Introducing Hashing

Chapter 21

Data Structures and Abstractions with Java, 4e, Global Edition
Frank Carrano

Definition

- Hashing: a technique that determines this index using only an entry's search key
- Hash function
  - Takes a search key and produces the integer index of an element in the hash table
  - Search key—maps, or hashes, to the index

Ideal Hashing

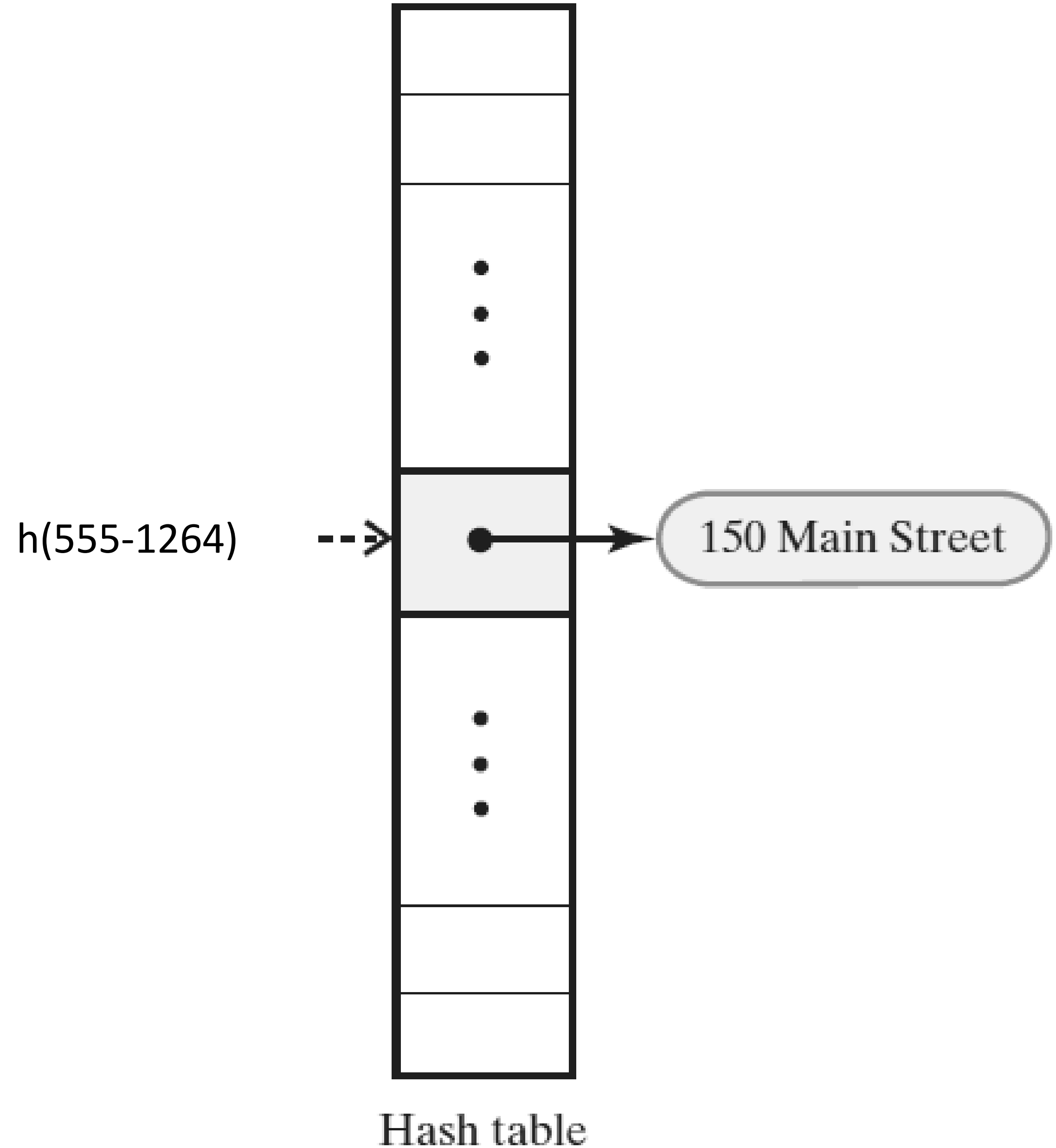

FIGURE 21-1 A hash function indexes its hash table

Ideal Hashing

```
Algorithm add(key, value)
index = h(key)
hashTable[index] = value

Algorithm getValue(key)
index = h(key)
return hashTable[index]
```

Simple algorithms for the dictionary operations that add and retrieve

Typical Hashing

Typical hash functions—perform two steps:

1. Convert search key to an integer called the hash code.
2. Compress hash code into the range of indices for hash table.

```
Algorithm getHashIndex(phoneNumber)
// Returns an index to an array of tableSize locations.

i = last four digits of phoneNumber
return i % tableSize
```

Typical Hashing

- Typical hash functions are not perfect,
  - Can allow more than one search key to map into a single index
  - Causes a collision in the hash table
- Example
  - Consider tableSize = 101
  - `getHashIndex(555-1264)` = 52
  - `getHashIndex(555-8132)` = 52 also!!!

Typical Hashing

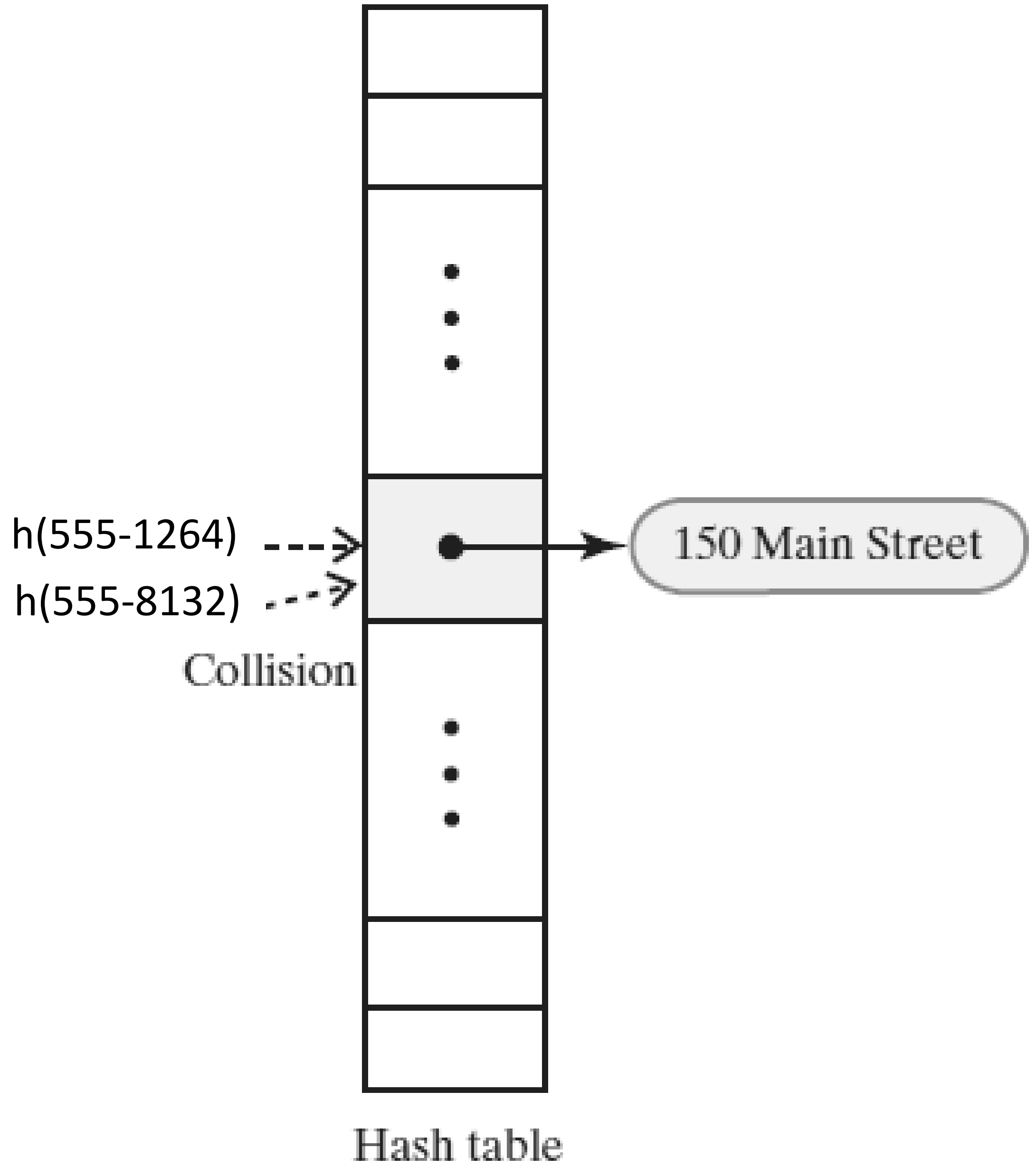

FIGURE 21-2 A collision caused by the hash function *h*

Hash Functions

- A good hash function should
  - Minimize collisions
  - Be fast to compute
- To reduce the chance of a collision
  - Choose a hash function that distributes entries uniformly throughout hash table.

Computing Hash Codes

- Java's base class `Object` has a method `hashCode` that returns an integer hash code
  - A class should define its own version of `hashCode`
- A hash code for a string
  - Using a character's Unicode integer is common
  - Better approach: multiply Unicode value of each character by factor based on character's position, then sum

Computing Hash Codes

- Simple hash code for a string example:

```
int sascii(String x, int M) {
  char ch[];
  ch = x.toCharArray();
  int xlength = x.length();
  int i, sum;
  for (sum=0, i=0; i < x.length(); i++)
    sum += ch[i];
  return sum % M;
}
```

Hash Code for a Primitive type

- If data type is `int`,
  - Use the key itself
- For `byte`, `short`, `char`:
  - Cast as `int`
- Other primitive types
  - Manipulate internal binary representations

Compressing a Hash Code

- Common way to scale an integer
  - Use Java % operator, `code % n`
- Best to use an odd number for n
- Prime numbers often give good distribution of hash values

Compressing a Hash Code

```
private int getHashIndex(K key)
{
   int hashIndex = key.hashCode() % hashTable.length;
   if (hashIndex < 0)
      hashIndex = hashIndex + hashTable.length;

   return hashIndex;
} // end getHashIndex
```

Hash function for the ADT dictionary

Resolving Collisions

- Definition: hash function maps search key into a location in hash table already in use
- Two choices:
  - Use another location in the hash table
  - Change the structure of the hash table so that each array location can represent more than one value

Resolving Collisions

- Linear probing
  - Resolves a collision during hashing by examining consecutive locations in hash table
  - Beginning at original hash index
  - Find the next available one
- Table locations checked make up *probe sequence*
- If probe sequence reaches end of table, go to beginning of table (circular hash table)

Resolving Collisions

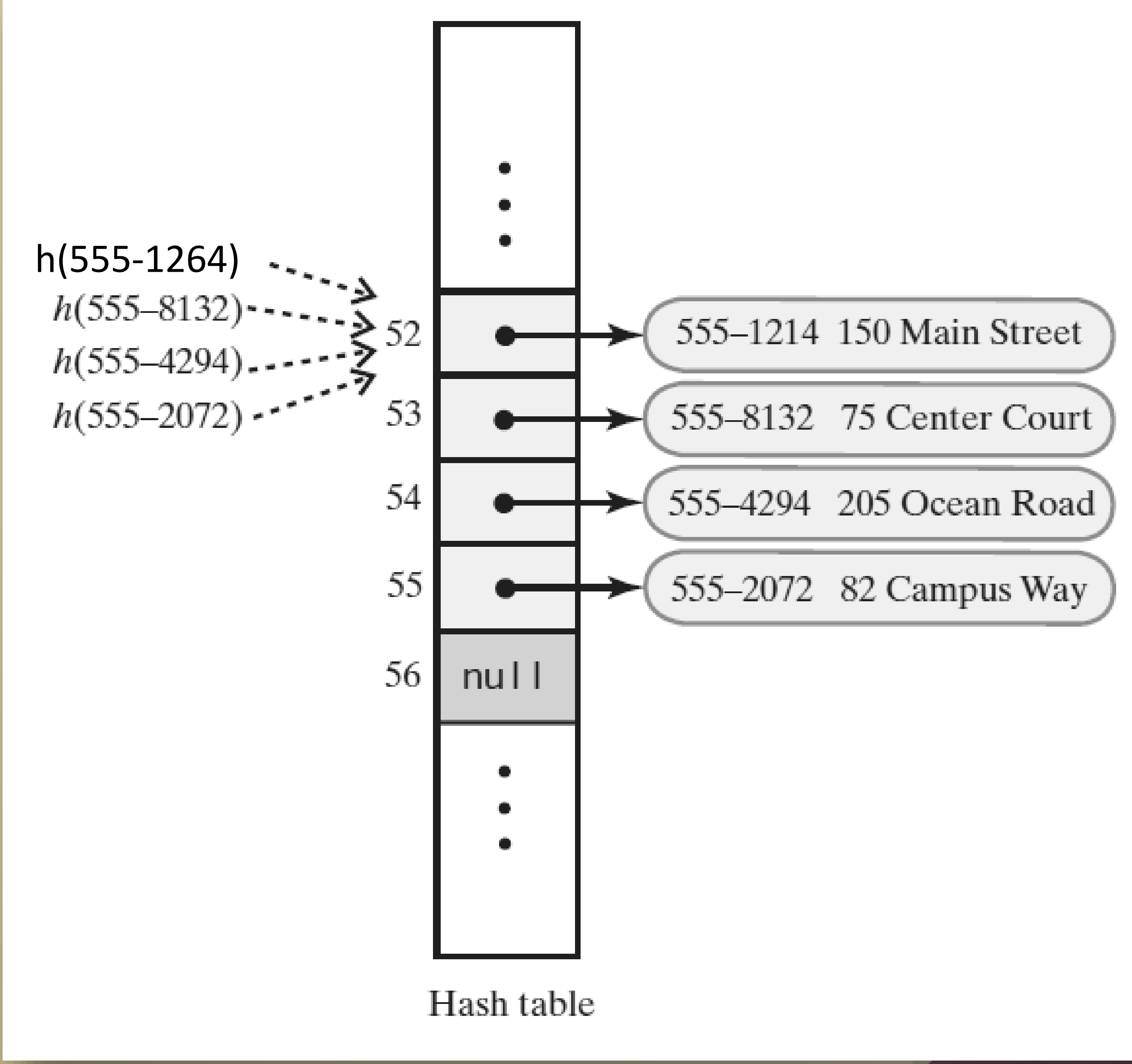

FIGURE 21-4 A revision of the hash table shown in Figure 21-3 when linear probing resolves collisions; each entry contains a search key and its associated value

Resolving Collisions

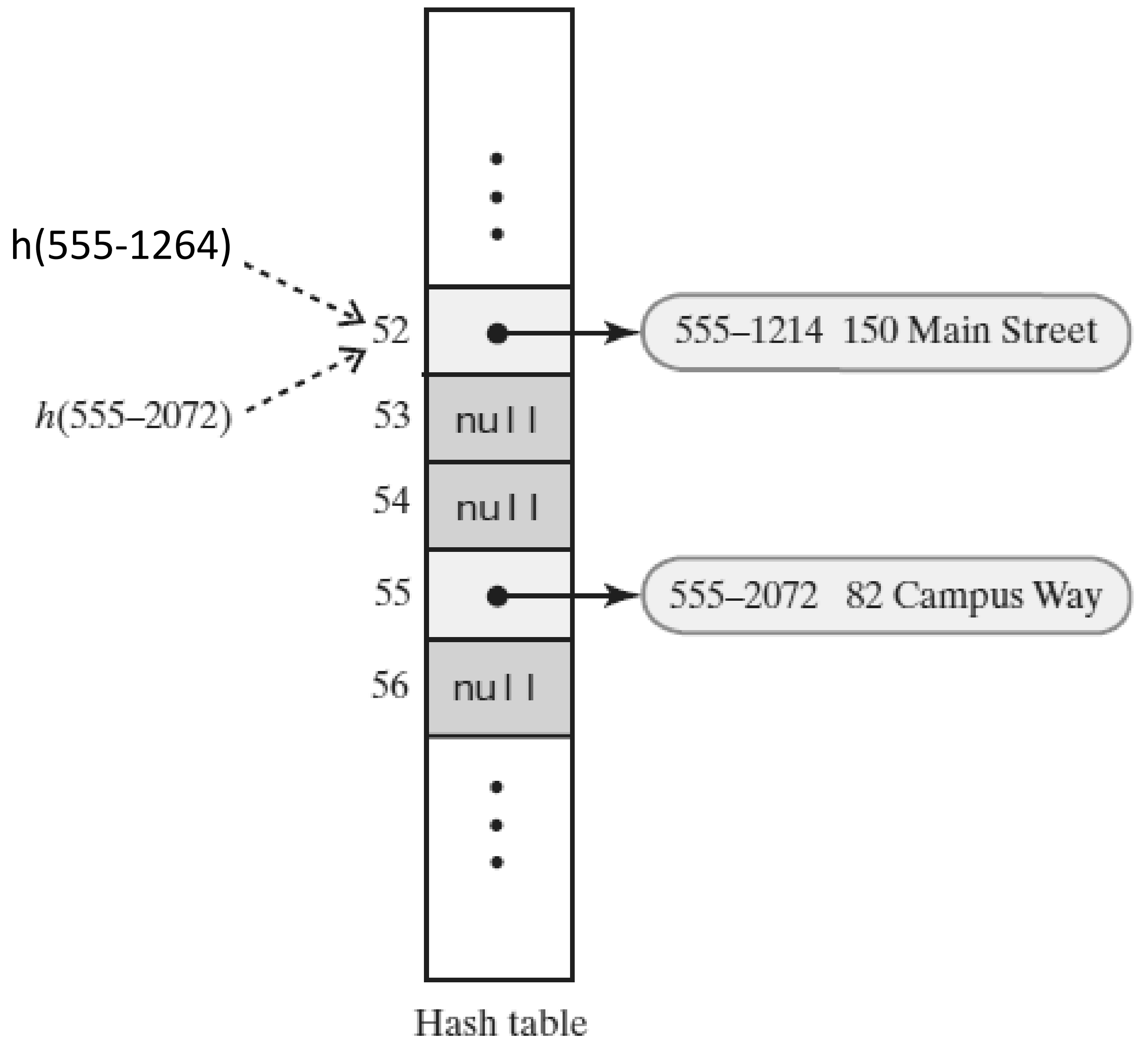

FIGURE 21-5 A hash table if `remove` used `null` to remove entries.

What problem do you see?

Resolving Collisions

- Need to distinguish among three kinds of locations in the hash table
  - Occupied—the location references an entry in the dictionary
  - Empty—the location contains `null` and always has
  - Available—the location's entry was removed from the dictionary

Clustering

- Collisions resolved with linear probing cause groups of consecutive locations in hash table to be occupied
  - Each group is called a *cluster*
- Bigger clusters mean longer search times following collision

Clustering

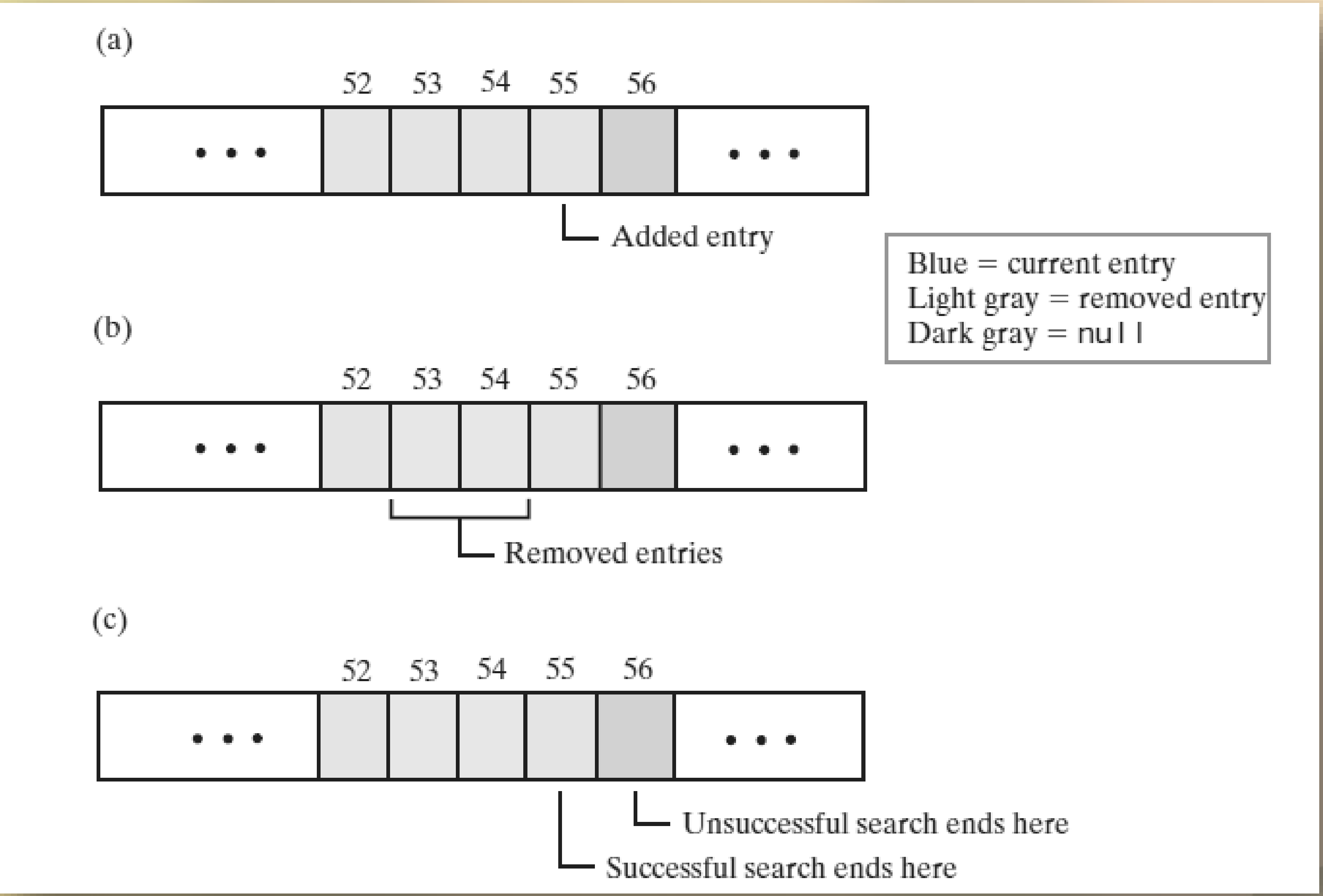

FIGURE 21-6 A linear probe sequence (a) after adding an entry; (b) after removing two entries; (c) after a search;

Clustering

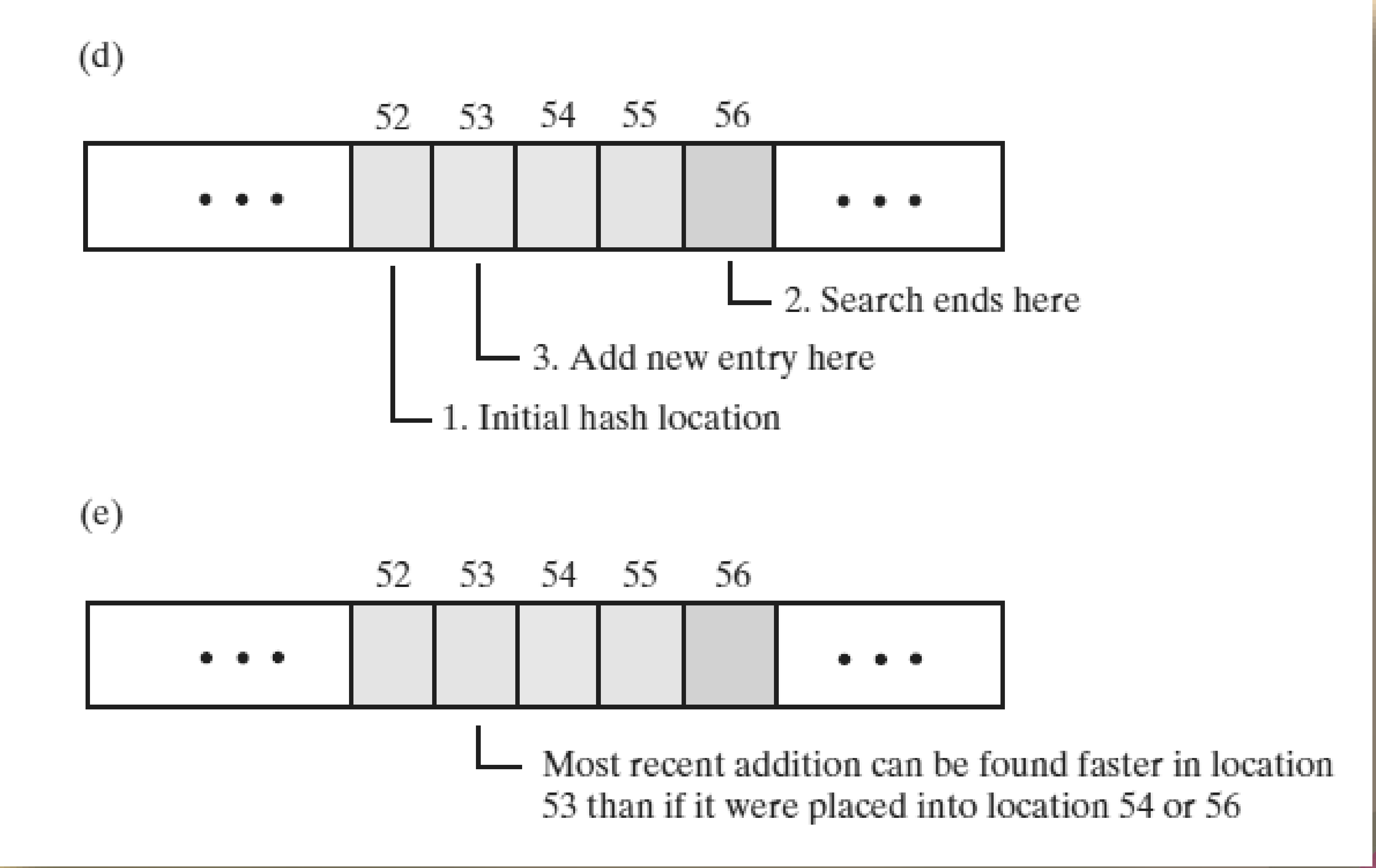

FIGURE 21-6 A linear probe (d) during the search while adding an entry; (e) after an addition to a formerly occupied location

Open Addressing with Quadratic Probing

- Linear probing looks at consecutive locations beginning at index *k*
- Quadratic probing, considers the locations at indices $k + j^2$
  - Uses the indices *k*, *k* + 1, *k* + 4, *k* + 9, …

Open Addressing with Quadratic Probing

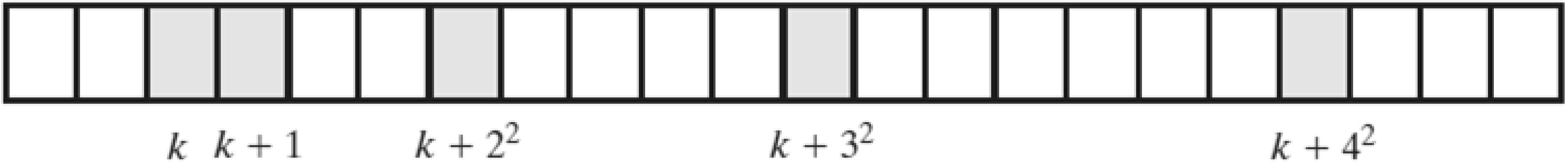

FIGURE 21-7 A probe sequence of length five using quadratic probing

Open Addressing with Double Hashing

- Linear probing and quadratic probing add increments to k to define a probe sequence
  - Both are independent of the search key
- Double hashing uses a second hash function to compute these increments
  - This is a key-dependent method.

Open Addressing with Double Hashing

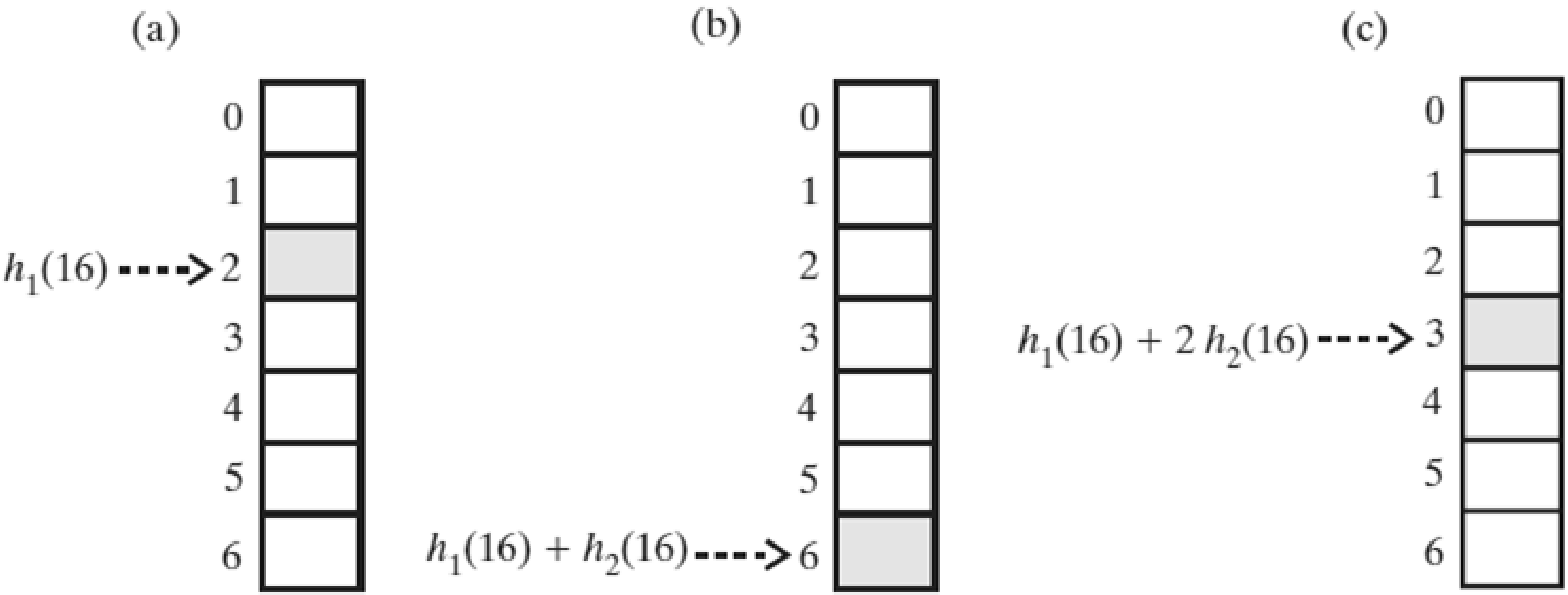

FIGURE 21-8 The first three locations in a probe sequence generated by double hashing for the search key 16

Potential Problem with Open Addressing

- Recall each location is either occupied, empty, or available
  - Frequent additions and removals can result in *no* locations that are null
- Thus searching a probe sequence will not work
- Consider separate chaining as a solution

Separate Chaining

- Alter the structure of the hash table
  - Each location can represent more than one value.
  - Such a location is called a *bucket*
- Decide how to represent a bucket
  - list, sorted list
  - array
  - linked nodes
  - vector

Separate Chaining

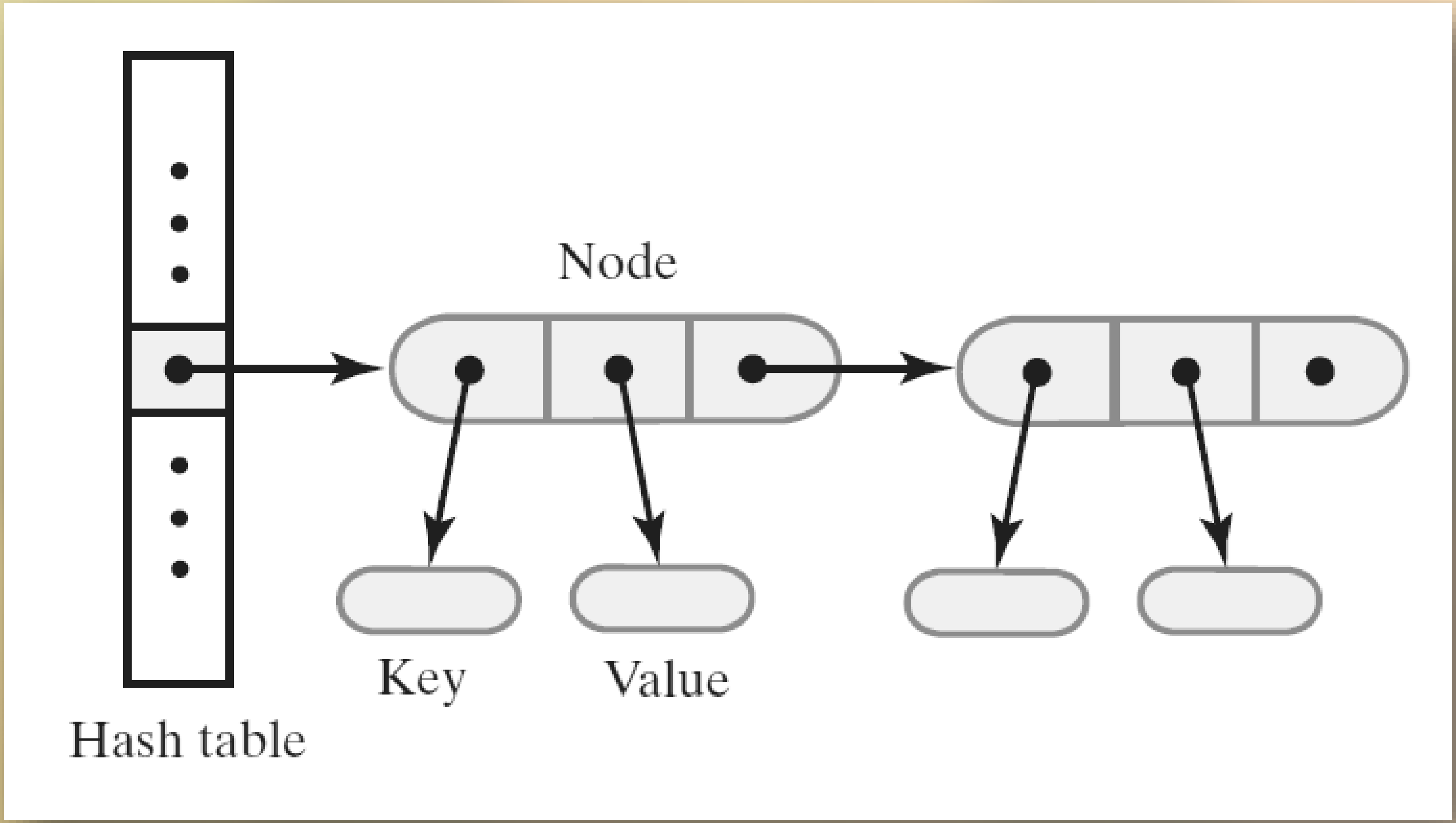

FIGURE 21-9 A hash table for use with separate chaining; each bucket is a chain of linked nodes

Separate Chaining

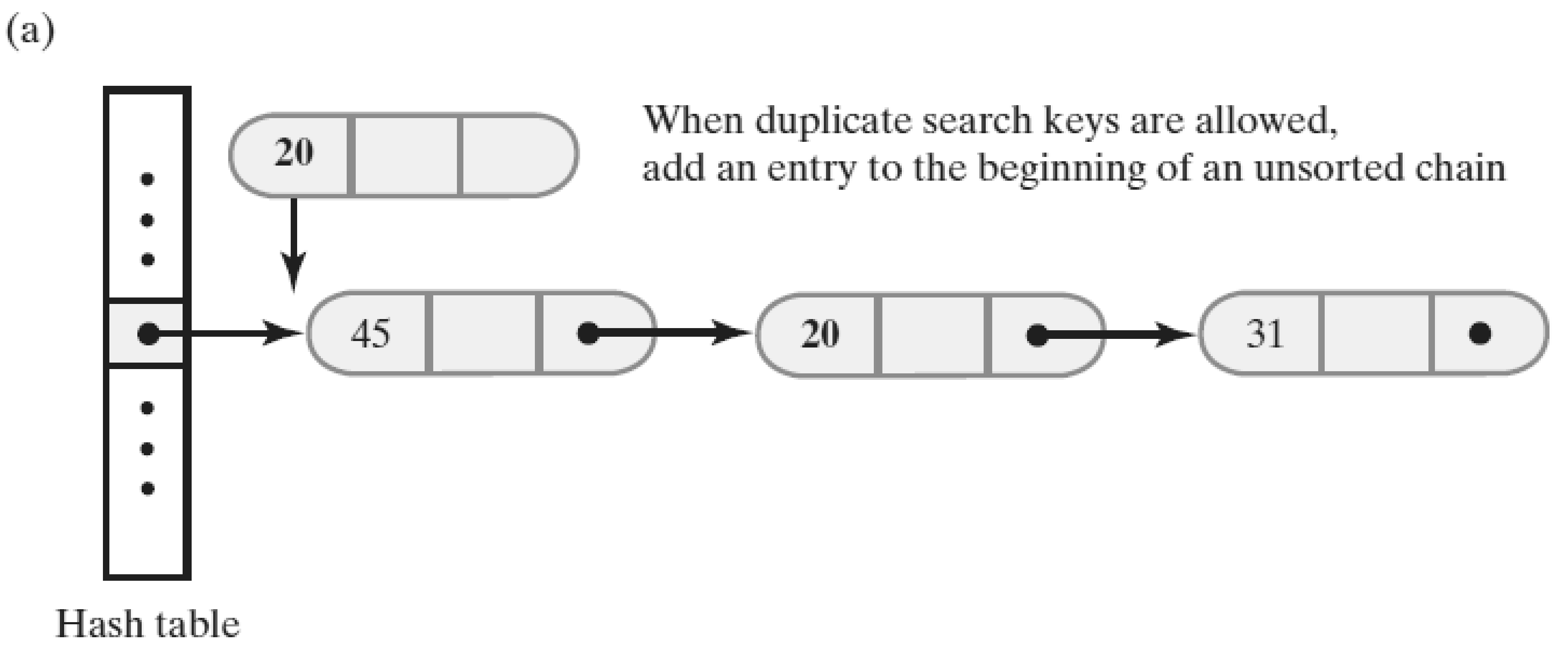

FIGURE 21-10 Where to insert a new entry into a linked bucket when the integer search keys are
(a) unsorted and possibly duplicate;

Separate Chaining

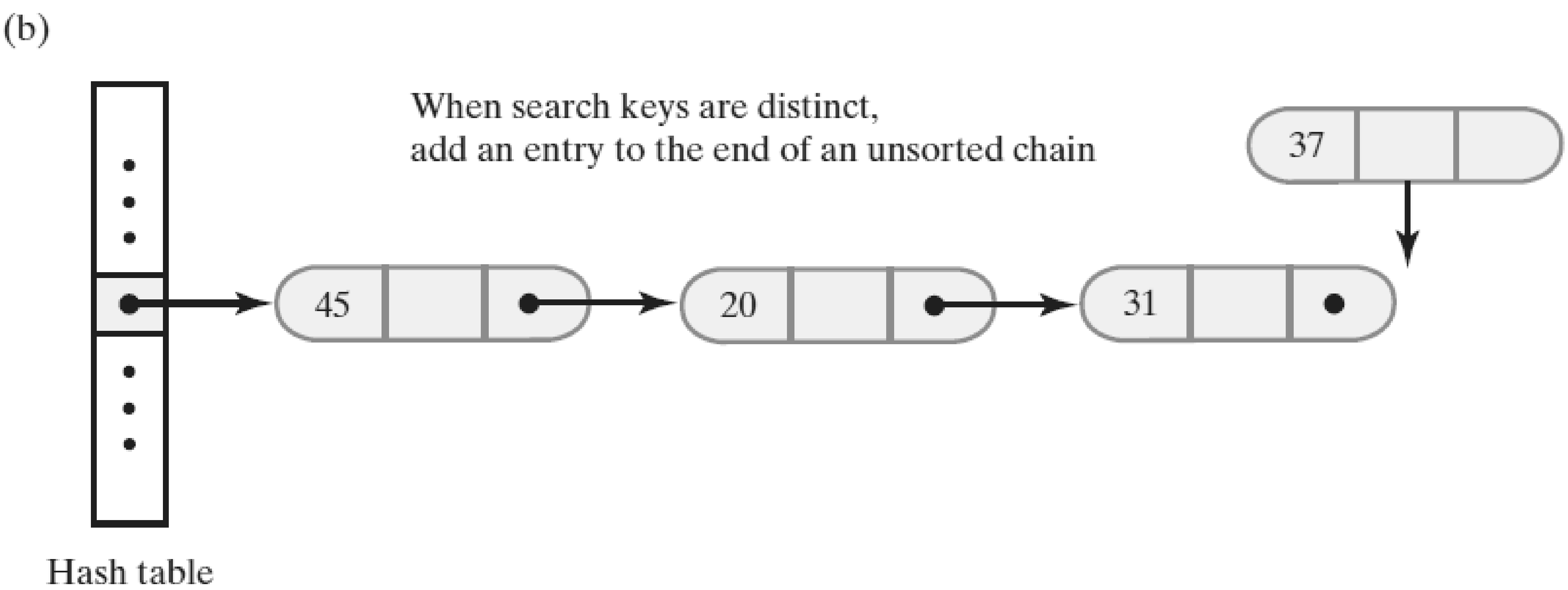

FIGURE 21-10 Where to insert a new entry into a linked bucket when the integer search keys are (b) unsorted and distinct;

Separate Chaining

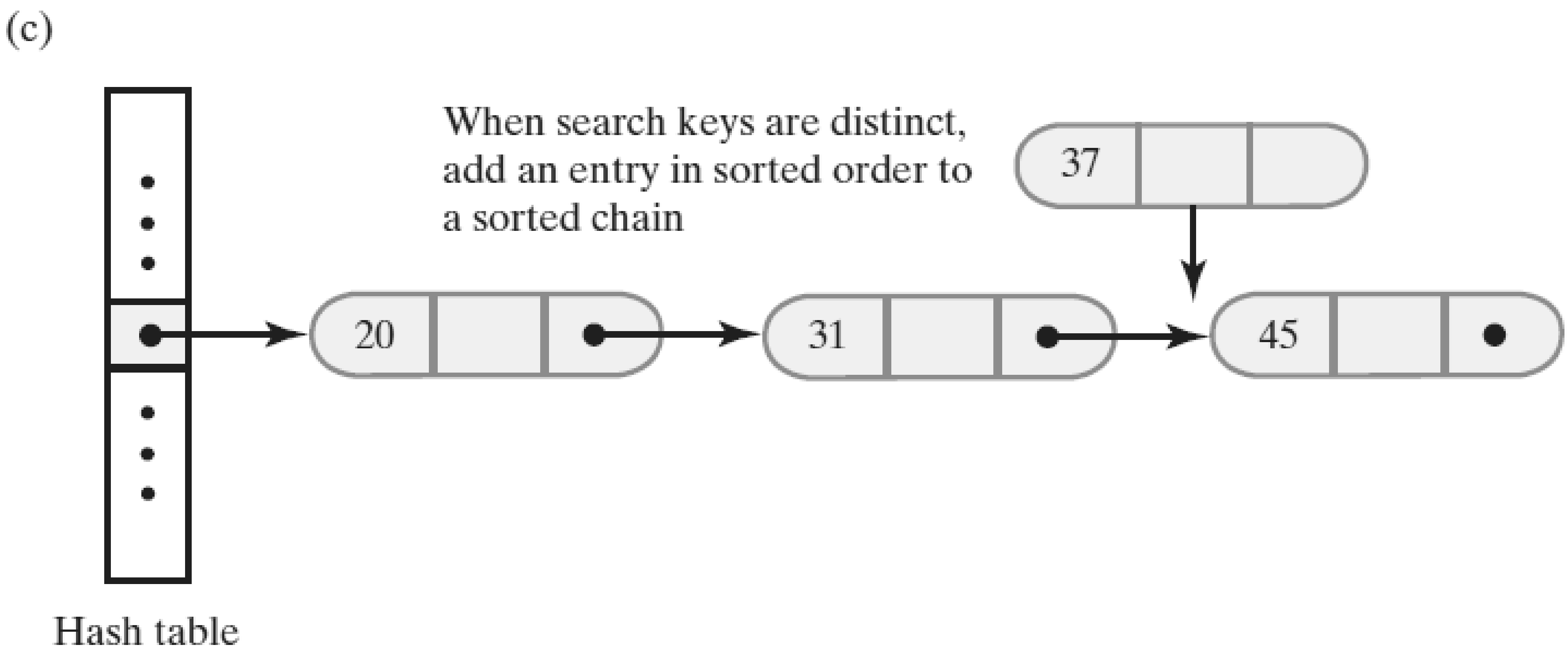

FIGURE 21-10 Where to insert a new entry into a linked bucket when the integer search keys are (c) sorted and distinct

Separate Chaining

```
Algorithm add(key, value)
index = getHashIndex(key)
if (hashTable[index] == null)
{
   hashTable[index] = new Node(key, value)
   numberOfEntries++
   return null
}
else
{
   Search the chain that begins at hashTable[index] for a node that contains key
   if (key is found)
   { // Assume currentNode references the node that contains key
```


Algorithm for the dictionary's **add** method.

Separate Chaining

```
if (key is found)
{ // Assume currentNode references the node that contains key
    oldValue = currentNode.getValue()
    currentNode.setValue(value)
    return oldValue
}
else // Add new node to end of chain
{ // Assume nodeBefore references the last node
    newNode = new Node(key, value)
    nodeBefore.setNextNode(newNode)
    numberOfEntries++
    return null
}
}
```

Algorithm for the dictionary's `add` method.

Separate Chaining

```
Algorithm remove(key)
index = getHashIndex(key)
Search the chain that begins at hashTable[index] for a node that contains key
if (key is found)
{
    Remove the node that contains key from the chain
    numberOfEntries--
    return value in removed node
}
else
    return null

Algorithm getValue(key)
index = getHashIndex(key)
Search the chain that begins at hashTable[index] for a node that contains key
if (key is found)
    return value in found node
else
    return null
```


Algorithm for the dictionary's `remove` method

End

Chapter 21