6.851: Advanced Data Structures Spring 2007

Lecture 11 — March 19, 2007

Oren Weimann *Scribe: Daw-Sen Hwang*

1 Overview

In this lecture, we discuss hashing as a solution to dictionary/membership problem. Various results on hashing are presented with emphasis on static perfect hashing via FKS and dynamic Cuckoo hashing.

2 Dictionary/Membership Problem

In dictionary/membership problem, we want to keep a set S of n items from a universe U with possibly some extra information associated with each one of them. For the membership problem, the goal is to create a data structure that allows us to ask whether a given item x is in S or not. For a dictionary, the data structure should also return the information associated with x. For example, S can be a set of Swahili words such that each of the words is associated with a piece of text which describes its meaning. (Duh!)

The problems have two versions: *static* and *dynamic*. In the static version, S is predetermined and never changes. On the other hand, the dynamic version allows items to be inserted to and removed from S.

We can list the operations as follows:

- *query(x)* – determine $x \in S$ or not. (+ information associated with x)
- *inert(x)* – (dynamic only)
- *delete(x)* – (dynamic only)

3 Hashing with Chaining

Let U denote the universe of items, and let m be a positive integer. A *hash function* is a function from U to $[m]$ for some integer $m < |U|$.

Suppose there exists a hash function $h : U \to [m]$. We can solve the dictionary problem as follows. We maintain a table $T[1 \dots m]$ of linked lists (chains). To insert an item x, we compute $h(x)$ and add x to $T[h(x)]$. To test membership of x, we scan $T[h(x)]$ to see if x is in it or not. (From now on, we use m to represent the size of the table.)

Ideally, we want a hash function that maps every item in the universe to a unique integer. Unfortunately, if $|U| > m$, every hash function will map two different items to the same spot. The next

best thing we can hope for is that our hash function does not make a chain too long. The following theorem says that there are many functions with this property, given that we asymtotically have more than enough spots to hold every item.

Theorem 1. *If $m > n$ and h is selected uniformly from all hash functions then insert/delete/query take $O(1)$ expected time.*

Nevertheless, using a random hash function is infeasible because we cannot represent the function efficiently. The shear size of its representation: we need at least $|U| \log m$ bits. It is also very costly to generate one because we need to generate $|U|$ random numbers. One might try to reduce the space requirement to $O(n \log m)$ bits by generating a new random number as he encounters a new item. However, he will run into the problem of keeping track of seen and unseen items, the dictionary problem itself!

4 Universal Hashing

Fortunately, we do not need a hash function to have such a strong property like randomness to establish $O(1)$ time on every operation. With weaker properties, we can hope for compact representation. For example, *universal family.*

Definition 2. *A set $\mathcal{H}$ of hash functions is said to be a* weak universal family *if for all $x, y \in U$, $x \neq y$,*

$$\Pr[h \leftarrow \mathcal{H} : h(x) = h(y)] = \frac{O(1)}{m}.$$

Example: $\mathcal{H}_{p,m} = \{h_{a,b} \mid a \in \{1, 2, \ldots, p\}, b \in \{0, 1, 2, \ldots, p\}\}$, for some prime $p > |U|$, where $h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$. Each function requires only $O(\log |U|)$ bits to represent, and we can evaluate it in constant time.

Definition 3. *A set $\mathcal{H}$ of hash functions is said to be a* strong universal family *if for all $x, y \in U$ such that $x \neq y$, and for all $a, b \in [m]$,*

$$\Pr[h \leftarrow \mathcal{H} : h(x) = a \wedge h(y) = b] = \frac{O(1)}{m^2}.$$

Definition 4. *A set $\mathcal{H}$ of hash functions is said to be a* k-independent *if for all k distinct items $x_1, x_2, \ldots, x_k \in U$, and for all $a_1, a_2, \ldots, a_k \in [m]$,*

$$\Pr[h \leftarrow \mathcal{H} : h(x_1) = a_1 \wedge h(x_2) = a_2 \wedge \ldots \wedge h(x_k) = a_k] = \frac{O(1)}{m^k}.$$

where k is a constent, we are gonna use $k = O(\log n)$ in cuckoo hashing

Example: pick some $p > |U|$.

$\mathcal{H} = \{h \mid h(x) = (c_0 + c_1 x + \ldots + c_{k-1} x^{k-1}) \bmod m$, for some $c_0, c_1, \ldots c_{k-1} \in [p]\}$.

To see that universal hashing gives what we want, it's enough to show $weakuniversalfamily$ does, suppose we pick m so that $\frac{n}{m} = O(1)$, and let h be a random element of $\mathcal{H}$ Now, an operation that

involves item x has running time proportional to the length of the chain that x is in, which is equal to $\sum_{y \in S} I_y$, where I_y is the indicator random variable that is 1 if and only if $h(x) = h(y)$. So, the expected length (and the expected running time) is

$$E\left[\sum_{y \in S} I_y\right] = \sum_{y \in S} E[I_y] = 1 + \sum_{y \neq x} \Pr[h(x) = h(y)] \leq 1 + n \cdot \frac{O(1)}{m} = O(1)$$

Theorem 5 (Siegel, 1989). *For any $\varepsilon > 0$, there exists a $n^{\Omega(1)}$-independent family of hash functions such that each function can be represented in n^{ε} space, and can be evaluated in $O(1)$ time.*

Theorem 6 (Pagh, Ostlin, 2003). *There exists a n-independent family of hash functions such that each function takes $O(n)$ words to describe, and can be evaluated in $O(1)$ time.*

5 Worst-case Guarantees in Static Hashing

Still, universal hashing gives us only good performance in expectation, making it vulnerable to an adversary who always insert/query items that make the data structure perform the worst. In static hashing, however, we can establish some worst-case upper bounds.

Theorem 7 (Gonnet, 1981). *Let $\mathcal{H}$ be an n-independent family of hash functions. The expected length of the longest chain is $\Theta\left(\frac{\lg n}{\lg \lg n}\right)$.*

By this theorem, we can construct a static hash table with $\Theta\left(\frac{\lg n}{\lg \lg n}\right)$ worst-case running time per each operation. We start by picking a random hash function from the family, hash every item in S, and see if the length of the longest chain is at most twice the expected length. If so, we stop. If not, we pick a new function and start over again. Since the probability that we pick a bad hash function is at most $\frac{1}{2}$, we will find a good hash function after a constant number of trials. The construction thus takes $O(n)$ time in expection.

In fact, we can do better than this. By using two hash functions, adding the inserted item to the shorter list, and searching in both lists when an item is queried, we achieve $\Theta(\lg \lg n)$ length of longest chain in expectation [3].

6 FKS - Static Hashing (Fredman, Komlós, Szemerédi)

A major breakthrough is that we can build a static hash table without any collisions in expected $O(n)$ time with $O(n)$ worst-case space, and any query takes $O(1)$ time [1]. Moreover, the construction requires only a weak universal hashing, and is very easy to implement. This is sometimes called the FKS dictionary, after the initials of the authors.

First attempt: Let $\mathcal{H}$ be a weak universal family of hash function. The main idea is that if $m = \Omega(n^2)$, we have that the expected number of collisions is given by

$$E[\text{number of collisions}] = \sum_{x,y \in S,\ x \neq y} \Pr[h \leftarrow \mathcal{H} : h(x) = h(y)] = \binom{n}{2} \cdot \frac{c}{m} \leq \frac{1}{2}$$

for some constant c. We can choose m large enough, say cn^2, so that the expected number of collision is less than $\frac{1}{2}$. This means that if we pick a random function from $\mathcal{H}$, the probability that the function does not produce any collision is at least $\frac{1}{2}$. Thus, after a constant number of trials, we obtain a collision-free hash function for S.

Second attempt: Now, if $m = n$, we have that the same calculation yields

$$E[\text{number of collisions}] = \binom{n}{2} \cdot \frac{c}{n} = O(n).$$

We also have that we can find a function h' that produces $O(n)$ collisions in linear time.

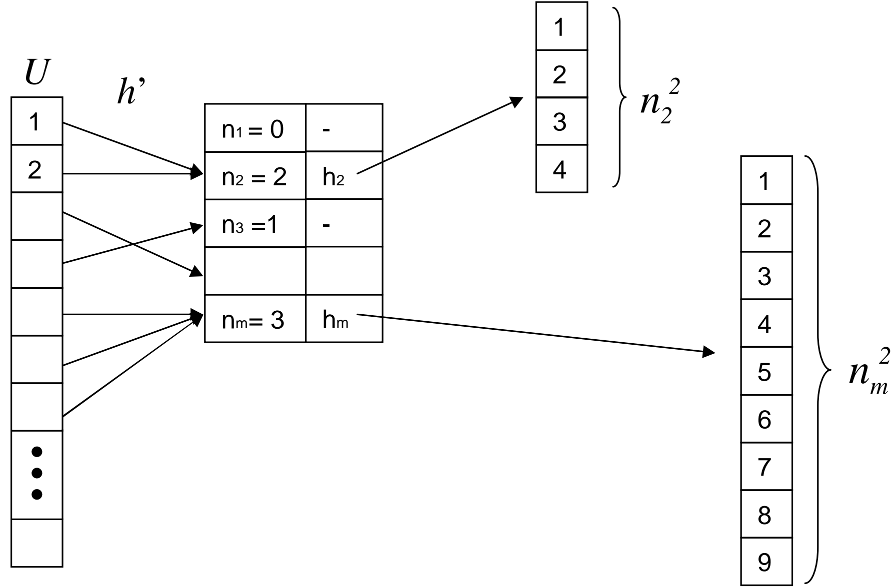


Figure 1: FKS static Hashing

FKS: Let S_i denotes the set of items $x \in S$ such that $h'(x) = i$, and n_i denotes $|S_i|$. Clearly, the S_i's are disjoint, and $\bigcup_{i \in [m]} S_i = S$. Moreover, the number of collisions is $\sum_{i \in [m]} \binom{n_i}{2}$. However, we know that the number of collision is $O(n)$ because we choose h' so. Thus,

$$\sum_{i \in [m]} cn_i^2 \leq \sum_{i \in [m]} 4c \cdot \binom{n_i}{2} = O(n). \tag{1}$$

So, for each i, we use the procedure outlined above to hash S_i into a table of size $O\left(n_i^2\right)$ without any collisions in expected $O(n_i^2)$ time. Thus, the construction takes $O(n) + O(n_1^2) + \ldots + O(n_m^2) = O(n)$ time in expectation. Because of (1) and the fact that a universal hash function can be represented using constant amount of space, the space requirement is $O(n)$ in the worst case. Finally, to answer a query, we first compute $h'(x)$, then use the result to find the hash function for elements in $S_{h'(x)}$, and hash again using that function. Therefore, a query takes constant time in the worst case.

7 Cuckoo - Dynamic Hashing (Pagh and Rodler 2001 [5])

Cuckoo hashing achieves $O(1)$ expected time for insert, and $O(1)$ worst-case time for queries/deletes. However, it make uses of $O(\log n)$-independent hashing. Whether the scheme can achieve the same bound using only $O(1)$-independent hash family is still an open problem.

Cuckoo hashing is on the nesting habit of the Cuckoo bird, when a Cuckoo bird is flying away from his nest for some food, another Cuckoo bird might occupied its nest. When it comes back and sees the nest is occupied, rather than fight with another bird, it will fly away and occupy another nest.

The scheme requires two $O(\log n)$-independent hash functions, say h_1 and h_2. (We'll use $6\log n$-independent hash functions.) The table size m is required to be greater than $2n$. (We'll use $m = 4n$.) Throughout the life of the data structure, it maintains the following invariant: an item x that has already been inserted is either at $T[h_1(x)]$ or at $T[h_2(x)]$. Thus, a query takes at most two probes in the worst case, and deleting an item is very easy as well.

To insert an element x, we carry out the following process.

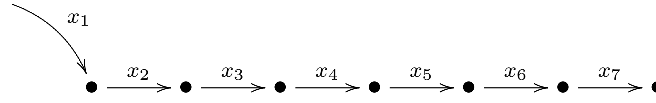
1. Compute $h_1(x)$,
2. If $T[h_1(x)]$ is empty, we put x there, and we are done.. Otherwise, suppose $T[h_1(x)]$ is occupied by y. We evict y, and put x in $T[h_1(x)]$.
3. We find a new spot for y by looking at the other position it can be at, one of the $T[h_1(y)]$ and $T[h_2(y)]$ that is not occupy by x. If the other position is not occupied, we are done. If it is, we put y there and evict the old item. We name the evicted item y, and the item that kicks it out of its old spot x.
4. We repeat step 3 until we find an empty spot, or until we have evicted $6\log n$ items. In the later case, we pick a new pair of hash functions and rehash.

It remains is to analyze the running time of inserting an item. Consider the process of inserting an item x_1. Let $x_1, x_2, \ldots, x_t$ be the sequence of items, with the exception of x_1, that are evicted during the process, in the order they are evicted.

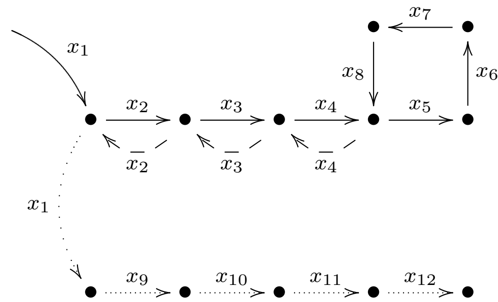
One possibility is that the process continues, without coming back to previously visited cell, until it finds an empty cell. Another possibility is that, at some time, the process comes back to a cell that it has already visited. That is, for some j, the other cell that x_j can be in is occupied by a previously evicted item x_i. Then, $x_i, x_{i-1}, \ldots, x_1$ will be evicted in that order, and x_1 will be sent to $T[h_2(x)]$, and the sequence continues. It may end at an empty cell, or it may run into a previously visited cell. In the later case, the process continues indefinitely if we do not require that it stops after $\log n$ evictions.

We can visualize the above behaviors by considering the cuckoo graph G, whose vertex set is $[m]$ and whose edge set contains edges of the form $(h_1(x), h_2(x))$ for all $x \in U$. In this way, the process of inserting item x_1 can be viewed as a walk on G starting at $h(x_1)$. Visualizations of the three different behaviors are given in figure 1.

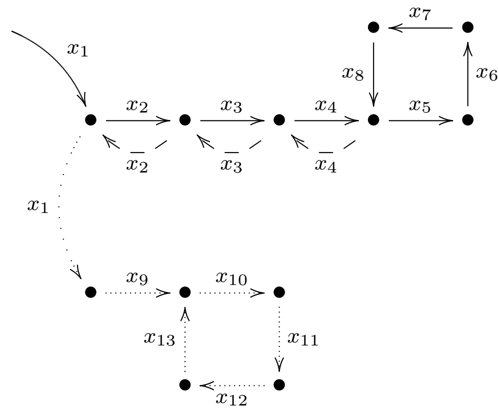
We shall analyze the running time of the three cases. The key observation is that when inserting a new element, we never examine more than $6\log n$ items. Since our functions are $6\log n$-independent, we can treat them as truly random functions.



(a) no cycle



(b) one cycle



(c) two cycles

Figure 2: Three different behaviors of the process of insert element x_1.

- **No cycle:** Let us calculate the probability that the insertion process evicts t items. The process carries out the first eviction if and only if $T[h_1(x_1)]$ is occupied. By the union bound, this event has probability at most

$$\sum_{x \in S, x \neq x_1} (\Pr[h_1(x) = h_1(x_1)] + \Pr[h_2(x) = h_1(x_1)]) < 2\frac{n}{4n} = \frac{1}{2}.$$

  Similarly, the process carries out the second if and only if it has carried out the first eviction, and the cell for the first evicted element is occupied. Thus, by the same reasoning, it carries out the second eviction with probability at most 2^{-2}. And we can argue that the process carries out the tth eviction with probability at most 2^{-t}. Therefore, the expected running time of this case is $\sum_{t=1}^{\infty} t2^{-t} = O(1)$.

  Also, $\Pr r[rehash] \leq 2^{-6 \log n} \leq \frac{1}{n^2}$

- **One cycle:** By considering figure 1(b), we claim that in the sequence $x_1, x_2, \ldots, x_t$ of evicted items, there exists a consecutive subsequence distinct items of length at least $t/3$ that starts with x_1. The reason should be obvious; the sequence can be partition into 3 parts — the solid line part, the dashed line part, and the dotted line part — and one of them must contains at least $t/3$ items. By the same reasoning as in the previous case, we have that the probability that the insertion process evicts all these items is at most $2^{-t/3}$. So, the expected running time in this case is at most $\sum_{t=1}^{\infty} t2^{-t/3} = O(1)$.

  Also, $\Pr r[rehash] \leq 2^{\frac{-6 \log n}{3}} \leq \frac{1}{n^2}$

- **Two cycles:** We shall calculate the probability of a sequence of length t with two cycles, and we do so by a counting argument. How many two-cycle configurations are there?

  - The first item in the sequence is x_1.
  - At most n^{t-1} choices of other items in the sequence.
  - At most t choices for when the first loop occurs, at most t choices for where this loop returns on the path so far, and at most t choices for when the second loop occurs.$\rightarrow t^3$
  - We also have to pick $t-1$ hash values to associate with the items. (The sequence has t edges, but only $t-1$ vertices. See the figure.)$\rightarrow (4n)^{t-1}$

  Thus, there are at most $t^3 n^{t-1}(4n)^{t-1}$ configurations. We have to choose the value of h_1 and the value of h_2 of each item, so the probability that a configuration occurs is given by $2^t(4n)^{-2t}$. Why do we have the 2^t factor? We said that an element x should correspond to some edge (u, v); but this can be achieved in two ways: either $h_1(x) = u, h_2(x) = v$ or $h_1(x) = v, h_2(x) = u$. Thus, the probability that a two-cycle configuration occurs is at most

$$\frac{t^3 n^{t-1}(4n)^{t-1}2^t}{(4n)^{2t}} = \frac{t^3}{4n^2 2^{t-1}}.$$

  Therefore, the probability that a two-cycle occurs at all is at most

$$\sum_{t=2}^{\infty} \frac{t^3}{4n^2 2^{t-1}} = \frac{1}{4n^2} \sum_{t=2}^{\infty} \frac{t^3}{2^t} = \frac{1}{2n^2} \cdot O(1) = O\left(\frac{1}{n^2}\right).$$

So, an insertion causes the data structure to rehash with probability $O(1/n^2)$. Therefore, n insertions can cause the data structure to rehash with probability at most $O(1/n)$. Thus, rehashing, which is basically n insertions, succeeds with probability $1 - O(1/n)$, which means that it succeeds after a constant number trials in expectation. In a successful trial, every insertion must fall into the first two cases. Therefore, a successful trial takes $n \times O(1) = O(n)$ time in expectation. In an unsuccessful trial, however, the last insertion can take $O(\log n)$ time, so it takes $O(n) + O(\log n) = O(n)$ time in expectation as well. Since we are bound to be successful after a constant number of trials, the whole process of rehashing takes $O(n)$ time in expectation. Hence, the expected running time of an insertion is $O(1) + O(1/n^2) \cdot O(n) = O(1) + O(1/n) = O(1)$.

References

[1] 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.

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

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

[4] A. Ostlin, R. Pagh, *Uniform hashing in constant time and linear space*, 35th STOC, p. 622-628, 2003.

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

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