Fibonacci Heaps

Jonathan Turner

January 9, 2013

We have seen that the d-heap provides a simple and efficient method for selecting an item of minimum key from among a set of items. It also allows us to adjust the key of any item, and for applications where the keys usually decrease, we can optimize the overall performance by adjusting the value of d.

The *Fibonacci heap* (or *fheap*, for short) provides much the same functionality as the d-heap but has two key advantages. First, Fibonacci heaps allow us to decrease the key of an item in "effectively" constant time, which allows us to improve the worst-case performance of Prim's algorithm for minimum spanning trees and Dijkstra's algorithm for shortest paths, on sparse graphs. The second advantage of Fibonacci heaps is that they are *meldable*, meaning that we can efficiently combine two fheaps into a single fheap.

Fibonacci heaps are also interesting in another way. With most data structures, we characterize their performance by deriving bounds on the time required for each individual operation. In the case of Fibonacci heaps, we can get a better performance bound by considering the overall time for a sequence of operations. That is, for fheaps, while certain individual operations may be expensive, the average cost per operation over any sequence of operations is guaranteed to be acceptably small.

The Fibonacci heaps data structure defines a collection of heaps defined over a set of items $a_1, \ldots, a_n$, each having a *key*. Each item belongs to exactly one heap at all times and each heap is identified by one of its members (called its *id*). The data structure supports the following operations.

- *findmin*(h): return an item of minimum key in heap h (or more precisely, the heap with id h).

- *insert*(i, x, h): insert item i into heap h with key x; i must be a singleton heap.

- *delete*(i, h): delete item i from h and return the resulting heap's id (note that the heap's id may change even if i is not the id of the heap before the operation).

- *deletemin*(h): delete a minimum key item from h; return it and the id of the resulting heap.

- *meld*(h_1, h_2): return the id of the heap formed by combining h_1 and h_2.

- *decreasekey*(Δ, i, h): decrease the key of item i in h by Δ; return the new id.

We could easily define a general *changekey* operation; we have chosen to limit ourselves to *decreasekey* in order to simplify the ensuing discussion.

An fheap can be implemented as a collection of heap-ordered trees, in which all the tree roots are joined together in a doubly-linked list. In addition to its key, each item has an auxiliary field, called its *rank*, which is equal to the number of children that it has, and additional bit called the *mark bit*, which is used to control certain decisions in the *decreasekey* operation. The tree structure is defined using a combination of parent and child pointers and *sibling lists*. Specifically, each non-root node has a pointer to its parent and each non-leaf node has a pointer to one of its children. Nodes with a common parent are joined together by a doubly-linked sibling list. See Figure 1 for an example of such a heap implementation. In the example,

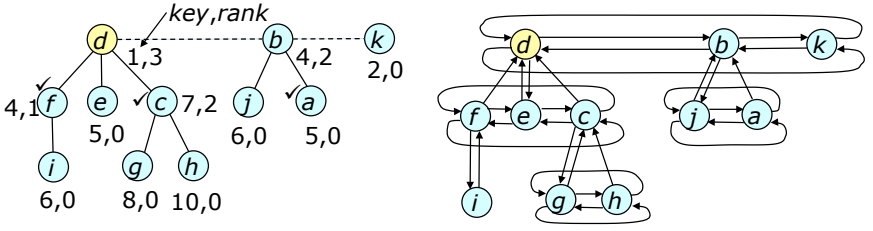

Figure 1: Structure of a single fheap

the mark bits are indicated by check marks. In general, the id of a heap is a node of minimum key, so in the example d is the id. Please take note that the example shows a *single heap* from a larger collection of heaps.

Next, we describe how the operations are implemented. The *meld* operation is particularly simple. We simply merge the root lists of the two heaps and return the item of minimum key as the id of the resulting heap. This

can be done in constant time. The *insert* operation can be implemented as a special case of *meld*, so it also takes constant time.

The *delete* operation can be implemented using the *decreasekey* and *deletemin* operations. Specifically, we first decrease the key of item i by an amount that makes i the minimum key item in the heap, then we do a *deletemin* on the heap. This leaves i in a singleton heap and we finish by restoring the original key value for i.

The *deletemin* and *decreasekey* operations both have more complex implementations and their running times are somewhat variable. We will show that any sequence of of m operations on a set of fheaps (starting from the initial state) that includes k *decreasekey* operations takes $O(k + m \log n)$ time. However, we will also see that individual *deletemin* and *decreasekey* operations cannot be bounded so tightly.

To implement *deletemin* we start by removing the item of minimum key from the root list and merge its list of children into the root list. (So for example, if we performed a *deletemin* on the heap in Figure 1, nodes f, e and c would all become tree roots.) At this point, we need to scan the root list in order to find the item with the smallest key, since we must return this item as the new id of the heap. Since we have to scan the list anyway, we take the opportunity to restructure the heap so as to make later operations more efficient.

Specifically, as we're scannng the root list, if we encounter two nodes of the same rank, we make one of them a child of the other (depending on their key values). This can be done efficiently using an auxiliary array. Whenever we examine a tree root, we insert the tree root into the array at the position determined by its rank. If that position is already occupied by another tree root, we "demote" the one with the larger key, producing a new tree with a root of rank $k+1$. This new root may collide with another root of rank $k+1$ leading to another restructuring step (or possibly more). Figure 2 shows an example of how a set of tree roots are combined during the restructuring phase of a *deletemin*.

To implement the operation *decreasekey*(Δ, i, h), we first "cut" the edge from the tree node for i to its parent, and merge the node into the root list for h. We then decrease the key of i by Δ and check to see if this makes the key of i smaller than the key of h. In order to obtain the desired bound on the ranks, we need to limit the number of cuts that occur at any node during *decreasekey* operations. Specifically, we require that no non-root node lose more than one child. If a *changekey* would cause a non-root node x to lose a second child, we trigger an additional cut between x and its parent, making x a tree root also. In general, a *decreasekey* can produce a sequence

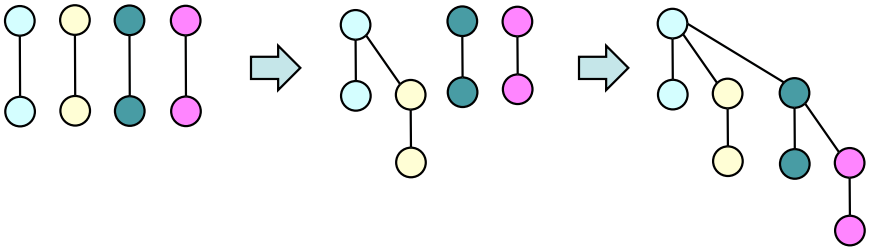

Figure 2: Linking of tree roots by deletemin

of *cascading cuts*, causing several new trees to be added to the heap. The *mark* bits are used to implement this restructuring. Mark bits are cleared whenever a node becomes a tree root and are set whenever a non-root node loses a child. Before setting the mark bit for a node, we first check to if it's already set. If it is, we trigger another cut. An example of the restructuring performed by a *decreasekey* operation is shown in Figure 3.

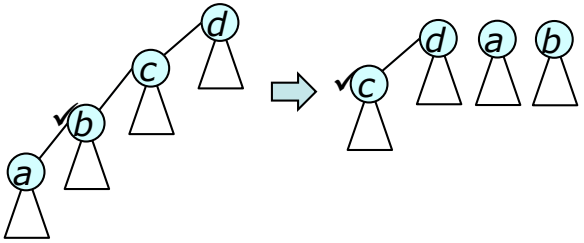

Figure 3: Restructuring performed by decreasekey