7 Fibonacci Heaps (February 11 and 16)

7.1 Mergeable Heaps

A *mergeable heap* is a data structure that stores a collection of *keys*[1] and supports the following operations.

- INSERT: Insert a new key into a heap. This operation can also be used to create a new heap containing just one key.
- FINDMIN: Return the smallest key in a heap.
- DELETEMIN: Remove the smallest key from a heap.
- MERGE: Merge two heaps into one. The new heap contains all the keys that used to be in the old heaps, and the old heaps are (possibly) destroyed.

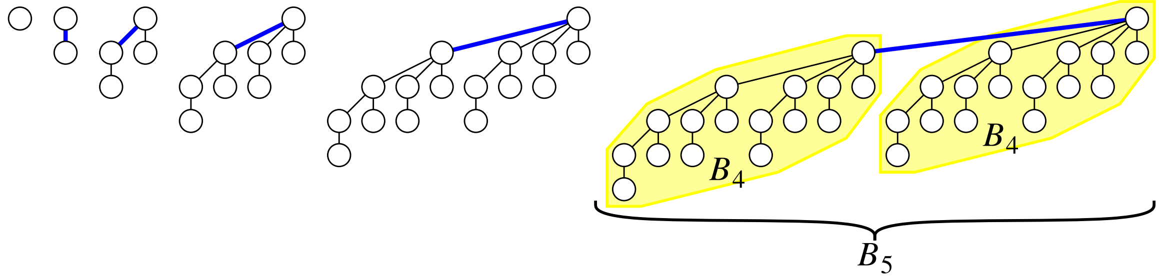
If we never had to use DELETEMIN, mergeable heaps would be completely trivial. Each "heap" just stores to maintain the single record (if any) with the smallest key. INSERTS and MERGES require only one comparison to decide which record to keep, so they take constant time. FINDMIN obviously takes constant time as well.

If we need DELETEMIN, but we don't care how long it takes, we can still implement mergeable heaps so that INSERTS, MERGES, and FINDMINS take constant time. We store the records in a circular doubly-linked list, and keep a pointer to the minimum key. Now deleting the minimum key takes $\Theta(n)$ time, since we have to scan the linked list to find the new smallest key.

In this lecture, I'll describe a data structure called a *Fibonacci heap* that supports INSERTS, MERGES, and FINDMINS in constant time, even in the worst case, and also handles DELETEMIN in $O(\log n)$ *amortized* time. That means that any sequence of n INSERTS, m MERGES, f FINDMINS, and d DELETEMINS takes $O(n + m + f + d \log n)$ time.

7.2 Binomial Trees and Fibonacci Heaps

A *Fibonacci heap* is a circular doubly linked list, with a pointer to the minimum key, but the elements of the list are not single keys. Instead, we collect keys together into structures called *binomial heaps*. Binomial heaps are trees[2] that satisfy the heap property (every node has a smaller key than its children) and have the following special structure.



Binomial trees of order 0 through 5.

[1]In the previous lecture on treaps, I called the keys *priorities* to distinguish them from search keys.

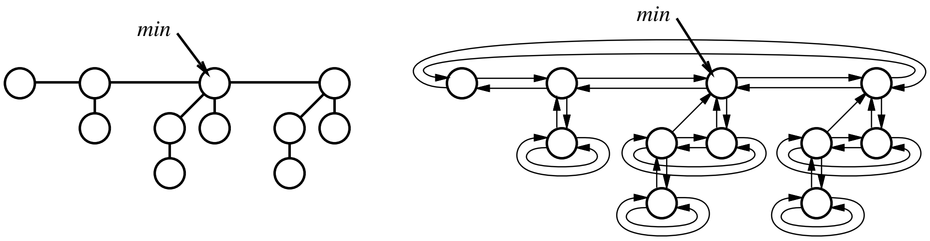
[2][CLR] uses the name "binomial heap" to describe a more complicated data structure consisting of a set of heap-ordered binomial trees, with at most one binomial tree of each order.

A k*th order binomial tree*, which I'll abbreviate B_k, is defined recursively. B_0 is a single node. For all $k > 0$, B_k consists of two copies of B_{k-1} that have been *linked* together, meaning that the root of one B_{k-1} has become a new child of the other root.

Binomial trees have several useful properties, which are easy to prove by induction (hint, hint).

- The root of B_k has degree k.
- The children of the root of B_k are the roots of $B_0, B_1, \ldots, B_{k-1}$.
- B_k has height k.
- B_k has 2^k nodes.
- B_k has $\binom{k}{r}$ nodes at depth r, for all $0 \leq r \leq k$.
- B_k has 2^{k-r-1} nodes with height r, for all $0 \leq r < k$, and one node (the root) with height k.

Although we normally don't care in this class about the low-level details of data structures, we need to be specific about how Fibonacci heaps are actually implemented, so that we can be sure that certain operations can be performed quickly. Every node in a Fibonacci heap points to four other nodes: its parent, its "next" sibling, its "previous" sibling, and one of its children. The sibling pointers are used to join the roots together into a circular doubly-linked *root list*. In each binomial tree, the children of each node are also joined into a circular doubly-linked list using the sibling pointers.



A high-level view and a detailed view of the same Fibonacci heap. Null pointers are omitted for clarity.

With this representation, we can add or remove nodes from the root list, merge two root lists together, link one two binomial tree to another, or merge a node's list of children with the root list, in constant time, and we can visit every node in the root list in constant time per node. Having established that these primitive operations can be performed quickly, we never again need to think about the low-level representation details.

7.3 Operations on Fibonacci Heaps

The INSERT, MERGE, and FINDMIN algorithms for Fibonacci heaps are exactly like the corresponding algorithms for linked lists. Since we maintain a pointer to the minimum key, FINDMIN is trivial. To insert a new key, we add a single node (which we should think of as a B_0) to the root list and (if necessary) update the pointer to the minimum key. To merge two Fibonacci heaps, we just merge the two root lists and keep the pointer to the smaller of the two minimum keys. Clearly, all three operations take $O(1)$ time.

Deleting the minimum key is a little more complicated. First, we remove the minimum key from the root list and splice its children into the root list. Except for updating the parent pointers, this takes $O(1)$ time. Then we scan through the root list to find the new smallest key and update the parent pointers of the new roots. This scan could take $\Theta(n)$ time in the worst case. To bring down the *amortized* deletion time, we apply a CLEANUP algorithm, which links pairs of equal-size binomial heaps until there is only one binomial heap of any particular size.

Let me describe the CLEANUP algorithm in more detail, so that we can analyze its running time. In the following algorithm, each $B[i]$ is a pointer to some already-visited binomial heap of order i, or NULL if there is no such binomial heap. Notice that CLEANUP also resets all the parent pointers and updates the pointer to the minimum key.

```
CLEANUP:
        minkey ← some node in the root list
        for i ← 0 to lg n
            B[i] = NULL

        for each node v in the root list
(⋆)         parent(v) ← NULL
            w ← B[deg(v)]
            while w ≠ NULL
                B[deg(v)] ← NULL
                if key(v) ≤ key(w)
                    swap v ⟷ w
(⋆⋆)            remove w from the root list
                link w to v
                w ← B[deg(v)]
            B[deg(v)] ← v
            if key(minkey) > key(v)
                minkey ← v
```

The running time of CLEANUP is $O(r)$, where r is the length of the root list just before CLEANUP is called. The easiest way to see this is to count the number of times the two starred lines can be executed: $(\star)$ is executed once for every node v on the root list, and $(\star\star)$ is executed *at most* once for every node w on the root list. In the worst case (if there have been no previous deletions, for example) $r = n$, so the worst-case running time is linear. After CLEANUP is finished, $r = O(\log n)$, since all the binomial heaps have unique orders and the largest has order at most $\log_2 n$.

The time for a DELETEMIN is $O(r + \deg(\min))$, where min is the node deleted. Although $\deg(\min)$ is at most $\lg n$, we can still have $r = \Theta(n)$, so the worst-case time for a DELETEMIN is $\Theta(n)$. After a DELETEMIN, $r = O(\log n)$.

7.4 Amortized Analysis of DELETEMIN

To bound the amortized cost, observe that each insertion increments r. If we charge each insertion a constant "cleanup tax", and use the collected tax to pay for the CLEANUP algorithm, the unpaid cost of a DELETEMIN is only $O(\deg(\min)) = O(\log n)$.

If we want to be more formal about it, we can define the *potential* of the Fibonacci heap to be an appropriate constant times the number of roots: $\boxed{\Phi = \alpha r}$. Recall that the amortized time of an operation can be defined as its actual time plus the change in the potential, as long as $\Phi = 0$ initially (it is) and we always have $\Phi \geq 0$ (we do). Each INSERT increases the potential Φ by $\alpha = \Theta(1)$, so the amortized cost is still constant. A MERGE actually doesn't change Φ at all, since the new Fibonacci heap has all the roots from its constituents and no others, so its amortized cost is $O(1)$. The actual cost of a DELETEMIN is $O(r + \log n)$, and it increases Φ by $O(\log n) - \alpha r$, so provided we choose a large enough constant α, the amortized cost of a DELETEMIN is $O(\log n)$.

7.5 Deleting Arbitrary Nodes

In some applications of heaps, we also need the ability to delete an arbitrary node. The usual way to do this is to decrease the node's key to $-\infty$, and then use DELETEMIN. Here I'll describe how to decrease the key of a node in a Fibonacci heap; the algorithm will take $O(\log n)$ time in the worst case, but the amortized time will be only $O(1)$.

Our algorithm for decreasing the key at a node v follows two simple rules.

1. Promote v up to the root list. (This moves the whole subtree rooted at v.)

2. As soon as two children of any node w have been promoted, immediately promote w.

In order to enforce the second rule, we now *mark* certain nodes in the Fibonacci heap. Specifically, a node is marked if exactly one of its children has been promoted. If some child of a marked node is promoted, we promote (and unmark) that node as well.

Here's a more formal description of the algorithm. The input is a pointer to a node v and the new value k for its key.

```
DecreaseKey(v, k):
    key(v) ← k
    unmark v
    update the pointer to the smallest key
    if parent(v) ≠ Null
        remove v from parent(v)'s list of children
        insert v into the root list
        if parent(v) is unmarked
            mark parent(v)
        else
            DecreaseKey(parent(v), key(parent(v)))
```

Note that the algorithm calls itself recursively. The effect of the recursive calls is a "cascading promotion"—each consecutive marked ancestor of v is promoted to the root list and unmarked, otherwise unchanged. (Notice that the recursive calls do not actually decrease the ancestors' keys!) The lowest unmarked ancestor is then marked, since one of its children has been promoted.

The time to decrease the key of a node v is

$$O(1 + \#\text{consecutive marked ancestors}),$$

which is $O(\text{depth}(v))$. If we still had full binomial heaps, then this would be $O(\log n)$, but we don't — promoting nodes destroys the nice recursive structure of binomial trees, so it is no longer obvious that our component heaps have logarithmic depth.

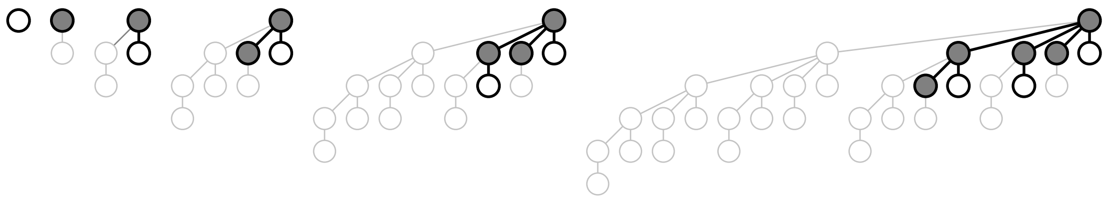
We have a similar problem with DeleteMin also. Our amortized analysis used the fact that the maximum degree of any node is $O(\log n)$, which implies that after a CLEEANUP the number of nodes in the root list is $O(\log n)$. But now that we don't have complete binomial heaps, this "fact" isn't quite as obvious.

7.6 Bounding the Degree (and the Depth)

First let's prove that the maximum degree is still only $O(\log n)$; the proof of depth will be almost exactly the same, For any node v, let $|v|$ denote the number of nodes in the subtree of v, including v itself. Our proof uses the following lemma, which *finally* tells us why these things are called Fibonacci heaps!

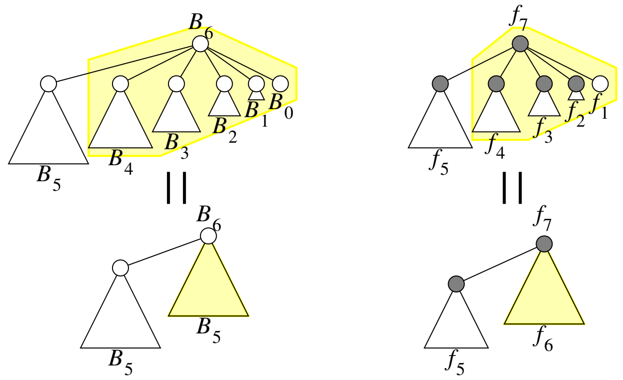
Lemma 1. *For any node v in a Fibonacci heap, $|v| \geq F_{\deg(v)+2}$.*

Before we get to the proof, let me give some intuition about why this lemma is true. Consider how many nodes we can remove from a binomial heap of order k without causing any cascading promotions. The most damage we can do is by removing the largest subtree of every node in B_k. Call the result a *Fibonacci tree* of order $k+1$, and denote it f_{k+1}. As a base case, let f_1 be the tree with one (unmarked) node, that is, $f_1 = B_0$. The reason for shifting the index will become clear shortly (if it isn't already).



Fibonacci trees of order 1 through 6. Light nodes have been promoted away; dark nodes are marked.

Recall that the root of a binomial tree B_k has k children, which are roots of $B_0, B_1, \ldots, B_{k-1}$. To convert B_k to f_{k+1}, we promote the root of B_{k-1}, and recursively convert each of the other subtrees B_i to f_{i+1}. The root of the resulting tree f_{k+1} has degree $k-1$, and the children are the roots of smaller Fibonacci trees $f_1, f_2, \ldots, f_{k-1}$. We can also consider B_k as two copies of B_{k-1} linked together. It's quite easy to show that an order-k Fibonacci tree consists of an order $k-2$ Fibonacci tree linked to an order $k-1$ Fibonacci tree. (See the picture below.)



Comparing the recursive structures of B_6 and f_7.

Since f_1 and f_2 both have exactly one node, the number of nodes in an order-k Fibonacci tree is exactly the kth Fibonacci number. (That's why we changed in the index.) Since the degree of the root of f_k is $k-2$, Lemma 1 is true for Fibonacci trees.

Fibonacci trees will turn out to be the worst case for Lemma 1, but let's prove it in general.

Proof (Lemma 1): Let w_i be the ith child (in chronological order) added to v. I claim that $\deg(i) \geq i-2$. We can prove this claim using induction. Assume that when w_k was linked to v, our claim was true for all $i < k$. Then w_k was linked to v, $\deg(w_k) = \deg(v) = k-1$. Since that time, at most one child of w_k has been removed, since otherwise w_k would have been promoted to the root list already. So $\deg(w_k) \geq k-2$, as claimed. (What's the base case?)

We can also quickly observe that $\deg(w_1) \geq 0$. (Duh.)

Now we're almost done. Let s_d be the minimum possible size of a tree with degree d in any Fibonacci heap. My earlier claim implies that

$$s_d \geq 2 + \sum_{i=2}^{d} s_{i-2}$$

If we assume inductively that $s_i \geq F_{i+2}$ for all $i < d$ (with the easy base case $s_0 \geq F_2 = 1$), we have

$$s_d \geq 2 + \sum_{i=2}^{d} F_i = 1 + \sum_{i=0}^{d} F_i = F_{d+2}.$$

The last step was a practice problem on HW0! □

You can easily show (using either induction or the annihilator method) that $F_{k+2} > \phi^k$ where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the golden ratio. Thus, Lemma 1 implies that

$$\deg(v) \leq \log_\phi |v| = O(\log |v|),$$

which is just what we wanted. Since the size of any subtree in an n-node Fibonacci heap is obviously at most n, the degree of any node is $O(\log n)$, which is exactly what we wanted.

A similar proof shows that the depth of any node is also $O(\log n)$. Intuitively we can see this by observing that an order-k Fibonacci tree has depth exactly $k-2$. As it turns out, though, we don't actually need to prove this, since the depth won't be considered in the amortized analysis.

7.7 Analyzing DecreaseKey

To compute the amortized cost of DecreaseKey, we'll use the potential method, just as we did for DeleteMin. One general idea that helps to find an appropriate potential function Φ is to try to follow two simple rules:

- The potential Φ should go *up a little* whenever we do a little work.
- The potential Φ should go *down a lot* whenever we do a lot of work.

Recall that the actual cost of DecreaseKey(v, k) is 1 plus the number of consecutive marked ancestors of v. Our algorithm unmarks each of those marked ancestors, and possibly also marks one node. So *the number of marked nodes* might be an appropriate potential function here.

Whenever we do a little bit of work, the number of marks goes up by at most one; whenever we do a lot of work, the number of marks goes down a lot.

More precisely, let m and m' be the number of marked nodes before and after a DecreaseKey operation. The actual time is

$$t = 1 + \#\text{consecutive marked ancestors of } v$$

and if we set $\Phi = m$, the change in potential is

$$\Delta\phi = m' - m \leq 1 - \#\text{consecutive marked ancestors of } v,$$

so the amortized cost of DecreaseKey is $t + \Delta\Phi \leq 2 = O(1)$.

7.8 Reanalyzing DeleteMin (and everything else)

Unfortunately, our analyses of DeleteMin and DecreaseKey used two different potential functions. Unless we can find a *single* potential function that works for *both* operations, we can't claim both amortized time bounds simultaneously. So we need to find a potential function Φ that goes up a little during a cheap DeleteMin or a cheap DecreaseKey, and goes down a lot during an expensive DeleteMin or an expensive DecreaseKey.

Let's look a little more carefully at the cost of each Fibonacci heap operation, and its effect on both the number of roots and the number of marked nodes, the things we used as out earlier potential functions. Let r and m be the numbers of roots and marks before each operation, and let r' and m' be the numbers of roots and marks after the operation.

operation	actual cost	$r' - r$	$m' - m$
Insert	1	1	0
Merge	1	0	0
DeleteMin	$r + r'$	$r' - r$	0
DecreaseKey	$1 + m - m'$	$1 + m - m'$	$m' - m$

In particular, notice that promoting a node in DecreaseKey requires constant time and increases the number of roots by one, and that we promote (at most) one unmarked node.

If we guess that the correct potential function is a linear combination of our old potential functions r and m and play around with various possibilities for the coefficients, we will eventually stumble across the correct answer:

$$\boxed{\Phi = r + 2m}$$

To see that this potential function gives us good amortized bounds for every Fibonacci heap operation, let's add two more columns to our table.

operation	actual cost	$r' - r$	$m' - m$	$\Phi' - \Phi$	amortized cost
Insert	1	1	0	2	3
Merge	1	0	0	0	1
DeleteMin	$r + r'$	$r' - r$	0	$2r$	r'
DecreaseKey	$1 + m - m'$	$1 + m - m'$	$m' - m$	$1 + m' - m$	2

Since Lemma 1 implies that $r' = O(\log n)$, we're done!