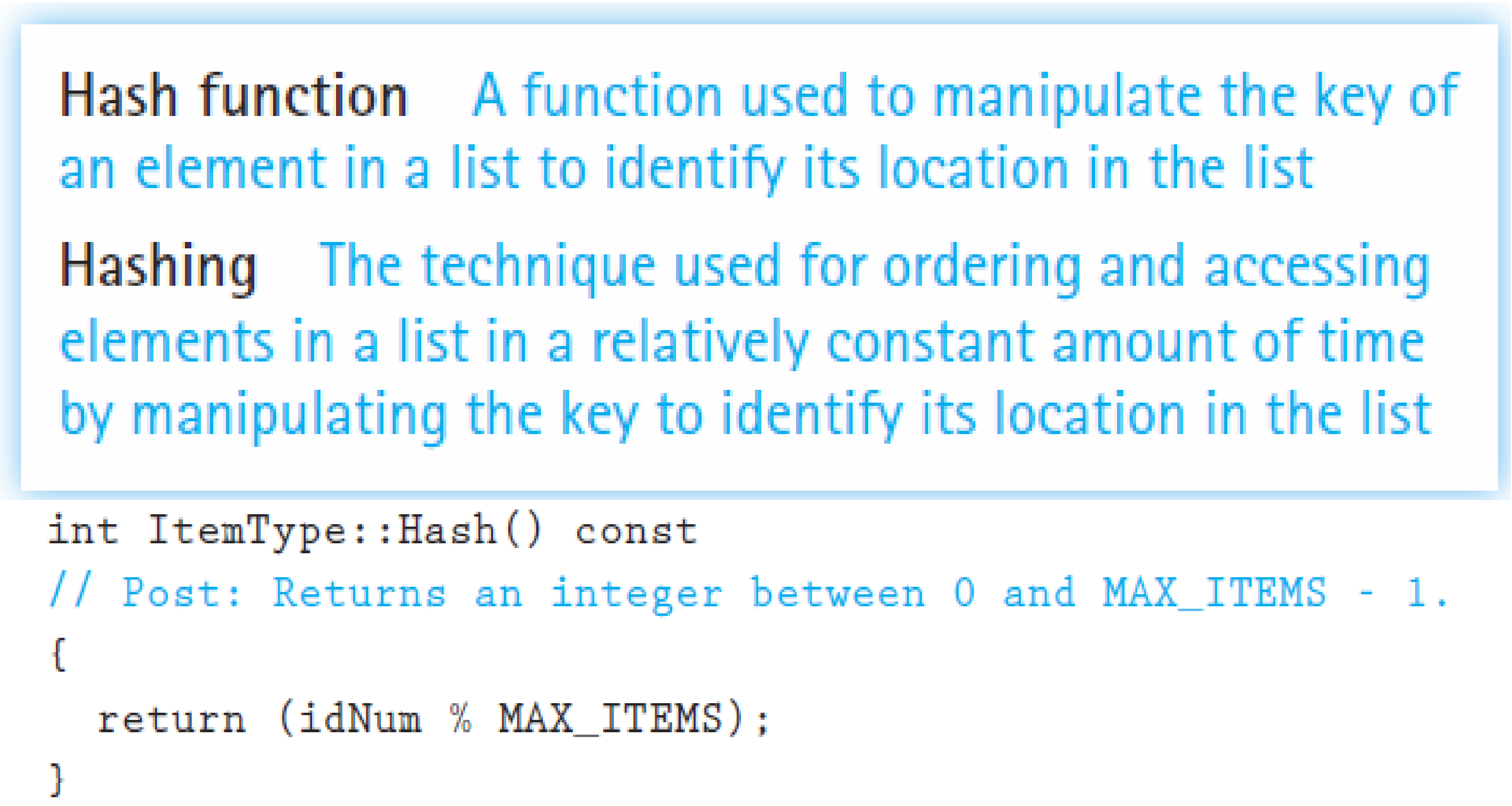


Hash function A function used to manipulate the key of an element in a list to identify its location in the list

Hashing The technique used for ordering and accessing elements in a list in a relatively constant amount of time by manipulating the key to identify its location in the list

```
int ItemType::Hash() const
// Post: Returns an integer between 0 and MAX_ITEMS - 1.
{
  return (idNum % MAX_ITEMS);
}
```

Hash Table

Hash table is a data structure that supports searching, insertion and deletions (implementation of a hash table is called hashing)

- The idea of hash table is an array of fixed size , containing no of elements
- Search is performed based on keys
- Each key is mapped to same position in the range (0 to tablesize-1)
- The mapping is called **hash function**

```
int ItemType::Hash() const
// Post: Returns an integer between 0 and MAX_ITEMS - 1.
{
  return (idNum % MAX_ITEMS);
}
```

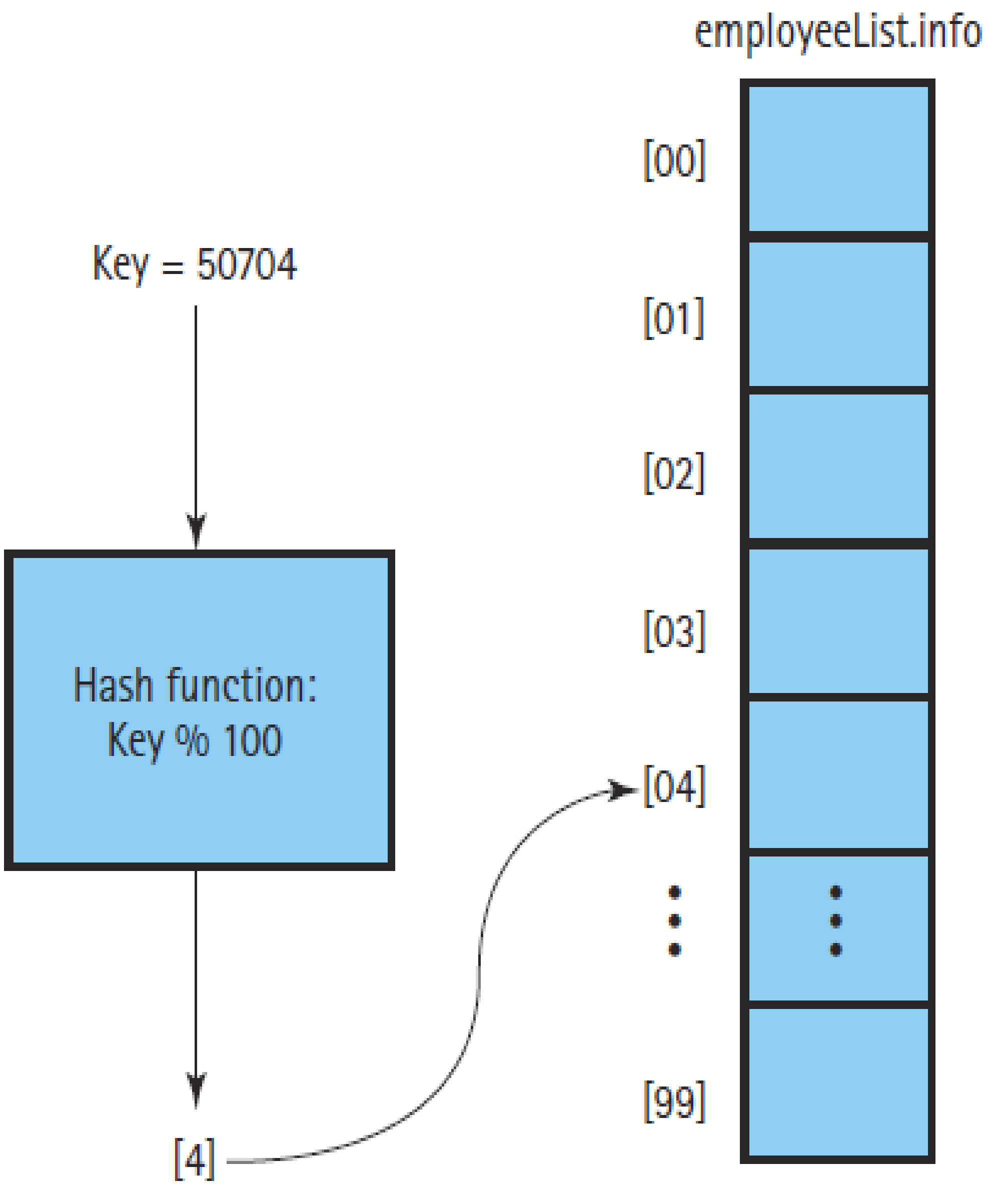


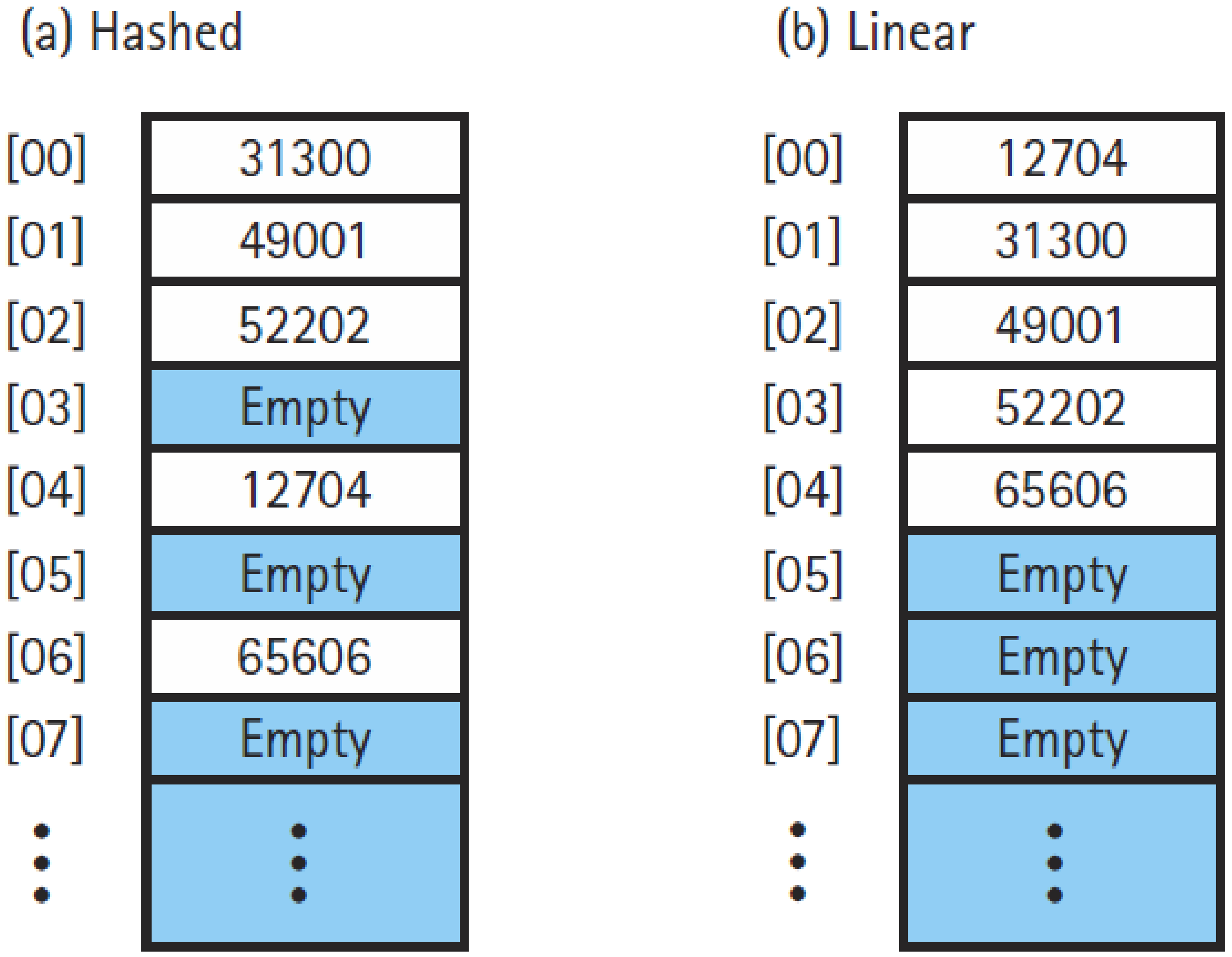
Figure 10.17 *Using a hash function to determine the location of the element in an array*

```
template<class ItemType>
void ListType<ItemType>::InsertItem(ItemType item)
// Post: item is stored in the array at position item.Hash().
{
  int location;

  location = item.Hash();
  info[location] = item;
  length++;
}
```

Collision The condition resulting when two or more keys produce the same hash location

Linear probing Resolving a hash collision by sequentially searching a hash table beginning at the location returned by the hash function



(a) Hashed

[00]	31300
[01]	49001
[02]	52202
[03]	Empty
[04]	12704
[05]	Empty
[06]	65606
[07]	Empty
⋮	⋮

(b) Linear

[00]	12704
[01]	31300
[02]	49001
[03]	52202
[04]	65606
[05]	Empty
[06]	Empty
[07]	Empty
⋮	⋮

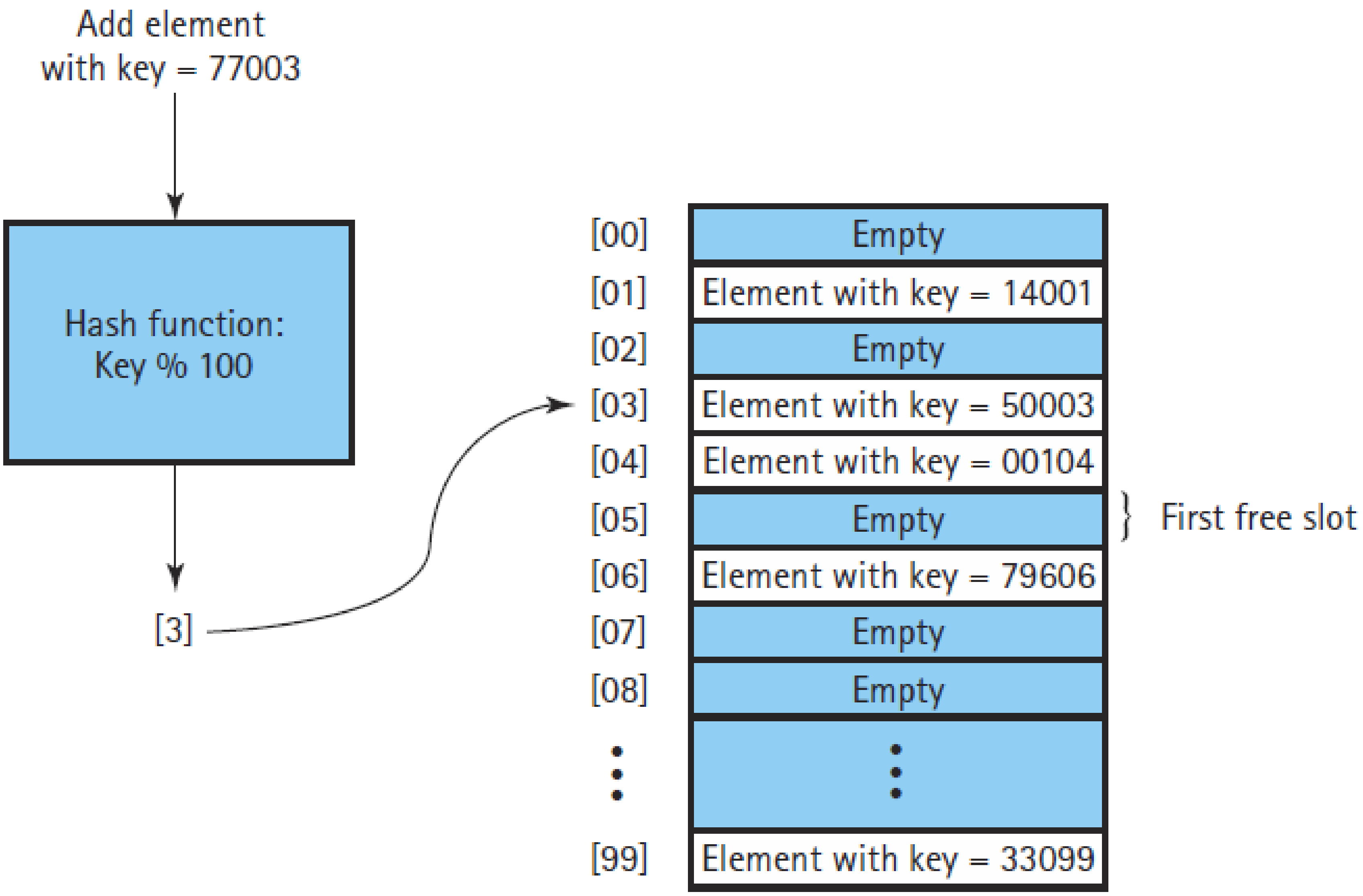


Figure 10.19 *Handling collisions with linear probing*

```
template<class ItemType>
void ListType<ItemType>::InsertItem(ItemType item)
// Post: item is stored in the array at position item.Hash()
//       or the next free spot.
{
  int location;

  location = item.Hash();
  while (info[location] != emptyItem)
    location = (location + 1) % MAX_ITEMS;
  info[location] = item;
  length++;
}
```

```
template<class ItemType>
void ListType<ItemType>::RetrieveItem(ItemType& item, bool& found)
{
  int location;
  int startLoc;
  bool moreToSearch = true;

  startLoc = item.Hash();
  location = startLoc;
  do
  {
    if (info[location] == item || info[location] == emptyItem)
      moreToSearch = false;
    else
      location = (location + 1) % MAX_ITEMS;
  } while (location != startLoc && moreToSearch);
  found = (info[location] == item);
  if (found)
    item = info[location];
}
```

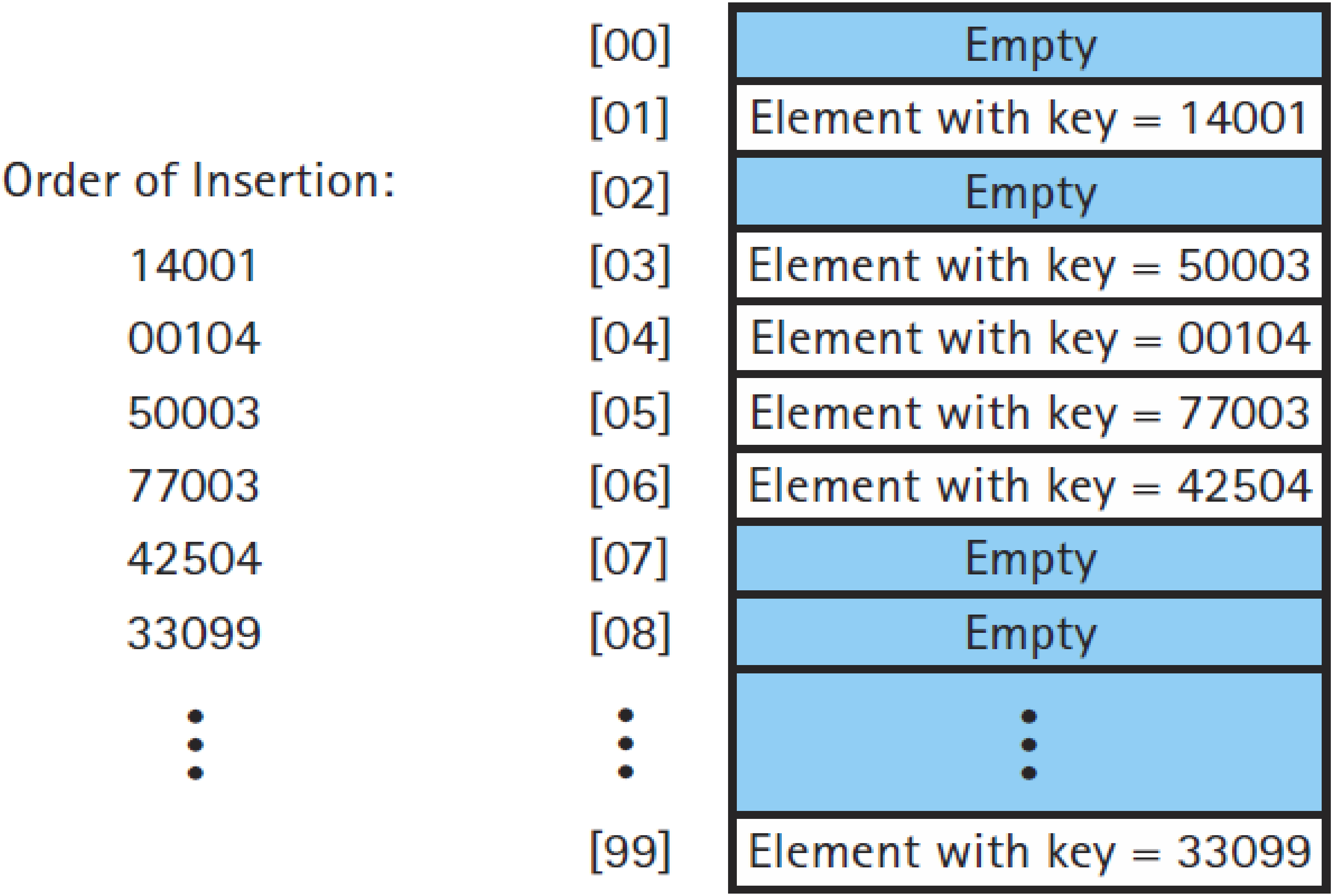


Figure 10.20 *A hash program with linear probing*

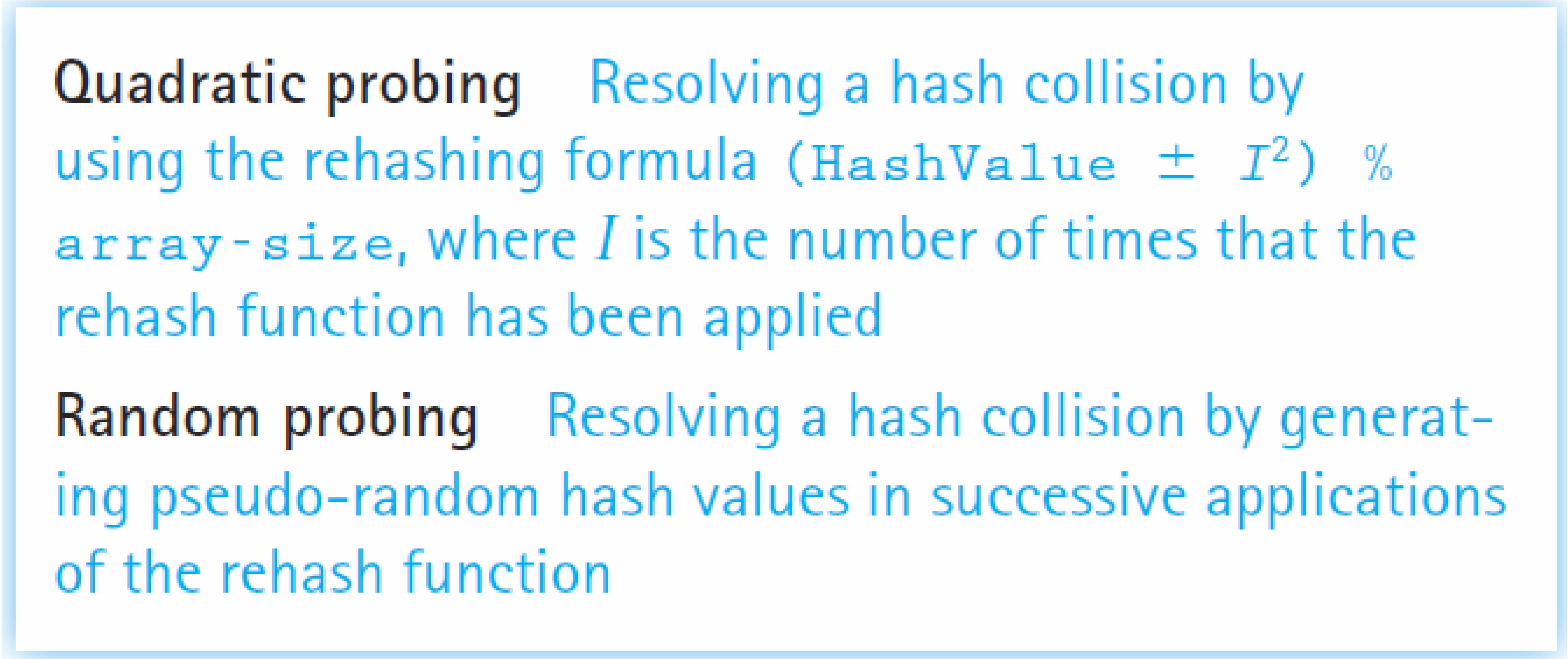
Clustering The tendency of elements to become unevenly distributed in the hash table, with many elements clustering around a single hash location

Rehashing Resolving a collision by computing a new hash location from a hash function that manipulates the original location rather than the element's key

For rehashing with linear probing, you can use any function

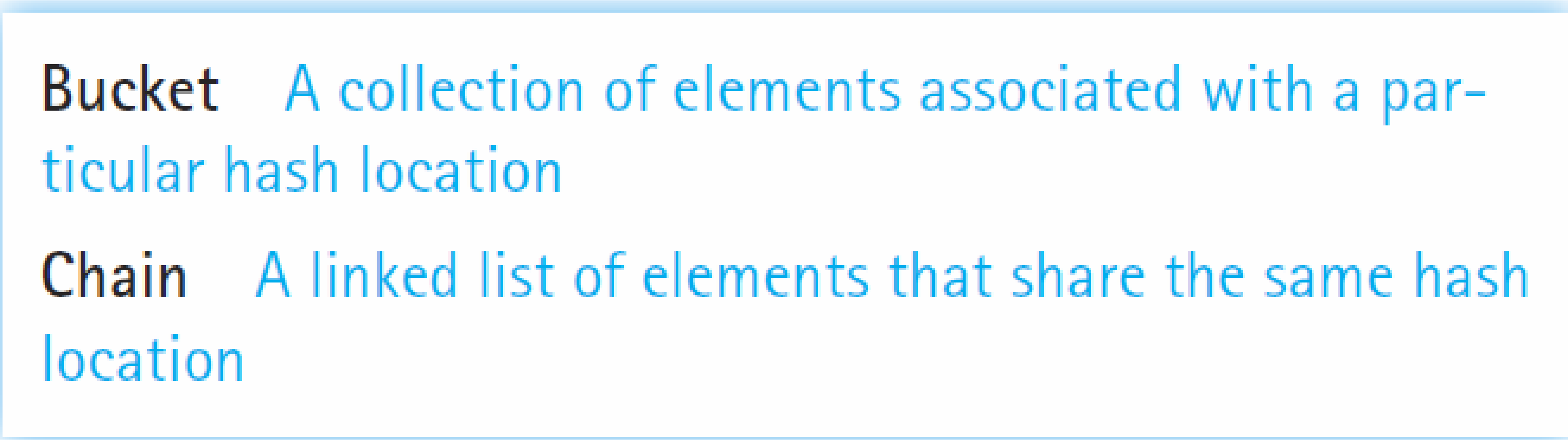
(HashValue + constant) % array-size

(HashValue + 3) % 100



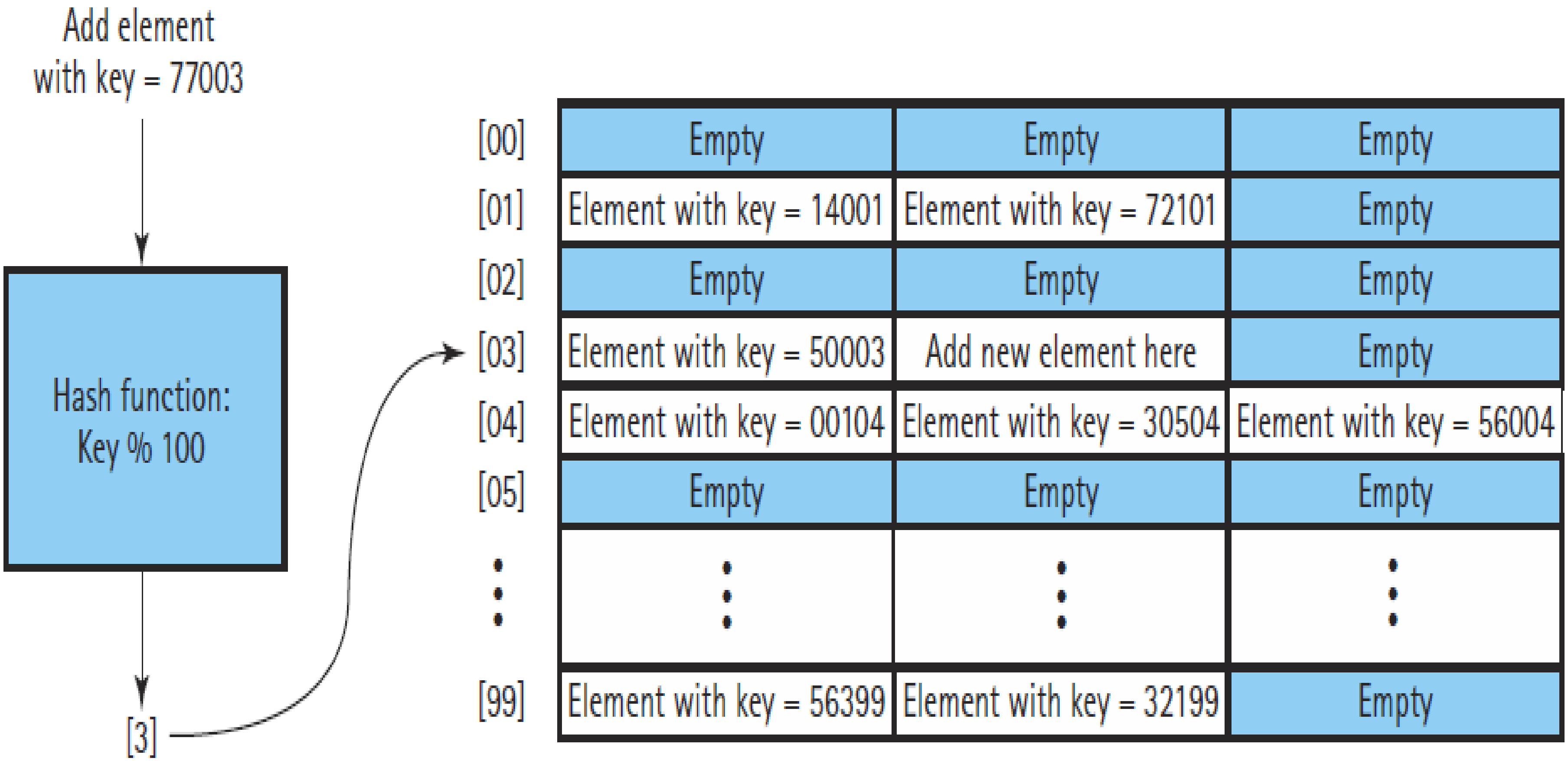
Quadratic probing Resolving a hash collision by using the rehashing formula (`HashValue` ± I^2) % `array-size`, where I is the number of times that the rehash function has been applied

Random probing Resolving a hash collision by generating pseudo-random hash values in successive applications of the rehash function



Bucket A collection of elements associated with a particular hash location

Chain A linked list of elements that share the same hash location



Add element with key = 77003

Hash function:
Key % 100

[3]

[00]	Empty	Empty	Empty
[01]	Element with key = 14001	Element with key = 72101	Empty
[02]	Empty	Empty	Empty
[03]	Element with key = 50003	Add new element here	Empty
[04]	Element with key = 00104	Element with key = 30504	Element with key = 56004
[05]	Empty	Empty	Empty
⋮	⋮	⋮	⋮
[99]	Element with key = 56399	Element with key = 32199	Empty

Figure 10.22 *Handling collisions by hashing with buckets*

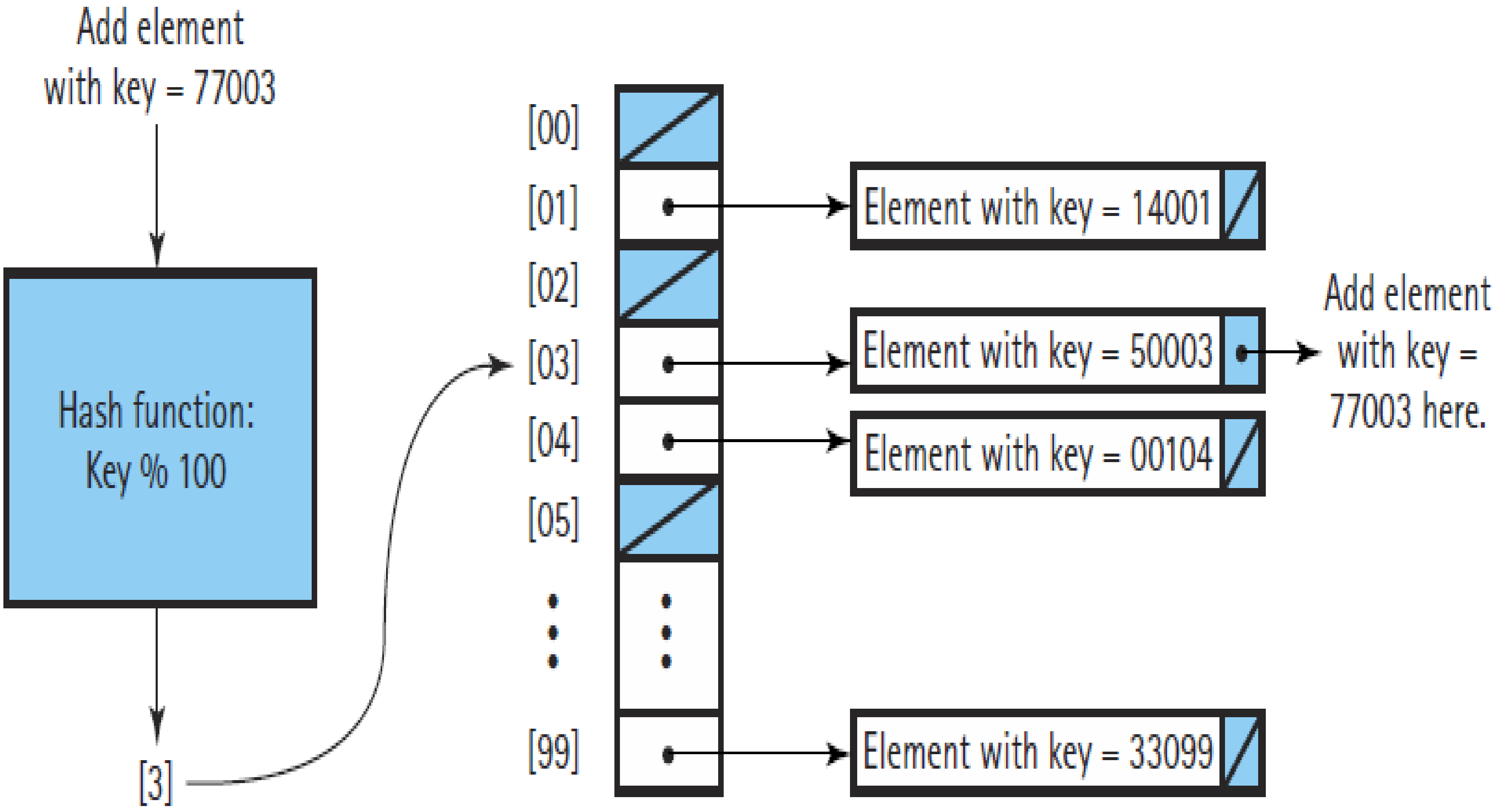


Figure 10.23 *Handling collisions by hashing with chaining*

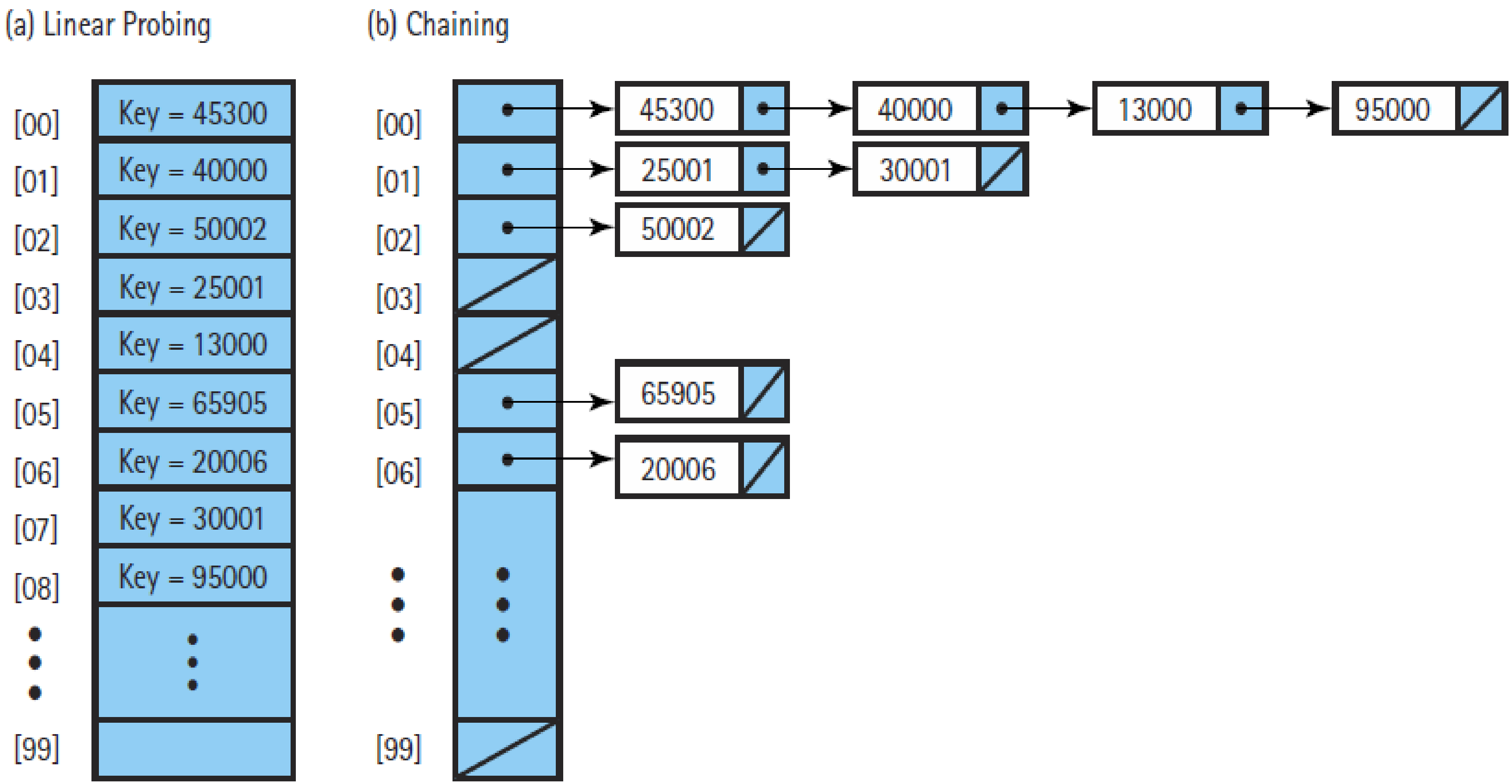
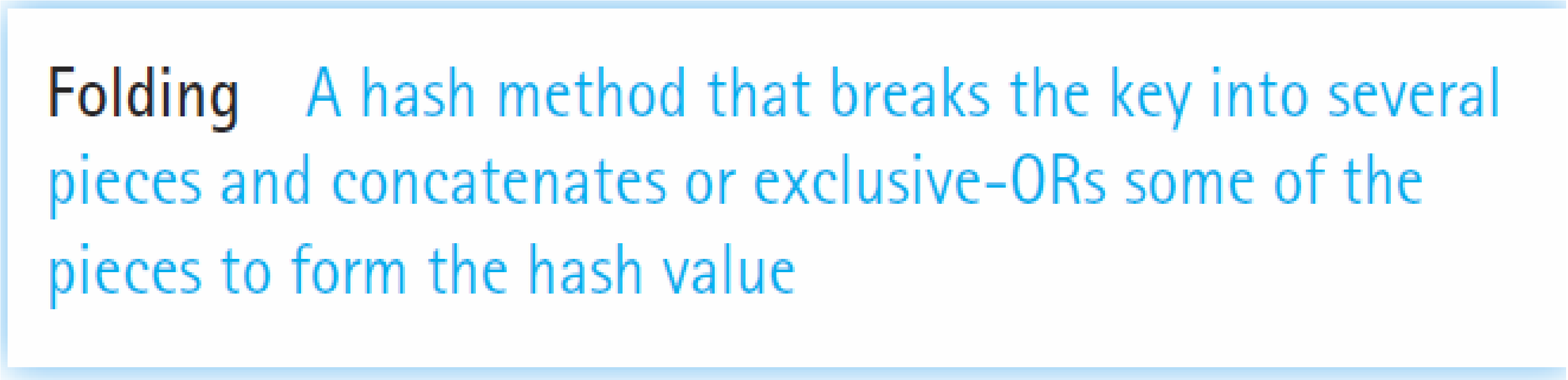


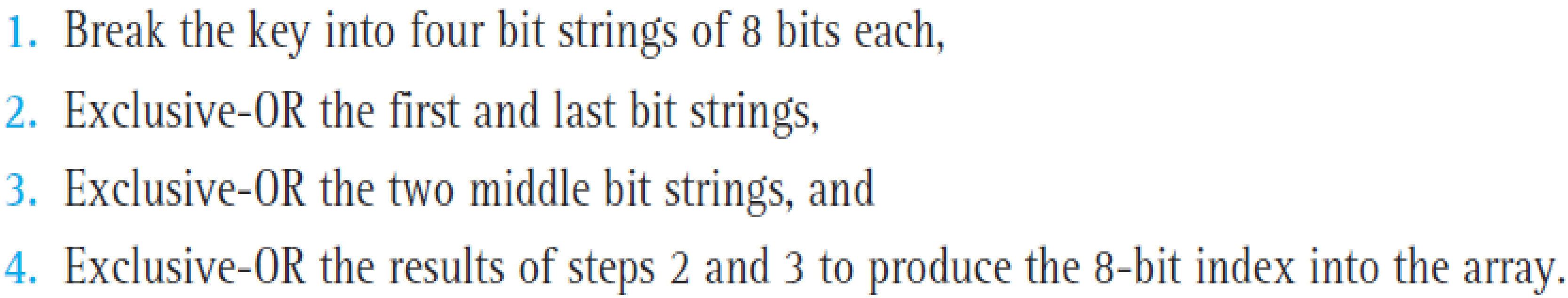
Figure 10.24 *Comparison of linear probing and chaining schemes*

Choose a good hash function

- Division method- **key %table-size**
- **Folding**



Folding A hash method that breaks the key into several pieces and concatenates or exclusive-ORs some of the pieces to form the hash value



1. Break the key into four bit strings of 8 bits each,
2. Exclusive-OR the first and last bit strings,
3. Exclusive-OR the two middle bit strings, and
4. Exclusive-OR the results of steps 2 and 3 to produce the 8-bit index into the array.

00000000000010010110111110100011