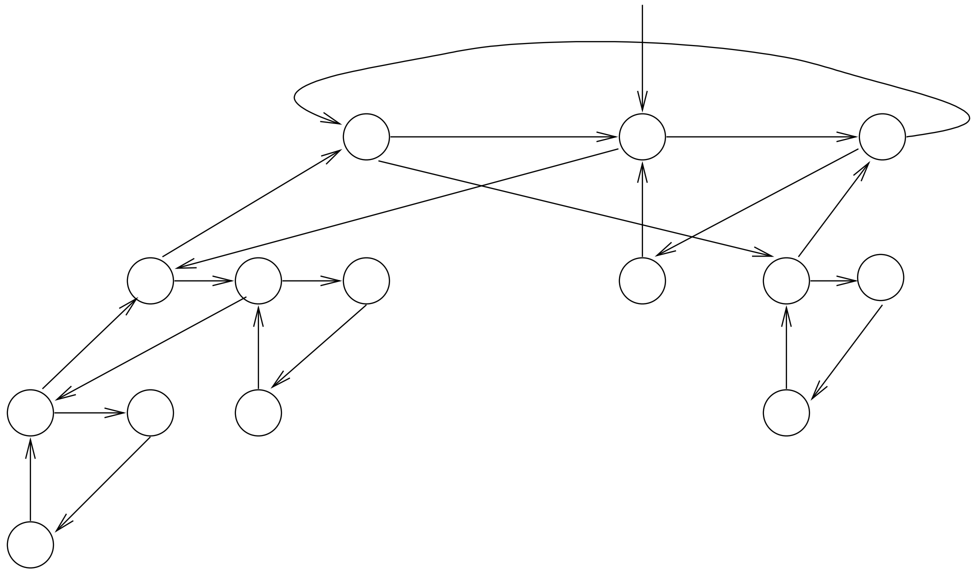

Figure 5: Two pointer-per-node representation of a heap. The heap consists of three binomial trees, a B_3, a B_1, and a B_2. This representation is essentially the reverse of Brown's "structure K".

Gabow et al. [9] gave an application of heaps to the problem of finding minimum spanning trees in directed graphs that requires an additional heap operation, that of *moving* a node x, along with all of its descendants, from one heap to another. This operation is only allowed if x does not have minimum key in its heap. Gabow et al. [9] describe how to perform this operation on a Fibonacci heap in $O(1)$ amortized time; their method also works on thin or thick heaps. A move is just like a *decrease-key* operation, except that the tree with root x is moved to the new heap.

On thin and thick heaps the worst-case time of *delete-min* is $O(n)$. We can reduce this to $O(\log n)$ without affecting the amortized time bounds of any of the operations, as follows. Keep track of the number of trees and the number of nodes in each heap. Whenever the number of trees in an n-node heap exceeds $2\log_\phi n$ for thin heaps or $2\log n$ for thick heaps, do a cleanup by repeatedly linking trees of equal rank until there is at most one tree per rank. Such a cleanup takes zero amortized time, because the $O(\log n)$ actual time is paid for by the drop in potential caused by the links. If cleanups are done, the amortized time of all the heap operations remains the same; the worst-case times are $O(\log n)$ for all operations except *find-min*, which remains $O(1)$.

A refinement of the cleanup idea preserves the $O(\log n)$ worst-case time for *delete-min*, *meld*, and *decrease-key*, while reducing the worst-case time for *insert* back to $O(1)$. This is Høyer's tree list method: maintain, for each heap, a list of pairs of trees of equal rank, plus at most one "odd" tree per rank. When adding a new tree by an *insert* or a *decrease-key*, link two paired trees, if there are any. This guarantees that the number of trees is $O(\log n)$. To obtain an $O(1)$ amortized time bound for *meld*, one must add $O(\log n)$ to the potential of a n-node heap; this pays for combining the tree lists during the *meld* of two heaps. A drawback of this method is that each heap requires an array of trees, one per rank; there is no obvious way to implement this idea on a pointer machine.

The *decrease-key* and *delete* operations have as a parameter the heap containing the affected item. If we omit this parameter and we allow *meld* as an operation, then the amortized time per *decrease-key* is no longer $O(1)$, because we need a disjoint set data structure to keep track of the partition of the items into heaps. Kaplan et al. [11] discuss upper and lower bounds for this modified version of the problem.

5 Other related work

We have discussed three alternatives to Fibonacci heaps: thin heaps with two forms of *decrease-key*, and thick heaps. We believe that all these structures will perform better than Fibonacci heaps in practice. A good experimental study would verify or disprove this, as well as determining, whether any of these three structures is significantly better than the others.

One may ask how other heap implementations compare. There are many meldable heap implementations in the literature. Some of these have an $O(1)$ time bound for *find-min* and $O(\log n)$

for the other heap operations, either worst-case [3, 11, 16] or amortized [7, 14]. We shall look in detail only at those that achieve an $O(1)$ bound for *decrease-key*. In addition to Fibonacci heaps, these include a structure of Peterson [13] based on half-ordered AVL-trees, several structures of Høyer [10] based on half-ordered trees, the rank-relaxed and run-relaxed heaps of Driscoll et al. [6], a structure of Brodal [2] improving a previous structure of his [1], and the fat heaps of Kaplan et al. [11].

Among the structures of Peterson and Høyer, the simplest is one-step heaps, which as we mentioned in Section 3 are equivalent to thick heaps. Rank-relaxed heaps, run-relaxed-heaps, and fat heaps, as well as being more complicated than thin and thick heaps, do not have an $O(1)$ time bound for *meld*. Brodal's structure achieves the same time bounds as Fibonacci heaps for all the heap operations in the worst case, but it is significantly more complicated than thin and thick heaps.

References

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

[2] 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.

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

[4] B. V. Cherkasky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. *Mathematical Programming*, 73:129–174, 1996.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, Cambridge, Massachusetts, second edition, 2001.

[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, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1:111–129, 1986.

[8] 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.

[9] H. N. Gabow, Z. Galil, T. Spencer, and R. E. Tarjan. Efficient algorithms for finding minimum spanning trees in directed and undirected graphs. *Combinatorica*, 6(2):109–122, 1986.

[10] P. Høyer. A general technique for implementation of efficient priority queues. In *Proc. 3rd Israel Symp. on Theory of Computer Systems*, pages 57–66. IEEE Computer Society Press, 1995.

[11] H. Kaplan, N. Shafrir, and R. E. Tarjan. Meldable heaps and boolean union-find. In *Proc. 34th Annual ACM Symposium on Theory of Computing (STOC)*, pages 573–582, 2002.

[12] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, third edition, 1997.

[13] G. L. Peterson. A balanced tree scheme for meldable heaps with updates. Technical Report GIT-ICS-87-23, School of Information and Computer Science, Georgia Institute of Technology, Altanta, Georgia, 1987.

[14] D. Sleator and R. E. Tarjan. Self-adjusting heaps. *SIAM J. Comput.*, 15(1):52–69, 1986.

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

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