Massachusetts Institute of Technology
6.854J/18.415J: Advanced Algorithms
David Karger

Handout 5
Wednesday, September 19, 2012

Problem Set 3

Due: Wednesday, September 26, 2012.

Collaboration policy: collaboration is *strongly encouraged*. However, remember that

1. You must write up your own solutions, independently.
2. You must record the name of every collaborator.
3. You must actually participate in solving all the problems. This is difficult in very large groups, so you should keep your collaboration groups limited to 3 people in a given week.
4. **No bibles. This includes solutions posted to problems in previous years.**

Problem 1. 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. But remember: the extra space also starts out with garbage entries!

Problem 2. Our Van Emde Boas construction gave a high-speed priority queue, but with a little more work it can turn into a high-speed "binary search tree" (which still supports find- and delete- min). Augment the van Emde Boas priority queue to track the maximum as well as the minimum of its elements, and use the augmentation 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 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 is $\{1, 2, \ldots, nC\}$.

Problem 4. In class we showed how to construct *perfect hash functions* that can be evaluated in constant time while producing no collisions at all. 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 (i.e., all function values are uniformly random and mutually independent) is used to map each item to buckets. Give a good upper bound on the expected number of collisions (i.e., the number of pairs of items that are placed in the same bucket). **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 hash 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. 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?

Problem 6. How long did you spend on this problem set? Please answer this question using the Google form that is sent to you via a separate email. This problem is mandatory, and thus counts towards your final grade. It is due by the 50%-deduction due date, which is the next Monday 2:30pm after the general due date.