TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Hashing

Algorithms, Data and Security
A.Y. 2024/25

Valeria Cardellini

Global Governance, 3rd year
Science and Technology Major

What is hashing?

- **Hashing** is a powerful technique (algorithm and data structure) primarily used for efficiently storing and retrieving data
- It allows us to efficiently map input data of variable length to smaller data of fixed length
- Widely used in many kinds of computer software: databases, caches, …

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

Examples of how hashing is used

- In universities, each student is assigned a unique roll number that can be used to retrieve information about them
- A phone book has name, address and phone number as fields. To find somebody's phone number, you search the phone book based on name
- An account on Instagram has username and password. You log in using your username and password and it takes you to your personal profile with 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

?

	Name	Surname	Tel.
6	Andrea	Smith	34523785
8	Clare	Hubers	34234673
10	Adam	Johin	12356245
11	Zoe	Klark	56292345

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**

	Name	Surname	Tel.
6	Andrea	Smith	34523785
8	Clare	Hubers	34234673
10	Adam	Johin	12356245
11	Zoe	Klark	56292345

Why do we need hashing?

- Many apps deal with lots of data
- There are myriad of data accesses which require data lookups
- But lookups are time critical
- Data structures like arrays may not be sufficient to handle efficient lookups
  - We have to search through all the elements of the array: O(n)
- In general: we use hashing when lookups need to occur in near constant time: O(1)

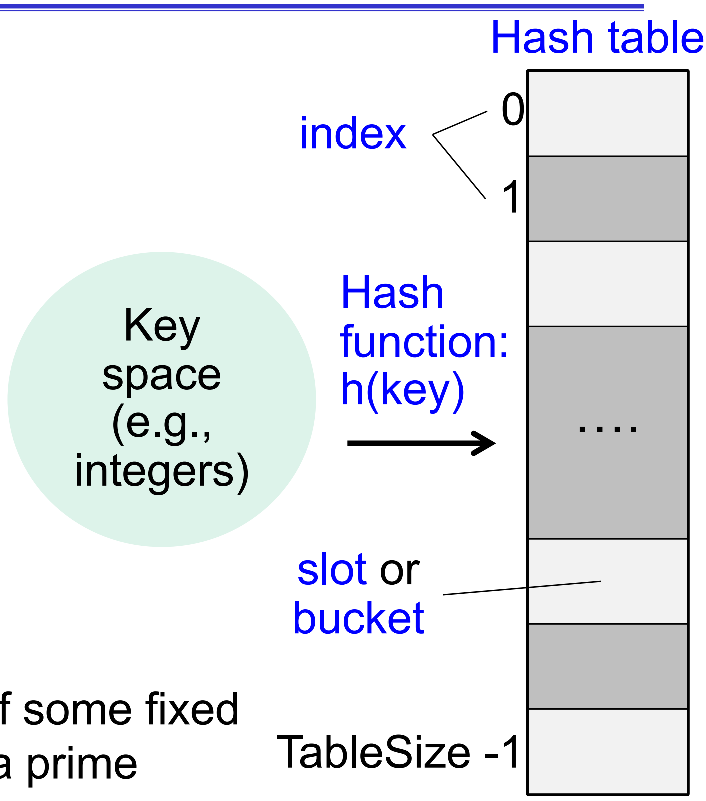
Why do we need hashing?

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

- Unsorted array of size n
  - Lookup: sequential search, so O(n)
  - Insert: insert at the end, so O(1)
  - Delete: search element and then delete it, so O(n)
- Sorted array of size n
  - Lookup: binary search, so O(log n)
  - Insert: shift elements following element to be inserted, so O(n)
  - Delete: search element and then shift all elements following element to be removed, so O(n)
- Ideal implementation: hash table

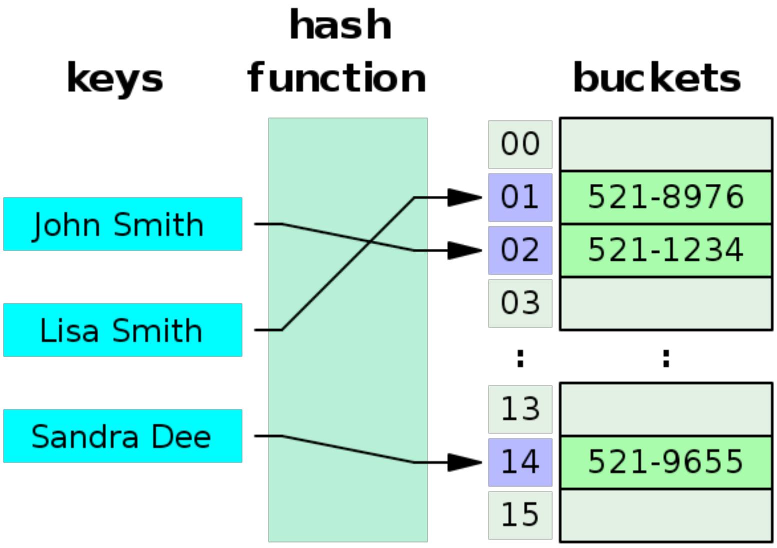
Hash table

- A **hash table** (or hash map) is a data structure to efficiently map keys to values, for efficient search and retrieval
- It uses a hash function to compute an *index* into an array of *buckets* or *slots*, from which the desired value can be found
- Constant time access!
- A hash table is an array of some fixed size (TableSize), usually a prime number



Hash table: example

- A phone book as a hash table



Hash table: operations

- Search (or lookup)
  - lookup(item): find the slot which contains "item"
- Insertion
  - insert(item): add the new value "item"
- Deletion
  - delete(item): remove the value "item"

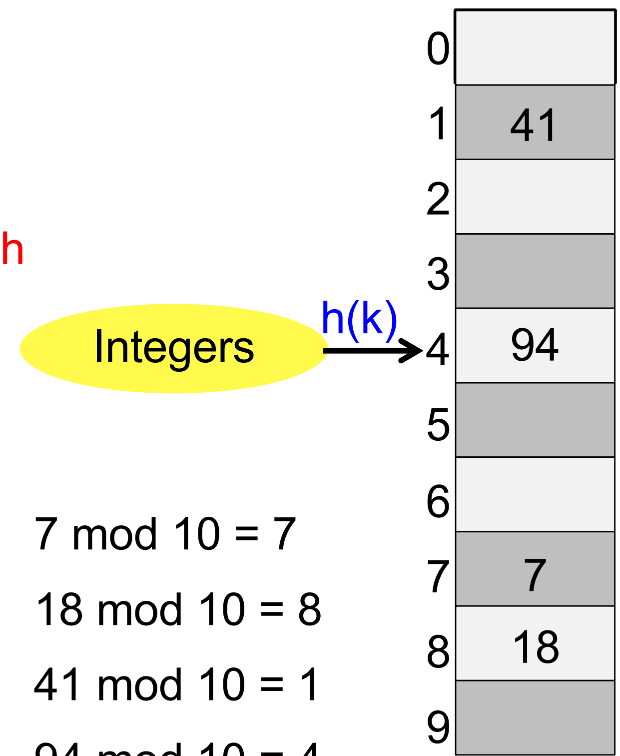
- Operations are very fast irrespective of data size

Hash function

- The hash function takes any item in the dataset and returns a slot index in the range 0, …, TableSize-1
- We consider a simple hash function: **mod**
- Modulo operation (mod) finds the *remainder* after division of one number by another
  - Given two positive numbers *a* and *b*, *a* mod *b* is the remainder of division of *a* by *b*
    - E.g., 5 mod 2 = 1, because 5 divided by 2 has a quotient of 2 and a remainder of 1
    - E.g., 9 mod 3 = 0 because 9 divided by 3 has a quotient of 3 and a remainder of 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 division of one number by another
- Insert: 7, 18, 41, 94



7 mod 10 = 7

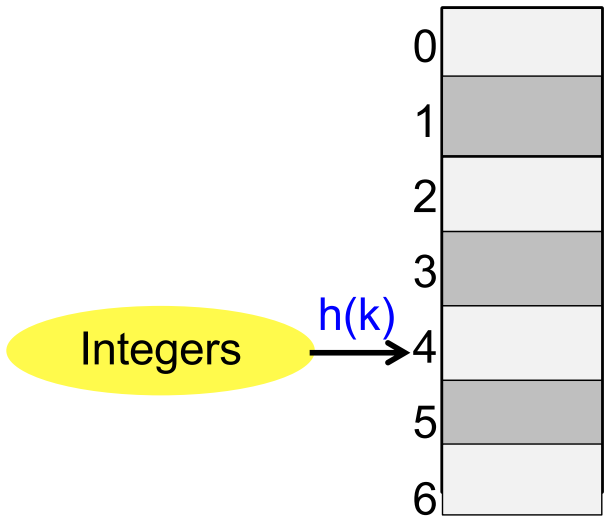
18 mod 10 = 8

41 mod 10 = 1

94 mod 10 = 4

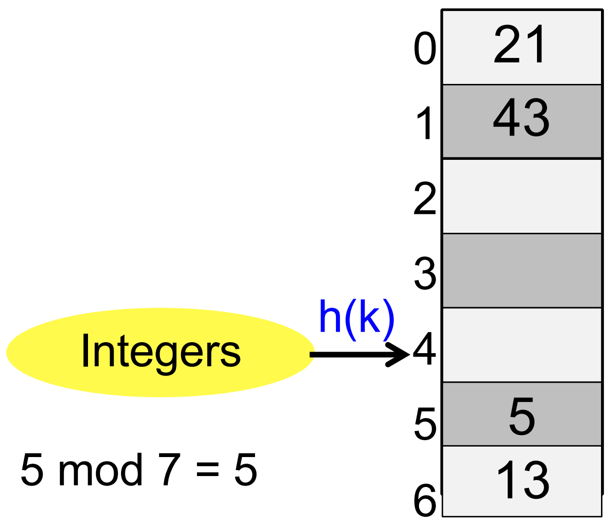
Hash table: example 2

- 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?

0	21
1	43
2	
3	
4	
5	5
6	13

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

Hash function and collisions

- Desirable properties of hash functions:
  - Simple/fast to compute
  - Spread key values evenly over the hash table
  - Avoid collisions

- *Collision*: when two keys map to the same slot in the hash table

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 to make the probability 50% that at-least two people in the room have same birthday?*
  - Answer is 23, surprisingly very low!
- We need only 71 people to make the probability 99.9%
- We assume each day of the year (excluding February 29) is equally probable for a birthday

An example of collision in real life

- How do we calculate the probability that two persons among *n* have same birthday?

  p(same): probability that two persons in a room with *n* have same birthday

  p(same) = 1 – *p(different)*, where *p(different)* is the probability that all of them have different birthday

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

  … x (1 - (*n*-1)/365)

  - Because the 1st person can have any birthday among 365, the 2nd person should have a birthday which is not same as 1st person, the 3rd person should have a birthday which is not same as first two persons, 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 $n \approx \sqrt{2 \times 365 ln\left(\frac{1}{1-p(same)}\right)}$

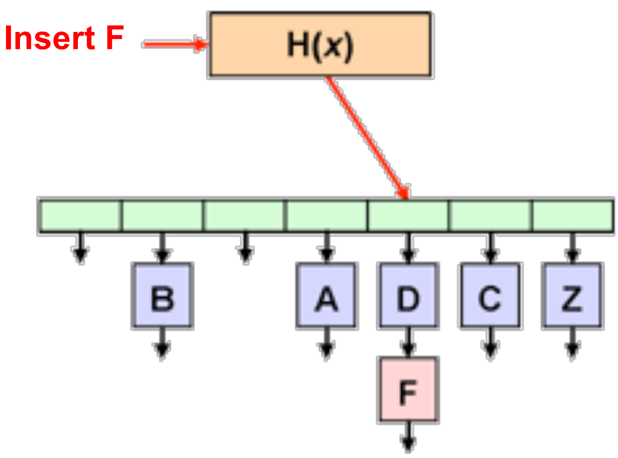
How to handle collisions in hash table

- Collisions must be handled using some collision handling technique
- Two ways to resolve collisions:

1. Separate chaining
2. Open addressing
   a) linear probing
   b) quadratic probing
   c) double hashing

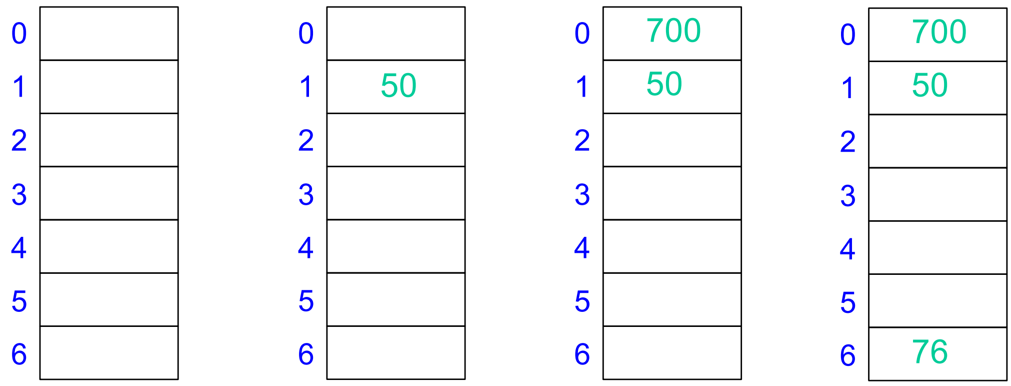
Separate chaining

- **Separate chaining**: all keys that map to the same hash value (i.e., slot) are kept in a list (*linked list* to store elements with collided key)



Separate chaining: example

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



Initial empty table

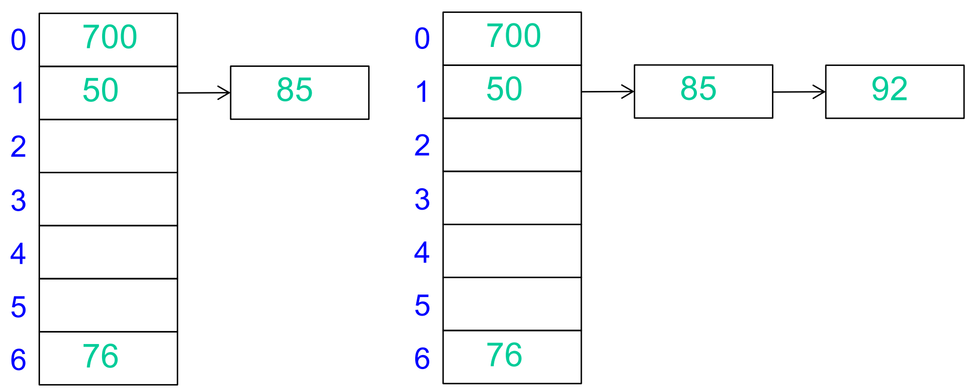
Insert 50
50 mod 7 = 1

Insert 700
700 mod 10 = 0

Insert 76
76 mod 7 = 6

Separate chaining: example

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

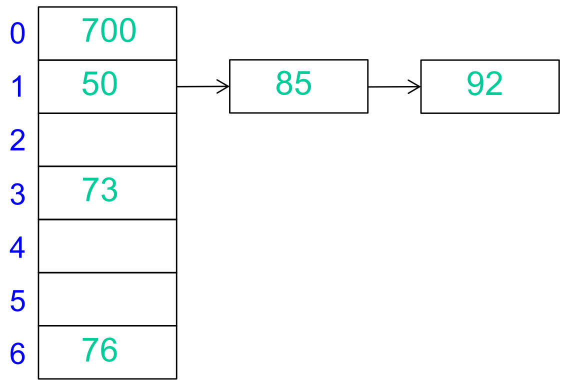


Insert 85, 85 mod 7 = 1
Collision occurs! Add to chain

Insert 92, 92 mod 7 = 1
Collision occurs! Add to chain

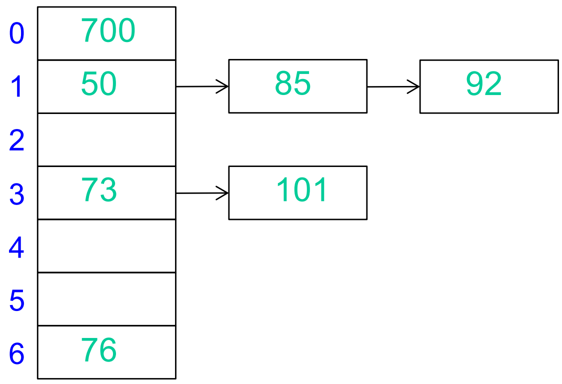
Separate chaining: example

- 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: performance

- Insertion insert(number): add new entry "number" into hash table A
  - Insert data into A[h(number)]: takes O(1) time
- Retrieval find(key): find entry "key"
  - Find key from A[h(key)]: takes O(1+c) time on average, where c is the average length of the linked list
- Deletion: delete(number): remove entry "number"
  - Delete A[h(number)]: takes O(1+c) time on average
- If c is bounded by some constant, then all three operations are O(1)

Separate chaining: pros and cons

Pros

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

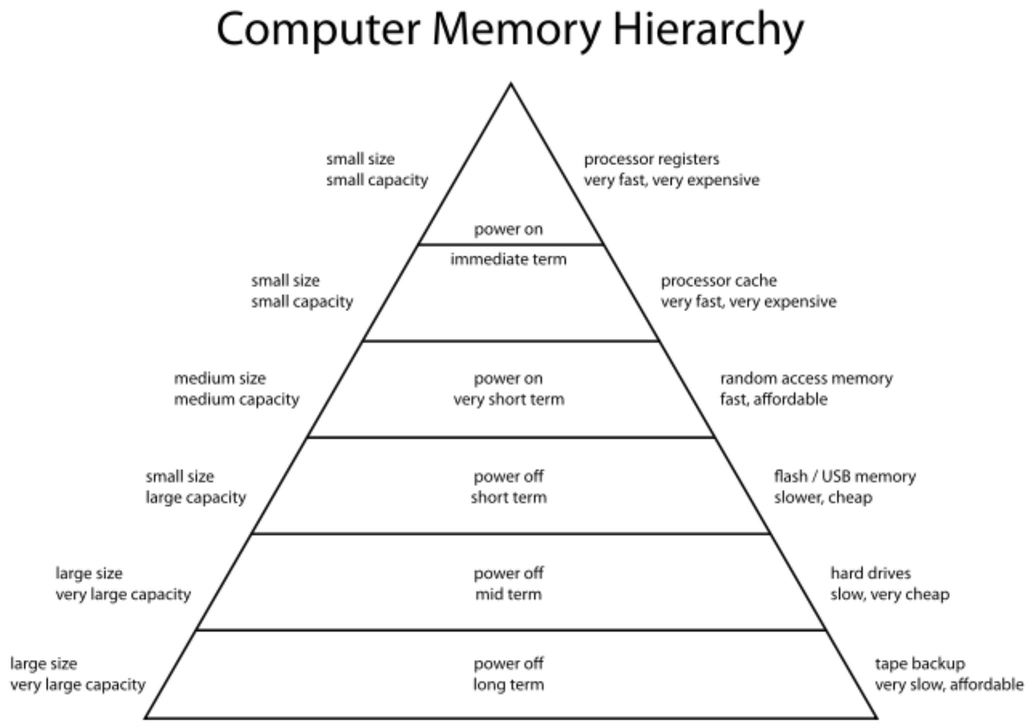
Cons

- Wastage of space of hash table (some parts are never used)
- If chain becomes long, then search time can become O(n) in worst case
- Make use of storage outside of the hash table itself, including extra space to store links
- Not well performing (because of poor cache performance)

Break: memory hierarchy

- The memory of modern computer architectures has a number of levels
  - From fast registers inside CPU
  - Through one or more levels of cache memory
  - To main memory (RAM)
  - To flash and USB memories
  - To SSDs and hard disks
- Each successive level stores more data than the previous level and costs less, but access is slower
- Computation that works entirely using higher memory levels takes less time
- But higher memory levels are expensive: the memory hierarchy gives us the *illusion* of a fast, large and cheap memory

Break: memory hierarchy



Open addressing

- **Open addressing**: try to find the next *open* (i.e., free) slot in the hash table
  – No linked list as in separate chaining, now all elements are stored in the hash table itself
- Idea: let’s define a *probe sequence*
  – When a new element is to be inserted into the table, it is placed in its “first-choice” slot if possible
  – If that slot is already occupied, it is placed in its “second-choice” slot
  – The process continues until an empty slot is found in which to place the new element

Open addressing

- How do we define the probe sequence?

  $h_i(k) = (h(k) + F(i)) \bmod TableSize$
  - i is the probe number
    - i=0: first choice
    - i=1: second choice
    - i=2: third choice, and so on
  - mod TableSize because we wrap around when we reach the last slot of the hash table
- When searching for key k, if collision occurs on slot $h_0(k)$, then check the probe sequence of slots $h_1(k)$, $h_2(k)$, $h_3(k)$, … until either k is found or we find an empty slot, which indicates that k is not in the table

Open addressing

- $h_i(k) = (h(k) + F(i)) \bmod TableSize$
- Various types of addressing differ in which probe sequence they use
- F is the collision resolution function, it can be:
  - Linear: $F(i) = i$
  - Quadratic: $F(i) = i^2$
  - Double hashing: $F(i) = i * g(k)$
    - where g(k) is a second hash function that we use to compute the step size for the probe sequence

Open addressing: linear probing

- Open addressing: try to find the next open (i.e., free) slot in the hash table
- By systematically visiting each slot one at a time, we perform an open addressing technique called linear probing
- In linear probing, when there is a collision we scan forward for the next slot
  - Wrapping around when we reach the last slot

Open addressing: linear probing

- When searching for key k, check slots h(k), h(k)+1, h(k)+2, h(k)+3, … until either k is found or we find an empty slot (i.e., k is not present)
- Probe sequence
  - 0^{th} probe: $h_0(k) = h(k)$
  - 1^{st} probe: $h_1(k) = (h(k)+1) \bmod \text{TableSize}$
  - 2^{nd} probe: $h_2(k) = (h(k)+2) \bmod \text{TableSize}$
  - i^{th} probe: $h_i(k) = (h(k)+i) \bmod \text{TableSize}$

Linear probing: example

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

0	
1	
2	
3	
4	
5	
6	

Initial empty table

0	
1	
2	
3	
4	18
5	
6	

Insert 18

18 mod 7 = 4

0	14
1	
2	
3	
4	18
5	
6	

Insert 14

14 mod 7 = 0

Linear probing: example

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

0	14
1	21
2	
3	
4	18
5	
6	

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

0	14
1	21
2	1
3	
4	18
5	
6	

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

0	14
1	21
2	1
3	35
4	18
5	
6	

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

What happens when we look for 35?

Linear probing: example

- Let's consider the 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!

0	14
1	21
2	1
3	35
4	18
5	
6	

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

0	14
1	21
2	1
3	35
4	18
5	
6	

What happens when we look for 8?

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

Linear probing: example

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

0	14
1	~~21~~
2	1
3	35
4	18
5	
6	

Be careful: delete is tricky

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

Linear probing: example

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

0	14
1	
2	1
3	35
4	18
5	
6	

Find 35, 35 mod 7 = 0

What happens when we look for 35?

Not found! Incorrect!

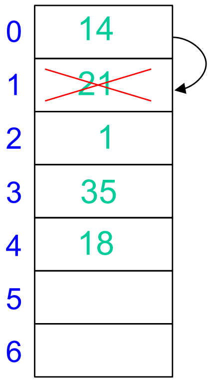
We cannot simply delete a value, because it can affect find!

Linear probing: deletion

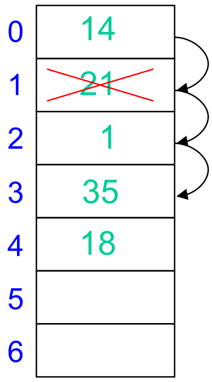
- For each slot, let's add a state slot, which can be:
  - Occupied
  - Deleted
  - Empty
- When an element is removed from hash table, we mark the slot state as "deleted", instead of emptying the slot
  - Implementation detail: need to use an additional array having the same size as the hash table, where we keep track of the slot state

Linear probing: example

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



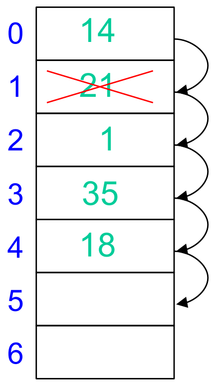
Delete 21, 21 mod 7 = 0. Collision occurs! After 2 probes 21 found and marked as deleted



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

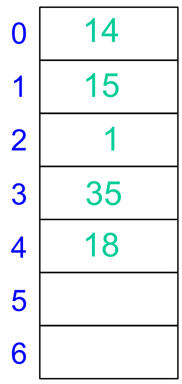


Insert 15, 15 mod 7 = 1

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

- A problem with linear probing: clustering
  - Table items tend to cluster together in the hast table, i.e., table contains groups of consecutively occupied locations
  - Clustering causes long probe searches and therefore decreases the efficiency

- E.g., insert 5, 6, 15, 16, 7, 17 with h(k) = k mod 10

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	1	No Item	No Item	4	No Item	No Item	No Item	No Item	No Item
No Item	1	No Item	No Item	4	5	No Item	No Item	No Item	No Item
No Item	1	No Item	No Item	4	5	6	No Item	No Item	No Item
No Item	1	No Item	No Item	4	5	6	15	No Item	No Item
No Item	1	No Item	No Item	4	5	6	15	16	No Item
No Item	1	No Item	No Item	4	5	6	15	16	7
17	1	No Item	No Item	4	5	6	15	16	7

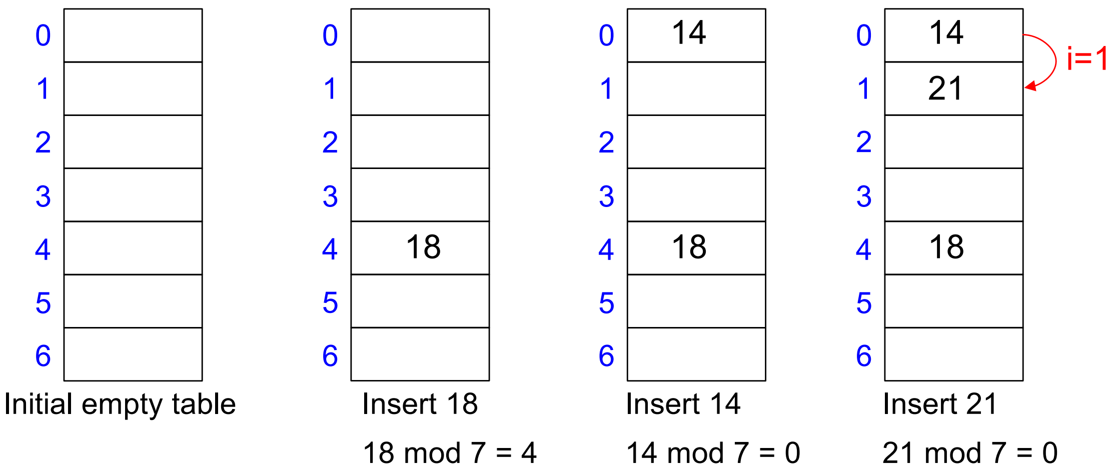
Open addressing: quadratic probing

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

- Less likely to encounter clustering

Quadratic probing: example

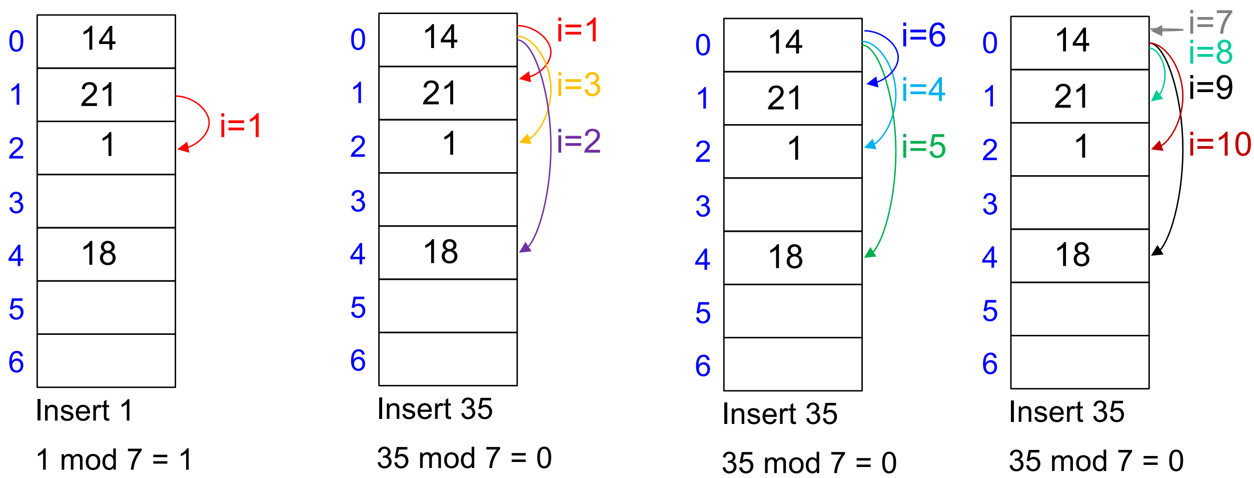
- Key space = integers
- TableSize = 7
- h(k) = k mod 7
- Insert: 18, 14, 21, 1, 35



Quadratic probing: example

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

Bad news: we are not able to find a free slot for 35, because the probing sequence does not cover all slots. We get the slot sequence 0, 1, 4, 2, 2, 4, 1 which repeats itself endlessly



Open addressing: double hashing

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

Open addressing: pros and cons

Pros

- Better performance with respect to separate chaining
  - In terms of cache (at the top of memory hierarchy in your computing device)
- Better space usage
  - A slot can be used even if no element maps to it
- No need of linked lists (and space to store them)

Cons

- Requires more computation than separate chaining
- Hash table may become full
- Requires extra care to avoid clustering

Exercise

- Insert the 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 and 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 and 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 and 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”, Mc-Graw Hill, 2008 (in Italian)
- Wikipedia, “Hash table” https://en.wikipedia.org/wiki/Hash_table