

# *Introduction to Algorithms*

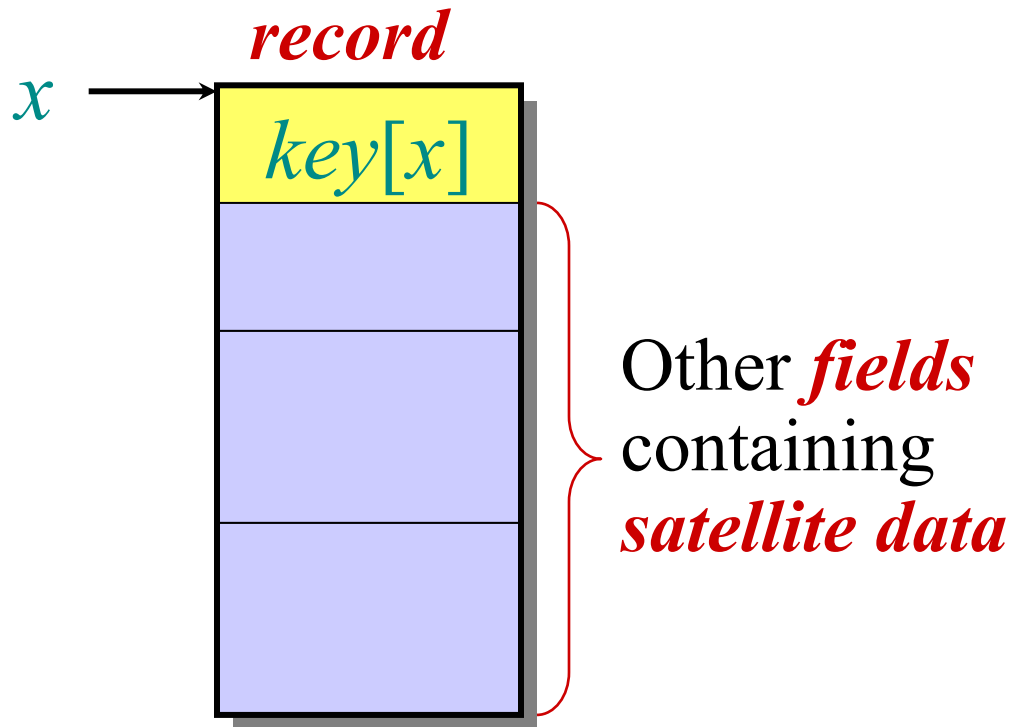
6.046J/18.401J/SMA5503

## *Lecture 7*

**Prof. Charles E. Leiserson**

# Symbol-table problem

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



Operations on  $T$ :

- $\text{INSERT}(T, x)$
- $\text{DELETE}(T, x)$
- $\text{SEARCH}(T, k)$

How should the data structure  $T$  be organized?

# Direct-access table

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

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

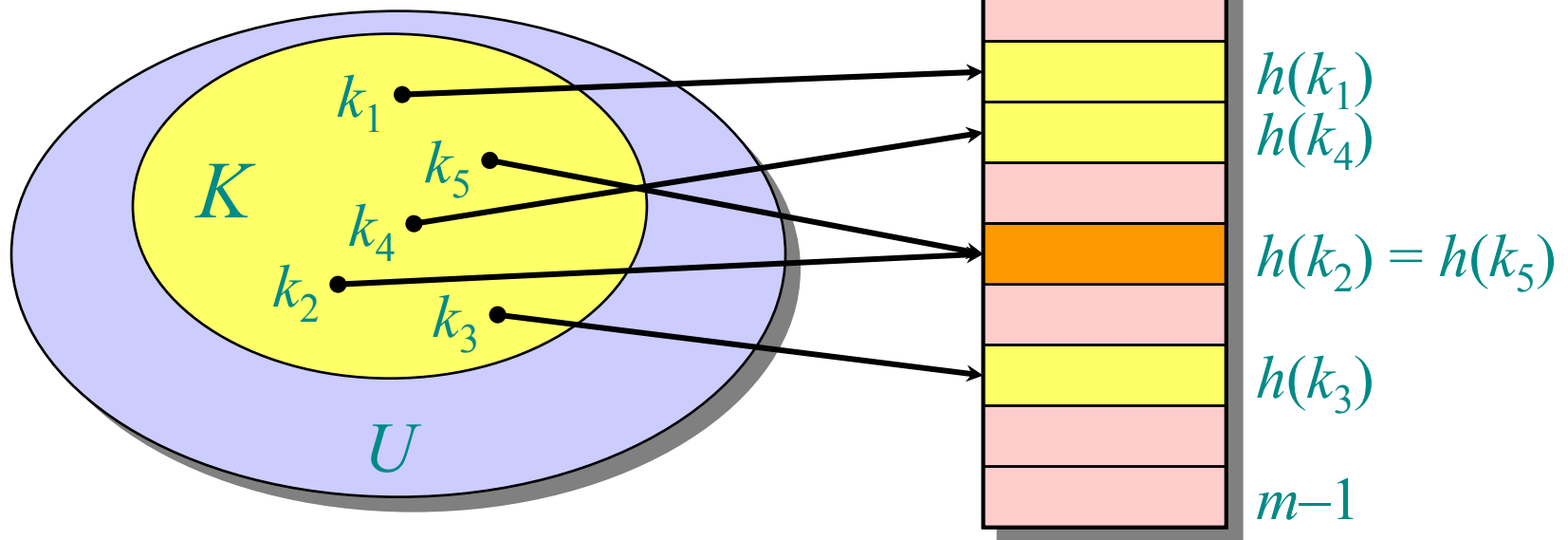
Then, operations take  $\Theta(1)$  time.

**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

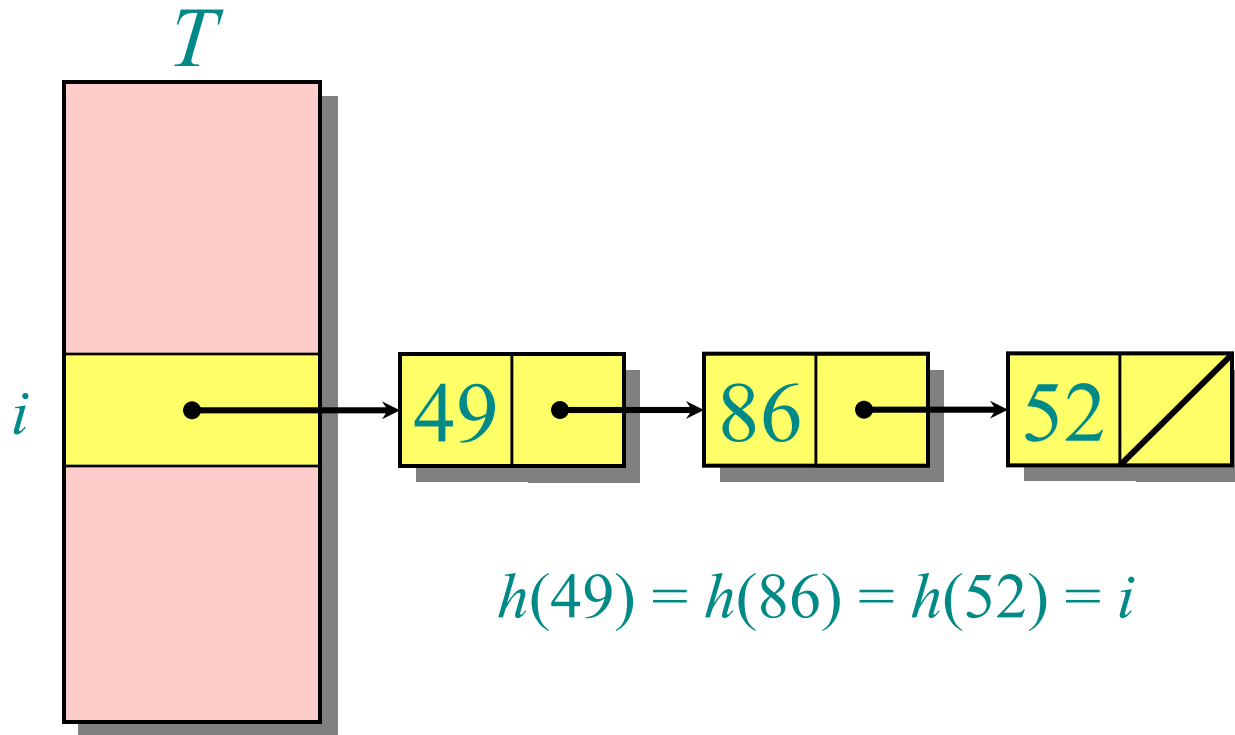
**Solution:** Use a *hash function*  $h$  to map the universe  $U$  of all keys into  $\{0, 1, \dots, m-1\}$ :



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

# Resolving collisions by chaining

- Records in the same slot are linked into a list.



# Analysis of chaining

We make the assumption of *simple uniform hashing*:

- Each key  $k \in K$  of keys 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

Expected time to search for a record with a given key =  $\Theta(1 + \alpha)$ .



*apply hash  
function and  
access slot*

*search  
the list*

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

# 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.

## Desirata:

- 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  $k = 1011000111\underbrace{011010}_2$  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 “bit-wise right-shift” operator and  $A$  is an odd integer in the range  $2^{w-1} < A < 2^w$ .

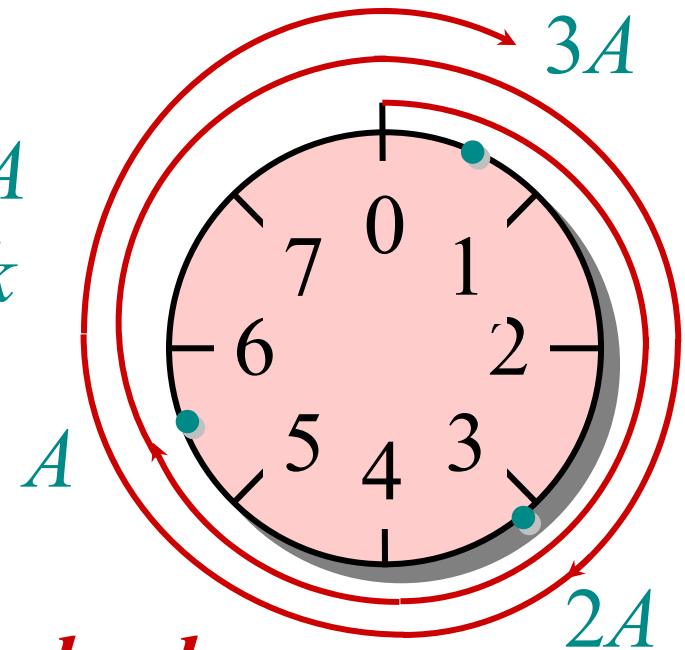
- Don't pick  $A$  too close to  $2^w$ .
- Multiplication modulo  $2^w$  is fast.
- The **rsh** operator is fast.

# 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:

$$\begin{array}{r}
 \phantom{\times} \phantom{10010100} 1011001 = A \\
 \times \phantom{10010100} 1101011 = k \\
 \hline
 10010100 \phantom{00110011} \\
 \phantom{10010100} \underbrace{0110011}_{h(k)}
 \end{array}$$



*Modular wheel*

# Dot-product method

## Randomized strategy:

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  $k = \langle k_0, k_1, \dots, k_{m-1} \rangle$ , where  $0 \leq k_i < m$ . Pick  $a = \langle a_0, a_1, \dots, a_{m-1} \rangle$  where each  $a_i$  is chosen randomly from  $\{0, 1, \dots, m-1\}$ .

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

- Excellent in practice, but expensive to compute.

# 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:

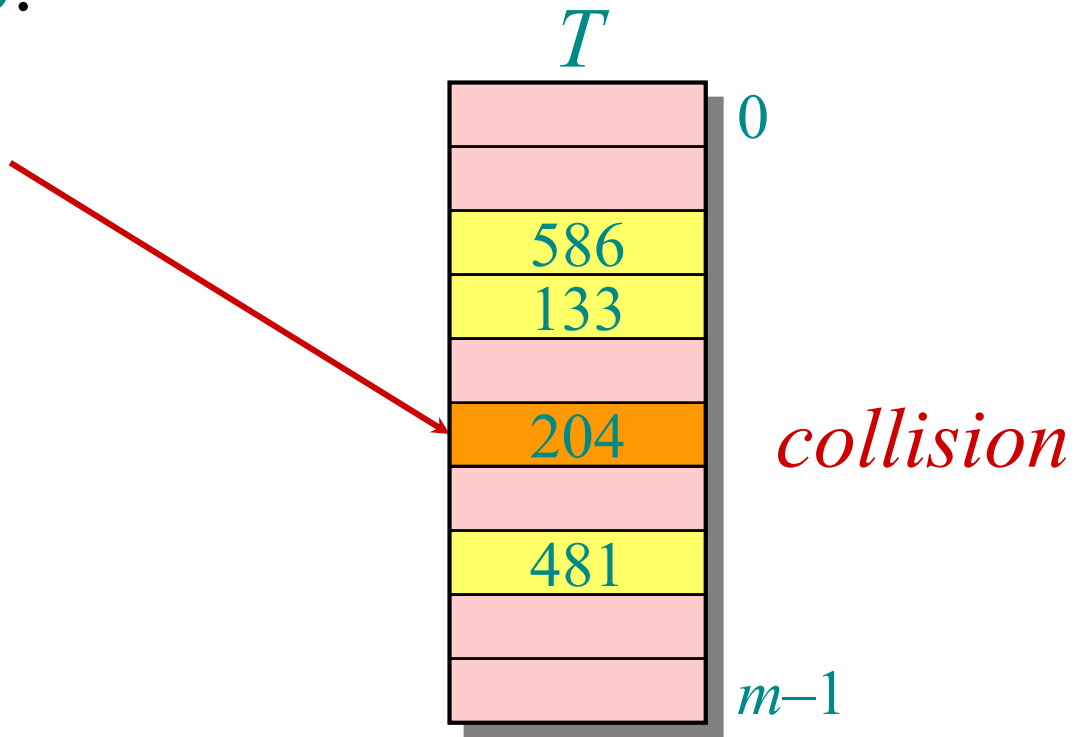
$$h : U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}.$$

- 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  $h(496, 0)$

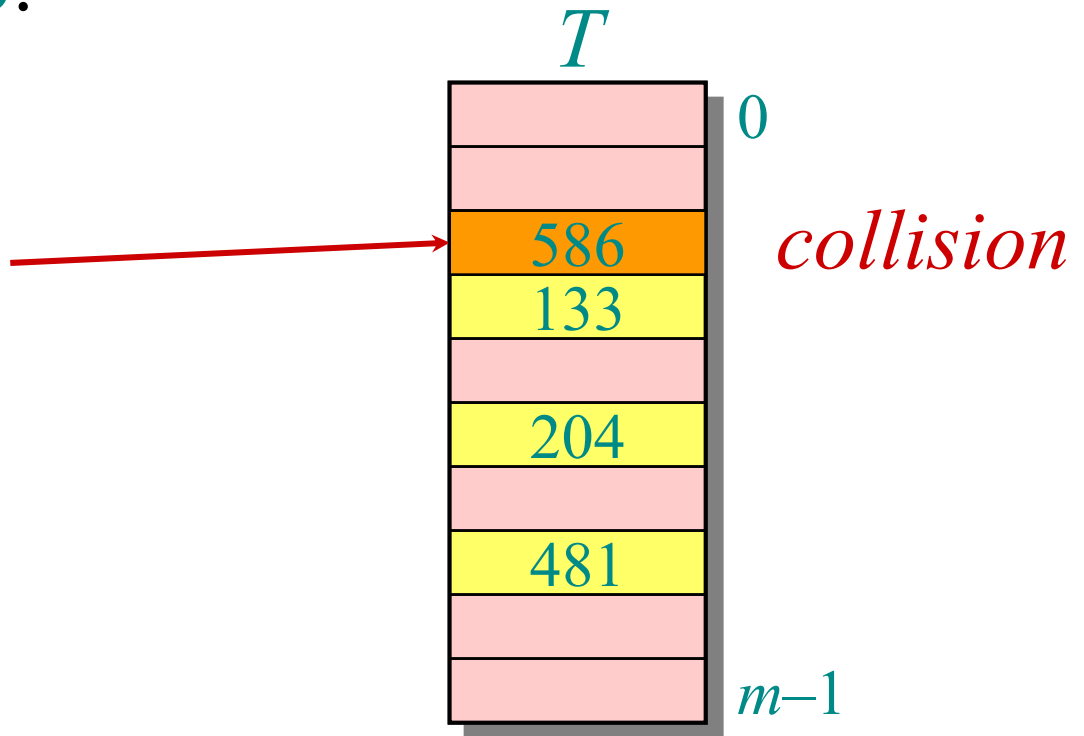


# Example of open addressing

Insert key  $k = 496$ :

0. Probe  $h(496, 0)$

1. Probe  $h(496, 1)$

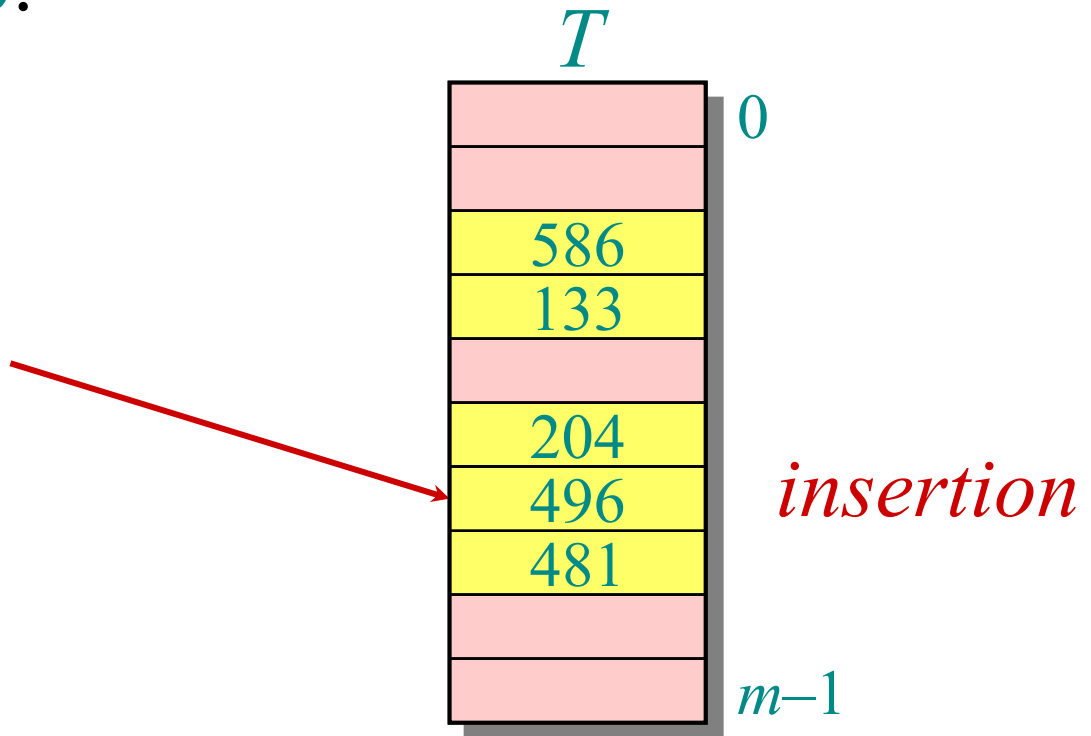




# Example of open addressing

Insert key  $k = 496$ :

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



# Example of open addressing

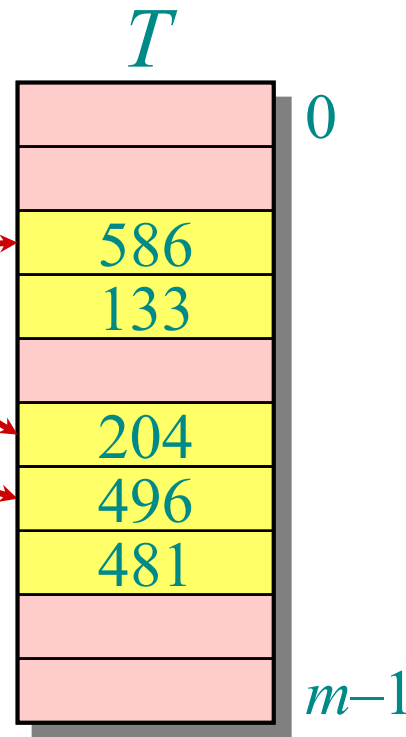
Search for key  $k = 496$ :

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

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

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

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

$$= \sum_{i=0}^{\infty} \alpha^i$$

$$= \frac{1}{1 - \alpha} . \quad \square$$

*The textbook has a more rigorous proof.*

# 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  $1/(1-0.9) = 10$ .