B Fibonacci Heaps

B.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.

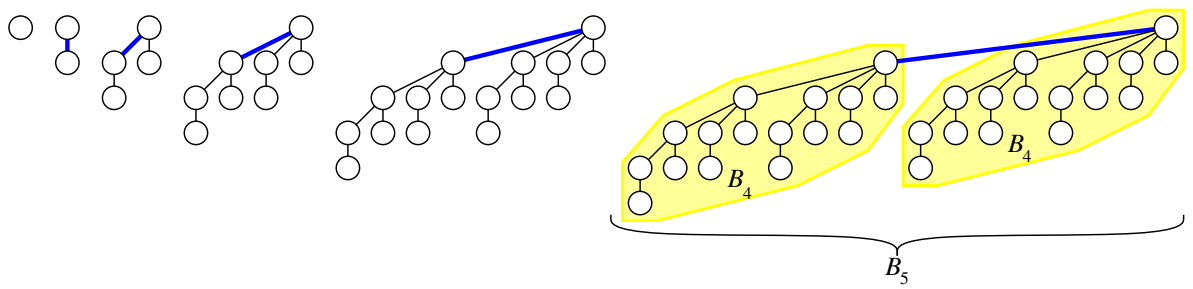
If we never had to use DeleteMin, mergeable heaps would be completely trivial. Each "heap" just stores to maintain the single record (if any) with the smallest key. Inserts and Merges require only one comparison to decide which record to keep, so they take constant time. FindMin obviously takes constant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

B.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 earlier lecture on treaps, I called these keys *priorities* to distinguish them from search keys.

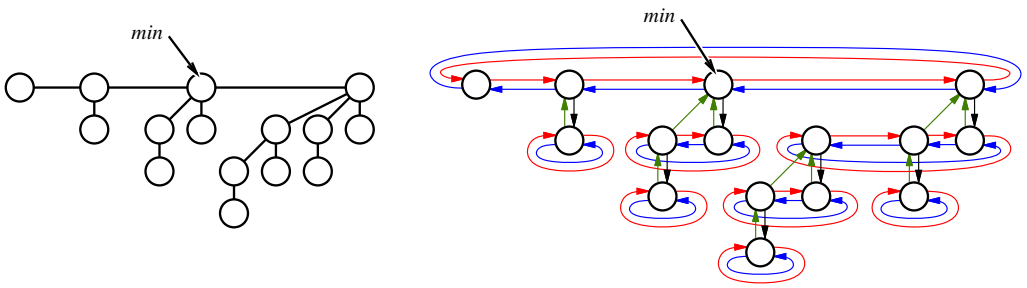
[2] CLR uses the name 'binomial heap' to describe a more complicated data structure consisting of a set of heap-ordered binomial trees, with at most one binomial tree of each order.

A *kth 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 can be obtained from B_{k-1} by adding a new child to every node.
- B_k has $\binom{k}{d}$ nodes at depth d, for all $0 \le d \le k$.
- B_k has 2^{k-h-1} nodes with height h, for all $0 \le h < k$, 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.

B.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.

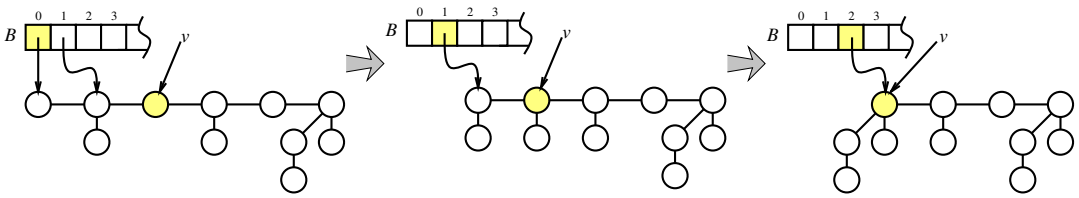
Deleting the minimum key is a little more complicated. First, we remove the minimum key from the root list and splice its children into the root list. Except for updating the parent pointers, this takes $O(1)$ time. Then we scan through the root list to find the new smallest key and update the parent 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 we can analyze its running time. The following algorithm maintains a global array $B[1..\lfloor \lg n \rfloor]$, where $B[i]$ is a pointer to some previously-visited binomial heap of order i, or NULL if there is no such binomial heap. Notice that CLEANUP simultaneously resets the parent pointers of all the new roots and updates the pointer to the minimum key. I've split off the part of the algorithm that merges binomial heaps of the same order into a separate subroutine MERGEDUPES.

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

  for all nodes v in the root list
      parent(v) ← NULL     (⋆)
      if key(newmin) > key(v)
          newmin ← v
      MERGEDUPES(v)
```

```
MERGEDUPES(v):
  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
```



MERGEDUPES(v), ensuring that no earlier root has the same degree as v.

Notices that MERGEDUPES is careful to merge heaps so that the heap property is maintained—the heap whose root has the larger key becomes a new child of the heap whose root has the smaller key. This is handled by swapping v and w if their keys are in the wrong order.

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: line $(\star)$ is executed once for every node v on the root list, and line $(\star\star)$ is executed *at most* once for every node w on the root list. Since DELETEMIN does only a constant amount of work before calling CLEANUP, the running time of DELETEMIN is $O(r') = O(r + \deg(\text{min}))$ where r is the number of roots before DELETEMIN begins, and min is the node deleted.

Although $\deg(\text{min})$ is at most $\lg n$, we can still have $r = \Theta(n)$ (for example, if nothing has been deleted yet), so the worst-case time for a DELETEMIN is $\Theta(n)$. After a DELETEMIN, the root list has length $O(\log n)$, since all the binomial heaps have unique orders and the largest has order at most $\lfloor \lg n \rfloor$.

B.4 Amortized Analysis of DeleteMin

To bound the amortized cost, observe that each insertion increments r. If we charge a constant 'cleanup tax' for each insertion, and use the collected tax to pay for the CLEANUP algorithm, the

unpaid cost of a DELETEMIN is only $O(\deg(\min)) = O(\log n)$.

More formally, define the *potential* of the Fibonacci heap to be the number of roots. Recall that the amortized time of an operation can be defined as its actual running time plus the increase in potential, provided the potential is initially zero (it is) and we never have negative potential (we never do). Let r be the number of roots before a DELETEMIN, and let r'' denote the number of roots afterwards. The actual cost of DELETEMIN is $r + \deg(min)$, and the number of roots increases by $r'' - r$, so the amortized cost is $r'' + \deg(\min)$. Since $r'' = O(\log n)$ and the degree of any node is $O(\log n)$, the amortized cost of DELETEMIN is $O(\log n)$.

Each INSERT adds only one root, so its amortized cost is still constant. A MERGE actually doesn't change the number of roots, since the new Fibonacci heap has all the roots from its constituents and no others, so its amortized cost is also constant.

B.5 Decreasing Keys

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.

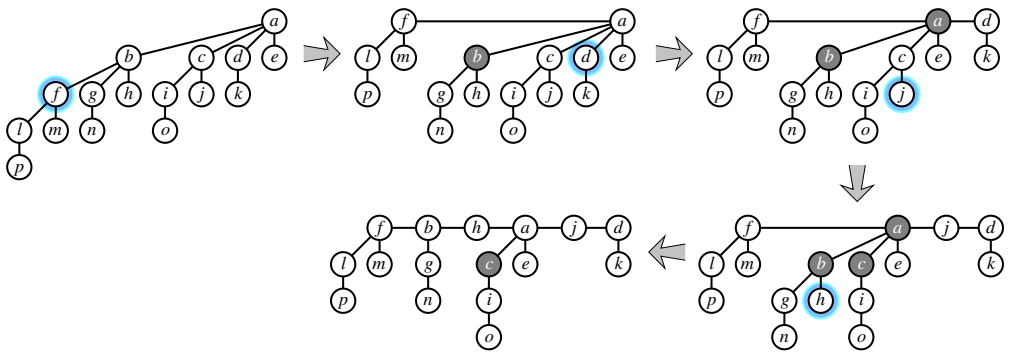
In order to enforce the second rule, we now *mark* certain nodes in the Fibonacci heap. Specifically, a node is marked if exactly one of its children has been promoted. If some child of a marked node is promoted, we promote (and unmark) that node as well. Whenever we promote a marked node, we unmark it; this is the*only* way to unmark a node. (Specifically, splicing nodes into the root list during a DELETEMIN is not considered a promotion.)

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
  update the pointer to the smallest key
  PROMOTE(v)
```

```
PROMOTE(v):
  unmark v
  if parent(v) ≠ NULL
      remove v from parent(v)'s list of children
      insert v into the root list
      if parent(v) is marked
          PROMOTE(parent(v))
      else
          mark parent(v)
```

The PROMOTE algorithm calls itself recursively, resulting in a 'cascading promotion'. Each consecutive marked ancestor of v is promoted to the root list and unmarked, otherwise unchanged. The lowest unmarked ancestor is then marked, since one of its children has been promoted.



Decreasing the keys of four nodes: first f, then d, then j, and finally h. Dark nodes are marked. DecreaseKey(h) causes nodes b and a to be recursively promoted.

The time to decrease the key of a node v is $O(1+\#\text{consecutive marked ancestors of } v)$. Binomial heaps have logarithmic depth, so if we still had only full binomial heaps, the running time would be $O(\log n)$. Unfortunately, promoting nodes destroys the nice binomial tree structure; our trees no longer have logarithmic depth! In fact, DecreaseKey runs in $\Theta(n)$ time in the worst case.

To compute the amortized cost of DecreaseKey, we'll use the potential method, just as we did for DeleteMin. We need to find a potential function Φ that goes up a little whenever we do a little work, and goes down a lot whenever we do a lot of work. DecreaseKey unmarks several 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 (ignoring constant factors) is

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

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

$$m' - m \leq 1 - \#\text{consecutive marked ancestors of } v.$$

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

B.6 Bounding the Degree

But now we have a problem with our earlier analysis of DeleteMin. The amortized time for a DeleteMin is still $O(r + \deg(\min))$. To show that this equaled $O(\log n)$, we used the fact that the maximum degree of any node is $O(\log n)$, which implies that after a Cleanup the number of roots is $O(\log n)$. But now that we don't have complete binomial heaps, this 'fact' is no longer obvious!

So let's prove it. 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, $|v| \geq F_{\deg(v)+2}$.*

Proof: Label the children of v in the chronological order in which they were linked to v. Consider the situation just before the ith oldest child w_i was linked to v. At that time, v had at least $i-1$ children (possibly more). Since Cleanup only links trees with the same degree, we had $\deg(w_i) =$

$\deg(v) \geq i - 1$. Since that time, at most one child of w_i has been promoted away; otherwise, w_i would have been promoted to the root list by now. So currently we have $\deg(w_i) \geq i - 2$.

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

Let s_d be the minimum possible size of a tree with degree d in any Fibonacci heap. Clearly $s_0 = 1$; for notational convenience, let $s_{-1} = 1$ also. By our earlier argument, the ith oldest child of the root has degree at least $\max\{0, i-2\}$, and thus has size at least $\max\{1, s_{i-2}\} = s_{i-2}$. Thus, we have the following recurrence:

$$s_d \geq 1 + \sum_{i=1}^{d} s_{i-2}$$

If we assume inductively that $s_i \geq F_{i+2}$ for all $-1 \leq i < d$ (with the easy base cases $s_{-1} = F_1$ and $s_0 = F_2$), we have

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

(The last step was a practice problem in Homework 0.) By definition, $|v| \geq s_{\deg(v)}$. □

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

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

Thus, 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. Our earlier analysis is still good.

B.7 Analyzing Everything Together

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	$r + \deg(min)$	$r' - r$	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:

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

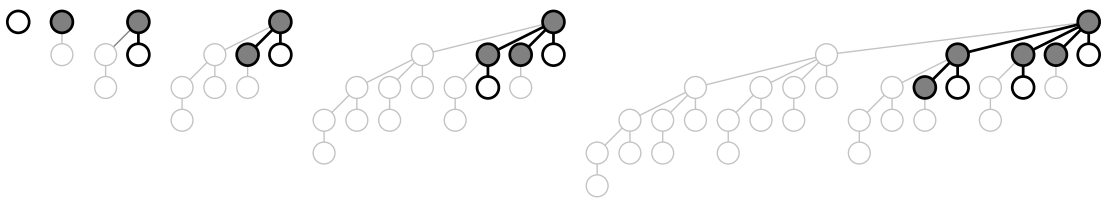
To see that this potential function gives us good amortized bounds for every Fibonacci heap operation, let's add two more columns to our table.

operation	actual cost	$r' - r$	$m' - m$	$\Phi' - \Phi$	amortized cost
INSERT	1	1	0	1	2
MERGE	1	0	0	0	1
DELETEMIN	$r + \deg(\min)$	$r' - r$	0	$r' - r$	$r' + \deg(\min)$
DECREASEKEY	$1 + m - m'$	$1 + m - m'$	$m' - m$	$1 + m' - m$	2

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

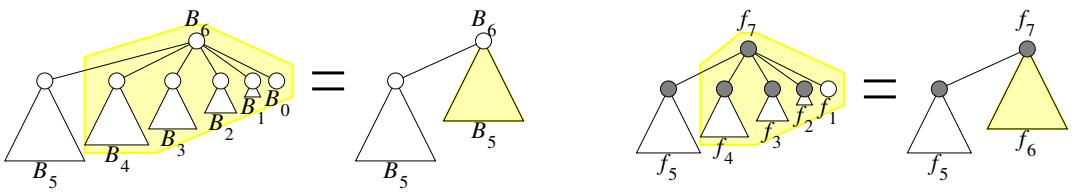
B.8 Fibonacci Trees

To give you a little more intuition about how Fibonacci heaps behave, let's look at a worst-case construction for Lemma 1. Suppose we want to remove as many nodes as possible from a binomial heap of order k, by promoting various nodes to the root list, but without causing any cascading promotions. The most damage we can do is to promote the largest subtree of every node. 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 should be obvious after a few seconds.



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.) Like binomial trees, Fibonacci trees have lots of other nice properties that easy to prove by induction (hint, hint):

- The root of f_k has degree $k - 2$.
- f_k can be obtained from f_{k-1} by adding a new unmarked child to every marked node and then marking all the old unmarked nodes.

- f_k has height $\lceil k/2 \rceil - 1$.
- f_k has F_{k-2} unmarked nodes, F_{k-1} marked nodes, and thus F_k nodes altogether.
- f_k has $\binom{k-d-2}{d-1}$ unmarked nodes, $\binom{k-d-2}{d}$ marked nodes, and $\binom{k-d-1}{d}$ total nodes at depth d, for all $0 \le d \le \lfloor k/2 \rfloor - 1$.
- f_k has F_{k-2h-1} nodes with height h, for all $0 \le h \le \lfloor k/2 \rfloor - 1$, and one node (the root) with height $\lceil k/2 \rceil - 1$.