Lecture 6: Hashing and Friends

For the next two chapters, we will depart a little from the previous focus which was by and large on *algorithms*, and look into their usual companion: *data structures*. This is not saying there will not be algorithms, or analysis of algorithms, or extensive use of these concentration inequalities and mathematical notions we introduced and went over at great length!

First, just so that we are clear on *what* a data structure is: it is a way to store and organize data while providing a set of methods to access (and, usually, update) this data *efficiently*. This set of methods is the interface to the data (the analogue of an API), and you may have seen it referred in previous courses by what is called an *abstract data type* (ADT). So an ADT specifies an "API to the data," and a data structure is a concrete implementation of this API.[29]

[29] Not an implementation in terms of code, but in terms of algorithms, etc. Once you have a data structure defined and you've analyzed it, it still remains to code it at some point...

Second, we will typically care about time efficiency, but also space (memory) efficiency of a data structure: if we currently store 5 m-bit strings, we would rather avoid using $\Theta(2^m)$ bits of memory – even though in this case that's the size of the "universe" the data comes from (there are 2^m distinct m-bit strings). To quantify this, let's introduce some notation: we will have n *elements* (data points), each of them coming from a *universe* (set of all possible data points) $\mathcal{X}$ of size m. Our two main parameters will be n and m: n can increase or decrease as we add or remove elements from our data structure, and we usually have $n \ll m$ (the universe is big, our dataset is much smaller).

As an example: we want to store 10,000 high-resolution pictures, each 12.5 MP (3072×4080 resolution). Assuming 8 bits per pixel, what would be the corresponding values of n and m?

This is a very naive way to bound m, as not all sets of pixels will be valid images. But it is good enough as a first approximation.

- $n = 10,000$ and $m = 12,533,760$?
- $n = 10,000$ and $m = 100,270,080$?
- $n = 10,000$ and $m = 2^{100,270,080}$?

Clearly, we do not want to use space proportional to m.

Now, one of the most basic and fundamental examples of ADTs is that of the *dictionary*, which only requires to provide 3 operations to maintain a set $S \subseteq \mathcal{X}$:

Or *map*, or *associative array*.

- INSERT(x): insert the element x to S (do nothing if it is already in S)
- LOOKUP(x): return whether $x \in S$
- REMOVE(x): remove the element x from S (do nothing if it is not in S)

Of course, there are several options to implement this, and you most likely have seen or easily come up with a few data structures for that:

Check how you would get these. Can you think of others?

- A linked list! Space $O(n \log m)$, all three operations in (worst-case) time $O(n)$.
- An array! Space $O(m)$, all three operations in (worst-case) time $O(1)$.
- A self-balancing binary search tree (BST), *e.g.*, an AVL tree! Space $O(n \log m)$, all three operations in (worst-case) time $O(\log n)$.

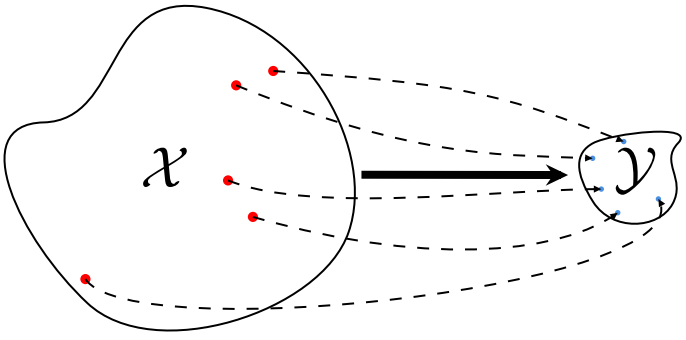
Each of these has its drawbacks, especially the first two. Is there a way to do better? Specifically, can we do space $O(n)$ *and* all three operations in time complexity $O(1)$?

Not quite, but *almost*. Enter hash tables, which, at a high level, use randomization to make the array-based approach much more space-efficient.

Hash tables

The basic idea of hash tables is that "the universe is a big place, but it's mostly empty." So if we could "map" our universe $\mathcal{X}$ to a much, much smaller set $\mathcal{Y}$ such that any subset $S \subseteq \mathcal{X}$ of n distinct elements gets mapped to a subset $S' \subseteq \mathcal{Y}$ *still* of n distinct elements, we would be in good shape: then, we could apply the array-based solution above to $\mathcal{Y}$, only paying space proportional $m' = |\mathcal{Y}| \ll m$. Ideally, we could even take $m' = O(n)$?

Sanity check: we cannot hope for $m' < n$. Can you see why?



Unfortunately, there is an issue with the above approach: *it is not possible*. That is, *no matter what mapping we choose, there will be a set of n elements mapped to fewer than n points.*

Fact 30.1 (Pigeonhole Principle)**.** *Fix any two sets $\mathcal{X}, \mathcal{Y}$ with $m > m'$. Then, for any mapping $h : \mathcal{X} \to \mathcal{Y}$, there exists a set $S \subseteq \mathcal{X}$ of $\left\lfloor \frac{m-1}{m'} \right\rfloor + 1 \geq 2$ elements all mapped to the same value in $\mathcal{Y}$.*

Importantly, this "bad set of elements" depends on the function h. This is merely telling us that, *if* we do this mapping ("hashing") from a large universe $\mathcal{X}$ to a smaller set $\mathcal{Y}$ *deterministically*, then there will be a *worst-case* set of elements for which our strategy fails catastrophically. But what if we did things *at random*?
We can consider three options:

- *The data is randomly distributed:* maybe our n elements are not worst-case, but "typical" in some way, and we can model that as if they were chosen uniformly at random in $\mathcal{X}$. Then the above argument does not go through, and we could use a single, deterministic hash function $h : \mathcal{X} \to \mathcal{Y}$ while still getting good guarantees on average (over the randomness of the data). The main issue is that this is not a very realistic assumption, and what we can prove under this assumption will be more a heuristic as to why we could hope things to work in practice than a rigorous guarantee. Still, better than no guarantees at all.

- *The hash function is totally random:* This would be nice. Then all the values $\{h(x)\}_{x \in \mathcal{X}}$ are independent, uniformly distributed in $\mathcal{Y}$ we can bring in all the tools we have seen to analyze random variables in order to check the probability of a collision, the average number of elements hashed to a bucket $y \in \mathcal{Y}$, the maximum number of elements hashed to any bucket, etc. The main issue is that this will not solve our space issue: a totally random function takes a *lot* of space to store, basically

  $$m \log_2 m'$$

  bits: even worse than the array-based solution! We could try to only define h on-the-fly, by generating $h(x)$ at random only the first time we need to hash $x \in \mathcal{X}$. This would only require $n \log_2 m'$ bits of space... but now, we need to be consistent, and that means first checking if we already decided the value of $h(x)$ earlier. And for that, we need a dictionary – that's the problem we are trying to solve in the first place!

- *The hash function is "somewhat random":* since a single hash function (deterministic) is bad (for collisions), and a truly uniform hash function (picking a function uniformly at random from all $(m')^m$ functions from $\mathcal{X}$ to $\mathcal{Y}$) is also bad (for space), we could try to pick h uniformly at random from a much smaller set of functions $\mathcal{H}$. Such an h will only require $\log_2 |\mathcal{H}|$ bits to store, so if we can design $\mathcal{H}$ to be both small enough that this is space-efficient, and large enough that taking a random h from it looks

like we are picking a truly random function, then we are in good shape. And we are lucky: we have had a glimpse of these *hash families* in Chapter 4, and *they exist*.

Let us start with some more on these hash families: recall the notion of a *strongly universal hash family* $\mathcal{H}$ from Definition 31.1, which was asking that, for any *pair* of distinct $x, x' \in \mathcal{X}$, the two values $h(x), h(x')$ behave exactly (over the random choice of h from $\mathcal{H}$) like two independent and uniformly distributed elements in $\mathcal{Y}$. We saw that such a family $\mathcal{H}$ of size only $2^{\lceil \log(m+1) \rceil}$ existed for the case $|\mathcal{Y}| = 2$ (Fact 22.2).
For the general case, we can invoke the following result:[30]

One can also ask for more than just pairwise independence, and require that, for any k-tuple of distinct $x_1, \dots, x_k \in \mathcal{X}$, their hashed values $h(x_1), \dots, h(x_k)$ behave like k independent uniformly random values in $\mathcal{Y}$. This is called a family of *k-wise independent hash functions*, and again for the specific case of $|\mathcal{Y}| = 2$ can be achieved with a family $\mathcal{H}$ of size $2^{O(\log m)}$.

[30] Try to prove it! This is similar to one of the exercises in Tutorial 4.

Theorem 31. *Fix a prime number $p \geq 2$ and an integer $k \geq 1$. For given $a = (a_0, a_1, \dots, a_k) \in \mathbb{Z}_p^{k+1}$, define the function $h_a \colon \mathbb{Z}_p^k \to \mathbb{Z}_p$ by*

$$h_a(x) = a_0 + \sum_{i=1}^{k} a_i x_i \bmod p, \qquad x \in \mathbb{Z}_p^k$$

and let $\mathcal{H} = \{h_a\}_{a \in \mathbb{Z}_p^{k+1}}$. Then $\mathcal{H}$ is a strongly universal hash family of size $|\mathcal{H}| = 2^{(k+1)\log_2 p}$.

In particular, by Bertrand's postulate, for every m' there exists a prime number $m' \leq p < 2m'$. By choosing the smallest integer k such that $p^k \geq m$, we get a strongly universal hash family from $\mathcal{X}$ to some $\mathcal{Y}$ of cardinality $O(m')$, of size $2^{(k+1)\log_2 p} = 2^{O(k \log m')} = 2^{O(\log m)}$. A little cumbersome, but it works.

Still, strongly universal hash families are a very... strong (!) notion. For hash tables, all we need, in the end, is to have as few *collisions* among hash values are possible: so it makes sense to only ask for this, which brings us to the (weaker) notion of ~~strongly~~ *universal hash family*:

Definition 31.1. A family of functions $\mathcal{H} \subseteq \{h \colon \mathcal{X} \to \mathcal{Y}\}$ is a *universal hash family*, if, for every $x, x' \in \mathcal{X}$ with $x \neq x'$,

$$\Pr_{h \sim \mathcal{H}} \left[h(x) = h(x') \right] \leq \frac{1}{|\mathcal{Y}|}$$

where the probability is over the uniformly random choice of $h \in \mathcal{H}$.

Why this RHS? From Chapter 3 on Balls and Bins, we know that $\frac{1}{|\mathcal{Y}|}$ is the collision probability of two independent uniformly random values in $\mathcal{Y}$: so this definition is basically asking to do, collision-wise, at least as well as if each pair of hashed values behaved like two independent uniform random variables. And asking for an inequality instead of an equality just gives us more freedom when designing our $\mathcal{H}$, so why not?

We will see in the tutorial that it is possible to build universal hash families for which the inequality is strict for *some* pairs x, x'.

This second notion will usually be enough for hash tables. But is it *actually* weaker? As it turns out, yes:

Lemma 31.1. *Every strongly universal hash family is also a universal hash family. Moreover, there exist universal hash families which are not strongly universal.*

Proof. See Tutorial 4. □

To provide an example of a relatively simple (and small) universal hash family, fix any prime number $m \leq p < 2m$ and invoke the following construction:

Theorem 32. *Fix a prime number p. For given integers a, b, define the function $h_{a,b} \colon \mathbb{Z}_p \to [m']$ by*

$$h_a(x) = (ax + b \bmod p) \bmod m', \qquad x \in \mathbb{Z}_p$$

and let $\mathcal{H} = \{h_{a,b}\}_{\substack{1 \leq a < p \\ 0 \leq b < p}}$. Then $\mathcal{H}$ is a universal hash family of size $|\mathcal{H}| \leq 2^{2 \log_2 p}$.

Proof. The last part is clear, as $|\mathcal{H}| = p(p-1)$ (number of choices for the pair (a, b).) To see that it is a universal hash family, note that if $x, x' \in \mathbb{Z}_p$ are distinct and $1 \leq a < p$, then a and $x - x'$ are both among the $p - 1$ invertible elements of $\mathbb{Z}_p$ (which is a field since p is prime). This implies $ax + b \neq ax' + b \bmod p$. As a result, again in the field $\mathbb{Z}_p$, the linear system

$$\begin{aligned} ax + b &= y \\ ax' + b &= y' \end{aligned}$$

has a unique solution in $\mathbb{Z}_p \setminus \{0\} \times \mathbb{Z}_p$ for distinct $y, y' \in \mathbb{Z}_p$ (and no solution for $y = y'$): $a = (x - x')^{-1}(y - y')$ and $b = a(y' - y)^{-1}(yx' - y'x)$. The probability that the two independently chosen a and b take these two unique values is $\frac{1}{p-1} \cdot \frac{1}{p}$. We thus have, over the random choice of $1 \leq a < p$, $0 \leq b < p$, that

$$\Pr_{a,b}\left[\, ax + b = y \bmod p, ax' + b = y' \bmod p \,\right] = \begin{cases} 0 & \text{if } y = y' \bmod p \\ \frac{1}{p(p-1)} & \text{if } y \neq y' \bmod p \end{cases}$$

Finally, $h_{a,b}(x) = h_{a,b}(x')$ if, and only if, $ax + b = y$ and $ax' + b = y'$ for two values $y, y' \in \mathbb{Z}_p$ such that $y = y' \bmod m'$. For any of the p choices of $y \in \mathbb{Z}_p$, there are at most $\lfloor p/m' \rfloor$ such choices of $y' \in \mathbb{Z}_p$:

$$y + m', y + 2m', \ldots, y + \lfloor p/m' \rfloor \cdot m'$$

and so

$$\begin{aligned} \Pr_{a,b}\left[\, h_{a,b}(x) = h_{a,b}(x') \,\right] &\leq p \cdot \left\lfloor \frac{p}{m'} \right\rfloor \cdot \frac{1}{p(p-1)} \\ &\leq p \cdot \frac{p-1}{m'} \cdot \frac{1}{p(p-1)} = \frac{1}{m'} \end{aligned}$$

where we used that, since p is prime, $\lfloor \frac{p}{m'} \rfloor = \left\lfloor \frac{p-1}{m'} \right\rfloor \leq \frac{p-1}{m'}$. □

The above shows that, indeed, *we have good universal hash families* (and even strongly universal ones if needed), with hash functions very easy to evaluate on any given input: so in what follows, we will, unless specified otherwise, go with the third option of "somewhat random hash functions." The name of the game is to establish

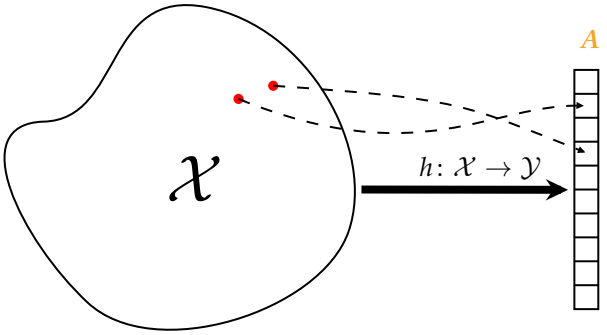
every statement we want to prove as if we were in the "second option" (the most convenient for us!), and at the end check the proof to verify we only used randomness in a way consistent with the third. This typically means relying on linearity of expectation and variance-based arguments such as Chebyshev's inequality, but no Chernoff or Hoeffding bounds (as the versions we have seen in this class require full independence).

There exist Chernoff-type bounds using limited independence, but this is beyond the scope of these lecture notes.

Alright, so what is *a hash table?* We finally get to it. A hash table consists of 3 things:

- A hash function h from the universe $\mathcal{X}$ to a much smaller set $\mathcal{Y}$, usually of size $m' = |\mathcal{Y}| = O(n)$. *[This h is, at the initialization of the data structure, drawn from a "good" hash family $\mathcal{H}$]*;
- An array A of size m', where $A[h(x)]$ will indicate whether element $x \in \mathcal{X}$ is in the data structure; and
- a strategy to handle collisions in when two distinct $x, x' \in \mathcal{X}$ end up in the same bucket (cell) of A because they have the same hash value (*i.e.*, $h(x) = h(x')$).

You may be wondering at this point – what is this third bullet? Did not we do all this hoping to *minimize* the probability of collisions? Why do we still have to worry (and handle) them?



The sad truth is that collisions are inevitable, no matter how carefully we design our hash functions; and, even worse, we already saw why! This is the birthday paradox.

Fact 32.1. *Suppose A is of size $m' \leq c \cdot n^2$, for some absolute constant $c > 0$. Then, even if the hash function $h\colon \mathcal{X} \to [m']$ was truly random, or even if the n data points were truly independent uniformly random elements of $\mathcal{X}$, there would still be a 99% probability at least two elements of the data structure are hashed to the same bucket of A.*

It is even worse than that: since we would like to take $m' = O(n)$, we also have this other result we saw earlier...

At least, for this one, we have some idea of how we could try and resolve it: the power of two choices might help? Peeking ahead, this is the idea behind Cuckoo Hashing.

Fact 32.2. *Suppose A is of size $m' \leq c \cdot n$, for some absolute constant $c > 0$. Then, even if the hash function $h\colon \mathcal{X} \to [m']$ was truly random,*

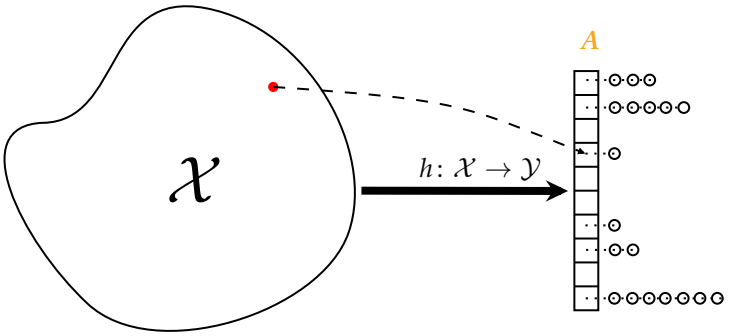
or even if the n data points were truly independent uniformly random elements of $\mathcal{X}$, the expected maximum load among all buckets of A would be $\Omega\left(\frac{\log n}{\log \log n}\right)$. That is, we would expect at least one of the buckets to have at least this many hash collisions.

Handling collisions

Fortunately, there are many strategies to handle collisions, each with its advantages and drawbacks. We can divide them in two broad families: *separate chaining*, and *open addressing*.

(Separate) chaining Separate chaining is the most natural strategy: everybody hash value gets a list! If several of our n data points are hashed to the same bucket in A, add them to a linked list! That is, $A[y]$ will link to a list of all the elements x we want to store such that $h(x) = y$. To implement the 3 operations, we then just delegate to the list stored in the bucket:

- INSERT(x): call $A[h(x)]$.INSERT(x)
- LOOKUP(x): return $A[h(x)]$.LOOKUP(x)
- REMOVE(x): call $A[h(x)]$.REMOVE(x)



In that sense, it combines the hash table (which is based on the naive array-based approach) with the linked-list approach, in an attempt to get the best of both worlds. Let

$$\alpha = \frac{n}{m'} \tag{51}$$

denote the *load factor* of the hash table. For $m' = O(n)$, this will be a (small) constant. Then:

Note that chaining allows the load α to be greater than 1. The next strategy, open addressing, does not.

- the total space used will be

  $$O(\log m + (1+\alpha)n \log m) = O((1+\alpha)n \log m)$$

  (the cost of storing the hash function, and the total space used by the array itself and by all the lists: that last one is proportional to the numbers n of elements currently stored).

- by linearity of expectation, the *expected* time complexity of all 3 operations is $O(1+\alpha)$, since α is the expected size of any given list.
- ... but (think of the max load argument), most likely than not we will have some of the buckets for which the list has size $\Omega(\log n / \log\log n)$.

This leads, for those, to a performance comparable to that of the BST approach!

TODO Give more detail here? Empirical evaluation of the maximum load for the family of hash functions given above.

Open addressing Another approach to handle collisions is *open addressing*, which itself comes in several variants. The basic idea is quite simple: instead of a single hash function, we have a sequence of hash functions $h_1, \dots, h_t, \dots, h_{m'}$. If we are trying to insert an element x in the hash table, we look at the bucket $h_1(x)$: if it's already taken (collision!), then we go to $h_1(x)$: if taken, we look at $h_3(x)$; etc. We stop when we found an empty bucket.[31]

That is, we have the following:

[31] If there is no empty bucket, then this means the hash table is full (the load factor is $\alpha = 1$) and we need to increase m' to resize A – an expensive operation, as this means re-hashing all elements.

- INSERT(x):

  for all $1 \le t \le m'$ **do**
  if $A[h_t(x)] = x$ **then**
  return ▷ Already there
  else if $A[h_t(x)] = \emptyset$ or $A[h_t(x)] = \perp$ **then**
  $A[h_t(x)] \leftarrow x$ ▷ Insert it in the first available bucket
  return

- LOOKUP(x):

  for all $1 \le t \le m'$ **do**
  if $A[h_t(x)] = x$ **then**
  return yes
  else if $A[h_t(x)] = \emptyset$ **then**
  return no ▷ If x was present, it'd have been found earlier

- REMOVE(x):

  for all $1 \le t \le m'$ **do**
  if $A[h_t(x)] = x$ **then**
  $A[h_t(x)] \leftarrow \perp$ ▷ Special symbol to indicate there was something before
  return
  else if $A[h_t(x)] = \emptyset$ **then**
  return ▷ If x was present, it'd have been found earlier

You may wonder why we have this strange symbol $\perp$ when we remove an element x. The reason is that if we just emptied that bucket by (making it $\emptyset$), this would potentially mess up future lookups: we would not know when to stop searching!

But *how do we choose this sequence of hash functions?* We would like a few things: first, to make sure we cover all possible buckets: namely, for every $x \in \mathcal{X}$, we want

$$(h_1(x), \dots, h_{m'}(x))$$

to me a permutation of $\{0, 1, 2, \ldots, m'\}$. This is to make sure we do explore all possible buckets when trying to lookup or insert an element, if we keep finding collisions. Second, we would like to be able to *store* (and evaluate) them all succinctly. We had only one hash function before, and we went to great lengths to make sure it did not take too much space to store, only $O(\log m)$ bits: now, if we have m', we'd like to avoid blowing up our space usage by that factor! For $m' = O(n)$, that would use space $O(n \log m)$...

Before discussing (briefly) some common choices for this sequence of hash functions, let us first analyze the resulting (expected) time complexities of our 3 methods, under some very idealized (and unrealistic) assumptions:

Theorem 33. *Assume that our sequence of hash functions is such that, for every element $x \in \mathcal{X}$, the (random) sequence $(h_1(x), \ldots, h_{m'}(x))$ is a uniformly random permutation of $[m']$. Then, for every $x \in \mathcal{X}$,* Lookup *runs in expected time*

$$O\left(\frac{1}{1-\alpha}\right)$$

where α is the load factor *of the hash table, as defined in* (51).

Proof. Fix any $x \in \mathcal{X}$. Since we want to upper bound the expected running time, it is enough to consider the case where x is *not* in the data structure (unsuccessful lookup), since otherwise the search will end earlier (once it finds x). So the time here will be the number of steps until an empty bucket is found (in which case Lookup finally will return `no`).

By symmetry, the probability that any given bucket is empty is equal to $1 - \frac{n}{m'}$. Let $T(n, m')$ be the time taken by an (unsuccessful) search for item x in the hash table of size m' containing n hash values. If the first index checked corresponds to an empty bucket (with by the above happens with probability $\frac{n}{m'}$), then the search ends after this one step; otherwise, we continue on the remaining subarray of $m' - 1$ buckets, containing the remaining $n - 1$ hash values. And, crucially, the remaining sequence of hash functions values $(h_2(x), \ldots, h_{m'}(x))$ is *still* a uniformly random permutation of these remaining $m' - 1$ buckets (all except the bucket $h_1(x)$, which has been looked at already). So we have the recurrence relation:

$$\mathbb{E}\left[T(n, m')\right] = 1 + \frac{n}{m'} \cdot \mathbb{E}\left[T(n-1, m'-1)\right]$$

We can then prove by induction (over n) that the solution is

$$\mathbb{E}\left[T(n, m')\right] \leq \frac{1}{1 - \frac{n}{m'}},$$

which concludes the proof. □

With this in hand, here are some of the common strategies to implement open addressing:

- Linear probing: forget about using completely distinct hash functions! We have *one* hash function h, let us make the most out of it and just look at the next bucket at each step:

$$h_t(x) = h(x) + (t-1) \bmod m'$$

On the plus side, this is very good in terms of space complexity (we still only store *one* hash function), and very fast to evaluate (as long as h itself is fast to evaluate). How well does this do? The time complexity of any of the 3 functions is again related to the load factor of the hash table, and quickly degrades as α gets close to 1. Namely, we have:

Theorem 34 (Knuth'62). *Assume for simplicity that the hash function $h\colon \mathcal{X} \to \mathcal{Y}$ is a truly random function, and furthermore that the loads of each bucket , $(|h^{-1}(y)|)_{y\in[m']}$, are independent. Then, the expected time complexities of* INSERT, LOOKUP, *and* REMOVE *are all $O\left(\frac{1}{(1-\alpha)^2}\right)$.*

We will not prove this here, but this is actually quite surprising (and bad), and shows that linear probing actually performs much worse than one would think! Indeed, compare this to the wishful analysis of Theorem 33.

- Quadratic probing: the same idea, but now

$$h_t(x) = h(x) + c_1 t + c_2 t^2 \bmod m'$$

for two constants c_1, c_2 with $c_2 \neq 0$ chosen somewhat arbitrarily (but in order to get a permutation of $\{0, 1, 2, \ldots, m'\}$). This is supposed to incur less "clustering" of values than linear probing, while having the same advantages.

- Double hashing: we use *two* hash functions, h, g, and set

$$h_t(x) = h(x) + (t-1) \cdot g(x) \bmod m'$$

- Cuckoo hashing: this one is really neat, as it goes beyond *expected* running times, and actually provides *worst-case* running time guarantees for 2 out of 3 operations. This hashing strategy was proposed and analyzed by Pagh and Flemming in 2001,[32] and relies on a beautiful idea you have seen in an earlier lecture: the power of two choices. Specifically, the data structure uses two hash tables, h_1, A_1 and h_2, A_2. An element $x \in \mathcal{X}$ can only be hashed to one of its two locations, either $A_1[h_1(x)]$ or $A_2[h_2(x)]$: so for lookups and removals, it suffices to check both of these.

  - LOOKUP(x):

    if $A_1[h_1(x)] = x$ or $A_2[h_2(x)] = x$ **then**
        return yes
    else
        return no

[32] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *ESA*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2001; and Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004

– REMOVE(x):
  if $A_1[h_1(x)] = x$ **then**
    $A_1[h_1(x)] \leftarrow \emptyset$
  else if $A_2[h_2(x)] = x$ **then**
    $A_2[h_2(x)] \leftarrow \emptyset$

For insertions, it is a bit more complicated. When inserting an element x, if either $A_1[h_1(x)]$ or $A_2[h_2(x)]$ is empty, we are done: we can insert x to that empty slot. If both are currently occupied, say by two other elements x' and x''... then, an *eviction* occurs (which is where the "cuckoo" part of the name comes from. Cuckoos are... not very nice birds.). That is, x takes the spot of x', forcing x' to go to its own other location in A_2. If that location is empty, then x' goes there and everyone is happy: but if *another* element was there... then x' takes that spot, forcing that element itself to go to its alternate location in the other hash table. And so on and so forth, until the cycle ends or the maximum number of evictions has been reached.

– INSERT(x):
  if $A_1[h_1(x)] = \emptyset$ **then**
    $A_1[h_1(x)] \leftarrow x$
  else if $A_2[h_2(x)] = \emptyset$ **then**
    $A_2[h_2(x)] \leftarrow x$
  else
    $T \leftarrow 0$ ▷ The evictions start
    $x' \leftarrow A_1[h_1(x)]$
    $A_1[h_1(x)] \leftarrow x$
    while $T < T_{\max}$ **do**
      x' goes to $A_2[h_2(x')]$, and if there was something there, that element now needs to move, etc.
      $T \leftarrow T + 1$

Theorem 35. *Cuckoo hashing achieves worst-case time $O(1)$ for* LOOKUP *and* REMOVE*, and expected time $O(1)$ for* INSERT*.*

The guarantees for LOOKUP and REMOVE are immediate; the expected guarantee for INSERT, however, is quite involved. We will not prove it here (but will discuss some of it during the tutorial).

Remark 35.1 (And there is more!)**.** There are other strategies, such as 2-level hashing (where we use a second hash table for each bucket). We will see more during the tutorial.

Bloom filters

As we saw above, a hash table allows us to store and retrieve data very quickly (in expectation, or "for typical data"); the data structures never "make mistakes", since the result is always correct, and the only random aspect is the time complexity. But while they are

typically faster and very space efficient, hash tables do still use some *space*: if each element $x \in \mathcal{X}$ takes $\log_2 m$ (where $m = |\mathcal{X}|$) bits to store, a hash table will take $O(n \log_2 m)$ space to store n elements. Which is very little, and usually alright, but *sometimes* is not.

In this short section, we will see a related data structure, the *Bloom filter*, which uses much less space while still providing efficient access and insertion: only $O(n)$ space to store n elements (regardless of how many bits an element takes to encode)! But this comes at a price: sometimes, the result of a query to the data structure will be wrong. However, when designed well, the frequency with which those mistakes occur is relatively low, and can be controlled.

Let us start with what a Bloom filter is. For simplicity, here we will only allow insertions (INSERT) and lookups (LOOKUP), but no deletions.[33] The Bloom filter is an array A of size m' (containing m' bits, initialized to 0), along with k distinct hash functions $h_1, \dots, h_k$, each mapping the data universe $\mathcal{X}$ to $[m'] = \{1, 2, \dots, m'\}$:

[33] They could be implemented, but this adds quite a bit of complexity to the data structure.

$$h_i \colon \mathcal{X} \to \mathcal{Y} = [m'], \qquad i \in \{1, 2, \dots, m\}$$

where m' and k are parameters to choose. Here is how it works: given an element $x \in \mathcal{X}$,

- INSERT(x) evaluates the k hash functions on x and sets the bit of all k corresponding cells to 1.

  function INSERT(x)
    for all $1 \le i \le k$ **do**
      $A[h_i(x)] \leftarrow 1$

- LOOKUP(x) evaluates the k hash functions on x and checks that all the k bits in the corresponding cells are equal to 1.

  function LOOKUP(x)
    for all $1 \le i \le k$ **do**
      if $A[h_i(x)] = 0$ **then**
        return no
    return yes

,

That's all! In the rest of this lecture, we will try to see what this does, how to analyze the performance, and see how to choose the parameter k (number of hash functions).

What type of "mistakes" can LOOKUP *make?* A Bloom filter can only make one type of errors: *false positives* (returning yes when the element is not in the data structure), never any *false negative* (returning no although the element *is* in the data structure). This is because once its corresponding bits are set to 1, the element will always be reported as present. But a non-present element might have its k bits set to 1 by several other elements. Here's an example

with $m = 3$, $N = 10$, and $\mathcal{X} = \{1,2,3,4\}$: consider the hash functions

	h_1	h_2	h_3
1	1	5	10
2	9	4	6
3	1	6	3
4	7	3	8

Say we insert $S = \{1,2,4\}$: the corresponding bits set to 1 will be (indexing starting at 1)

$$\begin{aligned} 1 &\rightsquigarrow A[1], A[5], A[10] \\ 2 &\rightsquigarrow A[9], A[4], A[6] \\ 4 &\rightsquigarrow A[7], A[3], A[8] \end{aligned}$$

Now, when calling LOOKUP(3), we will check if the bits $A[1], A[6], A[3]$ are all equal to 1. And they all are, so LOOKUP(3) will return yes even though 3 was *not* inserted.

Space complexity Assuming each hash function can be stored in $O(\log m)$ space and takes $O(1)$ time to evaluate, the space complexity of the Bloom filter is

$$O(k \log m + m') \tag{52}$$

and the (worst-case) time complexities of INSERT and LOOKUP are $O(k)$.

Error probability Under the idealized (that is: wrong) simplifying assumption that all k hash functions behave like independent, truly random functions, one can show that the probability that LOOKUP makes an error after n elements have been inserted in the data structure is

$$\left(1 - \left(1 - \frac{1}{m'}\right)^{nk}\right)^k \approx \left(1 - e^{-\frac{nk}{m'}}\right)^k \tag{53}$$

By looking at the tradeoff between space (52) and error probability (53), one can then set the parameters k and m' as desired. For instance, for a fixed n and a target value of space m', we can derive the optimal value of k to set in order to minimize the probability of error.