Lecture 14 — March 30, 2010

Prof. André Schulz *Scribe: Rishi Gupta*

1 Overview

In the last lecture we covered the Separator Decomposition, which rearranges any tree into a balanced tree, and the ART/leaf-trimming Decomposition, which is a tree decomposition used to solve the marked ancestor problem and decremental connectivity problem.

In this lecture we talk about solutions to the static and dynamic **dictionary problem**. Our goal is to store a small set $S = \{1, 2, \ldots, n\}$ of keys from a large universe U, with perhaps some information associated to every key. We want to find a compact data structure with low pre-processing time that will support Query(x) in the static case and Query(x), Insert(x), and Delete(x) in the dynamic case. Query(x) checks if $x \in S$, and returns any information associated with x.

In particular, **FKS hashing** achieves $O(1)$ worst-case query time with $O(n)$ expected space and takes $O(n)$ construction time for the static dictionary problem. **Cuckoo Hashing** achieves $O(n)$ space, $O(1)$ worst-case query and deletion time, and $O(1)$ amortized insertion time for the dynamic dictionary problem.

2 Hashing with Chaining

Our first attempt at a dictionary data structure is *hashing with chaining.* We find a **hash function** $h : U \to \{1, 2, \ldots, m\}$, and maintain a table T with m rows, where $T[j]$ = a linked list (or chain) of $\{x \in S : h(x) = j\}$.

Key look up takes $\frac{\sum |T[j]|^2}{2 \sum |T_j|}$ expected time, so we want to make the chains as equal as possible. We also generally want h to be easy to evaluate. Luckily, such hash functions exist[1]:

2.1 Universal Hashing

Definition 1. *A class of functions $\mathcal{H}$ is c-universal if and only if for all $x \neq y$,*

$$|\{h \in \mathcal{H} : h(x) = h(y)\}| \leq c \cdot \frac{|\mathcal{H}|}{m}.$$

In particular, if h is picked at random from $\mathcal{H}$, $Pr[h(x) = h(y)] \leq \frac{c}{m}$.

We assume for the rest of this section that $m = O(n)$.

[1] It's worth noting that a random function from U to $\{1, \ldots, m\}$ would actually have O(1) expected query time (provided $n = O(m)$). However, such a random function would be infeasibly inefficient to store and evaluate.

Define $Z_x := |\{y \in S : h(x) = h(y)\}|$, namely, the collisions with x in the set of elements we want to store. We can also write this as $Z_x = \sum_y \delta_{h(x),h(y)}$, where δ is the Kronecker delta ($\delta_{a,b} = 1$ if $a = b$ and 0 otherwise). The average query time is simply $E[Z_x]$, and we have

$$E[Z_x] = E\left[\sum_y \delta_{h(x),h(y)}\right] = \sum_y E[\delta_{h(x),h(y)}] = 1 + \sum_{y \neq x} Pr[h(x) = h(y)] = 1 + \frac{nc}{m} = O(1).$$

All that remains then is to find a class of such c-universal functions. An example of a 2-universal class is

$$\mathcal{H} = \{h_a(x) = (ax \pmod p) \pmod m\}_{0<a<p}$$

for a prime $p > m$.

The drawback here is the worst case situation (ie. the longest chain) is $\Theta(\log n / \log\log n)$ [1]. We can actually do a bit better, by using $2d$ hash functions, and inserting the item into the shortest of the $2d$ possible lists, after which the worst case lookup time becomes $\Theta(\log\log n / \log d)$ [2].

2.2 Perfect Hashing

If we don't care about linear space, we can build a hash function with *no* collisions in the static case. Let

$$Z(h) = \sum_{\substack{x<y \\ x,y \in S}} Pr[h(x) = h(y)]$$

which is the expected number of collisions. Pick h randomly from a c-universal class of hash functions. We have from above that for any given x, y, $Pr[h(x) = h(y)] \leq \frac{c}{m}$, so $Z(h) \leq \binom{n}{2}\frac{c}{m}$. If we choose $m = cn^2$, we have that the expected number of collisions is $< 1/2$, and by Markov's inequality, $Pr(h \text{ has no collisions}) > 1/2$. This means that we can pick a collision-free h in O(1) trials.

This is great, except of course that $m = cn^2$ is a lot of space.

3 FKS – Fredman, Komlós, Szemerédi (1984) [3]

FKS hashing is a two-layered hashing solution to the static dictionary problem that achieves $O(1)$ worst-case query time in $O(n)$ expected space, and takes $O(n)$ time to build.

The main idea is to hash to a small table T with collisions, and have every cell T_i of T be a collision-free hash table on the elements that map to T_i.

If S is the original set, we let $S = S_1 \dot\cup S_2 \dot\cup \cdots \dot\cup S_m$, where $S_i = \{x : h(x) = i\}$ (note that $\dot\cup$ denotes disjoint union). Using perfect hashing, we can find a collision-free hash function h_i from S_i to a table of size $O(|S_i|^2)$ in constant time. To make a query then we compute $h(x) = i$ and then $h_i(x)$.

The expected size of the data structure is

$$E\left(\sum |S_i|^2\right) = E\left(\sum |S_i|\right) + 2 \cdot E\left(\sum \binom{|S_i|}{2}\right) = n + 2 \cdot Z(h) = n + O\left(\frac{n^2}{m}\right).$$

If we let $m = O(n)$, we have expected space $O(n)$ as desired, and since the creation of each T_i takes constant time, the total construction time is $O(n)$.

4 More Universal Hashing

Definition 2. *A class $\mathcal{H}$ of hash functions is* (c, k)**-universal** *if for all distinct $x_1, x_2, \ldots, x_k$ and for any $a_1, a_2, \ldots, a_k$*

$$Pr(h(x_1) = a_1 \wedge h(x_2) = a_2 \wedge \cdots \wedge h(x_k) = a_k) \leq \frac{c}{m^k}$$

for all $h \in \mathcal{H}$.

Note that c-universal is the same as $(c, 1)$-universal.

Siegel (1989) [4] showed that a $(c, \log n)$-universal class of hash functions exists, where the functions have $O(1)$ query time and $O(\log n)$ storage (ie. each hash function can be described in $O(\log n)$ bits), which we will need in the next section.

5 Cuckoo Hashing – Pagh and Rodler (2001) [5]

Cuckoo hashing is inspired by the Cuckoo bird, which lays its eggs in other birds' nests, bumping out the eggs that are originally there. Cuckoo hashing solves the dynamic dictionary problem, achieving $O(1)$ worst-case time for queries and deletes, and $O(1)$ expected time for inserts.

Let f and g be $(c, 6 \log n)$-universal hash functions. As usual, f and g map to a table T with m rows. We implement the functions as follows:

- *Query(x)* – Check $T[f(x)]$ and $T[g(x)]$ for x.
- *Delete(x)* – Query x and delete if found.
- *Insert(x)* – If $T[f(x)]$ is empty, we put x in $T[f(x)]$ and are done.

  Otherwise say y is originally in $T[f(x)]$. We put x in $T[f(x)]$ as before, and bump y to whichever of $T[f(y)]$ and $T[g(y)]$ it didn't just get bumped from. If that new location is empty, we are done. Otherwise, we place y there anyway and repeat the process, moving the newly bumped element z to whichever of $T[f(z)]$ and $T[g(z)]$ doesn't now contain y.

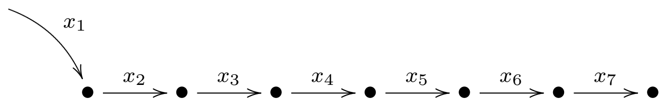
  We continue in this manner until we're either done or reach a hard cap of bumping $6 \log n$ elements. Once we've bumped $6 \log n$ elements we pick a new pair of hash functions f and g and rehash every element in the table.

Note that at all times we maintain the invariant that each element x is either at $T[f(x)]$ or $T[g(x)]$, which makes it easy to show correctness. The time analysis is harder.

It is clear that query and delete are $O(1)$ operations. The reason Insert(x) is not horribly slow is that the number of items that get bumped is generally very small, and we rehash the entire table very rarely when m is large enough. We take $m = 4n$.

Since we only ever look at at most $6 \log n$ elements, we can treat f and g as random functions. Let $x = x_1$ be the inserted element, and $x_2, x_3, \ldots$ be the sequence of bumped elements in order. It is convenient to visualize the process on the *cuckoo graph*, which has verticies $1, 2, \ldots, m$ and edges $(f(x), g(x))$ for all $x \in S$. Inserting a new element can then be visualized as a walk on this graph. There are 3 patterns in which the elements can be bumped.

- *Case 1* Items $x_1, x_2, \ldots, x_k$ are all distinct. The bump pattern looks something like[2]

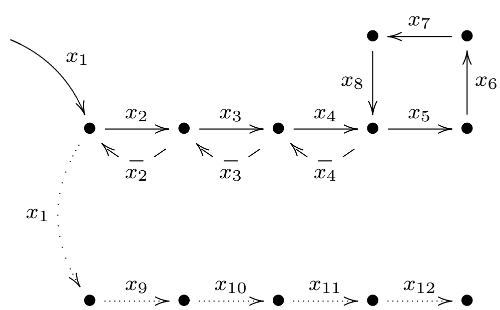


  The probability that at least one item (ie. x_2) gets bumped is

$$Pr(T[f(x)]\text{is occupied}) = Pr(\exists\, y : f(x) = g(y) \vee f(x) = f(y)) < \frac{2n}{m} = \frac{1}{2}.$$

  The probability that at least 2 items get bumped is the probability the first item gets bumped ($< 1/2$, from above) times the probability the second item gets bumped (also $< 1/2$, by the same logic). By induction, we can show that the probability that at least t elements get bumped is $< 2^{-t}$, so the expected running time ignoring rehashing is $< \sum_t t2^{-t} = O(1)$. The probability of a full rehash in this case is $< 2^{-6 \log n} = O(n^{-6})$.

- *Case 2* The sequence $x_1, x_2, \ldots, x_k$ at some point bumps an element x_i that has already been bumped, and $x_i, x_{i-1}, \ldots, x_1$ get bumped in turn after which the sequence of bumping continues as in the diagram below. In this case we assume that after x_1 gets bumped all the bumped elements are new and distinct.

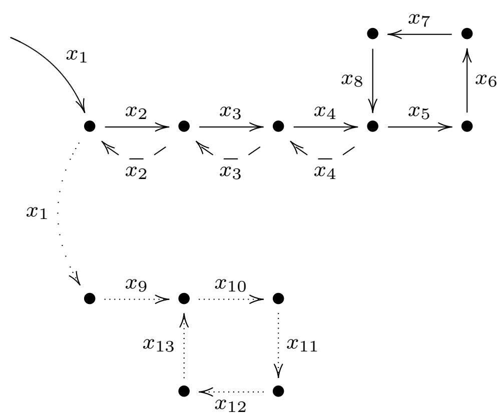


  The length of the sequence (k) is at most 3 times $\max\{\#\text{solid arrows}, \#\text{dashed arrows}\}$ in the diagram above, which is expected to be $O(1)$ by Case 1. Similarly, the probability of a full rehash is $O(2^{\frac{-6 \log n}{3}}) = O(n^{-2})$.

- *Case 3* Same as Case 2, except that the dotted lines again bump something that has been bumped before (diagram on next page).

  In this case, the cost is $O(\log n)$ bumps plus the cost of a rehash. We compute the probability Case 3 happens via a counting argument. The number of Case 3 configurations involving t

[2]Diagrams courtesy of Pramook Khungurn, Lec 1 scribe notes from when the class was taught (as 6.897) in 2005



distinct x_i given some x_1 is ($\leq n^{t-1}$ choices for the other x_i) $\cdot$ ($< t^3$ choices for the index of the first loop, where the first loop hits the existing path, and where the second loop hits the existing path) $\cdot$ (m^{t-1} choices for the hash values to associate with the x_i) $= O(n^{t-1}t^3m^{t-1})$.

The total number of configurations we are choosing from is $\binom{m}{2}^t = O(2^{-t}m^{2t})$, since each x_i corresponds to a possible edge in the cuckoo graph. So the total probability of a Case 3 configuration (after plugging in $m = 4n$) is

$$\sum_t \frac{O(n^{t-1}t^3m^{t-1})}{O(2^{-t}m^{2t})} = O(n^{-2}) \sum_t \frac{t^3}{2^{3t}} = O(n^{-2}).$$

If there is no rehash, the cost of insertion is $O(1)$ from Cases 1 and 2. The probability of a rehash is $O(n^{-2})$. So, we have Cost-of-Insert $=$ Pr(rehash)$\cdot$($O(\log n)$ $+$ $n\cdot$Cost-of-insert) $+$ (1$-$Pr(rehash))$\cdot O(1)$ $=$ $O(1/n)\cdot$Cost-of-insert $+O(1)$, so overall Cost-of-Insert is $O(1)$ in expectation.

References

[1] G. Gonnet, *Expected Length of the Longest Probe Sequence in Hash Code Searching*, Journal of the ACM, 28(2):289-304, 1981.

[2] M. Mitzenmacher, *The Power of Two Choices in Randomized Load Balancing*, Ph.D. Thesis 1996.

[3] M. Fredman, J. Komlós, E. Szemerédi, *Storing a Sparse Table with $O(1)$ Worst Case Access Time*, Journal of the ACM, 31(3):538-544, 1984.

[4] A. Siegel, *On universal classes of fast hash functions, their time-space tradeoff, and their applications,* 30th FOCS, p. 20-25, Oct. 1989.

[5] R. Pagh, F. Rodler, *Cuckoo Hashing*, Journal of Algorithms, 51(2004), p. 122-144.