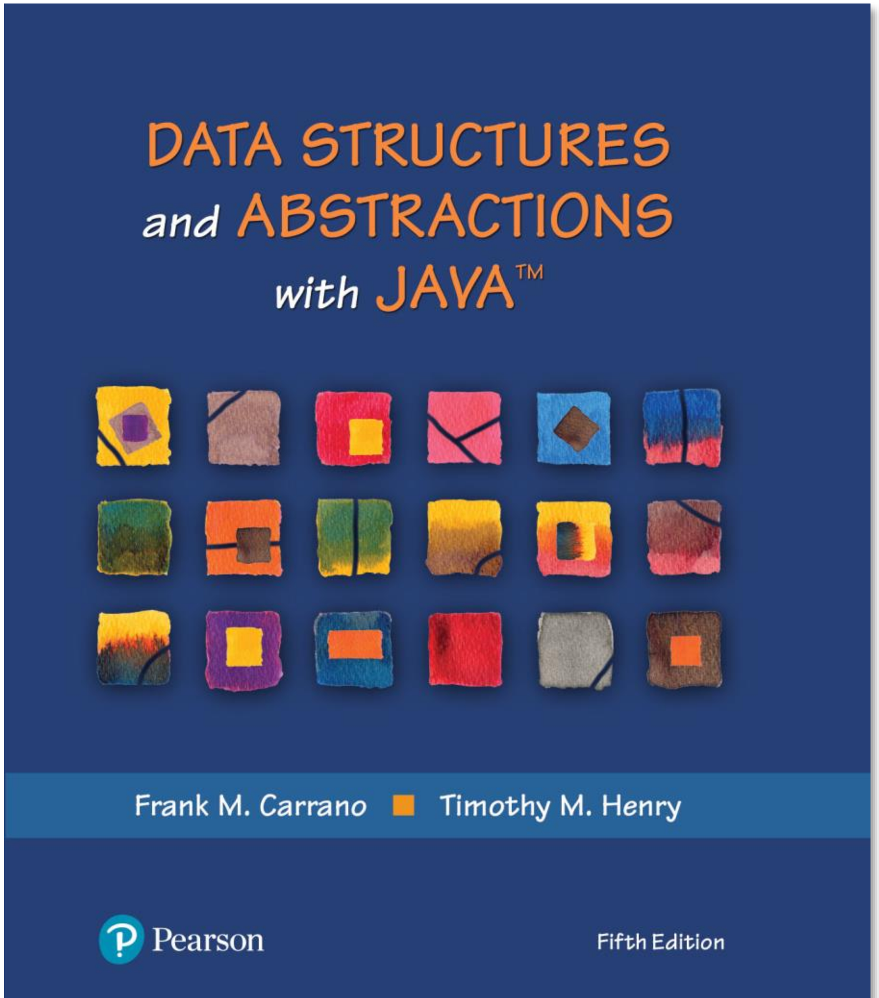
Data Structures and Abstractions with Java™

5th Edition



Chapter 22

Introducing Hashing

Hashing

- A technique that determines an index into a table 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 is mapped, or hashed, to the index

Hash Table

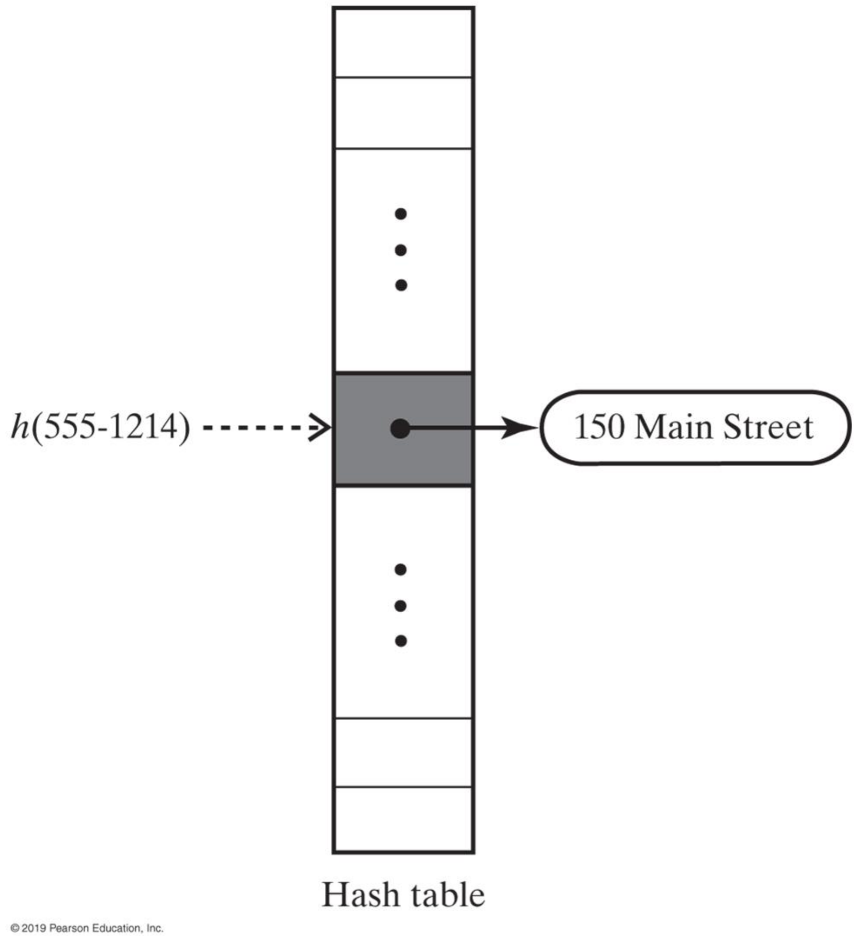


FIGURE 22-1 A hash function indexes its hash table

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:
  - Convert search key to an integer
    - Called the hash code.
  - Compress hash code into the range of indices for hash table.

Algorithm **getHashIndex(phoneNumber)**
// Returns an index to an array of tableSize *elements.*
i = last four digits of phoneNumber
return i % tableSize

Typical Hashing

- Most 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
- Consider **`tableSize = 101`**
- **`getHashIndex(555-1214) = 52`**
- **`getHashIndex(555-8132) = 52`** ***also!!!***

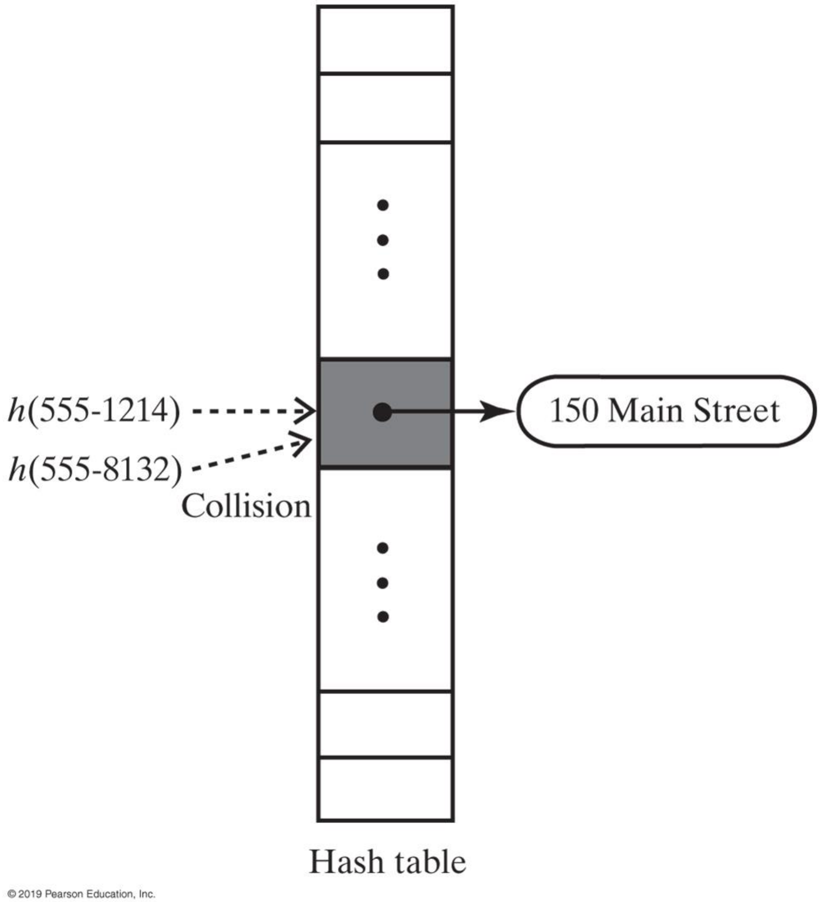


FIGURE 22-2 A collision caused by the hash function *h*

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 values

Computing Hash Codes

- Hash code for a string example:

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

- Java code to do this:

```java
int hash = 0;
int n = s.length();
for (int i = 0; i < n; i++)
   hash = g * hash + s.charAt(i);
```

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 mod 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

- Collision:
  - 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)

Linear Probing

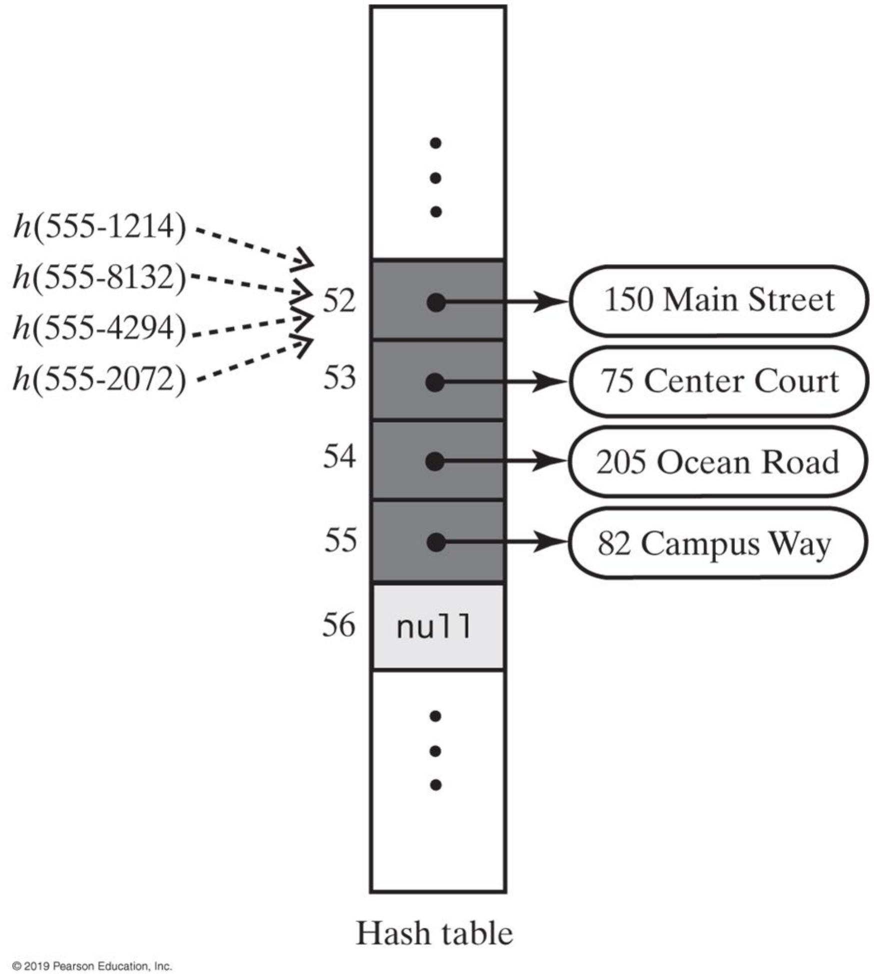


FIGURE 22-3 The effect of linear probing after adding four entries whose search keys hash to the same index

Linear Probing

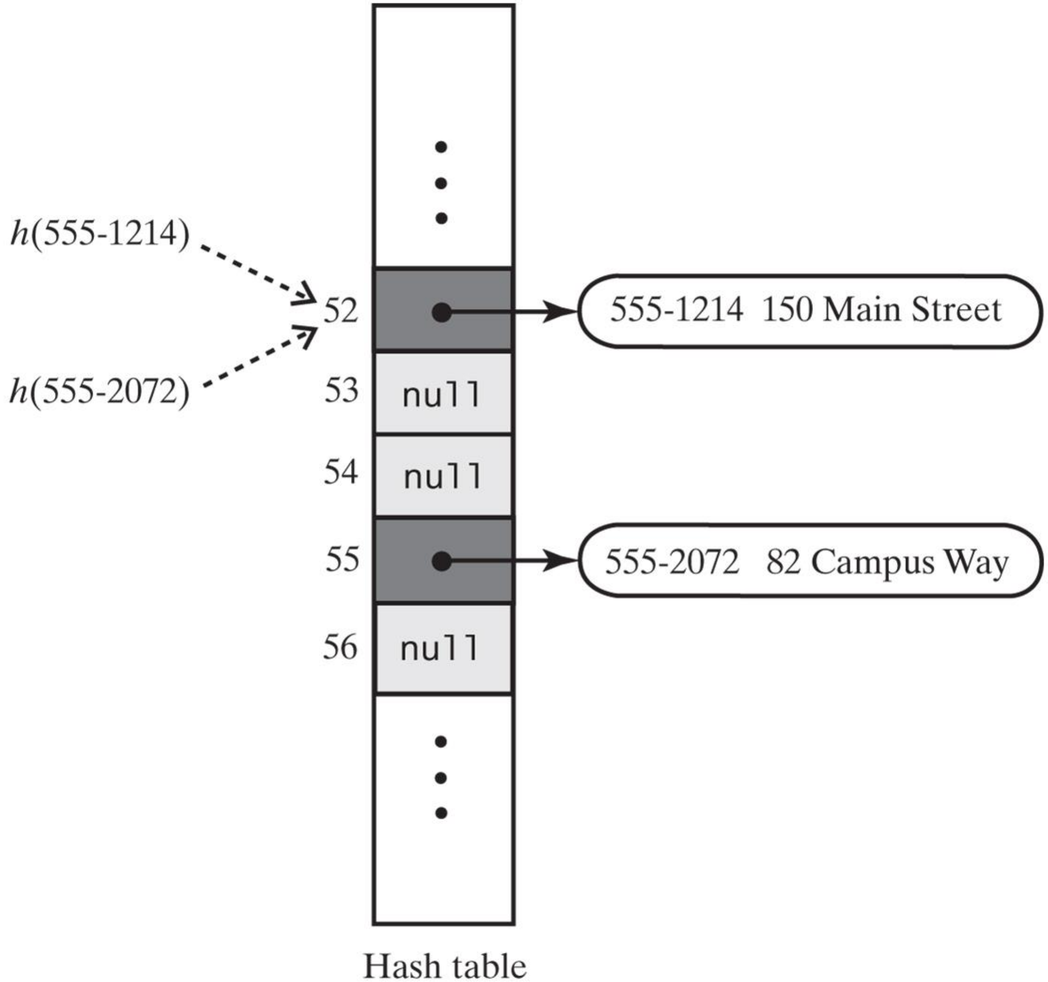


FIGURE 22-5 A hash table if remove used `null` to remove entries

Resolving Collisions

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

Linear Probing

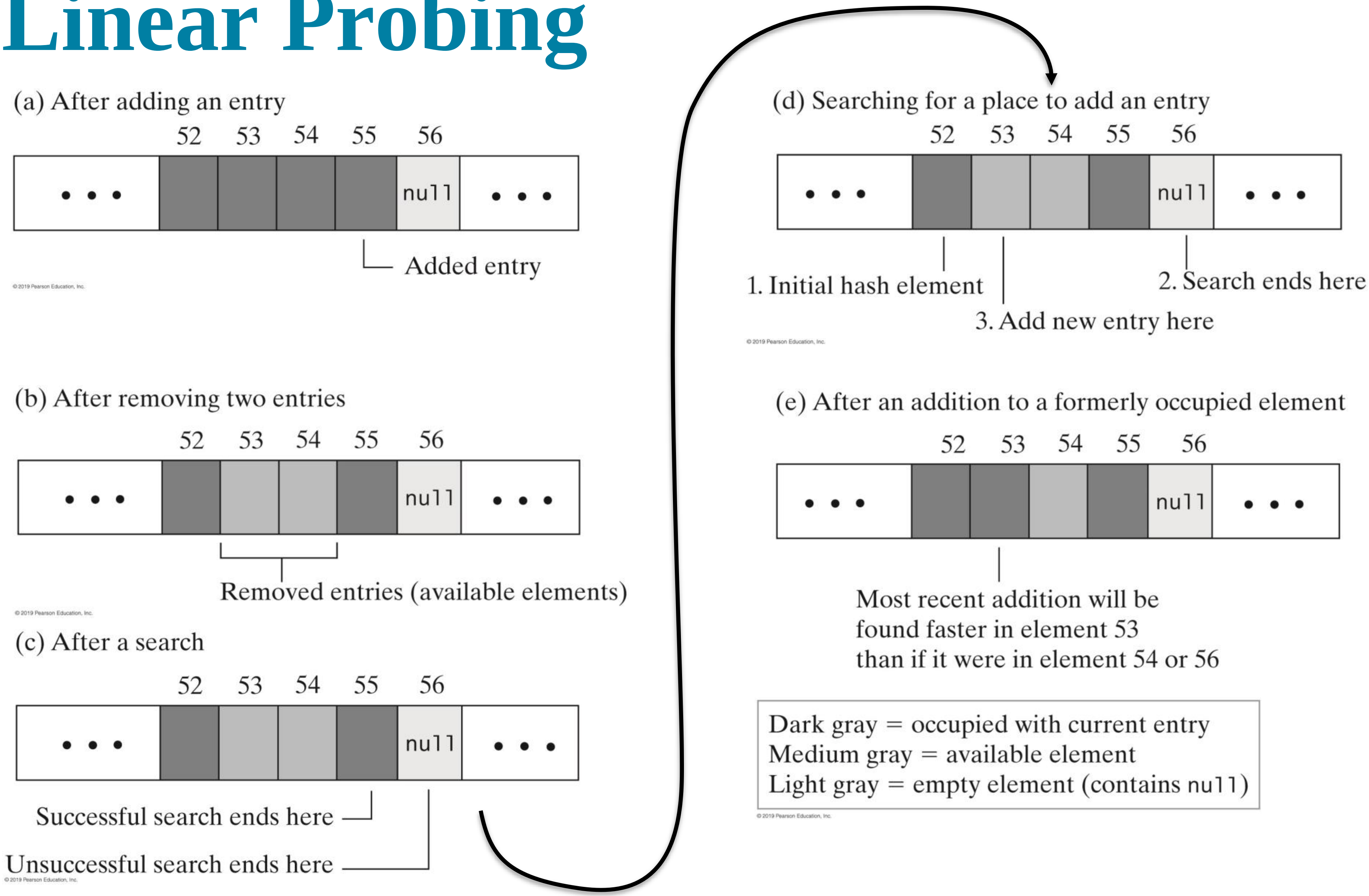


FIGURE 22-6 The linear probe sequence in various situations

Linear Probing - Probe Algorithm

```
Algorithm probe(index, key)
// Searches the probe sequence that begins at index. Returns the index of either the element
// containing key or an available element in the hash table.
while (key is not found and hashTable[index] is not null)
{
	if (hashTable[index] references an entry in the dictionary)
	{
		if (the entry in hashTable[index] contains key)
			Exit loop
		else
			index = next probe index
	}
	else // hashTable[index] is available
	{
		if (this is the first available element encountered)
			availableStateIndex = index
		index = next probe index
	}
}
if (key is found or an available element was not encountered)
	return index
else
	return availableStateIndex // Index of first entry removed
```

Linear Probe Algorithm

```
// Precondition: checkIntegrity has been called.
private int linearProbe(int index, K key)
{
    boolean found = false;
    int availableStateIndex = −1; // Index of first element in available state
    while ( !found && (hashTable[index] != null) )
    {
        if (hashTable[index] != AVAILABLE)
        {
            if  (key.equals(hashTable[index].getKey()))
                found = true; // Key found
            else                                    // Follow probe sequence
                index = (index + 1) % hashTable.length; // Linear probing
        }
        else // Element in available state; skip it, but mark the first one encountered
        {
            // Save index of first element in available state
            if (availableStateIndex == −1) availableStateIndex = index;
                index = (index + 1) % hashTable.length;         // Linear probing
        } // end if
    } // end while
    // Assertion: Either key or null is found at hashTable[index]
    if (found || (availableStateIndex == −1) )
        return index;                                  // Index of either key or null
    else
        return availableStateIndex; // Index of an available element
} // end linearProbe
```

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

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, …`

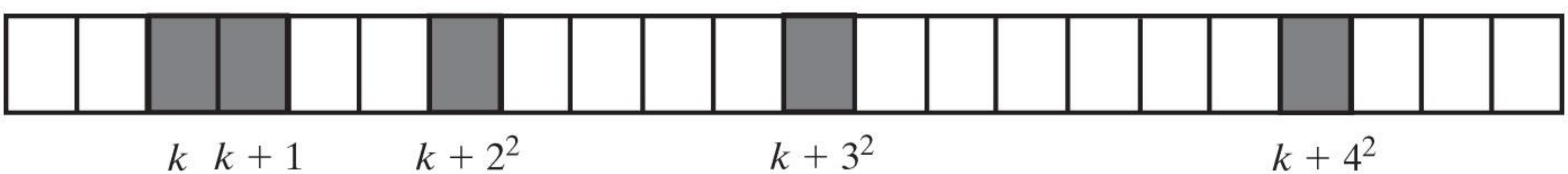


FIGURE 22-7 A probe sequence of length five using quadratic probing

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

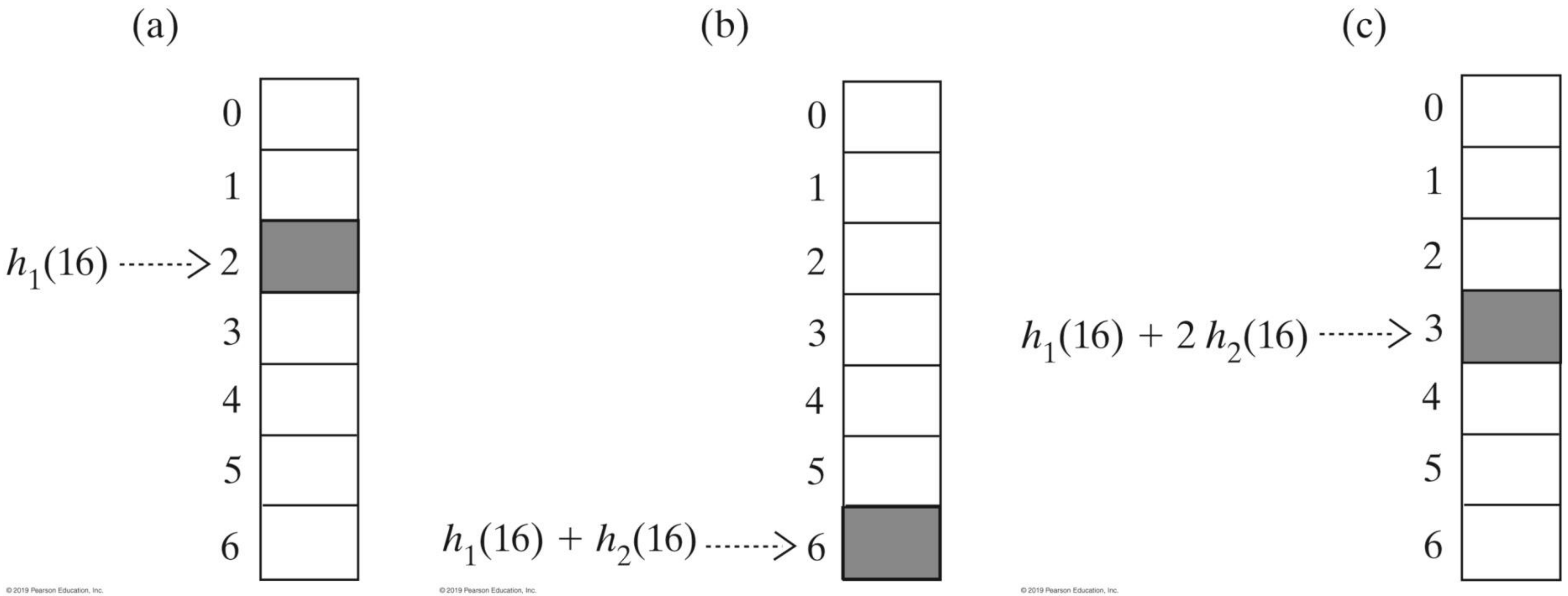


FIGURE 22-8 The first three elements in a probe sequence generated by double hashing for the search key 16

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`

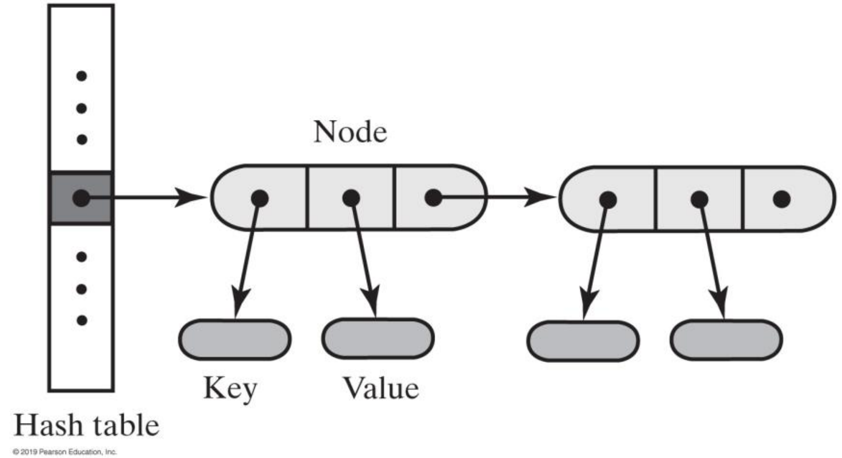


FIGURE 22-9 A hash table for use with separate chaining; each bucket is a chain of linked nodes

Separate Chaining

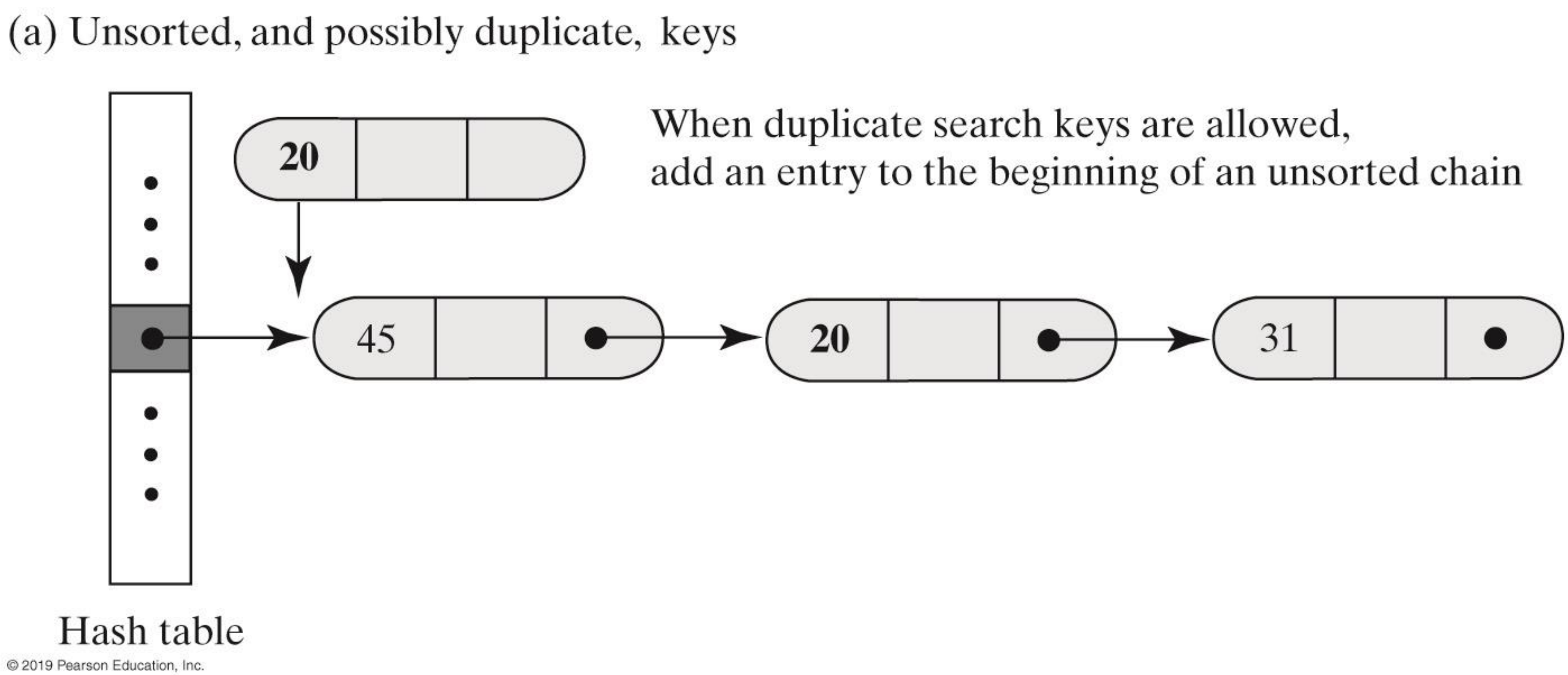


FIGURE 22-10a Inserting a new entry into a linked bucket according to the nature of the integer search keys

Separate Chaining

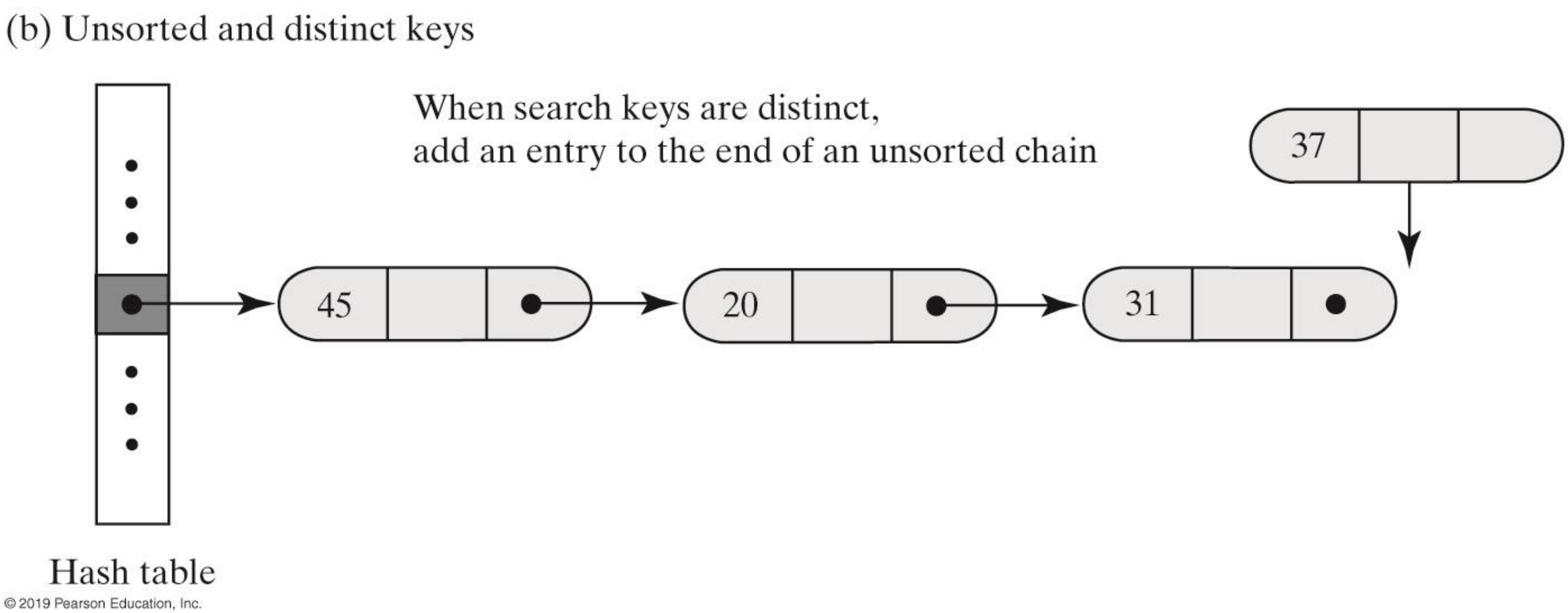


FIGURE 22-10b Inserting a new entry into a linked bucket according to the nature of the integer search keys

Separate Chaining

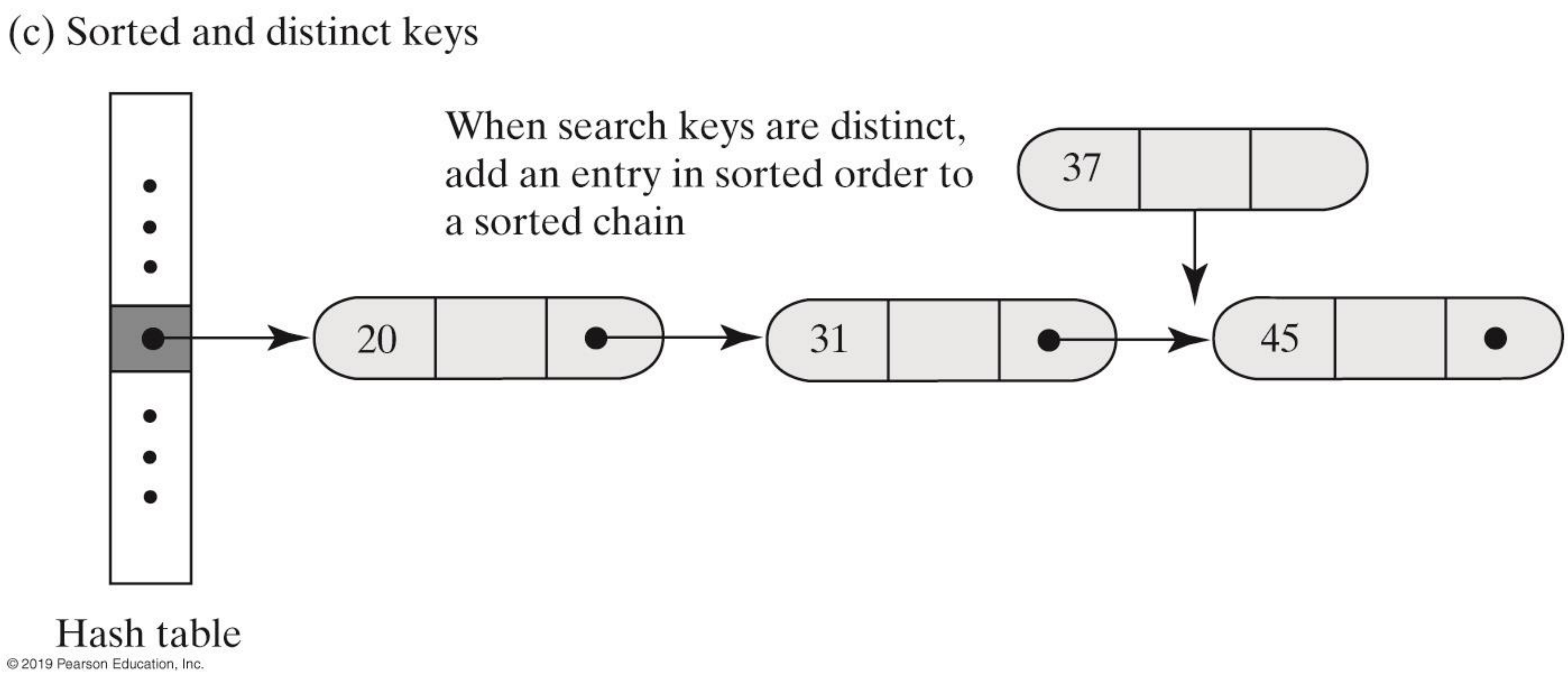


FIGURE 22-10c Inserting a new entry into a linked bucket according to the nature of the integer search keys

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 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 for the dictionary's `remove` method.

Separate Chaining

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 `getValue` method.

End

Chapter 22