SETS, MAPS AND HASHING

After studying this chapter, you should be able to

- Describe a set at the logical level
- Implement a set both explicitly and implicitly
- Describe a map at the logical level
- Implement a map
- Define the following terms:
  - Hashing
  - Linear probing
  - Rehashing
  - Clustering
  - Collisions
- Design and implement an appropriate hashing function for an application
- Describe how to resolve collisions through linear probing, rehashing, and chaining

Sets: Logical Level

- **Set:** An unordered collection of distinct values, based on the mathematical concept
- **Component (base) Type:** The data type of the items in a set
- **Subset:** A set whose items are contained entirely within another set
- **Universal Set**: The set containing all the possible values of the base type
- **Empty Set:** A set with no members

Set Operations

- **Store** and **Delete** for adding and removing items from the set
- **Union:** Combine two sets into one
- **Intersection:** Takes two sets and creates a third containing the values that are in both sets
- **Difference:** Returns the set of all elements that are in the first set but not the second
- **Cardinality:** The number of items in a set

Set Operation Examples

SetA = {A, B, D, G, Q, S}
SetB = {A, D, P, S, Z}
Union (SetA, SetB) = {A, B, D, G, P, Q, S, Z}
Intersection (SetA, SetB) = {A, D, S}
Difference (SetA, SetB) = {B, G, Q}
Difference (SetB, SetA) = {P, Z}

Additional Operations

- Observers: IsEmpty and IsFull
- Transformer: MakeEmpty
- Utility: Print
- Note that Store and Delete are equivalent to Union and Intersection with the set containing only the target element

Sets: Application Level

- One unique property of sets is that storing an item that is already in the set does not change the set
- Similarly, deleting an item that’s not in the set does not affect anything
- This can useful for building a list of the words in a text, for example

Sets: Implementation Level

- There are two general ways to implement sets
- **Explicit representation:** The presence and absence of every possible value in the base type is recorded
- **Implicit representation:** Only the items in the set are recorded

Explicit Representation

- The explicit representation maps each item in the base type to a Boolean flag
- If the item is in the set, the flag is set to true
- **Bit Vector:** An array of bits, typically used as a compact list of Boolean flags
- It must be possible to enumerate every possible value of the base type
- Works best for base types with small domains

Explicit Representation (cont.)

- Store finds the bit for the given item and sets it to 1; Delete sets the appropriate bit to 0
- Union, Intersection, and Difference loop through the bit vector and use bitwise operations:
  - Union: OR the two bit vectors
  - Intersection: AND the two bit vectors
  - Difference: AND with the NOT (inversion) of the second bit vector

Implicit Representation

- Store only the items that are in the set
- The set itself is stored as a list
- Store uses RetrieveItem to see if the item is already in the set, and inserts it if not
- Delete uses RetrieveItem and only tries to delete the item if it’s actually in the set

Implicit Set Intersection

- Naive algorithm: For every item in Set A, check if it's in Set B. If yes, store it in the result set.
- This is O(N*N) or O(N $\log_2 N$), depending on the implementation of RetrieveItem.
- Can we do better? Yes, by changing the assumptions about ItemType to get an algorithm with O(N) performance.

Implicit Set Intersection (cont.)

- If ItemType supports “<” a sorted list can be used to store the sets
  - Sets are still unsorted collections, but the implementation can be sorted
- Compare the first items in Sets A and B; if they’re equal, store one of them in the result
  - If A < B, get the next item in A
  - If B < A, get the next item in B

Intersection Pseudocode

```
SetType Intersection(SetType setB):
Reset self
Reset setB
while NOT (self.IsEmpty OR setB.IsEmpty)
    GetNextItem(itemA) from self
    setB.GetNextItem(itemB)
    if (itemA < itemB)
        GetNextIem(itemA) from self
    else if itemB < itemA
        setB.GetNextItem(itemB)
    else
        result.Store(itemA)
        GetNextItem(itemA) from self
        setB.GetNextItem(itemB)
return result
```

Explicit or Implicit?

- Explicit representation must be able to represent every item in the base type
  - That is, the length of the bit vector is the cardinality of the universal set
- Implicit representation requires searching the list to find if an item is already in the set
- Explicit is better for base types with a very small number of unique values

Maps: Logical Level

- An ADT that is a collection of **key-value pairs**
  - **Key:** A value of the base type of the map, used to look up an associated value
- Like sets, maps store unsorted, unique values
- They do not support Union, Intersection, and Difference and have no mathematical basis
- Supports Find, an operation for retrieving a key-value pair by searching for the key

Maps: Application Level

- Arrays can be seen as maps that use integers as keys
- Maps, then, are like arrays whose keys cannot easily be converted into array indices
- Example: mapping names to phone numbers

Maps: Implementation Level

- The most common map operation is Find
- Binary Search Trees support very efficient searching, along with insertion and deletion
- Insertion: If the key is already in the map, the map is not changed
- Find is basically identical to GetItem, except it only checks for the correct key
- ItemType contains the key and the value

Maps: Other Approaches

- Find could return void and use an empty value or a bool flag to indicate failure
- It is very useful for clients that search does not “fail” (throw an exception) and instead returns a dummy value or a flag that can be checked
- The C++ STL overloads [ ] so that maps can be treated like arrays

O(1) Search

- Is it possible to have O(1) search?
- There would have to be a 1-to-1 mapping between element keys and array indices
- For example: Employees with IDs ranging from 00 to 99, stored in an array by ID
- But if IDs were from 00000 to 99999 and there were only a few hundred employees, tons of memory would be wasted

Hashing

- **Hashing:** A technique for ordering and accessing elements in a list by manipulating the key of the element
- **Hash Function:** A function that manipulates the key to produce an array index
- For example, the 5-digit employee IDs could be hashed using Key % 100 to produce a 2-digit array index

Hashing (cont.)

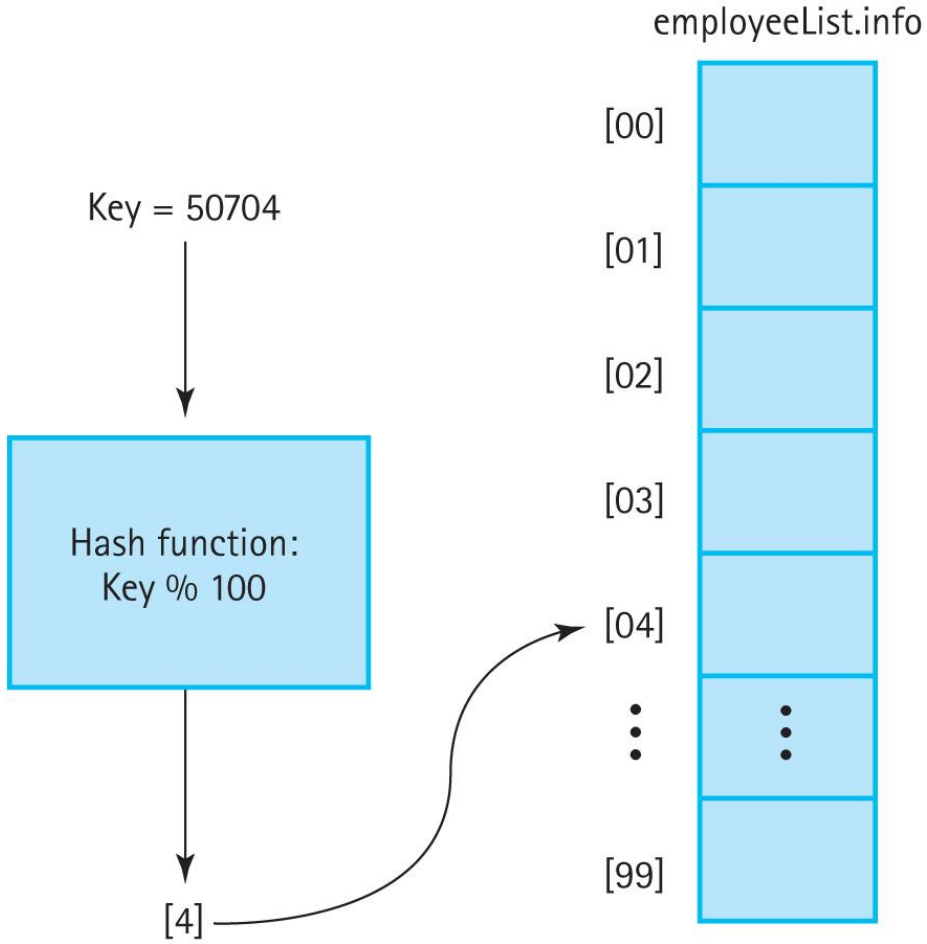

Figure 11.1 Using a hash function to determine the location of the element in an array

Hashing vs. Sequential Lists

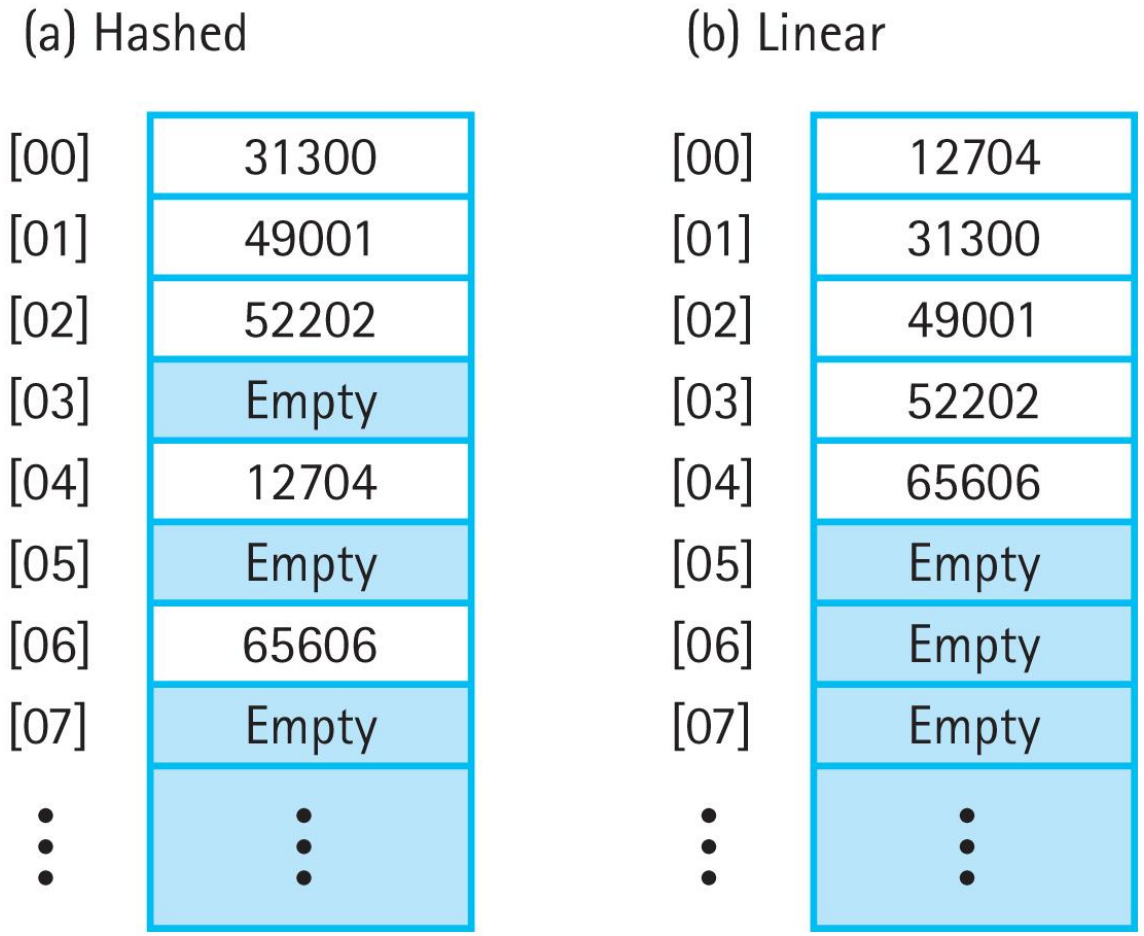

Figure 11.2 Comparing hashed and sequential lists of identical elements (a) Hashed (b) Linear

Collisions

- What happens if we hash 01234 and 97534?
  - Both produce the hash value “34”
- **Collision:** When multiple keys produce the same hash location
- A good hash function minimizes collisions, but they’re impossible to completely avoid
- How can collisions be resolved?

Linear Probing

- Resolves hash collisions by searching for the next available space
- If the end of the hash is reached, linear probing loops around to the beginning
- To check if an item is in the hash table, calculate the hash and search sequentially until the matching key is found; stop when an empty space or the original hash is found

Linear Probing (cont.)

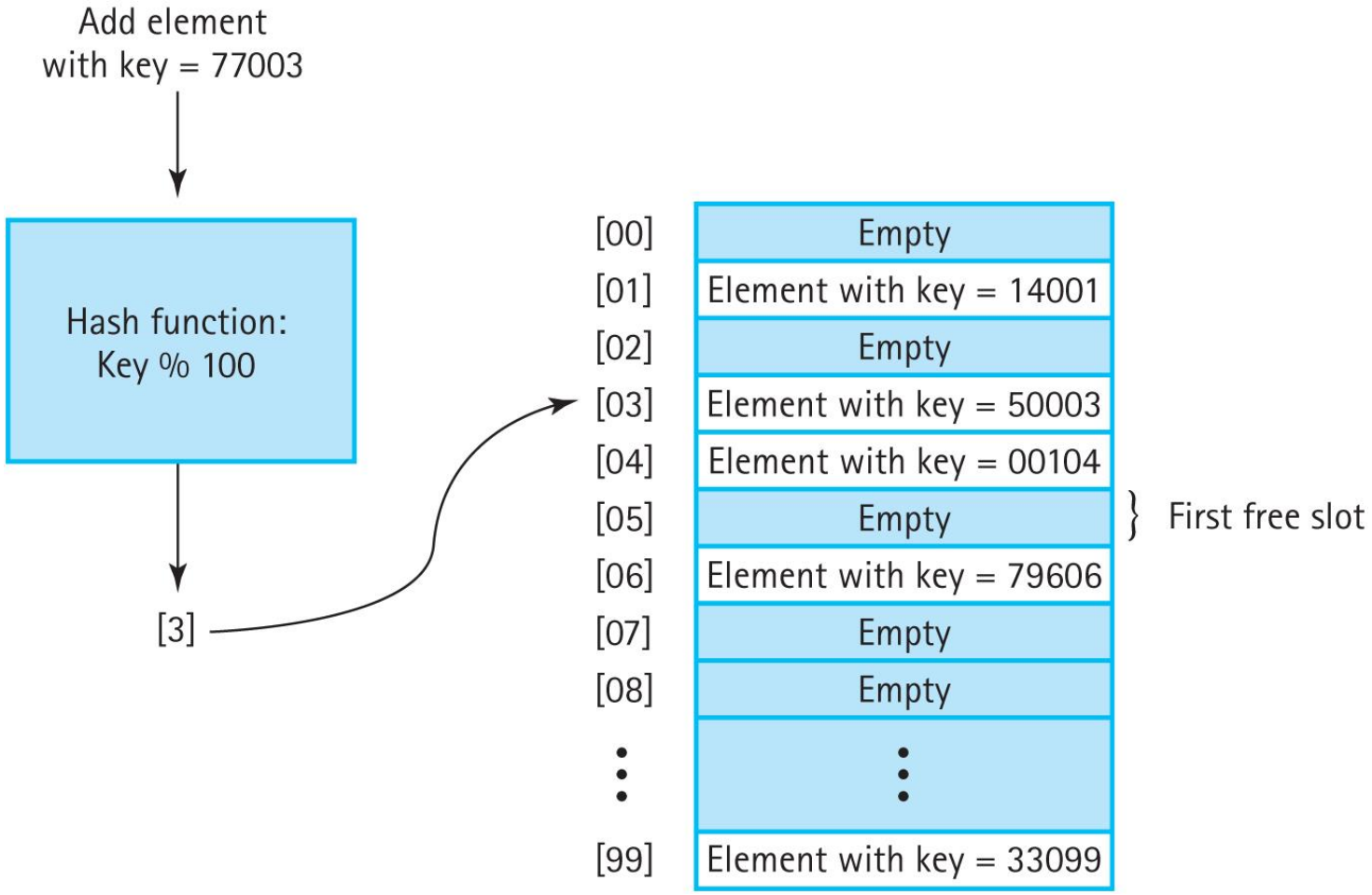

Figure 11.3 Handling collisions with linear probing

Linear Probing and Deletion

- Deleting an item creates an empty spot
- What happens if three items with the same hash are inserted, the second one is deleted, and then the client searches for the third?
- Linear probing will stop when it encounters the deleted second item’s slot, reporting that the third item does not exist
- A dummy “deleted item” value tells search to keep looking

Clustering

- The tendency of elements to become unevenly distributed throughout a hash table
- A good hash function will have a uniform distribution of hash keys
- But linear probing will cause clustering anyway
- This makes insertion and retrieval less efficient

Rehashing

- Using the hash value as the input to a hash function in order to find a new location
- A good rehashing function for linear probing:
  (hash value + c) % s
  - Where c and s are two integers whose greatest common divisor is 1, e.g., c = 3 and s = 100
- The two constants must be *relatively prime* so that rehashing covers the whole array

Quadratic and Random Probing

- **Quadratic probing** rehashes based on the number of times the rehashing function has been called (*I*)
  - (HashValue ± I^2) % array size
  - Reduces clustering, but doesn't use every index
- **Random Probing:** Generate random rehash values; eliminates clustering, but slow

Buckets and Chaining

- Handle collisions by allowing multiple elements to have the same hash value
- **Bucket:** A collection of values associated with the same hash key; has limited space
- **Chain:** A linked list of elements that share the same hash key; the hash table has pointers to list of values

Buckets

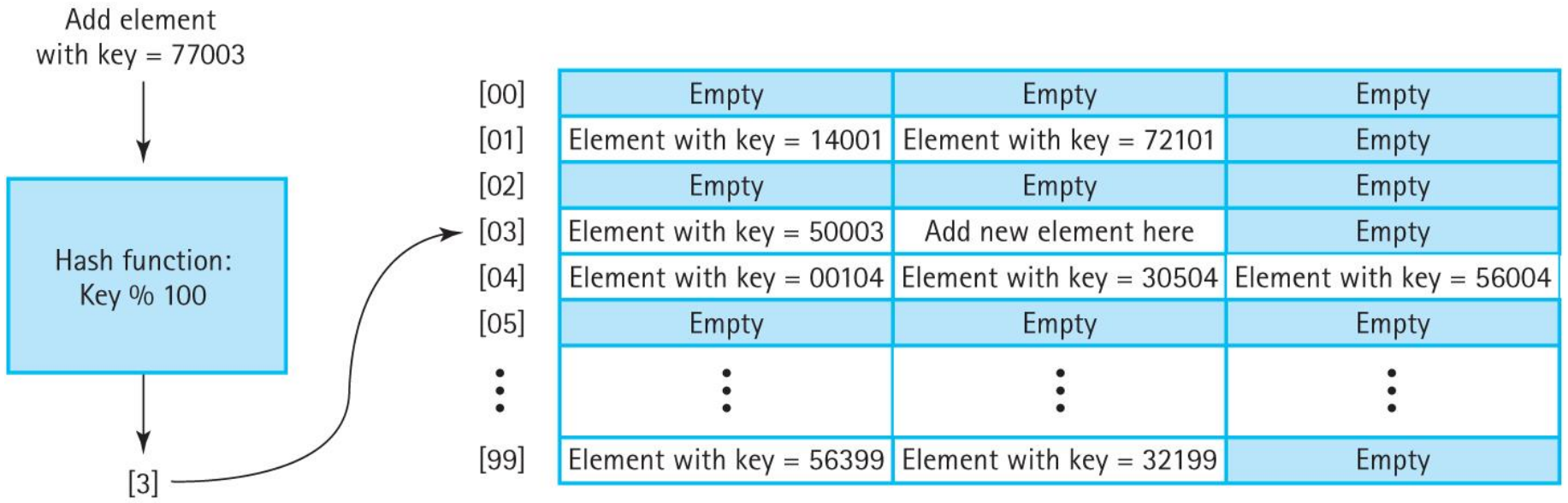

Figure 11.6 Handling collisions by hashing with buckets

Chaining

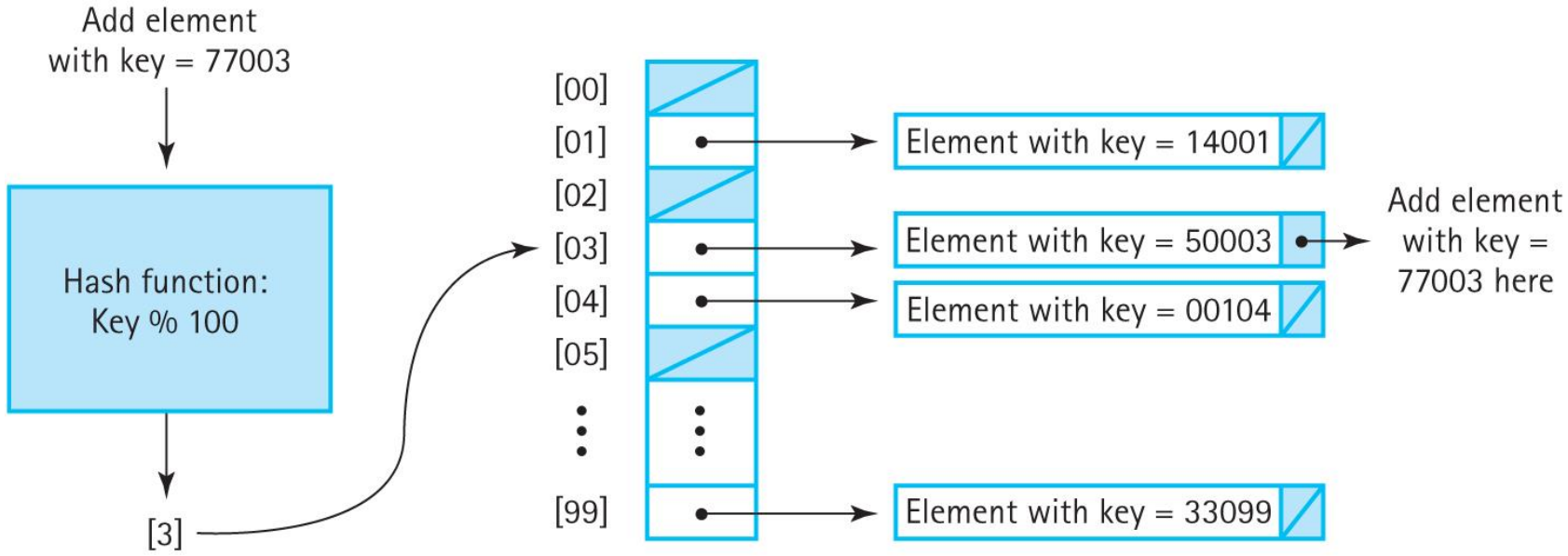

Figure 11.7 Handling collisions by hashing with chaining

Probing vs. Chaining

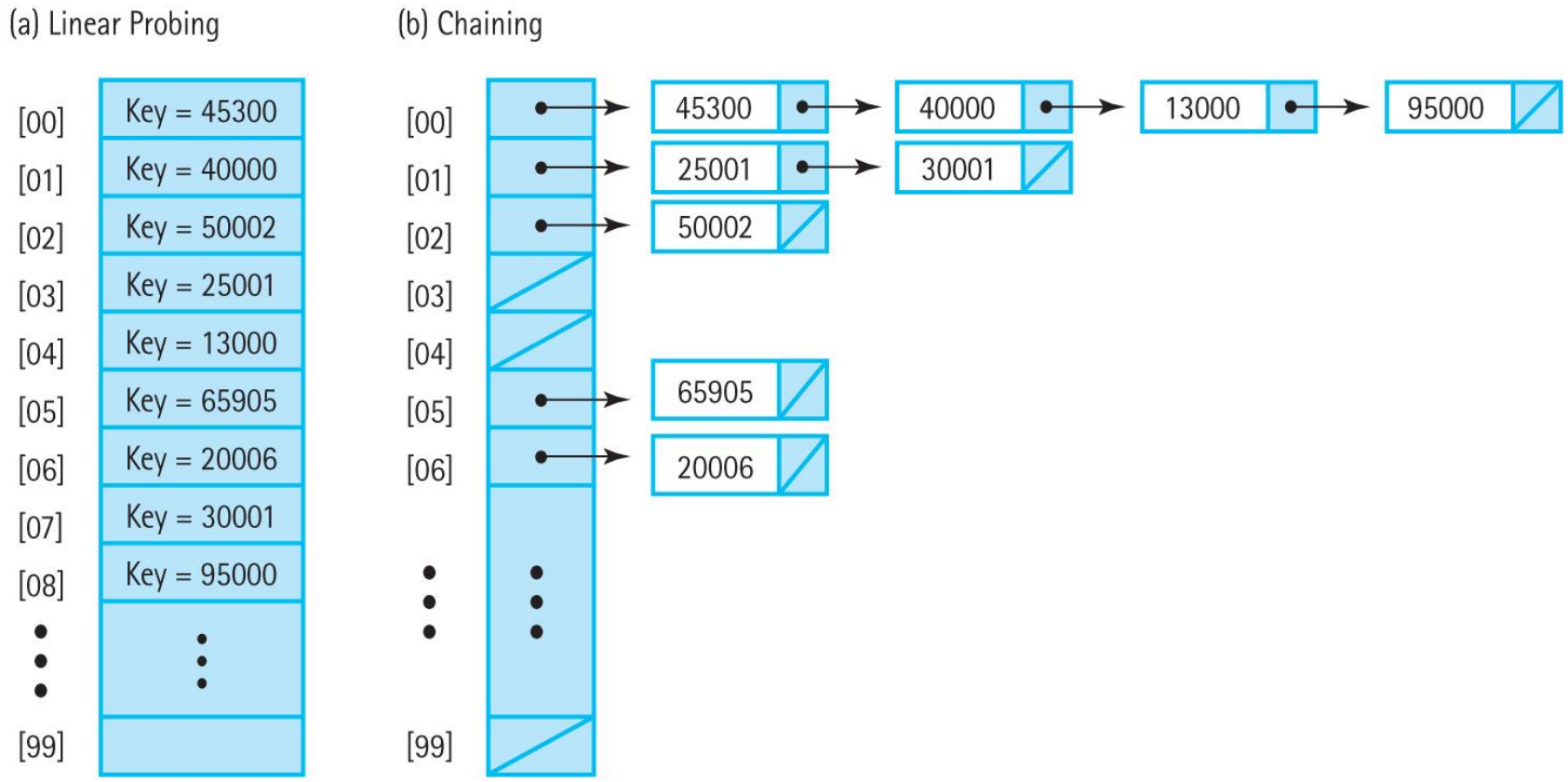

Figure 11.8 Comparison of linear probing and chaining schemes (a) Linear Probing (b) Chaining

Probing vs. Chaining (cont.)

- Search: Probing must continue searching the array until all possible hash keys have been checked. Chaining only looks through the chain, which is likely to be relatively small.
- Deletion: Chaining simply removes the element from the linked list. Probing requires special signal values.

Choosing a Good Hash Function

- A good hash function reduces collisions by uniformly distributing elements
- Knowledge about the domain of keys helps
  - For example, Key % 100 is terrible for employee IDs if the last two digits are the year the employee was hired
- Probing, buckets, and chaining can only do so much when there’s many collisions

Hashing Employee IDs

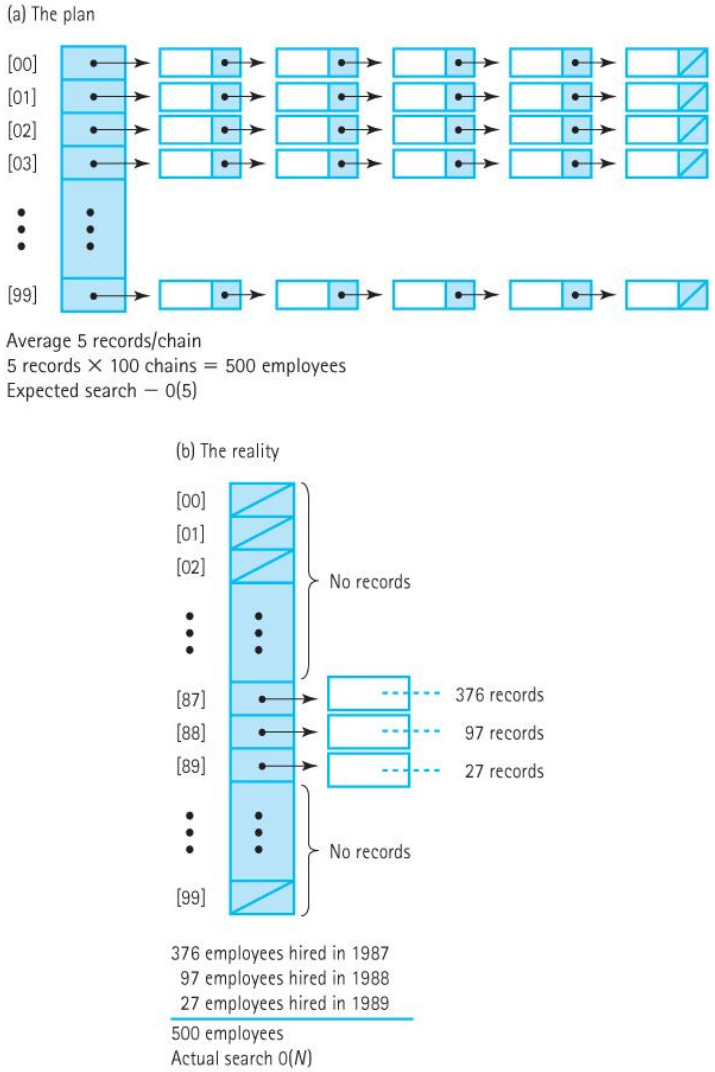

Figure 11.9 Hash scheme to handle employee records (a) The plan (b) The reality

Division Method

- The most common hash method: key % size
- By making the table larger than the expected number of elements, collisions can be reduced
- Very efficient, but collisions can be common

Other Hash Methods

- The division method requires integers
- A string of characters could be hashed by summing up the individual letters first
- **Folding:** Break the key into multiple pieces and concatenate or XOR the pieces to make a complete hash key

Creating a Good Hash Function

- **Efficiency:** Hash tables have O(1) search. This deteriorates if the hash function is inefficient
- **Simplicity:** Don’t forget hash functions need to be written and maintained! Complex functions may incur a technical cost.
- **Perfect hash functions** are difficult but not impossible, especially for small sets of values.

The End!