6.854 Problem Set 1

Nada Amin
namin@mit.edu

September 12, 2007

Collaborators

- Thomas Belulovich (thobel@mit.edu)
- Thomas Mildorf (tmildorf@mit.edu)

Problem 1

For each n, I exhibit a sequence of Fibonacci heap operations on n items that produce a heap ordered tree of depth $\geq n - 3 = \Omega(n)$.

For $n \leq 3$, the solution is trivial as the required lower-bound on the depth is ≤ 0.

For $n \geq 4$, below is a sequence of operations on n items with keys from 1 to n (each item will be noted by its key) that produce a heap-ordered tree of depth $n - 3$, specifically either the tree $3 \to 4$ if $n = 4$ or, if $n > 4$, the tree $2 \to \ldots \to n-3 \to n-1 \to n$, i.e. the tree with item 2 as a root and with each greater item except item $n - 2$ present as the only child of the previous smaller item present.

```
MAKE-TREE(n)
INSERT(n)
INSERT(n − 1)
INSERT(n − 2)
DELETE-MIN()
for k ← 4 to n − 1
        do INSERT(n − 2)
           INSERT(n − k + 1)
           INSERT(n − k)
           DELETE-MIN()
           ▷ to DELETE(n − 2):
           DECREASE-KEY(n − 2, −∞)
           DELETE-MIN()
```

Problem 2

I show that modifying Fibonacci heaps so that a node is cut only after losing k children improves the amortized cost of DECREASE-KEY (to a better constant) at the cost of a worse cost for DELETE-MIN (by a constant factor).

DECREASE-KEY

I analyze the amortized cost of DECREASE-KEY, when a node is cut only afer losing k children (i.e. having k marks). I choose the potential function:

$$\Phi = \frac{2}{k-1}(\text{number of marks}) + (\text{number of roots})$$

so that the cost of a cascading cut is 0:

	1	(for cutting the node)
$+$	1	(for adding a root)
$-$	$(k-1)2/(k-1)$	(for removing the $k-1$ marks on this node)
	0	(total)

Hence, the amortized cost of decrease-key is $2 + 2/(k-1)$:

	1	(for cutting the node)
$+$	1	(for adding a root)
$+$	$2/(k-1)$	(for adding a mark)
$+$	0	(for cascading cuts)
	$2 + 2/(k-1)$	(total)

As $k > 2$, the constant cost of DECREASE-KEY decreases.

DELETE-MIN

For DELETE-MIN, I show that the trees are exponential in degree, so that adding the children of the min as roots result in $O(\log(n))$ new roots.

By the union-by-rank procedure, the i^{th} added child will have degree $\geq i-k$. Indeed, by virtue of being the i^{th} child added, it must have been in the $(i-1)^{\text{th}}$ bucket, so it must have started with $i-1$ children. Since it losts at most $k-1$ children, it still has at least $i-k$ children. Now, let

$$S_k = \text{the number of descendants of a tree with } k \text{ children}$$

$$S_0 = 1$$
$$S_1 = 2$$
$$S_n \geq \sum_{i=0}^{n-k} S_i$$
$$S_n \geq S_{n-1} + S_{n-k}$$
$$S_n = \Omega(C^k)$$

Hence, DELETE-MIN will add $O(\log_C(n))$ roots. When $k = 2$, $C = \Phi$ (the golden number). Clearly, for $k > 2$, $C < \Phi$, so $\log_C(n) < \log_\Phi(n)$. Therefore, as $k > 2$, the performance of DELETE-MIN will be worse by a constant factor (precisely $\log_C(\Phi)$).

Problem 3

Part (a)

I show how to use a priority queue P that performs INSERT, DELETE-MIN and MERGE in $O(\log(n))$ time and MAKE-HEAP in $O(n)$ time to construct a priority queue Q that performs INSERT in $O(1)$ amortized time while still performing DELETE-MIN and MERGE in $O(\log(n))$ amortized time.

For my new priority queue Q, I maintain two structures, a list m and the priority queue p. On INSERT, I simply add the new element to the list m. On DELETE-MIN and MERGE, I make a priority queue out of the list m and merge it with the priority queue p, before calling the DELETE-MIN or MERGE procedure of p.

Q:MAKE-HEAP(l)

$m \leftarrow ()$
$p \leftarrow$ P:MAKE-HEAP(l)

Q:INSERT(i)

m.ADD(i)

Q:DELETE-MIN(i)

$p2 \leftarrow$ P:MAKE-HEAP(m)
$m \leftarrow ()$
p.MERGE($p2$)
return p.DELETE-MIN()

Q:MERGE($p2$)

$p3 \leftarrow$ P:MAKE-HEAP(m)
$m \leftarrow ()$
p.MERGE($p3$)
p.MERGE($p2$)

Amortized Analysis

Let $\Phi(Q)$ = number of elements in the list m. Then:

The amortized cost of INSERT is $O(1)$:

real cost :	1
$\Delta\Phi$:	1
amortized cost:	2

The amortized cost of DELETE-MIN is $O(\log(n))$:

real cost :	$\Phi + O(\log(n))$
$\Delta\Phi$:	$-\Phi$
amortized cost:	$O(\log(n))$

Similarly, the amortized cost of MERGE is $O(\log(n))$.

Part (b)

I show how even binary heaps can be modified to support INSERT in $O(1)$ amortized time while maintaining an $O(\log(n))$ time bound for DELETE-MIN.

As in part (a), on INSERT, I add the new element to a list m, which I make into a binary heap during DELETE-MIN. However, instead of merging, I add the new heap in a heap of heaps, a "super" heap.

Therefore, for my new priority queue Q, I maintain two structures: a list m and a heap of heaps s. The super-heap is keyed by the smallest element in each inner heap. On DELETE-MIN, I make a new inner heap out of the list m and add it to the super-heap. I find the min inner heap in the super-heap, perform DELETE-MIN on the min inner heap, and then MIN-HEAPIFY the super-heap to maintain the (super-)heap order (or delete the inner heap from the super-heap if it's empty).

Q:MAKE-HEAP(l)

$m \leftarrow ()$

$s \leftarrow$ P:MAKE-HEAP(P:MAKE-HEAP(l))

Q:INSERT(i)

m.ADD(i)

Q:DELETE-MIN(i)

```
s.INSERT(P:MAKE-HEAP(m))
m ← ()
h ← s.MIN()
x ← h.DELETE-MIN()
if h.EMPTY()
   then s.DELETE(h)
   else s.MIN-HEAPIFY()
return x
```

Amortized Analysis

The amortized analysis in similar to part (a). In DELETE-MIN, deleting the min from the min inner heap and maintaining the heap order of the super-heap each takes $O(\log(n)$. Indeed, an inner heap has at most n elements while the super-heap has at most n inner heaps (with 1 element each). Therefore, as in part (a), the amortized cost of DELETE-MIN is $O(\log(n))$:

real cost :	$\Phi + O(\log(n))$
$\Delta\Phi$:	$-\Phi$
amortized cost:	$O(\log(n))$

Problem 4

I show how to use the techniques of persistent data structures to preprocess a tree in $O(n \log(n))$ time so as to allow LCA queries to be answered in $O(\log(n))$ time.

The time axis consists of the nodes of the tree in postorder. For each node, I maintain a timestamped pointer to its parent, a pointer to a binary search tree of timestamped names and an ephemeral rank for the union-by-rank heuristic.

Initially, at time $t = 0$, each node is isolated, having a null parent pointer and an empty tree of names and a rank of 0.

I support 3 operations:

UNION-WITH-NAME(a, b) increments time, UNION node a with node b and set the name of the representative to b.

FIND-AT-TIME(a, t) finds the representative of node a at time t.

NAME-AT-TIME(r, t) finds the name associated with the representative r at time t.

```
UNION-WITH-NAME(a, b)
t ← t + 1
r_a ← FIND-AT-TIME(a, t)
r_b ← FIND-AT-TIME(b, t)
▷ apply the union-by-rank heuristic
if rank(r_a) = rank(r_b)
    then set parent pointer of r_b to r_a and timestamp it
         r ← r_a
         rank(r_a) ← rank(r_a) + 1
elseif rank(r_a) < rank(r_b)
    then set parent pointer of r_a to r_b and timestamp it
         r ← r_b
else ▷ rank(r_b) < rank(r_a)
         set parent pointer of r_b to r_a and timestamp it
         r ← r_a
add the name b at time t to the binary search tree of names of r
```

```
FIND-AT-TIME(a, t)
if parent(a) is null or timestamp-parent(a) > t
    then return a
    else return FIND-AT-TIME(parent(a), t)
```

```
NAME-AT-TIME(r, t)
return the name at time t in the binary search tree of names of r
```

During pre-processing, I simply call UNION-WITH-NAME(node, parent) for each node in postorder.

During a query for the pair (a, b), I proceed as follows. Assuming, without loss of generality that node a occurs before node b in postorder, I let t be the time just before b was processed. The query call is then simply NAME-AT-TIME(FIND-AT-TIME$(a, t), t)$.

Analysis

Thanks to the union-by-rank heuristic, I assure the union-find has only trees of depth $O(\log(n))$. Therefore, FIND-AT-TIME is $O(\log(n))$. UNION-WITH-NAME is therefore $O(\log(n) + \log(t))$ and NAME-AT-TIME is $O(\log(t))$. Since $t \leq n$, all operations are $O(\log(n))$. Therefore pre-processing is $O(n \log(n))$ and querying is $O(\log(n))$.

Problem 5

I show that the expected behavior of markless Fibonacci heaps (where, each time I do a cut, I flip an unbiased coint to decide whether to cascade the cut to the parent) is like that of the standard ones. Indeed, the expected number of children cut before a node is cut, E_{cut}, is 2:

$$\begin{aligned}
E_{\text{cut}} &= \frac{1}{2} && +2\cdot\frac{1}{4} && +3\cdot\frac{1}{8} && +\ldots \\
&= (\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots) && +(\frac{1}{4}+\frac{1}{8}+\ldots) && +(\frac{1}{8}+\ldots) && +\ldots \\
&= 1 && +\frac{1}{2} && +\frac{1}{4} && +\ldots \\
&= \sum_{i=0}^{\infty}\frac{1}{2^i} \\
&= 2
\end{aligned}$$

It is possible to bias the coint so that a cascade is more than 50% likely to achieve the effect of cascading after (say) one and a half children are cut. Let p be the probability of cascading a cut, i.e. the bias of the coin. Then:

$$\begin{aligned}
E_{\text{cut}} &= p + 2\cdot p^2 + 3p^3 + \ldots \\
&= \sum_{i=0}^{\infty} p^i \\
&= \frac{1}{p}
\end{aligned}$$

So in order to have $E_{\text{cut}} = 1.5 = \frac{3}{2}$, we need a bias of $p = \frac{2}{3}$.

This can be used to improve the expected amortized time for DELETE-MIN at the cost of increasing the expected amortized time for DECREASE-KEY. The analysis is similar to Problem 2, with $k = 1.5 < 2$: DECREASE-KEY is more costly because the expected number of cascading cuts is higher, DELETE-MIN is less costly because the expected exponent in the exponential-in-degree trees is greater.