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

Handout 3
Wednesday, September 14, 2016

Problem Set 2

Due: Wednesday, September 21, 2016.

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 hard in very large groups, so keep your collaboration groups limited to 3 or 4 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 $i \in \{1, \dots n\}$ and supports the following operations in $O(1)$ time (worst case; i.e. not amortized, not expected) per operation:

Init Initializes the data structure (assuming that the necessary space has been allocated) 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. You can assume that n fits in one machine word.

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. 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 $\{1, 2, \dots, u\}$. Show that for the single-source shortest paths problem on a graph with n nodes and range of edge lengths $\{1, 2, \dots, C\}$, we can obtain $O(\log \log C)$ time per queue operation, even though the range of values is $\{1, 2, \dots, nC\}$.

NONCOLLABORATIVE Problem 3. 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-min 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 $\{0, 1, 2, \dots, u-1\}$ 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.

In addition, argue that these augmentations preserve find-min and delete- min in $O(\log \log u)$ worst-case time.

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 in (a), 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 (the space required to encode the function) of the resulting bash function?

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

Problem 5. In class we saw how to build a perfect hash table for a set of keys given in advance. In this problem we will generalize this to allow insertion and deletion of keys in the table online. Recall that perfect hashing involves a top-level table hashing to multiple second-level tables using 2-universal hash functions. We initially consider only inserts. The basic approach is to *rebuild* any table (top level or second tier) when it violates the invariants on which perfect hashing relies.

(a) Consider a second-level table, that in the static case uses $f(s) = O(s^2)$ space to store s items. We gave a construction of such a table that succeeds with probability $1/2$. Suppose that we take the following approach to insertions: if inserting the s^{th} item violates perfection of the table, repeatedly build a table of size $f(2s)$ until you get a perfect one. Show that when s items are inserted (one at a time, with rebuilds as necessary) the expected total cost of all the rebuilds is $O(s)$. You may assume that array initialization takes constant time.

(b) Let s_i be the number of items in bucket i in the second level table. Make a similar modification for the top level: any time the table on n items has bucket sizes s_i violating the requirement that $\sum s_i^2 = O(n)$, we rebuild the top table and all the second level tables. Show the expected total time is $O(n)$ when n items are inserted.

(c) Now consider deletions. When an item is deleted, we can simply mark it as deleted without removing it (and if it reinserted, unmark the deletion). This makes deletion cheap. But if there are many deletions, the table might end up using much more space than it needs, which could invalidate the amortized bounds of the previous parts. To fix this, we can rebuild the hash table any time the number of "holes" (items marked deleted) exceeds half the total items in the table. Show that this cleanup work does not change (i.e., can be charged against) the amortized $O(1)$ cost of insertions.

Problem 6. Suppose you are an Internet router and you see a constant stream of IP addresses passing by. You'd like to keep track of which IP addresses occur frequently during a day.

(a) Suppose first that you have space for just a single counter and a single register (that can hold a single IP address). Propose a scheme such that, if there is an IP address that appears with frequency $> 1/2$ during the day, then at the end of the day, that IP address is stored in the register. (Otherwise, i.e., if there is no such "strict majority IP" address, there is no requirement on what the register stores at the end.)

Hint: Your scheme can be made very simple.

(b) Generalize the above solution to the case when you have k counters with k corresponding registers (again, each one of them can hold a single IP address). Develop a scheme with the property that, for any IP address that appears with frequency $> 1/(k+1)$ during the day, that address is stored in one of the registers at the end of the day.

OPTIONAL Problem 7. Prove (or make progress on) the *dynamic optimality conjecture:* over any given access sequence, splay trees take time proportional to the best possible (pointer based) data structure for the problem, even if that data structure is allowed to adjust itself during accesses (the adjustment time counts toward the overall cost, of course). Partial credit will be given for proving special cases of the conjecture (for particular kinds of access sequences).

Problem 8. How long did you spend on this problem set? Please answer this question using the Google form that is located on the course website. This problem is mandatory, and thus counts towards your final grade. It is due by the Monday 2:30pm after the pset due date.