1 Efficient Data Structures

Processing data efficiently almost always *also* depends on storing and accessing it efficiently. Storing a zettabyte (10^{12} gigabytes) of data in a linked list is a terrible idea if we want to do anything other than read it linearly, once.

Efficient data structures enable efficient algorithms. In fact, an existing algorithm can often be made more efficient simply by reducing the cost of accessing the data by using more efficient data structures. Examples of efficient data structures include deterministic self-balancing binary search trees like red-black trees (CLRS Chapter 13), AVL trees, and splay trees, as well as probabilistically balanced binary search trees like skip lists (these are super cool).

Consider a sequence of n items $\sigma = \sigma_1, \sigma_2, \ldots, \sigma_n$ and operations on those items $\gamma = \gamma_1, \gamma_2, \ldots, \gamma_n$, where an operation is either `insert`, `find`, or `delete`. If we use a simple linked list to perform γ_i on σ_i for $i = 1..n$, then *in the worst case*, the adversary will choose the γ and σ that cause the ith operation to cost $\Theta(i)$, and the total cost will be $\Theta(n^2)$ (do you see why?). However, if we use a data structure for which each operation costs $O(\log i)$, then the worst case cost is $O(n \log n)$.

Just as algorithms can be more or less efficient, the same is true of data structures, and we use tools like asymptotic analysis and *amortized analysis* to quantify their efficiency. And, because algorithms always depend on data structures, using a more efficient data structure typically makes the algorithm more efficient, too.

Every data structure has an *interface* or represents an *abstract data type* (ADT), which formalizes a consistent set of methods that outside functions use to interact with the data inside the ADT, along with guarantees on the cost of those interactions. (It is not a bad analogy to say that an ADT is like an *application programming interface* (API) for a data structure.) Both an ADT and an API abstract away the implementation details for a specified set of actions (the interface), which allows us to choose different implementations without impacting the behavior of any program that uses the interface, although it can change their running time if our implementation is inefficient.

A self-balancing binary search tree (SBBST) data structure is an efficient implementation of the *dictionary* ADT, which provides methods for inserting, finding, and removing an element from a set of items that permit a comparison operation (e.g., a set of numbers, or strings, or functions under asymptotic growth). Here is the ADT for a dictionary implemented by a self-balancing binary search tree:

SBBST H	behavior	$T(n)$
`Add(x)`	add x to H	$O(\log n)$
`Find(x)`	determine whether x is in H	$O(\log n)$
`Remove(x)`	remove x from H	$O(\log n)$

And, here is the same interface for a linked list implementation of a dictionary, which is much less efficient for everything except insertion (into an unordered list):

linked list H	behavior	$T(n)$
`Add(x)`	add x to H	$O(1)$
`Find(x)`	determine whether x is in H	$O(n)$
`Remove(x)`	remove x from H	$O(n)$

2 Hash tables

Hash tables are another popular way to implement the dictionary interface, and hash tables are best understood as a kind of *randomized data structure*. That is, they are a data structure that uses probabilities to decouple the behavior of the data structure from the particular sequence of operations the adversary may give us (the same strategy we saw with randomized Quicksort).

Because hash tables are probabilistic data structures, we have to think both about their *average* and their *worst* case behaviors; here is the ADT for a hash table implementation of a dictionary ADT:

hash table H	behavior	expected	worst case
`Add(x)`	add x to H	$O(1+\alpha)$	$\Theta(n)$
`Find(x)`	determine whether x is in H	$O(1+\alpha)$	$\Theta(n)$
`Remove(x)`	remove x from H	$O(1+\alpha)$	$\Theta(n)$

where α is a measure of how "full" the hash table is. We will come back to α a little later, when we derive these performance results.

As with other data structures, these methods easily generalize to storing more complicate data, such as $(key, data)$ pairs, where the *data* associated with a particular key x can be arbitrarily complex. For instance, *data* could be a scalar value, a vector of scalars, a graph, or a pointer to some arbitrary data type.

When we store these $(key, data)$ pairs in a dictionary, `Find(x)` returns the *data* associated with the key x iff x is in the data structure; otherwise, it returns `NULL` or *false*. If we are only storing the keys, then `Find(x)` simply determines whether or not x is in the dictionary.

Similarly, `Remove(x)` only deletes x from the dictionary if x is already in the dictionary; if x is associated with some stored data, then we delete both x and its associated data.

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. To choose the row in which to store a given item, we use a *hash function* $h(x) : U \to [1, n]$, which is a deterministic *map* from the universe of hashable items to the set of valid locations in the table. (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, such that each x_i is hashed to one and only one location $h(x_i)$.

If this is true, then 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. If `x` is a key and `v` is its value, then pseudocode functions are simply:

```
PerfectHash-Add(A,x,v)  { A[h(x)] = v    }
PerfectHash-Find(A,x)   { return A[h(x)] }
PerfectHash-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.[1]

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 size of the universe of items $|U|$, we are 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. The efficiency of the hash table thus depends on how evenly we can spread them items across the ℓ possible locations.

[1] We typically assume that evaluating $h(x)$ for any x is a constant-time operation. However, this may not be a good assumption. For instance, if x can be arbitrarily long, as in a string of characters, the time to evaluate $h(x)$ will depend on the input length $|x|$, which may have an impact on our asymptotic analysis.

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*.[2] 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:

```
ChainedHash-Add(A,x,v)  { A[h(x)].prepend([x,v]) }
ChainedHash-Find(A,x)   { A[h(x)].search(x)      }
ChainedHash-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 raises 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, 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 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 each 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

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

[2] 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 $\alpha = 1 - o(1)$.

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 $O(1 + \alpha)$: the cost of computing the hash plus the cost of searching the bucket for the item. Relative to other implementations of the dictionary ADT, hash tables are inefficient 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

Assume there are n people in the room and ℓ days in the year. Furthermore, assume that each person is born on a day chosen uniformly at random from the ℓ days.[3] 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?

We begin 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 $1 \leq i < j \leq n$, 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

[3] The actual distribution of birthdays across the year is fairly non-uniform, for a number of understandable reasons. (See `http://andrewgelman.com/2013/12/19/happy-birthday/` for a nice visualization of the data.) Non-uniform distributions can also be analyzed mathematically and they make the calculations only a little more complicated.

pairwise comparisons, linearity of expectations allows us to finish the calculation:

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

Thus, if $n(n-1) \geq 2\ell$, then 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, meaning we expect at least one collision (shared birthday) if we put any 28 randomly chosen people in a room.[4]

Replace "birthday" with "key" and "people" with "elements," and we see that the same result applies for hash tables in the sparse case. The expected number of hash collisions increases quadratically with the number of elements n, and only decreases linearly with the size of the table ℓ. This implies that the space-time tradeoff we are making by using a hash table to implement a dictionary is a losing game: in order to stay in the sparse loading regime, the size of the table has to be *large* relative to the number of items we are storing, but the rate of collision grows superlinearly as the table gets more full.

2.3.2 When to Expect The Birthday Paradox

We now 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

[4] For the $n = 260$ individuals in our classroom and $\ell = 365$ days, the expected number of birthday "collisions" is a mind-blowing $n(n-1)/2\ell = 92.2$.

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 $n!\,\ell^{-n}\binom{\ell}{n}$, an expression that is not particularly clear or insightful.

However, if we use the Taylor expansion of $\mathrm{e}^{-x} = 1 - x + O(x^2)$, which implies the inequality $\mathrm{e}^{-x} > 1 - x$, then we can bound the probability that no birthdays coincide:

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

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 in our hash table decreases very quickly as the number of items we hash increases, for a hash table of a given size.[5] This implies that to keep α very small, we need to keep the hash table very sparse, such that n is not too much bigger than ℓ.

2.3.3 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

[5] 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.

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. Furthermore, 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,[6] we'll do it.

2.3.4 Coupon Collecting Like a Boss

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

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

[6] The technical definition of "with high probability" is the following: A (parameterized) event E_α occurs *with high probability* if, for any $\alpha \geq 1$, E_α occurs with probability at least $1 - c_\alpha / n^\alpha$, where c_α is a "constant" depending only on α. Informally, something happens w.h.p. if it occurs with probability $1 - O(1/n^\alpha)$. The term $O(1/n^\alpha)$ is called the *error probability*. The idea is that the error probability can be made extremely small by setting α large.

where we have again used a bound on the Taylor expansion of $1 - x$, and we substitute $n = c\ell \log \ell$.

What we now want to do is bound the probability that we have not found the ith coupon after n trials. To do this, we use a tool called the Union Bound. (This is what is called a "concentration bound", because we prove that the probability is concentrated around its expected value.)

The Union Bound says that for a set of events $A_1, A_2, \ldots$, the probability that at least one of the events happens is bounded from above by the sum of the probabilities of the individual events.[7] Mathematically, we say

$$\begin{aligned} \Pr(A_1 \text{ or } A_2 \text{ or } \ldots \text{ or } A_n) &\leq \Pr(A_1) \text{ or } \Pr(A_2) \text{ or } \ldots \text{ or } \Pr(A_n) \\ &\leq \sum_{i=1}^{n} \Pr(A_i) \ . \end{aligned}$$

For our coupon problem, there are n random variables we need to bound, one for each coupon, where each is drawn from the geometric distribution ℓ^{-c} we derived above in Eq. (1).

Because each of these events is independent and occurs with equal probability, we may apply the Union Bound, which reduces the sum to the product of the probability that one event occurs and the number of coupons ℓ:

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

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

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 that after some number of operations we cross that 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, 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

[7] The union bound can be derived by induction on n of the statement $\Pr(A_1 \text{ or } A_2 \text{ or } \ldots \text{ or } A_n) \leq \Pr(A_1) + \Pr(A_2 \text{ or } A_3 \text{ or } \ldots \text{ or } A_n)$, using the identity $\Pr(A \text{ or } B) = \Pr(A) + \Pr(B) - \Pr(A \text{ and } B)$.

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.[8] (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*, 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 are careful with the analysis, our guarantees will hold asymptotically.[9]

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

3 On your own

1. Read Chapter 11: Hash Tables
2. Subsequently, read Chapter 17: Amortized Analysis

[8] 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.

[9] Nora Ephron, the acclaimed essayist, novelist, screenwriter and director, on her love for an apartment in New York City she could not afford (reprinted in The New Yorker in 2006): "I should point out that I don't normally use the word *amortize* unless I'm trying to prove that something I can't really afford is not just a bargain but practically free. This usually involves dividing the cost of the item I can't afford by the number of years I'm planning to use it, or, if that doesn't work, by the number of days or hours or minutes, until I get to a number that is less than the cost of a cappuccino."