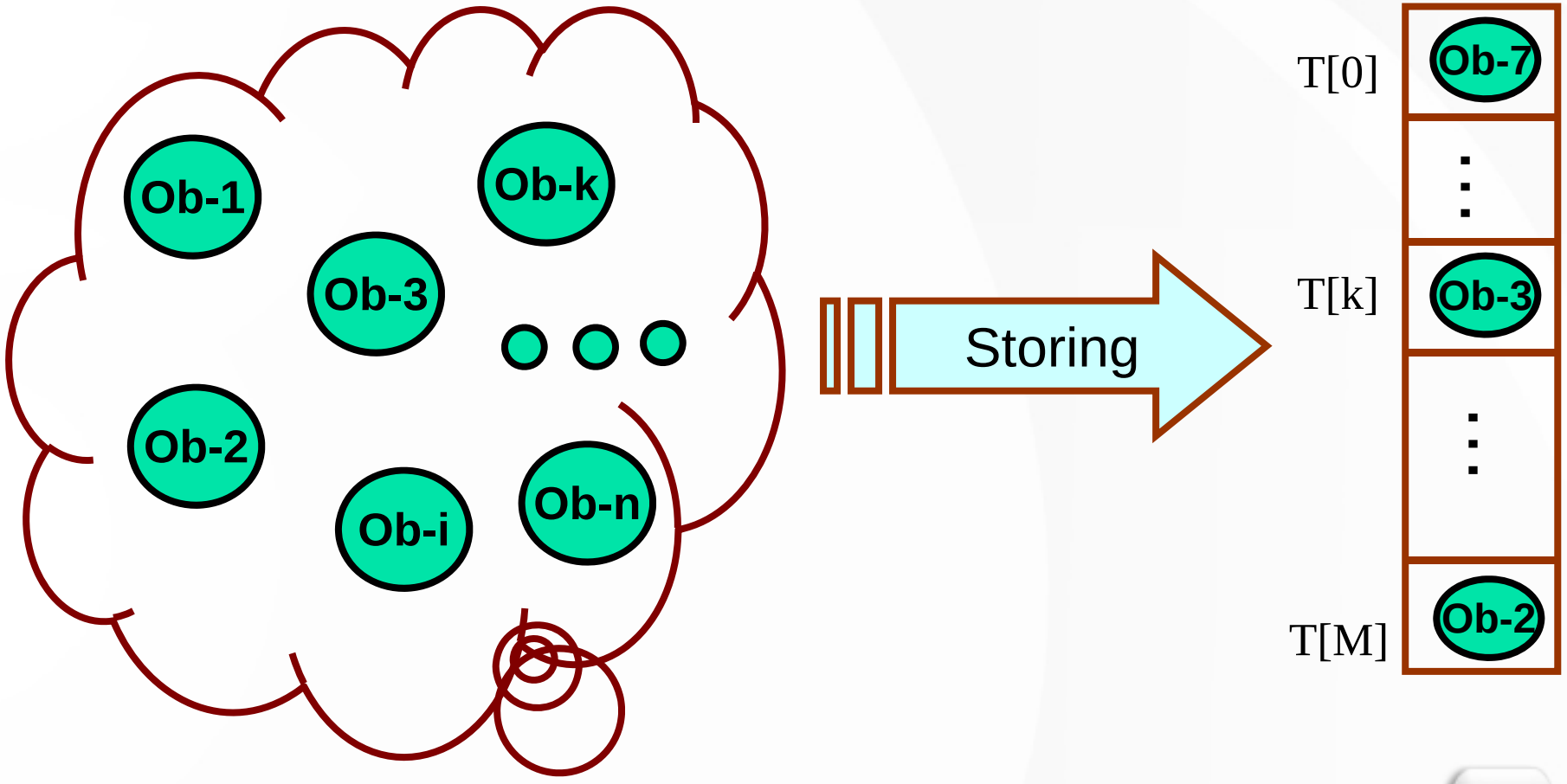
Hashing

Search v.s. Lookup

- Search
  - Start from a location
  - Go to the next
  - Repeat until the target is found (not found)
- Lookup
  - Know where the location is
- O(n) v.s. O(C)

Simple Idea



Simple Idea

- To store
  - Know where the location is
- To retrieve
  - Know where the location is
- Problem
  - Unique location for every item?

Possible Scenario

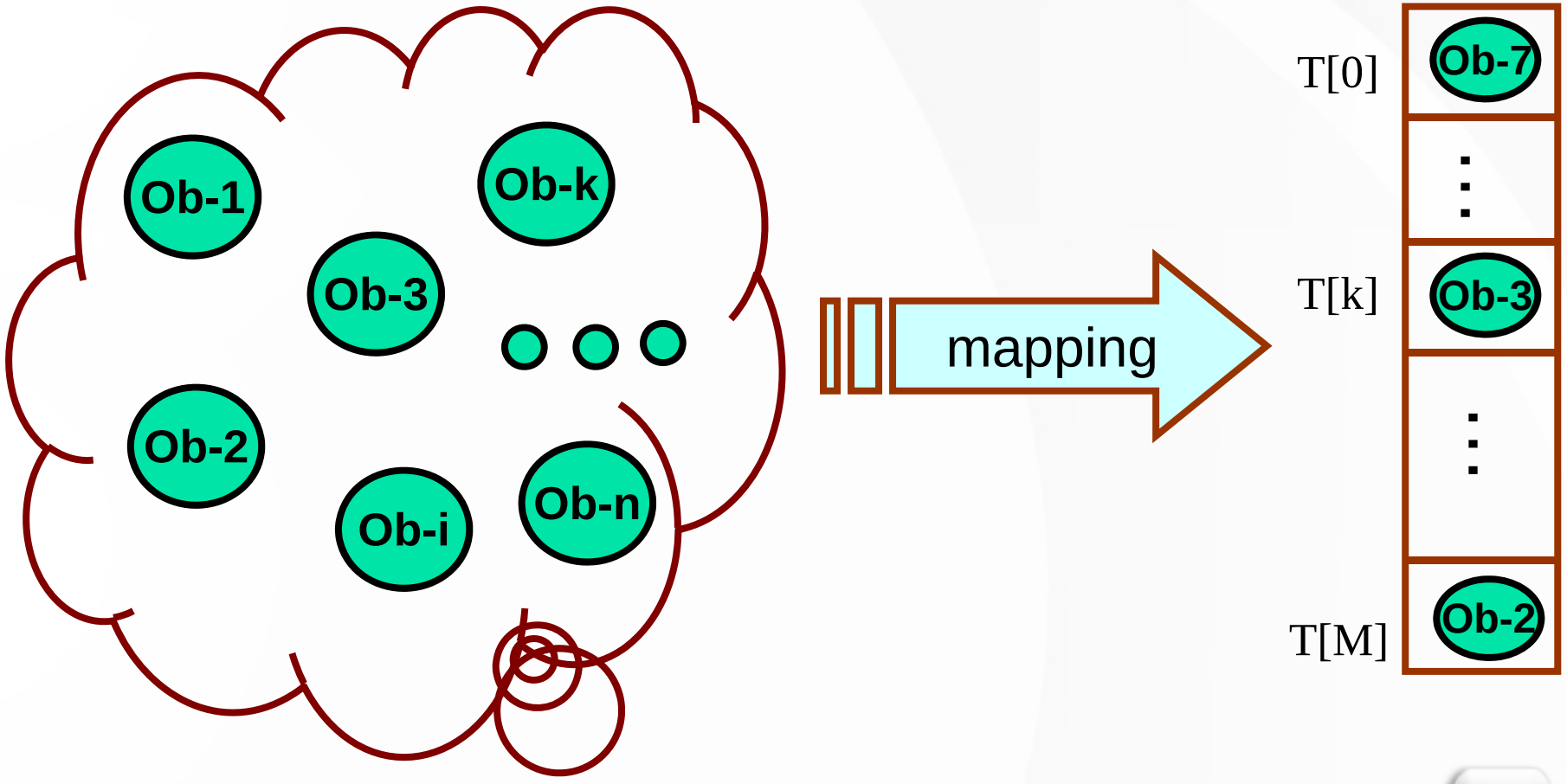
- Good case
  - Unique key for each item
  - Unique locations found according to keys
- Unfortunate case
  - Unique key for each item
  - “not-so-unique” locations
  - Collision

Collision

- Problem
  - Same location for two or more items
- Solution
  - Find a new location
- Issue
  - “storage”
    - New location
  - “retrieval”
    - Easy to find

Hash Functions

Simple Idea

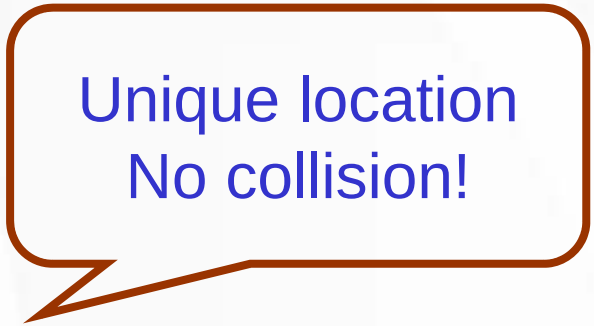


Motivation

- How to place an item in the table (hash table)?
- How to avoid collision?
  - Unique location (for each item)
  - As distributed as possible
- Easy to locate

Simple Mapping

- ID → location
- Case 1
  - 100 students
  - ID: 00 – 99
  - Location: T[0] – T[99]
- Case 2
  - 100 students
  - ID: u4510001 – u4510100
  - Location: T[0] – T[99]



Simple Mapping Variation

- ID → location
- Case 1
  - 50 students
  - ID: 00 – 99
  - Location: T[0] – T[99]
- Case 2
  - 50 students
  - ID: u4510001 – u4510100
  - Location: T[0] – T[49]

Wasted location
No collision!

Unique location
No collision!
How to map?

What do you see?

- ID → location
- Simple mapping (Direct Mapping)
  - Location = ID number
  - Location = calculation on ID number
- More challenged mapping
  - 50 students
  - ID: u4510001 – u4510100
  - 100 numbers → 50 locations

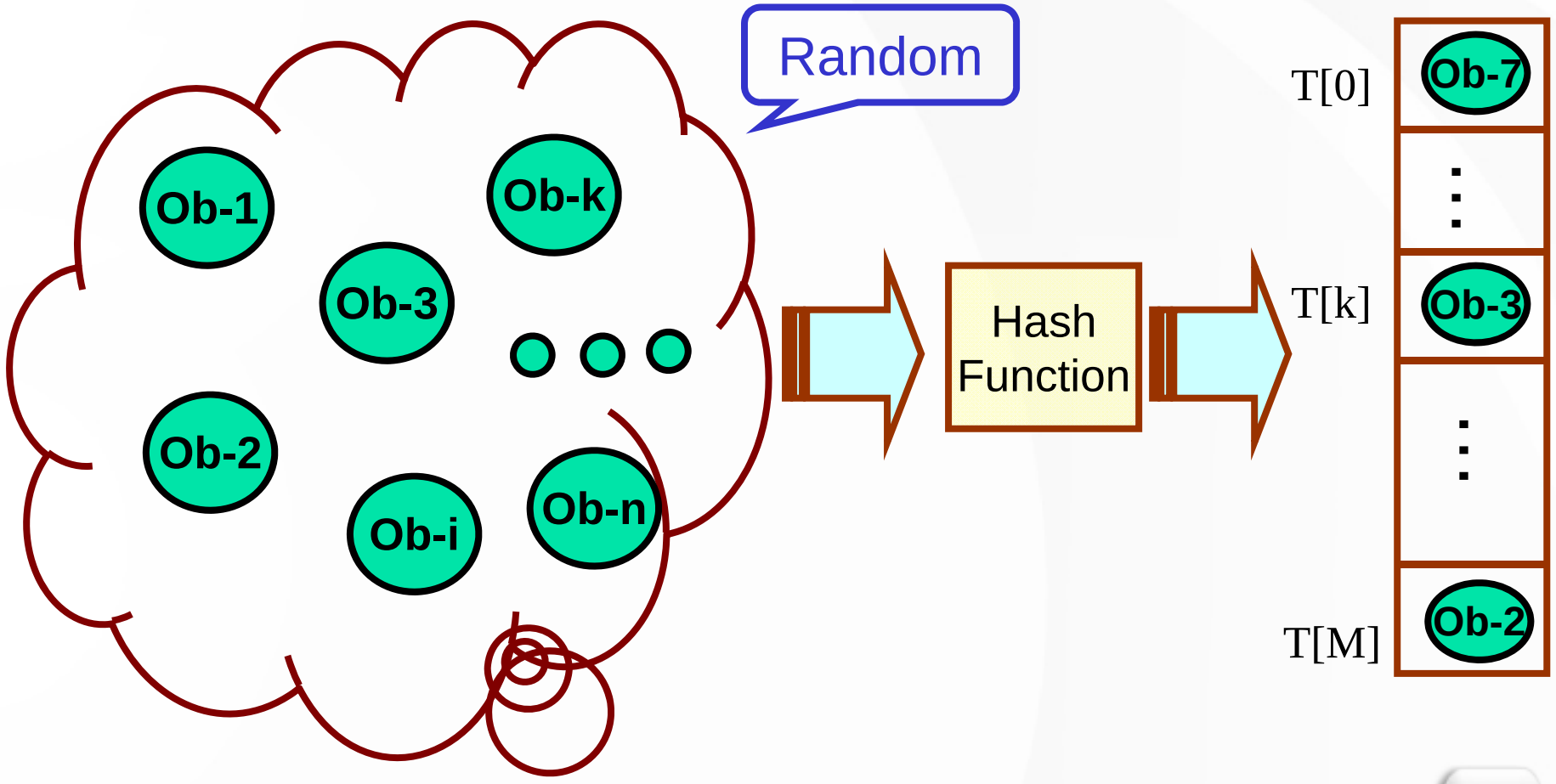
2 ID numbers per location
Collision?

Hash Function Re-considered

- ID numbers (Key)
  - Ordered? 1,5,25 → 25 is stored in the 3rd location
  - Random?
- Size of the ID set and the location set
  - More Keys?
  - More location numbers?
- Relationship between
  - Keys
  - Location numbers

Hash Functions and Collisions

Concept of Hash Function



Remainder Function

- Location = Key % table_size
  - Range: 0 – (table_size – 1)
  - Key Range: 0-99
  - More variation in remainder?

Collision!!
0 % 50 = 0
50 % 50 = 0

Other Methods

- Digit extraction function
- Shift function
- Exclusive-OR function

- Purpose
  - To have locations as **dispersed** as possible
  - To make the output as **random** as possible

Digit Extraction Function

- Assumption
  - Key: integers or real numbers
  - Containing “digits”
    - 12345
    - 12468.9
- Method
  - Predefined digits
  - E.g. H(key) = 2nd, 3rd, 5th digit
    - H(12345) = 235
    - H(12468.9) = 248

Shift Function

- Assumption
  - Digits representation
  - Binary representation
- Method
  - Predefined shift direction and number
  - E.g. H(key) = left shift 2 digits
    - H(12345) = 345
    - H(12345) = shift-left-2(11000000111001)
      = 000000111001 = 57

Exclusive-OR function

- Assumption
  - Key: subkey1, subkey2
  - Binary representation
- Method
  - H(key) = H(subkey1.subkey2)
    = subkey1 XOR subkey2
  - H(123.45) = 123 XOR 45
    = 01111011 XOR 00101101
    = 01010110 = 86

Recap

- As “disperse” as possible
- Combination of different methods

Key -> H1(key) → La
H2(La) → Lb

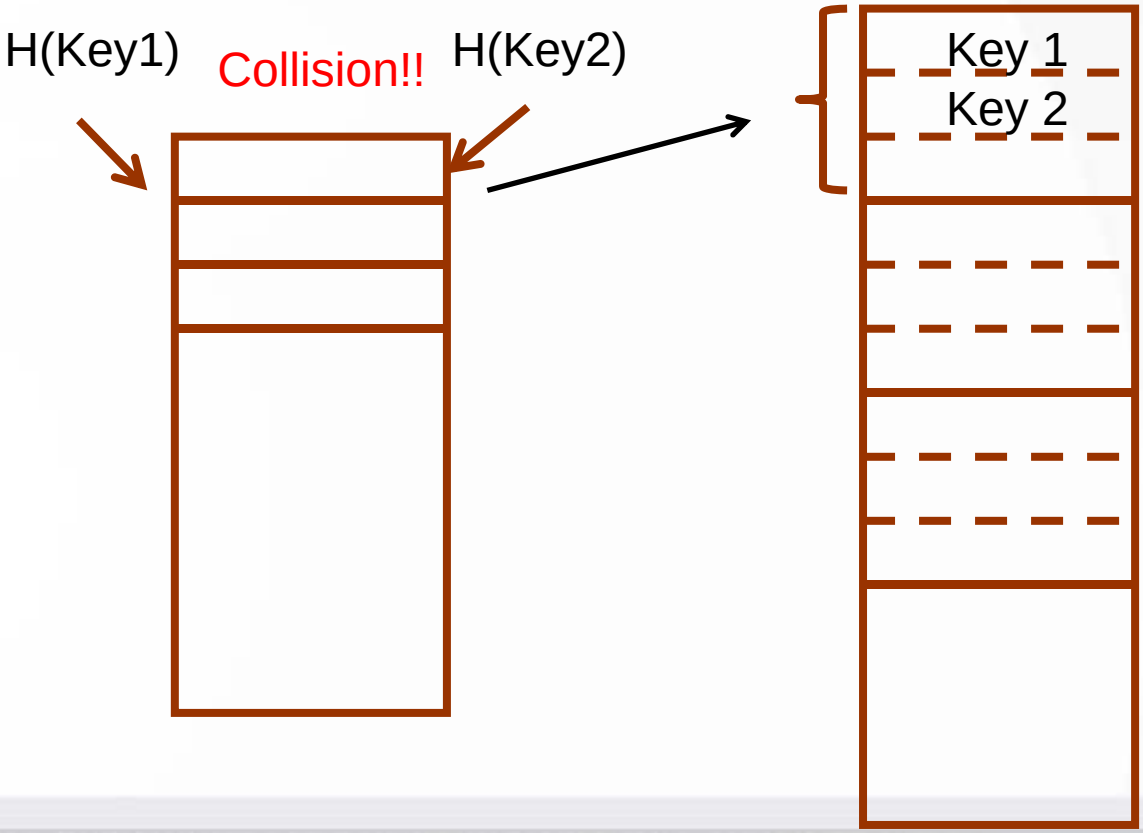
- Issues
  - Collision!

How to Deal with Collisions

- Buckets
- Chaining

Buckets

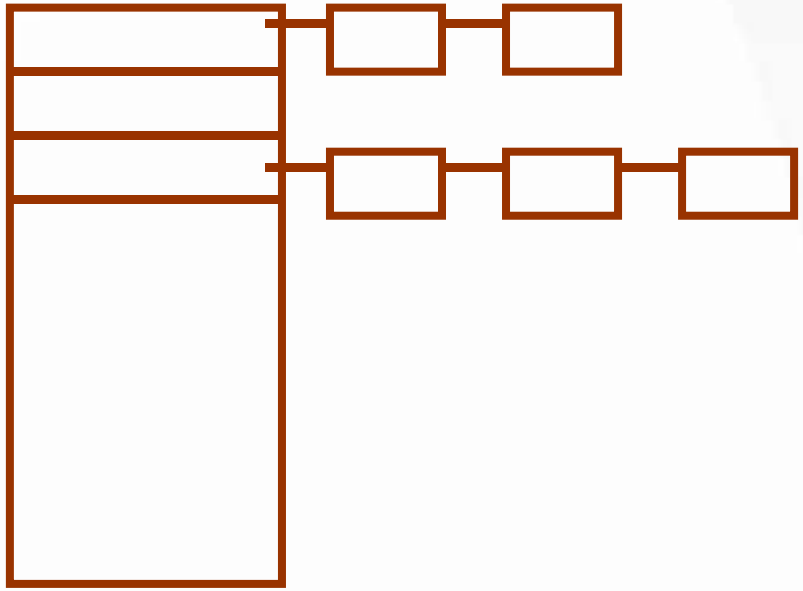
- Extra space in a location



Delaying Collisions

Chaining

- Creating a space in a location



Collision Resolution

Issues to Consider

- Clustering
  - Calculated locations: near-by
- Load factor
  - Percentage of occupied locations

$$\frac{\#\ of\ occupied\ location}{table\ size}$$

Probing Methods

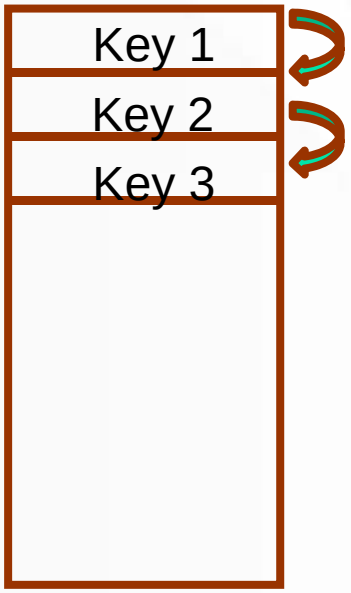
- Open addressing
  - Looking for an **available** location after collision
- Methods
  - Linear probing
  - Quadratic probing
  - Double hashing

Linear Probing

- If the location is occupied
  - Find next
  - Repeat if the next is still occupied
- H(key) + i, where i=1, 2, 3, ...
- Variation
  - i = -1, -2, -3, ...
  - i = 1, -2, 3, -4, ...

Observation on Linear Probing

- E.g. collision at H(key1)
  - H(key1) = H(key2) = H(key3)
  - Key1 at H(key1)
  - Key2 at H(key2) = H(key1)
    - New location: H(key2) + 1
  - Key3 at H(key3) = H(key2) = H(key1)
    - New location: H(key3) + 1
    - New location: H(key3) + 2
- Issues
  - Clustering
  - Probing path



Quadratic probing

- H(key) + i^2, where i = 1, 2, 3, ...
- E.g. collision after H(key)
  - H(key1) = H(key2) = H(key3)
  - Key1 at H(key1)
  - Key2 at H(key2) = H(key1)
    - New location: H(key2) + 1
  - Key3 at H(key3) = H(key2) = H(key1)
    - New location: H(key3) + 1
    - New location: H(key3) + 4
- Issues
  - Clustering, Probing path
  - Possibility of not using all locations

Double hashing

- H(key)
- Probing: H(H(key))
- E.g. H(key) = key % listSize
  - Collision at H(key)
  - New location: H(H(key)) = (key % listSize) % listSize
- Issues
  - Clustering, Probing path
  - Possibility of not using all locations

Retrieving Data from Hash Tables

- Looking up the data with “key”
  - Hash function: H(key)
  - Complexity: O(C)
- Collision, clustering
  - Looking up at H(key)
  - Looking up at H(key) + 1
  - Looking up at H(key) + 2
  - Looking up at H(key) + 3
  - ...
  - O(n)

How to Avoid the Same Probing Path

- Problem
  - Same collision resolution method
  - Same probing path
- Solution
  - Unique offset for a key
  - E.g. offset = key/listSize
    - H(key)
    - H(H(key)+offset)
    - ((key % listSize) + offset) % listSize