TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Hashing

Algorithms, Data and Security
A.Y. 2025/26

Valeria Cardellini

Global Governance, 3rd year
Science and Technology Major

What is hashing?

- **Hashing** is a powerful technique (both algorithm and data structure) primarily used for efficient data storage and retrieval
- It allows us to map input data of variable length to fixed-size values efficiently
- Widely used in many types of software: databases, caches, …

- Hash: from French hacher (“to chop”), derived from Old French hache (“axe”)

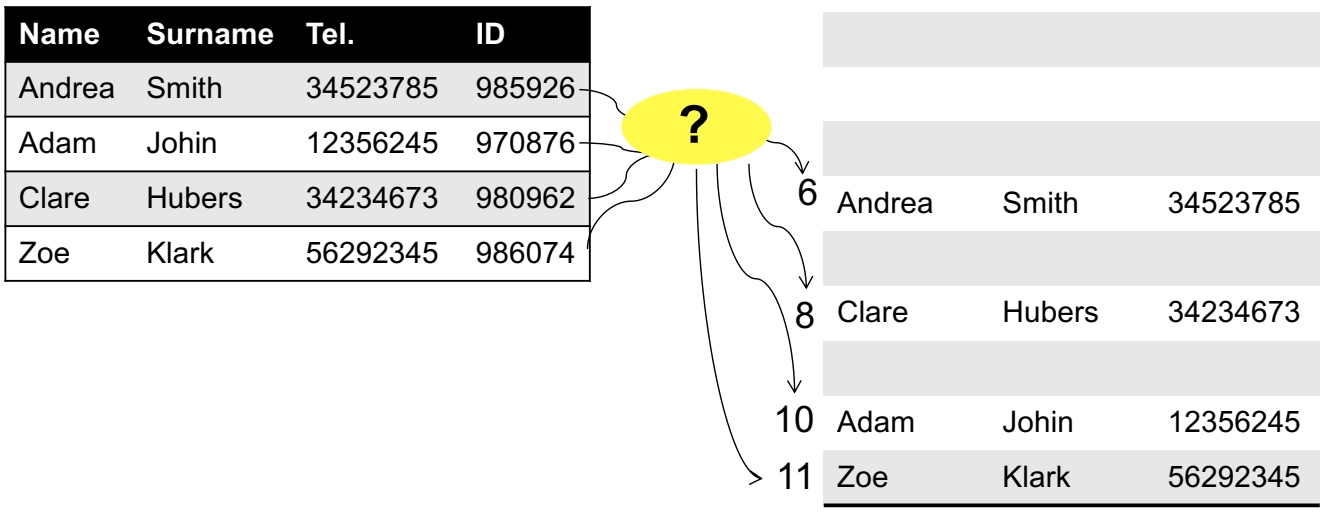
Real-world examples of hashing

- University: each student has a unique roll number that can be used to quickly retrieve their information
- Phone book: contains name, address, and phone number as fields. To find a phone number, you search using the person's name
- Instagram account: each account has a username and password. Logging with these takes you to your personal profile with all your data

What is hashing?

- Catalogue of student's ID

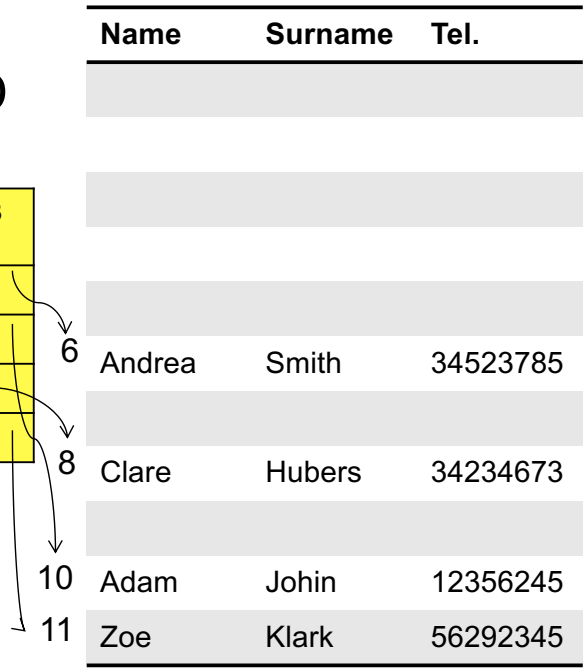
Name	Surname	Tel.	ID
Andrea	Smith	34523785	985926
Adam	Johin	12356245	970876
Clare	Hubers	34234673	980962
Zoe	Klark	56292345	986074



What is hashing?

- Catalogue of student's ID

Name	Surname	Tel.	ID	ID mod 13
Andrea	Smith	34523785	985926	**6**
Adam	Johin	12356245	970876	**10**
Clare	Hubers	34234673	980962	**8**
Zoe	Klark	56292345	986074	**11**



Why we need hashing

- Many applications handle lots of data
- There are myriad of data accesses which require data lookups
- Lookups are time-critical operations
- Data structures like arrays or lists are not sufficient to handle efficient lookups
  - Unsorted arrays require O(n) time to search for a value (recall sequential search algorithm)
- In general: we use hashing whenever we need near-constant-time lookups, i.e., O(1)

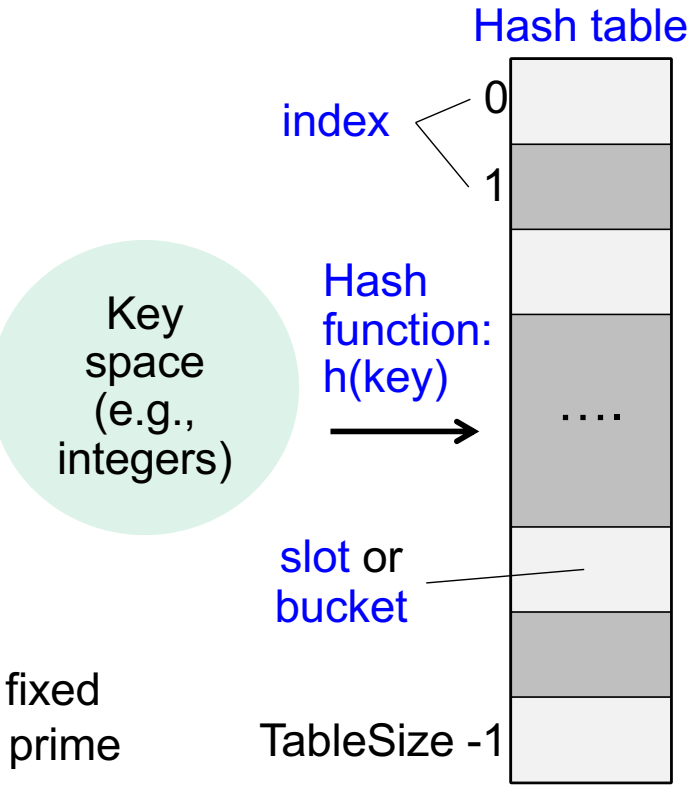
Why we need hashing

Operation	Unsorted array	Sorted array	Hashing
lookup	O(n)	O(log n)	O(1)
insert	O(1)	O(n)	O(1)
delete	O(n)	O(n)	O(1)

- Unsorted array (size n)
  - Lookup: sequential search → O(n)
  - Insert: insert at end → O(1)
  - Delete: search element + remove it → O(n)
- Sorted array (size n)
  - Lookup: binary search → O(log n)
  - Insert: find position + shift elements → O(n)
  - Delete: find position + shift elements → O(n)
- Ideal implementation: hash table
  - Lookup: compute position → O(1) on average
  - Insert: compute position + place element → O(1) on average
  - Delete: compute position + remove element → O(1) on average

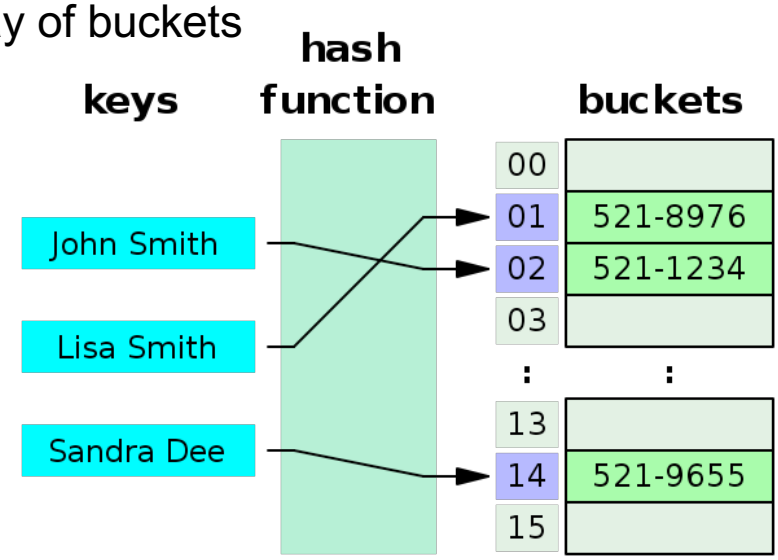
Hash table

- A **hash table** (or hash map) is a data structure that maps keys to values, for efficient search and retrieval
- It uses a hash function to compute an *index* into an array of *buckets* or *slots*, where the corresponding is stored
- Constant time access!
- A hash table is an array of fixed size (TableSize), usually a prime number



Hash table: example

- Phone book using a hash table
  - Key: person's name
  - Value: Phone number
  - Hash function: converts the name into an index in an array of buckets



Hash table: operations

- Search (or lookup)

  `lookup(key)`: find the value associated with the given key
- Insertion

  `insert(key, value)`: add the new value to the table
- Deletion

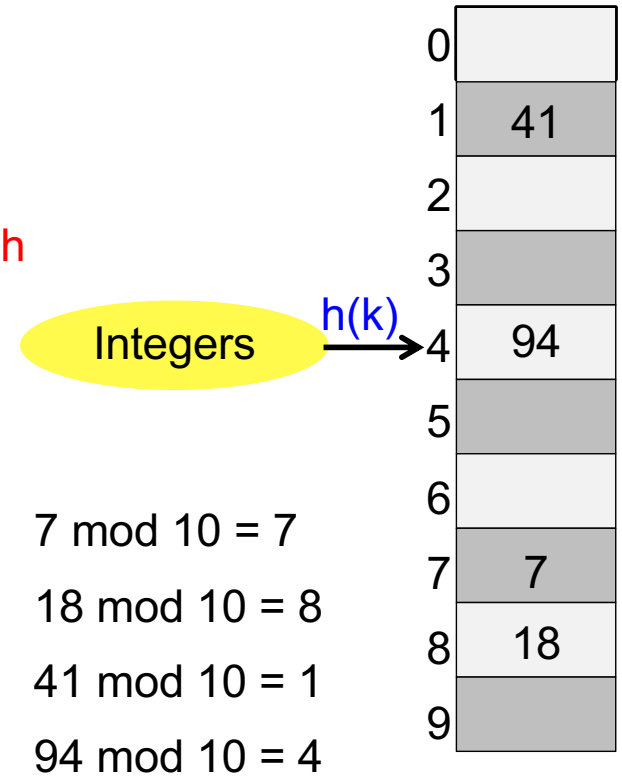
  `delete(key)`: remove the value associated with the given key
- Operations are very fast and their speed is mostly independent of the total data size
  - Unlike arrays or lists

Hash function

- The hash function takes an item (usually the key) and returns a slot index in the range 0, 1, …, TableSize-1
- We consider a simple hash function: **mod**
- Modulo operation (mod) finds the *remainder* after dividing one number by another
  - For two positive numbers *a* and *b*, *a* mod *b* is the remainder when *a* is divided by *b*
    - E.g., 5 mod 2 = 1, because 5 divided by 2 gives quotient 2 and remainder 1
    - E.g., 9 mod 3 = 0 because 9 divided by 3 gives quotient 3 and remainder 0

Hash table: example 1

- Key space = integers
- TableSize = 10
- h(k) = k mod 10
  - We consider a simple hash function: mod
  - Modulo operation (mod) finds the remainder after dividing one number by another
- Insert: 7, 18, 41, 94



7 mod 10 = 7

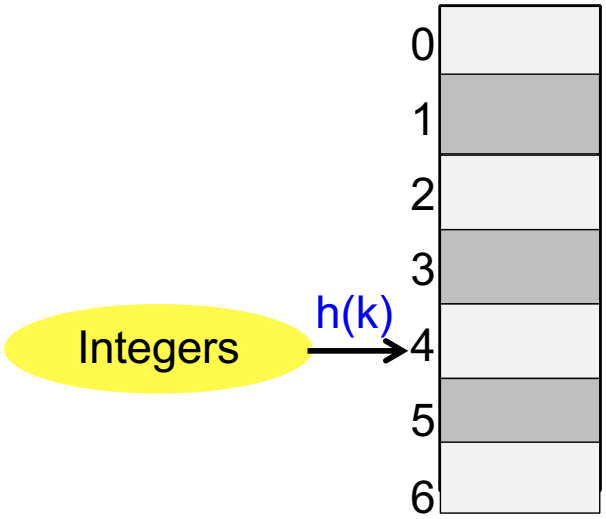
18 mod 10 = 8

41 mod 10 = 1

94 mod 10 = 4

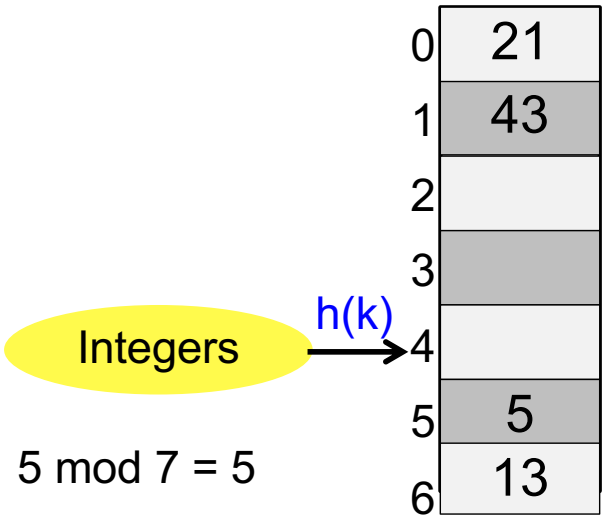
Hash table: example 2

- Key space = integers
- TableSize = 7
- h(k) = k mod 7
- Insert: 5, 13, 21, 43



Hash table: example 2

- Key space = integers
- TableSize = 7
- h(k) = k mod 7
- Insert: 5, 13, 21, 43



5 mod 7 = 5

13 mod 7 = 6

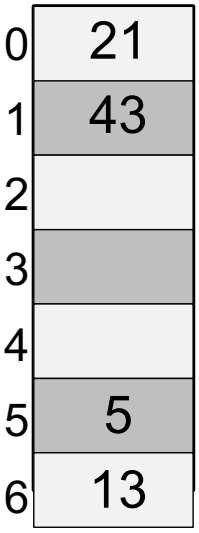
21 mod 7 = 0

43 mod 7 = 1

Hash table: example 2

- Key space = integers
- TableSize = 7
- h(k) = k mod 7
- Insert: 5, 13, 21, 43

- Insert 4231988
- What happens?



4231988 mod 7 = 5
but slot 5 is busy:
collision!

Hash function and collisions

- Desirable properties of hash functions:
  - Simple and fast to compute
  - Spread key values evenly across the hash table
  - Minimize collisions

- *Collision*: two keys map to the same slot in the hash table
  - Resolving collisions adds extra work

An example of collision in real life

- The birthday paradox
  https://en.wikipedia.org/wiki/Birthday_problem
- *How many people must be there in a room for there to be a 50% chance that at least two people share the same birthday?*
- Surprising answer: only 23!
- To reach a 99.9% probability, you need just 71 people
- Assumption: each day of the year (excluding February 29) is equally likely for a birthday

- Connection to hash tables: collisions occur sooner than expected, so table size and hash function matter

An example of collision in real life

- Let's calculate the probability that two people among *n* share a birthday

  p(same): probability that at least two people in a room of *n* have the same birthday

  p(same) = 1 – *p(different)*, where *p(different)* is the probability that all *n* people have different birthdays

  p(different) = 1 x (364/365) x (363/365) x (362/365) x …

  … x (1 - (*n*-1)/365)

  - The 1st person can have any of the 365 days
  - The 2nd person must have a birthday different from the first
  - The 3rd person must have a birthday different from the first two, and so on

- With some math (using Taylor's series) we find that

  p(same) $\approx 1 - e^{-n^2/(2\times 365)}$

  that is $n \approx \sqrt{2 \times 365 ln\left(\frac{1}{1-p(same)}\right)}$

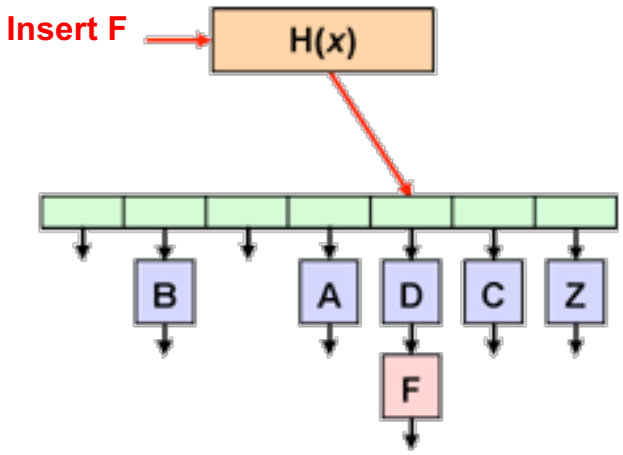
Handle collisions in hash tables

- When two keys map to the same slot, collisions occur
- Resolved using collision handling techniques

1. Separate chaining
   - Each slot contains a linked list of all keys that hash to that slot
2. Open addressing
   - All elements are stored within the table
   - If a slot is occupied, try another slot according to a probing sequence
     a) linear probing
     b) quadratic probing
     c) double hashing

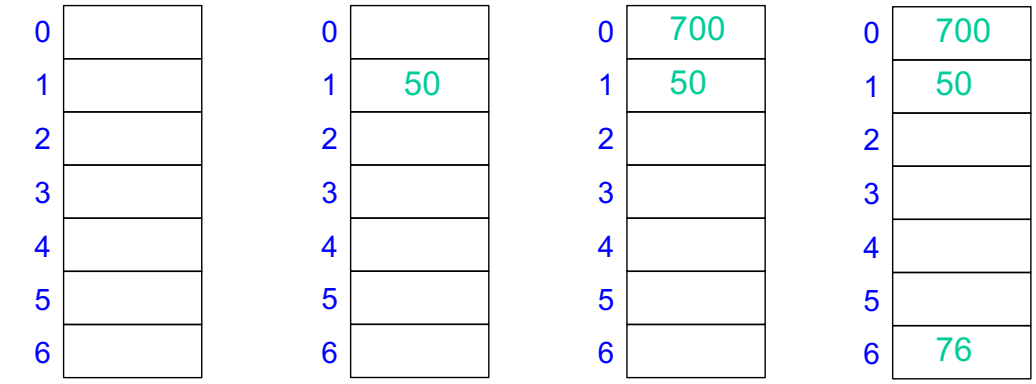
Separate chaining

- Idea: all keys that map to the same hash value (i.e., slot) are stored in a list (*linked list* to store elements that collide)
  - Insert: add the new key to the list at that slot



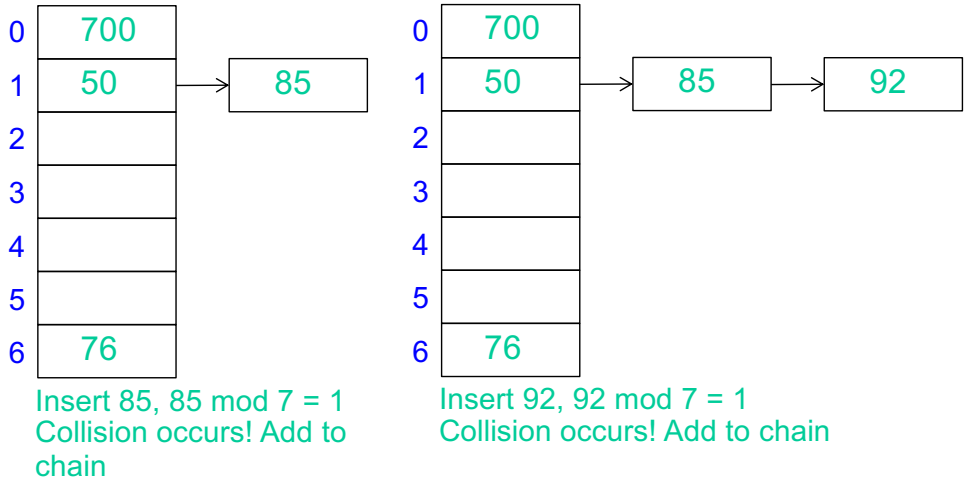
Separate chaining: example

- Key space = integers
- TableSize = 7
- h(k) = k mod 7
- Insert: 50, 700, 76, 85, 92, 73, 101



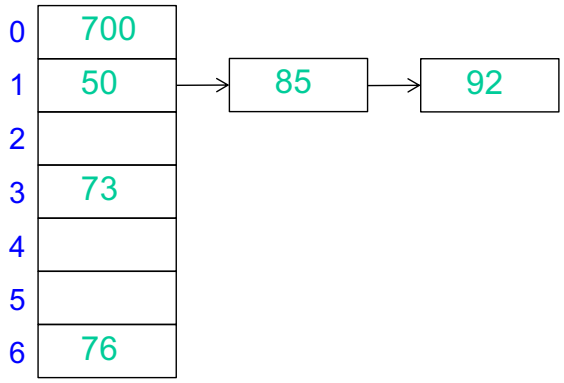
Separate chaining: example

- Key space = integers
- TableSize = 7
- h(k) = k mod 7
- Insert: 50, 700, 76, 85, 92, 73, 101



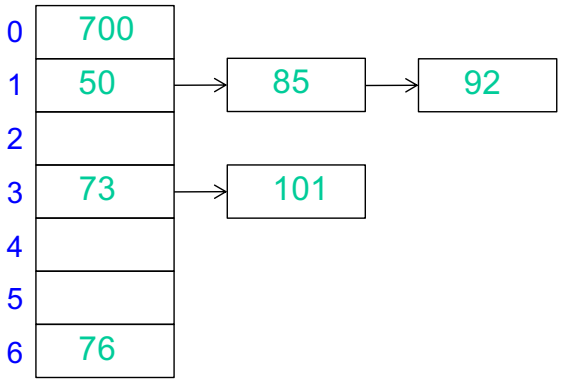
Separate chaining: example

- Key space = integers
- TableSize = 7
- h(k) = k mod 7
- Insert: 50, 700, 76, 85, 92, 73, 101



Insert 73

73 mod 7 = 3



Insert 101, 101 mod 7 = 3

Collision occurs! Add to chain

Separate chaining: operations and performance

- Let A be the hash table
- insert(key, value): add new key-value pair into A[h(key)]
  - Takes O(1) on average
- lookup(key): search for the key in A[h(key)]
  - Takes O(1 + c) on average, where c = average length of the linked list
- delete(key): remove the key-value pair from A[h(key)]
  - Takes O(1 + c) on average
- If c is bounded by a constant, then all the operations are O(1) on average

Separate chaining: pros and cons

Pros

- Simple to implement
- Hash table never fills up, we can always add new elements to the chain
- Less sensitive to the hash function's choice

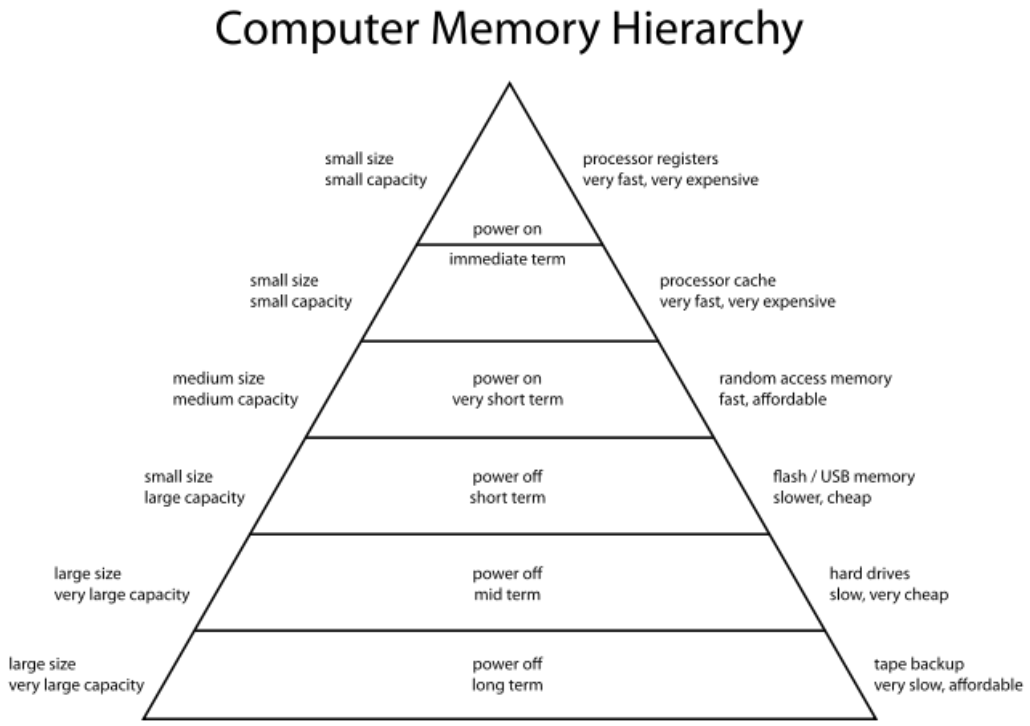
Cons

- Wastage of space: some parts of the hash table may never be used
- Long chains: if many keys collide, search time can become O(n) in the worst case
- Extra storage: chains require additional memory, including extra space to store links
- Not well performing due to cache inefficiency
  - Slots scattered in memory

Break: memory hierarchy

- Modern computers use multiple levels of memory
  - Registers inside CPU: fastest
  - Cache memory
  - Main memory (RAM)
  - Flash and USB memories
  - SSDs and hard disks
- Trade-off:
  - Higher levels → faster but smaller and more expensive
  - Lower levels → slower but larger and cheaper
- Key idea:
  - Programs run faster when data is in higher (faster) memory levels
- Memory hierarchy effect: gives the illusion of large, fast, and low-cost memory

Break: memory hierarchy



Open addressing

- Idea: if a slot is occupied, find another *open* (i.e., free) slot in the hash table
  - No linked list as in separate chaining: all elements are stored directly in the hash table
- How it works: define a *probe sequence* (order of slots to check)
- When inserting a new element into the table
  - Try the first-choice slot (from the hash function)
  - If occupied, try the next slot in the probe sequence
  - Repeat until an empty slot is found

Open addressing: probe sequence

- The probe sequence defines the order of alternative slots to resolve collisions
- How to define the probe sequence

  $h_i(k) = (h(k) + F(i)) \bmod TableSize$

  - k: key
  - h(k): hash function (first-choice slot)
  - F(i): collision resolution function (or probe function)
  - i: probe number
    - i=0: first choice
    - i=1: second choice
    - i=2: third choice, and so on
  - mod TableSize to wrap around the hash table when we reach the last slot

Open addressing: probe sequence

- When searching for key k:
  - Start at slot $h_0(k)$ (first-choice)
  - If collision occurs, check $h_1(k)$, $h_2(k)$, $h_3(k)$, …
  - Stop when
    - k is found or
    - An empty slot is found: k is not in the table

Open addressing: probe function

- $h_i(k) = (h(k) + F(i)) \bmod \text{TableSize}$
- Types of addressing use different collision resolution function F
- F can be:
  - Linear: F(i) = i
    - Step size: +1, +2, +3, +4, … : check slots sequentially
  - Quadratic: $F(i) = i^2$
    - Step size: +1, +4, +9, +16, ... : spread out probes
  - Double hashing: F(i) = i * g(k)
    - g(k): second hash function (step size)
    - Provides better distribution of probes
- The choice of F affects performance

Open addressing: linear probing

- If a slot is occupied, find the next free slot in the table
- Method:
  - Check slots one by one in sequence starting from first-choice
  - If collision occurs, check the next slot
  - Continue until an empty slot is found
- Wrap-around
  - When reaching the last slot, continue from the beginning of the table
- Linear probing resolves collisions by sequentially searching for the next slot

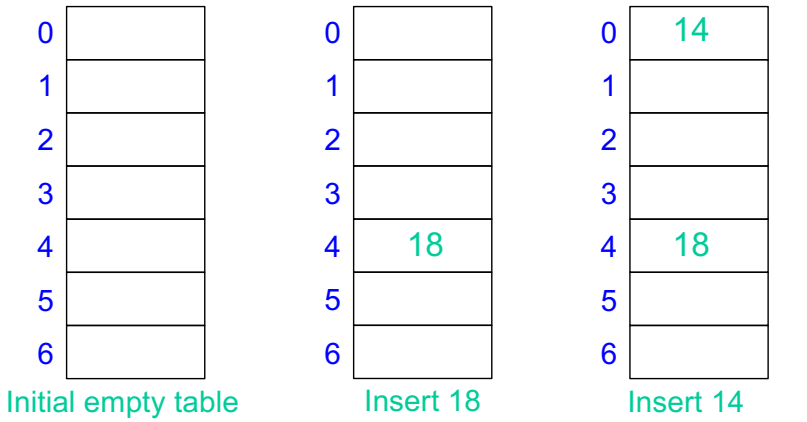
Open addressing: linear probing

- To find key k, check slots

  h(k), h(k)+1, h(k)+2, h(k)+3, …
- Stop when k is found or reach an empty slot (i.e., k is not present)
- Probe sequence
  - 0^{th} probe: $h_0(k) = h(k)$
  - 1^{st} probe: $h_1(k) = (h(k)+1) \bmod TableSize$
  - 2^{nd} probe: $h_2(k) = (h(k)+2) \bmod TableSize$
  - i^{th} probe: $h_i(k) = (h(k)+i) \bmod TableSize$
- Linear probing checks slots sequentially with wrap-around

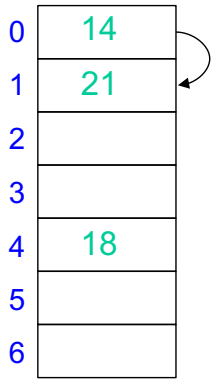
Linear probing: example

- Key space = integers
- TableSize = 7
- h(k) = k mod 7
- Insert: 18, 14, 21, 1, 35

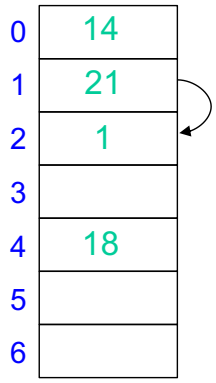


Linear probing: example

- Key space = integers
- TableSize = 7
- h(k) = k mod 7
- Insert: 18, 14, 21, 1, 35



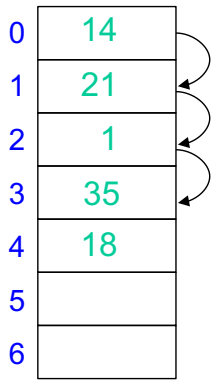
Insert 21, 21 mod 7 = 0
Collision occurs! Look
for next empty slot



Insert 1, 1 mod 7 = 1
Collision occurs! Look
for next empty slot

Linear probing: example

- Key space = integers
- TableSize = 7
- h(k) = k mod 7
- Insert: 18, 14, 21, 1, 35

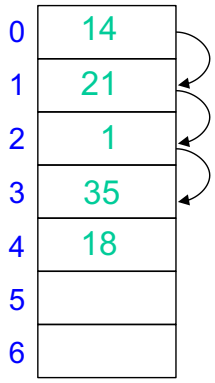


Insert 35, 35 mod 7 = 0
Collision occurs! Look
for next empty slot

What happens when we look for 35?

Linear probing: example

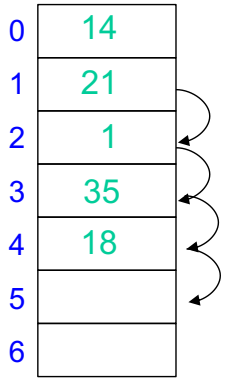
- Probe sequence when we look for 35
  - 0^{th} probe: $h_0(35) = h(35) = 0$
  - 1^{st} probe: $h_1(35) = (h(35)+1) \bmod 7 = (0+1) \bmod 7 = 1$
  - 2^{nd} probe: $h_2(35) = (h(35)+2) \bmod 7 = (0+2) \bmod 7 = 2$
  - 3^{rd} probe: $h_3(35) = (h(35)+3) \bmod 7 = (0+3) \bmod 7 = 3$ found!



Look for 35, 35 mod 7 = 0 It is occupied: look for next slot.
35 found after 4 probes

Linear probing: example

- Key space = integers
- TableSize = 7
- h(k) = k mod 7
- Find: 35, 8

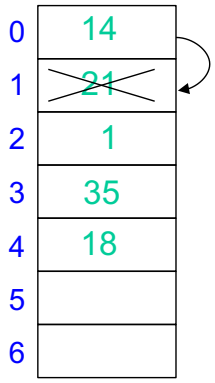


Look for 8, 8 mod 7 = 1.
Collision occurs! After 5 probes empty slot: not found

What happens when we look for 8?

Linear probing: example

- Key space = integers
- TableSize = 7
- h(k) = k mod 7
- Delete: 21

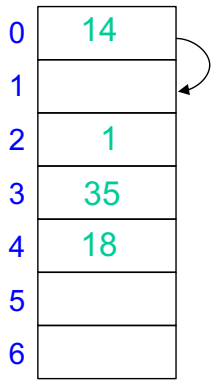


Delete 21, 21 mod 7 = 0.
Collision occurs! After 2 probes 21 found and deleted

Be careful: delete is tricky

Linear probing: example

- Key space = integers
- TableSize = 7
- h(k) = k mod 7
- Find: 35



Find 35, 35 mod 7 = 0

What happens when we look for 35?

Not found → wrong!

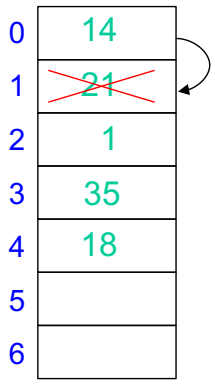
We cannot simply delete a value, because it can affect find!

Linear probing: deletion

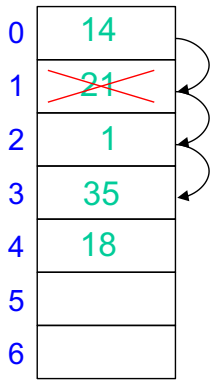
- For each slot, add a slot state which can be:
  - Occupied
  - Deleted
  - Empty
- How to delete
  - Mark the slot as deleted instead of clearing it
  - Ensures that searches following the probe sequence still work correctly
  - Implementation: use an additional array of the same size as the table to track slot

Linear probing: example

- Key space = integers
- TableSize = 7
- h(k) = k mod 7
- Delete 21, find 35, insert 15



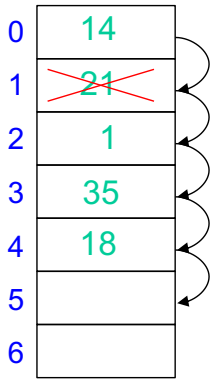
Delete 21, 21 mod 7 = 0. Collision occurs! After 2 probes 21 found and marked as deleted



Find 35, 35 mod 7 = 0. Collision occurs! After 4 probes 35 found

Linear probing: example

- Key space = integers
- TableSize = 7
- h(k) = k mod 7
- Delete 21, find 35, insert 15

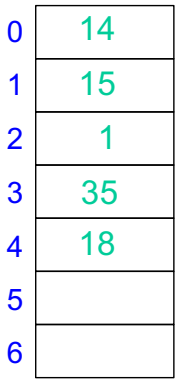


Insert 15, 15 mod 7 = 1

Slot 1 is marked as deleted

Search for 15, and found that 15 is not in the hash table

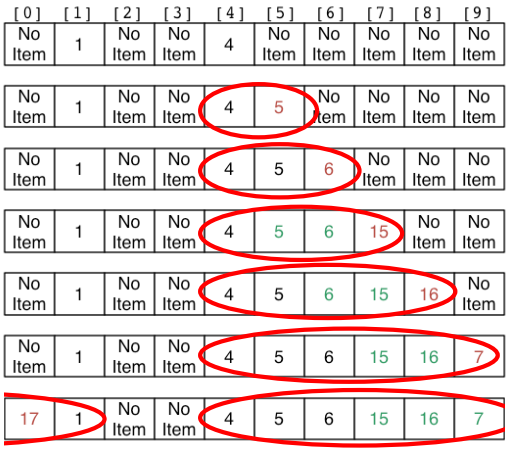
Insert 15 into the slot that has been marked as deleted



Insert 15

Linear probing: clustering

- Problem with linear probing: clustering
  - Table items tend to cluster together in the hash table, i.e., table contains groups of consecutively occupied slots
  - Clustering causes long probe sequence and decreases efficiency
- E.g., insert 5, 6, 15, 16, 7, 17 with h(k) = k mod 10



Open addressing: quadratic probing

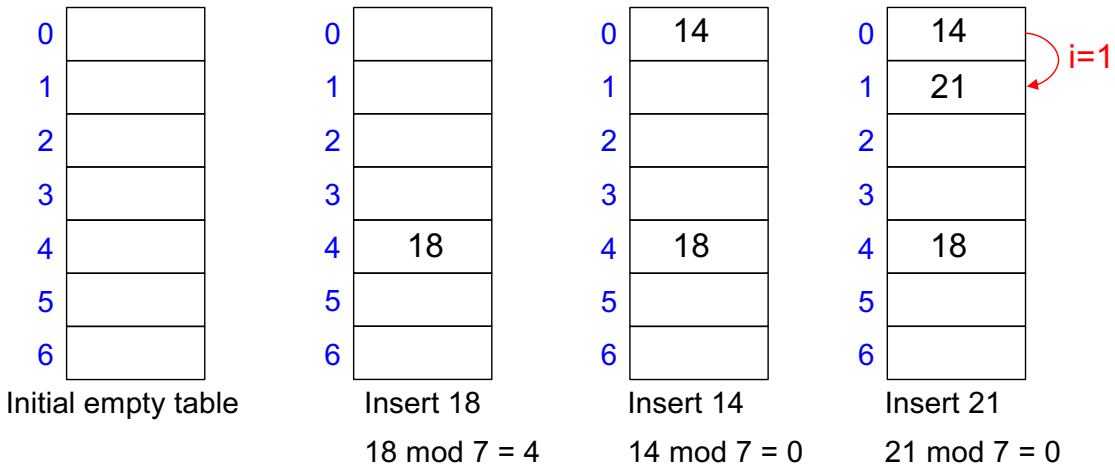
- $h_i(k) = (h(k) + F(i)) \bmod \text{TableSize}$
- Quadratic probing: $F(i) = i^2$
- Probe sequence
  - 0^{th} probe: $h_0(k) = h(k)$
  - 1^{st} probe: $h_1(k) = (h(k)+1) \bmod \text{TableSize}$
  - 2^{nd} probe: $h_2(k) = (h(k)+4) \bmod \text{TableSize}$
  - i^{th} probe: $h_i(k) = (h(k)+i^2) \bmod \text{TableSize}$

1^2 2^2

- Less clustering compared to linear probing
- Probes are spread out, reducing long sequences of occupied slots

Quadratic probing: example

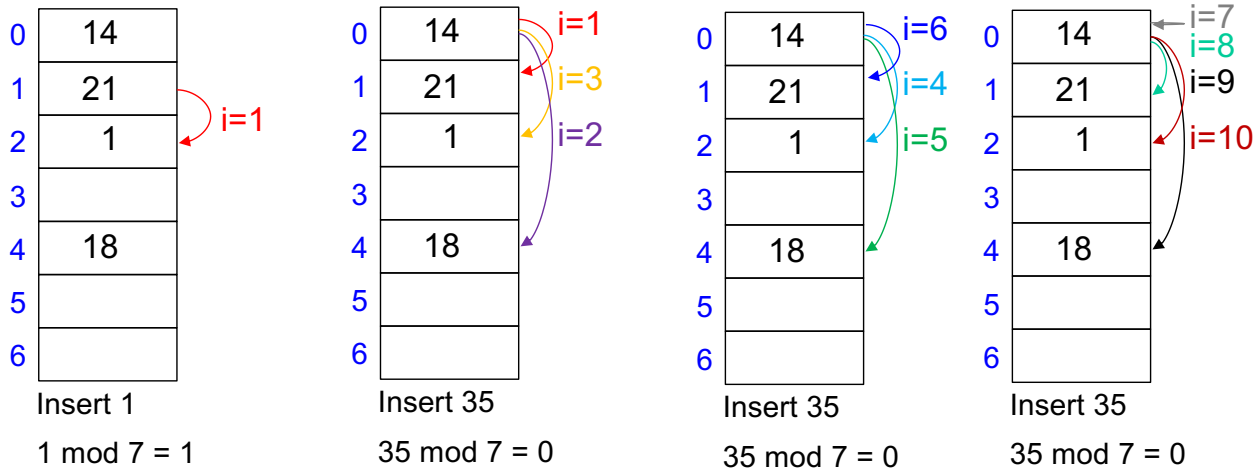
- Key space = integers
- TableSize = 7
- h(k) = k mod 7
- Insert: 18, 14, 21, 1, 35



Quadratic probing: example

- Key space = integers
- TableSize = 7
- h(k) = k mod 7
- Insert: 18, 14, 21, 1, 35

- Issue: the probe sequence may not cover all slots
- For 35, no free slot found, insertion fails
- We get the slot sequence 0, 1, 4, 2, 2, 4, 1 which repeats endlessly



Open addressing: double hashing

- $h_i(k) = (h(k) + F(i)) \bmod \text{TableSize}$
- Double hashing: $F(i) = i * g(k)$
  - g(k) is a second hash function, independent of h(k)
- Probe sequence
  - 0^{th} probe: $h_0(k) = h(k)$
  - 1^{st} probe: $h_1(k) = (h(k)+g(k)) \bmod \text{TableSize}$
  - 2^{nd} probe: $h_2(k) = (h(k)+2*g(k)) \bmod \text{TableSize}$
  - i^{th} probe: $h_i(k) = (h(k)+i*g(k)) \bmod \text{TableSize}$
- Pros: no clustering
- Cons: more computation, as two hash functions need to be computed

Open addressing: pros and cons

Pros

- Better cache performance with respect to separate chaining
  - Slots contiguous in memory
- Better space usage
  - A slot can be used even if no element maps to it
- No need of linked lists (and extra memory to store them)

Cons

- More computation than separate chaining
- Hash table can become full
- Clustering risk: extra care needed in probe sequence design
- May fail to find an empty slot, depending on probe function and table size

Exercise

- Insert keys 12, 18, 13, 2, 3, 23, 5 and 15 into an initially empty hash table of length 10 using separate chaining and hash function h(k) = k mod 10
  1. Which is the resulting hash table?
  2. Which are the steps to find 23 in the resulting hash table?
  3. Now consider again an empty table and use open addressing with linear probing: which is the resulting hash table after the insertions?
  4. How do you find 23 in that resulting hash table?

Exercise

5. If you rather use open addressing with quadratic probing, which is the resulting hash table after the insertions?
6. How do you find 23 in that resulting hash table?
7. If you rather use open addressing with double hashing probing, which is the resulting hash table after the insertions? Use g(k) = 1 + k mod 7

References

- T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, “Introduction to Algorithms”, MIT Press, Chapter 11, 2022.
- C. Demetrescu, I. Finocchi, G. F. Italiano, “Algoritmi e Strutture Dati 3ª edizione”, Mc-Graw Hill, 2025 (in Italian)
- Wikipedia, “Hash table” https://en.wikipedia.org/wiki/Hash_table