Lecture 8 Binomial Heaps

Binomial heaps were invented in 1978 by J. Vuillemin [106]. They give a data structure for maintaining a collection of elements, each of which has a *value* drawn from an ordered set, such that new elements can be added and the element of minimum value extracted efficiently. They admit the following operations:

makeheap(i)	return a new heap containing only element i
findmin(h)	return a pointer to the element of h of minimum value
insert(h, i)	add element i to heap h
deletemin(h)	delete the element of minimum value from h
meld(h, h')	combine heaps h and h' into one heap

Efficient searching for objects is not supported.

In the next lecture we will extend binomial heaps to *Fibonacci heaps* [35], which allow two additional operations:

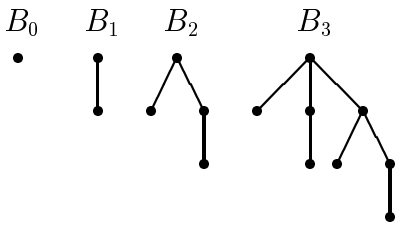
decrement(h, i, Δ)	decrease the value of i by Δ
delete(h, i)	remove i from heap h

We will see that these operations have low *amortized* costs. This means that any particular operation may be expensive, but the costs average out so that over a sequence of operations, the number of steps per operation of each type is small. The amortized cost per operation of each type is given in the following table:

makeheap	$O(1)$
findmin	$O(1)$
insert	$O(1)$
deletemin	$O(\log n)$
meld	$O(1)$ for the lazy version
	$O(\log n)$ for the eager version
decrement	$O(1)$
delete	$O(\log n)$

where n is the number of elements in the heap.

Binomial heaps are collections of *binomial trees*, which are defined inductively: the i^{th} binomial tree B_i consists of a root with i children $B_0, \ldots, B_{i-1}$.



It is easy to prove by induction that $|B_i| = 2^i$.

If data elements are arranged as vertices in a tree, that tree is said to be *heap-ordered* if the minimum value among all vertices of any subtree is found at the root of that subtree. A *binomial heap* is a collection of heap-ordered binomial trees with a pointer **min** to the tree whose root has minimum value. We will assume that all children of any vertex are arranged in a circular doubly-linked list, so that we can link and unlink subtrees in constant time.

Definition 8.1 The *rank* of an element x, denoted rank (x), is the number of children of x. For instance, rank (root of B_i) $= i$. The *rank* of a tree is the rank of its root. □

A basic operation on binomial trees is *linking*. Given two B_i's, we can combine them into a B_{i+1} by making the root of one B_i a child of the root of the other. We always make the B_i with the larger root value the child so as to preserve heap order. We never link two trees of different rank.

8.1 Operations on Binomial Heaps

In the "eager meld" version, the trees of the binomial heap are accessed through an array of pointers, where the i^{th} pointer either points to a B_i or is **nil**. The operation **meld**(h, h'), which creates a new heap by combining h and h', is reminiscent of binary addition. We start with $i = 0$. If either h or h' has a B_0 and the other does not, we let this B_0 be the B_0 of **meld**(h, h'). If neither h nor h' have a B_0, then neither will **meld**(h, h'). If both h and h' have a B_0, then **meld**(h, h') will not; but the two B_0's are linked to form a

B_1, which is treated like a carry. We then move on to the B_1's. At stage i, we may have 0, 1, or 2 B_i's from h and h', plus a possible B_i carried from the previous stage. If there are at least two B_i's, then two of them are linked to give a B_{i+1} which is carried to the next stage; the remaining B_i, if it exists, becomes the B_i of **meld**(h, h'). The entire operation takes $O(\log n)$ time, because the size of the largest tree is exponential in the largest rank. We will modify the algorithm below to obtain a "lazy meld" version, which will take constant amortized time.

The operation **insert**(i, h) is just **meld**$(h,$ **makeheap**$(i))$.

For the operation **deletemin**(h), we examine the **min** pointer to x, the root of some B_k. Removing x creates new trees $B_0, \ldots, B_{k-1}$, the children of x, which are formed into a new heap h'. The tree B_k is removed from the old heap h. Now h and h' are melded to form a new heap. We also scan the new heap to determine the new **min** pointer. All this requires $O(\log n)$ time.

8.2 Amortization

The $O(\log n)$ bound on **meld** and **deletemin** is believable, but how on earth can we do **insert** operations in constant time? Any particular **insert** operation can take as much as $O(\log n)$ time because of the links and carries that must be done. However, intuition tells us that in order for a particular **insert** operation to take a long time, there must be a lot of trees already in the heap that are causing all these carries. We must have spent a lot of time in the past to create all these trees. *We will therefore charge the cost of performing these links and carries to the past operations that created these trees.* To the operations in the past that created the trees, this will appear as a constant extra overhead.

This type of analysis is known as *amortized analysis*, since the cost of a sequence of operations is spread over the entire sequence. Although the cost of any particular operation may be high, over the long run it averages out so that the cost per operation is low.

For our amortized analysis of binomial heaps, we will set up a savings account for each tree in the heap. When a tree is created, we will charge an extra credit to the instruction that created it and deposit that credit to the account of the tree for later use. (Another approach is to use a *potential function*; see [100].) We will maintain the following *credit invariant*:

> Each tree in the heap has one credit in its account.

Each **insert** instruction creates one new singleton tree, so it gets charged one extra credit, and that credit is deposited to the account of the tree that was created. The amount of extra time charged to the **insert** instruction is $O(1)$. The same goes for **makeheap**. The **deletemin** instruction exposes up to $\log n$ new trees (the subtrees of the deleted root), so we charge an extra

$\log n$ credits to this instruction and deposit them to the accounts of these newly exposed trees. The total time charged to the **deletemin** instruction is still $O(\log n)$.

We use these saved credits to pay for linking later on. When we link a tree into another tree, we pay for that operation with the credit associated with the root of the subordinate tree. The **insert** operation might cause a cascade of carries, but the time to perform all these carries is already paid for. We end up with a credit still on deposit for every exposed tree and only $O(1)$ time charged to the **insert** operation itself.

8.3 Lazy Melds

We can also perform **meld** operations in constant time with a slight modification of the data structure. Rather than using an array of pointers to trees, we use a doubly linked circular list. To **meld** two heaps, we just concatenate the two lists into one and update the **min** pointer, certainly an $O(1)$ operation. Then **insert**(h, i) is just **meld**$(h,$ **makeheap**$(i))$.

The problem now is that unlike before, we may have several trees of the same rank. This will not bother us until we need to do a **deletemin**. Since in a **deletemin** we will need $O(\log n)$ time anyway to find the minimum among the deleted vertex's children, we will take this opportunity to clean up the heap so that there will again be at most one tree of each rank. We create an array of empty pointers and go through the list of trees, inserting them one by one into the list, linking and carrying if necessary so as to have at most one tree of each rank. In the process, we search for the minimum.

We perform a constant amount of work for each tree in the list in addition to the linking. Thus if we start with m trees and do k links, then we spend $O(m + k)$ time in all. To pay for this, we have k saved credits from the links, plus an extra $\log n$ credits we can charge to the **deletemin** operation itself, so we will be in good shape provided $m + k$ is $O(k + \log n)$. But each link decreases the number of trees by one, so we end up with $m - k$ trees, and these trees all have distinct ranks, so there are at most $\log n$ of them; thus

$$\begin{aligned} m + k &= 2k + (m - k) \\ &\leq 2k + \log n \\ &= O(k + \log n) \ . \end{aligned}$$

Lecture 9 Fibonacci Heaps

Fibonacci heaps were developed by Fredman and Tarjan in 1984 [35] as a generalization of binomial heaps. The main intent was to improve Dijkstra's single-source shortest path algorithm to $O(m + n \log n)$, but they have many other applications as well. In addition to the binomial heap operations, Fibonacci heaps admit two additional operations:

decrement(h, i, Δ)	decrease the value of i by Δ
delete(h, i)	remove i from heap h

These operations assume that a pointer to the element i in the heap h is given.

In this lecture we describe how to modify binomial heaps to admit **delete** and **decrement**. The resulting data structure is called a *Fibonacci heap*. The trees in Fibonacci heaps are no longer binomial trees, because we will be cutting subtrees out of them in a controlled way. We will still be doing links and melds as in binomial heaps. The *rank* of a tree is still defined in the same way, namely the number of children of the root, and as with binomial heaps we only link two trees if they have the same rank.

To perform a **delete**(i), we might cut out the subtree rooted at i, remove i, and **meld** in its newly freed subtrees. We must also search these newly freed subtrees for the minimum root value; this requires $O(\log n)$ time. In **decrement**(i, Δ), we decrement the value of i by Δ. The new value of i might violate the heap order, since it might now be less than the value of i's parent. If so, we might simply cut out the subtree rooted at i and **meld** it into the heap.

The problem here is that the $O(\log n)$ time bound on **deletemin** described in the last lecture was highly dependent on the fact that the size of B_k is exponential in k, *i.e.* the trees are bushy. With **delete** and **decrement** as described above, cutting out a lot of subtrees might make the tree scraggly, so that the analysis is no longer valid.

9.1 Cascading Cuts

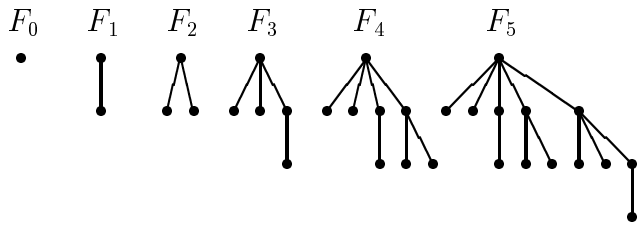
The way around this problem is to limit the number of cuts among the children of any vertex to two. Although the trees will no longer be binomial trees, they will still be bushy in that their size will be exponential in their rank.

For this analysis, we will set up a savings account for every vertex. The first time a child is cut from vertex p, charge to the operation that caused the cut two extra credits and deposit them to the account of p. Not only does this give two extra credits to use later, it also marks p as having had one child cut already. When a second child is cut from p, cut p from its parent p' and **meld** p into the heap, paying for it with one of the extra credits that was deposited to the account of p when its first child was cut. The other credit is left in the account of p in order to maintain the invariant that each tree in the heap have a credit on deposit. If p was the second child cut from its parent p', then p' is cut from its parent; again, this is already paid for by the operation that cut the first child of p'. These cuts can continue arbitrarily far up the tree; this is called *cascading cuts*. However, all these cascading cuts are already paid for. Thus **decrement** is $O(1)$, and **delete** will still be $O(\log n)$ provided our precautions have guaranteed that the sizes of trees are still exponential in their rank.

Theorem 9.1 *The size of a tree with root r in a Fibonacci heap is exponential in* rank (r).

Proof. Fix a point in time. Let x be any vertex and let $y_1, \ldots, y_m$ be the children of x at that point, arranged in the order in which they were linked into x. We show that rank (y_i) is at least $i-2$. At the time that y_i was linked into x, x had at least the $i-1$ children $y_1, \ldots, y_{i-1}$ (it may have had more that have since been cut). Since only trees of equal rank are linked, y_i also had at least $i-1$ children at that time. Since then, at most one child of y_i has been cut, or y_i itself would have been cut. Therefore the rank of y_i is at least $i-2$.

We have shown that the i^{th} child of any vertex has rank at least $i-2$. Let F_n be the smallest possible tree of rank n satisfying this property. The first few F_n are illustrated below.



Observe that F_0, F_1, F_2, F_3, F_4, F_5, ..., are of size $1, 2, 3, 5, 8, 13 \ldots$, respectively. This sequence of numbers is called the *Fibonacci sequence*, in which each number is obtained by adding the previous two. It therefore suffices to show that the n^{th} Fibonacci number $f_n = |F_n|$ is exponential in n.

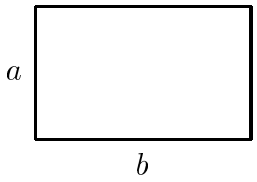
Specifically, we show that $f_n \geq \varphi^n$, where $\varphi = \frac{1+\sqrt{5}}{2} \approx 1.618 \ldots$, the positive root of the quadratic $x^2 - x - 1$. The proof proceeds by induction on n.

For the basis, $f_0 = 1 \geq \varphi^0$ and $f_1 = 2 \geq \varphi^1$. Now assume that $f_n \geq \varphi^n$ and $f_{n+1} \geq \varphi^{n+1}$. Then

$$\begin{aligned} f_{n+2} &= f_{n+1} + f_n \\ &\geq \varphi^{n+1} + \varphi^n \\ &= \varphi^n(\varphi + 1) \\ &= \varphi^n \cdot \varphi^2 \text{ since } \varphi^2 = \varphi + 1 \\ &= \varphi^{n+2} \ . \end{aligned}$$

□

The real number φ is often called the *golden ratio*. It was considered the most perfect proportion for a rectangle by the ancient Greeks because it makes the ratio of the length of the longer side to the length of the shorter side equal to the ratio of the sum of the lengths to the length of the longer side.



$$\varphi = \frac{b}{a} = \frac{a+b}{b}$$

(The picture is actually 81pt × 50pt, giving a ratio of 1.62. Apologies to the ancient Greeks.)

The golden ratio φ is more closely related to the Fibonacci sequence than is apparent from the proof of Theorem 9.1. Consider the linear system

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} f_n \\ f_{n+1} \end{bmatrix} = \begin{bmatrix} f_{n+1} \\ f_{n+2} \end{bmatrix} \tag{14}$$

which generates the Fibonacci sequence:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} f_0 \\ f_1 \end{bmatrix} = \begin{bmatrix} f_n \\ f_{n+1} \end{bmatrix} .$$

Let F denote the 2×2 matrix in (14). The eigenvalues of F are φ and $\varphi' = \frac{1-\sqrt{5}}{2}$, the two roots of its characteristic polynomial

$$\det(xI - F) = x^2 - x - 1 .$$

The eigenvectors associated with φ and φ' are

$$\begin{bmatrix} 1 \\ \varphi \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} 1 \\ \varphi' \end{bmatrix} ,$$

respectively, of which the former is dominant. Successive applications of a matrix to a vector with a nonzero component in the direction of a dominant eigenvector, suitably scaled, will generate a sequence of vectors converging to that dominant eigenvector. Thus

$$\left(\varphi^{-1} \cdot \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}\right)^n \begin{bmatrix} f_0 \\ f_1 \end{bmatrix} = \varphi^{-n} \cdot \begin{bmatrix} f_n \\ f_{n+1} \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ \varphi \end{bmatrix}$$

as $n \rightarrow \infty$; in other words, the ratio of successive Fibonacci numbers tends to φ.

9.2 Fibonacci Heaps and Dijkstra's Algorithm

We can use Fibonacci heaps to implement Dijkstra's single-source shortest-path algorithm (Algorithm 5.1) in $O(m + n \log n)$ time. We store the elements of $V - X$ in a Fibonacci heap. The value of the element v is $D(v)$. The initialization uses the **makeheap** operation and takes linear time. We use the **decrement** operation to implement the statement

$$D(v) := \min(D(v), D(u) + \ell(u, v)) .$$

This requires constant time for each edge, or $O(m)$ time in all. We use the **deletemin** operation to remove a vertex from the set of unreached vertices. This takes $O(\log n)$ time for each deletion, or $O(n \log n)$ time in all.

Another application of Fibonacci heaps is in Prim's algorithm for minimum spanning trees. We leave this application as an exercise (Homework 4, Exercise 1).