Hashing

Subhash Suri

January 3, 2019

1 Dictionary Data Structure

- Many computing applications need data structures for fast searches over dynamically changing sets of items. Examples include symbol tables in compilers, address tables in IP routers, active clients at Web servers, spell checkers, caching of game configurations etc.

- A useful abstraction for these data structures is *dictionary*, which is an ADT for supporting the following three operations:

  1. lookup (x): is item with key x in the set?
  2. insert (x): insert x into the dictionary
  3. delete (x): remove x from the d

- While this ADT is called a dictionary, a sorted array representation as in conventional dictionary, with periodic reprinting, is not the solution we want. Specifically, we want to lookup (find) any x in the set in $O(1)$ time, independent of the set size $|S| = n$.

- An important aspect of the dictionary problem is that the *key space*, the set of all possible set values, can be extremely large compared to the actual set of entries stored in the data structure.

- For instance, the english language dictionary contains only about $10,000$ words, but the number of all possible words of length, say, 20 is enormous: 26^{20}. Similarly, a router may need to remember a few hundred or few thousand IP addresses but the space of all IP addresses is 2^{32}, namely, (0.0.0.0) through (255.255.255.255).

- We denote the key space by U, as mnemonic for *universe*. The data set (actual keys present in the application) is denoted by S, with $|S| \ll |U|$. Conventionally, we assume the input set has n keys, that is, $|S| = n$.

2 Two Naive Implementations of Dictionary

- **Linked List**
  1. Just keep an (unordered) linked list of the keys in S.
  2. Insert is fast and easy in $O(1)$ time: add to the head of list.
  3. Find and Delete, however, require searching the list sequentially, and may be $\Omega(n)$ time in the worst-case.
  4. This method is *space efficient* but search and delete performance is very bad.
- **Bit Vector (direct mapping)**
  1. Use a bit array A of size $|U|$, whose ith bit is reserved for the ith key of U.
  2. For instance, the array for storing IP addresses will have size 2^{32}.
  3. To insert key i, we do $A[i] = 1$. To delete it we do $A[i] = 0$.
  4. lookup(i) simply returns $A[i]$.
  5. This method achieved $O(1)$ time for all operations, but requires $O(|U|)$ memory, which is both bad and often completely infeasible (for instance, storing user names in a data base).
- **Balanced Search Trees**
  1. Later we will also see a very general ADT, balanced search trees, which achieve $O(\log n)$ time for all operations using $O(n)$ space, where n is the size of S.
  2. BSTs are powerful but an overkill for implementing dictionaries.

3 Hash Tables

- Hash Tables are a simple and widely used data structure for efficiently implementing the dictionary ADT. They are easy to use in practice although their *analysis* rests on sophisticated math and theory.
- We will assume that all keys are *integers* from the key space $U = \{1, 2, \ldots, |U|\}$. This is not an onerous restriction since non-numeric keys (strings, webpage documents etc) are internally stored in ASCII (numerical) format.
- The basic idea behind hashing is very simple. We will store the keys of the set S in a *table* T of size m, where m is typically $O(n)$, and thus much smaller than $|U|$.

- To make searching for a key x fast, its location in T is determined entirely by the value of x using a *hash function* h which maps each element of U to a unique location in the table: $h : U \to \{0, 1, \ldots, m-1\}$. That is, x is stored at (or near) the location $h(x)$ of T.

- Think of h as assigning each x a *short nickname*, which is easier for the algorithm to remember compared to much larger name x.

- One can imagine many such nickname-assigning functions, such as "use the last few bits of x." For instance, suppose we expect to have at most 100 keys in our set S. Then, we can use the hash function $h(x) = x \pmod{100}$.

- Or, suppose we want to store IP addresses into a table of size 256. We could break the addresses into 4 chunks of 8-bits each. Add up each the chunks bit-by-bit modulo 2, which results in a 8-bit result.

0 1 0 0 1 0 0 0	
1 1 1 1 0 0 1 1	
0 0 1 0 1 1 0 0	
0 0 0 1 1 1 1 1	
1 0 0 0 1 0 0 0	sum modulo 2

- The important point of using a hash function is that each x maps to a unique value $h(x)$, and so applying the same hash function to x every time gives the same output. We want the hash function to have two properties:

  1. $h()$ should be easy and fast to compute, and
  2. $h()$ should *distribute* the keys in the table as uniformaly as possible. In particular, it should not cause multiple keys to hash to the same location.

- The simple hash function $x \pmod{100}$, for instance, satisfies the first property but not the second. If all our input keys happen to end in 00, then h hashes them all to the same location!

- Having multiple keys to hash to the same location is called a *collision*.

- One often sees the recommendation to use hash function $x \pmod{p}$ for some *prime number* p, but that does not solve the problem either: the gaps between input keys may be multiples of p. We need to develop better insights into hashing.

4 Collisions and an Impossibility Result

- Indeed, the hash function "$x \pmod{100}$" works well *if the keys are randomly selected from U*. In that case, the probability that many keys collide is small. But we do not want our data structure to depend on such strong (and unrealistic) assumptions about *input data*. Most data (e.g. program variables) are not random and have strong dependence, creating data clumping, and we want to be able to handle even adversarial data.

- A little bit of thought, however, shows that *no single hash function can solve our problem!* Intuitively, since $m \ll |U|$, the number of possible nicknames is much much smaller than the list of actual names, by the pigeonhole principle *any nickname-assigning rule* must assign the same nickname to many real names (or, same hash table position to many $x \in U$).

- We can formalize this in the following theorem.

- **Theorem:** Suppose a hash table of size m is used to store a set S of n keys drawn from the universe U, where $|U| > nm$. Then, *no matter which* hash function $h : U \to \{0, 1, \ldots, m\}$ is chosen, there is a set $S \subset U$ of n keys that *all map to the same location.* So, in the worst-case no single hash function can avoid linear search complexity!

- **Proof.**

  1. Pick any hash function h of your choosing.
  2. Map all the keys of U using h to the table of size m.
  3. By the pigeon-hole principle, at least one table slot gets n keys because $|U| > nm$.
  4. Choose those n keys as input set S.
  5. Now h will maps S to a single location.

5 Randomization to the Rescue

- The impossibility theorem brings out the key insight behind hashing:

  if the input data is not random, then let's make our hash function random!

- This subtle distiction is a deep and useful algorithm design principle, not just in hashing but in many CS algorithms. When an idea works well on random distributions of data but we cannot assume that the data will be necessarily random, we *simulate* our own randomness in the algorithm!

- In principle, one could imagine an *ideal* random hash function that maps each $x \in U$ to a totally random slot $h(x)$ in the hash table. *For instance, imagine rolling a new m-sided die for each x to determine the hash value $h(x)$.*

- Such a hash function will in theory deliver strong guarantees about minimizing collisions. Unfortunately, such an ideal hash function is also a theoretical fantasy: it is computationally infeasible and inefficient. (For instance, either we have to find a way to ensure that the dice rolls to the same value for a particular x each time, or store that value somewhere to lookup when x is to be inserted. But isn't looking up that value itself the hashing problem?)

- Instead, it turns out that much weaker form of randomness is enough to achieve good hashing performance. In particular, we use "pseudo-random" generators. These are functions that can be efficiently evaluated, and generate "random-seeming" outputs even though they are not really random. They use a seed value to start the random sequence, and the sequence is entirely determined by the seed.

- What does it mean to *choose a random hash function*?

  1. Choosing something at random necessarily means that there is an underlying *collection* from which one member is chosen at random.
  2. For instance, choosing a random card from a deck: there are 52 possibilities, and we can choose the top one after a full shuffle.
  3. So, our goal is to define and construct a *family of hash functions* with some nice properties, and then choose one at random.
  4. Our mathematical analysis will show that with a proper family, and a randomly chosen function, the hash table will have good performance *no matter what set S is fed to it.* In particular, the performance will be good *even if an adversary chooses S.*

- *Exactly what "randomness" do we want from our hash function?*

- One natural implication of randomness is that *any key x is likely to be mapped to each location of the table with equal probability.* That is, suppose we have a key x, and we choose a function h at random from our family of hash functions, then $h(x)$ will be i, with probability $1/m$, for $i = 0, 1, \ldots, m-1$.

- This sounds random enough, right? Unfortunately not!

- Consider the following family of hash functions. It has m members h_a, one each for $a = 0, 1, 2, \ldots, m-1$. Define the hash function as $h_a(x) = a$, for all x.

- Observe that it satisfies the "randomly spreading" property: given any x, the prob. that x maps to i is just $1/m$, because we have m possibilities for our hash function and we will choose h_i precisely with prob. $1/m$.
- But this is not a good hash function at all: it will map all keys to the same location i!
- Since our primary goal is to minimize (eliminate?) *collisions*, what we *really* want is that not too many keys should map to the same location. In other words, we should be asking the question from the *perspective of each table location*: a random hash function should treat all locations equally.

6 Universal Hash Functions

- The first successful definition of such hash functions was given by Carter and Wegman, who called it *universal hash functions*.
- A family of hash functions $\mathcal{H}$ mapping U to $\{0, 1, \ldots, m-1\}$ is called *universal* if for any hash function $h \in \mathcal{H}$ chosen randomly from it satisfies the following:

$$Prob[h(x) \;=\; h(y)] \;\;\leq\;\; \frac{1}{m}, \text{ for any } x, y \in U$$

- The prob. is over the choice of h. That is, adversary can choose x and y, but we (algorithm designers) choose h.
- Universal hashing delivers excellent hash table performance. Specifically, we will prove the following claim.
- **Theorem.** Suppose we apply universal hashing to map a set S of size n to a hash table of size m. Then, the *expected* number of collisions at any hash location is $\leq (n-1)/m$. (Again, the expectation is over our random choice of h, not on the choice of input set S.)
- **Proof.**
  1. Consider any hash location, say, i, and any $x \in S$ such that $h(x) = i$.
  2. By universal hashing, for any other y, the prob. of $h(y) = h(x)$ is at most $1/m$.
  3. Therefore, the expected number of $y \in S$ that map to $h(x)$ is $\leq (n-1)/m$ (by linearity of expectation).
- In particular, if $m = n$, the expected number of collisions is 1.

- We are going to defer the discussion of *how to construct universal hash functions* to later, and for now describe practical considerations of building a useful dictionary data structure.
- Later we will describe several ways to construct these hash functions. For concreteness, however, it is useful to know that one popular family has the following form:

$$h(x) \;=\; ax + b \pmod{p}$$

  where p is a prime number, which also fixes the size of the hash table.
- For now, let's discuss how we implement hash table, *given universal hash functions*. In particular, how do we deal with the problem of collisions. Universal hashing bounds the probability of collisions, but collisions are *inevitable* unless the hash table size is impractically large.

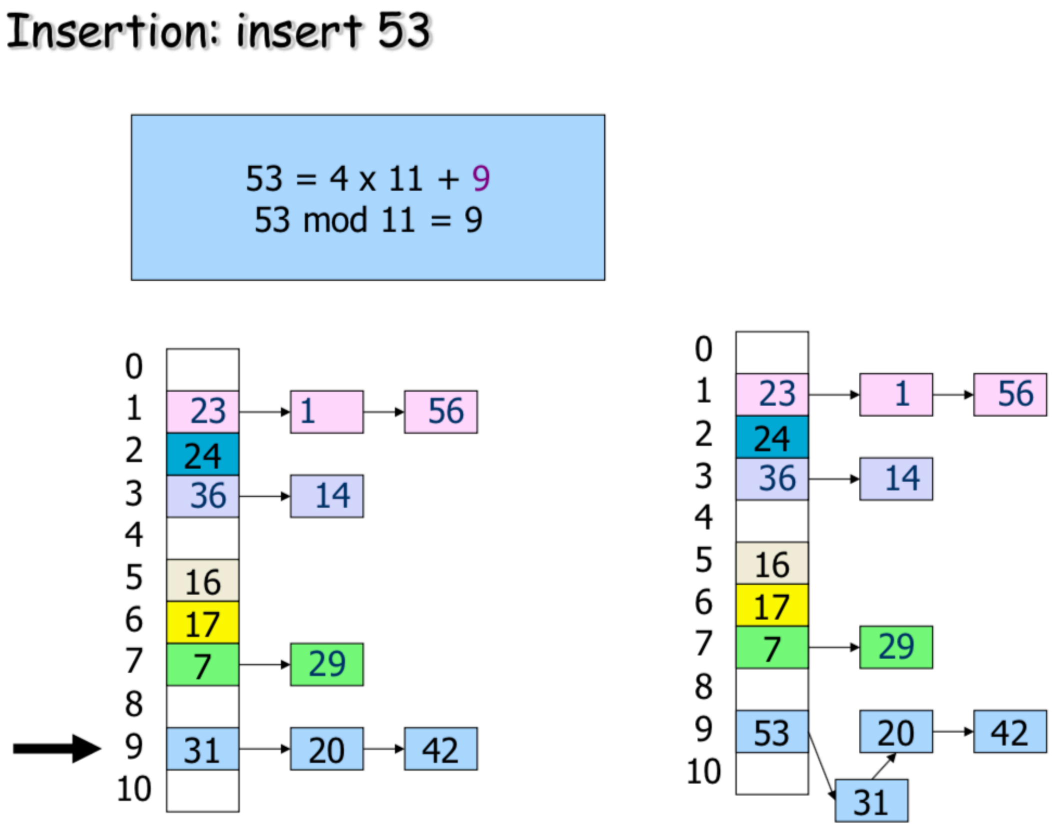
7 Hash Tables: Resolving Collisions

- The basic hash table set up is the following:
  1. Choose a pseudo-random hash function h. (This will be a universal hash function)
  2. An item with key x is put at location $h(x)$.
  3. To find an item with key x, we check location $h(x)$.
- This is easy enough. The only problem is to decide what to do when there is a *collision*: more than one key hashes to the same value.
- There are two main methods for handling collision:
  1. Separate Chaining
  2. Open Addressing

7.1 Separate Chaining

- Separate chaining resolves collisions by maintaining a separate linked list for each hash location, which stores all the keys hashed to that location.
- Insert, Delete, and Find for key x search the linked list at location $h(x)$.
- Each hash operation, therefore, takes time $O(1 + L(x))$, where 1 is for accessing the hash table location $h(x)$, and $L(x)$ is the size of the link list there.

- By the guarantees of universal hashing, the expected size of $L(x)$ is $O(1)$, assuming $m \geq n$.
- An example of searching and inserting with separate chaining. Delete is similar.

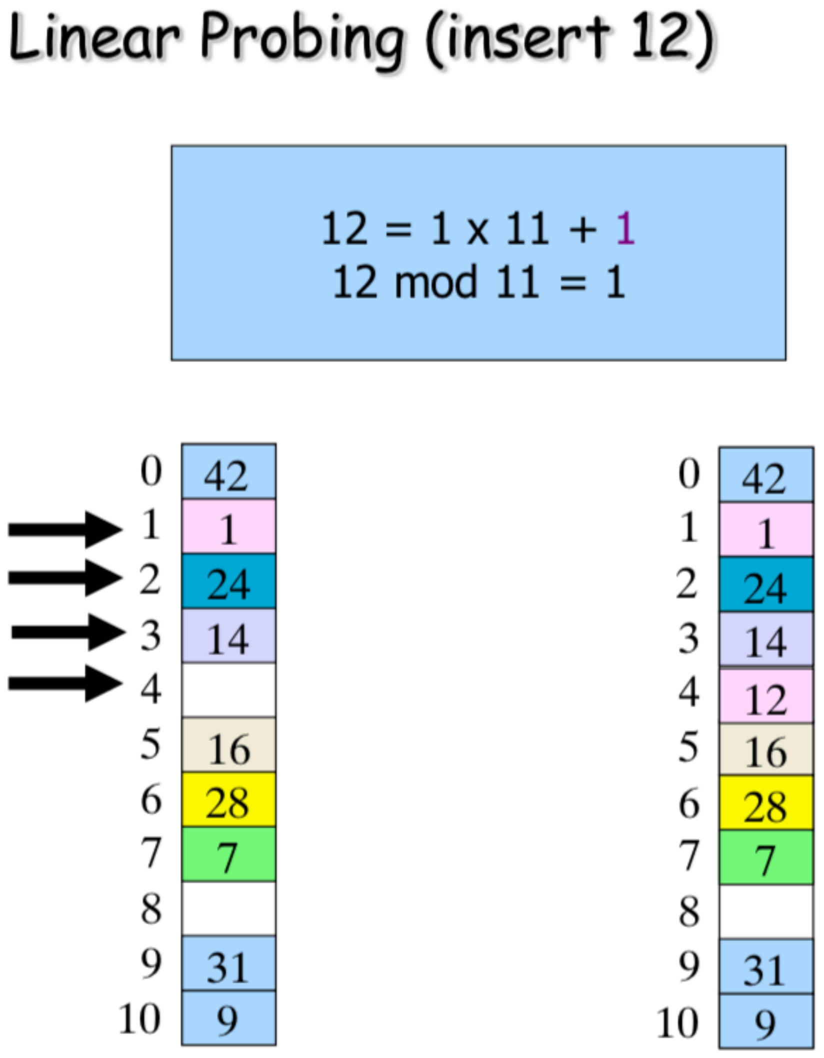


7.2 Open Addressing

- In open addressing, all data is kept in the m-size hash table; no additional memory.
- Upon a collision, we *search the table, starting at the hash location, for an available slot.* We store the new key at first open (unused) location.
- There are many implementations of open addressing, using different strategies for where to *probe* next. Some common examples.
  1. Linear Probing: search locations $h(x) + i$, for $i = 1, 2, \ldots$
  2. Quadratic Probing: search $h(x) + i^2$, for $i = 1, 2, \ldots$

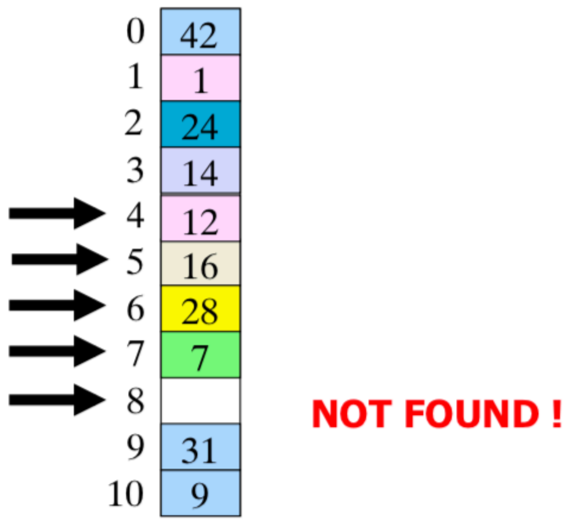
3. Double Hashing: search $h_1(x) + i \times h_2(x)$

- To ensure sufficient empty slots, m is typically a constant times larger than n.
- Illustration of linear probing.



Search with linear probing (Search 15)

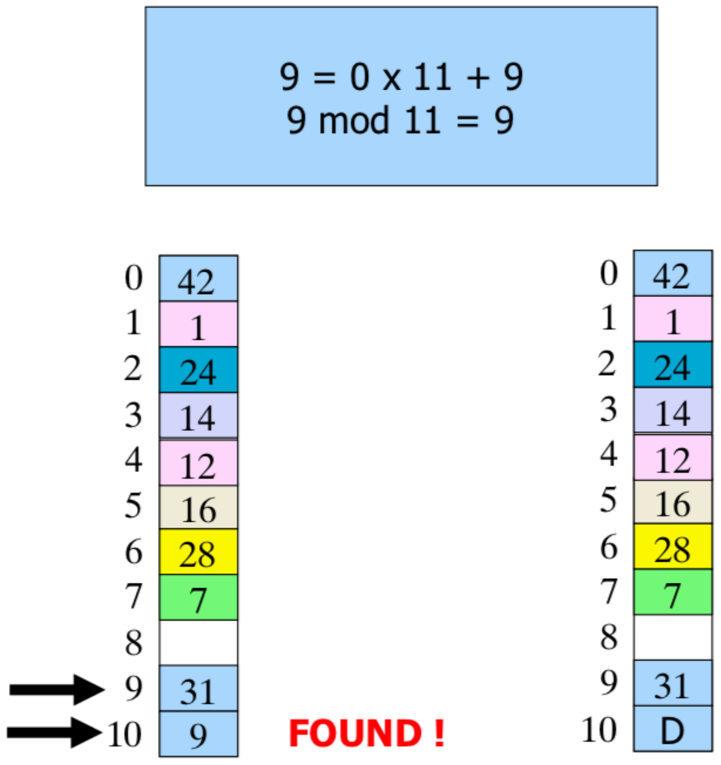
15 = 1 x 11 + 4
15 mod 11 = 4



7.3 Deletion in Linear Probing

- Deletion in open addressing is complicated because we cannot simply remove the item stored at $h(x)$.
- Removal will create an empty slot, which may invalidate the search path for a previous insert.
- One simple idea is to *mark* the slot as empty which is available for new key but lets the search proceed as if it is full.
- An example.

Deletion with linear probing: LAZY (Delete 9)

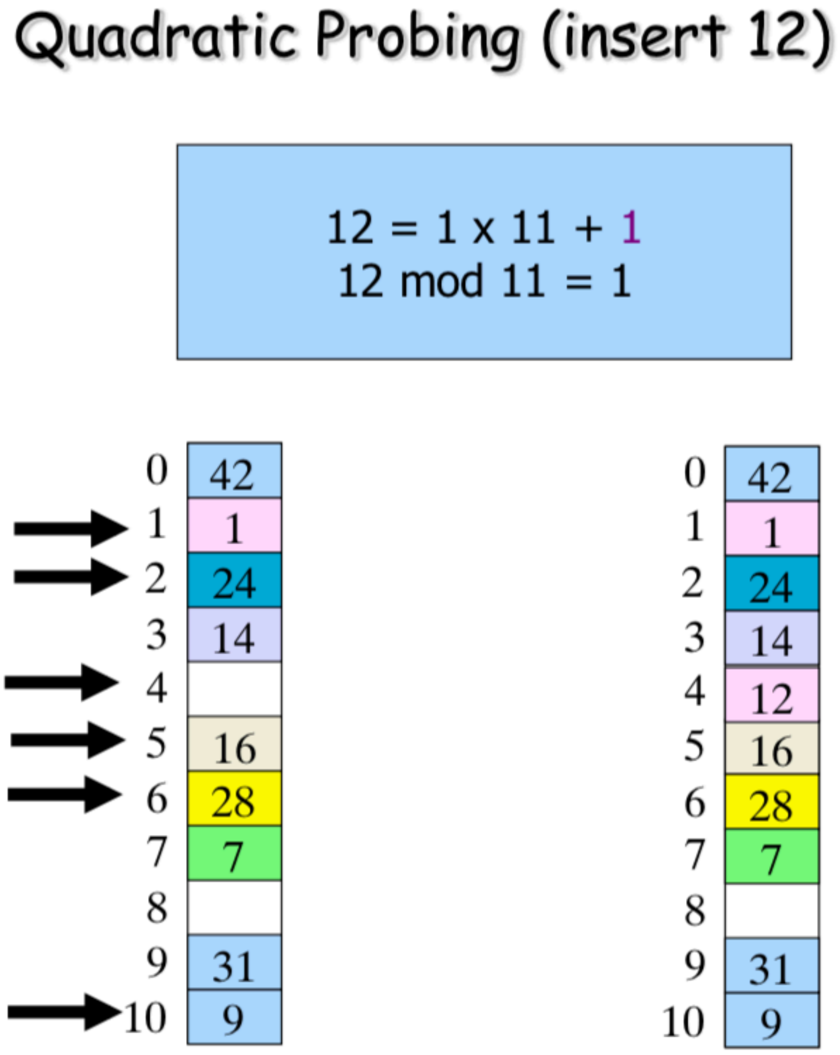


- Another, more complex, method is called Eager Deletion, which deletes the key, but then moves keys around to make sure all previous search paths still are valid.
- Analysis of linear probing is less intuitive, but has similar average case bounds as separate chain.

7.4 Quadratic Probing

- Quadratic probing does larger jumps to overcome the problem of occasional "crowding" in the hash table.
- Check $h(x)$.
- If collision, check $h(x) + 1$
- If collision, check $h(x) + 4$

- If collision, check $h(x) + 9$
- $\cdots$



7.5 Double Hashing

- When collision occurs, we use a *second* hash function.
- One example:

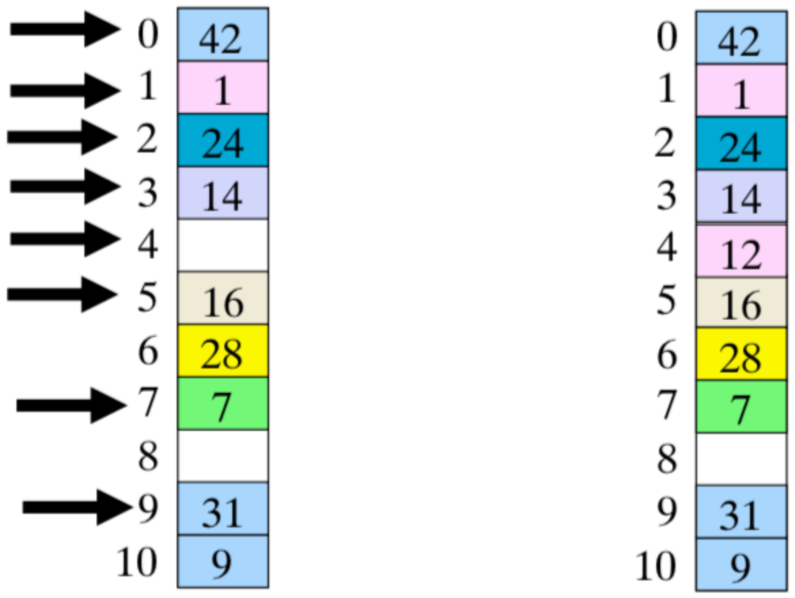
$$h_2(x) = R - (x \bmod R)$$

where R is the largest prime number smaller than table-size.

- Insert 12.
- $h_2(x) = 7 - (x\ mod\ 7)i \ = \ 7(12\ mod\ 7) \ = \ 2$
- Check $h_1(x)$.
- If collision, check $h(x) + i \times h_2(x)$.
- That is, $h(x) + 2$, $h(x) + 4$, etc.

Double Hashing (insert 12)

12 = 1 x 11 + 1
12 mod 11 = 1
7 –12 mod 7 = 2



7.6 Table Size and Resizing

- What do we do if we do not know the size of the input set S when initializing the hash table?
- A common practice is to start with a good guess, and then rebuild the hash table with *double that estimate* when the table gets too full.
- This periodic resizing increases the overall cost of table building by a factor of two.

8 Theory: Constructing Universal Hash Functions

- We present two methods for constructing universal hash functions. The first one is called the Matrix Method. We will call the second one Modular Method.

8.1 The Matrix Method

- We assume that the keys are u-bits long. That is, the size of the universe is $|U| = 2^u$.
- Suppose we want a table of size 2^b, that is, $m = 2^b$.
- We choose a *random* $b \times u$ binary matrix h, and define the hash function has $h(x) = hx$. That is, to get the hash value, we multiply the matrix h with the u-bit long vector x.
- For instance, our universe could be the IP address space of 32 bits, and the table could have size 256. In that case, the matrix h is a 8×32 matrix of 0s and 1s, where each bit is equally likely to be 0 or 1.
- The theoretical guarantee of this hash function is given by the following theorem.
- **Claim.** Suppose $x \neq y$ are two keys. Then,

$$Prob[h(x) = h(y)] \; = \; 1/M \; = \; 1/2^b$$

- **Proof.**
  1. Multiplying matrix h with x means that we add some columns of h (performing vector addition modulo 2), where the 1 bits in x indicate which columns to add.
  2. For instance, consider a toy example where the three rows of h are (1 0 0 0), (0 1 1 1), and (1 1 1 0), and x is the column vector $(1, 0, 1, 0)$.

3. In this case, hx adds first and third columns of h, which are $(1, 0, 1)$ and $(0, 1, 1)$. Their sum modulo 2 is $(1, 1, 0)$. Verify that this is exactly the matrix-vector product hx.
4. Now suppose x and y are two arbitrary keys. Since $x \neq y$, they must differ in at least one coordinate. For concreteness, suppose they differ in ith coordinate, namely, $x_i = 0$ and $y_i = 1$.
5. Imagine now that we have chosen all of h (randomly) *except the column i*.
6. Given this h, the function $h(x)$ remains fixed over all possible choices of column i, since $x_i = 0$. (That is, ith column is not used in the product.)
7. However, each of the 2^b different choices of the column i gives a different value of $h(y)$, since $h \times y$ adds the ith column in the product. In particular, every time we flip a bit in the ith column, we flip the corresponding bit in $h(y)$.
8. Thus, since the ith column is chosen uniformly at random, there is exactly $1/2^b$ chance that $h(x) = h(y)$.
9. This completes the proof.

- Observe that this method produces a *family of hash functions*. Each random matrix h corresponds to a distinct hash function of this family. (Specifically, the family has 2^{ub} hash functions.) By choosing h randomly, we are selecting a member of this family uniformly at random.

8.2 A little bit of number theory: modular arithmetic

- Before we describe the second method, a quick review of the modular arithmetic is helpful.

- Our clocks reset every day, in fact twice a day, at noon. Our calendars reset every year. These modest tricks keep the "numbers that track elapsed time" from getting cumbersomely large. The use of 32 bit numbers in computers also means that the number 2^{32} resets to 0.

- Modular arithmetic is a system for performing arithmetic in which numbers remain bounded in a restricted range.

- Specifically, $x \pmod N$ is defined to be the *remainder* when x is divided by N: that is, if $x = qN + r$, with $0 \leq r < N$, then $x \pmod N = r$.

- This creates equivalence classes of numbers in which all numbers that differ by a multiple of N are *equivalent*. That is,

$$x \;=\; y \pmod{N} \quad \iff \quad N \text{ divides } (x-y)$$

- For instance, $253 = 13 \pmod{60}$ because $253 - 13$ is a multiple of 60.
- Intuitively, the modular arithmetic limits numbers to the range $\{0, 1, \ldots, N-1\}$, and wraps around like the hands of a clock.
- For example, in modulo 3 arithmetic, all integers are grouped into three equivalence classes:

$$\begin{array}{ccccccccc} \cdots & -9 & -6 & -3 & 0 & 3 & 6 & 9 & \cdots \\ \cdots & -8 & -5 & -2 & 1 & 4 & 7 & 10 & \cdots \\ \cdots & -7 & -4 & -1 & 2 & 5 & 8 & 11 & \cdots \end{array}$$

- For instance, when viewed modulo 3, numbers 5 and 11 are no different. Similarly, 4 and -2 are no different.
- In modular arithmetic, the addition and multiplication rules hold: if $x = x' \pmod{N}$ and $y = y' \pmod{N}$, then

$$(x+y) = (x'+y') \pmod{N} \quad \text{and} \quad xy = x'y' \pmod{N}$$

- In real arithmetic, every number $a \neq 0$ has an *inverse* $1/a$. That is, $a \times (1/a) = 1$.
- In modular arithmetic, we say that x is a *multiplicative inverse* of a modulo N if $ax = 1 \pmod{N}$. We denote the inverse by a^{-1}.
- In modular arithmetic, however, the inverse does not always exist! For instance, consider $x \pmod{6}$, whose elements are $\{0, 1, 2, 3, 4, 5\}$. There is no element in this set which is the inverse of 2, that is, $2x \neq 1 \pmod{6}$ for all choices of x.
- To see this, note that $a = 2$ and $N = 6$ are both even, and by the definition of modular system, $N = a - kN$, for some k, $2 \pmod{6}$ is always even.
- It turns out that a necessary and sufficient condition for this anomaly is $gcd(a, N) > 1$. Consequently, if $gcd(a, N) = 1$, that is, if a and N are relatively prime, then inverse always exists. In particular, we have the following theorem.
- *Let N be a prime. Then, for any number $a \in \{1, 2, \ldots, N-1\}$, there exists a unique a^{-1} such that $aa^{-1} = 1 \pmod{N}$.*
- As an example, for $a = 1, 2, 3, 4, 5, 6$, the respective inverses are $1, 4, 5, 2, 3, 6$.

8.3 Modulo Primes Method

- In the modulo primes method, we interpret the input keys as *vector of integers* instead of vector of bits.
- Specifically, we assume that the table size is M, *for some prime* M.
- Then, each key x is written as $[x_1, x_2, \ldots, x_k]$ where each x_i is in the range $[0, M-1]$.
- For instance, suppose keys are IP addresses (32 bits long integers), and the table size is 257, a prime number. We can, therefore, write each key as the vector of four bytes, each bite taking values in the range $\{0, 1, \ldots, 255\}$.
- To select a hash function h, we choose k *random integers* $\{a_1, a_2, \ldots, a_k\}$ in $[0, M-1]$, and define the hash function as:

$$h(x) \quad = \quad a_1x_1 + a_2x_2 + \cdots + a_kx_k \pmod{M}$$

- For instance, in the IP address example, a key $x = 128.0.59.80$ will hash to

$$h(x) \quad = \quad 128a_1 + 0a_2 + 59a_3 + 80a_4 \pmod{257}$$

- The proof that this hash function is also universal follows closely along the lines of matrix method's proof. *We will show that $Pr[h(x) = h(y)] \leq 1/M$, for $x \neq y$.*
- **Proof.**
  1. Since $x \neq y$, there must be some index i such that $x_i \neq y_i$.
  2. Now imagine choosing all the random numbers $a_1, a_2, \ldots, a_k$ *except* a_i.
  3. Let $h'(x) = \sum_{j \neq i} a_j x_j \pmod{M}$ be the "hash" of x using these $k-1$ coefficients.
  4. Clearly, once we pick the last coefficient a_i, we will get $h(x) = h'(x) + a_i x_i \pmod{M}$.
  5. Thus, we have a collision between x and y exactly when $h'(x) + a_i x_i = h'(y) + a_i y_i \pmod{M}$, or equivalently, when

$$a_i(x_i - y_i) \quad = \quad h'(x) \; - \; h'(y) \qquad \pmod{M}$$

  6. Since M is a prime, division by a non-zero value is legal, since every integer in $\{0, 1, \ldots, M-1\}$ has a multiplicative inverse.

7. This means that there is precisely one value of $a_i \pmod M$ for which this equation is true, namely,

$$a_i \;=\; (h'(x) \;-\; h'(y))\,/(x_i \;-\; y_i) \pmod M$$

8. Since a_i is chosen uniformly at random between 0 and $M-1$, the probability of this occurring is precisely $1/M$.

9 Pairwise and k-wise Independent Hashing

- A few years after the original paper, Carter and Wegman proposed a stronger requirement for hashing, which they called *pairwise independent.* A more general idea is the following, called k-wise independence.

- **k-wise independent.** A family of hash functions $\mathcal{H}$ mapping U to $\{0, 1, \ldots, M-1\}$ is k-wise independent if for any distinct k keys $x_1, x_2, \ldots, x_k$ and any k values $a_1, a_2, \ldots, a_k \in [0, M-1]$ (not necessarily distinct), for any $h \in \mathcal{H}$, we have the following:

$$Prob[h(x_1) = a_1 \wedge h(x_2) = a_2 \wedge \cdots \wedge h(x_k) = a_k] \;\;\leq\;\; 1/M^k$$

- The case of $k = 2$ is called *pairwise independent.*

- One can easily prove the following facts about a k-wise independent hash family $\mathcal{H}$, for $k \geq 2$.

  1. $\mathcal{H}$ is also $(k-1)$-wise independent.
  2. $\mathcal{H}$ is universal.

- So, 2-wise independence is at least as strong a condition as universality. On the other hand, a universal hash function is not necessarily 2-wise independent.

10 Theory: How Many Collisions to Expect

- With a random universal hash function, we can guarantee that the expected number of collisions per slot is $O(n/m)$, if n keys are hashed into a table of size m.

- The guarantee holds for *any set S of keys from U*, with the expectation over the choices of hash functions h from the universal family.

- But a guarantee on the *average load* means that some slots may have fewer than average, while others more than average.

  1. How much larger?
  2. How likely is that, say, 100 keys collide?
  3. How uneven can the load be across the hash table?
  4. Suppose we have a hash table of size $M = 10^6$. How large should n be before we start to see collisions?

- The answers to these questions are actually surprising, and may seem counter-intuitive.

- The analysis is often carried out using the classical probability theory setting of "balls and bins."

- Suppose we have n bins, labeled $1, 2, \dots, n$, and a set of k identical balls. We throw each ball in turn *at random* into one of the bins. What is the *distribution of balls into the bins*? In particular, we may ask:

  1. How many balls can be expected to land in each bin?
  2. How large is k before we expect to see a collision?
  3. What is the maximum number of balls expected to land in one bin?

- The balls-and-bins setting an abstract mathematical device for analyzing many problems, including Hashing, but also load balancing in assigning jobs to servers.

- Let us first quickly review the relevant concepts from probability theory, and then consider a famous probability puzzle called *birthday paradox*.

10.1 A Brief Review of Prob. Theory

- Random Variables. A random variable X is a function from the set of possible outcomes to a measurable space, such as R. Typically, we write $X : \Omega \to R$ for real-valued random variables.

- Important point: a random variable **does not** return a probability. It is a numerical function attached to an outcome.

- Examples of a random variable:

  – roll of a dice, where the r.v. is assigned the value of the face on top. In this case, the possible outcomes are $1, 2, ..., 6$.

  - Toss of a coin, where $+1$ for heads, and -1 for tails.
  - Roll of two dice, and $+100$ for sum > 10, and 0 otherwise.

- **Expectation of a random variable.**
  - If X is a random variable that takes on values $x_1, x_2, \ldots$, then the expected value is the average:
$$E[X] = \sum_i x_i.Pr[X = x_i]$$
  - The sum is replaced by integral when X is continuous.

- Example: Expected value of a dice roll.
$$E[X] = \sum_{i=1}^{6} i.Pr[X = i] = \frac{1}{6}\sum_i i = \frac{21}{6} = 3.5$$

- **Variance.** Measure of how much a random variable deviates from its expected value.
$$Var[X] = E[(X - E[X])^2] = E[X^2 - 2XE[X] + (E[X])^2]$$

- (For the middle term, use linearity of expectation, and take out the term $E[X]$ as a fixed constant, which gives us $2(E[X])^2$.)

- Variance simplifies to
$$Var[X] = E[X^2] - E^2[X],$$
where we write $E^2[X]$ for $(E[X])^2$.

- Corollary: $Var[cX] = c^2 Var[X]$.

- **Standard Deviation:** $\sigma(X) = \sqrt{Var[X]}$.
The std deviation has the same unit as expectation, whereas Variance has units "squared."

- Example: Variance of a dice roll.
  - $Var[X] = E[(X - 3.5)^2]$
  - $Var[X] = \sum_{i=1}^{6}(i - 3.5)^2.Pr[X = i]$
  - $Var[X] = \frac{1}{6}[6.25 + 2.25 + 0.25 + 0.25 + 2.25 + 6.25] = 17.5/6 = 2.917$.
  - Standard deviation of a dice roll is 1.7.

- Independence of R.Vs.

- Two r.v. X, Y are independent if and only if, for all x, y, we have

$$Pr[X = x, Y = y] = Pr[X = x].Pr[Y = y]$$

- Variables $X_1, X_2, \ldots, X_n$ are mutually independent iff

$$Pr[X_1 = x_1, \ldots, X_n = x_n] = Pr[X = x_1] \cdots Pr[X = x_n]$$

- X_is are pairwise independence if any two are independent.

- Examples of independent random variables.

- **Linearity of Expectation:**
  - $E[cX] = cE[X]$, for any c
  - $E[X + Y] = E[X] + E[Y]$
  - $E[aX + bY + c] = aE[X] + bE[Y] + c$, for any reals a, b, c.
  - A useful fact: $E[X] = \sum_{i=1}^{\infty} Pr[X \geq i]$, if all $x_i \geq 0$ and integer-valued.

- Linearity of expectation holds without need for independence:

$$E[\sum_{i=1}^{n} X_i] \ = \ \sum_{i=1}^{n} E[X_i]$$

- In the case of Variance, the linearity holds only for *pairwise independent variables*:

$$Var[\sum_{i=1}^{n} X_i] \ = \ \sum_{i=1}^{n} Var[X_i]$$

- The **Union Bound.** During the course, we will be dealing with situations where a set of "bad events" must be precluded. We will need an upper bound on the prob. that *none of the bad events* occur.

- Note that

$$Pr[E_1 \vee E_2 \vee \cdots \vee E_n] \ = \ 1 - \Pi_{i=1}^{n}(1 - Pr[E_i])$$

- A very simple, but still powerful, bound is called the Union Bound:

$$Pr[E_1 \vee E_2 \vee \cdots \vee E_n] \ \leq \ \sum_{i=1}^{n} Pr[E_i]$$

- Union bound holds even for dependent variables, but sometimes it can be too loose to be useful. But we will see many examples of its success.

10.2 Birthday Paradox

- Each person has a unique birthday, which is an integer between 1 and 365.
- We assume that birthdays are chance events: each person's date of birth is a random variable, uniformly distributed in $[1, 365]$.
- The paradox asks the following question:

  What are the chances that in a group of 30 people, at least two have the same birthday?

- One might think not very much: perhaps 10%. There are 365 different birthdays, and 30 is less than one tenth of that. In fact, the prob. is quite large.
- We will show that prob. already with 23 people is 50%. With 30, the prob. is about 73%!
- Let us do a careful analysis.

10.3 Analysis of the Birthday Paradox

- First consider $N = 2$. What is the prob. that 2 people have the same birthday?
- the answer is 1/365: all birthdays are equally likely, so the chance that B's birthday falls on A's birthday is 1 in 365.
- Now suppose we have $N = k$ people.
- It will be more convenient to calculate the prob. X that *no two have the same birthday.* Then, our desired answer will be $(1X)$.
- Define P_i to be the prob. that first i all have distinct birthdays
- For notational convenience, let us write $p = 1/365$.
- Then, we have the following equalities:

$$\begin{aligned}
P_1 &= 1 \\
P_2 &= (1 - p) \\
P_3 &= (1 - p) \times (1 - 2p) \\
&\vdots \\
P_k &= (1 - p) \times (1 - 2p) \times \cdots \times (1 - (k - 1)p)
\end{aligned}$$

- We use the inequality (from Taylor's series expansion of the exponential function) $1 - x \leq e^{-x}$, for all x. Thus, $1 - jp \leq e^{-jp}$.
- We also have $e^x + e^y = e^{x+y}$.
- Therefore,
$$P_k \leq e^{-(p+2p+3p \cdots +(k-1)p)} \leq e^{-k(k-1)p/2}$$
- For $k = 23$, we have $k(k-1)/2 \times 365 = 0.69$, and $e^{-0.69} \leq 0.4999$. Therefore, $1 - P_{23} > 0.5$.
- That is, with just 23 people in a room, there is more than 50% chance that two have the same birthday!
- The connection between birthday paradox and hashing is transparent: think of each person as an input key, and her birthday as a random slot in the table. Given a table of size $M = 365$, after only $n = 23$ keys, we have a 50% chance of collision.
- Going back to our earlier example, if we have a hash table of size $n = 10^6$, it just takes $\sqrt{n} = 1000$ key inserts before a collision occurs with probability at least 1/2!
- So, for instance, if we don't provide any collision-resolution policy, we expect to handle only $O(\sqrt{n})$ items in a table of size n before things break down.

10.4 Balls and Bins Analysis

- The classical balls-and-bins problems are intimately related to the the analysis of hashing.
- Hashing k elements into a table of size n has the same behavior as throwing k (numbered) balls into n (numbered) bins at random.
- The hashing collision problem = prob. of having at least two balls in the same bin. How large can k be to guarantee that prob. of collision is $\leq 1/2$?
- Suppose B is the event that there is at least one collision, and A is the complementary event, namely, all balls land in different bins. Thus, $B = \bar{A}$.
- We want to how large can k be such that $Prob[B] \leq 1/2$.
- Clearly,
$$Prob[A] = \frac{n(n-1)(n-2)\cdots(n-k+1)}{n^k}$$
The numerator counts the number of ways of putting k balls in k distinct bins, and denominator is the number of ways of placing k balls in k bins.

- Our earlier analysis worked out the exact value of k for which $Prob[B] \leq 1/2$. A simpler way to derive a less precise but still good enough bound uses the Union Bound. Suppose E_i are a set of events (no assumption of independence.) Then,

$$Prob[E_1 \cup E_2 \cdot \cup E_k] \leq \sum_{i=1}^{k} Prob[E_i]$$

  In many cases, this crude upper bound can be quite useless (even > 1). But in many CS applications, even this naive bound, which is easy to compute, can get us very close to a useful result.

- In our setting, let B_{ij} be the prob. that balls i and j land in the same bin. Then, clearly, $Prob[B_{ij}] = 1/n$.

- By the union bound, the prob. that some pair of balls land in the same bin is

$$Prob[B] \leq Prob[\bigcup_{ij} B_{ij}] \leq \binom{k}{2}\frac{1}{n} \approx \frac{k^2}{2n}$$

11 Perfect Hashing: table lookup in worst-case $O(1)$ time

- Universal hashing is a scheme for table lookup with *expected* $O(1)$ time complexity. But this result is not strong enough to guarantee a *worst-case* $O(1)$ complexity.

- *Perfect Hashing* guarantees $O(1)$ *worst-case access time for all lookups.*

- Specifically, suppose we are given a fixed set S of n keys, drawn from universe U; no need to support insert and delete; just searches. *What is the best search time we can achieve using $O(n)$ space?*

- We know how to achieve $O(\log n)$ time using a sorted array? Can that be beaten?

- Whether one can achieve perfect hashing, namely, $O(1)$ worst-case lookup using $O(n)$ space, was a major open problem for many years, memorably posed as the problem of *Should tables be sorted?*

- After a long series of increasingly complicated attempts, the problem was solved with a very elegant idea of using universal hash functions in a 2-*level scheme.*

- We first describe a solution that requires $O(n^2)$ space, and then present the final solution with $O(n)$ space.

11.1 Method 1: An $O(n^2)$ Space Solution

- If we are willing to sacrifice $O(n^2)$ space to store n keys, then there is a simple method to achieve perfect hashing.
- We use table size $M = n^2$, and choose a random hash function h from the universal hash function family to hash the keys of S.
- We claim that there is at least 50% chance of zero collisions.
- **Claim.** If we hash n keys into a n^2 size table using a random universal hash function h, then prob. that no two keys collide is $\geq 1/2$.
- **Proof.**
  1. The proof is just the flip side of the birthday paradox.
  2. There are $\binom{n}{2}$ pairs of keys in S.
  3. Since h is a universal hash function, the prob. that any pair collides is $\leq 1/M$.
  4. Therefore, the probability of at least one collision is $\leq \binom{n}{2}/M \; < \; 1/2$.
- So, we try a random h, and if we get any collisions, we discard this table, and choose a new h to try again. Since the prob. of success is $> 1/2$, on average we need only two tries to find a collision-free hashing.

11.2 Method 2: The Optimal $O(n)$ Space Solution

- The method works as follows.
  1. We first hash the keys of S into a table of size n using universal hashing.
  2. This will most likely produce some collisions.
  3. We will then rehash each "bin" (keys hashed to the same slot) using Method 1, with a quadratic size hash table.
  4. That is, we have a first level hash function h and a first level hash table A.
  5. We then have n second-level hash functions $h_1, h_2, \ldots, h_n$, and n second level hash tables $A_1, A_2, \ldots, A_n$.
  6. To look up an element x, we first compute $i = h(x)$, and then (assuming there were collisions at that slot) find the element at location $A_i[h_i(x)]$.
- The correctness of the method follows from correctness of Method 1, and so we can always find the key in *two* lookups.
- What remains is to prove that the method requires only $O(n)$ space.

11.3 Space Complexity Analysis of Perfect Hashing

- The memory needed for the first level table is n.
- Suppose after the first-level hashing, slot i received n_i elements. So, $\sum_i n_i = n$.
- By Method 1, thus, the total memory needed for second level hashing is $\sum_{i=1}^{n} n_i^2$. (We use the shorthand notation n_i^2 for $(n_i)^2$.) We will show below that this quantity is bounded by $O(n)$. Specifically, we prove the following.
- **Lemma.** If the first level hash function h is chosen at random from a universal hash family, then
$$Prob[\sum_i n_i^2 \ > \ 4n] \ < \ 1/2$$
- **Proof.**
  1. We will prove this claim by showing that $E[\sum_i n_i^2] \ < \ 2n$. That suffices because by Markov's inequality, $Pr[X > 2\mu] \ < \ 1/2$. (That is, a random variable cannot have more than 50% chance of assuming a value more than twice the expected value.)
  2. For any two elements $x, y \in S$, define an indicator random variable C_{xy} so that $C_{xy} = 1$ if x and y collide, and $X_{xy} = 0$ otherwise. Then, we have the following sequence of inequalities.

$$\begin{aligned}
E[\sum_i n_i^2] &= E[\sum_x \sum_y C_{xy}] \\
&= n + \sum_x \sum_{y \neq x} E[C_{xy}] \\
&\leq n + n(n-1)/M \\
&< 2n \quad \text{assuming } M = n
\end{aligned}$$

- So, we simply try a random hash function h until we find one that gives $\sum_i n_i^2 \ < \ 4n$, and then fixing that h, we find n secondary hash functions h_i for the second-level tables.
- By this lemma, on average we have to try just twice to find a good h, and once we do, the total memory needed by the data structure is $O(n)$.