Chapter 9: Hashing

- Basic concepts
- Hash functions
- Collision resolution
- Open addressing
- Linked list resolution
- Bucket hashing

Basic Concepts

- Sequential search: $O(n)$
- Binary search: $O(\log_2 n)$

} Requiring several key comparisons before the target is found

Basic Concepts

- Search complexity:

Size	Binary	Sequential (Average)	Sequential (Worst Case)
16	4	8	16
50	6	25	50
256	8	128	256
1,000	10	500	1,000
10,000	14	5,000	10,000
100,000	17	50,000	100,000
1,000,000	20	500,000	1,000,000

Basic Concepts

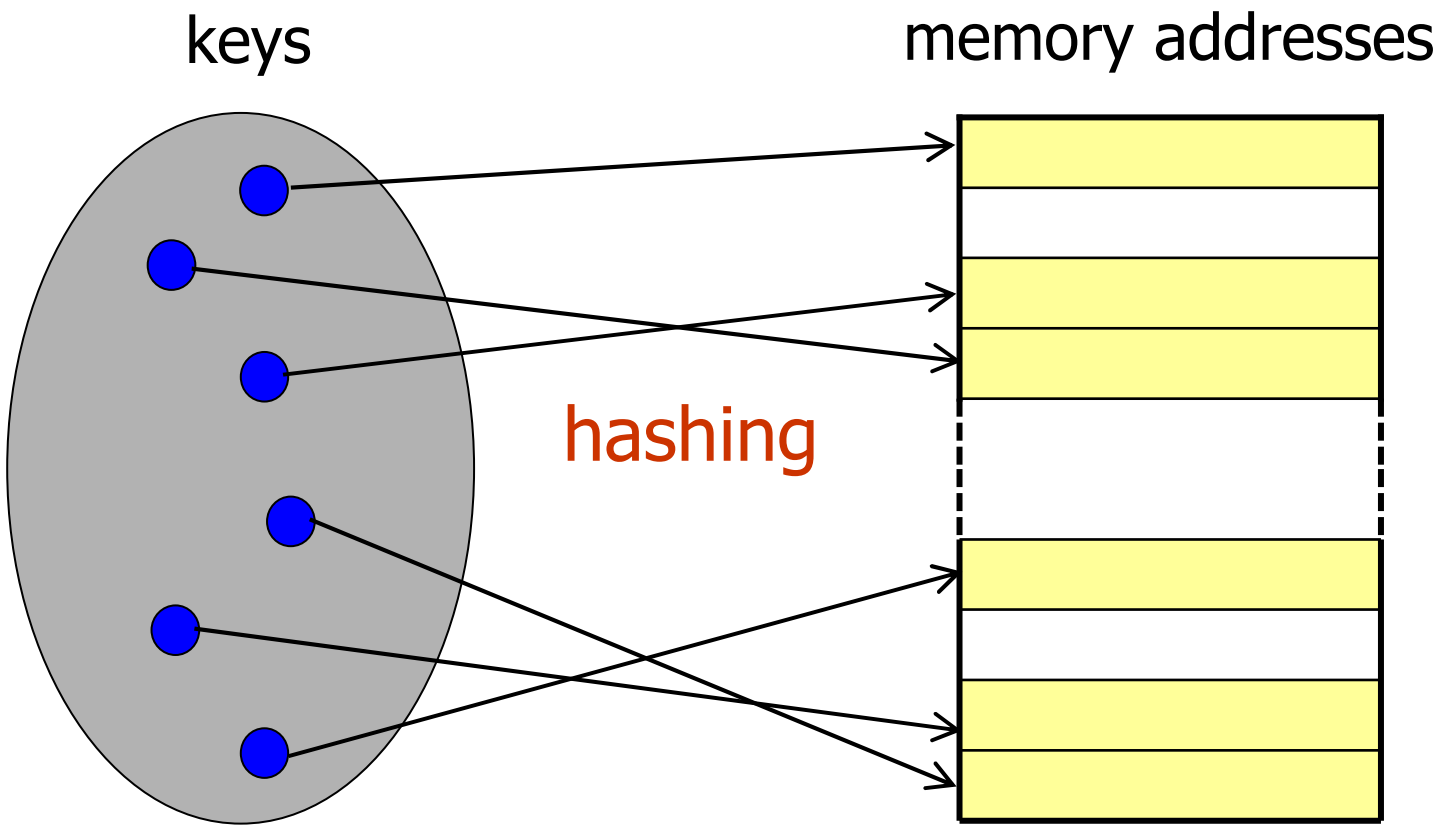
- Is there a search algorithm whose complexity is O(1)?

Basic Concepts

- Is there a search algorithm whose complexity is O(1)?

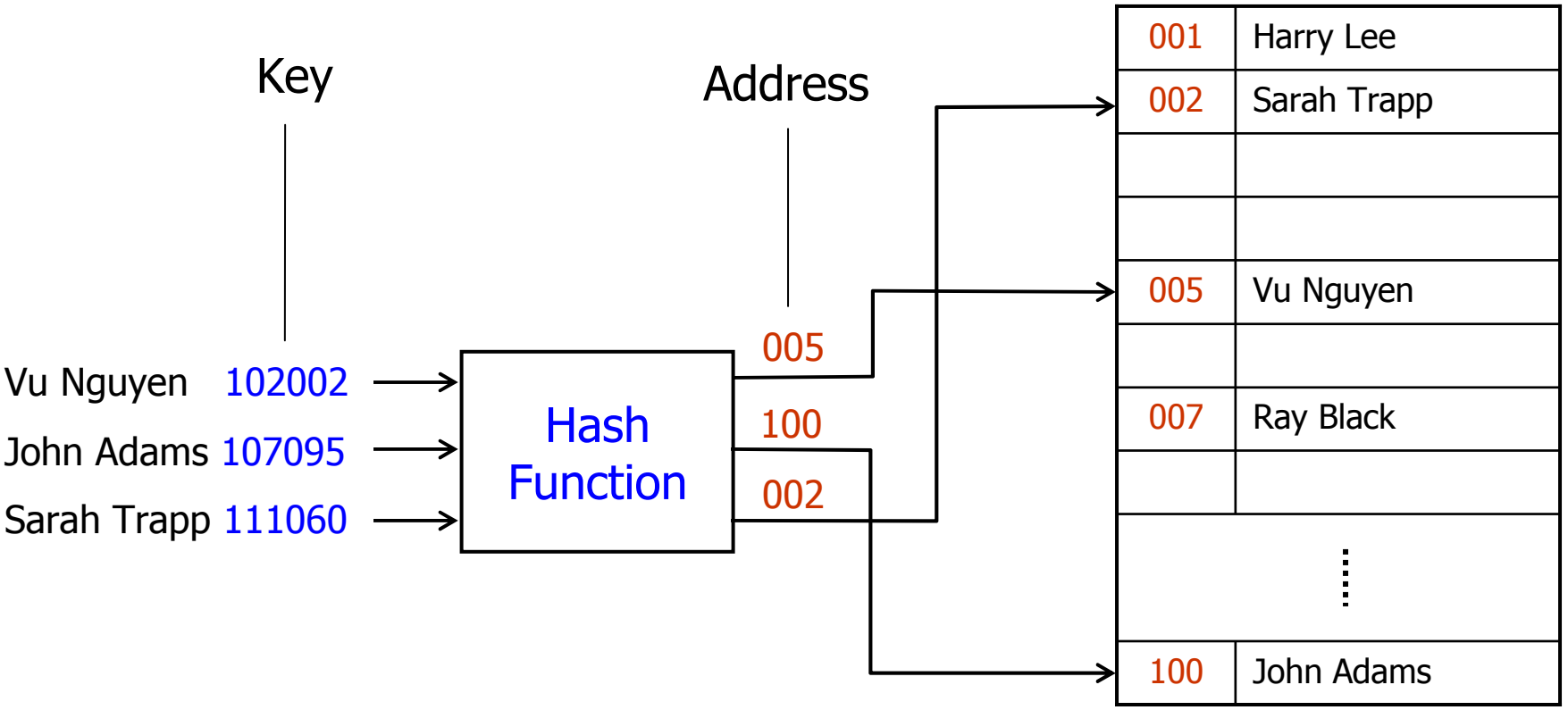
  YES.

Basic Concepts



Each key has only one address

Basic Concepts



Basic Concepts

- Home address: address produced by a hash function.
- Prime area: memory that contains all the home addresses.

Basic Concepts

- Synonyms: a set of keys that hash to the same location.
- Collision: the location of the data to be inserted is already occupied by the synonym data.

Basic Concepts

- Ideal hashing:
  - No location collision
  - Compact address space

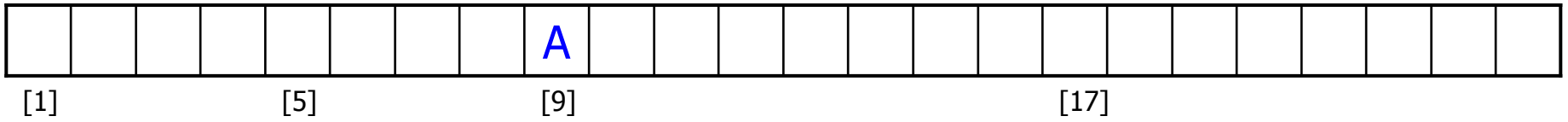
Basic Concepts

Insert A, B, C

hash(A) = 9

hash(B) = 9

hash(C) = 17



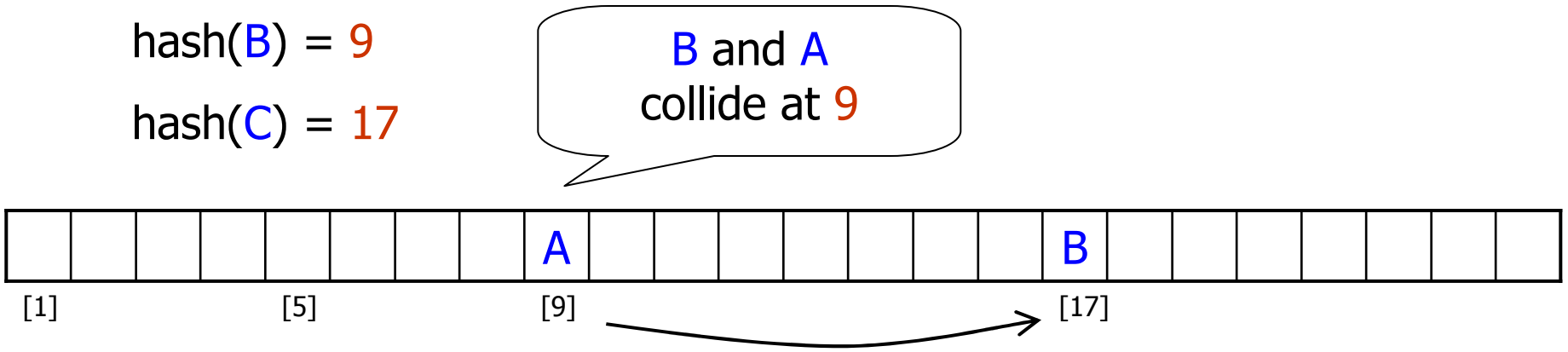
Basic Concepts

Insert A, B, C

hash(A) = 9

hash(B) = 9

hash(C) = 17



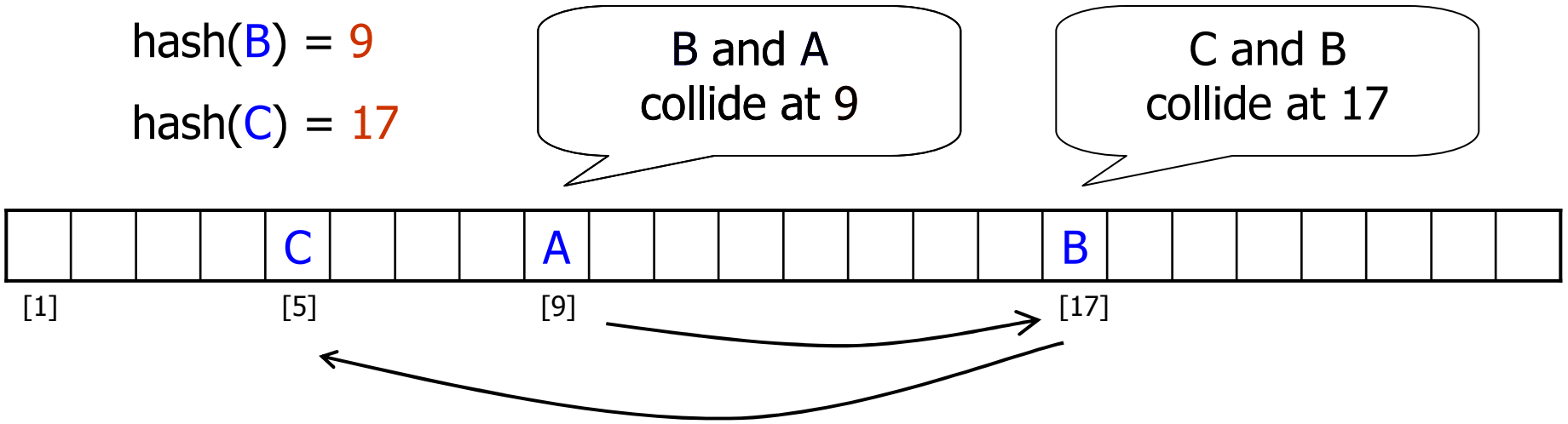
Basic Concepts

Insert A, B, C

hash(A) = 9

hash(B) = 9

hash(C) = 17



Collision Resolution

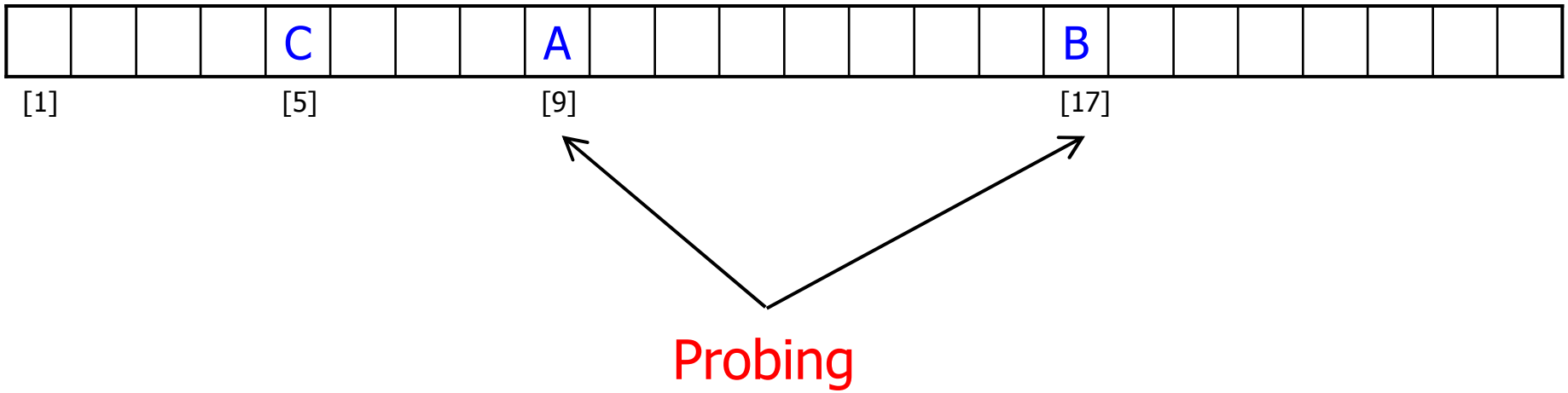
Basic Concepts

Searh for B

hash(A) = 9

hash(B) = 9

hash(C) = 17



Hash Functions

- Direct hashing
- Modulo division
- Digit extraction
- Mid-square
- Folding
- Rotation
- Pseudo-random

Direct Hashing

- The address is the key itself:

$$hash(Key) = Key$$

Direct Hashing

- Advantage: there is no collision.
- Disadvantage: the address space (storage size) is as large as the key space

Modulo Division

Address = Key MOD listSize + 1

- Fewer collisions if listSize is a prime number
- Example:

  Numbering system to handle 1,000,000 employees

  Data space to store up to 300 employees

  hash(121267) = 121267 MOD 307 + 1 = 2 + 1 = 3

Digit Extraction

Address = selected digits from Key

- Example:

$379452 \rightarrow 394$
$121267 \rightarrow 112$
$378845 \rightarrow 388$
$160252 \rightarrow 102$
$045128 \rightarrow 051$

Mid-square

Address = middle digits of Key^2

- Example:

$$9452 * 9452 = 89340304 \rightarrow 3403$$

Mid-square

- Disadvantage: the size of the Key^2 is too large
- Variations: use only a portion of the key

379452: 379 * 379 = 143641 $\rightarrow$ 364
121267: 121 * 121 = 014641 $\rightarrow$ 464
045128: 045 * 045 = 002025 $\rightarrow$ 202

Folding

- The key is divided into parts whose size matches the address size

  Key = 123|456|789

  <u>fold shift</u>

  123 + 456 + 789 = 1368

  ⇒ 368

Folding

- The key is divided into parts whose size matches the address size

Key = 123|456|789

fold shift

123 + 456 + 789 = 1368
⇒ 368

fold boundary

321 + 456 + 987 = 1764
⇒ 764

Rotation

- Hashing keys that are identical except for the last character may create synonyms.

- The key is rotated before hashing.

original key	rotated key
600101	160010
600102	260010
600103	360010
600104	460010
600105	560010

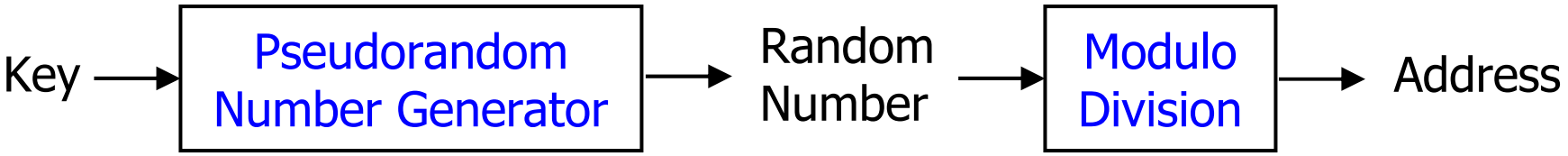
Rotation

- Used in combination with fold shift

original key	rotated key
600101 → 62	160010 → 26
600102 → 63	260010 → 36
600103 → 64	360010 → 46
600104 → 65	460010 → 56
600105 → 66	560010 → 66

Spreading the data more evenly across the address space

Pseudorandom



$$y = ax + c$$

For maximum efficiency, a and c should be prime numbers

Pseudorandom

- Example:

Key = 121267 a = 17 c = 7 listSize = 307

Address = ((17*121267 + 7) MOD 307 + 1

= (2061539 + 7) MOD 307 + 1

= 2061546 MOD 307 + 1

= 41 + 1

= 42

Collision Resolution

- Except for the direct hashing, none of the others are one-to-one mapping

  $\Rightarrow$ Requiring collision resolution methods

- Each collision resolution method can be used independently with each hash function

Collision Resolution

- A rule of thumb: a hashed list should not be allowed to become more than 75% full.

  Load factor:

  $$\alpha = (k/n) \times 100$$

  n = list size
  k = number of filled elements

Collision Resolution

- As data are added and collisions are resolved, hashing tends to cause data to group within the list

  $\Rightarrow$ Clustering: data are unevenly distributed across the list

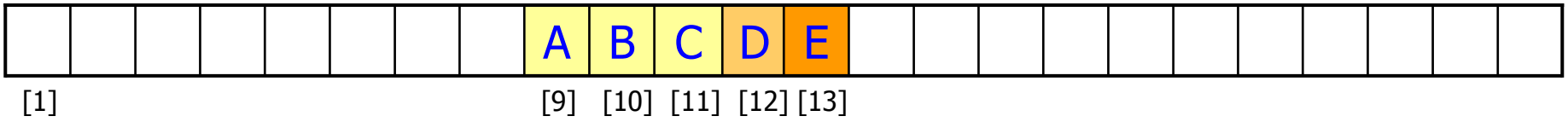
- High degree of clustering increases the number of probes to locate an element

  $\Rightarrow$ Minimize clustering

Collision Resolution

- Primary clustering: data become clustered around a home address.

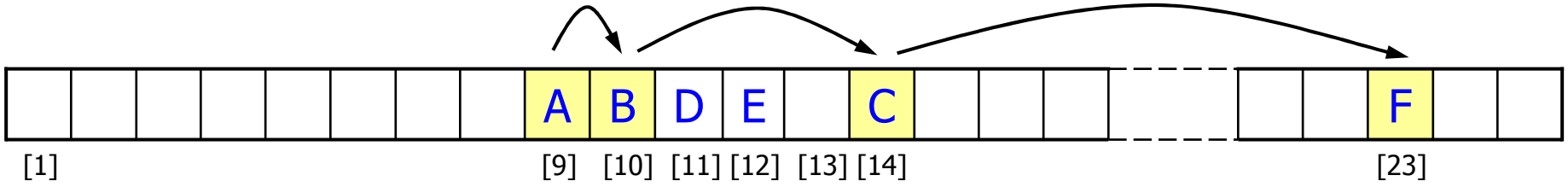
  Insert A_9, B_9, C_9, D_{11}, E_{12}



Collision Resolution

- Secondary clustering: data become grouped along a collision path throughout a list.

  Insert A_9, B_9, C_9, D_{11}, E_{12}, F_9



Collision Resolution

- Open addressing
- Linked list resolution
- Bucket hashing

Open Addressing

- When a collision occurs, an unoccupied element is searched for placing the new element in.

Open Addressing

- Hash function:

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

set of keys — addresses

Open Addressing

- Hash and probe function:

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

set of keys | probe numbers | addresses

Open Addressing

Algorithm hashInsert (ref T <array>, val k <key>)

Inserts key k into table T

```
1   i = 0
2   loop (i < m)
    1   j = hp(k, i)
    2   if (T[j] = nil)
        1   T[j] = k
        2   return j
    3   else
        1   i = i + 1
3   return error: "hash table overflow"
End  hashInsert
```

Open Addressing

Algorithm hashSearch (val T <array>, val k <key>)

Searches for key k in table T

```
1   i = 0
2   loop (i < m)
    1   j = hp(k, i)
    2   if (T[j] = k)
        1   return j
    3   else if (T[j] = nil)
        1   return nil
    4   else
        1   i = i + 1
3   return nil
End     hashSearch
```

Open Addressing

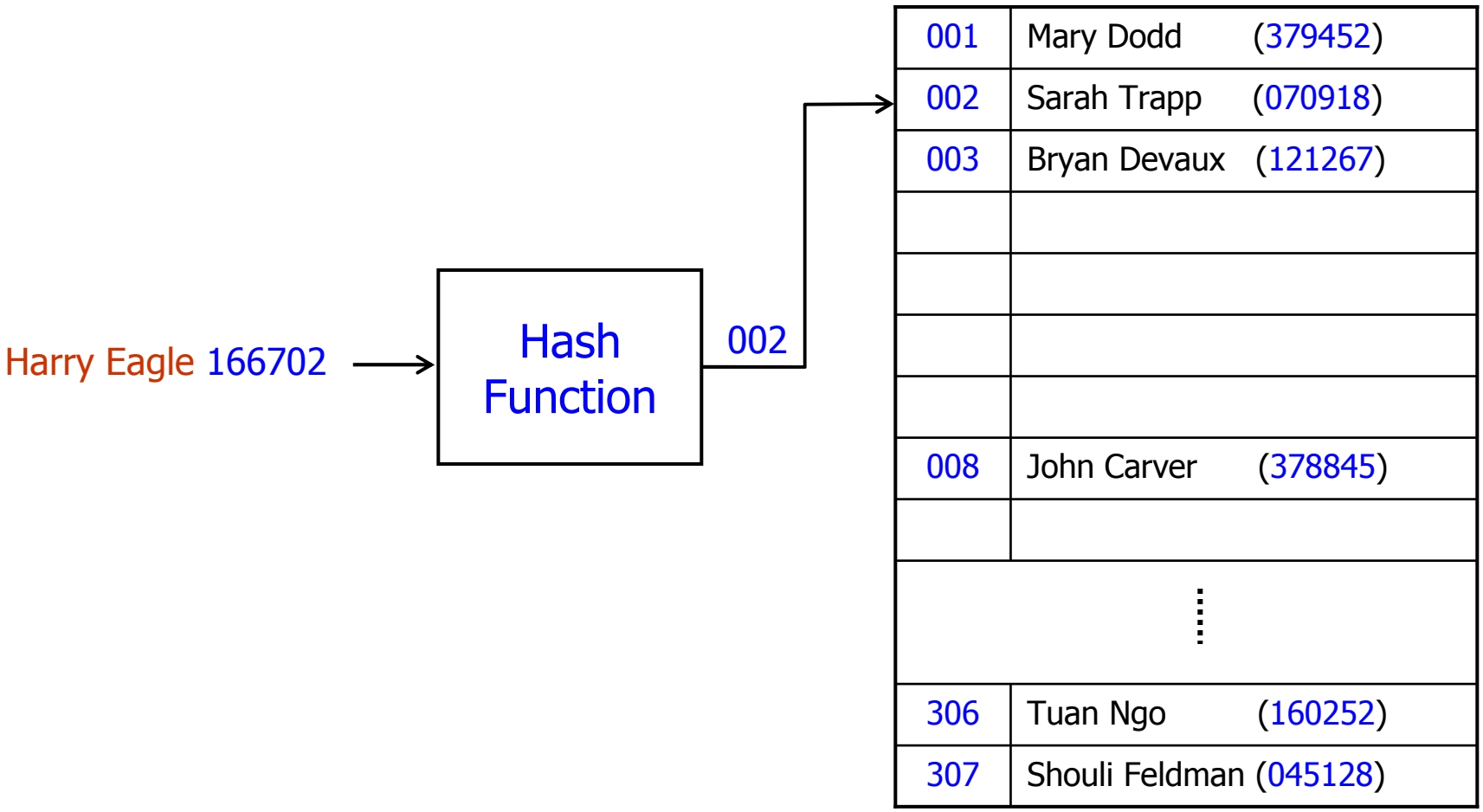
- There are different methods:
  - Linear probing
  - Quadratic probing
  - Double hashing
  - Key offset

Linear Probing

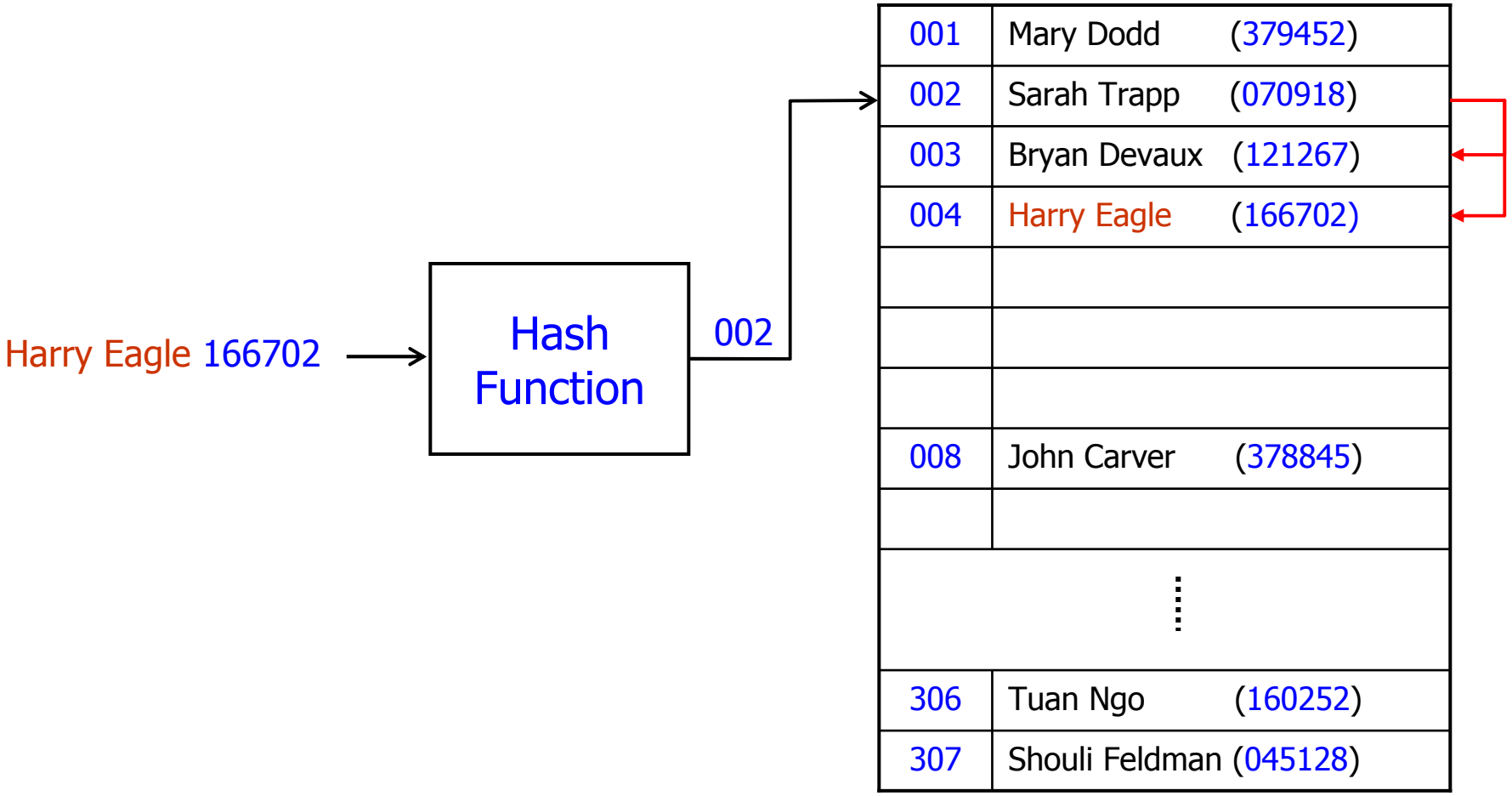
- When a home address is occupied, go to the next address (the current address + 1):

$$hp(k, i) = (h(k) + i) \text{ MOD } m$$

Linear Probing



Linear Probing



Linear Probing

- Advantages:
  - quite simple to implement
  - data tend to remain near their home address (significant for disk addresses)

- Disadvantages:
  - produces primary clustering

Quadratic Probing

- The address increment is the collision probe number squared:

$$hp(k, i) = (h(k) + i^2) \text{ MOD } m$$

Quadratic Probing

- Advantages:
  - works much better than linear probing

- Disadvantages:
  - time required to square numbers
  - produces secondary clustering

$$h(k_1) = h(k_2) \Rightarrow hp(k_1, i) = hp(k_2, i)$$

Double Hashing

- Using two hash functions:

$$hp(k, i) = (h_1(k) + ih_2(k)) \text{ MOD } m$$

Key Offset

- The new address is a function of the collision address and the key.

  offset = [key / listSize]

  newAddress = (collisionAddress + offset) MOD listSize

Key Offset

- The new address is a function of the collision address and the key.

offset = [key / listSize]

newAddress = (collisionAddress + offset) MOD listSize

$$hp(k, i) = (hp(k, i-1) + [k/m]) \text{ MOD } m$$

Open Addressing

- Hash and probe function:

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

set of keys | probe numbers | addresses

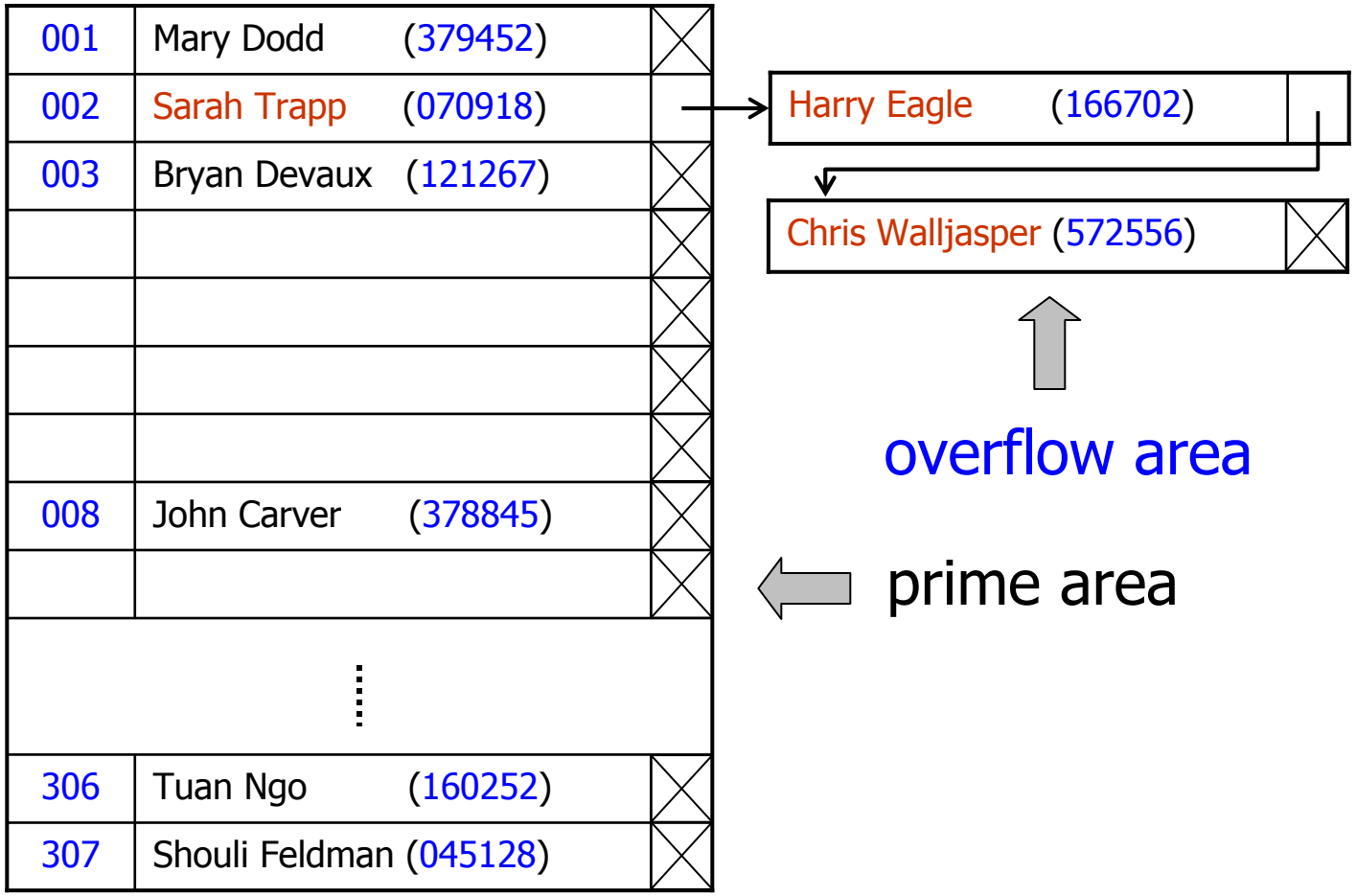
$\langle hp(k,0), hp(k,1), \ldots, hp(k,m-1) \rangle$ is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$

Linked List Resolution

- Major disadvantage of Open Addressing: each collision resolution increases the probability for future collisions.

  $\Rightarrow$ use linked lists to store synonyms

Linked List Resolution

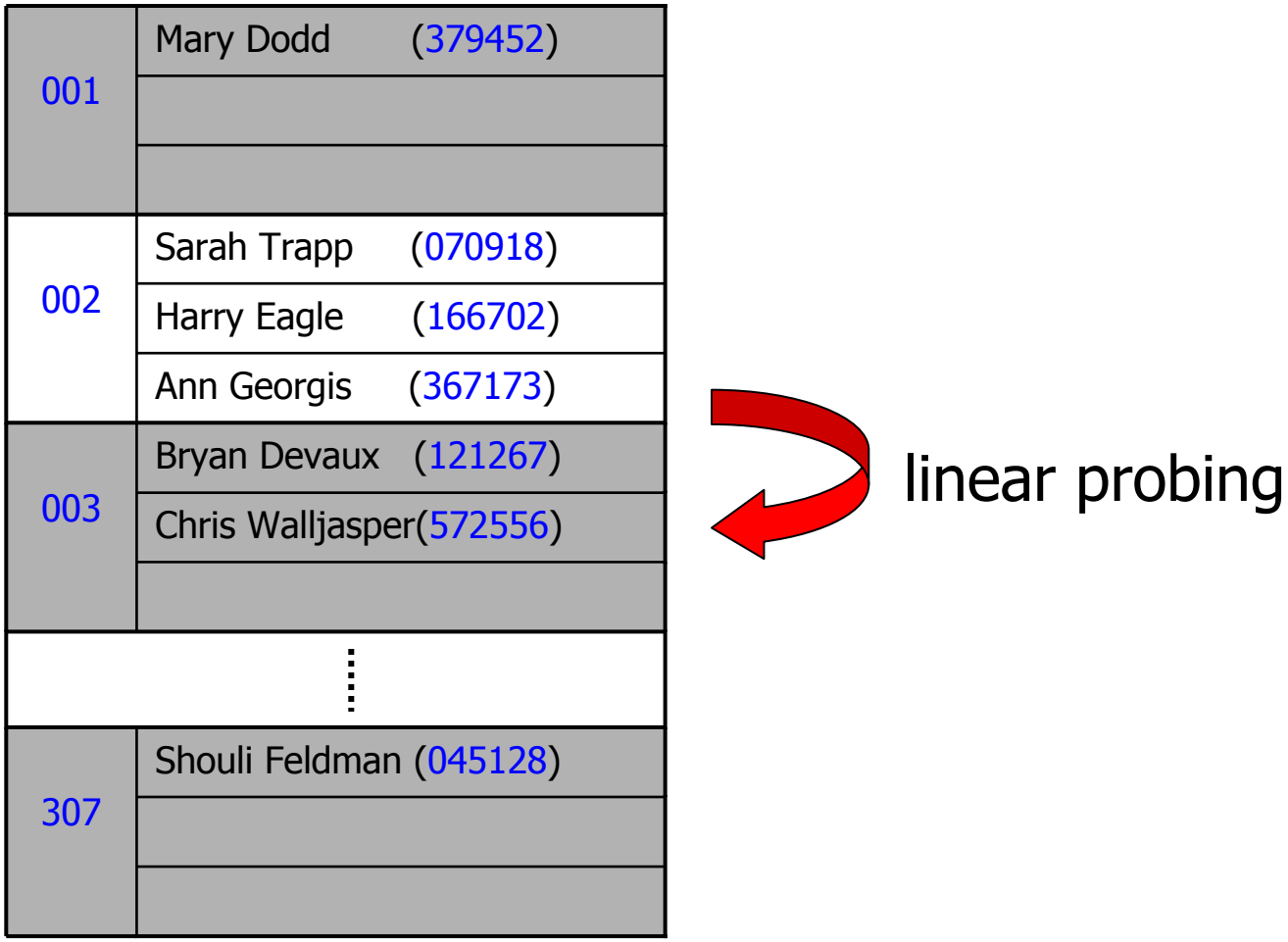


Bucket Hashing

- Hashing data to buckets that can hold multiple pieces of data.

- Each bucket has an address and collisions are postponed until the bucket is full.

Bucket Hashing



Indexing = Hashing