Problem Set 3

Due: Wednesday, September 28, 2005.

Notice that one problem is marked **noncollaborative.** As you might expect, this problem should be done without any collaboration.

NONCOLLABORATIVE Problem 1. In class we saw how to use a van Emde Boas priority queue to get $O(\log\log u)$ time per queue operation (insert, delete-min, decrease-key) when the range of values is $\{1, 2, \ldots, u\}$. Show that for the single-source shortest paths problem on a graph with n nodes and range of edge lengths $\{1, 2, \ldots, C\}$, we can obtain $O(\log\log C)$ time per queue operation, even though the range of values in the queue is $\{1, 2, \ldots, nC\}$,

Problem 2. Augment the van Emde Boas priority queue to support the following operations on integers in the range $\{0, 1, 2, \ldots, u-1\}$ in $O(\log\log u)$ worst-case time each and $O(u)$ space total:

Find (x, Q): Report whether the element x is stored in the structure.

Predecessor (x, Q): Return x's predecessor, the element of largest value less than x, or null if x is the minimum element.

Successor (x, Q): Return x's successor, the element of smallest value greater than x, or null if x is the maximum element.

Problem 3. In class we considered building a depth-k, base-Δ (implicit) trie over integers in the range from 1 to C (where $\Delta = C^{1/k}$) that supported insert in time $O(k)$ and delete-min in time $O(\Delta)$. By choosing k and Δ appropriately we found shortest paths in $O(m+n\log C)$ time. We now improve this bound. Consider modifying the data structure and delete-min operation, where we scan forward through a trie node and reach a new a bucket of items. If that bucket has more than t items in it, we expand it to multiple buckets in a node at the next trie level down as before. But if there are fewer than t items, we simply store them in a heap. During inserts or decrease-keys, new items may be added to the heap, and if the heap size grows beyond t, we expand it to a trie node of buckets as before.

(a) Let $I(t)$, $D(t)$, $X(t)$ denote the times to insert, decrease key, and extract min in a heap of size at most t. Prove that the amortized times for operations in the new data structure can be bounded by

- $O(k\Delta/t + I(t))$ for insert
- $O(D(t) + I(t))$ for decrease-key
- $O(X(t)$ for extract-min

Hint: When an item is inserted, give it $k\Delta/t$ units of potential energy. Each time the item gets pushed down into a new trie node, have it donate Δ/t of its potential energy to that node. Argue that this is a valid analysis, and that the potential energy at nodes is sufficient to pay for scanning trie nodes during an extract-min.

(b) Argue that using Fibonacci heaps and setting $k = \sqrt{\log C}$ and $t = 2^k$ gives a running time of $O(m + n\sqrt{\log C})$ for shortest paths.

Problem 4. Perfect hashing is nice, but does have the drawback that the perfect hash function has a lengthy description (since you have to describe the second-level hash function for each bucket). Consider the following alternative approach to producing a perfect hash function with a small description. Define *bi-bucket hashing*, or *bashing*, as follows. Given n items, allocate *two* arrays of size $n^{1.5}$. When inserting an item, map it to one bucket in *each* array, and place it in the emptier of the two buckets.

(a) Suppose a random function is used to map each item to buckets. Give a good upper bound on the expected number of collisions. **Hint:** What is the probability that the k^{th} inserted item collides with some previously inserted item?

(b) Argue that bashing can be implemented efficiently, with the same expected outcome, using the ideas from 2-universal hashing.

(c) Conclude an algorithm with linear expected time (ignoring array initialization) for identifying a perfect bash function for a set of n items. How large is the description of the resulting function?

OPTIONAL (d) Generalize the above approach to use less space by exploiting tri-bucket hashing (trashing), quad-bucket hashing (quashing), and so on.

OPTIONAL Problem 5. Our bucketing data structures (and in particular ven Emde Boas queues) use arrays, and we never worried about the time taken to initialize them. Devise a way to avoid initializing large arrays. More specifically, develop a data structure that holds n items according to an index $i \in \{1, \ldots n\}$ and supports the following operations in $O(1)$ time (worst case) per operation:

init Initializes the data structure to empty.

set(i, x) places item x at index i in the data structure.

get(i) returns the item stored in index i, or "empty" if nothing is there.

Your data structure should use $O(n)$ space and should work **regardless** of what garbage values are stored in that space at the beginning of the execution. **Hint:** use extra space to remember which entries of the array have been initialized.

OPTIONAL Problem 6. Can a van Emde Boas type data structure be combined with some ideas from Fibonacci heaps to support insert/decrease-key in $O(1)$ time and delete-min in $O(\log \log u)$ time?