Introducing Hashing

Chapter 21

THIRD EDITION

Data Structures and Abstractions with Java™

FRANK M. CARRANO

Contents

A demo of hashing (after)

						ARRAY		
insert	hash	index = hash % size		linear	quadratic	index	value	
Chris	289	0				0	Jack	<==Chris
Drew	312	6				1	Chris	
Darren	1039	2				2	Darren	<--Briana
Briana	512	2				3	Briana	
Sally	345	5		7	14 = 5 + 9	4	Joe	
						5	Jack	<- Sally
						6	Drew	
						7		
						8	Steve	
						9	Rock	
						10	Blue	
	add	total				11		
	1					12	John	
	3	4				13	Dave	
	5	9				14	Sally	
	7	16				15	Rex	
	9	25				16	Rianne	
	11	36						

Contents

Objectives

- Describe basic idea of hashing
- Describe purpose of hash table, hash function, perfect hash function
- Explain why to override method **`hashCode`** for objects used as search keys
- Describe how hash function compresses hash code into index to the hash table

Objectives

- Describe algorithms for dictionary operations **`getValue`**, **`add`**, and **`remove`** when open addressing resolves collisions
- Describe separate chaining as method to resolve collisions

Objectives

- Describe algorithms for dictionary operations **`getValue`**, **`add`**, and **`remove`** when separate chaining resolves collisions
- Describe clustering and problems it causes

What Is Hashing?

- Method to locate data quickly
  - Ideally has O(1) search times
  - Yet cannot do easy traversal of data items
- Technique that determines index using only a search key
- Hash function locates correct item in hash table
  - Maps or “hashes to” entry

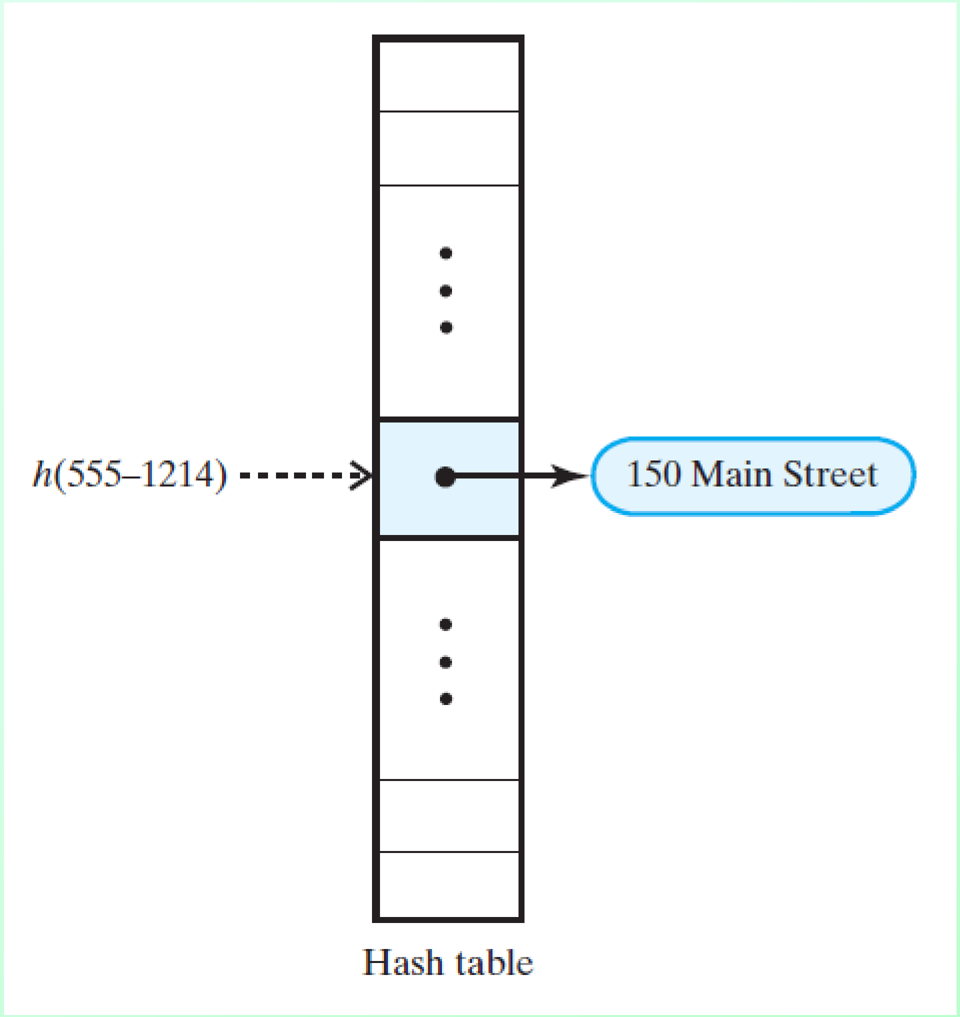


Figure 21-1 A hash function indexes its hash table

Typical Hashing

- Algorithm will
  - Convert search key to integer called hash code.
  - Compress hash code into range of indices for hash table.
- Typical hash functions not perfect
  - Can allow more than one search key to map into single index
  - Causes “collision” in hash table

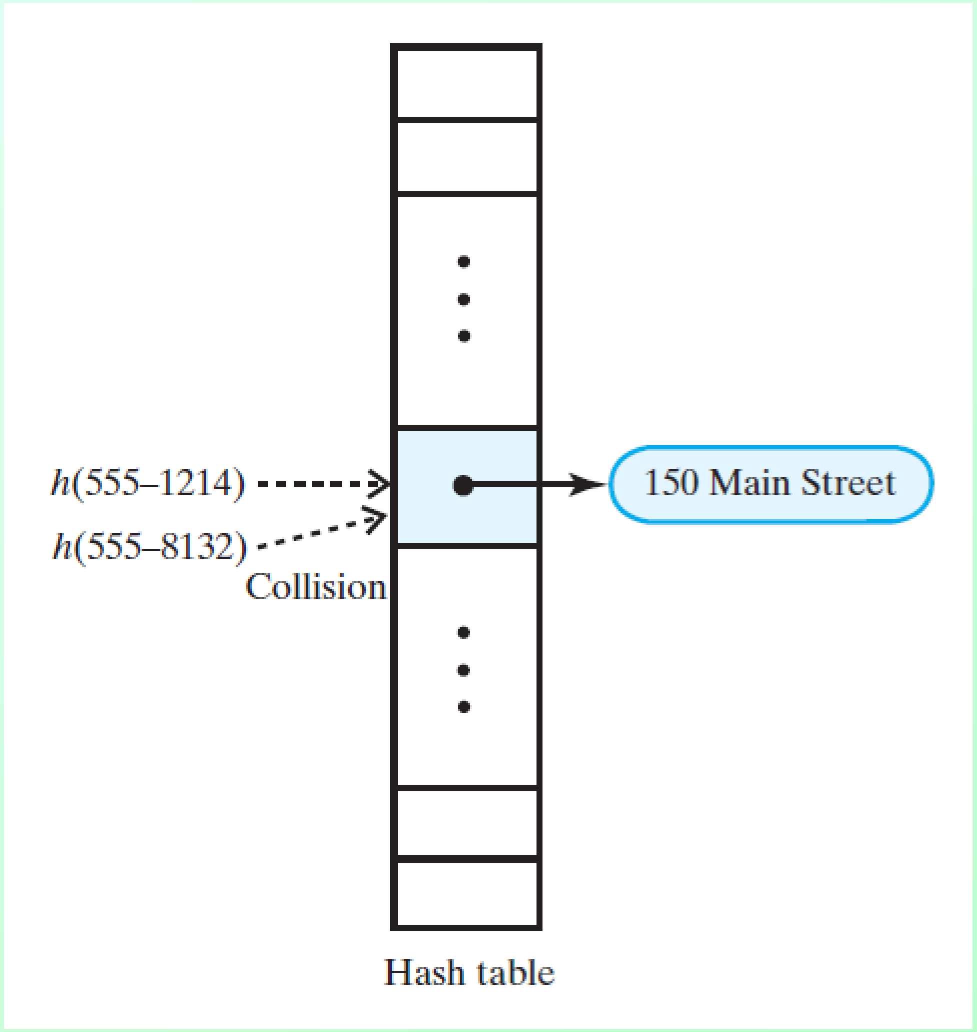


Figure 21-2 A collision caused by the hash function *h*

Computing Hash Codes

- Must override Java **`Object`** method **`hashCode`**
- Guidelines for new **`hashCode`** method
  - If class overrides method **`equals`**, it should override **`hashCode`**.
  - If method **`equals`** considers two objects equal, **`hashCode`** must return same value for both objects.

Computing Hash Codes

- Guidelines continued …
  - If you call an object's **`hashCode`** more than once during execution of a program, and if object's data remains same during this time, **`hashCode`** must return the same value.
  - Object's hash code during one execution of a program *can* differ from its hash code during *another* execution of the same program.

Hash Code for a String

- Assign integer to each character in string
  - Use 1 – 26 for ‘a’ to ‘z’
  - Use Unicode integer
- Possible to sum the integers of the characters for the hash code
- Better solution
  - Multiply Unicode value of each character by factor based on character’s position

$$u_0 g^{n-1} + u_1 g^{n-2} + \ldots + u_{n-2} g + u_{n-1}$$

Question 1 Calculate the hash code for the string Java when g is 31. Compare your result with the value of the expression “Java" .hashCode() .

Question 1 Calculate the hash code for the string Java when g is 31. Compare your result with the value of the expression “Java" .hashCode() .

2301506. "Java".hashCode() has the same value.

Hash Code for a Primitive Type

- For **`int`**
  - Use the value
- For **`byte`**, **`short`**, or **`char`**
  - Cast into an **`int`**
- Other primitive types
  - Manipulate internal binary representations

Compressing a Hash Code

- Scale an integer by using Java % operator
  - For code *c* and size of table *n*, use **`c % n`**
  - Result is remainder of division
- If *n* is prime, provides values distributed in range *0* to *n – 1*

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

   return hashIndex;
} // end getHashIndex
```

Question 2 The previous question asked you to compute the hash code for the string Java. Use that value to calculate what getHashIndex(" Java") returns when the length of the hash table is 101.

Question 3 What one-character string, when passed to getHashIndex, will cause the method to return the same value as in the previous question?

Question 2 The previous question asked you to compute the hash code for the string Java. (2301506) Use that value to calculate what getHashIndex(" Java") returns when the length of the hash table is 101.

19

Question 3 What one-character string, when passed to getHashIndex, will cause the method to return the same value as in the previous question?

“x”

Resolving Collisions

- Open addressing scheme locates alternate open location in hash table
- Linear probing
  - Collision at a[k]
  - Check for open slot at a[k + 1], a[k+2], etc.
- For retrievals
  - Must check for agreement of search key in successive elements of array

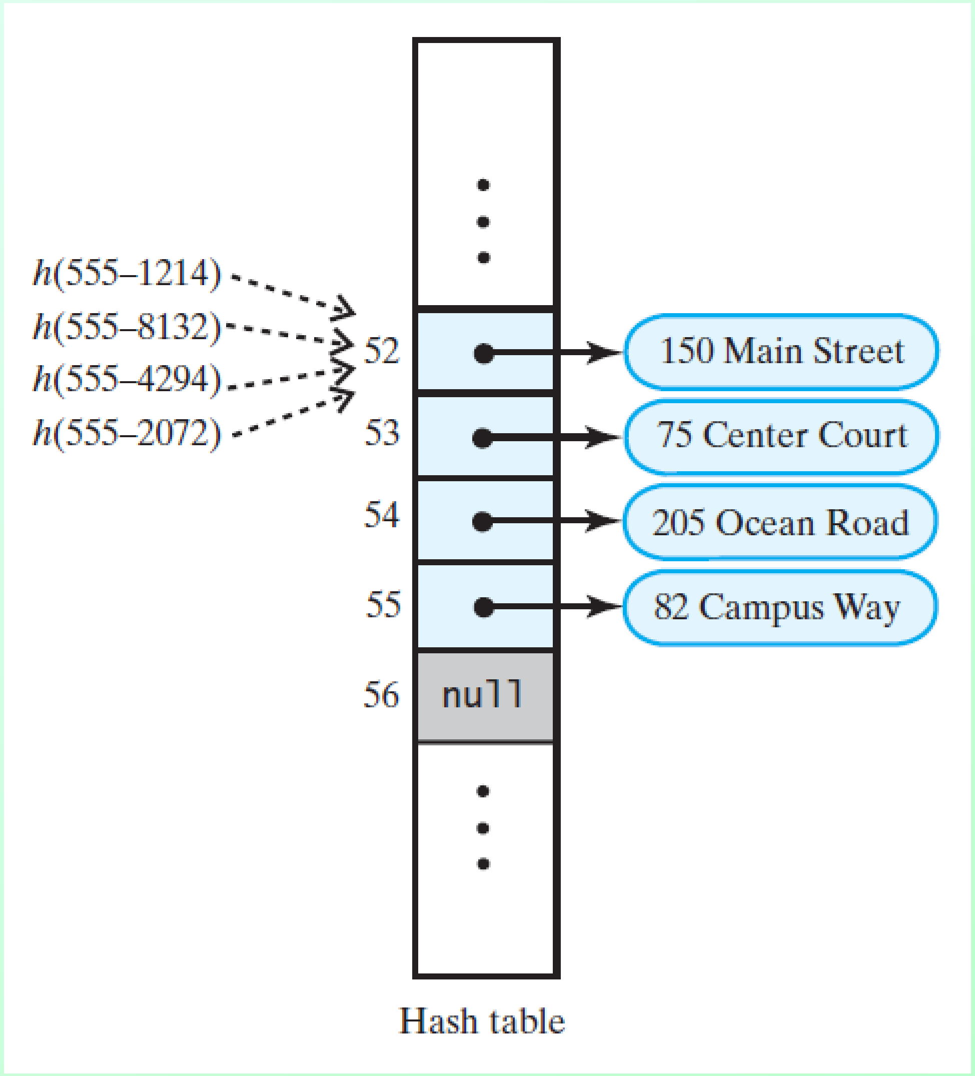


Figure 21-3 The effect of linear probing after adding four entries whose search keys hash to the same index

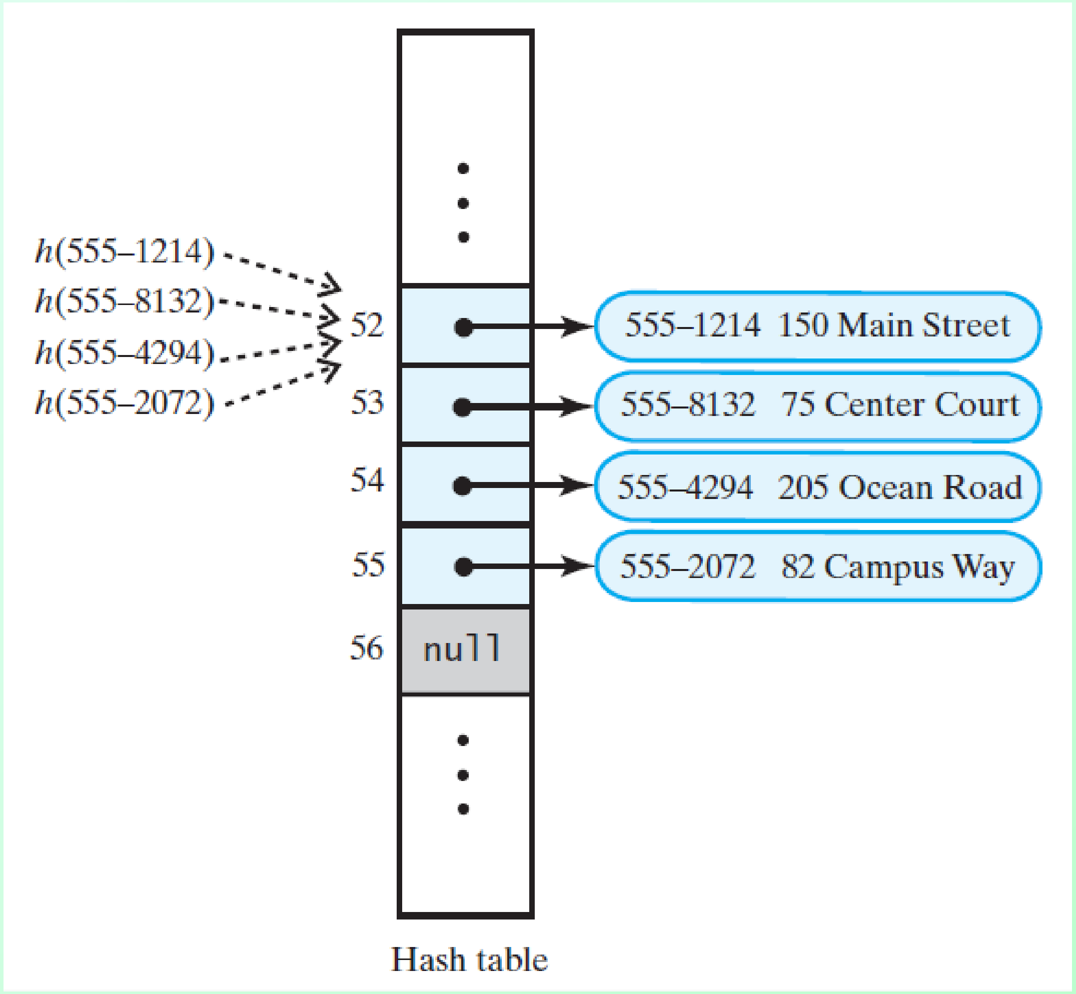


Figure 21-4 A revision of the hash table shown in Figure 21-3 when linear probing resolves collisions; each entry contains a search key and its associated value

Removals

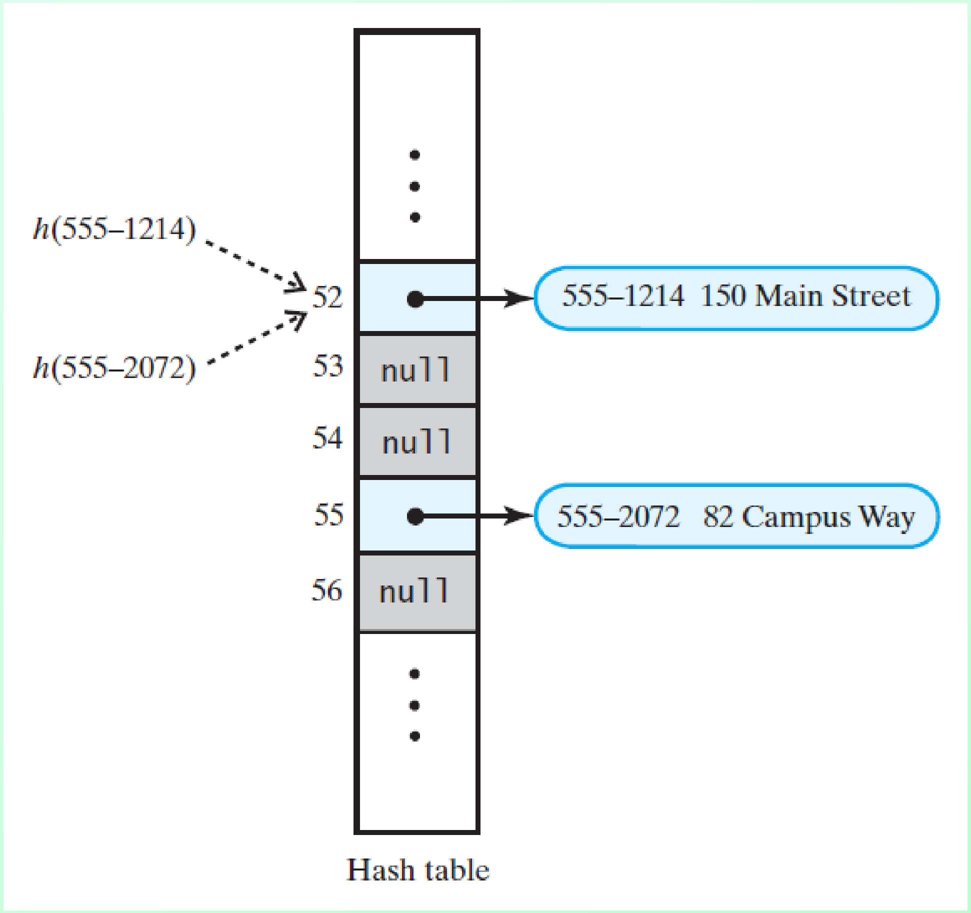
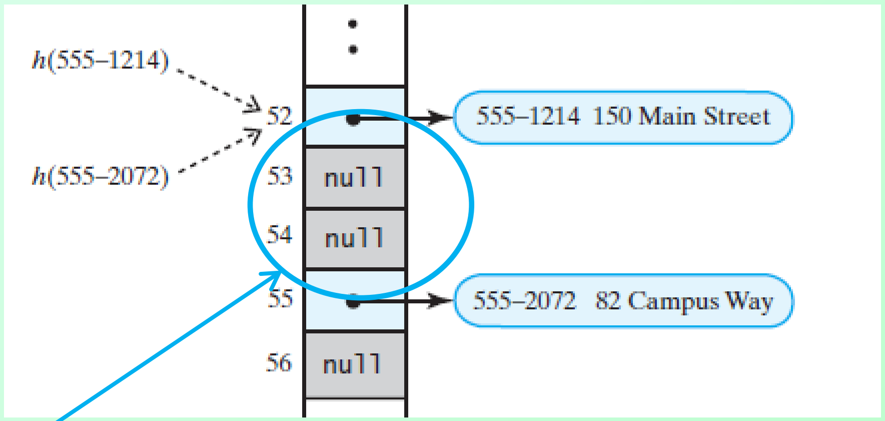


Figure 21-5 A hash table if **`remove`** used **`null`** to remove entries

Removals



- Problem
  - h(555-2027) goes to location 52
  - Collision occurs
  - Linear probing cannot find desired data
- Removal must be marked differently
  - Instead of null, use a value to show slot is available but location's entry was removed
- Location reused later for an add

Question 4 Suggest ways to implement the three states of a location in a hash table. Should this state be a responsibility of the location or of the dictionary entry that it references?

Question 4 Suggest ways to implement the three states of a location in a hash table. Should this state be a responsibility of the location or of the dictionary entry that it references?

Since the implementation defines both the hash table and the dictionary entry, you have a choice as to where to add a field to indicate the state of a location in a hash table. You could add a field having three states to each table location, but you really need only a boolean field, since a null location is empty. If the field is true, the location is occupied; if it is false, it is available.

Adding a similar data field to the dictionary entry instead of to the hash table leads to a cleaner implementation. As before, if the table location is null, it is empty. If the entry's field is true, the location is occupied; if it is false, it is available. Note that the implementation that appears in the next chapter uses this scheme.

Clustering

- When collisions resolved with linear probing
  - Groups of consecutive locations occupied
  - Called primary clustering
- If clusters grow (size and number) can cause problems
  - Longer searches for retrievals

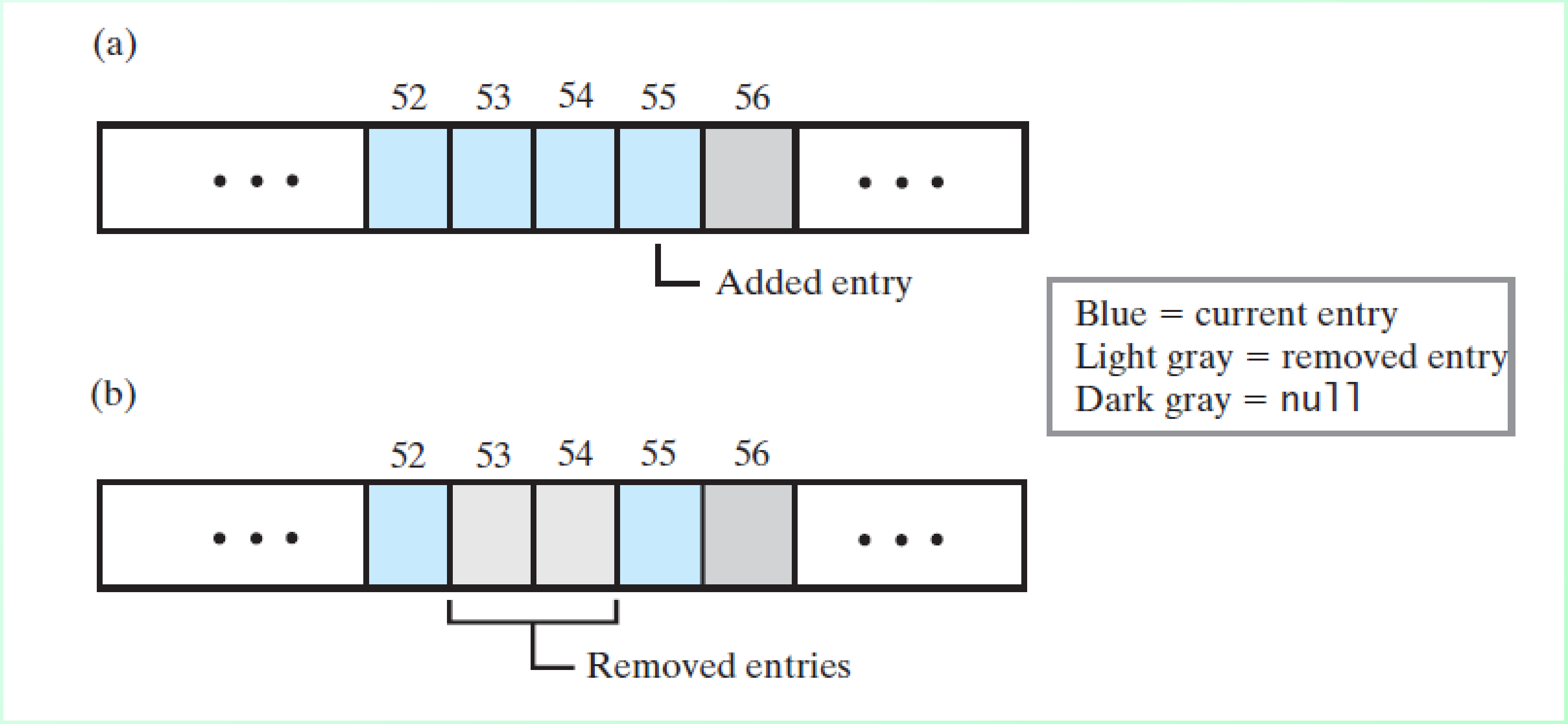


Figure 21-6 A linear probe sequence (a) after adding an entry; (b) after removing two entries;

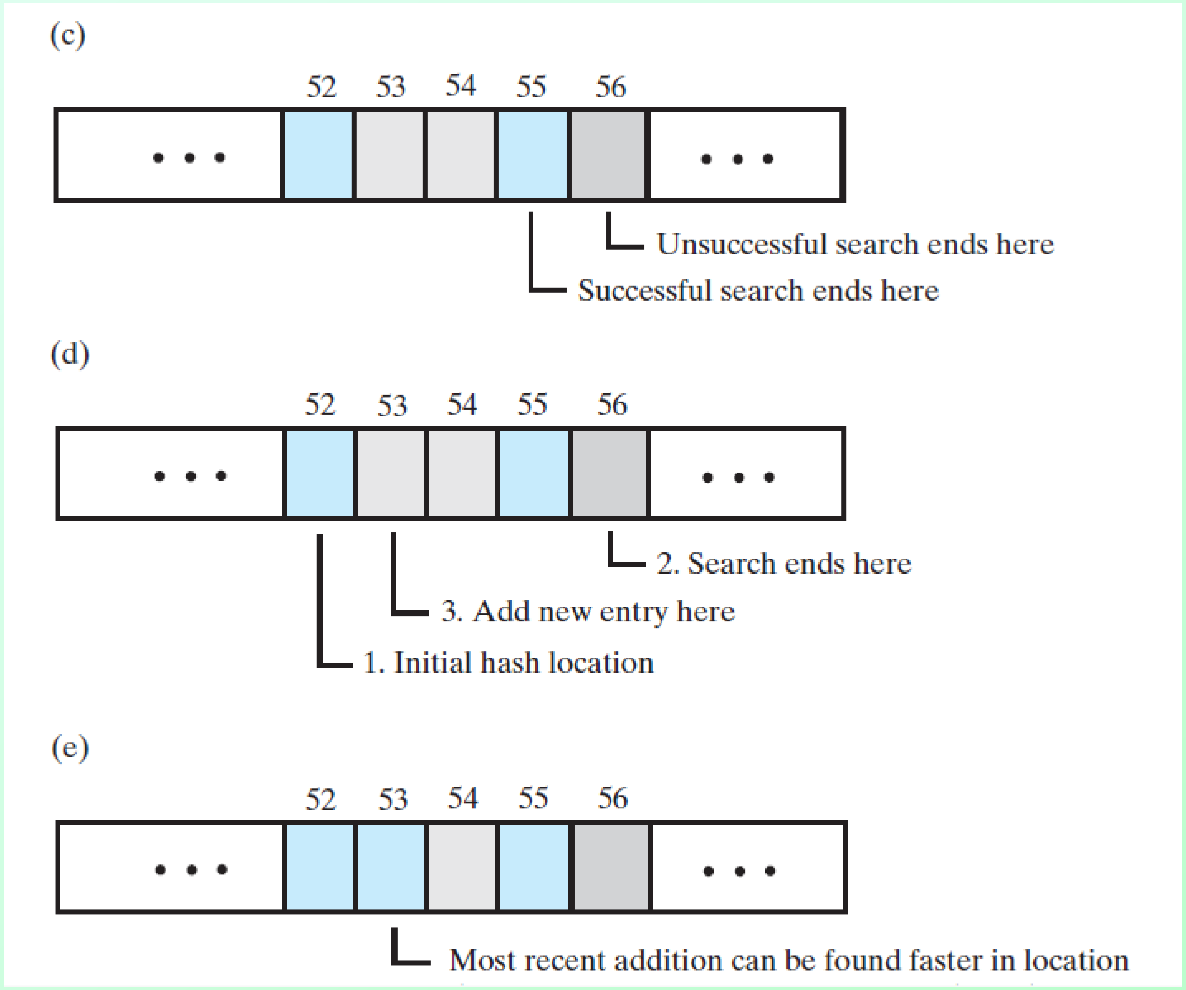


Figure 21-6 A linear probe sequence; (c) after a search; (d) during the search while adding an entry; (e) after an addition to a formerly occupied location

Open Addressing with Quadratic Probing

- Avoid primary clustering by changing the probe sequence
- Alternative to going to location k + 1
  - Go to k + 1, then k + 4, then k + 9
  - In general go to k + j^2 for j = 1, 2, 3, …

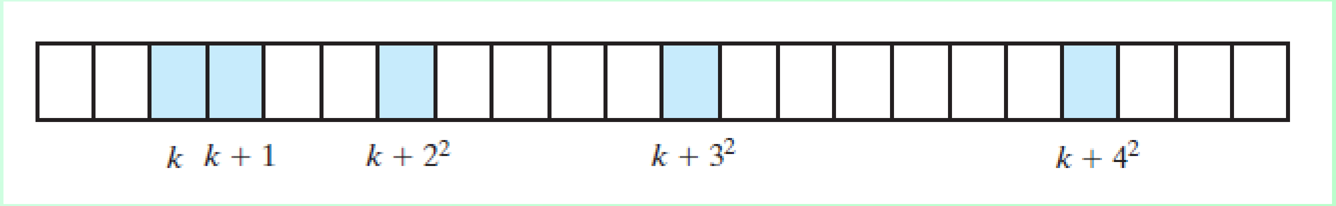


Figure 21-7 A probe sequence of length five using quadratic probing

Open Addressing with Double Hashing

- Use second hash function to compute increments in key-dependent way
- Second has function should reach entire table
- Avoids both primary and secondary clustering

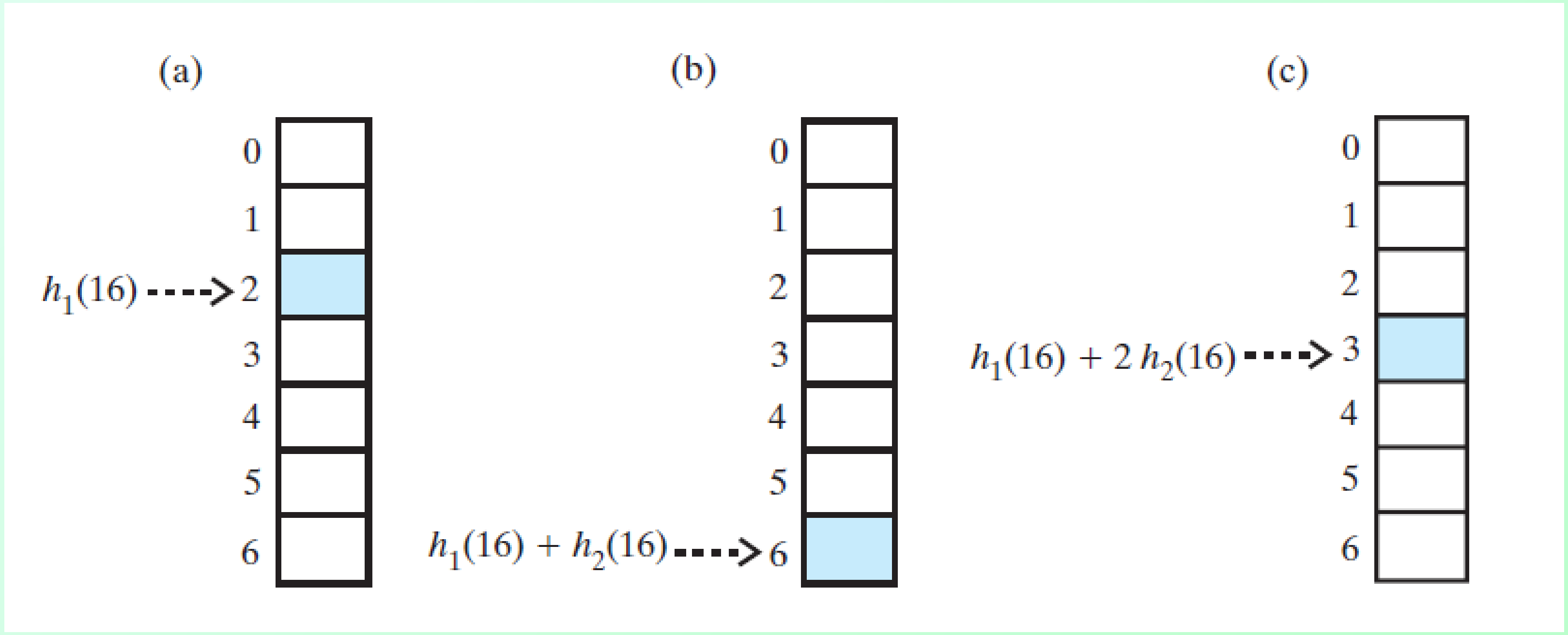


Figure 21-8 The first three locations in a probe sequence generated by double hashing for the search key 16

Question 5 What size hash table should you use with double hashing when the hash functions are

$h_1(\text{key}) = \text{key modulo } 13$

$h_2(\text{key}) = 7 - \text{key modulo } 7$

Why?

Question 6 What probe sequence is defined by the hash functions given in the previous question when the search key is 16?

Question 5 What size hash table should you use with double hashing when the hash functions are

$h_1(key) = key \text{ modulo } 13$

$h_2(key) = 7 - key \text{ modulo } 7$

Why?

13. Since 13 is both prime and the modulo base in h_1, the probe sequence can reach all locations in the table before it repeats.

Question 6 What probe sequence is defined by the hash functions given in the previous question when the search key is 16?

3, 8, 0, 5, 10, 2, 7, 12, 4, 9, 1, 6, 11, 3, ...

Potential Problem with Open Addressing

- Frequent additions and removals
  - Can cause every location in hash table to reference either current entry or former entry
- Could result in unsuccessful search requiring check of every location
- Possible solutions
  - Increase size of hash table (see Ch. 22)
  - Separate chaining

Separate Chaining

- Each location of hash table can represent multiple values
  - Called a “bucket”
- To add, hash to bucket, insert data in first available slot there
- To retrieve, hash to bucket, traverse bucket contents
- To delete, hash to bucket, remove item

Separate Chaining

- Bucket representation
  - List (sorted or not)
  - Chain of linked nodes
  - Array or vector
- Arrays or vectors require extra overhead
- Linked list, chain of linked nodes is reasonable choice

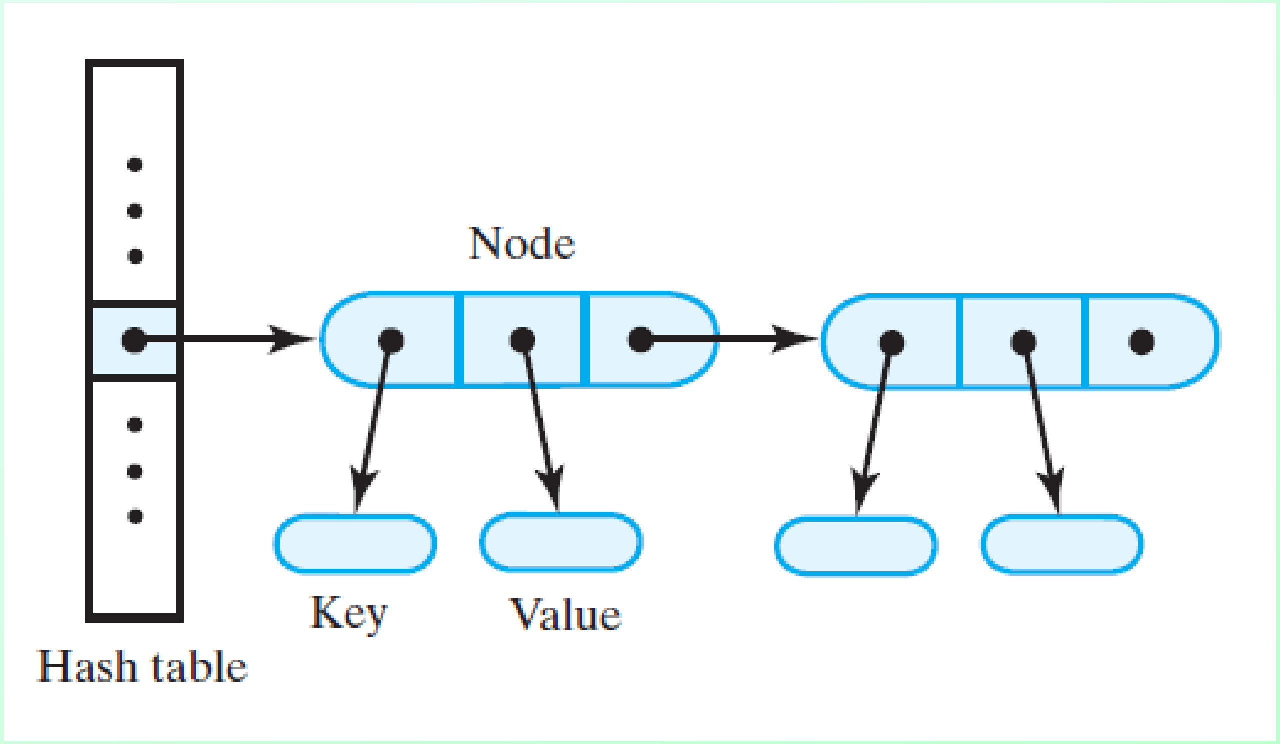


Figure 21-9 A hash table for use with separate chaining; each bucket is a chain of linked nodes

Question 7 Consider search keys that are distinct integers. If the hash function is

h(key) = key modulo 5

and separate chaining resolves collisions, where in the hash table do the following search keys appear after being added? 4, 6, 20, 14, 31, 29

Question 7 Consider search keys that are distinct integers. If the hash function is

h(key) = key modulo 5

and separate chaining resolves collisions, where in the hash table do the following search keys appear after being added? 4, 6, 20, 14, 31, 29

hashTable[0] → 20
hashTable[1] → 6 → 31
hashTable[2] is null
hashTable[3] is null
hashTable[4] → 4 → 14 → 29

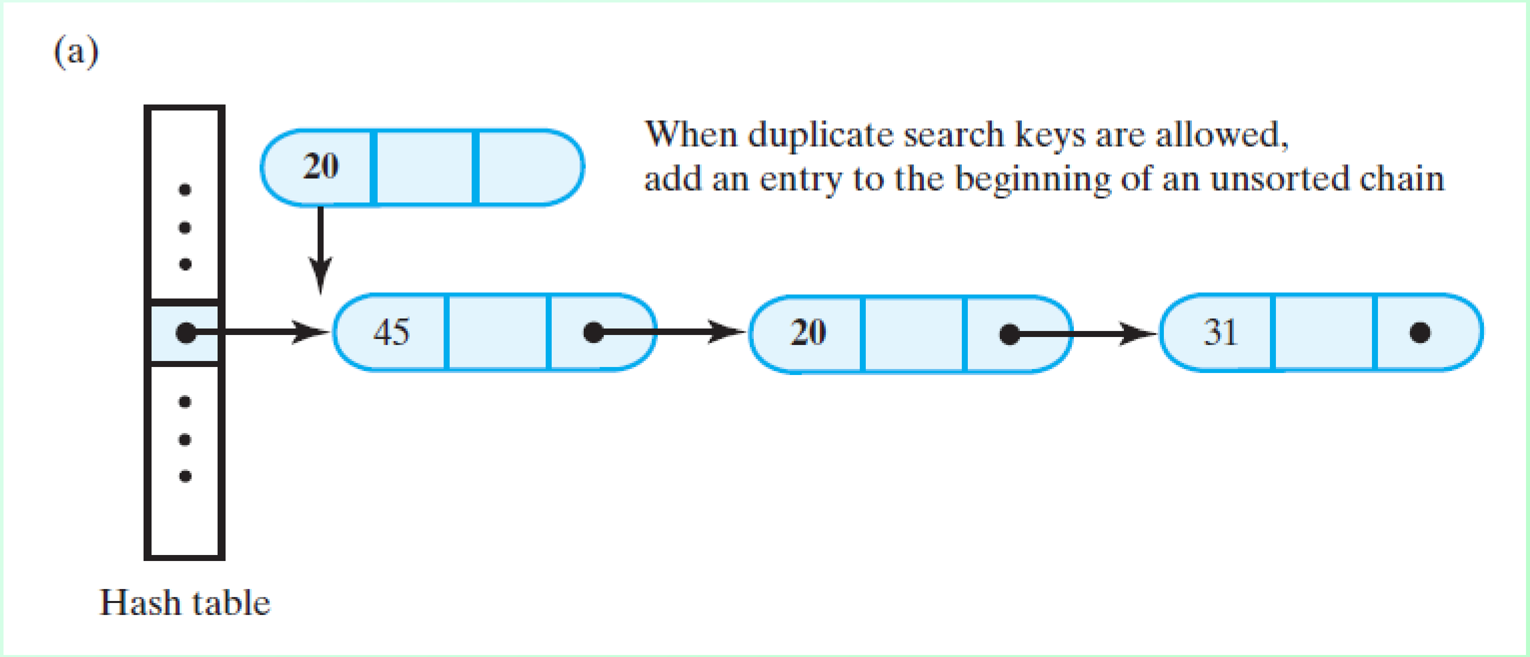

(a)
20
When duplicate search keys are allowed,
add an entry to the beginning of an unsorted chain
45
20
31
Hash table

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

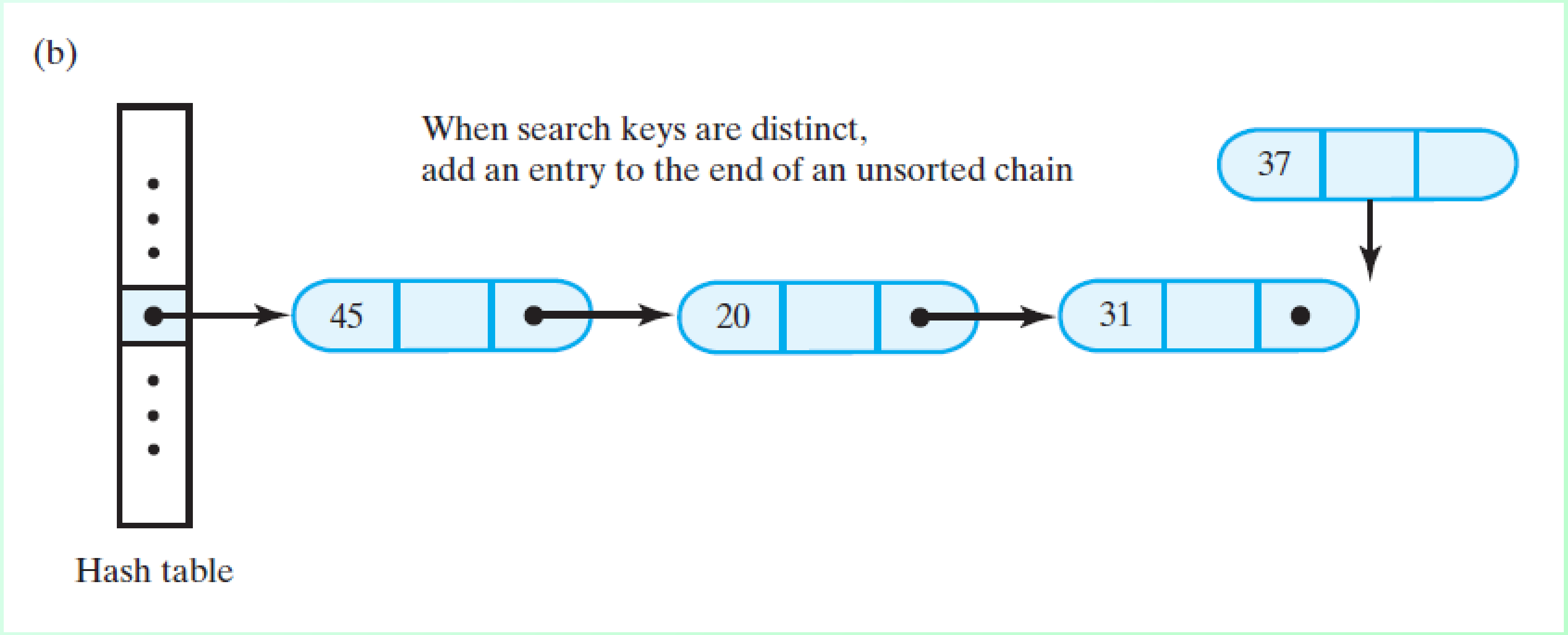

(b)
When search keys are distinct,
add an entry to the end of an unsorted chain
37
45
20
31
Hash table

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

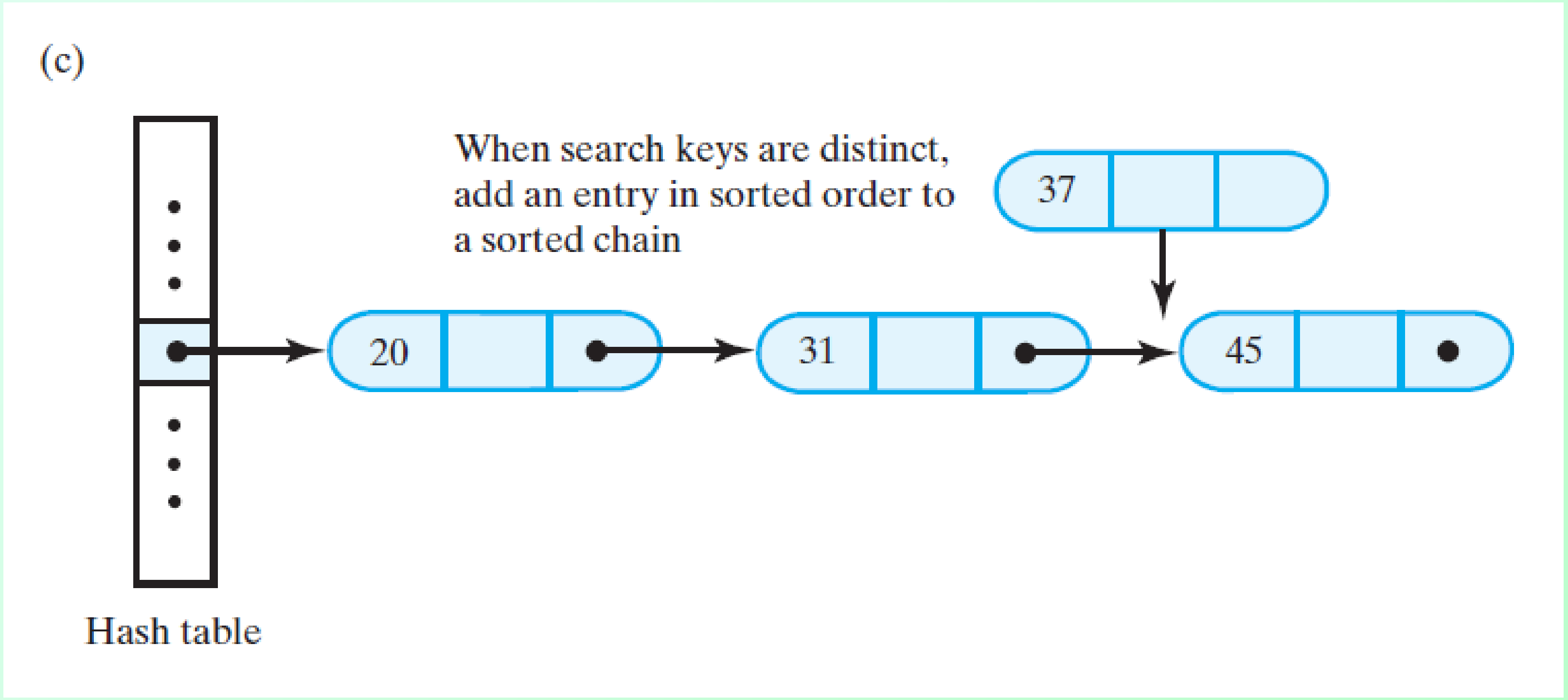

(c)
When search keys are distinct, add an entry in sorted order to a sorted chain
37
20
31
45
Hash table

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

Question 9 Can you define an iteration of a dictionary's search keys in sorted order when you use hashing in its implementation? Explain.

Question 9 Can you define an iteration of a dictionary's search keys in sorted order when you use hashing in its implementation? Explain.

No. Suppose that a, b, c, and d are search keys in sorted order in the hash table. With separate chaining, b and d might appear in one chain while the other keys appear in another. Traversing the chains in order will not visit the keys in sorted order. The same is true of open addressing when traversing the occupied array locations.

End

Chapter 21