11 Fibonacci Heaps

The Fibonacci heap is a data structure implementing the priority queue abstract data type, just like the ordinary heap but more complicated and asymptotically faster for some operations. We first introduce binomial trees, which are special heap-ordered trees, and then explain Fibonacci heaps as collections of heap-ordered trees.

Binomial trees. The *binomial tree* of height h is a tree obtained from two binomial trees of height $h-1$, by linking the root of one to the other. The binomial tree of height 0 consists of a single node. Binomial trees of heights up to 4 are shown in Figure 44. Each step in the construction increases the height by one, increases the *degree* (the number of children) of the root by one, and doubles the size of the tree. It follows that a binomial tree of height h has root degree h and size 2^h. The root has the largest degree of any node in the binomial tree, which implies that every node in a binomial tree with n nodes has degree at most $\log_2 n$.

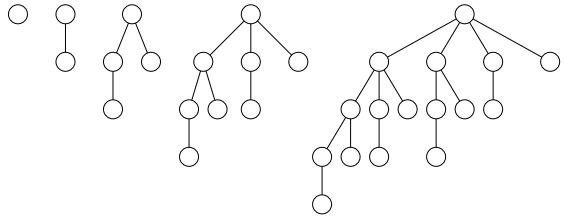

Figure 44: Binomial trees of heights 0, 1, 2, 3, 4. Each tree is obtained by linking two copies of the previous tree.

To store any set of items with priorities, we use a small collection of binomial trees. For an integer n, let n_i be the i-th bit in the binary notation, so we can write $n = \sum_{i\geq 0} n_i 2^i$. To store n items, we use a binomial tree of size 2^i for each $n_i = 1$. The total number of binomial trees is thus the number of 1's in the binary notation of n, which is at most $\log_2(n+1)$. The collection is referred to as a *binomial heap*. The items in each binomial tree are stored in heap-order. There is no specific relationship between the items stored in different binomial trees. The item with minimum key is thus stored in one of the logarithmically many roots, but it is not prescribed ahead of time in which one. An example is shown in Figure 45 where $11_{10} = 1011_2$ items are stored in three binomial trees with sizes 8, 2, and 1. In order to add a new item to the set, we create a new binomial tree of size 1 and we successively link binomial trees as dictated by the rules of adding 1 to the binary notation of n. In the example, we get $1011_2 + 1_2 = 1100_2$. The new collection thus consists of two binomial trees with sizes 8 and 4. The size 8 tree is the old one, and the size 4 tree is obtained by first linking the two size 1 trees and then linking the resulting size 2 tree to the old size 2 tree. All this is illustrated in Figure 45.

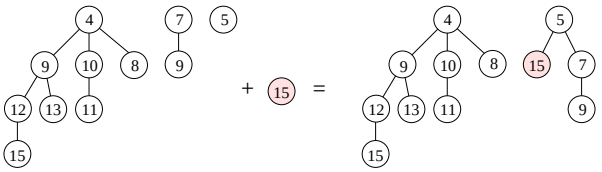

Figure 45: Adding the shaded node to a binomial heap consisting of three binomial trees.

Fibonacci heaps. A *Fibonacci heap* is a collection of heap-ordered trees. Ideally, we would like it to be a collection of binomial trees, but we need more flexibility. It will be important to understand how exactly the nodes of a Fibonacci heap are connected by pointers. Siblings are organized in doubly-linked cyclic lists, and each node has a pointer to its parent and a pointer to one of its children, as shown in Figure 46. Besides the pointers, each node stores a key, its degree, and a bit that can be used to mark or unmark the node. The roots of the heap-ordered trees are doubly-linked in a cycle, and there is an explicit pointer to the root that stores the item with the minimum key. Figure 47 illustrates a few basic operations we perform on a Fibonacci heap. Given two heap-ordered trees, we *link* them by making the root with the bigger key the child of the other root. To *unlink* a heap-ordered tree or subtree, we remove its root from the doubly-linked cycle. Finally, to *merge* two cycles, we cut both open and connect them at

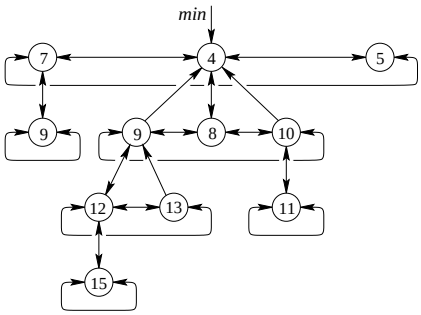

Figure 46: The Fibonacci heap representation of the first collection of heap-ordered trees in Figure 45.

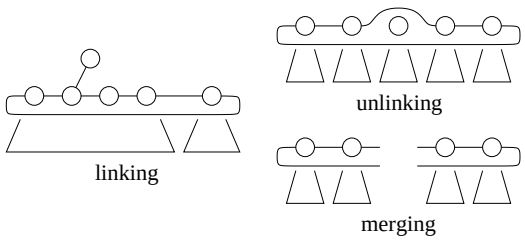

Figure 47: Cartoons for linking two trees, unlinking a tree, and merging two cycles.

their ends. Any one of these three operations takes only constant time.

Potential function. A Fibonacci heap supports a variety of operations, including the standard ones for priority queues. We use a potential function to analyze their amortized cost applied to an initially empty Fibonacci heap. Letting r_i be the number of roots in the root cycle and m_i the number of marked nodes, the *potential* after the i-th operation is $\Phi_i = r_i + 2m_i$. When we deal with a collection of Fibonacci heaps, we define its potential as the sum of individual potentials. The initial Fibonacci heap is empty, so $\Phi_0 = 0$. As usual, we let c_i be the actual cost and $a_i = c_i + \Phi_i - \Phi_{i-1}$ the amortized cost of the i-th operation. Since $\Phi_0 = 0$ and $\Phi_i \geq 0$ for all i, the actual cost is less than the amortized cost:

$$\sum_{i=1}^{n} c_i \;\leq\; \sum_{i=1}^{n} a_i \;=\; r_n + 2m_n + \sum_{i=1}^{n} c_i.$$

For some of the operations, it is fairly easy to compute the amortized cost. We get the **minimum** by returning the key in the marked root. This operation does not change the potential and its amortized and actual cost is $a_i = c_i = 1$. We **meld** two Fibonacci heaps, H_1 and H_2, by first merging the two root circles and second adjusting the pointer to the minimum key. We have

$$\begin{aligned} r_i(H) &= r_{i-1}(H_1) + r_{i-1}(H_2), \\ m_i(H) &= m_{i-1}(H_1) + m_{i-1}(H_2), \end{aligned}$$

which implies that there is no change in potential. The amortized and actual cost is therefore $a_i = c_i = 1$. We **insert** a key into a Fibonacci heap by first creating a new Fibonacci heap that stores only the new key and second melding the two heaps. We have one more node in the root cycle so the change in potential is $\Phi_i - \Phi_{i-1} = 1$. The amortized cost is therefore $a_i = c_i + 1 = 2$.

Deletemin. Next we consider the somewhat more involved operation of deleting the minimum key, which is done in four steps:

`Step 1.` Remove the node with minimum key from the root cycle.

`Step 2.` Merge the root cycle with the cycle of children of the removed node.

`Step 3.` As long as there are two roots with the same degree link them.

`Step 4.` Recompute the pointer to the minimum key.

For Step 3, we use a pointer array R. Initially, $R[i] =$ NULL for each i. For each root ϱ in the root cycle, we execute the following iteration.

$i = \varrho \rightarrow degree$;
`while` $R[i] \neq$ NULL `do`
  $\varrho' = R[i]$; $R[i] =$ NULL; $\varrho =$ LINK(ϱ, ϱ'); i`++`
`endwhile`;
$R[i] = \varrho$.

To analyze the amortized cost for deleting the minimum, let $D(n)$ be the maximum possible degree of any node in a Fibonacci heap of n nodes. The number of linking operations in Step 3 is the number of roots we start with, which is less than $r_{i-1} + D(n)$, minus the number of roots we end up with, which is r_i. After Step 3, all roots have different degrees, which implies $r_i \leq D(n)+1$. It follows that the actual cost for the four steps is

$$\begin{aligned} c_i &\leq 1 + 1 + (r_{i-1} + D(n) - r_i) + (D(n) + 1) \\ &= 3 + 2D(n) + r_{i-1} - r_i. \end{aligned}$$

The potential change is $\Phi_i - \Phi_{i-1} = r_i - r_{i-1}$. The amortized cost is therefore $a_i = c_i + \Phi_i - \Phi_{i-1} \leq 2D(n) + 3$. We will prove next time that the maximum possible degree is at most logarithmic in the size of the Fibonacci heap, $D(n) < 2\log_2(n+1)$. This implies that deleting the minimum has logarithmic amortized cost.

Decreasekey and delete. Besides deletemin, we also have operations that delete an arbitrary item and that decrease the key of an item. Both change the structure of the heap-ordered trees and are the reason why a Fibonacci heap is not a collection of binomial trees but of more general heap-ordered trees. The **decreasekey** operation replaces the item with key x stored in the node ν by $x - \Delta$, where $\Delta \geq 0$. We will see that this can be done more efficiently than to delete x and to insert $x - \Delta$. We decrease the key in four steps.

`Step 1.` Unlink the tree rooted at ν.

`Step 2.` Decrease the key in ν by Δ.

`Step 3.` Add ν to the root cycle and possibly update the pointer to the minimum key.

`Step 4.` Do cascading cuts.

We will explain cascading cuts shortly, after explaining the four steps we take to delete a node ν. Before we delete a node ν, we check whether $\nu = min$, and if it is then we delete the minimum as explained above. Assume therefore that $\nu \neq min$.

`Step 1.` Unlink the tree rooted at ν.

`Step 2.` Merge the root-cycle with the cycle of ν's children.

`Step 3.` Dispose of ν.

`Step 4.` Do cascading cuts.

Figure 48 illustrates the effect of decreasing a key and of deleting a node. Both operations create trees that are not

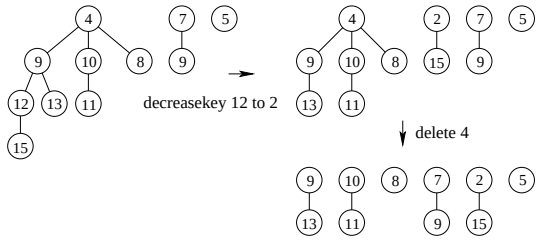

Figure 48: A Fibonacci heap initially consisting of three binomial trees modified by a decreasekey and a delete operation.

binomial, and we use cascading cuts to make sure that the shapes of these trees are not very different from the shapes of binomial trees.

Cascading cuts. Let ν be a node that becomes the child of another node at time t. We mark ν when it loses its first child after time t. Then we unmark ν, unlink it, and add it to the root-cycle when it loses its second child thereafter. We call this operation a *cut*, and it may cascade because one cut can cause another, and so on. Figure 49 illustrates the effect of cascading in a heap-ordered tree with two marked nodes. The first step decreases key 10 to 7, and the second step cuts first node 5 and then node 4.

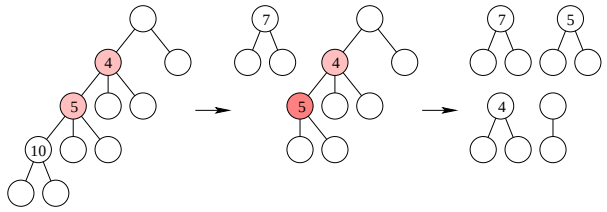

Figure 49: The effect of cascading after decreasing 10 to 7. Marked nodes are shaded.

Summary analysis. As mentioned earlier, we will prove $D(n) < 2\log_2(n+1)$ next time. Assuming this bound, we are able to compute the amortized cost of all operations. The actual cost of Step 4 in decreasekey or in delete is the number of cuts, c_i. The potential changes because there are c_i new roots and c_i fewer marked nodes. Also, the last cut may introduce a new mark. Thus

$$\begin{aligned} \Phi_i - \Phi_{i-1} &= r_i - r_{i-1} + 2m_i - 2m_{i-1} \\ &\leq c_i - 2c_i + 2 \\ &= -c_i + 2. \end{aligned}$$

The amortized cost is therefore $a_i = c_i + \Phi_i - \Phi_{i-1} \leq c_i - (2 - c_i) = 2$. The first three steps of a decreasekey operation take only a constant amount of actual time and increase the potential by at most a constant amount. It follows that the amortized cost of decreasekey, including the cascading cuts in Step 4, is only a constant. Similarly, the actual cost of a delete operation is at most a constant, but Step 2 may increase the potential of the Fibonacci heap by as much as $D(n)$. The rest is bounded from above by a constant, which implies that the amortized cost of the delete operation is O($\log n$). We summarize the amortized cost of the various operations supported by the Fibonacci heap:

find the minimum	O(1)
meld two heaps	O(1)
insert a new item	O(1)
delete the minimum	O($\log n$)
decrease the key of a node	O(1)
delete a node	O($\log n$)

We will later see graph problems for which the difference in the amortized cost of the decreasekey and delete operations implies a significant improvement in the running time.