6.897: Advanced Data Structures Spring 2005

Lecture 1 — February 1, 2005

Lecturer: Mihai Pătraşcu *Scribe: Pramook Khungurn*

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 and Cuckoo hashing.

2 Dictionary/Membership Problem

In dictionary/membership problem, we want to keep a set S of items with possibly some extra information associated with each one of them. (From now on, we denote the number of elements in S by n.) 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 infomation 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.

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 $\mathbb{Z}_m$.

Suppose there exists a hash function $h : U \to \mathbb{Z}_m$. We can solve the dictionary problem as follows. We maintain a table $T[0 \ldots m-1]$ 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. *For all $\varepsilon > 0$, if $m > (1+\varepsilon)n$ and h is selected uniformly from the set of all hash functions, then the expected running time of insertion, deletion, and membership query is $O(1)$.*

Nevertheless, using a random hash function is infeasible because 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. An example of such relaxation is *weak universal hashing*.

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

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

To see that weak universal hashing gives what we want, 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

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

Suppose we have such a family of hash function, we can build a dictionary data structure by choosing a hash function at random from the family. If we discover later that our hash function does not perform so well (i.e., there's a chain that is too long), we can pick a new random hash function and *rehash*, i.e., rebuild the table from scratch. This is the advantage of a universal family compared to a fixed heuristic hash function. We will see a more systematic use of this idea below.

An example of weak universal family of hash functions is the family of linear congruence functions, $\mathcal{H} = \{h : \mathbb{F}_p \to \mathbb{Z}_m \mid h(x) = ((ax + b) \bmod p) \bmod m, \text{ for all } x, \text{ and for some } a, b \in \mathbb{F}_p\}$. Here, U is the set of the first k non-negative integers for some k, and p is a prime number greater than $|U|$. Each function requires only $O(\log |U|)$ bits to represent, and we can evaluate it in constant time.

There are also some other variations of universal hashings with stronger properties.

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 $a_1, a_2, \ldots, a_k \in [m]$,*

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

If $k = O(1)$, we can achieve a k-independent family as follows. Suppose again that U is the finite set of integers 0 to $|U| - 1$. Pick a prime number $p > |U|$. We claim the family

$$\mathcal{H} = \{h : \mathbb{F}_p \to \mathbb{Z}_m \mid h(x) = (c_0 + c_1 x + \cdots + c_{k-1}x^{k-1}) \bmod m, \text{ for some } c_0, c_1, \ldots, c_{k-1} \in \mathbb{F}_p\}$$

works. To see this, let $f : \mathbb{F}_p \to \mathbb{F}_p$ denote the function $x \mapsto c_0 + c_1 x + \cdots + c_{k-1}x^{k-1}$. We first show that for any distinct $x_1, x_2, \ldots, x_k \in U$, the distribution

$$\{a_0 \leftarrow \mathbb{F}_p, a_1 \leftarrow \mathbb{F}_p, \ldots, a_k \leftarrow \mathbb{F}_p : (f(x_1), f(x_2), \ldots, f(x_k))\}$$

is uniform. The reason is that the tuple $(f(x_1), f(x_2), \ldots, f(x_k))$ is given by the equation

$$\begin{bmatrix} f(x_1) \\ f(x_2) \\ f(x_3) \\ \vdots \\ f(x_k) \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_k \\ x_1^2 & x_2^2 & \cdots & x_k^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{k-1} & x_2^{k-1} & \cdots & x_k^{k-1} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{k-1} \end{bmatrix}.$$

The big matrix on the right hand side is a Vandemonde matrix. Since $x_i \neq x_j$ for all $1 \leq i < j \leq k$, we have that its determinant is not zero. Thus, for any k-tuple $(f(x_1), f(x_2), \ldots, f(x_k)) \in \mathbb{F}_p^k$, there exists one and only one $(c_0, c_1, \ldots, c_{k-1})$ such that the above equation holds.

Therefore, for any $a_1, a_2, \ldots, a_k \in \mathbb{Z}_m$, and for any distinct $x_1, x_2, \ldots, x_k \in U$, the probability that $h(x_1) = a_1, h(x_2) = a_2, \ldots, h(x_k) = a_k$ is at most $\left(\frac{\lceil p/m \rceil}{p}\right)^k \leq \left(\frac{2}{m}\right)^k = \frac{O(1)}{m^k}$ since k is a constant.

If $k = \omega(1)$, then the above scheme breaks down in several ways. Nevetheless, it is possible to achieve non-constant independence withtout sacrificing time.

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 [4].

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 [2]. 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.

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

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

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.

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.

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

$$\sum_{i \in \mathbb{Z}_m} c|S_i|^2 \leq \sum_{i \in \mathbb{Z}_m} 4c \cdot \binom{|S_i|}{2} = O(n). \tag{1}$$

So, for each i, we use the procedure outlined above to hash S_i into a table of size $O\left(|S_i|^2\right)$ without any collisions in expected $O(|S_i|^2)$ time. Thus, the construction takes $O(n) + O(|S_0|^2) + \ldots + O(|S_{m-1}|^2) = O(n)$ time in expection. 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.

6 Dynamic Hashing

The best results so far in dynamic hashing is due to Dietzfelbinger and Meyer auf der Heide [1]. Their scheme achieves $O(1)$ insert/remove time with high probability and $O(1)$ worst-case query

time. However, the instructor feels that it is quite hairy and decides to cover something simpler and cuter, cuckoo hashing. (The name is kinda cute, isn't it?)

Cuckoo hashing is introduced by Pagh and Rodler in 2001 [6]. It achieves $O(1)$ expected time for updates, and $O(1)$ worst-case time for queries. 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.

The scheme requires two $O(\log n)$-independent hash functions, say h_1 and h_2. The table size m is required to be at least $(2+\varepsilon)n$ for some $\varepsilon > 0$. 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 removing 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 $O(\log n)$ items. In the later case, we pick a new pair of hash functions and rehash.

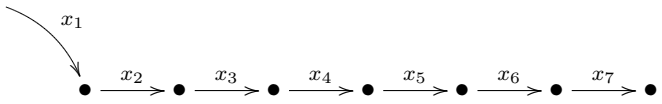
It remains is to analyze the running time of inserting an item. Consider the process of inserting an item x_1. Let $x_1, x_2, \ldots, x_t$ be the sequence of items, with the exception of x_1, that are evicted during the process, in the order they are evicted.

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.

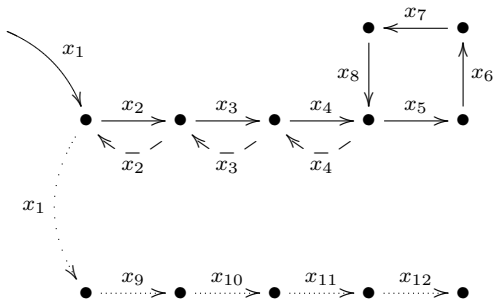
We can visualize the above behaviors by considering the cuckoo graph G, whose vertex set is $\mathbb{Z}_m$ and whose edge set contains edges of the form $(h_1(x), h_2(x))$ for all $x \in U$. In this way, the process of inserting item x_1 can be viewed as a walk on G starting at $h(x_1)$. Visualizations 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 $O(\log n)$ items. Since our functions are $O(\log n)$-independent, we can treat them as truly random functions. Also, to make calculation easy, let us say that $m = 4n$, and that the probability in definition 4 is exactly $m^{-O(\log n)}$.

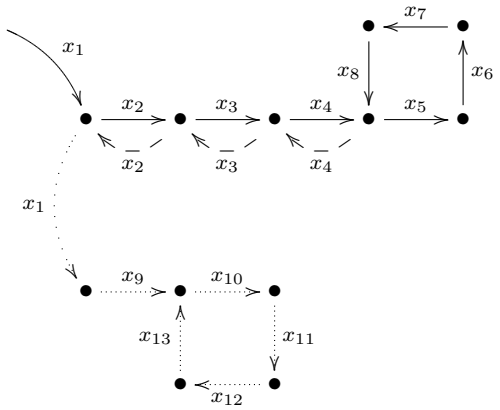
- **No cycle:** Let us calculate the probability that the insertion process evicts t items. The process carries out the first eviction if and only if $T[h_1(x_1)]$ is occupied. By the union bound,



(a) no cycle



(b) one cycle



(c) two cycles

Figure 1: Three different behaviors of the process of insert element x_1.

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 t^{th} eviction with probability at most 2^{-t}. Therefore, the expected running time of this case is $\sum_{t=0}^{\infty} 2^{-t}(t+1) = O(1)$.

- **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 $\sum_{t=0}^{\infty} 2^{-t/3}(t+1) = O(1)$.
- **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.
  - We have at most $(n-1)(n-2)\cdots(n-t+1) < n^{t-1}$ choices of other items in the sequence.
  - We have 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.
  - 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.)

  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

$$\frac{t^3 n^{t-1}(4n)^{t-1}2^t}{(4n)^{2t}} = \frac{t^3}{8n^2 2^{t-1}}.$$

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

$$\sum_{t=2}^{\infty} \frac{t^3}{8n^2 2^{t-1}} = \frac{1}{8n^2} \sum_{t=2}^{\infty} \frac{t^3}{2^t} = \frac{1}{2n^2} \cdot O(1) = O\left(\frac{1}{n^2}\right).$$

Also, the insertion process might stop and initiate rehashing after evicting $O(\log n)$ items while being in the no-cycle or the one-cycle configuration. In this case, we can the threshold to be, say $6 \lg n$, so that the probability of this happening it at most $2^{-(6 \lg n)/3} = 2^{-2 \lg n} = n^{-2}$.

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. Dietzfelbinger, F. Meyer auf der Heide, *A New Universal Class of Hash Functions and Dynamic Hashing in Real Time*, 17th ICALP, p. 6-19, 1990.

[2] 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.

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

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

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

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

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