Quake Heaps: A Simple Alternative to Fibonacci Heaps

Timothy M. Chan*

February 27, 2009

Abstract

This note describes a data structure that has the same theoretical performance as Fibonacci heaps, supporting decrease-key operations in $O(1)$ amortized time and delete-min operations in $O(\log n)$ amortized time. The data structure is simple to explain and analyze, and may be of pedagogical value.

1 Introduction

In their seminal paper [5], Fredman and Tarjan investigated the problem of maintaining a set S of n elements under the operations

- insert(x): insert an element x to S;
- delete-min(): remove the minimum element x from S, returning x;
- decrease-key(x, k): change the value of an element x to a smaller value k.

They presented the first method, called *Fibonacci heaps*, that can support insert() and decrease-key() in $O(1)$ amortized time, and delete-min() in $O(\log n)$ amortized time.

Since Fredman and Tarjan's paper, a number of alternatives have been proposed in the literature, including Driscoll *et al.*'s "relaxed heaps" and "run-relaxed heaps" [1], Peterson's "Vheaps" [8] (and more generally, Høyer's family of "ranked priority queues" [6]), Takaoka's "2-3 heaps" [10], Kaplan and Tarjan's "thin heaps" and "fat heaps" [7], and most recently, Elmasry's "violation heaps" [2]. *Pairing heaps* [4, 9, 3] are another popular variant that performs well in practice, although they do not guarantee $O(1)$ decrease-key cost.

Among all the methods that guarantee constant decrease-key and logarithmic delete-min cost, Fibonacci heaps are still among the easiest to implement. The decrease-key() operation is done via a simple "cascading cut" strategy. A direct implementation requires 4 pointers per node, plus an integer field and an extra bit for marking. In contrast, Vheaps and relatives, as well as thin heaps, can be implemented directly with 3 pointers (or 2 pointers with more effort) plus an integer field per node, although these methods require some tricky case analysis for "rebalancing" during decrease-key().

*School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, `tmchan@uwaterloo.ca`.

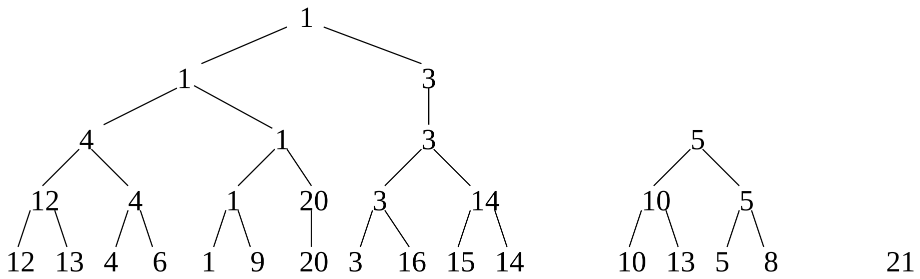

Figure 1: An example.

In this note, we describe a data structure that is arguably the easiest to understand among all the existing methods. There is no case analysis involved, and no "cascading" during decrease-key(). We use a very simple, and rather standard, idea to ensure balance: be lazy during updates, and just rebuild when the structure gets "bad". The method can be implemented with 3 (or 2) pointers plus an integer field per node. Previous methods differ based on what local structural invariants are imposed. Our method is perhaps the most "relaxed", completely forgoing local constraints, only requiring the tracking of some global counters. (The recent violation heaps [2] are of a similar vein but require multiple local counters; our method is simpler.)

In Section 2, we give a self-contained presentation of the method,[1] which should be helpful for classroom use; it only assumes basic knowledge of amortized analysis. We find a description based on tournament trees the most intuitive, although the method can also be expressed more traditionally in terms of heap-ordered trees or half-ordered trees, as noted in Section 3.

2 Quake Heaps

The approach. We will work with a collection of *tournament trees*, where each element in S is stored in exactly one leaf, and the element of each internal node is defined as the minimum of the elements at the children. We require that at each node x, all paths from x to a leaf have the same length (namely, the *height* of x). We also require that each internal node has degree 2 or 1. See Figure 1.

Two basic operations are easy to do in constant time under these requirements: First, given two trees of the same height, we can *link* them into one, simply by creating a new root pointing to the two roots, holding the smaller element among the two roots. Secondly, given a node x whose element is different from x's parent's, we can *cut* out the subtree rooted at x. Note that x's former parent's degree is reduced to 1, but we have explicitly allowed for this situation.

Inserting an element can be trivially done by creating a new tree of size 1. The number of trees in the collection grows, but can be reduced by repeated linking at a convenient time later.

For a decrease-key operation on an element, let x be the highest node holding the element. It would too costly to update the elements at all the ancestors of x. Instead we can perform a cut at x. Then we can decrease the value of x at will in the separate new tree.

We need to address one key issue: after many decrease-key operations, the trees may become too off-balanced. Let n_i denote the number of nodes at height i. (In particular, $n_0 = n = |S|$.) Our

[1] Tradition demands a name to be given. The one in the title will hopefully make some sense after reading Section 2.

approach is simple—we maintain the following invariant for some constant $\alpha \in (1/2, 1)$:

$$n_{i+1} \leq \alpha n_i.$$

(To be concrete, set $\alpha = 3/4$, say.) The invariant clearly implies that the maximum height is at most $\log_{1/\alpha} n$. When the invariant is violated for some i, a "seismic" event occurs and we remove everything from height $i+1$ and up, to allow rebuilding later. Since $n_{i+1} = n_{i+2} = \cdots = 0$ now, the invariant is restored. Intuitively, events of large "magnitude" (i.e., at low heights) should occur infrequently.

Pseudocode. We give pseudocode for all three operations below:

insert(x):
1. create a new tree containing $\{x\}$

decrease-key(x, k):
1. cut the subtree rooted at the highest node holding x [this causes 1 new tree]
2. change x's value to k

delete-min():
1. $x \leftarrow$ minimum of all the roots
2. remove the path of nodes holding x [this results in multiple new trees]
3. while there are 2 trees of the same height:
4.   link the 2 trees [this reduces the number of trees by 1]
5. if $n_{i+1} > \alpha n_i$ for some i then:
6.   let i be the smallest such index
7.   remove all nodes at heights $\geq i$ [this increases the number of trees]
8. return x

We can explicitly store the highest node holding x for each element x; it is easy to update the node as linkings are performed. It is also easy to update the n_i's as nodes are created and removed. Lines 3–4 in delete-min() can be done in time proportional to the current number of trees, by using an auxiliary array of pointers to trees indexed by their heights.

Analysis. In the current data structure, let N be the number of nodes, T be the number of trees, and B be the number of degree-1 nodes. Define the potential to be $N + T + \frac{1}{2\alpha-1}B$. The amortized cost of an operation is the actual cost plus the change in potential.

For insert(), the actual cost is $O(1)$, and N and T increase by 1. So, the amortized cost is $O(1)$.

For decrease-key(), the actual cost is $O(1)$, and T and B increase by 1. So, the amortized cost is $O(1)$.

For delete-min(), we analyze lines 1–4 first. Let $T^{(0)}$ be the value of T just before the operation. Recall that the maximum height, and thus the path length in line 2, is $O(\log n)$. We can bound the actual cost by $T^{(0)} + O(\log n)$. Since after lines 3–4 there can remain at most one tree per height, T is decreased to $O(\log n)$. So, the change in T is $O(\log n) - T^{(0)}$. Since linking does not create degree-1 nodes, the change in B is nonpositive. Thus, the amortized cost is $O(\log n)$.

For lines 5–7 of delete-min(), let $n_j^{(0)}$ be the value of n_j just before these lines. We can bound the additional actual cost by $\sum_{j>i} n_j^{(0)}$. The change in N is at most $-\sum_{j>i} n_j^{(0)}$. The change in T

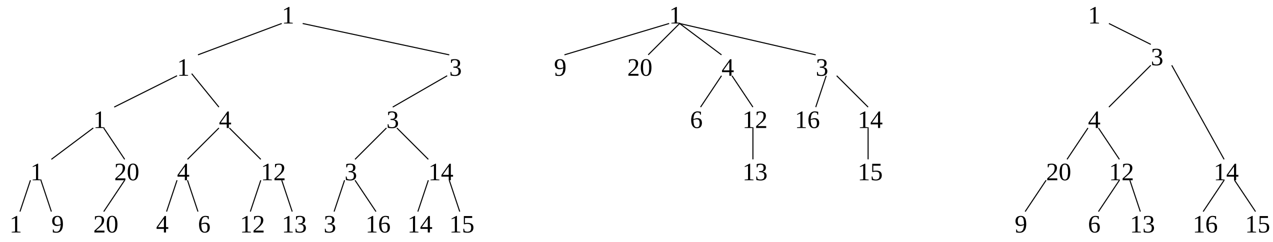

Figure 2: Transforming a tournament tree into a heap-ordered tree or a half-ordered tree.

is at most $n_i^{(0)}$. Let $b_i^{(0)}$ be the number of degree-1 nodes at height i just before lines 5–7. Observe that $n_i^{(0)} \geq 2n_{i+1}^{(0)} - b_i^{(0)}$. Thus, $b_i^{(0)} \geq 2n_{i+1}^{(0)} - n_i^{(0)} \geq (2\alpha - 1)n_i^{(0)}$. Hence, the change in B is at most $-(2\alpha - 1)n_i^{(0)}$. We conclude that the additional amortized cost is nonpositive, and the overall amortized cost for delete-min() is $O(\log n)$.

3 Comments

Implementation variants. Many variations of the method are possible. Linking of equal-height trees can be done at other places, e.g., immediately after an insertion or after lines 5–7 of delete-min(), without affecting the amortized cost. Alternatively, we can perform less linking in lines 3–4 of delete-min(), as long as the number of trees is reduced by a fraction if it exceeds $\Theta(\log n)$.

We can further relax the invariant to $n_{i+c} \leq \alpha n_i$ for any integer constant c. In the analysis, the potential can then be readjusted to $N + T + \frac{1}{c(2\alpha-1)}B$. It is straightforward to check that the amortized number of comparisons per decrease-key() is at most $1 + \frac{1}{c(2\alpha-1)}$, which can be made arbitrarily close to 1 at the expense of increasing the constant factor in delete-min(). (A similar tradeoff of constant factors is possible with Fibonacci heaps as well, by relaxing the "one child removed per node" constraint to c children [5].)

Like Fibonacci heaps, our method can support the meld operation in $O(1)$ amortized time, clearly.

In the tournament trees, it is convenient to assume that the smaller child of each node is always the left child (and if the node has degree 1, its only child is the left child).

An explicitly maintained tournament trees require a linear number of extra nodes, but more space-efficient representations are possible where each element is stored in only one node. One option is to transform a tournament tree T into a heap-ordered, $O(\log n)$-degree tree T': the children of x in T' are the right children of all the nodes holding x in T. See Figure 2(middle). Binomial heaps and Fibonacci heaps are usually described for trees of this form.

Another option is to transform T into a binary tree T'' as follows: after shortcutting degree-1 nodes in T, the right child of x in T' is the right child of the highest node holding x in T; the left child of x in T' is the right child of the sibling of the highest node holding x in T. See Figure 2(right). The resulting tree T'' is a *half-ordered* binary tree: the value of every node x is smaller than the value of any node in the right subtree of x. Høyer [6] advocated the use of such trees in implementation.

It is straightforward to redescribe our method in terms of such half-ordered binary trees. E.g., see [6] on the analogous linking and cutting operations. (The "height" of a node should now be renamed as "rank", and n_i corresponds to half the number of edges (x, y) that "cross" rank i, i.e., with x's rank $\geq i$ and y's rank $< i$.) Of course, we only need 3 pointers per node for binary trees. With more effort, 2 pointers are sufficient by a known trick: namely, for a left child, store its left child and sibling; for a right child, store its left child and parent (nodes with one child should be handled with care).

Comparison with other methods. Code for Fibonacci heaps is of roughly same length (perhaps slightly shorter) than that for our method. Fibonacci heaps require an extra loop for cascading cuts in decrease-key(), but bypass the cleaning step (lines 5–7). A disadvantage is that cascading cuts require an extra pointer per node: when recast in the framework of half-ordered binary trees (T'' above), we need a pointer from each element x to the lowest ancestor of x that is a right child.

Code for Vheaps or some version of ranked priority queues [8, 6] probably has about the same (perhaps slightly longer) length, if care is taken to cut down the number of cases. A loop is required during decrease-key() to restore balance e.g. by performing some rotations in the half-ordered binary tree.

Philosophically, Peterson's and Høyer's work demonstrated that obtaining a heap data structure with constant-time decrease-key() is similar to obtaining a balanced-search-tree data structure with constant-time deletion. Our main observation is that the heap problem is simpler: because of the lack of insertions (or more accurately, because insertions can be handled by having multiple trees for the heap problem), a simpler lazy strategy suffices.

We will leave the issue of which method actually gives the fastest implementation in practice. As Elmasry [2] hinted at, methods with more relaxed structural constraints might have better chance of being competitive with pairing heaps. If so, quake heaps, or at least some further variation of the idea, seem promising.

References

[1] J. Driscoll, H. Gabow, R. Shrairman, and R. Tarjan. Relaxed heaps: an alternative to Fibonacci heaps with applications to parallel computation. *Commun. ACM*, 31:1343–1354, 1988.

[2] A. Elmasry. Violation heaps: a better substitute for Fibonacci heaps. http://arXiv.org/abs/0812.2851, December, 2008.

[3] A. Elmasry. Pairing heaps with $O(\log\log n)$ decrease cost. In *Proc. 20th ACM-SIAM Sympos. Discrete Algorithms*, pages 471–476, 2009.

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

[5] M. Fredman and R. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34:596–615, 1987.

[6] P. Høyer. A general technique for implementation of efficient priority queues. In *Proc. 3rd Israel Sympos. Theory of Comput. Sys.*, pages 57–66, 1995.

[7] H. Kaplan and R. Tarjan. Thin heaps, thick heaps. *ACM Trans. Algorithms*, 4(1):article 3, 2008.

[8] G. Peterson. A balanced tree scheme for meldable heaps with updates. Tech. Report GIT-ICS-87-23, Georgia Institute of Technology, 1987.

[9] S. Pettie. Towards a final analysis of pairing heaps. In *Proc. 46th IEEE Sympos. Found. Comput. Sci.*, pages 174–183, 2005.

[10] T. Takaoka. Theory of 2-3 heaps. *Discrete Applied Math.*, 126:115–128, 2003.