Thin Heaps

Haim Kaplan* Robert E. Tarjan †

January 23, 2004

ABSTRACT

We describe a new heap data structure, the *thin heap*, which is closely related to the Fibonacci heap and has the same asymptotic performance. Specifically, *create*, *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.

Keywords: Data structures, priority queue, heap, Fibonacci heap, binomial tree, thin 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.

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

create: Return a new, empty heap.

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

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

delete-min(h): Delete an item from heap h of minimum key 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, destroying h_1 and h_2.

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

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

Vuillemin [7] 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 [3] invented a relaxation of binomial queues called *Fibonacci heaps*, which support *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 optimal implementation of Dijkstra's shortest path algorithm and are useful in many other algorithms. (See [3].)

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 requires four pointers: one to its parent, one to its 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 [2].

We present a new class of heaps that we call *thin heaps*. Thin heaps satisfy the same amortized time bound for each operation as Fibonacci heaps. The main advantage that thin heaps offer is the elimination of one pointer and the mark bit 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.

For any integer $k \geq 0$, the *binomial tree of rank k*, denoted by B_k, is defined 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, which is equal to the number of children of x.

A *thin tree* is a binomial tree in which each nonroot node may optionally have lost its leftmost child (and attached subtree). More formally (and more accurately), we define a *thin tree* to be an ordered tree, each of whose nodes has a non-negative integer *rank*, with the following properties:

(i) a rank-r node either has r children of ranks $r-1, r-2, \ldots, 0$ from leftmost to rightmost (we call such a node *thick*) or has $r-1$ children, of ranks $r-2, r-3, \ldots, 0$ from leftmost to rightmost (we call such a node *thin*);

(ii) the root is thick.

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

A *thin heap* is a set of node-disjoint heap-ordered thin trees. By *heap-ordered* we mean that each node has an associated real-valued *key*, and the keys obey heap order: every child has key no smaller than that of its parent. We represent a thin heap using the following information:

(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 leftmost child, its parent;

(d) for each node, its rank.

To store this information, an integer (the rank) and three pointers per node suffice: one to the leftmost 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.

A node is thick if and only if its rank is one greater than that of its leftmost child. A node is a root if and only if its left sibling/parent pointer is null. A node is a leftmost child if and only if it is the leftmost child of the node indicated by its left sibling/parent pointer.

We perform the various heap operations on thin heaps as follows. The implementation of *create*, *insert*, *find-min*, *meld*, and *delete-min* is exactly as for Fibonacci heaps. Specifically, to perform *create*, return a null pointer. To perform *insert*(x, h), create a new one-node thin tree from x and insert x into the root list of h, in first or second position depending on whether its key is smaller than the key of the old first root or not. 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 either the first root of h_1 or the first root of h_2, whichever has smaller key (breaking a tie arbitrarily).

To perform *delete-min*(h), remove from h the first root, say x, and make thick each child of x that is thin by decreasing 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: 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, form a list of the remaining roots, in the process finding a root of minimum key to be the first root of the new heap. Finally, return x. To find links to perform in $O(1)$ time per link, we can use a temporary array indexed by rank to store tree roots. See [3]. We 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 $k - 1$. 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 a path of ancestors are detached from their parents. Thin heaps use a related but different sequence of cuts, possibly with some intervening rank reductions.

We perform *decrease-key*(Δ, x, h) as follows. Subtract Δ from the key of x. If x is a root, make it the first root of h if its key is now smaller than that of the old first root of h, completing the *decrease-key* operation. If, on the other hand, x is not a root, let y be the left sibling of x, or the

parent of x if x has no left sibling. Remove x (and its attached subtree) from the list of children containing it, making it a new tree root. Make x thick if it is thin, and add x to the list of roots, as the first or second root depending on whether its key is less than that of the old first root.

Moving x and its attached subtree may create a violation of the rank rule (i). Specifically, if x was previously the right sibling of y, the new right sibling of y will have a rank two less than that of y (or y will have rank one and no right sibling). We call this a *sibling violation at* y. If x was previously the leftmost child of y and y was previously thin, y will now have a rank three greater than that of its new leftmost child (or have rank 2 and no children). We call this a *child violation at* y. 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:

Case 1 (sibling violation): Node y has rank two greater than that of its right sibling, or rank one and no right sibling.

Case 1a (node y is thick): Remove the leftmost child of y, say w, and its attached subtree, and insert w after y in the list of children containing y, making w the new right sibling of y. Since y was originally thick, w has rank one less than the rank of y, and this repairs the violation while making y thin. This case is terminating.

Case 1b (node y is thin): Decrease the rank of y by one, repairing the violation at y and making y thick. Replace y by its left sibling if it has one, or its parent if not. This case is possibly non-terminating.

Case 2 (child violation): Decrease the rank of y by two, repairing the violation and making y thick. If y is a non-root, proceed as follows. Let z be the left sibling of y, or its parent if it has no left sibling. Remove y (and its attached subtree) from the list of children containing it, and add y to the list of tree roots as the new second root. Replace y by z. This case is possibly non-terminating.

It is straightforward to verify that *decrease-key* produces a valid thin heap.

The last heap operation, *delete*(x, h), we perform by executing *decrease-key*(∞, x, h) followed by *delete-min*(h).

The analysis of thin heaps is essentially the same as the analysis of Fibonacci heaps. It uses the *potential function* technique [6]. 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 thin 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 [6].

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 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. The inequality $F_k \geq \phi^{k-1}$ is well-known [5]. We prove by induction on k that a node of rank k in a thin tree has at least F_k descendants. This is obvious for $k = 0$ and $k = 1$. suppose $k \geq 2$. Let x be a node of rank k in a thin tree, and consider the subtree rooted at k, which is itself a thin tree. Cutting the link between x and its leftmost child and decreasing the rank of x by one produces two thin trees, one with a root of rank $k - 1$, and one with a root of rank $k - 1$ or $k - 2$, depending on whether x was thick or thin, respectively. By the induction hypothesis, the original number of descendants of x is at least $F_{k-1} + \min\{F_{k-1}, F_{k-2}\} \geq F_k$. □

To bound the amortized time of *decrease-key*, let us charge one unit of time for each iteration of the violation-repair step. Each non-terminating iteration either makes a thin node thick, reducing the potential by two, or makes a thin node thick 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 bound for *decrease-key* of $O(1)$. The amortized time of *delete* 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, the total time is at most the total amortized time if we charge each* delete-min *and* delete $O(\log n)$ *amortized time and each other operation* $O(1)$ *amortized time.*

Remark. We note that the worst-case time of a *decrease-key* operation on a thin heap is $O(\log n)$, in contrast to Fibonacci heaps, in which a *decrease-key* operation can take $\Omega(n)$ time. One can reduce the number of pointers per node in a thin heap from three to two at a cost of a small constant factor in running time. See e.g. [1, pp. 306-308] Additional heap operations can be accomodated in variants of Fibonacci heaps [3, 4]; thin heaps have corresponding variants.

References

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

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

[3] 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.

[4] 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.

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

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

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