

# Dictionaries and Their Implementations

## Chapter 18

# The ADT Dictionary

<u>City</u>	<u>Country</u>	<u>Population</u>
Buenos Aires	Argentina	13,639,000
Cairo	Egypt	17,816,000
Johannesburg	South Africa	7,618,000
London	England	8,586,000
Madrid	Spain	5,427,000
Mexico City	Mexico	19,463,000
Mumbai	India	16,910,000
New York City	U.S.A.	20,464,000
Paris	France	10,755,000
Sydney	Australia	3,785,000
Tokyo	Japan	37,126,000
Toronto	Canada	6,139,000

FIGURE 18-1 A collection of data about certain cities



# The ADT Dictionary

- Consider need to search such a collection for
  - Name
  - Address
- Criterion chosen for search is *search key*
- The ADT dictionary uses a search key to identify its entries

# The ADT Dictionary

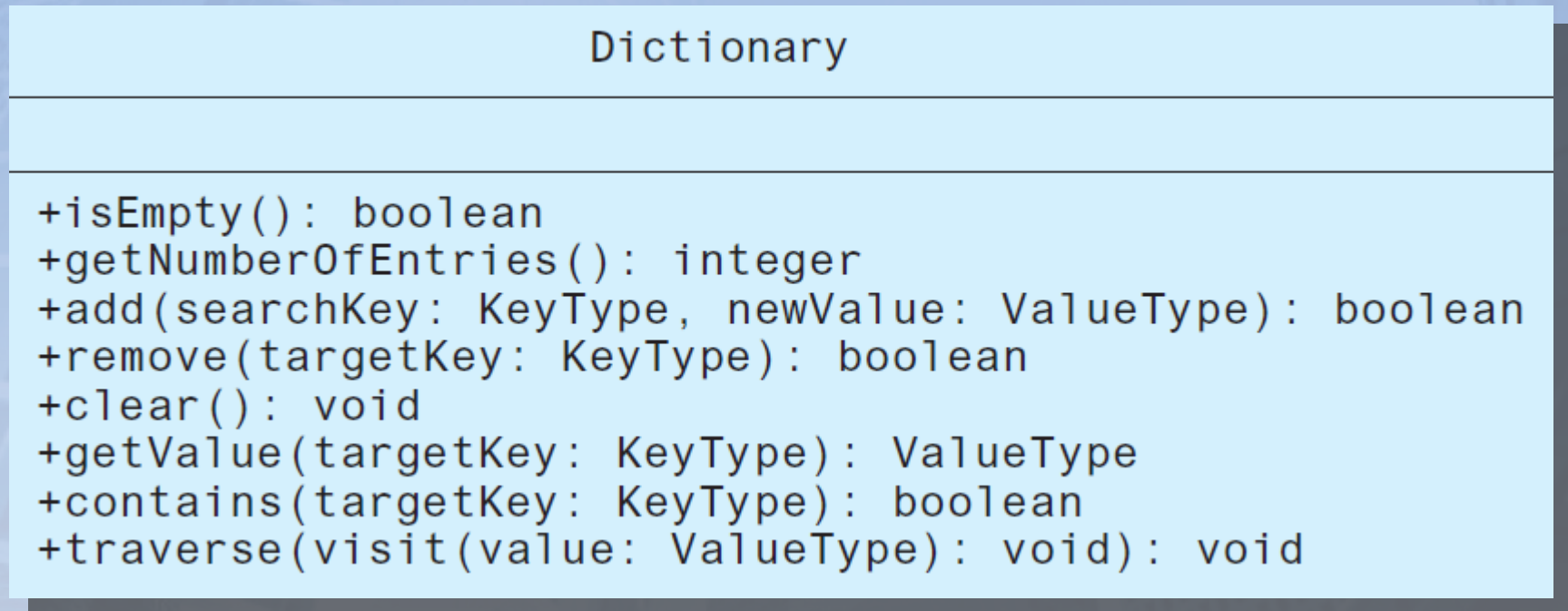


FIGURE 18-2 UML diagram for a class of dictionaries



# Possible Implementations

- Categories of linear implementations
  - Sorted by search key array-based
  - Sorted by search key link-based
  - Unsorted array-based
  - Unsorted link-based

Search key	Data Value
------------	------------

FIGURE 18-3 A dictionary entry

# Possible Implementations

```
1  /** An interface for the ADT dictionary.
2   @file DictionaryInterface.h */
3
4  #ifndef DICTIONARY_INTERFACE_
5  #define DICTIONARY_INTERFACE_
6
7  #include "NotFoundException.h"
8
9  template<class KeyType, class ValueType>
10 class DictionaryInterface
11 {
12 public:
13     /** Sees whether this dictionary is empty.
14     @return True if the dictionary is empty;
15         otherwise returns false. */
16     virtual bool isEmpty() const = 0;
17
18     /** Gets the number of entries in this dictionary.
19     @return The number of entries in the dictionary. */
20     virtual int getNumberOfEntries() const = 0;
```

LISTING 18-2 A header file for a class of dictionary entries

# Possible Implementations

```
21
22     /** Adds a new search key and associated value to this dictionary.
23     @pre  The new search key differs from all search keys presently
24           in the dictionary.
25     @post If the addition is successful, the new key-value pair is in its
26           proper position within the dictionary.
27     @param searchKey  The search key associated with the value to be added.
28     @param newValue   The value to be added.
29     @return True if the entry was successfully added, or false if not. */
30     virtual bool add(const KeyType& searchKey, const ValueType& newValue) = 0;
31
32     /** Removes a key-value pair from this dictionary.
33     @post If the entry whose search key equals searchKey existed in the
34           dictionary, the entry was removed.
35     @param searchKey  The search key of the entry to be removed.
36     @return True if the entry was successfully removed, or false if not. */
37     virtual bool remove(const KeyType& searchKey) = 0;
38
39     /** Removes all entries from this dictionary. */
40     virtual void clear() = 0;
```

LISTING 18-2 A header file for a class of dictionary entries



# Possible Implementations

```
40 virtual void clear() = 0;
41
42 /** Retrieves the value in this dictionary whose search key is given
43     @post If the retrieval is successful, the value is returned.
44     @param searchKey The search key of the value to be retrieved.
45     @return The value associated with the search key.
46     @throw NotFoundException if the key-value pair does not exist. */
47 virtual ValueType getValue(const KeyType& searchKey) const
48     throw (NotFoundException) = 0;
49
50 /** Sees whether this dictionary contains an entry with a given search key.
51     @post The dictionary is unchanged.
52     @param searchKey The given search key.
53     @return True if an entry with the given search key exists in the
54         dictionary. */
55 virtual bool contains(const KeyType& searchKey) const = 0;
56
```

LISTING 18-2 A header file for a class of dictionary entries



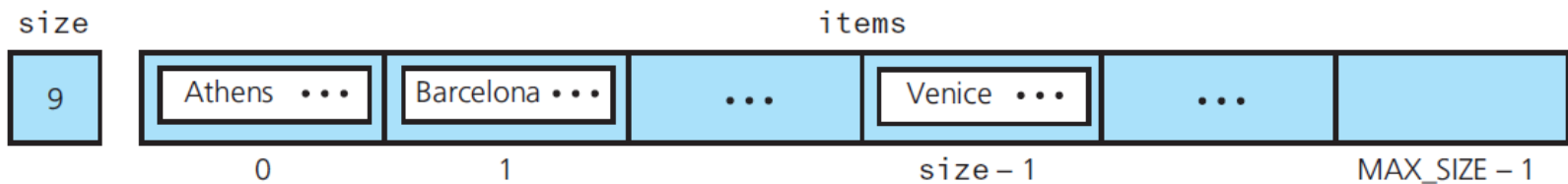
# Possible Implementations

```
55 virtual bool contains(const key_typed searchKey) const = 0;
56
57 /** Traverses this dictionary and calls a given client function once
58     for each entry.
59     @post The given function's action occurs once for each entry in the
60         dictionary and possibly alters the entry.
61     @param visit A client function. */
62 virtual void traverse(void visit(ValueType&)) const = 0;
63 /** Destroys this dictionary and frees its assigned memory. */
64
65 virtual ~DictionaryInterface(){ }
66 }; // end DictionaryInterface
67 #endif
```

LISTING 18-2 A header file for a class of dictionary entries

# Possible Implementations

(a) Array based



(b) Link based



Figure 18-4 Data members for two sorted linear implementations of the ADT dictionary for the data in Figure 18-1

# Possible Implementations

```
1  /** A class of entries to add to an array-based implementation of the
2      ADT dictionary.
3      @file Entry.h */
4
5  #ifndef ENTRY_
6  #define ENTRY_
7
8  template <class KeyType, class ValueType>
9  class Entry
10 {
11 private:
12     KeyType key;
13     ValueType value;
14
15 protected:
16     void setKey(const KeyType& searchKey);
17
```

Listing 18-2 A header file for a class of dictionary entries



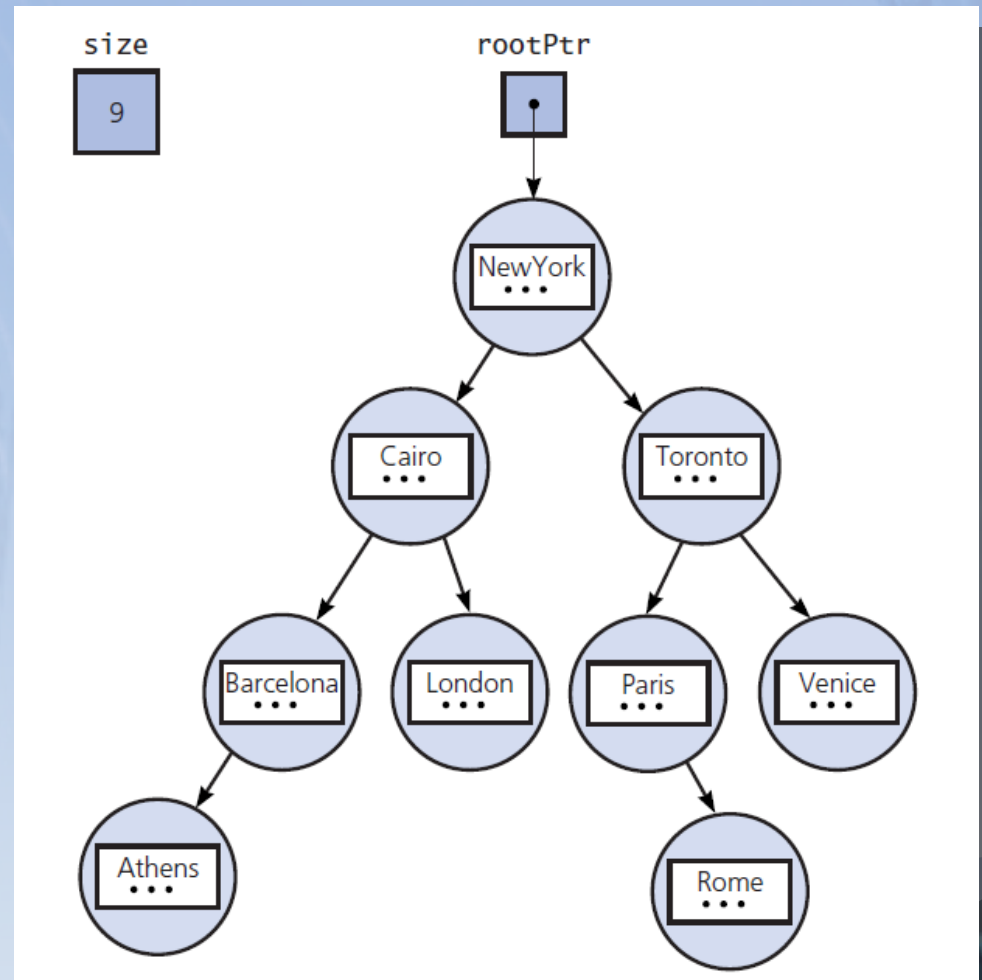
# Possible Implementations

```
16     void setKey(const KeyType& searchKey);
17
18 public:
19     Entry();
20     Entry(const KeyType& searchKey, const ValueType& newValue);
21     ValueType getValue() const;
22     KeyType getKey() const;
23     void setValue(const ValueType& newValue);
24
25     bool operator==(const Entry<KeyType, ValueType>& rightHandValue) const;
26     bool operator>(const Entry<KeyType, ValueType>& rightHandValue) const;
27 }; // end Entry
28 #include "Entry.cpp"
29 #endif
```

Listing 18-2 A header file for a class of dictionary entries

# Possible Implementations

FIGURE 18-5 The data members for a binary search tree implementation of the ADT dictionary for the data in Figure 18-1



# Sorted Array-Based Implementation of ADT Dictionary

```
25
26 public:
27     ArrayDictionary();
28     ArrayDictionary(int maxNumberOfEntries);
29     ArrayDictionary(const ArrayDictionary<KeyType, ValueType>& dictionary);
30
31     virtual ~ArrayDictionary();
32
33     bool isEmpty() const;
34     int getNumberOfEntries() const;
35     bool add(const KeyType& searchKey, const ValueType& newValue) throw(PrecondViolatedExcept);
36     bool remove(const KeyType& searchKey);
37     void clear();
38     ValueType getValue(const KeyType& searchKey) const throw(NotFoundException);
39     bool contains(const KeyType& searchKey) const;
40
41     /** Traverses the entries in this dictionary in sorted search-key order
42         and calls a given client function once for the value in each entry. */
43     void traverse(void visit(ValueType&)) const;
44 }; // end ArrayDictionary
45 #include "ArrayDictionary.cpp"
46 #endif
```

LISTING 18-3 A header file for the class `ArrayDictionary`



# Binary Search Tree

## Implementation of the ADT Dictionary

```
1  /** A binary search tree implementation of the ADT dictionary
2      that organizes its data in sorted search-key order.
3      Search keys in the dictionary are unique.
4      @file TreeDictionary.h */
5
6  #ifndef TREE_DICTIONARY_
7  #define TREE_DICTIONARY_
8
9  #include "DictionaryInterface.h"
10 #include "BinarySearchTree.h"
11 #include "Entry.h"
12 #include "NotFoundException.h"
13 #include "PrecondViolatedExcept.h"
14
15 template <class KeyType, class ValueType>
16 class TreeDictionary : public DictionaryInterface<KeyType, ValueType>
17 {
18 private:
```

LISTING 18-4 A header file for the class `TreeDictionary`

# Binary Search Tree

## Implementation of the ADT Dictionary

```
17 {
18     private:
19         // Binary search tree of dictionary entries
20         BinarySearchTree<Entry<KeyType, ValueType> > entryTree;
21
22     public:
23         TreeDictionary();
24         TreeDictionary(const TreeDictionary<KeyType, ValueType>& dictionary);
25
26         virtual ~TreeDictionary();
27
28         // The declarations of the public methods appear here and are the
29         // same as given in Listing 18-3 for the class ArrayDictionary.
30         ...
31     }; // end TreeDictionary
32     #include "TreeDictionary.cpp"
33     #endif
}
```

LISTING 18-4 A header file for the class **TreeDictionary**



# Binary Search Tree

## Implementation of the ADT Dictionary

```
template < class KeyType, class ValueType>
bool TreeDictionary<KeyType, ValueType>::add(const KeyType& searchKey,
                                             const ValueType& newValue)
                                             throw(PrecondViolatedExcept)
{
    Entry<KeyType, ValueType> newEntry(searchKey, newValue);

    // Enforce precondition: Ensure distinct search keys
    if (!itemTree.contains(newEntry))
    {
        // Add new entry and return boolean result
        return itemTree.add(Entry<KeyType, ValueType>(searchKey, newValue));
    }
    else
    {
        auto message = "Attempt to add an entry whose search key exists in dictionary.";
        throw(PrecondViolatedExcept(message)); // Exit the method
    } // end if
} // end add
```

Method **add** which prevents duplicate keys.



# Selecting an Implementation

- Linear implementations
  - Perspective
  - Efficiency
  - Motivation
- Consider
  - What operations are needed
  - How often each operation is required

# Three Scenarios

A. Addition and traversal in no particular order

- Unsorted order is efficient
- Array-based versus pointer-based

B. Retrieval

- Sorted array-based can use binary search
- Binary search impractical for link-based
- Max size of dictionary affects choice



# Three Scenarios

## C. Addition, removal, retrieval, traversal in sorted order

- Add and remove need to find position, then add or remove from that position
- Array-based best for find, link-based best for addition/removal

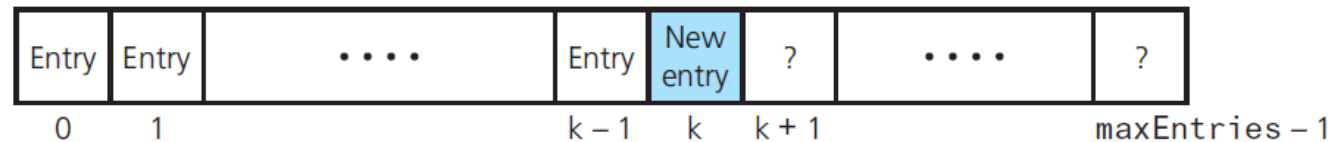


# Three Scenarios

(a) Array based

entryCount

k + 1



(b) Link based

entryCount headPtr

k + 1

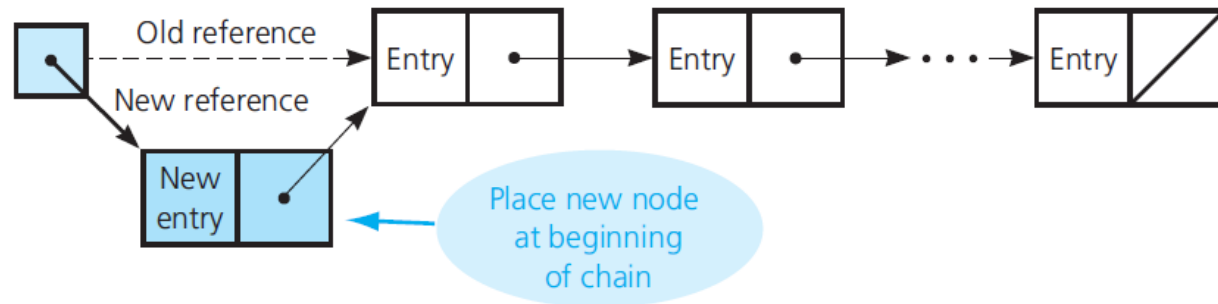
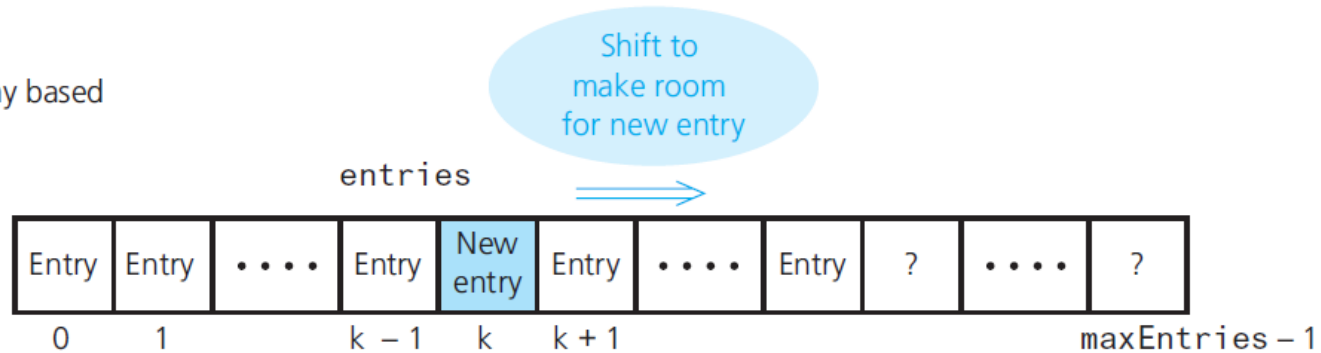


FIGURE 18-6 Addition for unsorted linear implementations

# Three Scenarios

(a) Array based



(b) Link based

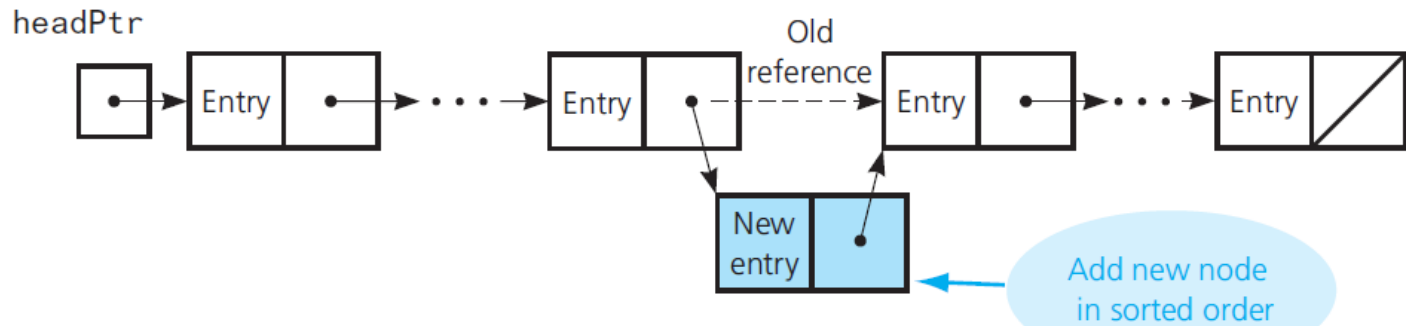


FIGURE 18-7 Addition for sorted linear implementations

# Three Scenarios

	<u>Addition</u>	<u>Removal</u>	<u>Retrieval</u>	<u>Traversal</u>
Unsorted array-based	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Unsorted link-based	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Sorted array-based	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$
Sorted link-based	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Binary search tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

FIGURE 18-8 The average-case order of the ADT dictionary operations for various implementations



# Hashing as a Dictionary Implementation

- Situations occur for which search-tree implementations are not adequate.
- Consider a method which acts as an “address calculator” which determines an array index
  - Used for **add**, **getValue**, **remove** operations
- Called a hash function
  - Tells where to place item in a hash table

# Hashing as a Dictionary Implementation

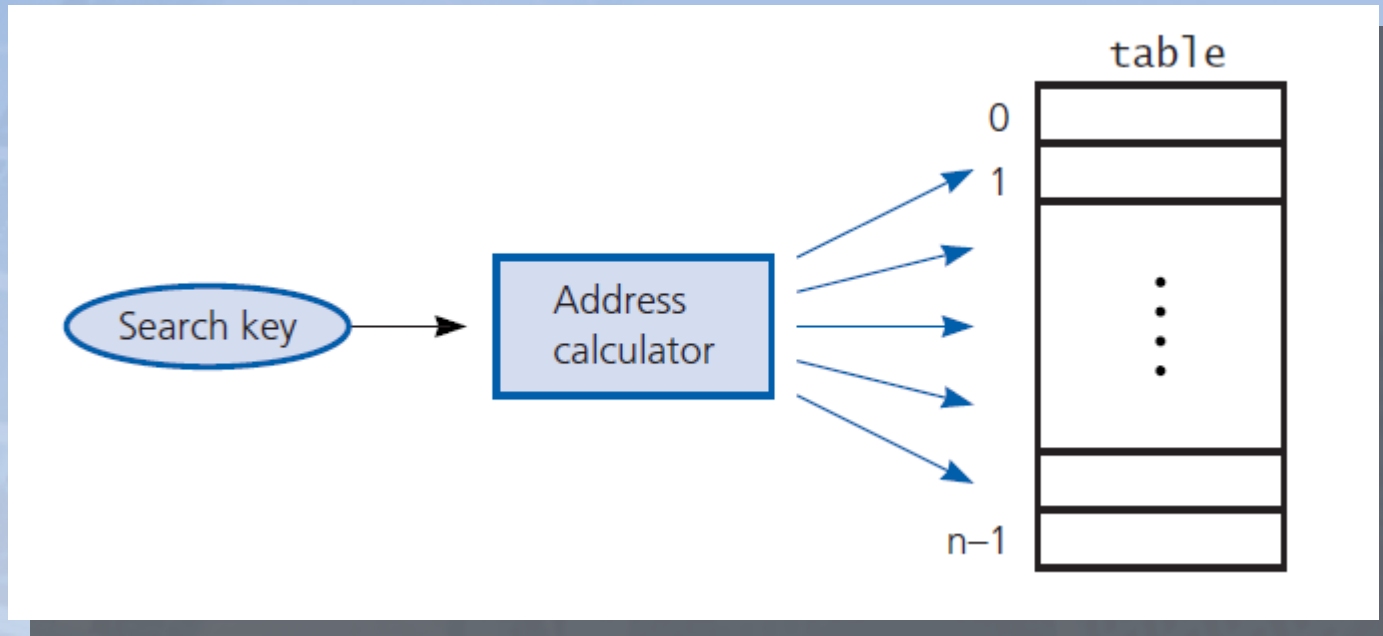


FIGURE 18-9 Address calculator



# Hashing as a Dictionary Implementation

- Perfect hash function
  - Maps each search key into a unique location of the hash table
  - Possible if you know all the search keys
- Collision occurs when hash function maps more than one entry into same array location
- Hash function should
  - Be easy, fast to compute
  - Place entries evenly throughout hash table



# Hash Functions

- Sufficient for hash functions to operate on integers – examples:
  - Select digits from an ID number
  - Folding – add digits, sum is the table location
  - Modulo arithmetic  $h(x) = x \bmod \textit{tableSize}$
  - Convert character string to an integer – use ASCII values

# Resolving Collisions with Open Addressing

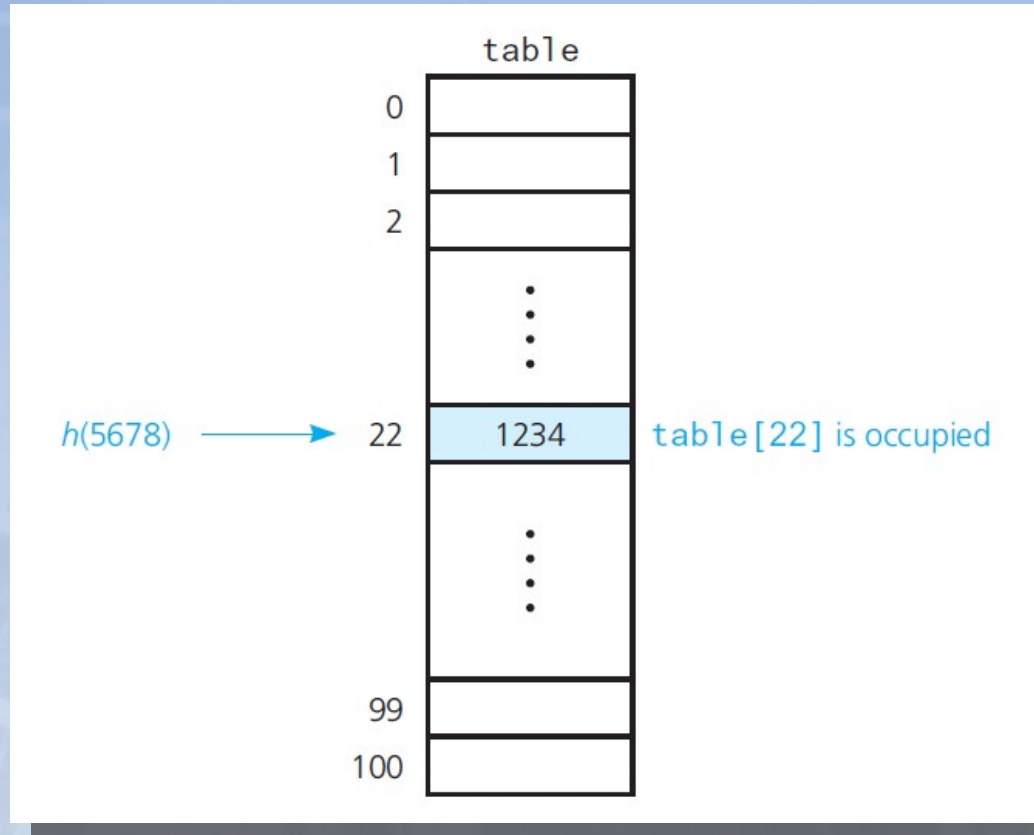


FIGURE 18-10 A collision



# Resolving Collisions with Open Addressing

- Approach 1: Open addressing
  - Linear probing
  - Quadratic probing
  - Double hashing
  - Increase size of hash table



# Resolving Collisions with Open Addressing

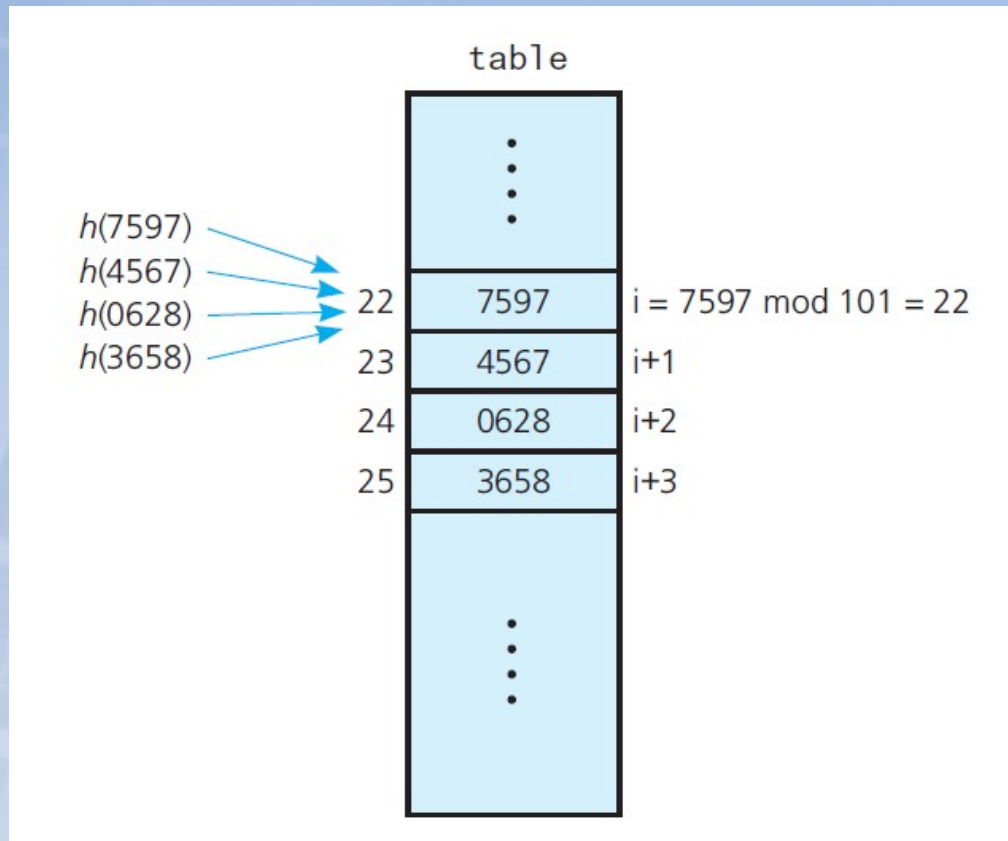


FIGURE 18-11 Linear probing with  $h(x) = x \bmod 101$

# Resolving Collisions with Open Addressing

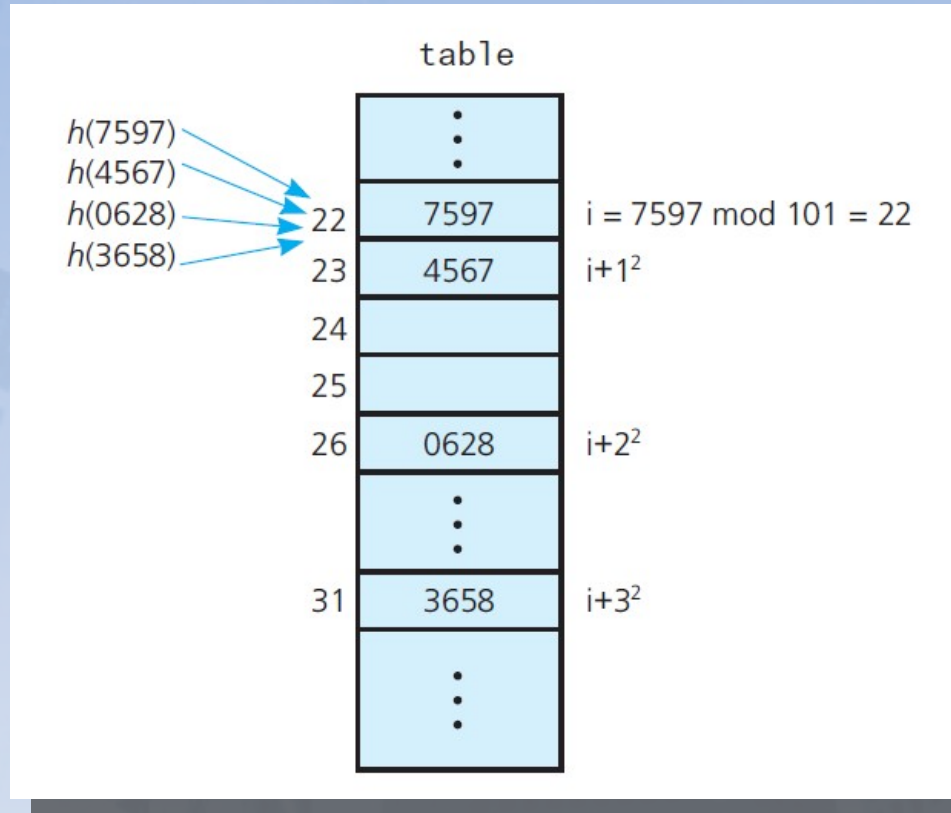


FIGURE 18-12 Quadratic probing with  $h(x) = x \bmod 101$

# Resolving Collisions with Open Addressing

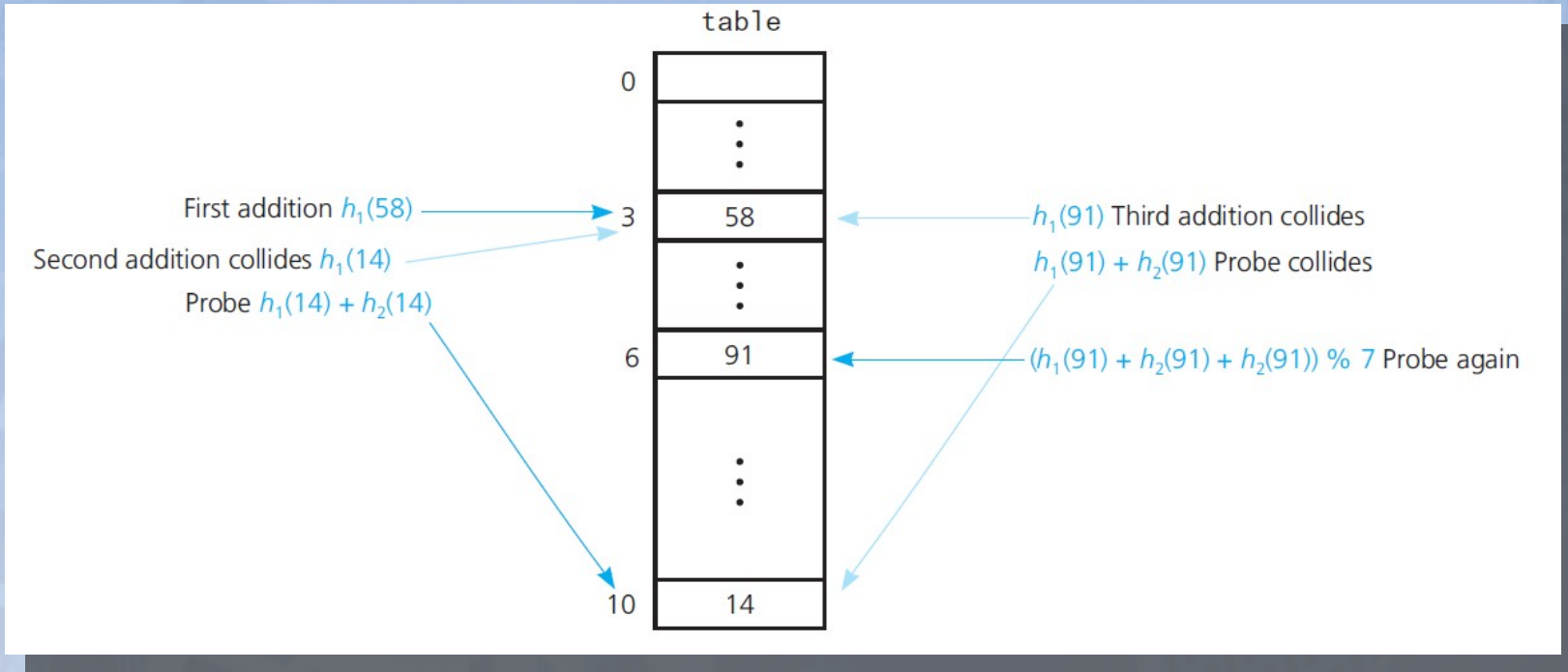


FIGURE 18-13 Double hashing during the addition of 58, 14, and 91



# Resolving Collisions with Open Addressing

- Approach 2: Resolving collisions by restructuring the hash table
  - Buckets
  - Separate chaining

# Resolving Collisions with Open Addressing

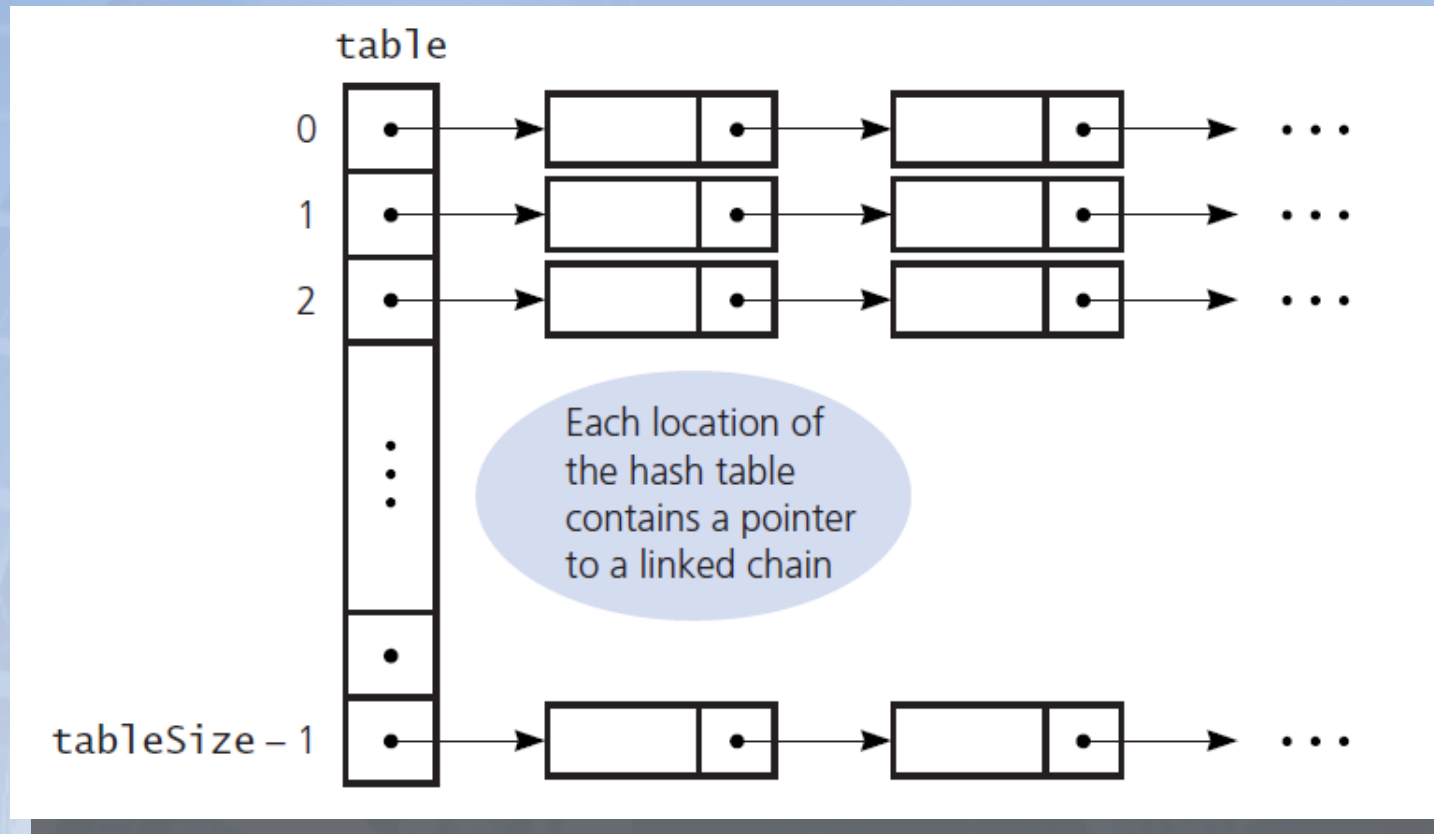


FIGURE 18-14 Separate chaining

# The Efficiency of Hashing

- Load factor measures how full a hash table is

$$\alpha = \frac{\text{Current number of table entries}}{\text{tableSize}}$$

- Unsuccessful searches
  - Generally require more time than successful
- Do not let the hash table get too full



# The Efficiency of Hashing

- Linear probing – average number of comparisons

$$\frac{1}{2} \left[ 1 + \frac{1}{1 - \alpha} \right] \quad \text{for a successful search, and}$$

$$\frac{1}{2} \left[ 1 + \frac{1}{(1 - \alpha)^2} \right] \quad \text{for an unsuccessful search}$$

# The Efficiency of Hashing

- Quadratic probing and double hashing – average number of comparisons

$$\frac{-\log_e(1 - \alpha)}{\alpha}$$

for a successful search, and

$$\frac{1}{1 - \alpha}$$

for an unsuccessful search

# The Efficiency of Hashing

- Efficiency of the retrieval and removal operations under the separate-chaining approach

$$1 + \frac{\alpha}{2} \quad \text{for a successful search, and}$$
$$\alpha \quad \text{for an unsuccessful search}$$



# The Efficiency of Hashing

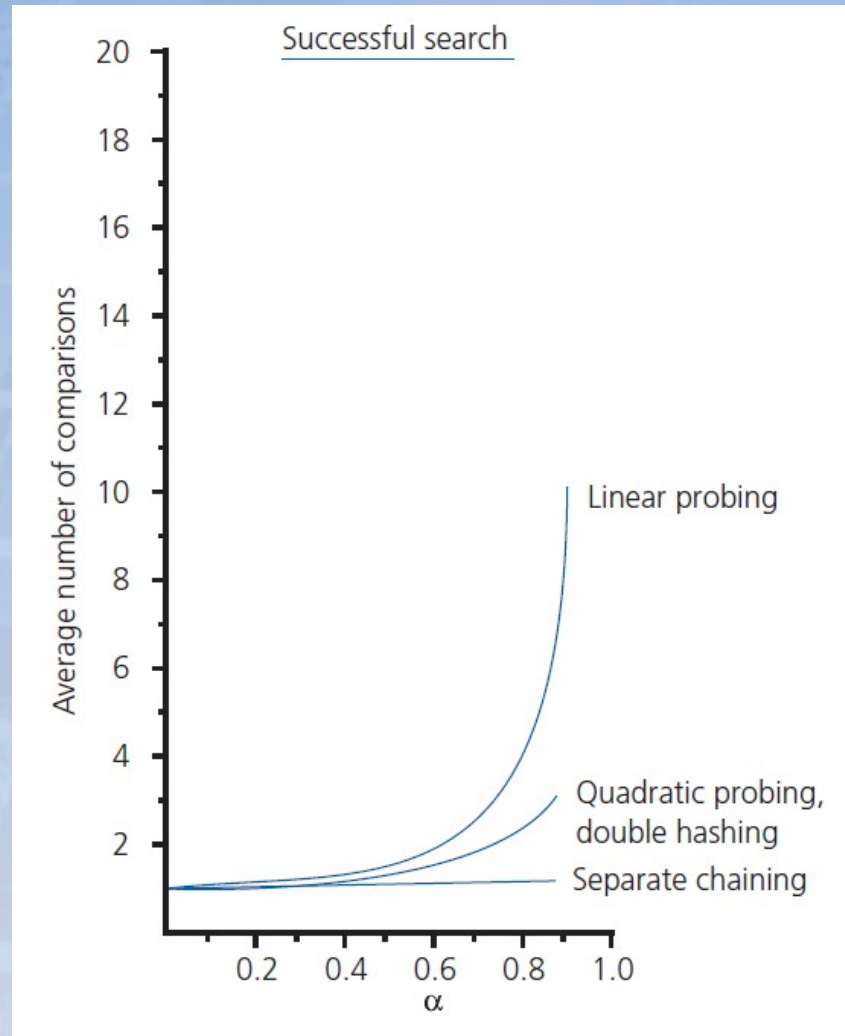


FIGURE 18-15 The relative efficiency of four collision-resolution methods

# The Efficiency of Hashing

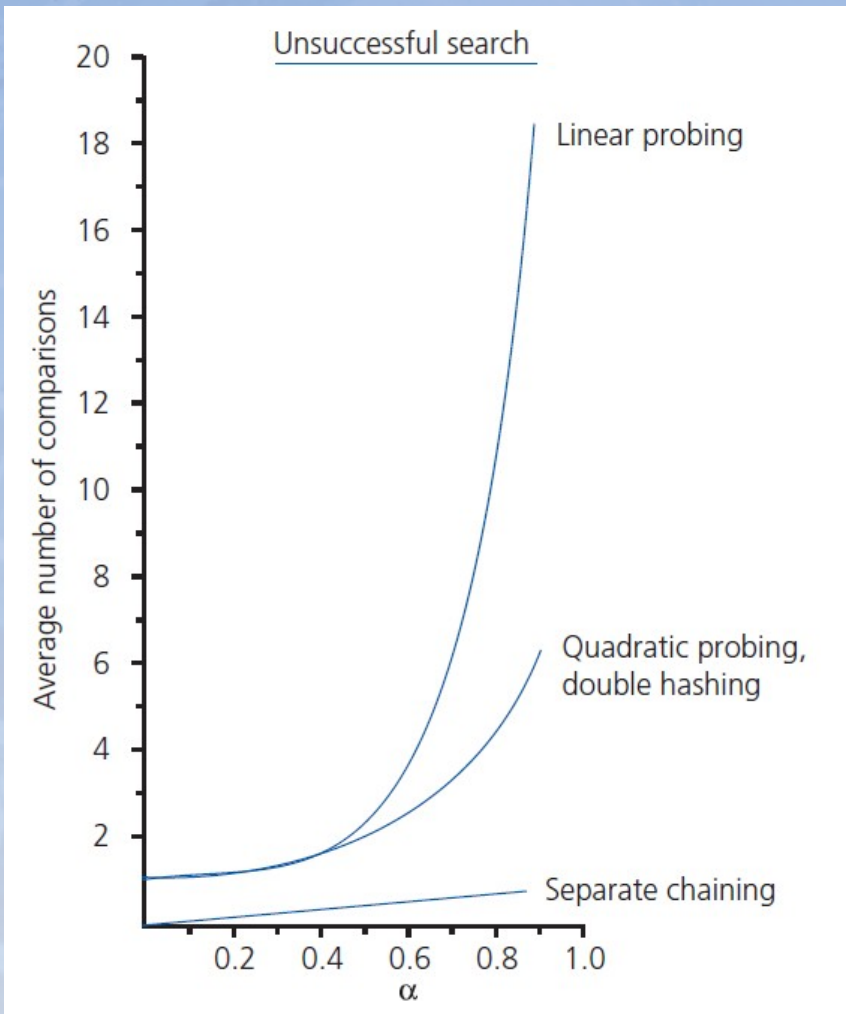


FIGURE 18-15 The relative efficiency of four collision-resolution methods

# What Constitutes a Good Hash Function?

- Is hash function easy and fast to compute?
- Does hash function scatter data evenly throughout hash table?
- How well does hash function scatter random data?
- How well does hash function scatter *non-random* data?



# Dictionary Traversal: An Inefficient Operation Under Hashing

- Entries hashed into `table[i]` and `table[i+1]` have no ordering relationship
- Hashing does not support well traversing a dictionary in sorted order
  - Generally better to use a search tree
- In external storage possible to see
  - Hashing implementation of `getValue`
  - And search-tree for ordered operations simultaneously

# Using Hashing, Separate Chaining to Implement ADT Dictionary



FIGURE 18-16 A dictionary entry when separate chaining is used

# Using Hashing, Separate Chaining to Implement ADT Dictionary

```
1  /** A class of entry objects for a hashing implementation of the
2      ADT dictionary.
3      @file HashedEntry.h */
4
5  #ifndef HASHED_ENTRY_
6  #define HASHED_ENTRY_
7
8  #include "Entry.h"
9
10 template<class KeyType, class ValueType>
11 class HashedEntry : public Entry<KeyType, ValueType>
12 {
13 private:
14     std::shared_ptr<HashedEntry<KeyType, ValueType>> nextPtr;
15 public:
```

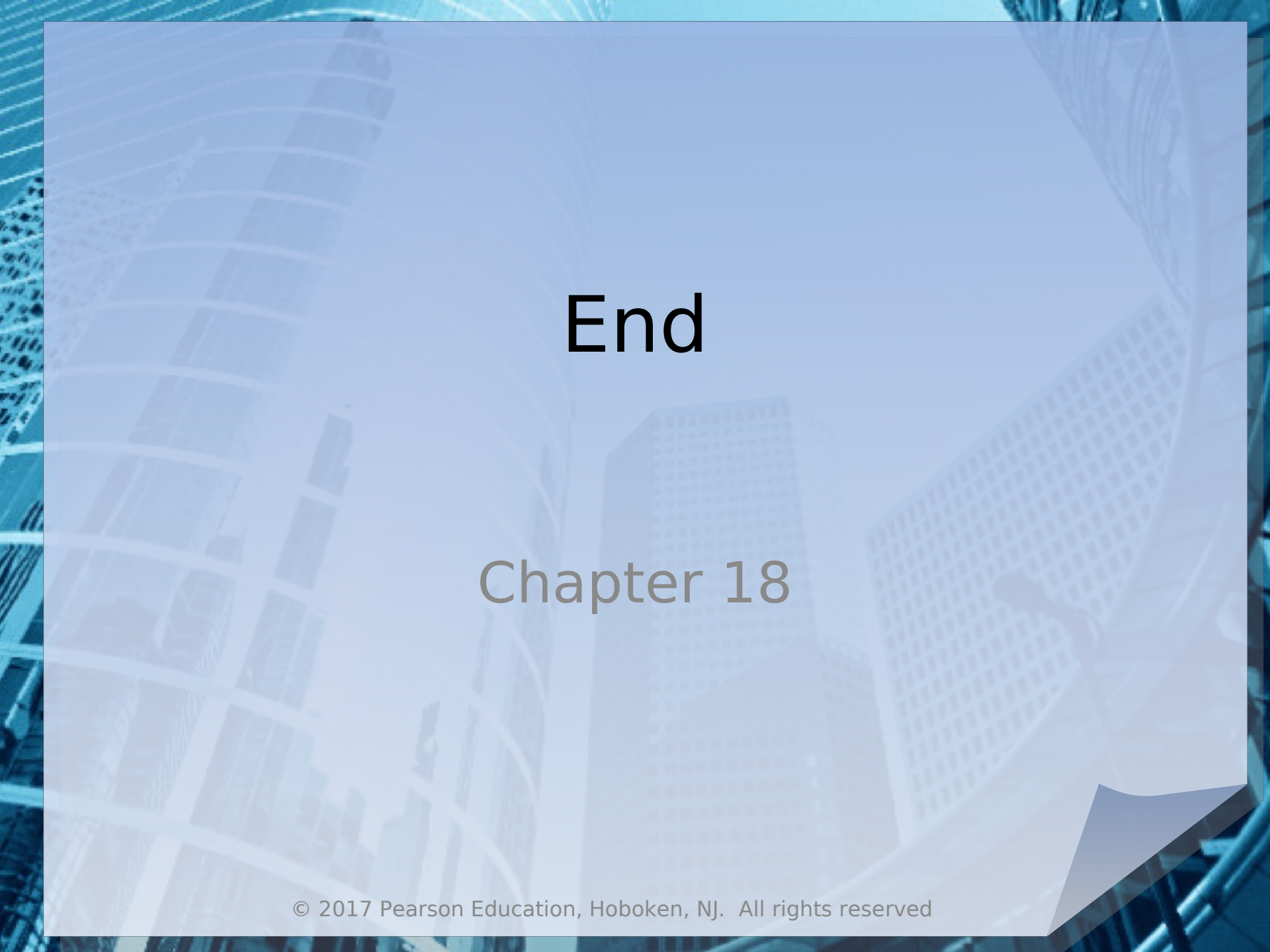
LISTING 18-5 The class `HashedEntry`



# Using Hashing, Separate Chaining to Implement ADT Dictionary

```
14     std::shared_ptr<HashedEntry<KeyType, ValueType>> nextPtr;  
15 public:  
16     HashedEntry();  
17     HashedEntry(KeyType searchKey, ValueType newValue);  
18     HashedEntry(KeyType searchKey, ValueType newValue,  
19                 std::shared_ptr<HashedEntry<KeyType, ValueType>> nextEntryPtr);  
20  
21     void setNext(std::shared_ptr<HashedEntry<KeyType, ValueType>>  
22                 nextEntryPtr nextEntryPtr);  
23     auto getNext() const;  
24 }; // end HashedEntry  
25  
26 #include "HashedEntry.cpp"  
27 #endif
```

LISTING 18-5 The class **HashedEntry**



# End

## Chapter 18