# Hashing

# Plan

- "Classical" hashing
  - hashing by chaining
  - hashing by open addressing
- Universal hashing
- Perfect hashing (quick review)
- Cuckoo hashing
- Bloom filters
- Locality-sensitive hashing

# Example 1: path finding

- Assume you want to implement A* shortest path search on a graph of $1000$ nodes. You can allocate an array of size $1000$ to store distances
- What about search on Rubik's cube graph (order of $10^6$ for $2\times2\times2$ cube, $10^{19}$ for $3\times3\times3$ cube)?

# Example 2: data bases

- Maintain a set of employees (students, messenger users, …), each identified by a social security number (student ID, phone number, …)

# Example 3: deduplication

- In a programming language compiler, how to store user-declared identifiers?

- Construct an index of a book with all terms pointing to their first occurrence in the book

# Hash tables: suppored operations

- A generalization of arrays (“direct addressing”)
- *Goal*: maintain a (possibly evolving) set of objects belonging to a large “universe” (e.g. configurations, ID numbers, words, etc.)
- *Applications*: deduplication, indexing, path finding, file integrity test (checksum), etc.

# Hash tables: suppored operations

- A generalization of arrays (“direct addressing”)
- *Goal*: maintain a (possibly evolving) set of objects belonging to a large “universe” (e.g. configurations, ID numbers, words, etc.)
- *Applications*: deduplication, indexing, path finding, file integrity test (checksum), etc.

- INSERT: add a new object
- DELETE: delete existing object
- LOOKUP: check for an object

possibly specified by a *key*
"associative array"

“Dictionary” data structure

# Naive solutions

- Bit array (bitmap)
  - still too big for huge applications
  - does not support access to objects
  - BUT … (cf Bloom filters at the end of this lecture)

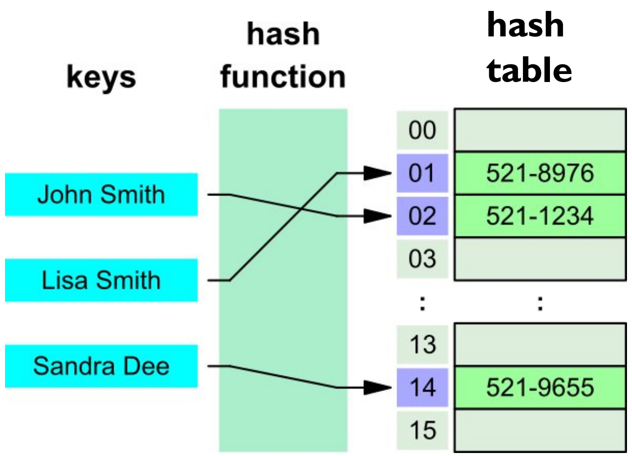
- Linked list
  - look-up too slow

- Search trees
  - better but still slow and memory demanding

# Hash tables

- Notation
  - $U$ : universe of all possible keys (*Ex*: strings, IP addresses, game configurations, …)
  - $K$ : subset of keys (actually stored in the dictionary), $|K| \ll |U|$
  - $|K| = n$

- Use a table of size proportional to $|K|$: hash table
  - we lose the direct-addressing ability
  - hash function maps keys to entries of the hash table (**buckets** or slots)



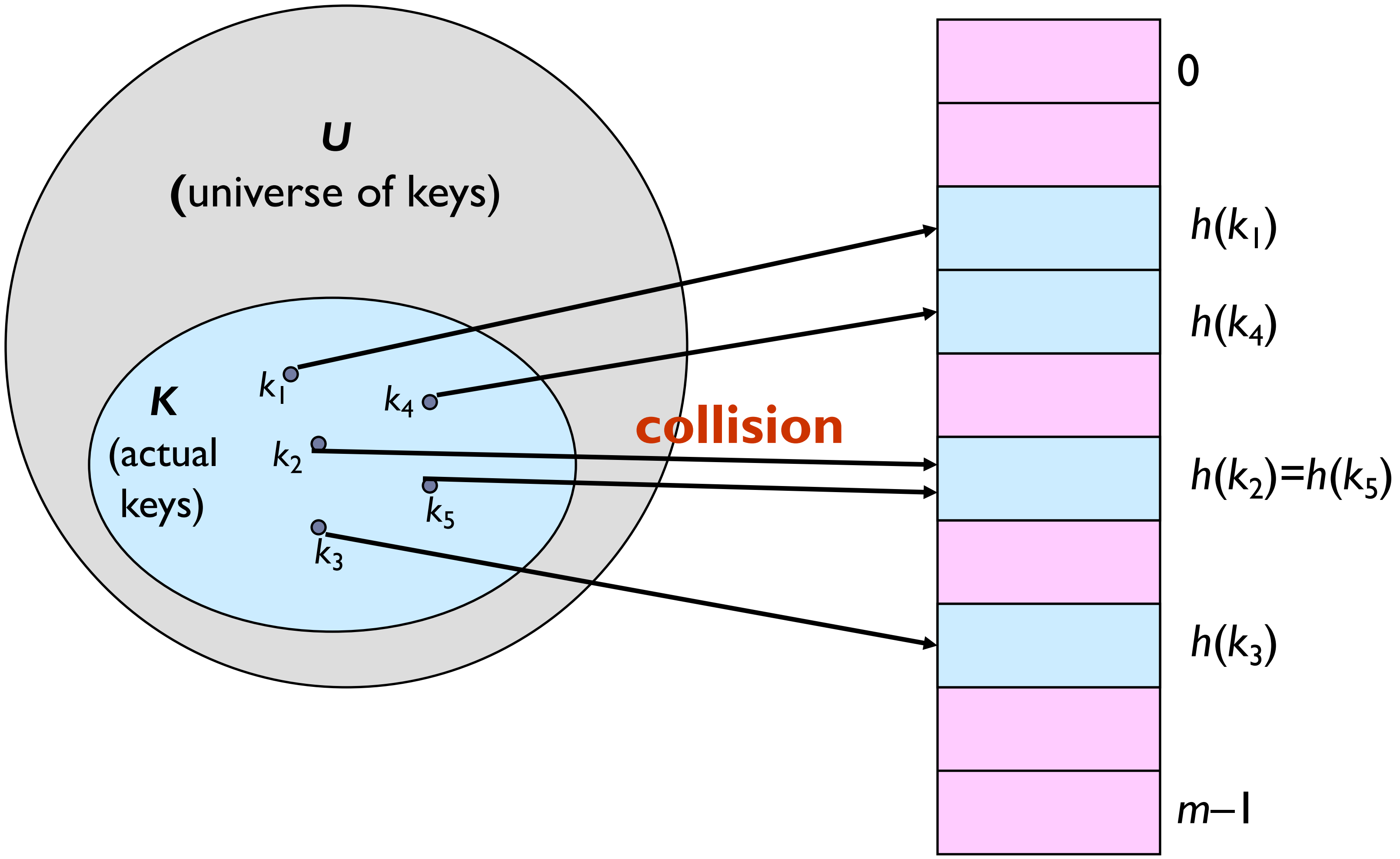
# Hash functions

- Hash function $h$: Mapping from $U$ to the slots of a hash table $T[0..m-1]$.

$$h : U \rightarrow \{0,1,\ldots,m-1\}$$

- With direct addressing, key $k$ maps to slot $A[k]$
- With hash tables, key $k$ maps or "hashes" to bucket $T[h[k]]$
- $h[k]$ is the *hash value* (or simpy *hash*) of key $k$

# Hashing and collisions



# Collisions: birthday "paradox"

- What is the probability that two people from this class (100 students) have their birthday the same day?

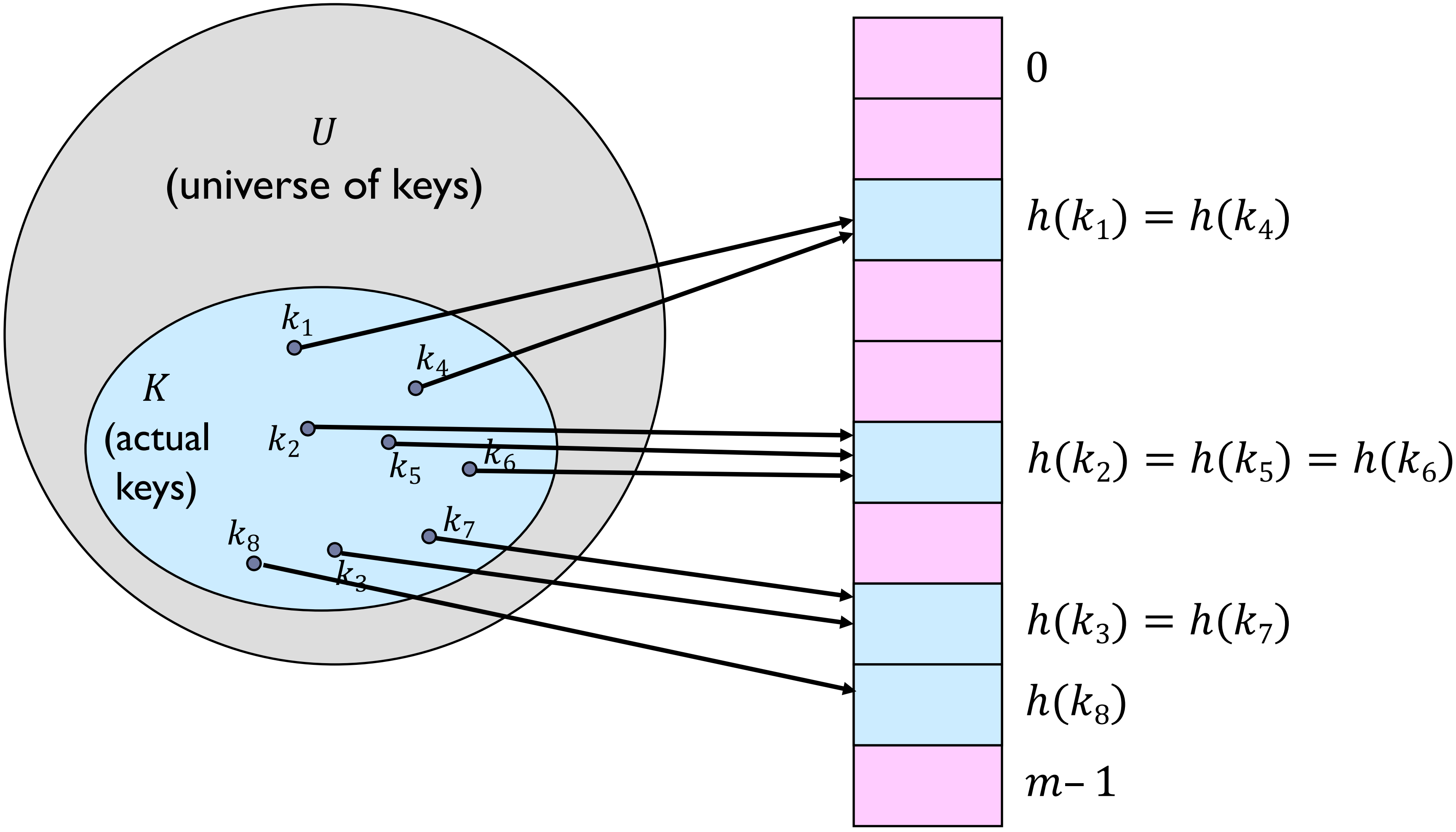
# Collisions: birthday "paradox"

- What is the probability that two people from this class (100 students) have their birthday the same day?
- Answer: ≈0.9999997

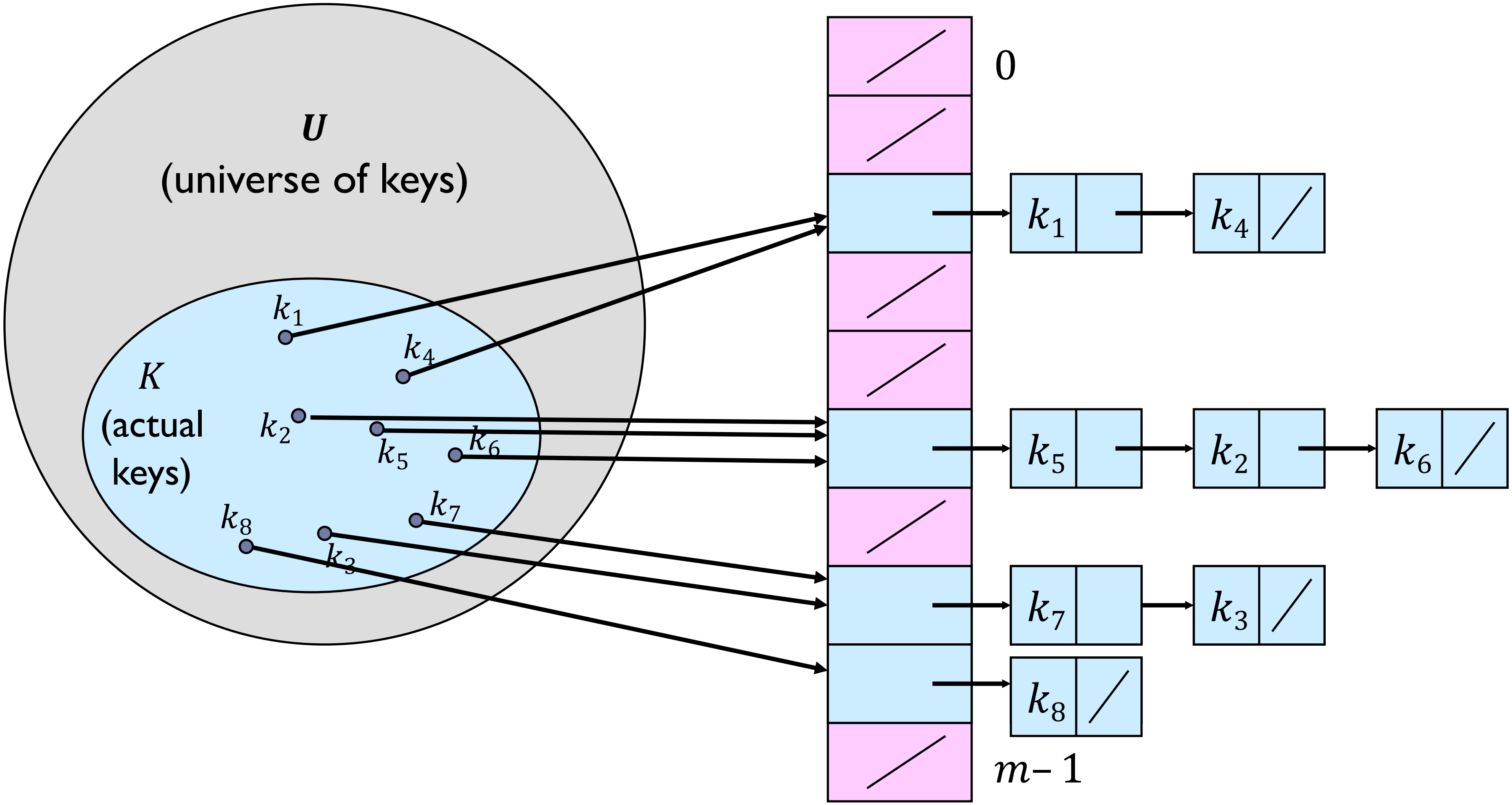
- Birthday paradox: in a group of 23 people, there is about 50% chance that two people have the same bithday
- 40 people: 89%, 60 people: 99.4%

- *Conclusion*: collisions are frequent

# I. Collision Resolution by Chaining



# Collision Resolution by Chaining



# Hashing with chaining

- INSERT$(T.k)$ : $O(1)$
- DELETE$(T.k)$, LOOKUP$(k)$: $O(list\ length)$

- $\Rightarrow$ a good hash function should distribute keys into buckets *as uniformly as possible*

- random hashing $\Rightarrow$ expected list length is $\alpha = n/m$ (*load factor*)

- the *average* time of DELETE and LOOKUP is $O(1+\alpha)$ $\Rightarrow O(1)$ if $n = O(m)$ (practical case)

# Good hash functions

- Hash function should be easy to compute
- Desiging good hash functions is tricky. It is easy to design a bad hash function

- *Examples*: Phone numbers. Benford's law (e.g. prices, population sizes, …)

- Keys are usually considered as natural numbers
- *Example*: Interpret a character string as an integer expressed in some radix notation. Suppose the string is CLRS:
  - ASCII values: C=67, L=76, R=82, S=83.
  - There are 128 basic ASCII values.
  - So, CLRS = $67 \cdot 128^3 + 76 \cdot 128^2 + 82 \cdot 128^1 + 83 \cdot 128^0 = 141{,}764{,}947$

# Division Method

- Map a key $k$ into one of the $m$ slots by taking the remainder of $k$ divided by $m$. That is,

$$h(k) = k \, mod \, m$$

- *Example:* $m = 31$ and $k = 78 \Rightarrow h(k) = 16$

- *Advantage:* Fast, since requires just one division operation
- *Disadvantage:* Have to avoid certain values of $m$.
  - Don't pick certain values, such as $m = 2^p$ (as the hash won't depend on all bits of $k$)

- *Good choice for $m$:*
  - Primes, not too close to power of 2 (or 10) are good.

# Multiplication Method

- If $0 < A < 1$, $h(k) = \lfloor m\ (kA\ mod\ 1) \rfloor = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ where $(kA\ mod\ 1) = kA - \lfloor kA \rfloor$: the fractional part of $kA$
- *Disadvantage:* Slower than the division method.
- *Advantage:* Value of $m$ is not critical.
  - Typically chosen as a power of 2, i.e., $m = 2^p$, which makes the implementation easy

- *Example:* $m = 1000, k = 123,\ A = 0.6180339887\ldots$

$h(k) = \lfloor 1000(123 \cdot\ 0.6180339887\ mod\ 1) \rfloor = \lfloor 1000 \cdot 0.018169\ldots \rfloor = 18$

# II. Collision Resolution by Open Addressing

- All elements are stored in the hash table itself
- $\Rightarrow \quad n \leq m, \quad$ no pointers

- hash function $h(k, i)$ where $i = 0,1,2, \dots, m-1, \quad$ and $< h(k,0), h(k,1), \dots, h(k, m-1) >$ is a permutation
- when inserting/looking up $k$, probe $h(k,0), h(k,1), \dots$ (*probe sequence*) until
  - we find $k$, or
  - the bucket contains $nil$, or
  - $m$ buckets have been unsuccessfully probed
- deletion is complicated, needs a special key "deleted", time may not be dependent on the load factor

# Open Addressing: Linear probing

- The colliding item is placed in a different cell of the table
- **Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell $h(k,i) = (h'(k) + i) \bmod m$
- Each table cell inspected is referred to as a "probe"
- Colliding items clump together, causing future collisions to cause a longer sequence of probes

- *Example:*
  - $h'(k) = k \bmod 13$
  - Insert keys $18, 41, 22, 44, 59, 32, 31, 73,$ in this order

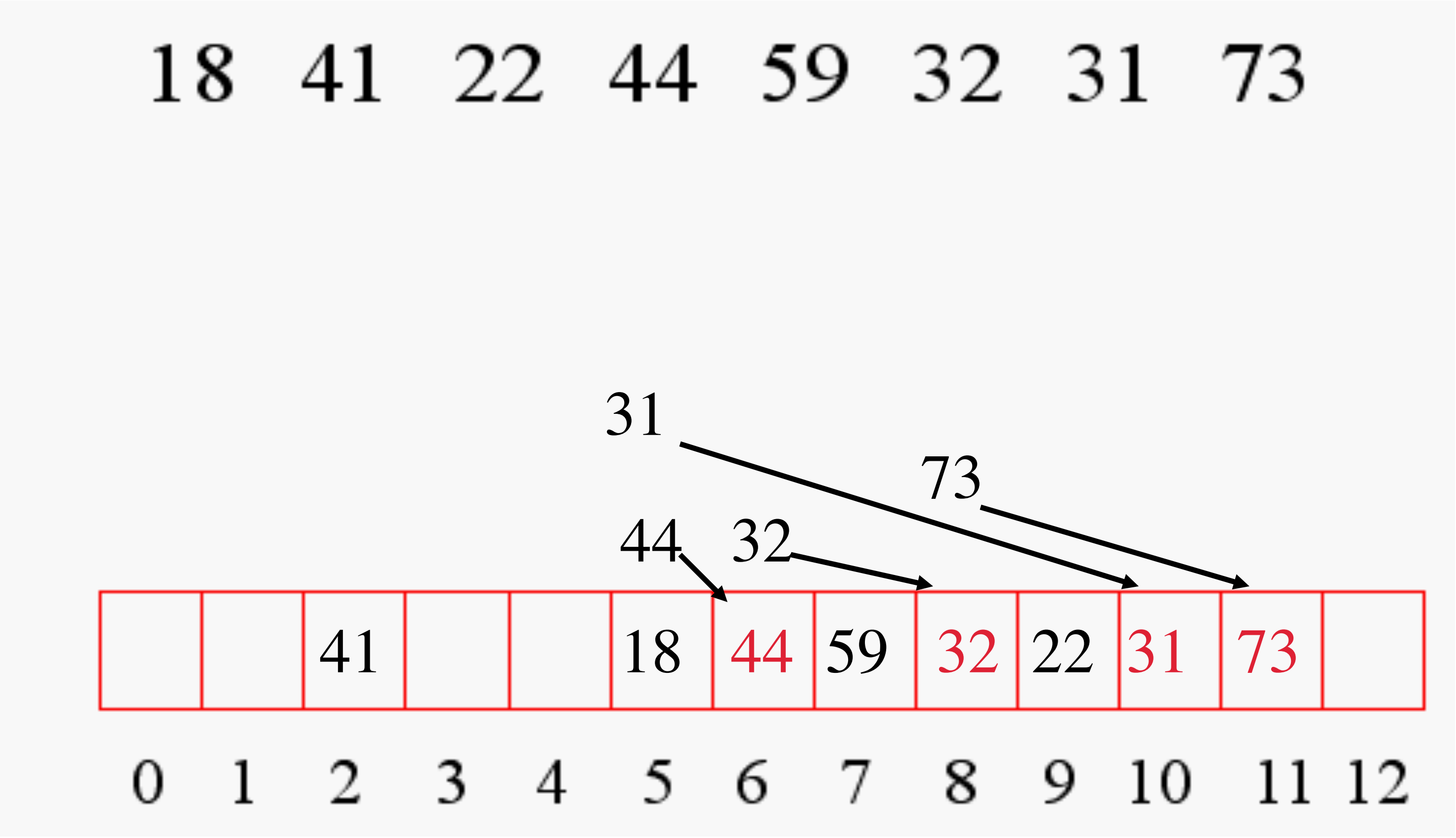
| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

⇩

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 41 | | | 18 | 44 | 59 | 32 | 22 | 31 | 73 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

# Example (cont.)

$$h'(k) = k \bmod 13$$



# Quadratic probing

- $h(k,j) = (h'(k) + c_1 \cdot j + c_2 \cdot\ j^2) \bmod m$

- for example, $h(k,j) = (h'(k) + \frac{1}{2} \cdot j + \frac{1}{2} \cdot j^2) \bmod m$

  $h(k,0), h(k,1), \dots, h(k,m-1)$ is a permutation if $m$ is a power of 2

- quadratic probing works better than linear probing (less clumping)

# Double Hashing

- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
  $$h(k,j) = (h(k) + jd(k)) \bmod m$$
  for $j = 0,1,\dots,m-1$

- The secondary hash function $d(k)$ cannot have zero values

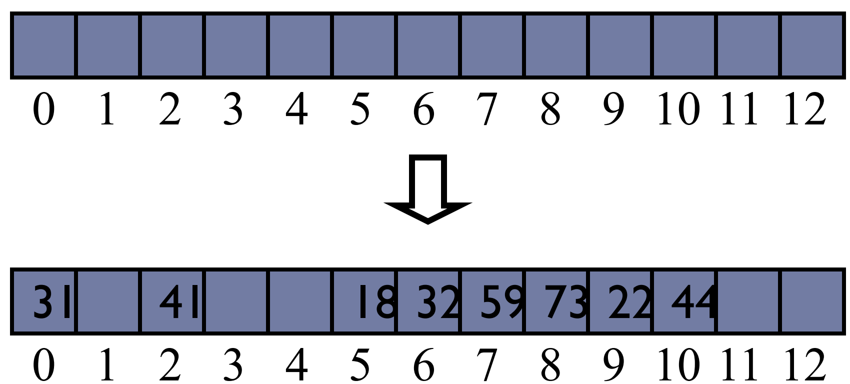
- $m$ should be relatively prime to $d(k)$, e.g. $m = 2^q$ and $d(k)$ odd, or $m$ is prime and $d(k) < m$

- Double hashing is usually more efficient than linear and quadratic probing

# Example of Double Hashing

- Consider a hash table storing integer keys that handles collision with double hashing
  - $m = 13$
  - $h(k) = k \bmod 13$
  - $d(k) = 7 - k \bmod 7$
- Insert keys $18, 41, 22, 44, 59, 32, 31, 73$, in this order

| $k$ | $h(k)$ | $d(k)$ | Probes | | |
|---|---|---|---|---|---|
| 18 | 5 | 3 | 5 | | |
| 41 | 2 | 1 | 2 | | |
| 22 | 9 | 6 | 9 | | |
| 44 | 5 | 5 | 5 | 10 | |
| 59 | 7 | 4 | 7 | | |
| 32 | 6 | 3 | 6 | | |
| 31 | 5 | 4 | 5 | 9 | 0 |
| 73 | 8 | 4 | 8 | | |



# Performance of Open Addressing

- Assuming that $< h(k,0), h(k,1), \dots, h(k,m-1) >$ is a random permutation (uniformly drawn), the expected number of probes in an insertion (or unsuccessful search) with open addressing is

  $1/(1-\alpha)$,

  where $\alpha = n/m$ the load factor

- *Explanation*:

  let $p_i = \mathrm{P}[i \text{ first buckets are full}] = \alpha^i \quad (p_0 = 0)$

  $E[\text{number of probes}] = 1 + \sum_{i=1..m-1} i \cdot P[i \text{ full buckets followed by an empty one}] = 1 + \sum_{i=1..m-1} i \cdot (p_{i-1} - p_i) = 1 + \sum_{i=1..m-1} p_i \approx$

  $1 + \alpha + \alpha^2 + \alpha^3 + \cdots = 1/(1-\alpha)$

# Performance of Open Addressing

- Assuming that $< h(k,0), h(k,1), \dots, h(k,m-1) >$ is a random permutation (uniformly drawn), the expected number of probes in an insertion (or unsuccessful search) with open addressing is

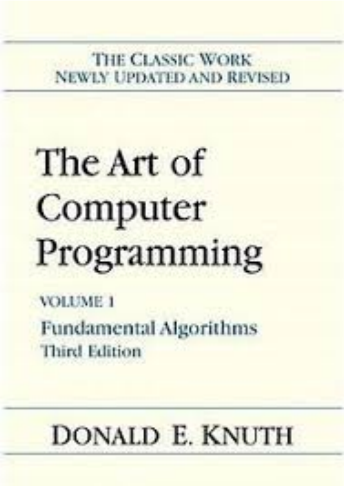
  $$1/(1-\alpha),$$

  where $\alpha = n/m$ the load factor

- The expected number of probes for a successful search is

  $$(1/\alpha)\log(1/(1-\alpha))$$

# Historical remarks




Hashing by open addressing:
Analysed by Donald Knuth in 1962 (invention attributed to Andrei Ershov)

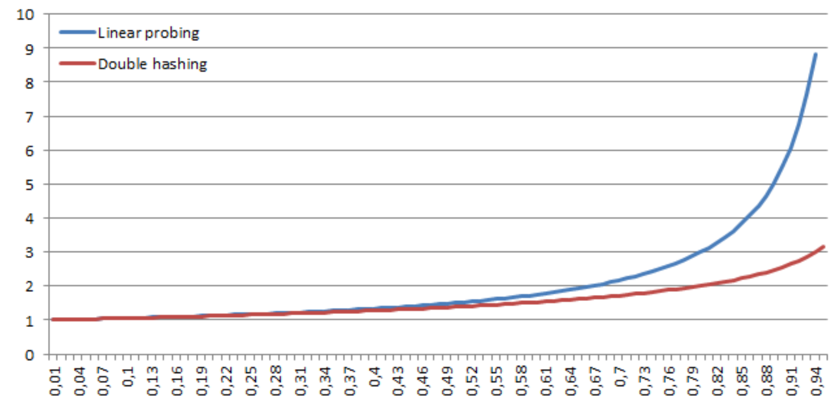
# Hashing: some conclusions

- Chaining:
  - easy implementation
  - fast in practice
  - uses more memory
- Open addressing:
  - uses less memory
  - more complex removals

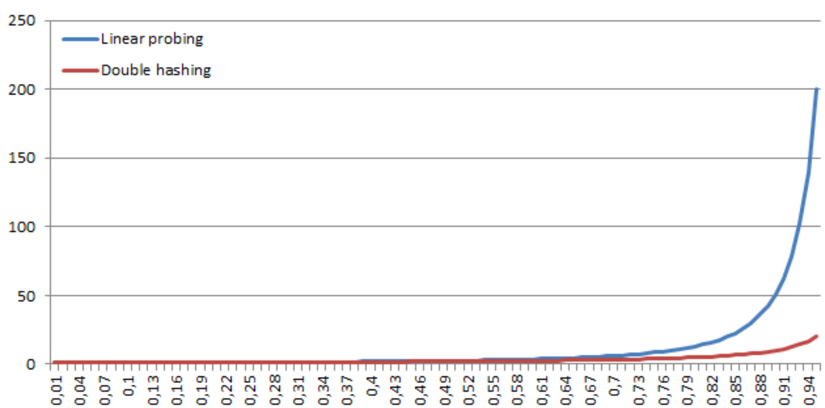
- Implemented in standard libraries, e.g. `std::unordered_map` in C++

# Linear probing vs Double hashing

comparison of average number of operations




successful search




unsuccessful search

# Exercise

- Consider inserting the keys $10, 22, 31, 4, 15, 28, 17, 88, 59$ into a hash table of length $m = 11$ using open addressing with the primary hash function $h'(k) = k \bmod m$. Illustrate the result of inserting these keys using linear probing, using quadratic probing with $c_1 = 1$ and $c_2 = 3$, and using double hashing with $h_2(k) = 1 + (k \bmod (m-1))$ .