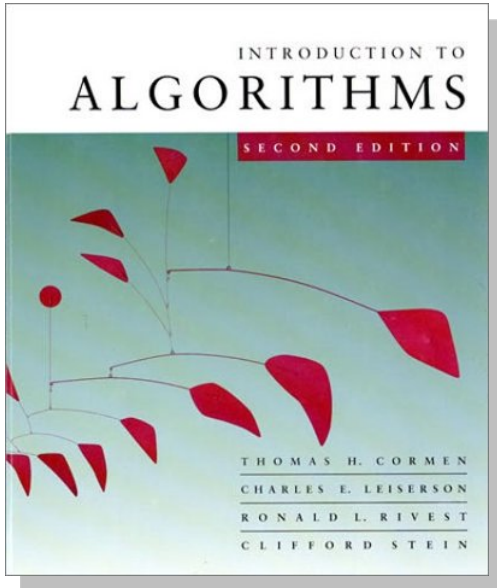


# *Algorithm Analysis*

## BBM408

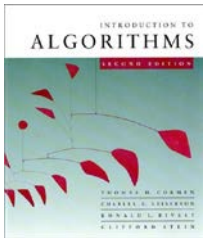


## LECTURE 7

### Hashing I

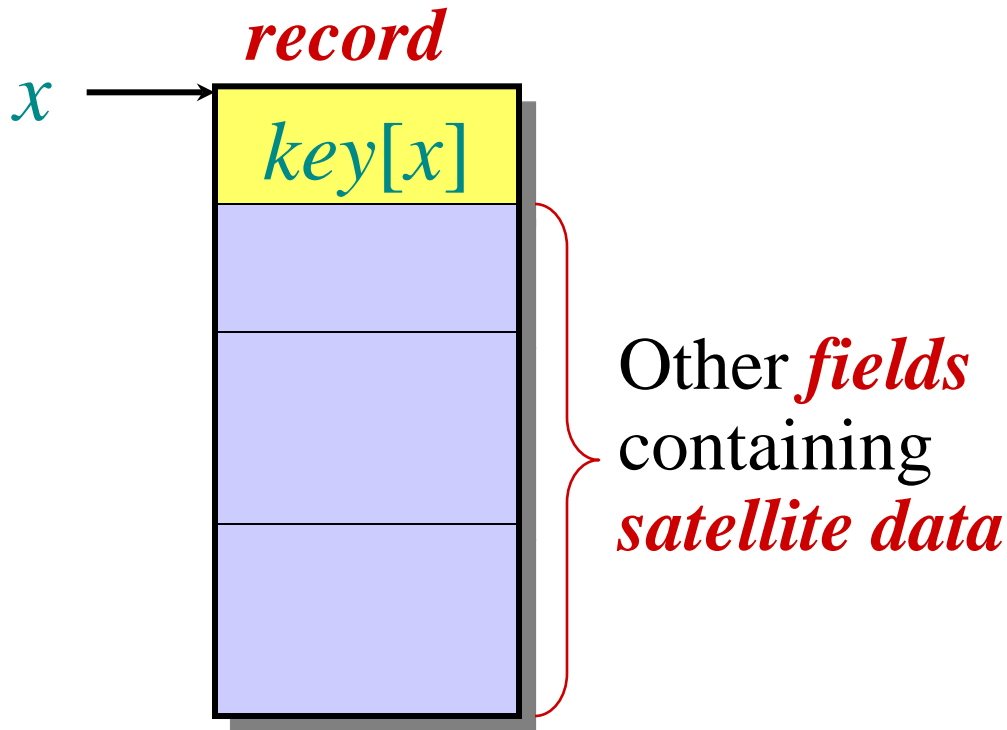
- Direct-access tables
- Resolving collisions by chaining
- Choosing hash functions
- Open addressing

**Assoc. Prof. Dr. Burkay Genç**



# Symbol-table problem

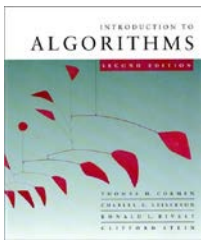
Symbol table  $S$  holding  $n$  *records*:



Operations on  $S$ :

- INSERT( $S, x$ )
- DELETE( $S, x$ )
- SEARCH( $S, k$ )

How should the data structure  $S$  be organized?



# Direct-access table

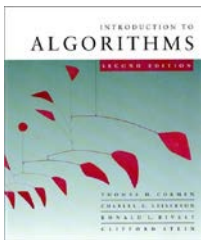
**IDEA:** Suppose that the keys are drawn from the set  $U \subseteq \{0, 1, \dots, m-1\}$ , and keys are distinct. Set up an array  $T[0 \dots m-1]$ :

$$T[k] = \begin{cases} x & \text{if } k \in K \text{ and } \text{key}[x] = k, \\ \text{NIL} & \text{otherwise.} \end{cases}$$

Then, operations take  $\Theta(1)$  time.

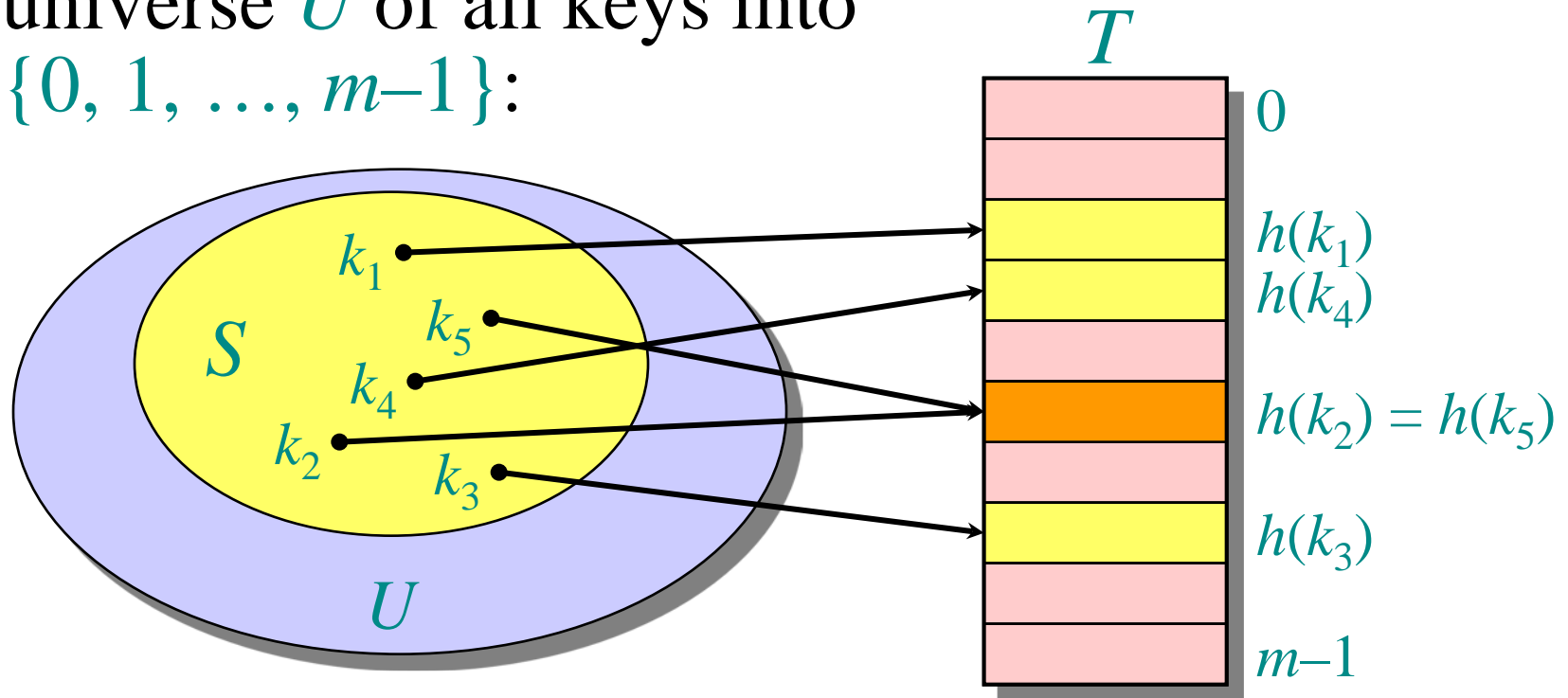
**Problem:** The range of keys can be large:

- 64-bit numbers (which represent 18,446,744,073,709,551,616 different keys),
- character strings (even larger!).

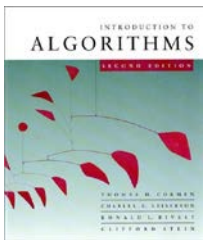


# Hash functions

**Solution:** Use a *hash function*  $h$  to map the universe  $U$  of all keys into  $\{0, 1, \dots, m-1\}$ :

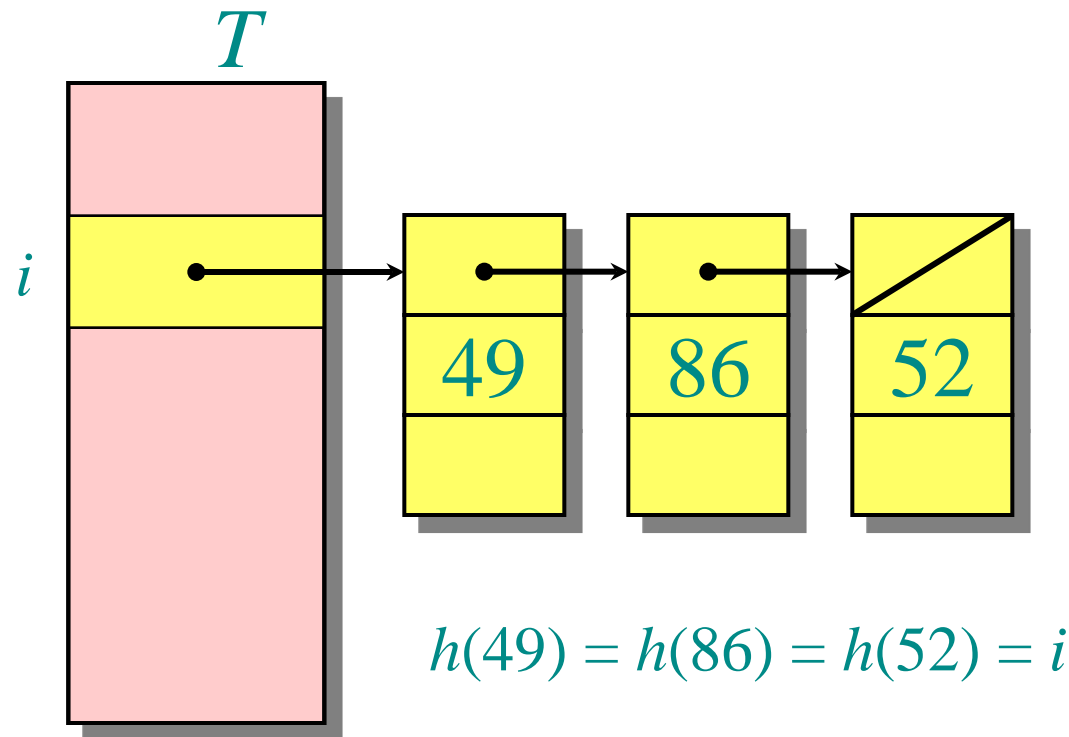


When a record to be inserted maps to an already occupied slot in  $T$ , a *collision* occurs.



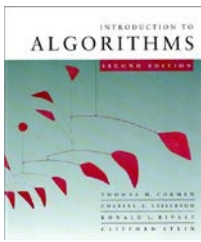
# Resolving collisions by chaining

- Link records in the same slot into a list.



## *Worst case:*

- Every key hashes to the same slot.
- Access time =  $\Theta(n)$  if  $|S| = n$



# Average-case analysis of chaining

We make the assumption of *simple uniform hashing*:

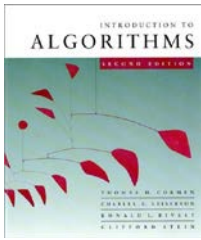
- Each key  $k \in S$  is equally likely to be hashed to any slot of table  $T$ , independent of where other keys are hashed.

Let  $n$  be the number of keys in the table, and let  $m$  be the number of slots.

Define the *load factor* of  $T$  to be

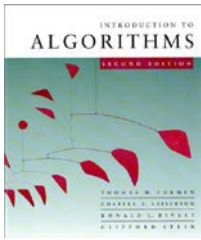
$$\alpha = n/m$$

= average number of keys per slot.



# Search cost

The expected time for an *unsuccessful* search for a record with a given key is  $= \Theta(1 + \alpha)$ .



# Search cost

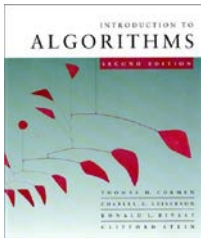
The expected time for an *unsuccessful* search for a record with a given key is

$$= \Theta(1 + \alpha).$$

*search  
the list*

*apply hash function  
and access slot*





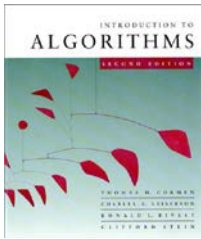
# Search cost

The expected time for an *unsuccessful* search for a record with a given key is

=  $\Theta(1 + \alpha)$ . *search the list*

*apply hash function and access slot*

Expected search time =  $\Theta(1)$  if  $\alpha = O(1)$ , or equivalently, if  $n = O(m)$ .



# Search cost

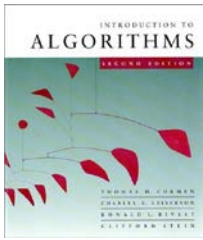
The expected time for an *unsuccessful* search for a record with a given key is

$= \Theta(1 + \alpha)$ . *search the list*

*apply hash function and access slot*

Expected search time  $= \Theta(1)$  if  $\alpha = O(1)$ , or equivalently, if  $n = O(m)$ .

A *successful* search has same asymptotic bound, but a rigorous argument is a little more complicated. (See textbook.)

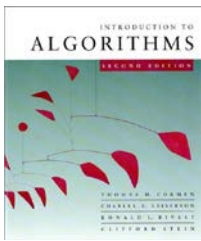


# Choosing a hash function

The assumption of simple uniform hashing is hard to guarantee, but several common techniques tend to work well in practice as long as their deficiencies can be avoided.

## Desiderata:

- A good hash function should distribute the keys uniformly into the slots of the table.
- Regularity in the key distribution should not affect this uniformity.



# Division method

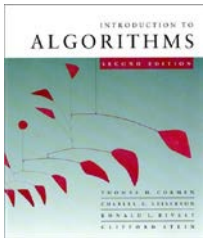
Assume all keys are integers, and define

$$h(k) = k \bmod m.$$

**Deficiency:** Don't pick an  $m$  that has a small divisor  $d$ . A preponderance of keys that are congruent modulo  $d$  can adversely affect uniformity.

**Extreme deficiency:** If  $m = 2^r$ , then the hash doesn't even depend on all the bits of  $k$ :

- If  $k = 1011000111011010_2$  and  $r = 6$ , then  
 $h(k) = 011010_2$ .



# Division method (continued)

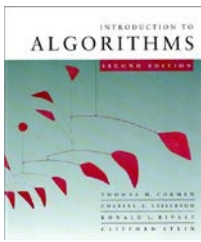
$$h(k) = k \bmod m.$$

Pick  $m$  to be a prime not too close to a power of 2 or 10 and not otherwise used prominently in the computing environment.

## **Annoyance:**

- Sometimes, making the table size a prime is inconvenient.

But, this method is popular, although the next method we'll see is usually superior.



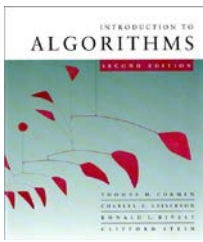
# Multiplication method

Assume that all keys are integers,  $m = 2^r$ , and our computer has  $w$ -bit words. Define

$$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w - r),$$

where **rsh** is the “bitwise right-shift” operator and  $A$  is an odd integer in the range  $2^{w-1} < A < 2^w$ .

- Don't pick  $A$  too close to  $2^{w-1}$  or  $2^w$ .
- Multiplication modulo  $2^w$  is fast compared to division.
- The **rsh** operator is fast.



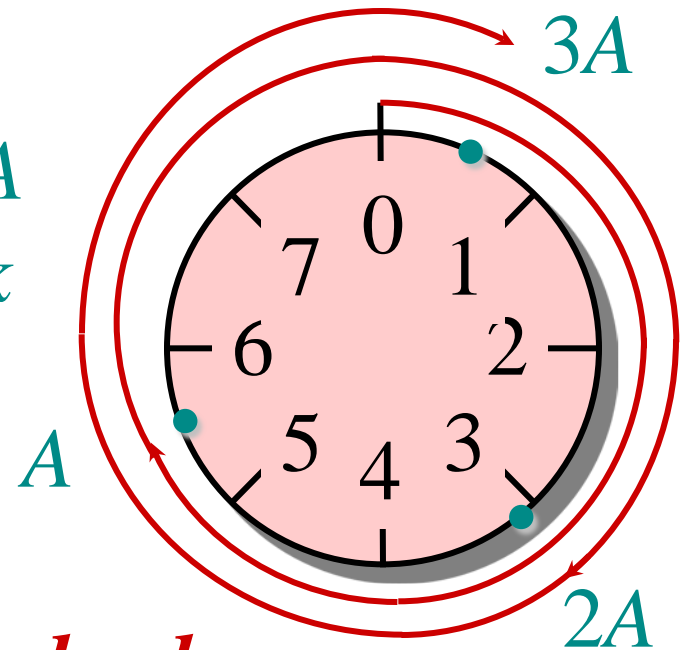
# Multiplication method example

$$h(k) = (A \cdot k \bmod 2^w) \text{ rsh } (w - r)$$

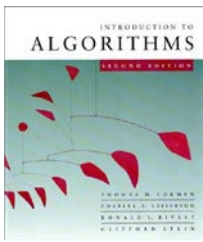
Suppose that  $m = 8 = 2^3$  and that our computer has  $w = 7$ -bit words:

$$\begin{array}{r}
 \phantom{\times} \phantom{10010100} 1011001 = A \\
 \times \phantom{10010100} 1101011 = k \\
 \hline
 10010100 \phantom{0110011} \\
 \phantom{10010100} 0110011 \\
 \hline
 \phantom{10010100} \phantom{0110011} \phantom{0110011} \\
 \phantom{10010100} \phantom{0110011} \phantom{0110011} \phantom{0110011} \\
 \phantom{10010100} \phantom{0110011} \phantom{0110011} \phantom{0110011} \phantom{0110011} \\
 \phantom{10010100} \phantom{0110011} \phantom{0110011} \phantom{0110011} \phantom{0110011} \phantom{0110011} \\
 \hline
 100101000110011
 \end{array}$$

$h(k)$ 
 $A$



*Modular wheel*



# Resolving collisions by open addressing

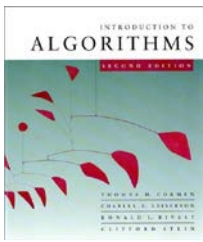
No storage is used outside of the hash table itself.

- Insertion systematically probes the table until an empty slot is found.
- The hash function depends on both the key and probe number:

$$h : U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}.$$

- The probe sequence  $\langle h(k,0), h(k,1), \dots, h(k,m-1) \rangle$  should be a permutation of  $\{0, 1, \dots, m-1\}$ .
- The table may fill up, and deletion is difficult (but not impossible).

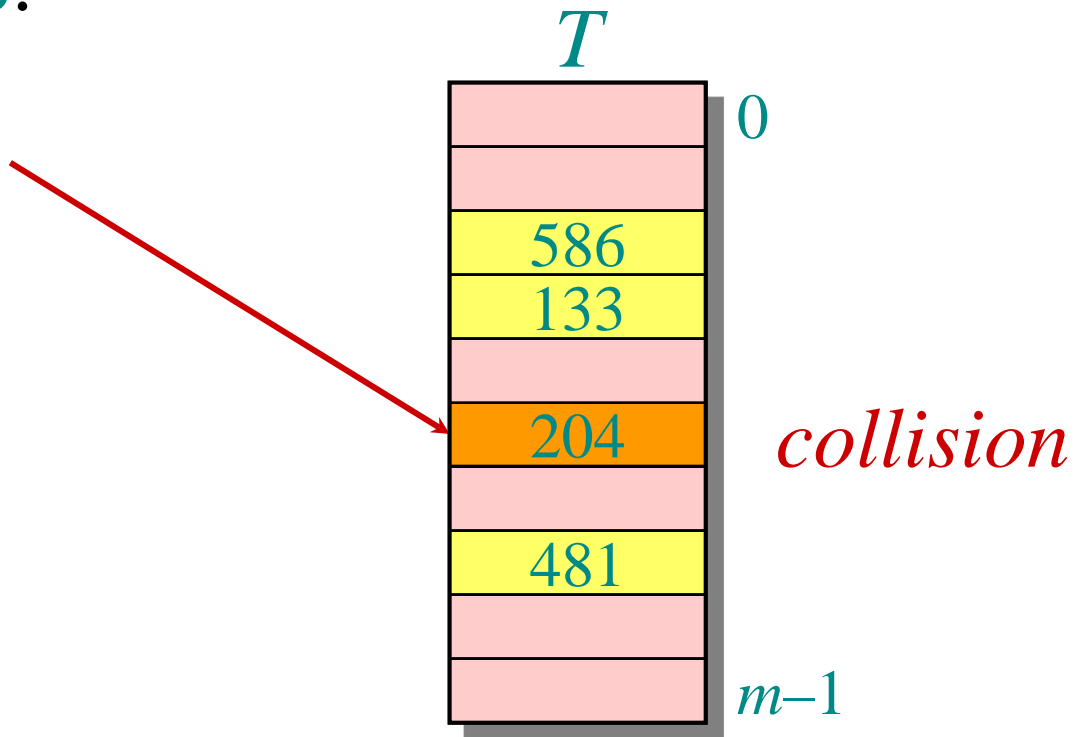


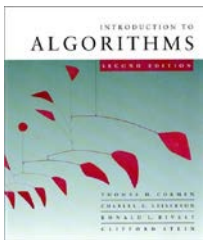


# Example of open addressing

Insert key  $k = 496$ :

0. Probe  $h(496, 0)$



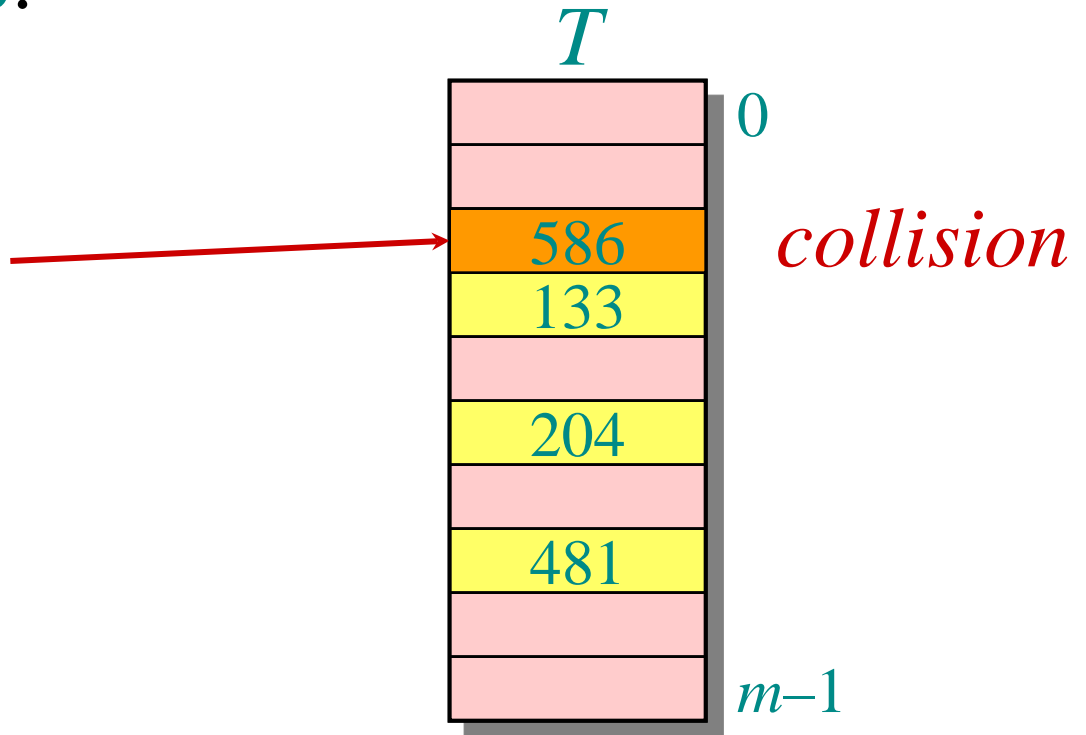


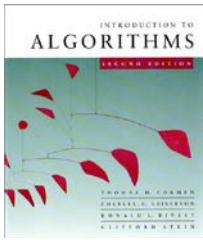
# Example of open addressing

Insert key  $k = 496$ :

0. Probe  $h(496,0)$

1. Probe  $h(496,1)$

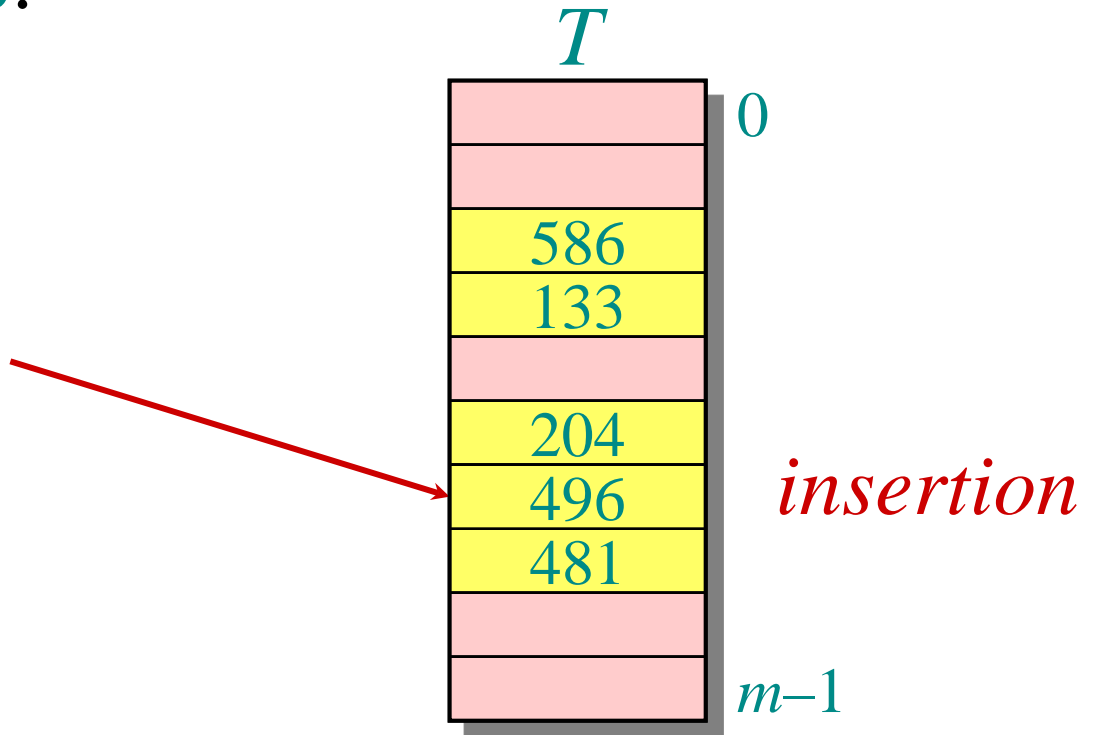


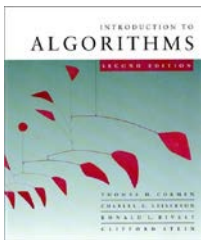


# Example of open addressing

Insert key  $k = 496$ :

0. Probe  $h(496,0)$
1. Probe  $h(496,1)$
2. Probe  $h(496,2)$



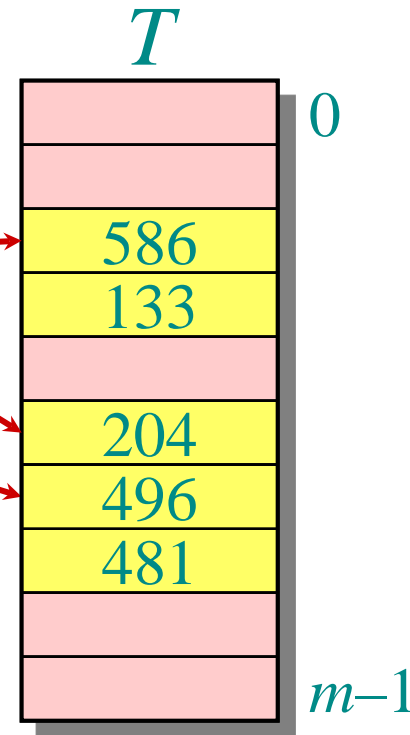


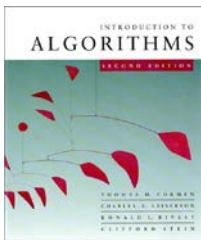
# Example of open addressing

Search for key  $k = 496$ :

0. Probe  $h(496,0)$
1. Probe  $h(496,1)$
2. Probe  $h(496,2)$

Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot.





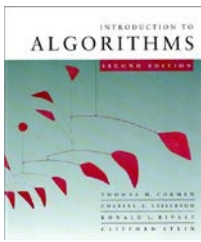
# Probing strategies

## Linear probing:

Given an ordinary hash function  $h'(k)$ , linear probing uses the hash function

$$h(k,i) = (h'(k) + i) \bmod m.$$

This method, though simple, suffers from *primary clustering*, where long runs of occupied slots build up, increasing the average search time. Moreover, the long runs of occupied slots tend to get longer.



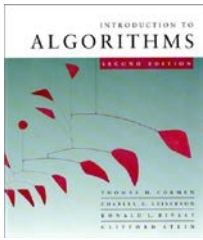
# Probing strategies

## Double hashing

Given two ordinary hash functions  $h_1(k)$  and  $h_2(k)$ , double hashing uses the hash function

$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m.$$

This method generally produces excellent results, but  $h_2(k)$  must be relatively prime to  $m$ . One way is to make  $m$  a power of 2 and design  $h_2(k)$  to produce only odd numbers.

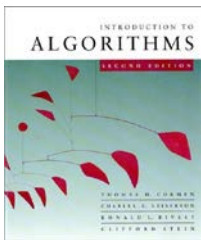


# Analysis of open addressing

We make the assumption of *uniform hashing*:

- Each key is equally likely to have any one of the  $m!$  permutations as its probe sequence.

**Theorem.** Given an open-addressed hash table with load factor  $\alpha = n/m < 1$ , the expected number of probes in an unsuccessful search is at most  $1/(1-\alpha)$ .



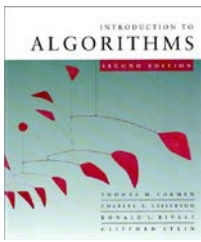
# Proof of the theorem

## *Proof.*

- At least one probe is always necessary.
- With probability  $n/m$ , the first probe hits an occupied slot, and a second probe is necessary.
- With probability  $(n-1)/(m-1)$ , the second probe hits an occupied slot, and a third probe is necessary.
- With probability  $(n-2)/(m-2)$ , the third probe hits an occupied slot, etc.

Observe that  $\frac{n-i}{m-i} < \frac{n}{m} = \alpha$  for  $i = 1, 2, \dots, n$ .





# Proof (continued)

Therefore, the expected number of probes is

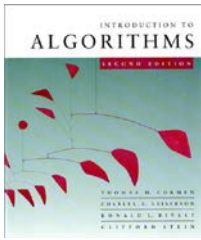
$$1 + \frac{n}{m} \left( 1 + \frac{n-1}{m-1} \left( 1 + \frac{n-2}{m-2} \left( \dots \left( 1 + \frac{1}{m-n+1} \right) \dots \right) \right) \right)$$

$$\leq 1 + \alpha (1 + \alpha (1 + \alpha (\dots (1 + \alpha) \dots)))$$

$$\leq 1 + \alpha + \alpha^2 + \alpha^3 + \dots$$

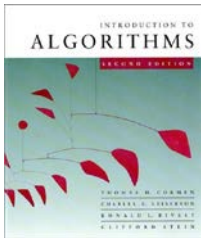
$$= \sum_{i=0}^{\infty} \alpha^i$$
$$= \frac{1}{1-\alpha} \cdot \square$$

*The textbook has a more rigorous proof and an analysis of successful searches.*



# Implications of the theorem

- If  $\alpha$  is constant, then accessing an open-addressed hash table takes constant time.
- If the table is half full, then the expected number of probes is  $1/(1-0.5) = 2$ .
- If the table is 90% full, then the expected number of probes is  $1/(1-0.9) = 10$ .



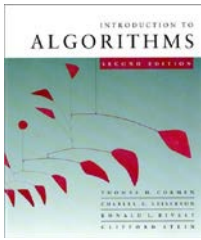
# A weakness of hashing

**Problem:** For any hash function  $h$ , a set of keys exists that can cause the average access time of a hash table to skyrocket.

- An adversary can pick all keys from  $\{k \in U : h(k) = i\}$  for some slot  $i$ .

**IDEA:** Choose the hash function at random, independently of the keys.

- Even if an adversary can see your code, he or she cannot find a bad set of keys, since he or she doesn't know exactly which hash function will be chosen.



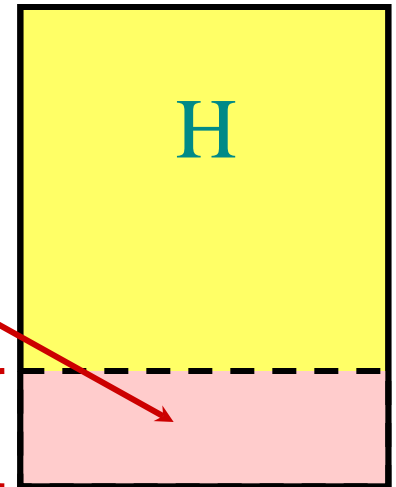
# Universal hashing

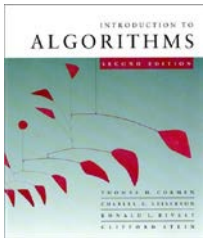
**Definition.** Let  $U$  be a universe of keys, and let  $H$  be a finite collection of hash functions, each mapping  $U$  to  $\{0, 1, \dots, m-1\}$ . We say  $H$  is *universal* if for all  $x, y \in U$ , where  $x \neq y$ , we have  $|\{h \in H : h(x) = h(y)\}| \leq |H|/m$ .

That is, the chance of a collision between  $x$  and  $y$  is  $\leq 1/m$  if we choose  $h$  randomly from  $H$ .

$$\{h : h(x) = h(y)\}$$

$$\frac{|H|}{m}$$

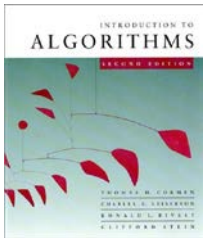




# Universality is good

**Theorem.** Let  $h$  be a hash function chosen (uniformly) at random from a universal set  $\mathbf{H}$  of hash functions. Suppose  $h$  is used to hash  $n$  arbitrary keys into the  $m$  slots of a table  $T$ . Then, for a given key  $x$ , we have

$$E[\text{\#collisions with } x] < n/m.$$

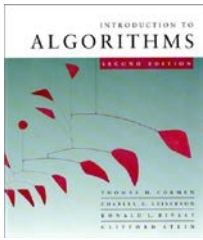


# Proof of theorem

*Proof.* Let  $C_x$  be the random variable denoting the total number of collisions of keys in  $T$  with  $x$ , and let

$$c_{xy} = \begin{cases} 1 & \text{if } h(x) = h(y), \\ 0 & \text{otherwise.} \end{cases}$$

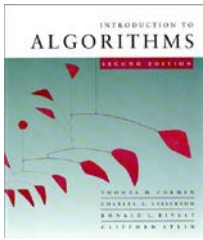
*Note:*  $E[c_{xy}] = 1/m$  and  $C_x = \sum_{y \in T - \{x\}} c_{xy}$ .



# Proof (continued)

$$E[C_x] = E \left[ \sum_{y \in T - \{x\}} c_{xy} \right]$$

- Take expectation of both sides.

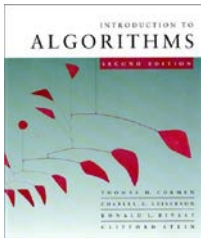


# Proof (continued)

$$\begin{aligned} E[C_x] &= E \left[ \sum_{y \in T - \{x\}} c_{xy} \right] \\ &= \sum_{y \in T - \{x\}} E[c_{xy}] \end{aligned}$$

- Take expectation of both sides.
- Linearity of expectation.

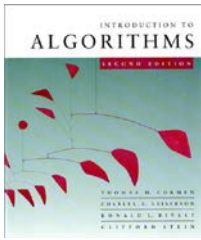




# Proof (continued)

$$\begin{aligned} E[C_x] &= E \left[ \sum_{y \in T - \{x\}} c_{xy} \right] \\ &= \sum_{y \in T - \{x\}} E[c_{xy}] \\ &= \sum_{y \in T - \{x\}} 1/m \end{aligned}$$

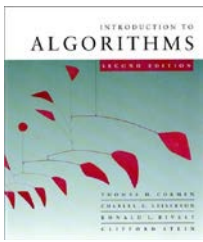
- Take expectation of both sides.
- Linearity of expectation.
- $E[c_{xy}] = 1/m$ .



# Proof (continued)

$$\begin{aligned} E[C_x] &= E \left[ \sum_{y \in T - \{x\}} c_{xy} \right] \\ &= \sum_{y \in T - \{x\}} E[c_{xy}] \\ &= \sum_{y \in T - \{x\}} 1/m \\ &= \frac{n-1}{m} . \quad \square \end{aligned}$$

- Take expectation of both sides.
- Linearity of expectation.
- $E[c_{xy}] = 1/m$ .
- Algebra.



# Constructing a set of universal hash functions

Let  $m$  be prime. Decompose key  $k$  into  $r + 1$  digits, each with value in the set  $\{0, 1, \dots, m-1\}$ . That is, let  $k = \langle k_0, k_1, \dots, k_r \rangle$ , where  $0 \leq k_i < m$ .

## Randomized strategy:

Pick  $a = \langle a_0, a_1, \dots, a_r \rangle$  where each  $a_i$  is chosen randomly from  $\{0, 1, \dots, m-1\}$ .

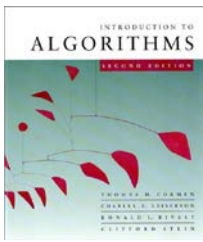
Define  $h_a(k) = \sum_{i=0}^r a_i k_i \bmod m$ .

*Dot product,  
modulo  $m$*

How big is  $H = \{h_a\}$ ?

$$|H| = m^{r+1}$$

**REMEMBER  
THIS!**



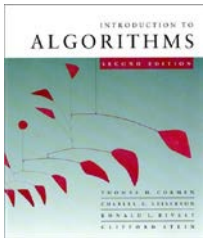
# Universality of dot-product hash functions

**Theorem.** The set  $H = \{h_a\}$  is universal.

*Proof.* Suppose that  $x = \langle x_0, x_1, \dots, x_r \rangle$  and  $y = \langle y_0, y_1, \dots, y_r \rangle$  be distinct keys. Thus, they differ in at least one digit position, wlog position 0. For how many  $h_a \in H$  do  $x$  and  $y$  collide?

We must have  $h_a(x) = h_a(y)$ , which implies that

$$\sum_{i=0}^r a_i x_i \equiv \sum_{i=0}^r a_i y_i \pmod{m}.$$



# Proof (continued)

Equivalently, we have

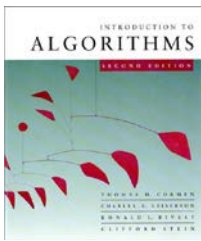
$$\sum_{i=0}^r a_i(x_i - y_i) \equiv 0 \pmod{m}$$

or

$$a_0(x_0 - y_0) + \sum_{i=1}^r a_i(x_i - y_i) \equiv 0 \pmod{m},$$

which implies that

$$a_0(x_0 - y_0) \equiv -\sum_{i=1}^r a_i(x_i - y_i) \pmod{m}.$$



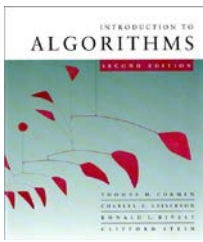
# Fact from number theory

**Theorem.** Let  $m$  be prime. For any  $z \in \mathbb{Z}_m$  such that  $z \neq 0$ , there exists a unique  $z^{-1} \in \mathbb{Z}_m$  such that

$$z \cdot z^{-1} \equiv 1 \pmod{m}.$$

**Example:**  $m = 7$ .

$z$	1	2	3	4	5	6
$z^{-1}$	1	4	5	2	3	6



# Back to the proof

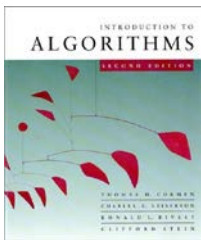
We have

$$a_0(x_0 - y_0) \equiv -\sum_{i=1}^r a_i(x_i - y_i) \pmod{m},$$

and since  $x_0 \neq y_0$ , an inverse  $(x_0 - y_0)^{-1}$  must exist, which implies that

$$a_0 \equiv \left( -\sum_{i=1}^r a_i(x_i - y_i) \right) \cdot (x_0 - y_0)^{-1} \pmod{m}.$$

Thus, for any choices of  $a_1, a_2, \dots, a_r$ , exactly one choice of  $a_0$  causes  $x$  and  $y$  to collide.



# Proof (completed)

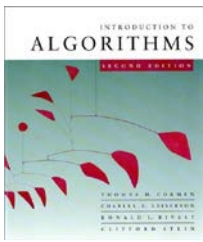
**Q.** How many  $h_a$ 's cause  $x$  and  $y$  to collide?

**A.** There are  $m$  choices for each of  $a_1, a_2, \dots, a_r$ , but once these are chosen, exactly one choice for  $a_0$  causes  $x$  and  $y$  to collide, namely

$$a_0 = \left( \left( - \sum_{i=1}^r a_i (x_i - y_i) \right) \cdot (x_0 - y_0)^{-1} \right) \bmod m.$$

Thus, the number of  $h_a$ 's that cause  $x$  and  $y$  to collide is  $m^r \cdot 1 = m^r = |\mathbf{H}|/m$ . □



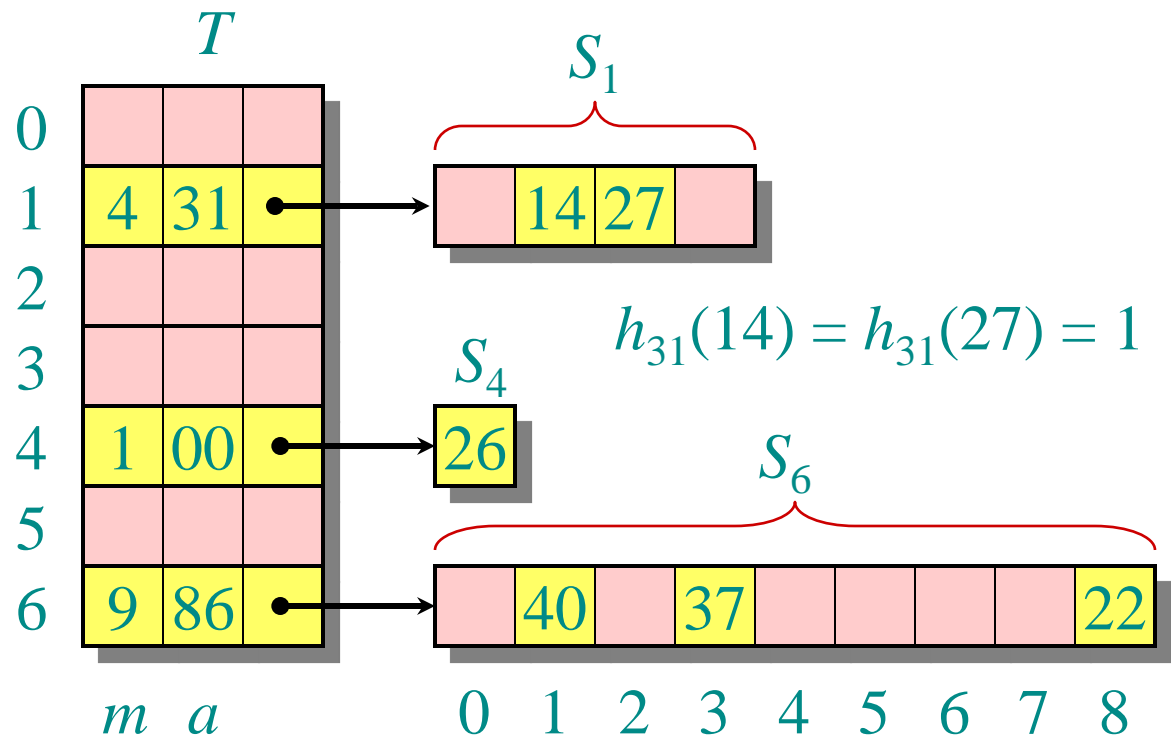


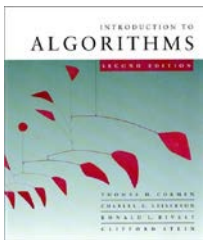
# Perfect hashing

Given a set of  $n$  keys, construct a static hash table of size  $m = O(n)$  such that **SEARCH** takes  $\Theta(1)$  time in the *worst case*.

**IDEA:** Two-level scheme with universal hashing at both levels.

*No collisions at level 2!*



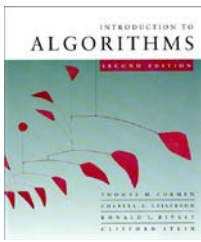


# Collisions at level 2

**Theorem.** Let  $\mathbf{H}$  be a class of universal hash functions for a table of size  $m = n^2$ . Then, if we use a random  $h \in \mathbf{H}$  to hash  $n$  keys into the table, the expected number of collisions is at most  $1/2$ .

*Proof.* By the definition of universality, the probability that 2 given keys in the table collide under  $h$  is  $1/m = 1/n^2$ . Since there are  $\binom{n}{2}$  pairs of keys that can possibly collide, the expected number of collisions is

$$\binom{n}{2} \cdot \frac{1}{n^2} = \frac{n(n-1)}{2} \cdot \frac{1}{n^2} < \frac{1}{2} \quad \square$$



# No collisions at level 2

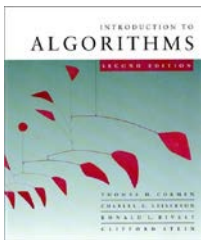
**Corollary.** The probability of no collisions is at least  $1/2$ .

*Proof. Markov's inequality* says that for any nonnegative random variable  $X$ , we have

$$\Pr\{X \geq t\} \leq E[X]/t.$$

Applying this inequality with  $t = 1$ , we find that the probability of 1 or more collisions is at most  $1/2$ . ■

*Thus, just by testing random hash functions in  $H$ , we'll quickly find one that works.*



# Analysis of storage

For the level-1 hash table  $T$ , choose  $m = n$ , and let  $n_i$  be random variable for the number of keys that hash to slot  $i$  in  $T$ . By using  $n_i^2$  slots for the level-2 hash table  $S_i$ , the expected total storage required for the two-level scheme is therefore

$$E \left[ \sum_{i=0}^{m-1} \Theta(n_i^2) \right] = \Theta(n),$$

since the analysis is identical to the analysis from recitation of the expected running time of bucket sort. (For a probability bound, apply Markov.)