**Lazy Fibonacci Heaps**

Robert E. Tarjan

April 9, 2013

1 Heaps as Heap-Ordered Trees

We denote by lg the base-two logarithm. We denote by *n* the number of items in the heap or heaps being accessed or updated. In stating time bounds we assume *n* ≥ 2 for simplicity. We assume all keys are distinct; if not, we break ties by item identifier. We represent a heap by a rooted tree whose nodes contain its items, arranged in *heap order*: if *x* is a node with parent *y* (y = *x.parent*), the key of the item in *x* is greater than that of the item in *y*. Heap order is not unique, but it does imply that the root contains the item of minimum key. The *key* of a node *x*, denoted by *x.key*, is the key of the item in *x*. The *size* of a tree is the number of nodes it contains. The *degree* of a node is its number of children.

The basic transformation on heap-ordered trees is *linking*. Given two nodes *x* and *y* in the same or different trees, to link *x* and *y*, swap *x* and *y* if necessary so that *x.key* < *y.key*, remove *y* from its set of siblings if it is a non-root, and make *y* a child of *x*, with *y* retaining its subtree. Node *x* is the *winner* of the link; node *y* is the *loser*.

A generic implementation of the various heap operations on heap-ordered trees is the following. To find the minimum in a heap, return the item in the root. To make an empty heap, return an empty tree. To insert an item into a heap, create a new one-node tree whose root contains the item, and meld this tree with the previously existing tree. To meld two heaps, link the roots of their trees. To delete an arbitrary item, decrease its key to minus infinity and do a minimum deletion. To delete the minimum in a heap, let *r* be the root. Delete *r*, producing a set of trees, one rooted at each old child of *r*. Repeatedly link two roots in any order until only one root is left. To decrease the key of the item in node *x*, in the tree with root *r*, update the key of the item in *x*. If *x* ≠ *r* and *x.key* < *x.parent.key*, remove *x* from its set of siblings, with *x* retaining its subtree, and link *x* and *r*.

With an appropriate tree representation, each **find-min**, **make-heap**, **insert**, **meld**, or **decrease-key** takes O(1) time worst-case, but each deletion takes time proportional to one plus the degree of the root.

2 Lazy Fibonacci Heaps

To speed up minimum deletion, we choose the links carefully. To do this, we give each node one of three states: *free*, *fixed*, or *marked*. Every root is free; a child can be in any state. The *rank* of a node is its number of non-free children. When linking two roots, if the winner and loser have the same rank, we make the loser fixed and increase the rank of its new parent; if the winner and loser have different ranks, the loser remains free and the rank of its new parent does not change.

To do a **delete-min**, destroy the root, make all new non-free roots free, link roots of equal rank until no two roots have equal rank, and then link the remaining roots in any order. To do the linking, put the roots into buckets indexed by rank. When a bucket receives a second root, link the two roots, and add the winner to the bucket of next-higher rank. Once all roots are in buckets, scan the buckets, removing roots and linking them.

To decrease the key of the item in node *x*, begin by updating *x.key*. If *x* is not a root and *x.key* < *x.parent.key*, make *x* free and apply the following rule until it no longer applies: if node *y* has a new free child, make *y* free if it is marked, or marked if it is fixed. Then detach *x* from its set of siblings, with *x* retaining its subtree, and link *x* and the root.

During a key decrease, only *x* and its ancestors can change state. The following pseudocode does the state and rank changes triggered by making *x* free; *y.state* and y.rank are the state and rank of node *y*, respectively.

if *x.state* ≠ *free* then

begin

*x.state* := free

*y* := *x.parent*

*y.rank* := *y.rank* – 1

while *y.state* = marked do

begin

*y.state* := free

*y* := *y.parent*

y.rank := y.rank – 1

end

if *y.state* = fixed then *y.state* := marked

end

This data structure, the *lazy Fibonacci heap*, is like a Fibonacci heap but with two changes: a heap consists of a single tree, not a set of trees, and marked nodes that lose a non-free child are not cut from their parents but merely change state. The implementation is a special case of the generic implementation given in Section 1.

To bound the running times of the heap operations, we begin by bounding the ranks. Then we do a potential-based amortized analysis.

**Lemma 1**: In a tree of size *n*, the ranks are at most lg*n*/logφ, where φ = (1 + sqrt(5))/2 is the golden ratio.

Proof: Let *x* be a node of rank *k* in a tree of size *n*. Order the *k* non-free children of *x* in the order they and *x* were linked, earliest to latest. If *y* is the *i*th such child, *y* has rank at least *i* – 2: when *x* and *y* were linked, *x* had rank at least *i* – 1; since *y* became fixed, y also had rank at least *i*– 1; after the link, *y* can lose only one non-free child, or it would become free itself. Let *nk* be the minimum number of descendants of a node of rank *k*, including itself. Then *nk* satisfies the recurrence *n*0 = 1, **n**1 = 2, *nk* ≥ 2 + *n*0 + *n*1 + … + *n*k – 2. This implies *n*k >= *nk* – 1 + *nk* – 2 for *k* ≥ 3, which also holds for k = 2. It follows that *n*k ≥ F*k* + 2, where F*k* is the *k*th Fibonacci number. Since F*k* + 2 ≥ φk [ ], *nk* ≥ φ*k*. The lemma follows. QED

We assign to each heap a potential equal to the number of free nodes plus twice the number of marked nodes. We define the amortized time of an operation to be its actual time (appropriately scaled) plus the net increase in potential it causes. If we begin with no heaps and do an arbitrary sequence of heap operations, the sum of their amortized times is an upper bound on the total actual time, since the initial potential is zero and the final potential is non-negative.

In an insertion, creation of a new root increases the potential by one. The link in a **meld** does not increase the potential, but it may decrease it by one. In a **decrease-key**, each change of state from marked to free decreases the potential by one, paying for the corresponding iteration of the while loop. There is also at most one change from fixed to free and at most one from fixed to marked, increasing the potential by at most three. It follows that each **insert**, **meld**, and **decrease-key** takes O(1) amortized time.

Consider a **delete-min** on a tree of size *n*. By Lemma 1, at most 1 + lg*n*/logφ new roots become free, increasing the potential by at most 2 + 2lg*n*/logφ. Each link of nodes of equal rank decreases the potential by one, paying for the link. Also by Lemma 1, the number of links of nodes of different ranks is at most lg*n*/logφ, and the time to scan and empty buckets is O(lg*n*). It follows that the amortized time of the delete-min is O(lg*n*). Thus we obtain the following theorem:

Theorem 1: On lazy Fibonacci heaps, each **delete-min** or **delete** takes O(lg*n*) amortized time, and each other operation takes O(1) amortized time.