W4231: Analysis of Algorithms

10/7/1999

- Fibonacci Heaps

From Binomial Heaps to Fibonacci Heaps

We first prove that in Binomial Heaps `insert` and `find-min` take amortized $O(1)$ time, while still having `insert`, `union`, `delete` and `decrease-key` running in $O(\log n)$ amortized (and worst-case) time.

Then we show how to change `insert`, `union`, and `decrease-key` so that they also run in $O(1)$ amortized time (but they do not maintain any more a Binomial Heap structure), while a modified `delete` will run in $O(\log n)$ amortized time.

Recall Counter

We said we can implement an integer counter using a linked list of bits.

The list goes from the least significant bit to the most significant bit.

We proved each `inc` operations takes $O(1)$ amortized time with a bit of handwaving. Let's see a potential argument.

Recall the complexity of `inc`(n) is big-Oh of the number of consecutive ones that we find in the binary representation of n, starting from the least significant bit.

Potential

The content of the data structure is just an integer n (represented with this reverse list of bits).

We define the potential of n to be the *number of ones in the binary representation of* n.

We want to make sure that for every call to `inc`, the following expression is $O(1)$:

`actual running time` + `new potential - old potential`.

Analysis

Say that n has a binary representation that ends with k consecutive ones (k could be zero).

Then the actual running time of `inc`(n) is $k+1$.

`inc`(n) will turn k 1s into 0s, and then a 1 (or a new element) into a 1. Hence `new potential - old potential` $= 1-k$.

The amortized running time is 2.

Binomial Heaps

Define the potential of a Binomial Heap to be the number of trees.

Then `insert` takes constant amortized time, by a similar argument.

`find` can be implemented in $O(1)$ worst case time by maintaining a pointer to the minimum (`insert` and `delete` have to update the pointer).

`union` takes $O(\log n)$ actual time, and does not increase the potential.

`delete-min` takes $O(\log n)$ actual time, even if we want to update the pointer to the minimum, and the potential increases at most by $\log n$. Amortized time is also $O(\log n)$.

`decrease-key` takes $O(\log n)$ actual time, and does not change the potential.

All Operations Except Delete

It is easy to implement all operations except `delete` in $O(1)$ worst-case time.

Put all the elements in a circular doubly-linked list, maintain a pointer to the minimum.

Implement `insert` by putting an element wherever (say, after the minimum) and updating the minimum if needed.

Implement `union` by opening and joining the two lists, and updating the minimum.

With (Almost) Binomial Heaps

We show we can implement all the operations except `delete` in $O(1)$ amortized time by keeping our data structure some sort of a Binomial Heap.

Then we will be able to add `delete` in $O(\log n)$ time.

General Heap-ordered Tree

A Fibonacci heap is made of Fibonacci trees. The definition of Fibonacci tree is bizarre (and driven by the analysis). We'll give it later.

Each node in the tree has pointer to father and first child. Siblings are connected by a doubly-linked circular list.

In each node we also store the degree.

For every Fibonacci tree with n nodes, the maximum degree is $D(n) = O(\log n)$. We'll prove it later.

Representation of the Heap

The Fibonacci heap is a doubly linked list of roots.

We hold a pointer $H.pmin$ to the root that contains the global minimum.

We also have a counter $H.n$ of the number of nodes in the data structure.

Operations

- $\texttt{insert}(u, H)$ add the new element as the root of a singleton tree. Update pointer $H.pmin$, if needed.
- $H = \texttt{union}(H_1, H_2)$ splice together the two circular lists. Updated the minimum.
- $v = \texttt{find} - \texttt{min}(H)$ read the element pointed by $H.pmin$.

Worst case and average case $O(1)$. Note `insert` increases the potential by 1.

Deletemin

As for Binomial Heaps: take away the root with the minimum, make each child into a new root.

This takes time $O($ degree of the root with the minimum $)$.

Who is the new minimum?

Now we pay for our laziness: we join together all pairs of roots having the same degree. In the process, we look at all the roots and find the new minimum.

Time to clean after your mess

We scan all the list of roots.

Each time we find a root with the same degree of one we have seen before, we join them together.

How do we recall which roots of which degree we have already seen?

Implementation

We know that there are n nodes in the heap. We know the maximum degree of any vertex is $D(n) = O(\log n)$.

We declare a vector $A[1, \ldots, D(n)]$ of pointers to roots. Each one is initially NIL.

When we scan a root of r degree d, we put a pointer to it in $A[d]$.

If $A[d]$ already contains a pointer to some other root r', then we do a Tree-join of r and r', and put the result in $A[d+1]$.

If $A[d+1]$ is already occupied, we do a join . . .

Running Time

At the beginning of `delete-min` we have to process $\leq D(n)$ children of the node to be deleted.

Then we process all the t trees in the Heap.

For each of them, we may or may not do some Tree-join.

Note that each Tree-join reduces the number of trees in the heap. And takes $O(1)$ time.

So total scan time is $O(t)$, and total Tree-join time is also $O(t)$.

The old potential was t. The new potential is $\leq D(n)$, because now all degrees are different.

The actual time is $O(D(n) + t)$. The difference in potential is $D(n) - t$.

Assuming one unit of potential can pay for $O(1)$ operations,

the amortized time is $O(D(n))$

Implementing `decrease-key`

We cannot implement `decrease-key` in the standard way (keep swapping node with father, until a safe place is found) because the particular implementation of `insert` and `delete` does not guarantee that the trees have bounded depth.

In particular, one can have trees of linear depth in the structure (see Homework 3).

Idea: make a new root out of the node whose key we want to decrease.

Problem: this way we can have very "fat" trees, whose root has a very big degree.

Recall: `delete-min` runs in time proportional to the degree of the root containing the minimum.

More Complicated Potential

Now we unveil another feature of Fibonacci Heap representation.

Each node has also a Boolean value called `mark`, that is initially set to FALSE.

The potential function is

number of trees in the heap + 2 * number of marked nodes

Meaning of `mark`

`mark` is set to TRUE the first time a node loses a child due to a `decrease-key`.

When we do `decrease-key` on node v, we make v into a new root, and we look at the father u of v.

If u.mark==FALSE, then we set u.mark=TRUE.

If u.mark was already TRUE, then we also make u as a new root, and unmark it. We then look at his father, etc.

This way, the second time a node loses a child due to a `decrease-key`, it is made into a new root.

Amortized Analysis

The addition of `2 * number of marked vertices` to the potential does not change analysis of `insert`, `find-min`, `union`, `delete`.

Let's see `decrease-key`(v, k). Each time we make a node into a root this takes constant time.

Say we do so for c ancestors of the node v.

The total time is proportional to $c + 1$.

The increase in potential due to the new roots is also $c + 1$.

But c nodes that were marked now are unmarked (and possibly an unmarked node is now marked).

The decrease in potential due to the unmarking is at least $2(c - 1)$.

Overall amortized time is 4.

Degree

We were left with the statement that amortized `delete` time is $O(D(n))$, where $D(n)$ is an upper bound on the degrees of nodes in a tree of size n.

We now prove $D(n) = O(\log n)$ and we are done.

Fibonacci Tree

A node is a Fibonacci tree of rank 0.

A Fibonacci tree of rank k is any tree that has a root of degree k, and its children are Fibonacci trees whose ranks, in ascending order, are at least $0, 0, 1, 2, \ldots, k-2$.

First Result

The operations described so far, maintain the invariant that each tree in the heap is a Fibonacci tree.

There are five ways of changing trees.

`insert` creates a singleton tree, which is a Fibonacci tree of rank 0.

`delete-min` creates trees that are subtrees of an older tree. The definition of a Fibonacci tree ensures that subtrees of a Fibonacci tree are Fibonacci trees.

also `delete-min` has calls to Tree-join. In a Tree-join operation

between two roots r and r' of degree k, the new root r gets degree $k+1$, and the new child r' has degree k (we would have been happy enough to have $k-1$).

with `decrease-key`, r' may later lose a child, and its degree would become $k-1$, which is still good enough. It cannot lose another child without being away from r.

when `decrease-key` takes away a child from a node, there is still no way to violate the property.

Degree of Fibonacci Trees

Now we come to the Fibonacci part.

The Fibonacci numbers are the sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

Recursive definition: $F_0 = 0$, $F_1 = 1$, $F_k = F_{k-1} + F_{k-2}$.

Connection

- A Fibonacci tree of rank $k \geq 2$ has at least F_{k+2} vertices.
- $F_{k+2} \geq \left(\frac{1+\sqrt{5}}{2}\right)^k > (1.6)^k$

Then if a Fibonacci tree has n nodes and root of degree k, $n \geq (1.6)^k$, and so $k \leq (\log n)/(\log 1.6) = O(\log n)$

A Fib. Tree of rank k has $\geq F_{k+2}$ vertices

First of all

$F_{k+2} = 1 + F_0 + F_1 + \ldots + F_k$ (proof: induction)

Then, call s_k the minimum number of vertices in a Fibonacci tree with root of degree k. Then $s_0 = 1$, $s_1 = 2$,

$s_k = 1 + s_0 + s_1 + s_2 + \ldots + s_{k-2}$ (by definition)

Then $s_k = F_{k+2}$ (again by induction)

$F_{k+2} \geq (1.6)^k$

Call $\phi = (1+\sqrt{5})/2 > 1.6$. Note that $1 + \phi = \phi^2$.

Prove by induction that $F_{k+2} > \phi^k$.

$F_2 = 1 = \phi^0$.

$F_3 = 2 > \phi^1$.

$$\begin{aligned} F_{k+2} &= F_{k+1} + F_k \text{ By definition} \\ &\geq \phi^{k-1} + \phi^{k-2} \text{ By inductive hypothesis} \\ &= \phi^{k-2}(1+\phi) \\ &= \phi^k \end{aligned}$$

Wrap it up

Having the number of trees in the potential allows us to do efficient, naive, `insert` and `union`. `delete-min` has the time to do major restructuring by borrowing time from the potential.

A bottleneck of `delete-min` is that it requires low degree in the trees.

`decrease-key` can be implemented so that it is efficient and does not mess up the low degree property.

Counting marked vertices in the potential makes sure that we have occasionally the time to restructure for low degree.