6.897: Advanced Data Structures Spring 2005

Lecture 2 — February 3, 2005

Lecturer: Mihai Pătraşcu *Scribe: Catherine Miller*

1 Overview

In the last lecture we saw how to achieve static dictionaries with $O(1)$ worst-case query time, $O(n)$ worst-case space, and $O(n)$ expected construction time (the FKS scheme). We also discussed Cuckoo Hashing, which in addition achieves $O(1)$ expected updates.

In this lecture we discuss how various parts of this result can be improved: the query time, the space, and the use of randomness during construction.

2 Overview

2.1 Query Time

We begin with a brief discussion of query time. The optimal time for a query is 3 cell probes. Two of these probes must be adaptive, but they may be independent, i.e. two locations may be probed at the same time without either needing the other's return value.

2.2 Space

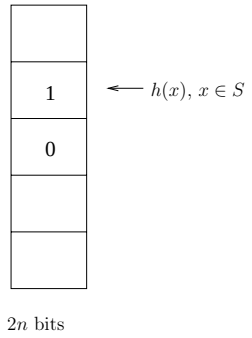
We first consider the membership problem, and how to improve our $O(n)$ space constraint. In a membership problem we are only concerned with deciding whether or not a given element x is a member of a set S. The optimal space needed to represent a subset of size n of U is $\lg \binom{U}{n}$. It is known how achieve space $\lg \binom{U}{n}$ plus a lower order term, with an $O(1)$ worst-case query time – a succinct membership structure. We will talk about succinct data structures later in the term.

Bloom Filters. Here we will attempt to solve the membership problem, but allow a small number of false positives. That is to say that we allow a small probability of mistaking an element which is not in the set for one in the set, but allow no margin of error for the inverse mistake.

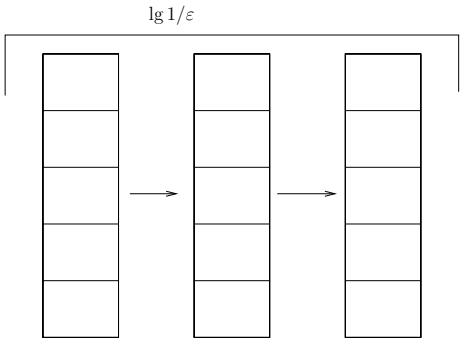
The optimal space is $n \lg \frac{1}{\varepsilon}$. This was achieved (plus a lower order term), together with a worst-case query time $O(1)$ [3]. We will instead discuss the classic Bloom filter. This Bloom filter, courtesy of Bloom, has a space $O(n \lg \frac{1}{\varepsilon})$ and query time $O(\lg \frac{1}{\varepsilon})$.

We start by having a table of size $2n$ bits which begins with a 0 in each space, and our choice of universal hashing function. Each $x \in S$ is hashed to some location in this table, in which location a 1 is written. Now any time we query this table with an x which is in S, we will definitely find a 1, and thus identify x as a member of S. When we query some x which is not is S, we will have a collision resulting in a false positive with some probability. The expected number if such collisions

is $n \cdot \frac{1}{2n} = \frac{1}{2}$. So using a standard Markov bound we can say that our probability of a false positive is $\leq 1/2$.



If one such filter gives a false positive with probability $\leq 1/2$, all we need do to get a better bound is chain many such filters together, and query each of them on the element x. If we chain $\lg 1/\varepsilon$ such filters, then we return a false positive with probability $(1/2)^{\lg 1/\varepsilon} = \varepsilon$, using $2n \lg 1/\varepsilon$ space and with a query time $O(\lg 1/\varepsilon)$.



Lossy Dictionaries. Of course, there is no particular reason to discriminate against false negatives while allowing such liberty to false positives. If we now allow false negatives with probability γ and false positives with probability ε, we have a lossy dictionary. By picking a random fraction of the elements equal to γ to discard before hashing, the space can be improved to $(1-\gamma)n \lg \frac{1}{\varepsilon}$. Interestingly, this is the best one can do [5].

Bloomier Filters. We now switch to improving the space for the dictionary problem. A search using a Bloomier filter has the following behavior:

- if $x \in S \rightarrow$ return associated data $[x]$
- if $x \notin S \rightarrow$ don't care

If the element is in our dictionary we want to return the data associated with it, but if it's not, we don't much care what the behavior is. This is useful in many interesting applications, because we are guaranteed that the queried elements are in the set.

In creating our data structures to support dictionary queries we will use a new type of data structure: a perfect hash function.

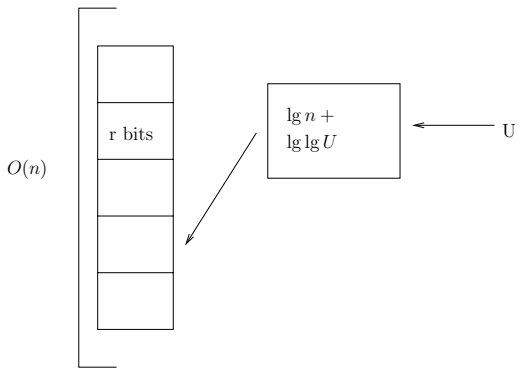
Perfect Hashing: Given a set S, construct a small data structure that can later be queried to assign an individual and unique ID to any element of S. The IDs should come from a set of size $O(n)$.

One example of this might be a business with one hundred employees who each need an ID number. One scheme might be to assign each employee an ID number based on his name. The trouble is that we need to store the whole list of names, and look up the name in that list each time. Instead, we might discover that assigning the first digit of the number based on eye color, the next on height, the next by weight gives a unique number for each person. A perfect hash function would then be just a description of this ID assigning algorithm.

Perfect hashing must not be confused with universal hashing: universal hashing attempts to emulate randomness (it is a study of randomness, independent of any set S), whereas in perfect hashing we are given a set and then asked to assign IDs; this is a data structure problem. One can also think of a perfect hash function as the minimum-length program (algorithm) that assigns unique IDs to elements of S, thus making it a complexity-theoretic program.

Optimal bounds are known both for static and dynamic perfect hashing. In the static case, the optimal space is $\Theta(n + \lg\lg U)$ bits. Note that this is much smaller than the space needed to represent S. This shows that small formulas for assigning unique IDs can always be found. In the dynamic case, an optimal bound of $\Theta(n \lg\lg \frac{U}{n})$ was recently obtained [2]. This may be considered an even more surprising result: the data structure can assign IDs to a changing set, even if it never knows what the set actually is (because it cannot remember it).

Using these results, it's easy to obtain a Bloomier filter. To support a dictionary search in which each piece of our data has r bits, we make one table of size $O(nr)$ bits for our data, and use a perfect hash function that tells us where in the array to forward queries for elements which are in our set S. Note that there is no guarantee for an element which is not in S. We obtain $\Theta(nr + \lg\lg U)$ bits of space in the static case, and $\Theta(n(r + \lg\lg \frac{U}{n}))$ in the dynamic case.



2.3 Determinism

As theorists, we want to be rid of anything that smacks of randomness – like an $O(n)$ expected construction time. Hagerup, Milterson, and Pagh [1] showed a completely deterministic dictionary

with $O(n \lg n)$ construction, $O(n)$ space and $O(1)$ queries. A remaining open problem is to get $O(n)$ deterministic construction while maintaining the same bounds on space and queries.

Dynamic bounds were obtained in [4]: poly($\lg n$) updates and poly($\lg \lg n$) queries. An unpublished manuscript of Sundar from 1993 claims a lower bound of $\Omega(\frac{\lg \lg U}{\lg \lg \lg U})$. This bound does not appear explicitly in Sundar's paper. This result must be viewed with some skepticism, pending further understanding of the proofs.

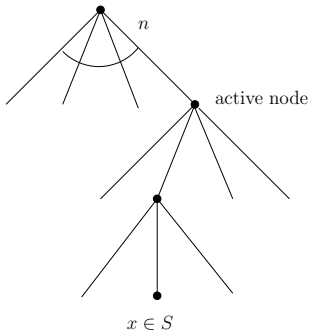
In the rest of the lecture, we concentrate of the static result of [1].

3 Static Deterministic Dictionaries

We will focus on the $O(n \lg n)$ construction. This is achieved by taking an arbitrary universe U and reducing it to a polynomial universe ($|U| = n^{O(1)}$) using error-correcting codes and bit tricks. Furthermore, this part of the algorithm is nonuniform (in the complexity theoretic sense). Then this universe is reduced to $U = [n^2]$, and a solution for such quadratic universes is given. We will ommit the first part, and instead discuss the details of the last two.

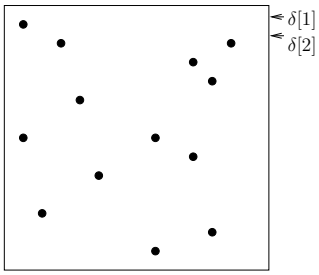
For each $x \in U$, write x in bits. Then take each $\lg n$ bit section and assign it a letter.

We use this to create a trie. This is a tree with a branching factor of n, into which we insert all of our x's, as root-to-leaf paths. We will consider a node to be "active" if it is the ancestor of an element $x \in S$. Thus there are $O(n)$ active nodes. To index into this trie, we start at the root, which has an ID of 0, and read the first letter of the query, perhaps "a". We then query a hash table for $(0, a)$, which retrieves the ID of the corresponding child, if the child is active. If the child corresponding to "a" is not active we know our query is not in the set and can stop. If the node is active, then we retrieve its ID, proceed to query the hash table with this ID and the second letter, and so on. The total query time is constant, because the height of the trie is constant.



Now that we've reduced our universe, we want to find a perfect hash function $h : U = [n]^2 \to [2^r]$, $r = \lg n + O(1)$. We can just expand the universe by a constant factor to $U = [2^r]^2$. We interpret the universe as a $2^r \times 2^r$ grid of points, in which we place each of our n elements of S. Now these points are not necessarily in a pattern which is of any intuitive use, but we have as our goal to

get one point in each column, so that each can be uniquely identified by its column number, and the row information becomes extraneous. In order to achieve this, we will construct an array of "rotation" factors $\delta[i] \in [2^r]$. The intuitive idea is that we displace each row i by $\delta[i]$, in order to reduce the number of elements which are on the same column. For reasons that will become clear later, we choose **xor** for computing displacements. After a displacement, a point is transformed by $(x, y) \mapsto (y \oplus \delta[x], x)$.



A fundamental result by Tarjan and Yao [6] which makes this scheme possible is that a double displacement suffices to cause each element to have its own column.

$$(x, y) \to (y \oplus \delta[x], x) \to (x \oplus \delta'[y \oplus \delta[x]], y \oplus \delta[x])$$

Theorem 1. *Let q = number of collisions = $\#\{\,((x_1, y_1), (x_2, y_2)) \in S \mid y_1 = y_2\,\}$. Then there exist displacements $\delta[i]$ such that the new number of collisions is $q' = \min\{n, \lfloor \frac{q}{2^{r-3}} \rfloor n\}$.*

Note that one displacement displacement reduces the number of collisions to $q' \leq n$. Now pick $r = \lg n + 4$. The second displacement gives $q'' = \lfloor \frac{n}{2^{\lg n + 1}} \rfloor n = 0$, so two displacements suffice.

Proof: Call the elements of each column S_1, S_2, ... according to their location, and reorder columns so that $|S_1| \geq |S_2| \geq |S_3| \geq \cdots$. We know that $q = \sum_i \binom{|S_i|}{2}$. Let $\delta[i]$ be chosen uniformly at random from $[2^r]$. The number of new collisions is the number of pairs (u, v) with $u \in S_i$, $v \in S_{j<i}$ such that $u \oplus \delta[i] = v \oplus \delta[j]$. By linearity of expectation, this equals $|S_i|(\sum_{j<i} |S_j|)/2^r$. By Markov bounds, with probability $\geq 1/2$, the number of new collisions is $\leq \lfloor 2|S_i|(\sum_{j<i} |S_j|)/2^r \rfloor$. Note that it was possible to add the floor because the number of collisions is always an integer. So $q' = \sum_i$ new collisions in row $i \leq \sum_i \lfloor 2|S_i| \sum_{j<i} |S_j|/2^r \rfloor$.

- first, $q' \leq \sum_i \lfloor |S_i| n 2^{-r+1} \rfloor \leq \sum_i |S_i| n 2^{-r+1} = n^2 2^{-r+1} \leq n$
- second, $|S_i| \leq |S_j|$ (by our ordering), so $q' \leq \sum_i \lfloor 2^{-r+1} \sum_{j \leq i} |S_j|^2 \rfloor$. If $|S_i| = 1$ then $\lfloor 2 \cdot 1 \cdot n/2^r \rfloor = 0$, so we don't care. Otherwise, $|S_j| \geq |S_i| > 1$. Then $|S_j|^2 \leq 4\binom{|S_j|}{2}$, so $q' \leq \sum_i \lfloor 2^{-r+3} \sum_{j<i} \binom{|S_j|}{2} \rfloor$. $\sum_j \binom{|S_j|}{2} = q$, so the number of collisions is at most $n \lfloor 2^{-r+3} q \rfloor$.

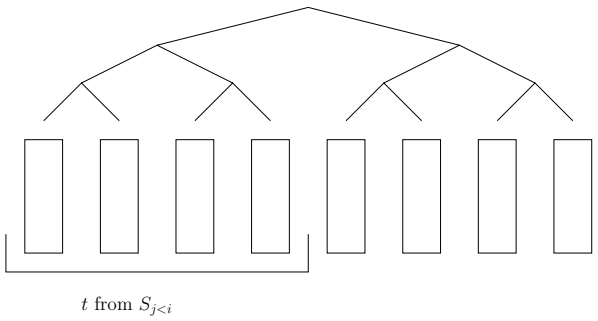
□

This gives us an $O(n)$ construction in expectation, and allows us a trivial $O(n^2)$ deterministic construction. To do so, we try all possible values for $\delta[i]$, and pick the one which leads to the fewest collisions. Choosing $\delta[i]$ then requires time $O(|S_i|n)$, which gives time $O(n^2)$ in total.

To get an $O(n \lg n)$ deterministic construction we need to use a derandomization technique: the method of conditional expectations.

We know that $\delta[i]$ is an integer between 1 and $O(n)$, so instead of trying to determine the entire number, we can try to determine each bit at a time. The first bit can be either 1 or 0. We know that the expected number of collisions when $\delta[i]$ is random is within a good bound, so either the expected number of collisions when the first bit is 0 is within the same bound, or the expected number of collisions when the first bit is 1 is. So we fix the first bit to whatever value gives us the best expected bound, and proceed to the second bit and do the same.

After we have fixed k bits, we know that $x \oplus \delta[i]$ has k fixed bits in the beginning, and $r-k$ random bits at the end. We can use a binary tree in which the leaves are the columns of our grid. Note that traversing an edge down fixes another bit. We know that x will map to a random location of a subtree of height $r-k$. We can thus compute the expected number of collisions due to x by divinding the number t of elements which are already in that subtree, by 2^{r-k}, the number of leaves. Then, the expectation is $\sum_{x \in S_i} \frac{t_x}{2^{r-k}}$. Each internal node needs to tell us the number of elements alredy mapped to some leaf under it. It is easy to maintain these counts: as we fix another bit of $\delta[i]$, we increment the counts for all subtrees that we know our elements will be in.



When all this is done we have computed each $\delta[i]$ in time $O(|S_i| \lg n)$, so the total is $O(n \lg n)$.

References

[1] Torben Hagerup, Peter Bro Miltersen, Rasmus Pagh: *Deterministic Dictionaries*, J. Algorithms 41(1): 69-85 (2001).

[2] Christian Worm Mortensen, Rasmus Pagh, Mihai Pătraşcu: *On Dynamic Range Reporting in One Dimension*, Proc. STOC 2005.

[3] Anna Pagh, Rasmus Pagh, S. Srinivasa Rao: *An Optimal Bloom Filter Replacement*, Proc. SODA 2005.

[4] Rasmus Pagh: *A New Trade-Off for Deterministic Dictionaries*, Proc. SWAT 2000, pp 22-31.

[5] Rasmus Pagh, Flemming F. Rodler: *Lossy Dictionaries*, Proc. ESA 2001, pp 300-311.

[6] Robert Endre Tarjan, Andrew Chi-Chih Yao: *Storing a Sparse Table*, Commun. ACM 22(11): 606-611 (1979).