Design and Analysis of Algorithms — CSCI 5454
Lecture 4 — 26 January 2011
Prof. Aaron Clauset

1 Skip lists (continued)

Last time we shows that the `Search(x)` operation for the skip list takes $O(\log n)$ time. We did this in two parts: first, we showed that w.h.p. the height of the tallest tower in the skip list is $O(\log n)$; second, we showed that the length of the search path on any particular level k is $O(1)$. To finish up, we'll show that the length of the search path on any particular level k is $O(1)$ w.h.p.

Recall that we analyzed the search trajectory backwards, moving from the identified element x in the bottom of the skip list structure "up" and "left" toward the "root" of the skip list. The probability of either of these moves is p, and thus the expected number of "left" moves on any given level is $1/p = O(1)$ when p is a constant. (In our analysis, we assumed $p = 1/2$.)

Because the probability of encountering a taller tower as we move left, and thus moving up to the next higher level, is p, the number of steps we take on a given level is given by a geometric distribution. Thus, the probability that we take more than s steps is

$$\Pr(\text{take more than } 1/p \text{ steps}) = p^{1/p} \ .$$

So long as p is a constant, this probability will also be a constant, and thus w.h.p. we take only a constant number of steps on any particular level in the skip list.

2 Hash tables

Skips lists are one way to implement a *dictionary* ADT, which stores a set of items and allows us to add, remove, and search for dictionary items. *Hash tables* are a more popular way to implement this interface:

Hash Table		
`Add(x)`	$O(1+\alpha)$ expected	$\Theta(n)$ worst
`Find(x)`	$O(1+\alpha)$ expected	$\Theta(n)$ worst
`Remove(x)`	$O(1+\alpha)$ expected	$\Theta(n)$ worst

where α is a measure of how "full" the hash table is. We'll come back to α shortly.

2.1 Perfect Hash

Suppose the things we want to store in our dictionary are drawn from some universe of items U where $|U| = m$ and m is not too large; let x denote an item. A hash table is basically a big array or look-up table where each row in the table stores some items. Crucially, we use a special function $h(x)$, called a *hash function*, which associates items with their locations in the table and allows us to quickly choose where in the table to put a new item and where in the table we put an old item. (What kind of functions should we use for hashing? Would randomization help us here?)

For the moment, suppose that we can choose a perfect hash function, one in which no two elements are assigned the same location or "key." Mathematically we would say $\forall_{x_i,x_j \in U}\ h(x_i) \neq h(x_j)$, and that $h(x)$ is *bijective.* For instance, if we know the set U ahead of time and m is fairly small, then we can construct a perfect hash function specifically for this set.

In this case, we can use an array $A[0 \ldots m-1]$ to store the elements, since each entry will have at most one element in it. Pseudocode for our operations is simply:

```
HT-Add(A,x)    { A[h(x)] = x    }
HT-Find(A,k)   { return A[k]    }
HT-Remove(A,x) { A[h(x)] = NULL }
```

Assuming that computing $h(x)$ takes $O(1)$ time, each of these operations is takes only $O(1)$ time.

Notably, however, if $|U| = m$ is very large, then our hash table will also be very large, which could be problematic. And worse, if the number of items we actually need to store is a small fraction of the set U, much of the table will be empty. Can we do better? Yes: we can trade time for space.

2.2 Practical Hash

By making the hash table smaller than the universe of items, we're confronted with a problem: in order to potentially fit m items into $\ell < m$ locations, some distinct items will have to assigned to the same locations in the table. How can we be smart about spreading items out across the ℓ locations?

2.2.1 Chained Hash

In this version of the hash table, the array A has k elements, each of which is a linked list, sometimes called a *bucket.*[1] The idea is that if $h(x_i) = h(x_j) = k$ for some pair of items, then we simply add the second item to the bucket at $A[k]$. In pseudocode:

[1]In general, any *set* data structure can be used to provide the basic functionality of a bucket. The linked list version is fairly standard, but one could also use a balanced binary tree, a skip list, or even another hash table. Even more generally, using a set data structure for the buckets is a type of *open hashing*, in which each bucket can store, potentially, any number of elements. *Closed hashing* (also, confusingly, called "open addressing") limits the number of elements each bucket can store, and thus has to employ *collision resolution* heuristics, e.g., "linear probing," "quadratic probing," "double hashing," to distribute the excess items to other buckets in a deterministic fashion. This can get complicated and should be avoided if possible. For instance, consider a load factor of $\alpha = 1 - o(1)$.

```
CH-Add(A,x)    { A[h(x)].prepend(x) }
CH-Find(A,k)   { A[k].search(x)     }
CH-Remove(A,x) { A[h(x)].remove(x)  }
```

Assuming that there are no duplicates in our insert sequence, adding an item to a bucket takes $O(1)$ time (if duplicates are possible, how long does insert take?), but searching takes $O(A[k].\text{length})$ time. Deleting takes $O(1)$ time if we know where in the list our target item is, but we don't, so delete takes the same time as search. Thus, we need to be careful about how long these lists get.

This stimulates a natural question: what property should $h(x)$ have in order to minimize the search time, i.e., minimize the length of these lists? The best and worst cases are straightforward:

- *Best case*: If we store n items in a hash table with ℓ slots, then the best we can do is have each bucket contain the same number of items $\alpha = n/\ell$; we call α the *load factor*.
- *Worst case*: $h(x)$ assigns each of n items to the same bucket, and thus search takes $\Theta(n)$ time.

Thus, the performance of a hash table depends wholly on how well $h(x)$ can evenly distributed the set of elements. To analyze the average case, we make the *uniform hashing* assumption:

$$\forall_x \ \Pr(h(x) = k) = 1/\ell \ .$$

That is, independent of the particular input x, the hash function is equally likely to assign it to any of the ℓ slots in the hash table. Note that this is a probabilistic argument for the behavior of a deterministic function. In other words, we want $h(x)$ to behave *as if* it were a uniformly random mapping of elements to locations. Under this assumption, the expected number of elements in any particular linked list $A[k]$ is

$$E(A[k].\text{length}) = \sum_{i=0}^{\ell-1}(1/\ell)A[i].\text{length} = n/\ell = \alpha \ .$$

2.3 The Sparse Case and Two Puzzles

If the load on a hash table is low, i.e., $\alpha = O(1)$, then add, remove and search all take $O(1)$ time. This case is sometimes called *sparse loading*. More precisely, because α counts the average size of a bucket, the expected time for an operation is $O(1+\alpha)$. Relative to other implementations of the dictionary ADT, hash tables become poor performers when $\alpha = \Omega(\log n)$. (Why?)

By assuming that $h(x)$ satisfies the uniform hashing property, we can answer some specific questions about the expected performance of hash tables. For instance, how large does n need to be before we start seeing buckets with multiple items? And, how large does n need to be before every bucket has at least one element in it? (And, when this is true, how large is the largest bucket?) These questions are equivalent to two classic puzzles in probability theory: The Birthday Paradox and the Coupon Collector's Problem. We'll analyze them both.

2.3.1 The Birthday Paradox

Let's assume there are n people in the room and ℓ days in the year; further, let's assume that each of the n people is born on a day chosen uniformly at random from the ℓ days.[2] What is the expected number of pairs of individuals that have the same birthday? And, more generally, as a function of n and ℓ, what is the probability of finding at least 1 pair of individuals with the same birthday?

Let's start with the first question. We can calculate the expected number using indicator random variables and linearity of expectations over pairwise comparisons among the n people. For all $1 \le i < j \le n$, let X_{ij} be an indicator random variable:

$$X_{ij} = \begin{cases} 1 & \text{if persons } i \text{ and } j \text{ have the same birthday} \\ 0 & \text{otherwise} \end{cases} .$$

And recall that

$$E(X_{ij}) = \Pr(\text{persons } i \text{ and } j \text{ have the same birthday}) = 1/\ell \ .$$

Let X be a random variable giving the number of pairs of people with the same birthday; we would like to calculate the expected value of this variable, $E(X)$. Since X is just the sum of the individual pairwise comparisons, we can use linearity of expectations to finish the calculation:

$$\begin{aligned} E(X) &= E\left(\sum_{ij} X_{ij}\right) \\ &= \sum_{ij} E(X_{ij}) \\ &= \sum_{ij} \frac{1}{\ell} \\ &= \binom{n}{2} \cdot \frac{1}{\ell} \\ &= \frac{n(n-1)}{2\ell} \ . \end{aligned}$$

[2]The actual distribution of birthdays across the year is fairly non-uniform, for a number of understandable reasons. Notably, any non-uniform distribution can also be analyzed mathematically.

Thus, if $n(n-1) \geq 2n$, the expected number of pairs of people with the same birthday is at least 1. In other words, if there are at least $\sqrt{2\ell}+1$ people in the room, we can expect to have at least one pair with the same birthday. For $\ell = 365$ and $n = 28$, the expected number of pairs is 1.04. Replace "birthday" with "key" and "people" with "elements," and we see that the same applies for hash tables in the sparse case.

Now, let's turn to the more general question: as a function of n (number of people) and ℓ (number of possible birthdays), what is the probability of finding at least 1 pair of individuals with the same birthday? (That is, what is the probability of a good hash function causing a collision?) Note that

$$\Pr(\text{at least 1 pair shares a birthday}) = 1 - \Pr(\text{no pair shares a birthday}) \ ,$$

which is easier to calculate: assuming the first $k-1$ people have distinct birthdays, the probability that the kth person also has a distinct birthday is $1-(k-1)/\ell$. Thus, the total probability of success is the probability that each birthday is distinct:

$$\Pr(\text{at least 1 pair shares a birthday}) = \begin{cases} 1 - \prod_{i=1}^{n-1}\left(1-\frac{i}{\ell}\right) & n \leq \ell \\ 1 & n > \ell \ . \end{cases}$$

There are a couple of ways to simplify and/or bound this expression. The exact solution is $n!\,\ell^{-n}\binom{\ell}{n}$, but that's not entirely helpful. If we use the Taylor expansion of $\mathrm{e}^{-x} = 1-x+O(x^2)$, which implies the inequality $\mathrm{e}^{-x} > 1-x$, we can bound the probability that no birthdays coincide:

$$\begin{aligned} \Pr(\text{no pair shares a birthday}) &= \prod_{i=1}^{n-1}\left(1-\frac{i}{\ell}\right) \\ &< \prod_{i=1}^{n-1} \mathrm{e}^{-i/\ell} \\ &= \left(\mathrm{e}^{-1/\ell}\right)^{n(n-1)/2} \quad . \end{aligned}$$

For instance, how large does n need to be before the probability of *no collisions* falls below 1/2?

$$\begin{aligned} \left(\mathrm{e}^{-1/\ell}\right)^{n(n-1)/2} &< \frac{1}{2} \\ n^2 - n &> 2\ell \ln 2 \ . \end{aligned}$$

If we choose $\ell = 365$ and $n = 23$ we find $506 > 505.997$; thus, we have $n < 23$. That is, if there are 23 people in the room, and birthdays are distributed uniformly, the probability is better than half that 1 pair of people will have the same birthday.

More generally, we see the probability of getting no collisions decreases very quickly as the number of items we hash increases, for a hash table of a given size.[3] This implies that to keep α very small, we want to keep the hash table very sparse, such that n is not too much bigger than ℓ.

2.3.2 The Coupon Collector's Problem

Our second puzzle was put like so: how large does n need to be before every bucket has at least one element in it? The Coupon Collector Problem models a person trying to collect a complete set of ℓ unique coupons. Coupons are sampled with replacement from the full set, and each occurs with equal probability.

Let n denote the number of coupons we need to collect before we have collected at least one coupon of each type, i.e., before the number of unique coupons is ℓ; let n_i be the number of coupons we have to collect in order to find the ith new coupon, once $i-1$ unique coupons have been collected. Given that we have $i-1$ unique coupons already, the probability that the next coupon will be unique is $p_i = (\ell - i + 1)/\ell$. Thus, the expected time to collect that new coupon is $1/p_i$, and expected total time to collect all ℓ coupons is, by linearity of expectations,

$$\begin{aligned} E(n) &= \sum_{i=1}^{\ell} E(n_i) = \sum_{i=1}^{\ell} \frac{1}{p_i} \\ &= \ell \sum_{i=1}^{\ell} \frac{1}{\ell - i} = \ell \sum_{i=1}^{\ell} \frac{1}{i} \\ &= \Theta(\ell \log \ell) \end{aligned}$$

That is, a good hash function will fill empty buckets fairly quickly. Since it's not hard to show that this claim holds with high probability, we'll do it.

What is the probability that we have *not* filled up the hash table (collected all ℓ coupons) after n trials? The probability that we have not collected the ith coupon in the expected number $n = c\ell \log \ell$ of trials is

$$\begin{aligned} \Pr(\text{no } i\text{th coupon in } n \text{ trials}) &= \left(1 - \frac{1}{\ell}\right)^n \leq \mathrm{e}^{-n/\ell} \\ &= \mathrm{e}^{-c \log \ell} = \ell^{-c} \end{aligned}$$

where we have again used a bound on the Taylor expansion of $1-x$, and we substitute $n = c\ell \log \ell$. This expression should look very familiar—it's exactly the same one we saw in Lecture 3 when

[3]Hash functions are often used in implementations of cryptographic protocols, and a slightly extended version of this analysis can show that the number of trials needed to find a collision is not 2^n for an n-bit hash function, but rather $2^{\sqrt{n}}$, which is much smaller. Searching for such collisions is called a "birthday attack," after the paradox.

we were bounding the height of the towers in skip lists. Taking the Union Bound over these probabilities, we get

$$\begin{aligned}\Pr(n > c\ell \log \ell) &\leq \ell \cdot \Pr(\text{no } i\text{th coupon in } n \text{ trials}) \\ &\leq \ell^{-c+1} \ .\end{aligned}$$

Thus, w.h.p. it only takes $n = \Theta(\ell \log \ell)$ trials to collect all ℓ coupons.

(At-home exercise: show that the largest bucket contains $O(\log \ell)$ items once the hash table has at least one element in each key location. See Problem 11-2.)

2.4 Resizing Hash Tables and Amortized Analysis

Suppose we set some upper limit $\alpha_{\max}$ on how loaded we want to allow our hash table to become, and suppose through a number of operations on it, we cross this threshold. What can we do to preserve the hash table's performance? One solution is to create a new hash table, which will use a new hash function $h'(x)$ with a larger range (number of buckets) $\ell' > \ell$, extract everything from the original table and rehash it into the new table using $h'(x)$.

Assuming chained hashing from Section 2.2.1, extracting n elements from a hash table can be done in $\Theta(n)$ time. Creating a new hash table takes $\Theta(\ell')$ time, and rehashing each of the n elements takes $O(1+\alpha')$ time each, where α' is the load of the new table. Thus, assuming $\alpha_{\max} = O(1)$, this whole operation takes $\Theta(n)$ time. Clearly, if we have to do this operation frequently, the overall cost of maintaining the hash table could be very high. How can we ensure that we're not resizing the hash table too often?

A standard trick is to choose $\ell' = 2\ell$, which gives us plenty of breathing room before we need to resize again.[4] (And, if we are also sensitive to how much space we're wasting, we could halve its size if the load falls below some $\alpha_{\min}$.) The way we would analyze the effective cost of the ADT operations in this case is to use a technique called *amortized analysis* (see Chapter 17), and to consider very long sequences of operations. Because we only incur the cost of a resize operation, either up or down, infrequently, we can amortize (average) the big cost of this rare event over the much more frequent low-cost operations. So long as we're careful with the analysis, our guarantees will hold asymptotically.

For hash tables and using the doubling/halving approach for resizing, the amortized cost of `Add(x)` and `Remove(x)` are $O(1)$.

[4]This same trick can be used to make a *vector* ADT, using an array of fixed length as the underlying data structure, which has no limit on its size. The idea is the same: when the underlying array is too small, make a new one twice as big, insert all the old items into the new one, delete the old one. The size can also be reduced once the effective size is under half of the actual size.

3 For Next Time

1. Read Chapter 16: Greedy Algorithms
2. Reminder: Problem Set 1 due Tuesday, February 1 by 11:59pm via email