Hashing

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 be 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

Implementation Using STL++

```
#include <iterator>
#include <set>

int main()
{
    // empty set container
    set<int, greater<int> > s1;

    s1.insert(40);
    s1.insert(50);
    s1.insert(50);

// print all elements of the set
 set<int, greater<int> >::iterator itr;
    for (itr = s1.begin();
         itr != s1.end(); ++itr)
    {
        cout << ',' << *itr;
    }

s1.erase(50);
```

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

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 keys of the elements
- **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

Hash Table

Hash function h: Mapping from Universe U to the slots of a hash table T[0..m–1].

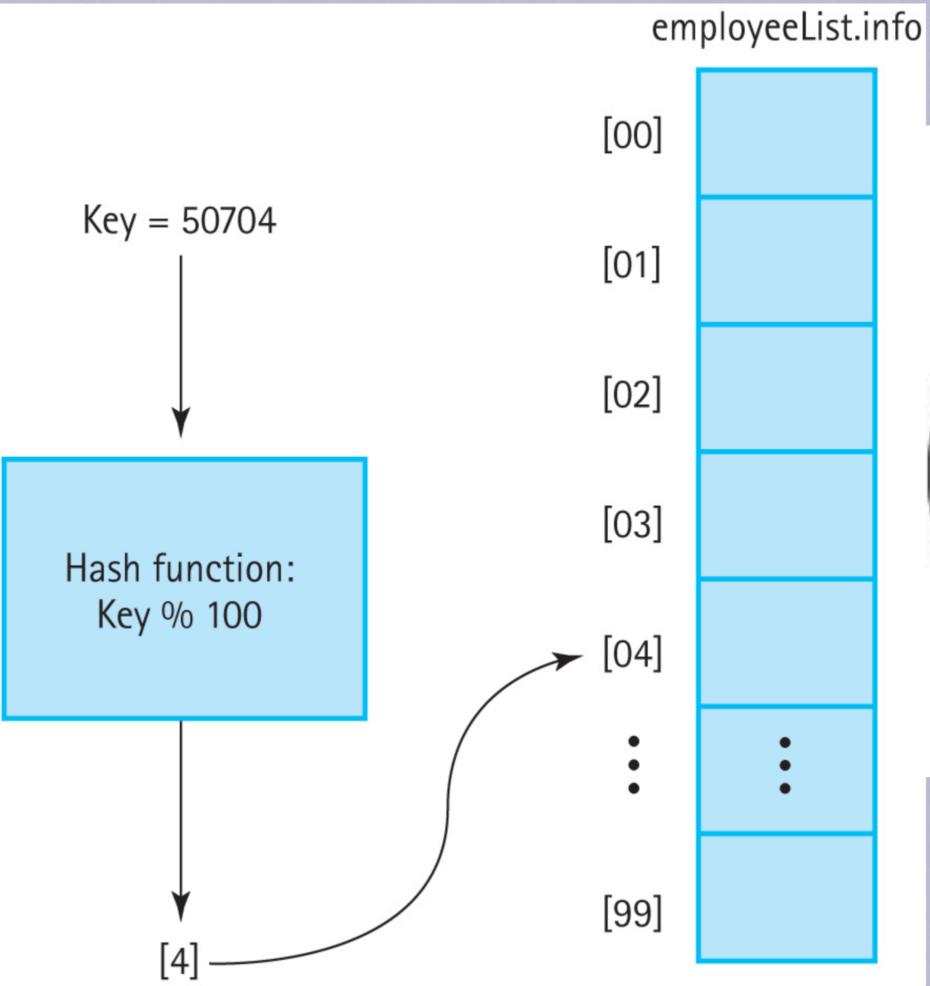
$h : U \rightarrow \{0,1,\ldots, m-1\}$

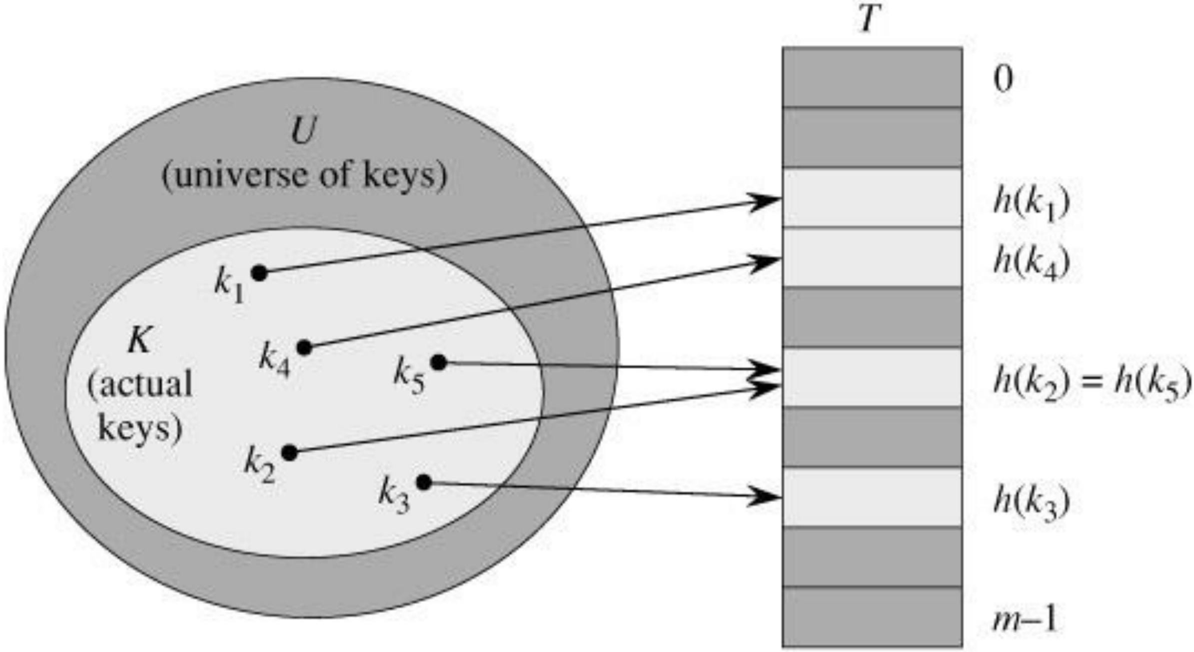
- With arrays, key k maps to slot A[k].
- With hash tables, key k maps or "hashes" to slot T[h(k)]
    - h(k) is the hash value of key k

Example of Hash Function

- h(k) = return (k mod m)
- where k is the key, and m is the size of the table

Hashing (cont.)





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

Hashing vs. Sequential Lists

Index	(a) Hashed	Index	(b) Linear
[00]	31300	[00]	12704
[01]	49001	[01]	31300
[02]	52202	[02]	49001
[03]	Empty	[03]	52202
[04]	12704	[04]	65606
[05]	Empty	[05]	Empty
[06]	65606	[06]	Empty
[07]	Empty	[07]	Empty
⋮	⋮	⋮	⋮

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

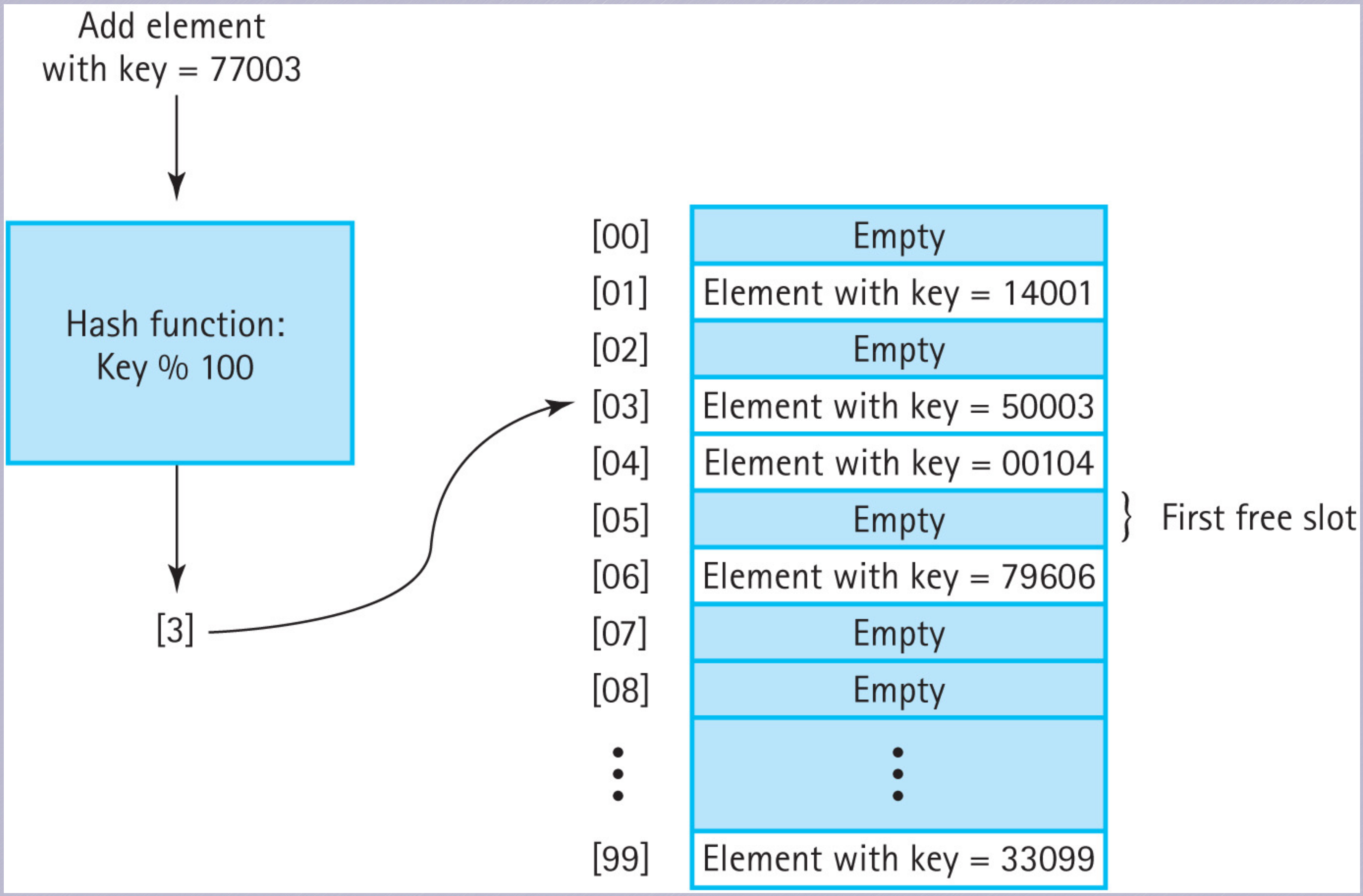
Collisions

- 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.)



Handling collisions with linear probing

Linear Probing: Example

Function f is linear, e.g., $f(i) = i$

$h(k, i) = (h'(k) + i) \bmod m$

- Offsets: 0, 1, 2, …, m-1
- Only probe m slots

With $H = h'(k)$, we try the following cells with wraparound:

- H, H + 1, H + 2, H + 3, …

What does the table look like after the following insertions? (assume $h'(k) = k \bmod m$)

- Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Linear Probing: Example (cont.)

Function f is linear, e.g., f(i) = i

h(k, i) = (h'(k) + i) mod m

- Offsets: 0, 1, 2, …, m-1
- Only probe m slots

With H = h'(k), we try the following cells with wraparound:

- H, H + 1, H + 2, H + 3, …

What does the table look like after the following insertions? (assume h'(k) = k mod m)

- Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

0	0
1	1
2	49
3	81
4	4
5	25
6	16
7	36
8	64
9	9

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 Issue

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 $\pm I^2$) % array size
  - Reduces clustering, but doesn't use every index
- **Random Probing:** Generate random rehash values; eliminates clustering, but slow

Example-- Quadratic 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

Quadratic Probing: Example

Function f is quadratic: $f(i) = i^2$

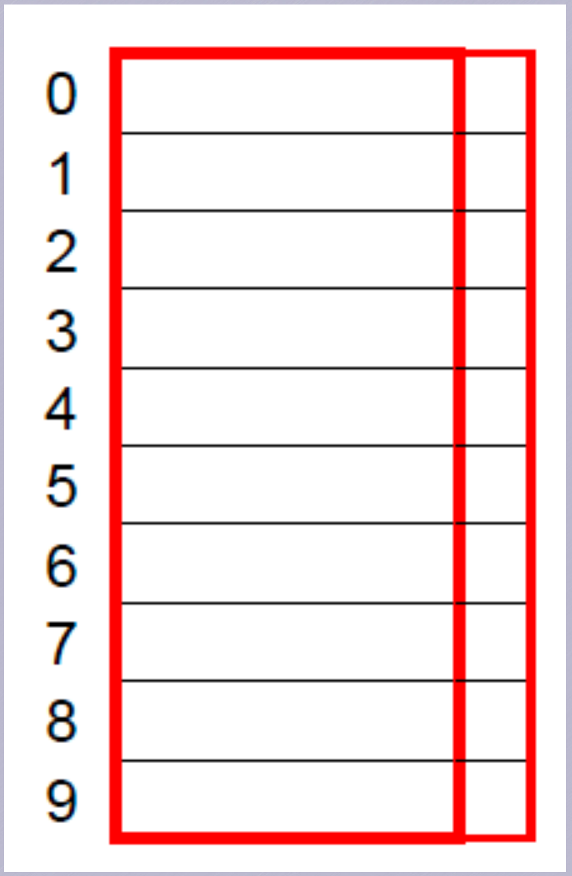
$h(k, i) = (h'(k) + i^2) \bmod m$

- Offsets: 0, 1, 4, 9, …

With $H = h'(k)$, we try the following cells with wraparound:
$H, H + 1^2, H + 2^2, H + 3^2 \ldots$

- A sequence of m slots

Insert Keys: 10, 23, 14, 9, 16, 25, 36, 44, 33



Quadratic Probing: Example (cont.)

Function f is quadratic: $f(i) = i^2$

$h(k, i) = (h'(k) + i^2) \bmod m$

- Offsets: 0, 1, 4, 9, …

With $H = h'(k)$, we try the following cells with wraparound:

$H, H + 1^2, H + 2^2, H + 3^2 \ldots$

- A sequence of m slots

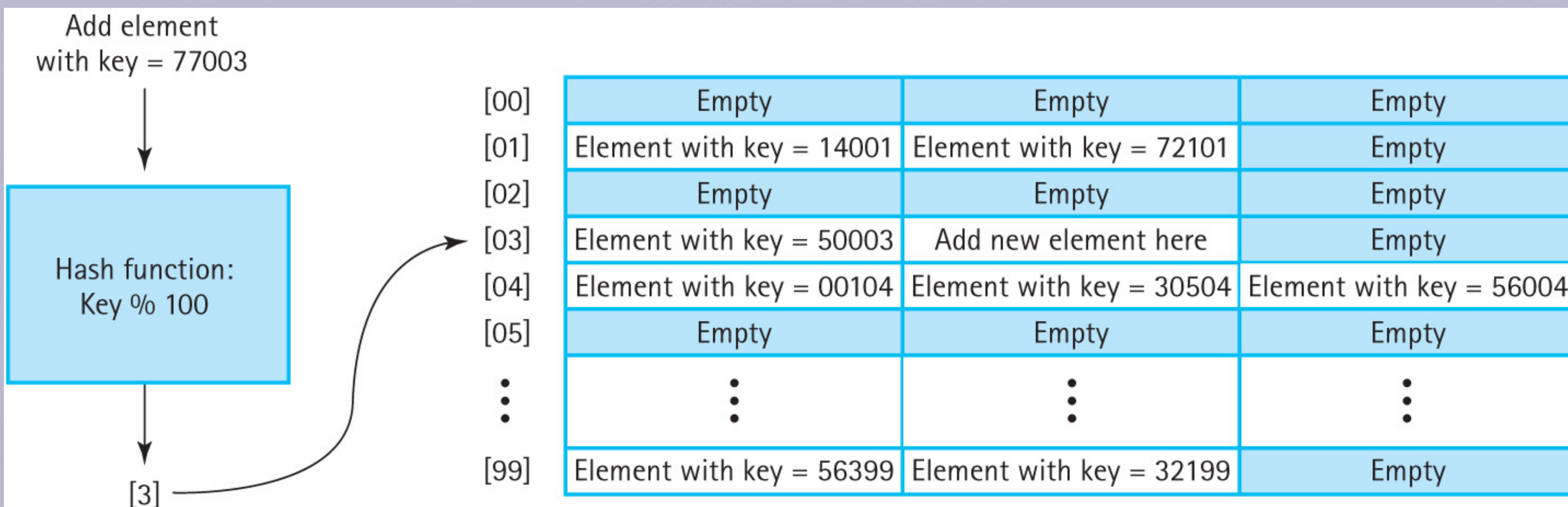
Insert Keys: 10, 23, 14, 9, 16, 25, 36, 44, 33

0	10
1	
2	33
3	23
4	14
5	25
6	16
7	36
8	44
9	9

Buckets and Chaining

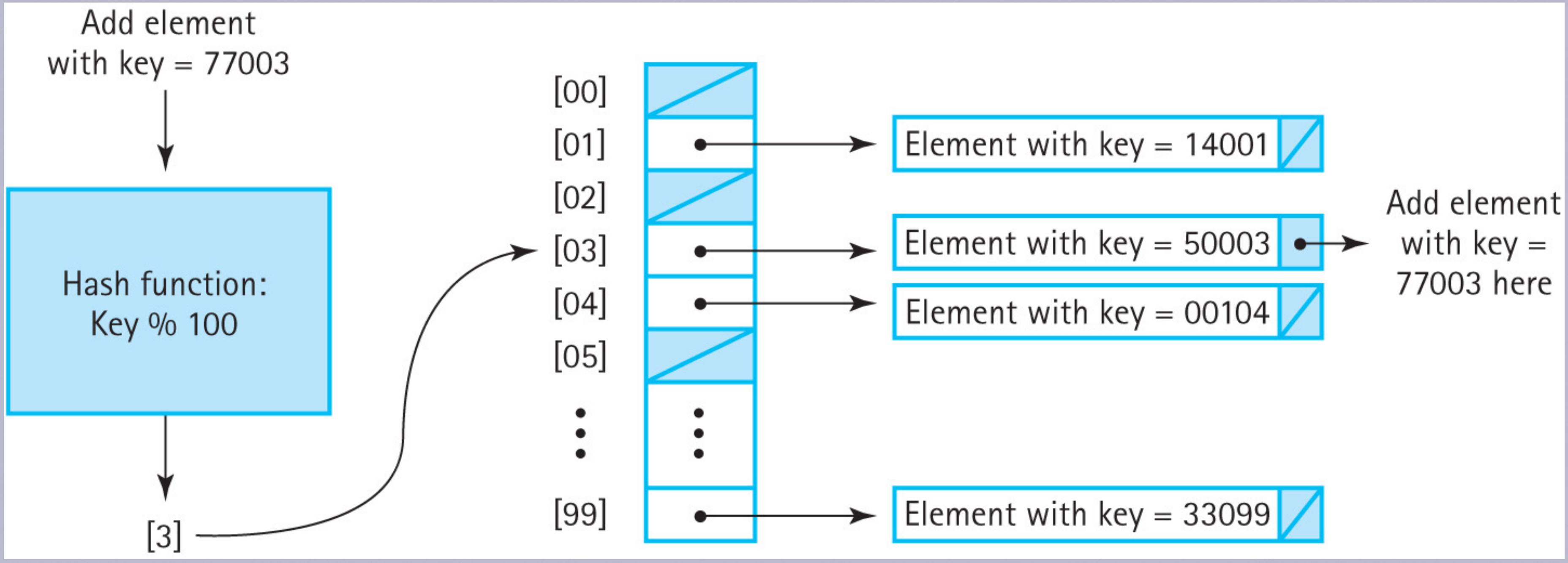
- Handle collisions by allowing multiple elements to have the same hash value
- **Bucket:** A collection of values associated with the same hash key; has limited space
- **Chain:** A linked list of elements that share the same hash key; the hash table has pointers to list of values

Buckets



Handling collisions by hashing with buckets

Chaining



Handling collisions by hashing with chaining

Chaining: Example

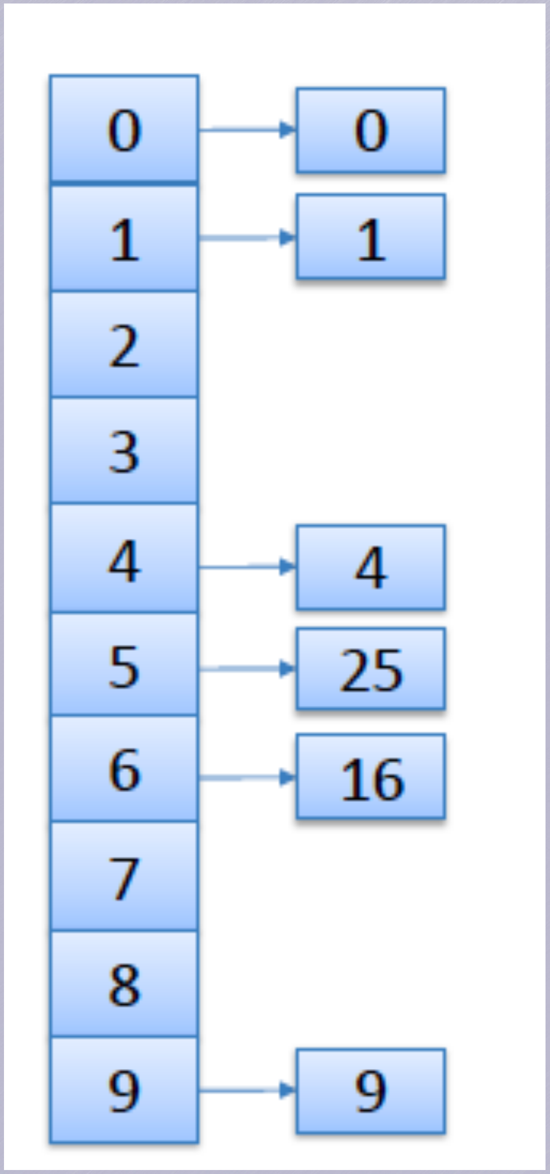
The hash table is an array of linked lists

Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

Notes:

- Elements would be associated with the keys

- We're using the hash function h(k) = k mod m
- m=10



Chaining: Example (Cont.)

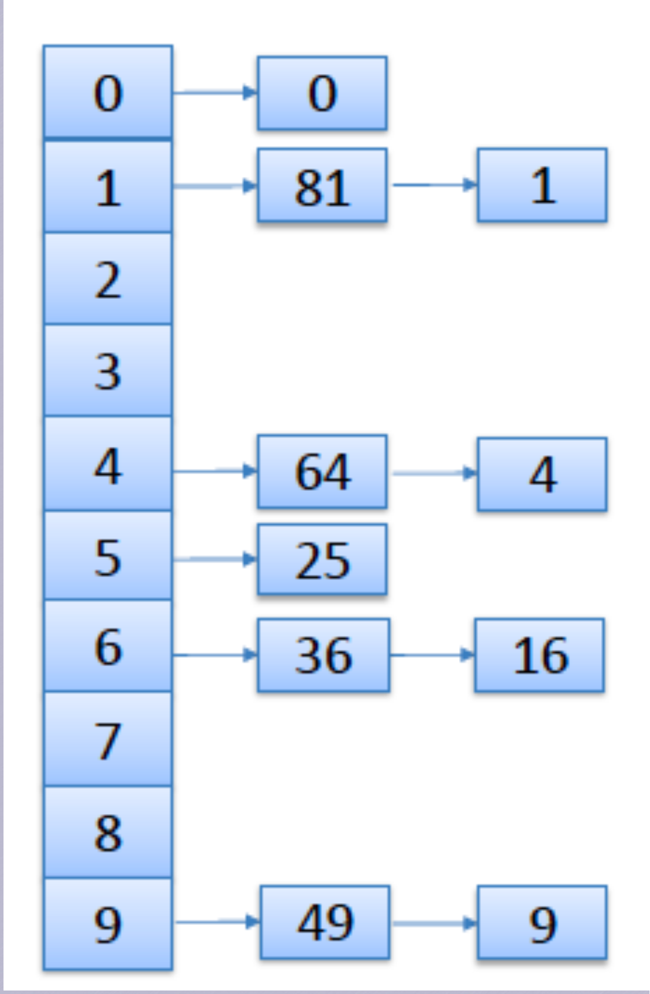
The hash table is an array of linked lists

Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

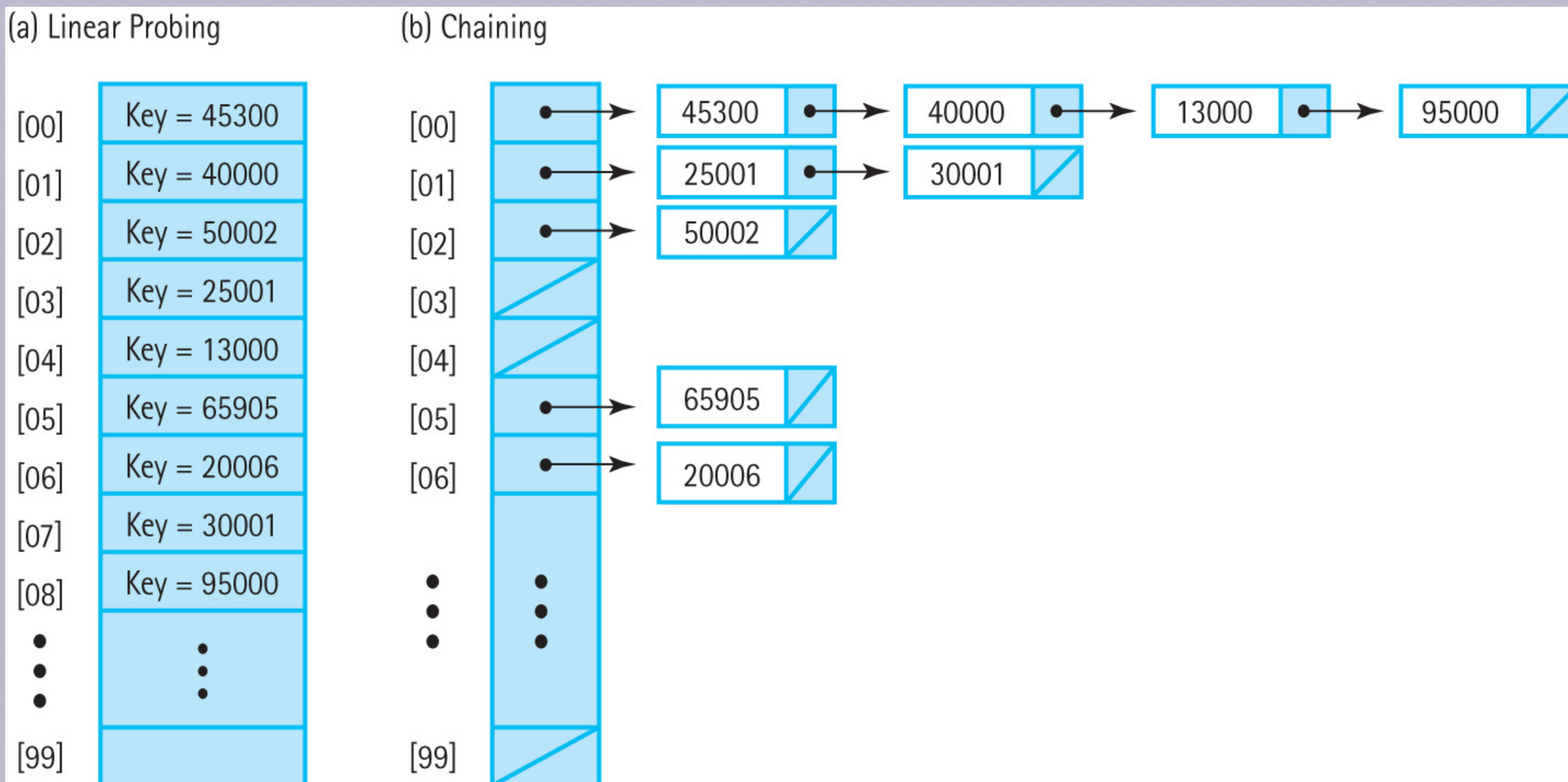
Notes:

- Elements would be associated with the keys

- We're using the hash function h(k) = k mod m
- m=10



Probing vs. Chaining



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

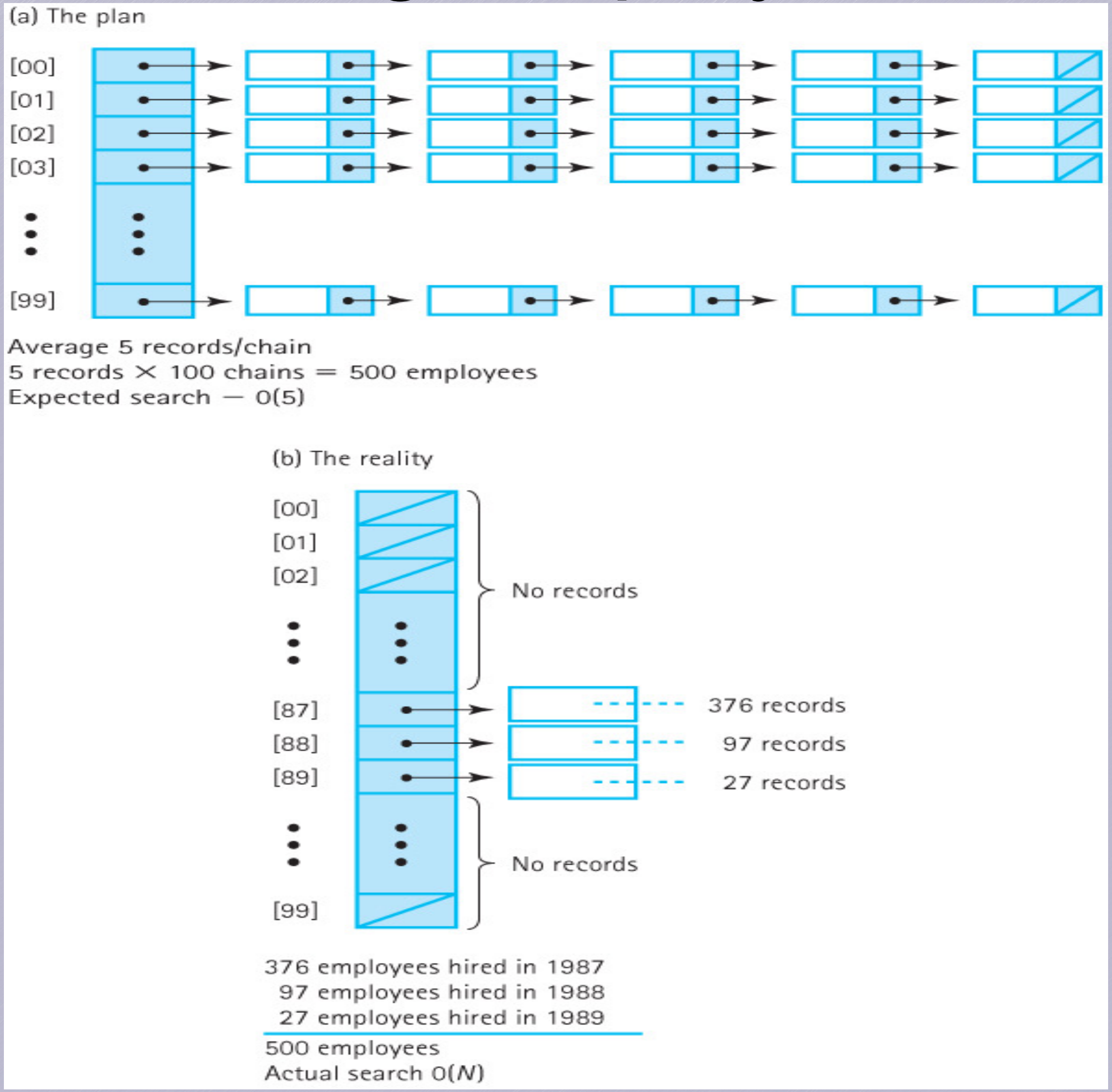
Probing vs. Chaining (cont.)

- 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



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.