Data structures and Algorithms

HASH TABLES

Nguyễn Ngọc Thảo
nnthao@fit.hcmus.edu.vn

ALLPPT.com _ Free PowerPoint Templates, Diagrams and Charts

Outline

- Hash tables: A definition
- Hashing functions: Examples
- Collision resolution schemes
- The efficiency of hashing

Hash tables: A definition

A faster search is needed

- Rapid search can be vital in certain situations.

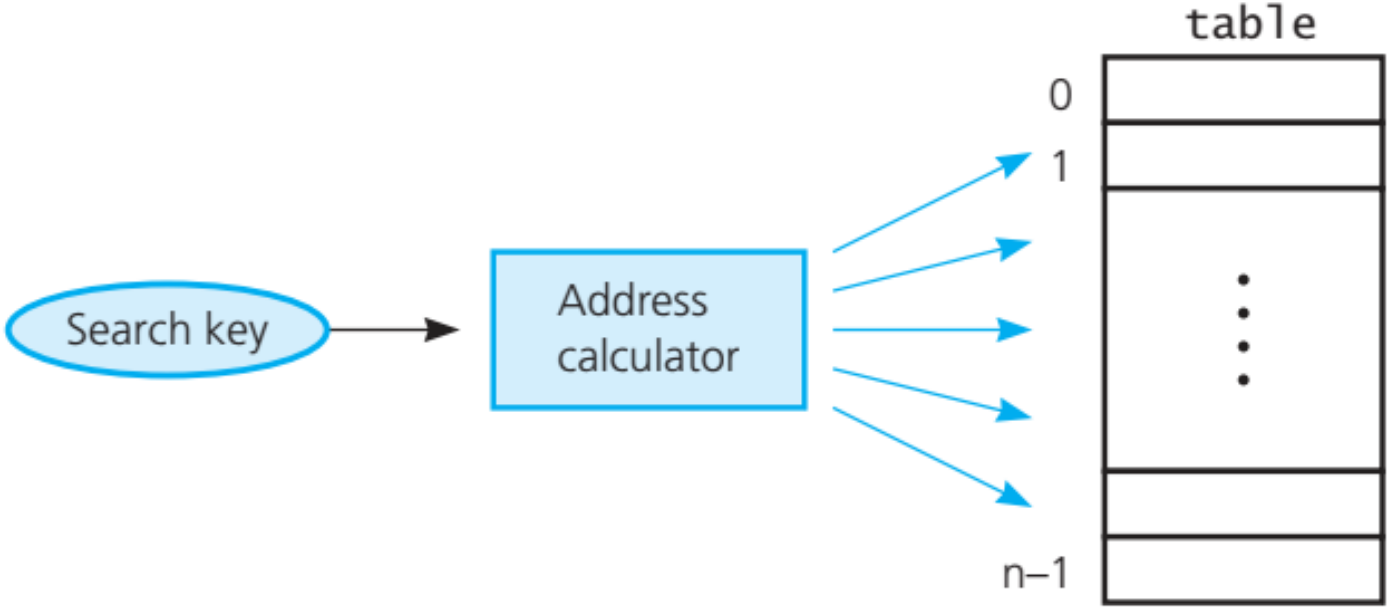
A person calls the 911, the system detects the caller's number and searches a database for the caller's address.

An air traffic control system searches a database of flight information, given a flight number

- A radically different strategy is necessary to locate (and insert or remove) an item virtually instantaneously.

The “address calculator” magic box

- Imagine that we could have an array table of N elements, with each slot capable of holding a single data item
  - That is, whenever a new item is inserted into the array, this magic box will tell you where you should place it.



```
add(newItem: ItemType): boolean
   i = the array index for newItem's search key given by the address
   calculator
   table[i] = newItem
```

```
getItem(searchKey: KeyType): ItemType throw NotFoundException
   i = the array index that the address calculator gives you for an item
   whose search key equals searchKey
   if (table[i].getKey() == searchKey)
     return table[i]
   else throw NotFoundException
```

```
remove(searchKey: KeyType): boolean
  i = the array index given for an item whose search key equals searchKey
  if (table[i].getKey() == searchKey){
    Remove the iter
    isSuccessful =
  }
  else
    isSuccessful =
  return isSuccessf
```

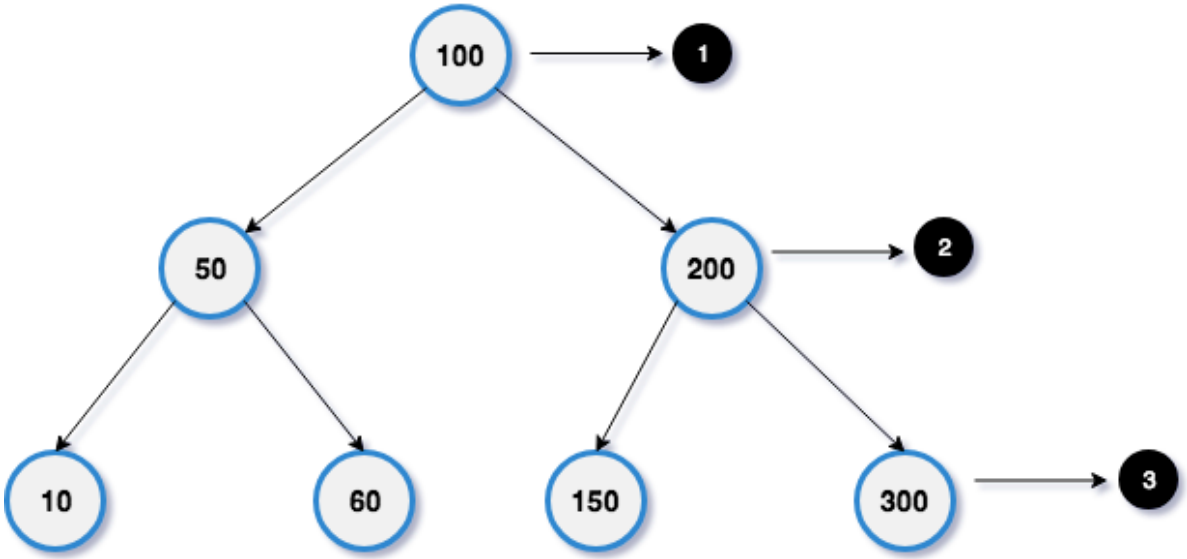
- **getItem**, **add**, and **remove**, approach $\boldsymbol{O(1)}$.
- The overall computational cost depends only on how quickly the address calculator decides where the item should be.

The “address calculator” magic box

- The scheme just described is an idealized description of a technique known as **hashing**.
- Such address calculator is referred to as a **hash function**.
- The array table is called the **hash table**.

How a hash function works

- Consider the 911 emergency system mentioned earlier.
- For each person, the system had a record whose search key was the person's telephone number.
- These records could be stored in a search tree $\rightarrow$ basic retrieval operations are $O(\log_2 n)$.

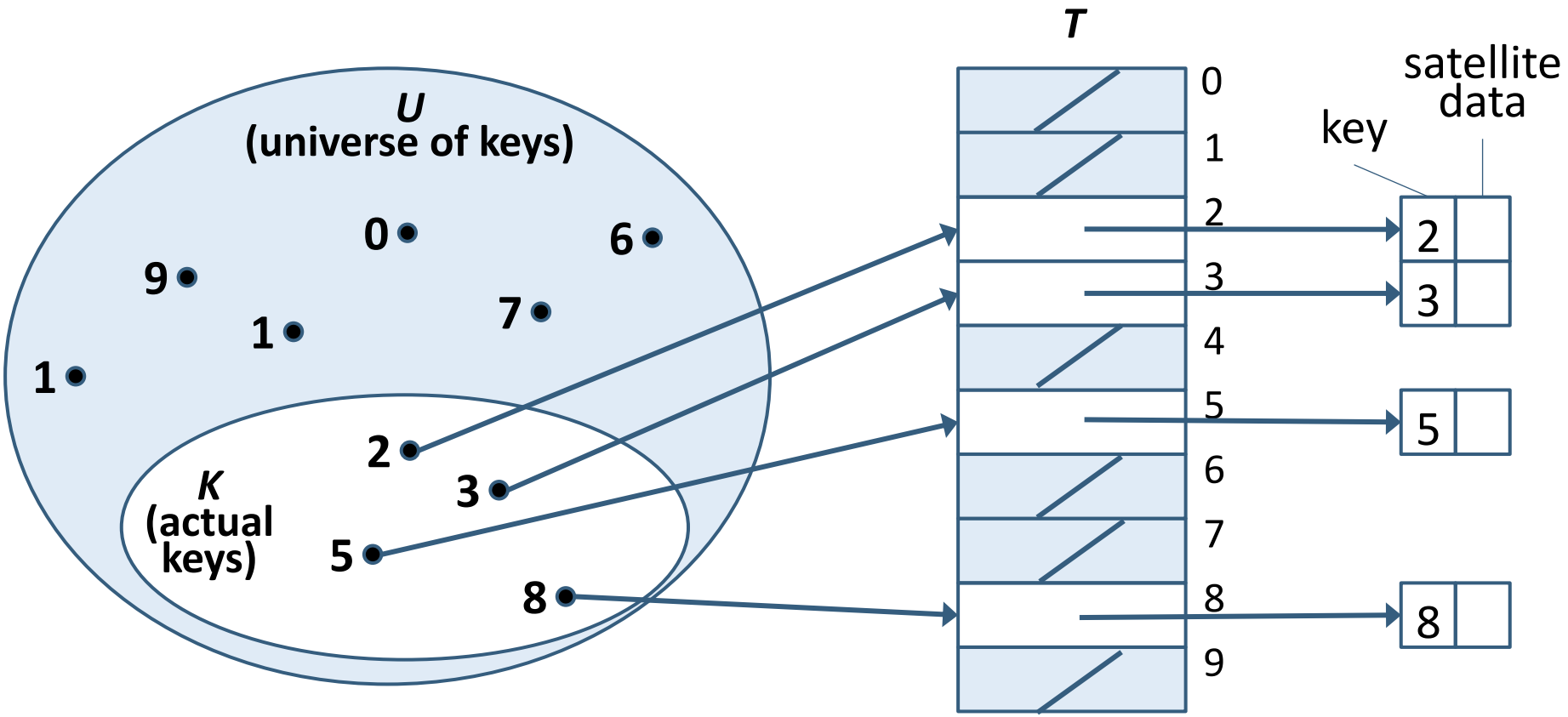


How a hash function works

- Faster access to a particular record would be possible with an array table $\rightarrow$ search time takes $O(1)$.
- For example, the telephone number 123-4567 could be stored in either of the two following ways
  - Each (whole) number is in a single slot, e.g., table[1234567] $\rightarrow$ 10 million slots required
  - Store only telephone exchange in a single slot, e.g., let 123-4567 be in table[4567] $\rightarrow$ 10,000 slots required
- The transformation from 1234567 into an array index 4567 is a simple example of a hash function.

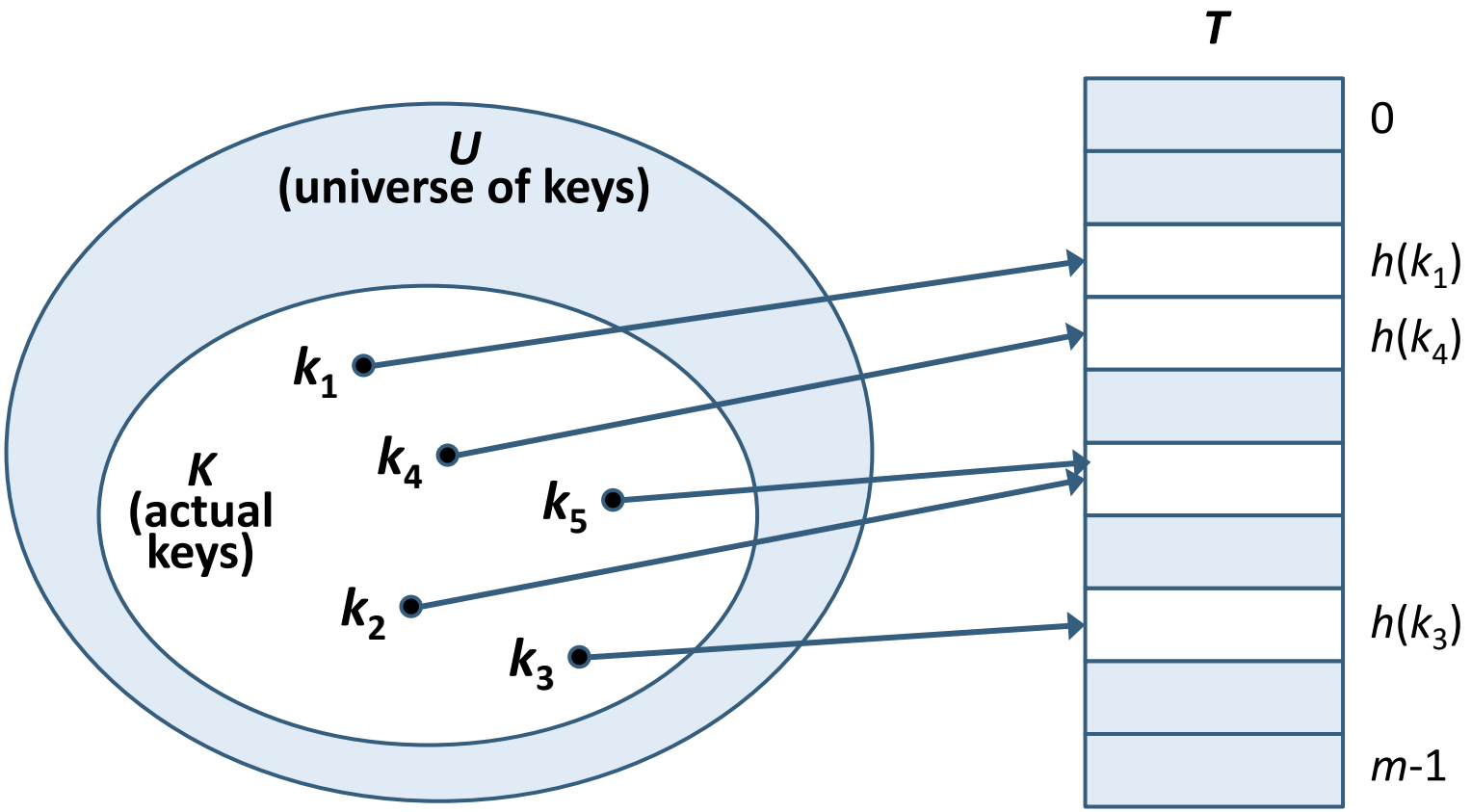
Direct-address tables

- Records are placed using their key values directly as indexes.
- The operations, **`getItem`**, **`add`**, and **`remove`**, are all $O(1)$.



Hashing techniques

- A hash function tells you where to place an item in an array called a hash table.



Hashing techniques

- A hash function h must take an arbitrary integer $searchKey$ and map it into an array index.

```
i = h(searchKey)
```

- A typical hash function must be easy and fast to compute and place items evenly throughout the hash table.

Hash functions: Examples

Digit-selection hash functions

- Simple and fast, yet do not evenly distribute the items in the hash table in general
- For example, consider the nine-digit ID number 001364825
  - Select the fourth digit and the last digit $\rightarrow h(001\mathbf{3}6482\mathbf{5}) = 35$
  - The item is “hashed into” the location 35 of the hash table.

- Pay attention to which digits you choose in certain situations
  - E.g., the first three digits of a Social Security number are for geographic region $\rightarrow$ selecting only these digits will map all people from the same state into the same location of the hash table.

Folding hash functions

- Improve the previous approach by adding more digits
- For example, consider the ID number 001364825 again
  - Add all the digits $\rightarrow h = 0 + 0 + 1 + 3 + 6 + 4 + 8 + 2 + 5 = 29$
  - The item is “hashed into” the location 29 of the hash table.
- We have different strategies for adding those nine digits.
  - Add all the digits: $0 \leq h(searchKey) \leq 81$
  - Form three 3-digit groups from the search key, e.g., $001 + 364 + 825$

    $$0 \leq h(searchKey) \leq 3 \times 999 = 2{,}997$$
  - And any other strategy that you can think of…
- You can apply more than one hash function to a search key.
  - E.g., select digits from 2997 or apply folding to it (29-97) once again

Modular hash functions

- Modulo arithmetic provides a simple and effective hash function that is most used.
- The function is $\boldsymbol{h(searchKey) = searchKey \bmod tableSize}$
  - where $tableSize$ is the size of the hash table.
  - If $tableSize$ is 101, $h(k) = k \bmod 101$ maps any integer $searchKey$ into the range [0,100], e.g., h maps 001364825 into 12.
- A prime number for the table size could help distributing the items evenly, and thus reducing collisions.

Character string to an integer

- If the search key is a character string, you could convert it into an integer before applying the hash function.
- First assign an integer value to each character in the string

Word	N	O	T	E
ASCII values	78	79	84	69
Values in [1,26]	14	15	20	5

- Simply adding these numbers gives an integer that is not unique to the character string
  - E.g., NOTE has the same result as TONE

Character string to an integer

- Write the binary value for each character and concatenate the results → an overflow could occur if large values

Word	N	O	T	E
Values in [1,26]	14	15	20	5
Binary values	01110	01111	10100	00101

$$k = 01110011111010000101_b = 474{,}757_d$$

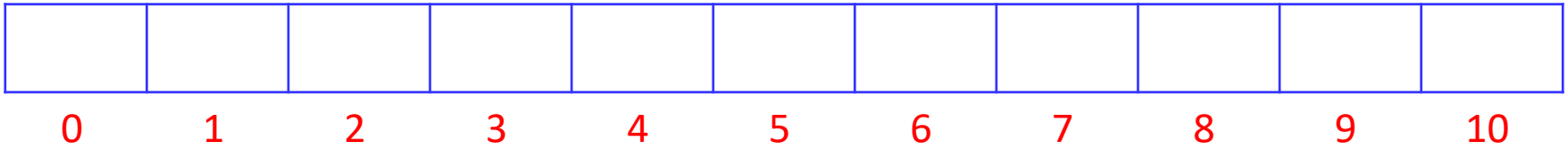
- Horner's rule prevents an overflow by applying the modulo operator after computing each parenthesized expression.

$$474{,}757 = 14 \times 32^3 + 15 \times 32^2 + 20 \times 32^1 + 5 \times 32^0$$

$$= ((14 \times 32 + 15) \times 32 + 20) \times 32 + 5$$

Checkpoint 01: Calculate the hash values for integers

Consider a hash table of $m = 11$ slots with hash function $h(k) = k \bmod m$.



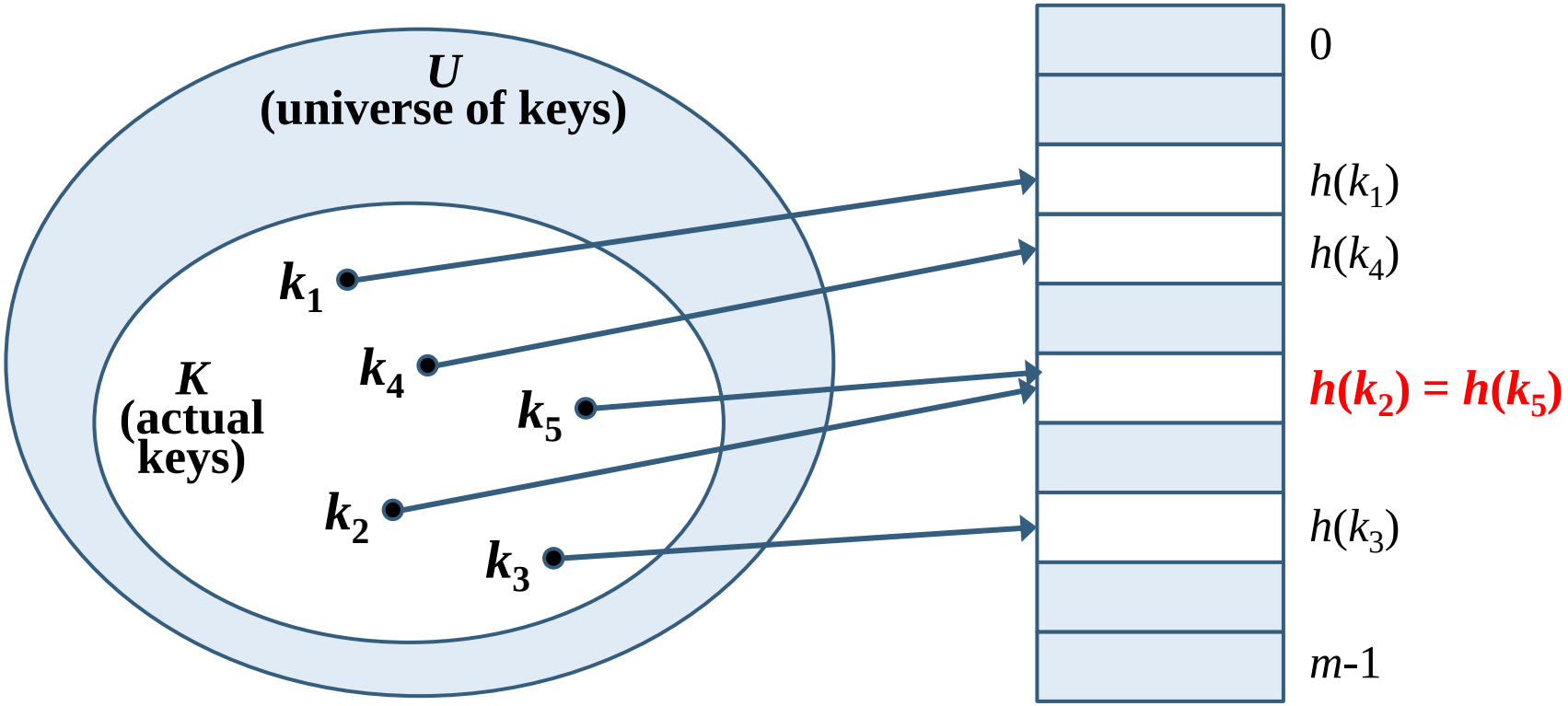
You are given an array of integers, 54, 77, 94, 89, 14.

For each element, identify which location in the hash table it should be inserted to?

Collision resolution schemes

What are collisions?

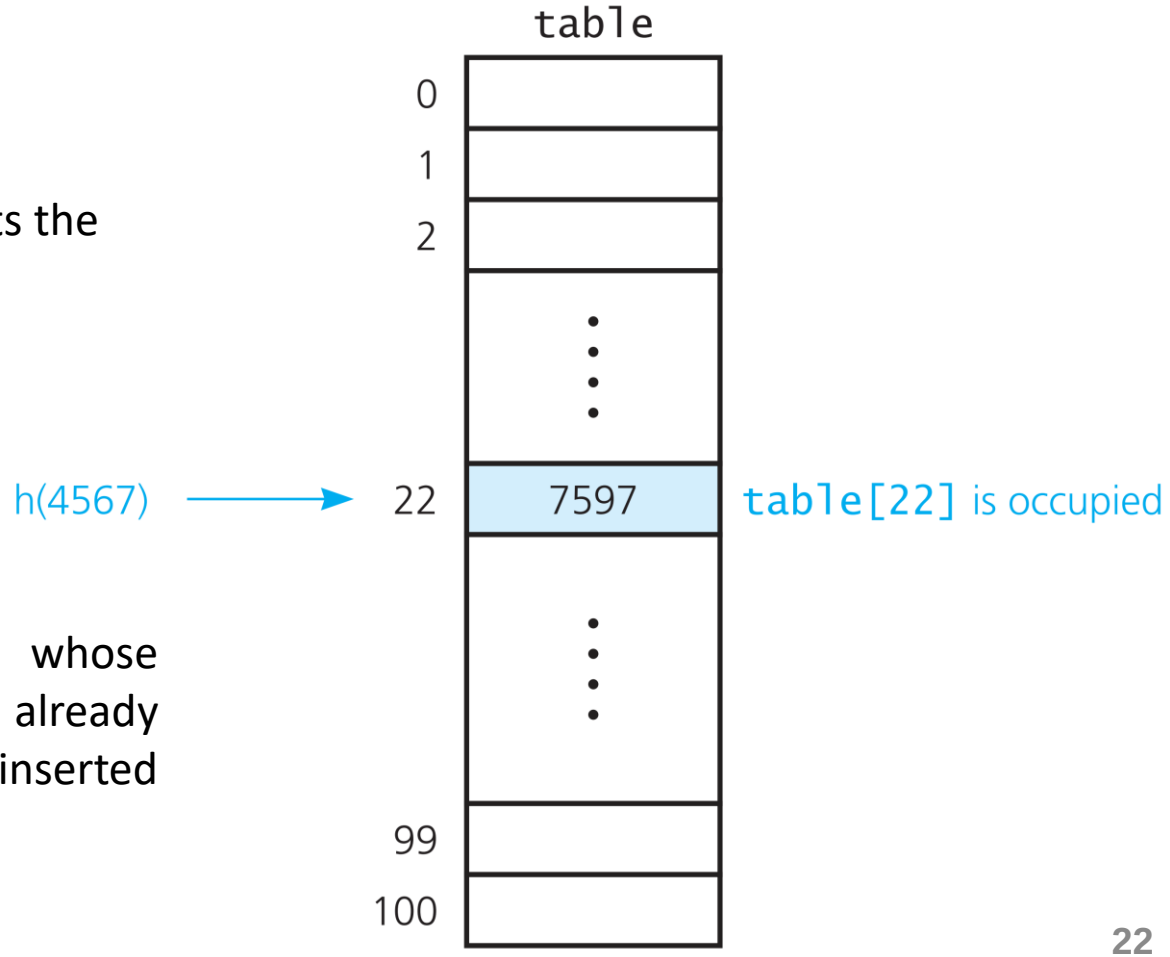
- The hash function tells you to store two or more items in the same array location.
- $\exists k_1, k_2 \in K\colon\ k_1 \neq k_2, h(k_1) = h(k_2)$



Example: An example of collisions

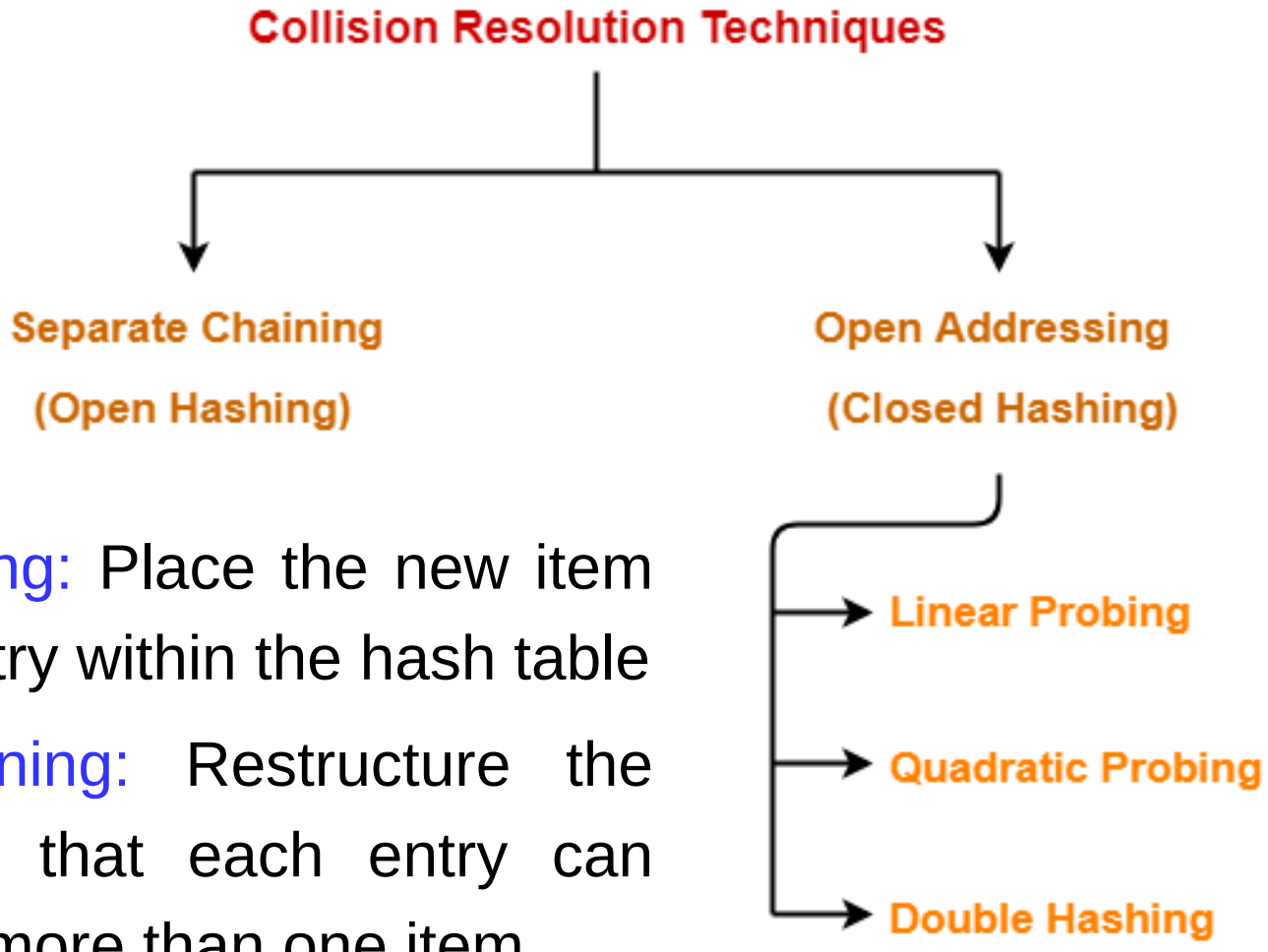
Suppose that you want to insert an item whose search key is 4567 into the hash table.

$h(k) = k \bmod 101$ directs the new item to $table[22]$.



However, another item whose search key is 7597 is already there because it was inserted before the new item.

Approaches to collision resolution



- Open addressing: Place the new item into another entry within the hash table
- Separate chaining: Restructure the hash table so that each entry can accommodate more than one item

Open addressing techniques

- If the hash function indicates a location already occupied, probe for another open location to place the item.
- Probe sequence: the series of locations that you examine.
- **`remove`** and **`getItem`** must efficiently reproduce the probe sequence that **`add`** used.

Open addressing: Linear probing

- Search the hash table sequentially, starting from the original hash location, until you find an available location
  - If $table[h(k)]$ is occupied, check the locations $table[h(k)+1]$, $table[h(k)+2]$, and so on.
- The probe sequence follows

$$\boldsymbol{(h(k)+i)\ mod\ tableSize\ with\ i=0,1,2,\ldots}$$

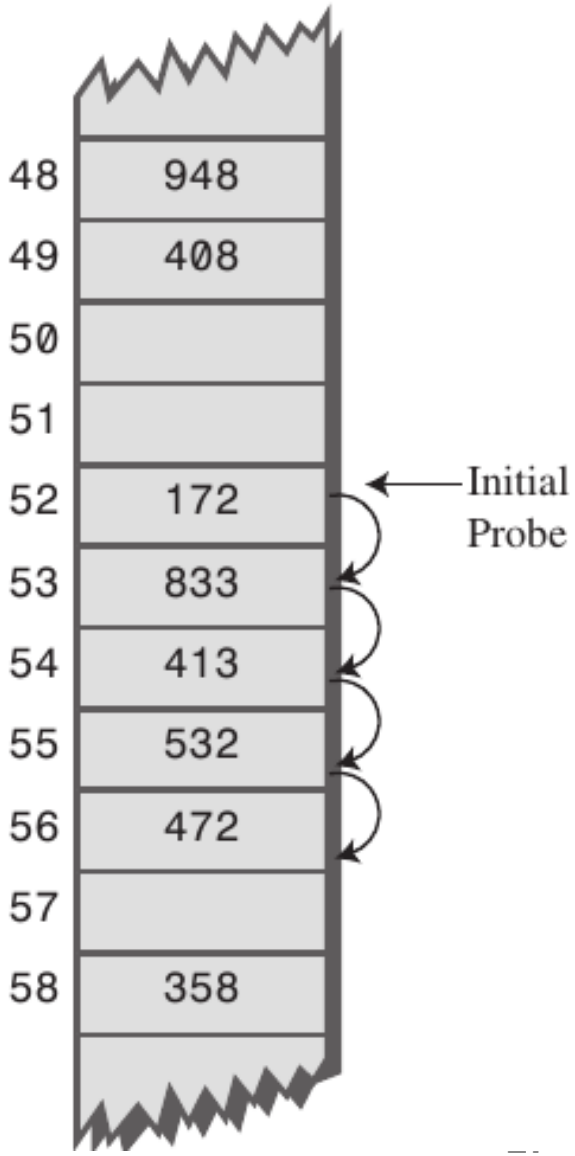
- Wrap around from the last array location to the first array location if necessary.

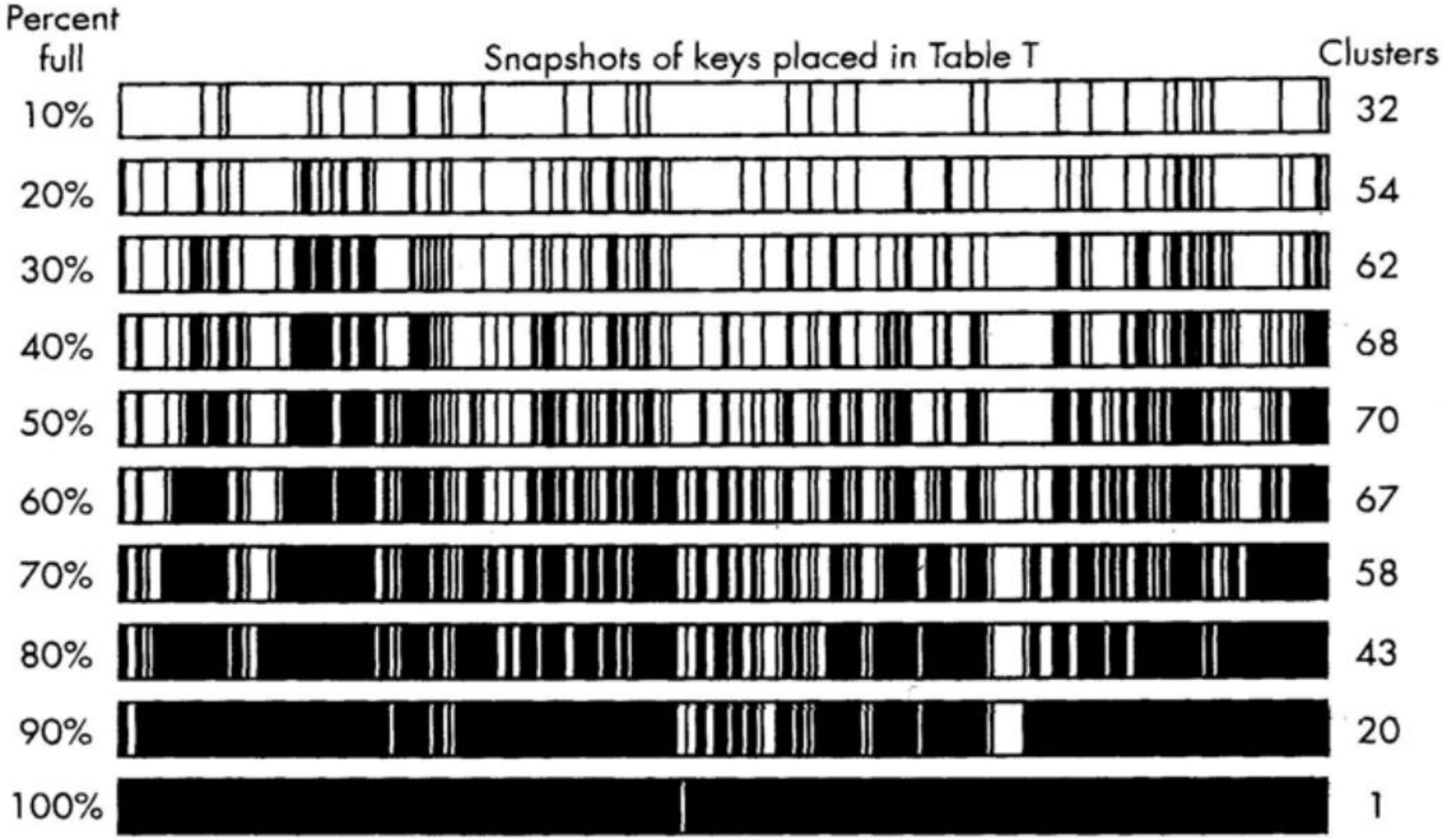
Open addressing: Linear probing

- **`getItem`** follows the same probe sequence that **`add`** used until
  - The item in consideration is found, or
  - The item is not present: the procedure has reached an empty location or visited every location.
- **`remove`** creates new empty entries along a probe sequence, which could cause **`getItem`** to stop prematurely.
  - A table location can be *occupied* (currently in use), *empty* (has not been used), or *removed* (once occupied but now available).
  - **`getItem`** continues probing when it encounters a "*removed"* location.
  - **`add`** inserts items into locations that are either "*empty*" or "*removed*".

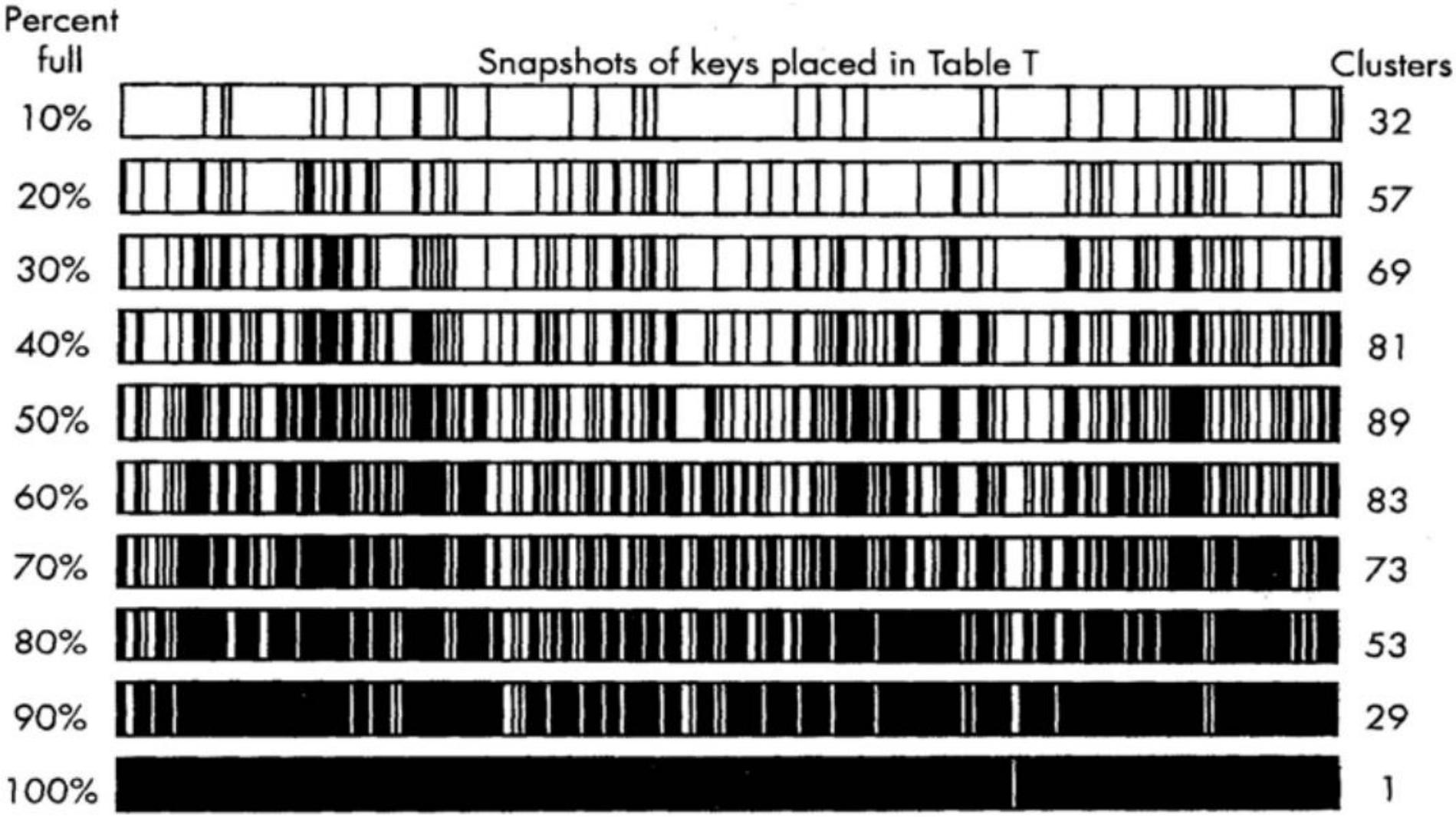
Linear probing: Primary clustering

- Primary clustering is when the table contains groups of consecutively occupied locations.
- One part of the hash table might be quite densely populated, even though another part has relatively few items.
- Long probe searches → decreases the overall efficiency of hashing





An example of primary clustering. As the occupancy rate increase, more groups consecutively occupied locations, shown as thick clusters in the hash tables.



Double hashing better scatters the elements throughout the hash tables.

Checkpoint 02a: The linear probing method

Consider a hash table of $m = 11$ slots with hash function $h(k) = k \bmod m$.

Some locations are already occupied.

Collison resolution uses linear probing.

77	89		14			96				54
0	1	2	3	4	5	6	7	8	9	10

You are given an array of integers, 45, 35 and 76.

For each element, identify which location in the hash table it should be inserted to?

Checkpoint 02b: Remove with linear probing

Write the pseudocode for the **`remove`** operation when linear probing is used to implement the hash table.

Open addressing: Quadratic probing

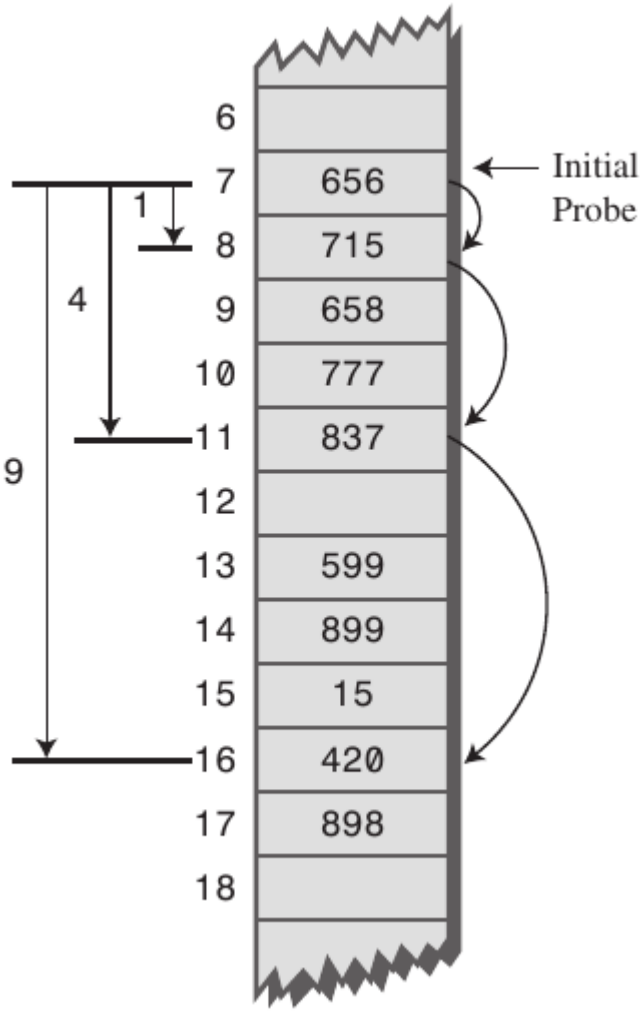
- Search the hash table in quadratic steps, starting from the original hash location, until you find an available location
  - If $table[h(k)]$ is occupied, check the locations $table[h(k) + 1^2]$, $table[h(k) + 2^2]$, $table[h(k) + 3^2]$, and so on.
- The probe sequence follows

$$\boldsymbol{(h(k) + i^2)\ mod\ tableSize\ with\ i = 0, 1, 2, \ldots}$$

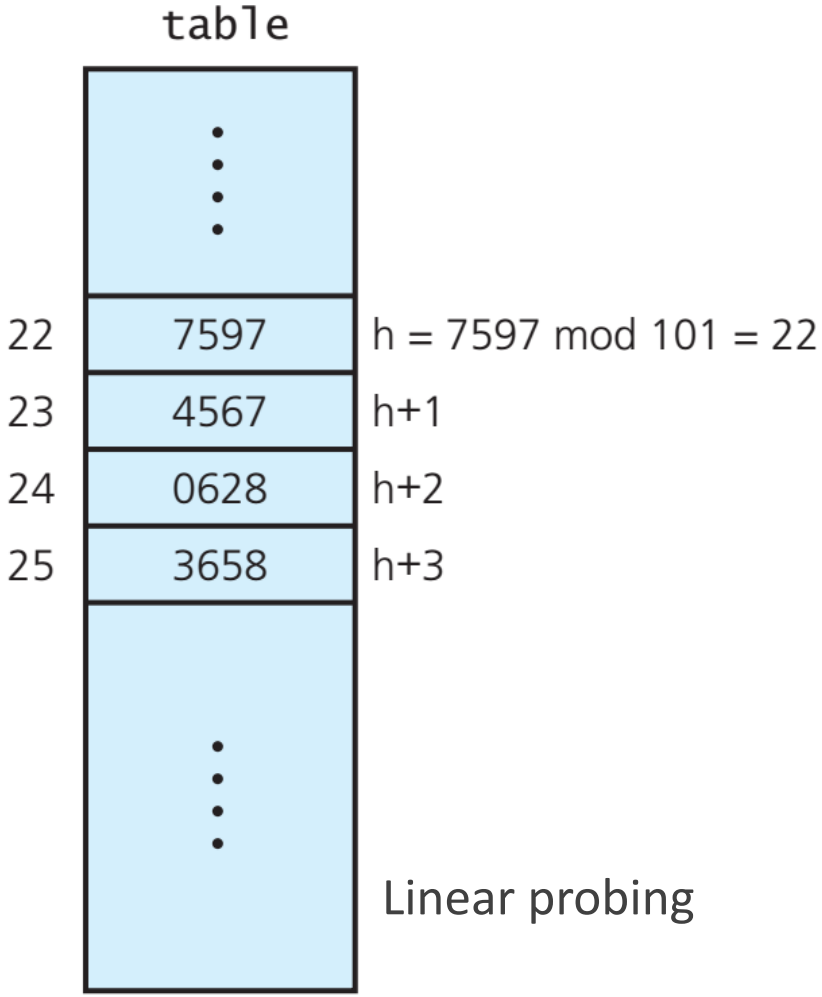
- **`getItem`**, **`remove`**, and **`add`** are like those in linear probing.

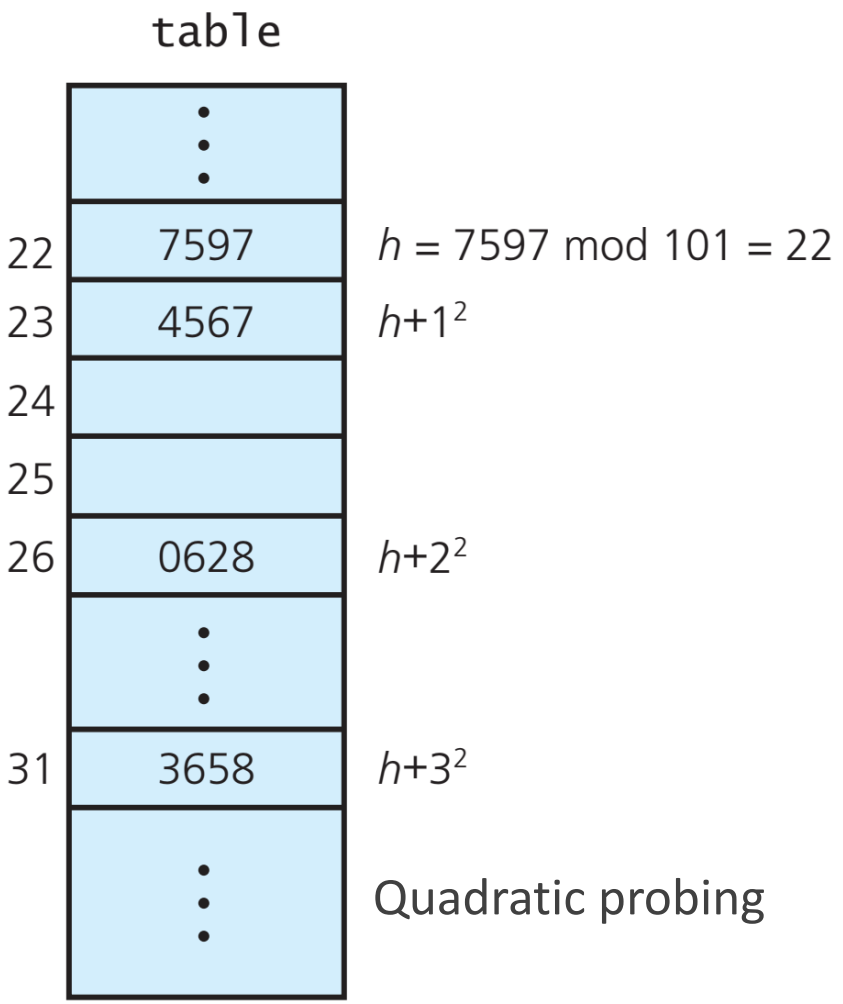
Quadratic probing: Secondary clustering

- Quadratic probing can effectively avoid primary clustering, yet it still suffers from secondary clustering.
- Secondary clustering: when two items hash into the same location, the same probe sequence is used for each item.
- It delays the collision resolution, yet it is not a severe problem.



Probing: Linear vs. Quadratic





Linear probing and quadratic probing with $h(k) = k \bmod 101$

Checkpoint 03: The quadratic probing method

Consider a hash table of $m = 11$ slots with hash function $h(k) = k \bmod m$.

Some locations are already occupied.

Collison resolution uses quadratic probing.

77	89		14			96				54
0	1	2	3	4	5	6	7	8	9	10

You are given an array of integers, 45, 35 and 76.

For each element, identify which location in the hash table it should be inserted to?

Open addressing: Double hashing

- Still search the hash table linearly with a primary hash function $h_1(k)$, but a secondary hash function $h_2(k)$ decides the size of the steps taken
- h_1 can be chosen as usual.
- $h_2(k) \neq 0$ and $h_2 \neq h_1$
- The probe sequence follows

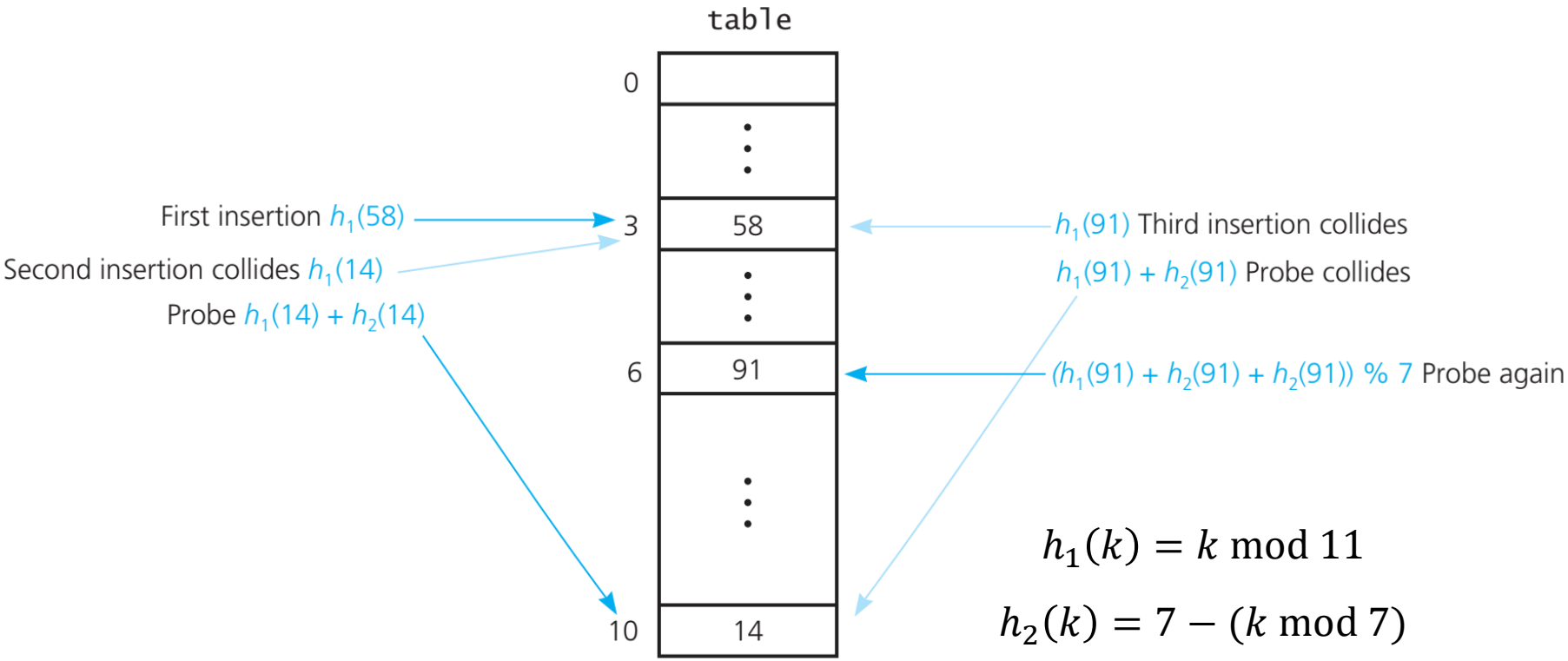
$$\boldsymbol{\left(h_1(k) + i \times h_2\,(k)\right) mod\ tableSize\ with\ i = 0, 1, 2, \ldots}$$

Open addressing: Double hashing

- Double hashing can drastically reduce clustering.
- For example, $h_1(k) = key \bmod 11$ and $h_2(k) = 7 - (k \bmod 7)$
  - If $k = 58$, probe sequence is 3, 8, 2 (wraps around), 7, 1 (wraps around), 6, 0, 5, 10, 4, 9. If $k = 14$, it is 3, 10, 6, 2, 9, 5, 1, 8, 4, 0.
  - Each probe sequence visits *all* the table locations.
- The size of the hash table and the size of the probe step should be relatively prime.

Example: An example of Double hashing

Insert the following keys: 58, 14, and 91.



$$h_1(k) = k \bmod 11$$

$$h_2(k) = 7 - (k \bmod 7)$$

Checkpoint 04: A probe sequence of double hashing

What is the probe sequence of double hashing when using the following primary and secondary hash functions with the search key $k = 19$?

$$h_1(k) = k \bmod 11$$

$$h_2(k) = 7 - (k \bmod 7)$$

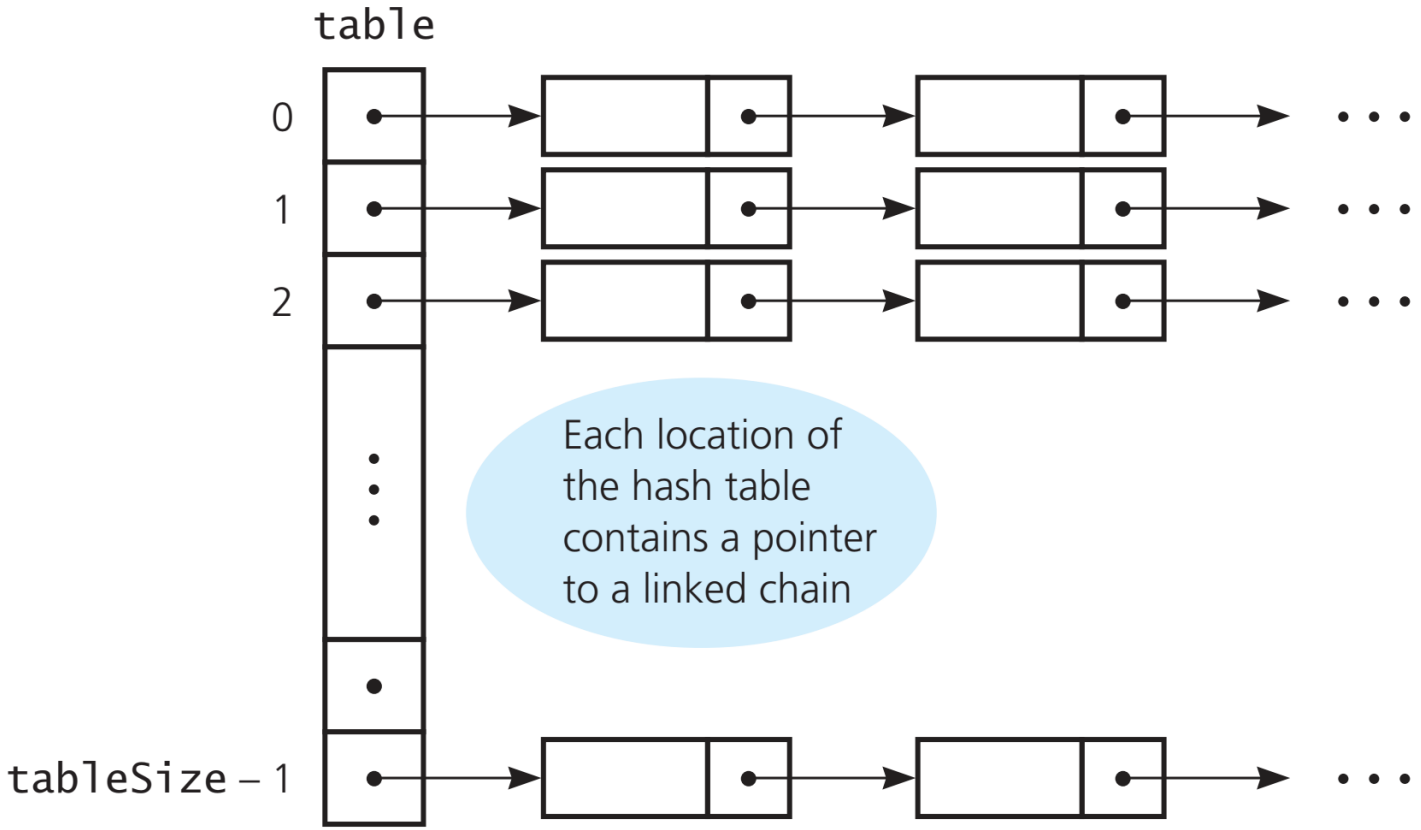
Increasing the size of the hash table

- As the hash table fills, the probability of a collision increases.
- At some point, a larger hash table becomes desirable.
- If the hash table is a dynamically allocated array, its size can be increase anytime.

- However, you cannot simply double the size of the array.
- The size of the hash table must remain prime.
- Every item in the old hash table must be **“rehashed”** by the new hash function to its new location in the new hash table.

Separate chaining

- Each entry $table[i]$ is a pointer to a linked list containing the items that the hash function has mapped into location i.



Separate chaining

- The number of items can exceed the size of the hash table.
  - That is, each linked list grows according to demand.
- The efficiency of retrievals and removals can be affected.
- Even so, separate chaining is the most time-efficient collision resolution scheme.

Buckets scheme

- A worse approach than using separate chaining
- Each location $table[i]$ is itself an array, or bucket, that stores the items that were hashed into location i.
- The size b of each bucket is the **major problem**.
  - You have postponed the problem of collisions until $b + 1$ items are mapped into some array location.
  - b must be large enough to accommodate the largest number of items that might map into each array location $\rightarrow$ waste much storage

Checkpoint 05: The separate chaining method

Consider the hash function $h(k) = k \bmod 7$ and the separate chaining collision resolution scheme.

What does the hash table look like after the following insertions occur?

8, 10, 24, 15, 32, 17

Assume that each item contains only a search key.

The efficiency of hashing

Load factor and hashing efficiency

- The **load factor α** measures how full a hash table is.
- It is the ratio of the current number of items in the table to the maximum size of the table.

$$\alpha = \frac{Current\ number\ of\ table\ items}{table\ Size}$$

- The hashing efficiency decreases as α increases.
  - As the table fills, α increases and the chance of collision increases, so search times increase.
- α should not exceed 2/3.

Load factor and hashing efficiency

- The probe sequences grow in length as collisions increase, causing increased search times.
- Hashing efficiency for a particular search also depends on whether the search is successful.
  - A failed search generally needs more time than a successful search.
- To maintain efficiency, it is important to prevent the hash table from filling up.

The efficiency of open addressing

- Linear probing: The average number of comparisons that a search requires approximates is

$$\frac{1}{2}\left[1+\frac{1}{1-\alpha}\right]$$ for a successful search, and

$$\frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right]$$ for an unsuccessful search

- For example, for a hash table that is two-thirds full ($\alpha = 2/3$)
  - An average unsuccessful search requires at most 5 comparisons, while
  - An average successful search needs at most 2 comparisons.

The efficiency of open addressing

- Quadratic probing and Double hashing: The average number of comparisons that a search requires approximates

$$\frac{-\log_e(1-\alpha)}{\alpha}$$ for a successful search, and

$$\frac{1}{1-\alpha}$$ for an unsuccessful search

- For example, for a hash table that is two-thirds full ($\alpha = 2/3$)
  - An average unsuccessful search requires at most 3 comparisons, while
  - An average successful search needs at most 2 comparisons.

The efficiency of open addressing

- Both quadratic probing and double hashing offer a smaller hash table than linear probing does.
- However, all open-addressing schemes suffer if it is unable to predict the number of insertions and removals that will occur in the hash table.

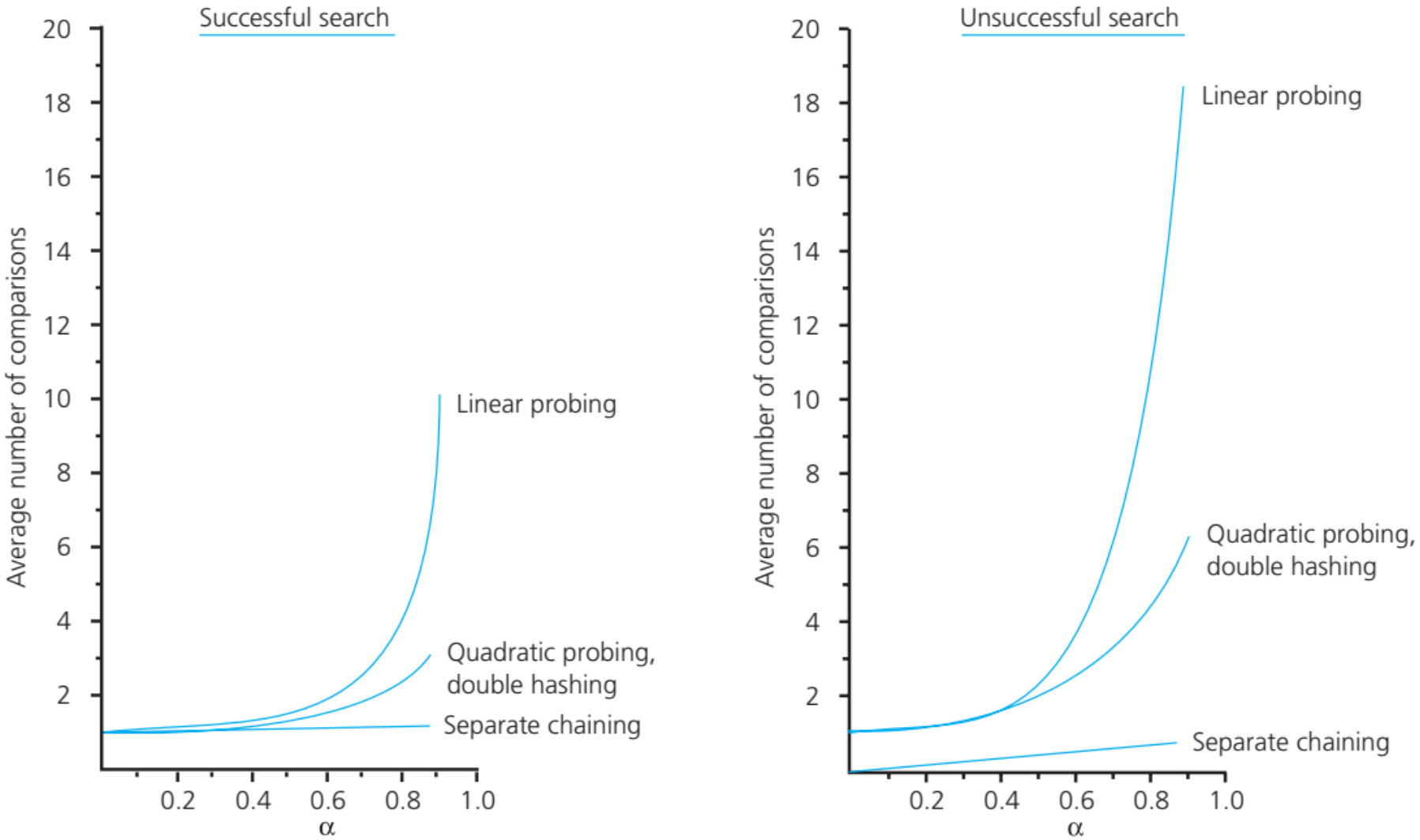
The efficiency of separate chaining

- **`add`** places the new item at the head of a linked list $\rightarrow O(1)$.
- **`getItem`** and **`remove`** may search the whole linked list.
- The load factor α can exceed 1, which denotes the average length of each linked list.
- The average number of comparisons that a search requires approximates

$1 + \frac{\alpha}{2}$ for a successful search, and

α for an unsuccessful search

Comparing hashing techniques



The relative efficiency of four collision-resolution methods in average case.

Comparing techniques

- Hashing offer better efficiency than search tree in general, yet in the worst case it can be much slower.
  - Hashing could be attractive if you can afford both an occasional slow search and a large $tableSize$.
  - A search tree could be a better choice if you want a guaranteed limit on its worst-case behavior.
- Separate chaining suffers the storage overhead of the pointers in the linked list $\rightarrow$ better for large data entries.

The performance of hashing

- To maintain efficiency of hash table's operations, you should restrict the size of α as follows
  - $\alpha < 0.5$ for open addressing
  - $\alpha < 1.0$ for separate chaining
- Should the load factor exceed these bounds, you must increase the size of the hash table using rehashing

Acknowledgements

This part of the lecture is adapted from the following materials.

[1] Pr. Nguyen Thanh Phuong (2020) "*Lecture notes of CS163 – Data structures"* University of Science - Vietnam National University HCMC.

[2] Pr. Van Chi Nam (2019) "*Lecture notes of CSC14004 – Data structures and algorithms"* University of Science - Vietnam National University HCMC.

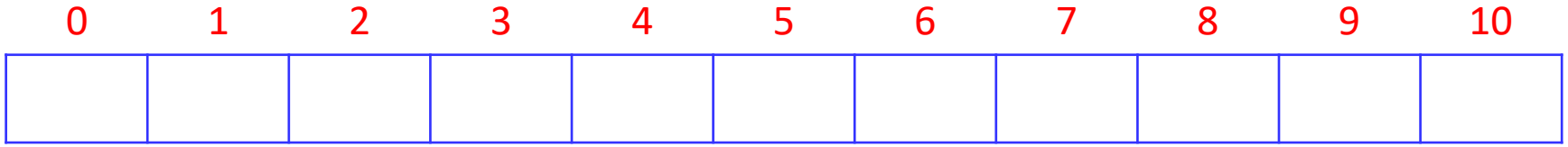
[3] Frank M. Carrano, Robert Veroff, Paul Helman (2014) "*Data Abstraction and Problem Solving with C++: Walls and Mirrors*" Sixth Edition, Addion-Wesley. **Chapter 10.**

[4] Anany Levitin (2012) "*Introduction to the Design and Analysis of Algorithms"* Third Edition, Pearson.

Exercises

01. Hashing and collisions handling

- Given a hash table of $m = 11$ slots. The hash function is $h(k) = k \bmod m$



- You are given an array of integers, {10, 22, 31, 4, 15, 28, 17, 88, 59}. For each element, identify its proper location in the hash table.
- Show the resulting hash table when collisions are resolved using either of the following methods
  a) Separate chaining
  b) Linear probing
  c) Quadratic probing
  d) Double hashing with $h_2(k) = 1 + (k \bmod (m - 1))$

02. Good hash functions

- A good hash function is one that is easy to compute and will evenly distribute the possible data.
- Comment on the appropriateness of the following hash functions. What patterns would hash to the same location?

  a) The hash table has size 2,048. The search keys are English words. The hash function is

  $h(key) = (Sum\ of\ positions\ in\ alphabet\ of\ key's\ letters) \bmod 2048$

  b) The hash table has size 2,048. The keys are strings that begin with a letter. The hash function is

  $h(key) = (position\ in\ alphabet\ of\ first\ letters\ key) \bmod 2048$

  Thus, “BUT” maps to 2. How appropriate is this hash function if the strings are random? What if the strings are English words?

02. Good hash functions

c) The hash table is 10,000 entries long. The search keys are integers in the range 0 through 9999. The hash function is

$$h(key) = (key * random) \text{ truncated to an integer}$$

where random represents a sophisticated random-number generator that returns a real value between 0 and 1.

d) The hash table is 10,000 entries long (HASH_TABLE_SIZE is 10000). The search keys are integers in the range 0 through 9999. The hash function is given by the following C++ function

```
int hashIndex(int x){
    for (int i = 1; i <= 1000000; i++)
        x = (x * x) % HASH_TABLE_SIZE;
    return x;
}
```


...the end.