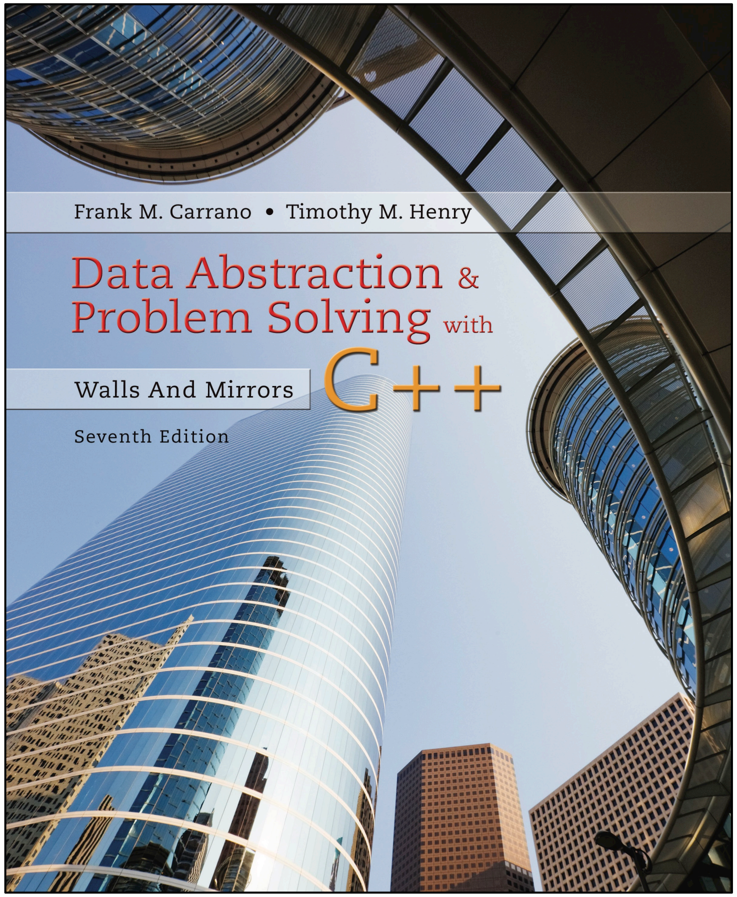
Data Abstraction & Problem Solving with C++: Walls and Mirrors

Seventh Edition



Chapter 18

Dictionaries and Their Implementations

The ADT Dictionary (1 of 3)

Figure 18-1 A collection of data about certain cities

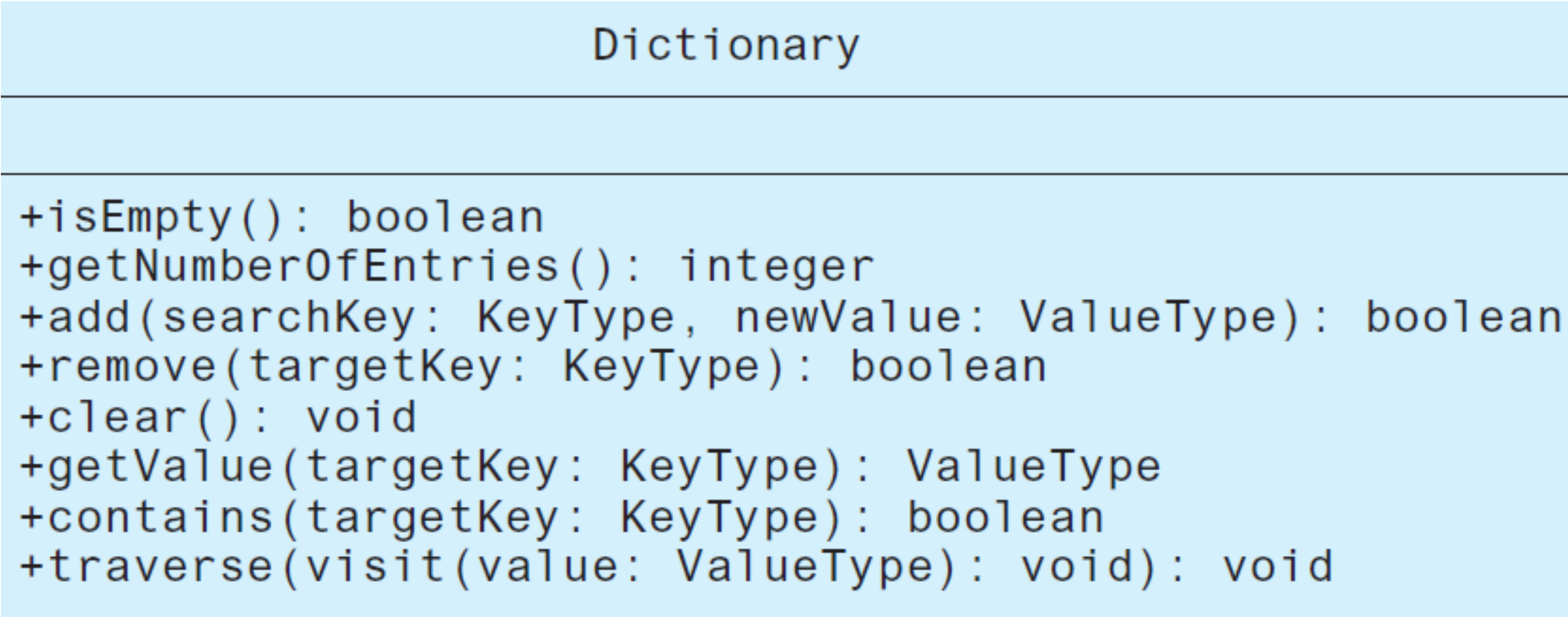
City	Country	Population
Buenos Aires	Argentina	13,639,000
Cairo	Egypt	17,816,000
Johannesburg	South Africa	7,618,000
London	England	8,586,000
Madrid	Spain	5,427,000
Mexico City	Mexico	19,463,000
Mumbai	India	16,910,000
New York City	U.S.A.	20,464,000
Paris	France	10,755,000
Sydney	Australia	3,785,000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

The ADT Dictionary (2 of 3)

- 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 (3 of 3)

Figure 18-2 UML diagram for a class of dictionaries

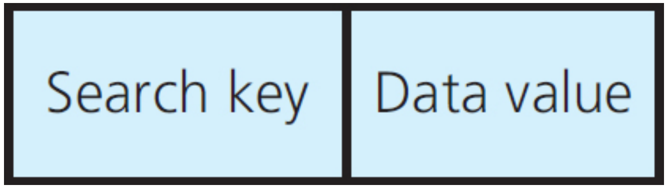


```
Dictionary

+isEmpty(): boolean
+getNumberOfEntries(): integer
+add(searchKey: KeyType, newValue: ValueType): boolean
+remove(targetKey: KeyType): boolean
+clear(): void
+getValue(targetKey: KeyType): ValueType
+contains(targetKey: KeyType): boolean
+traverse(visit(value: ValueType): void): void
```

Possible Implementations (1 of 9)

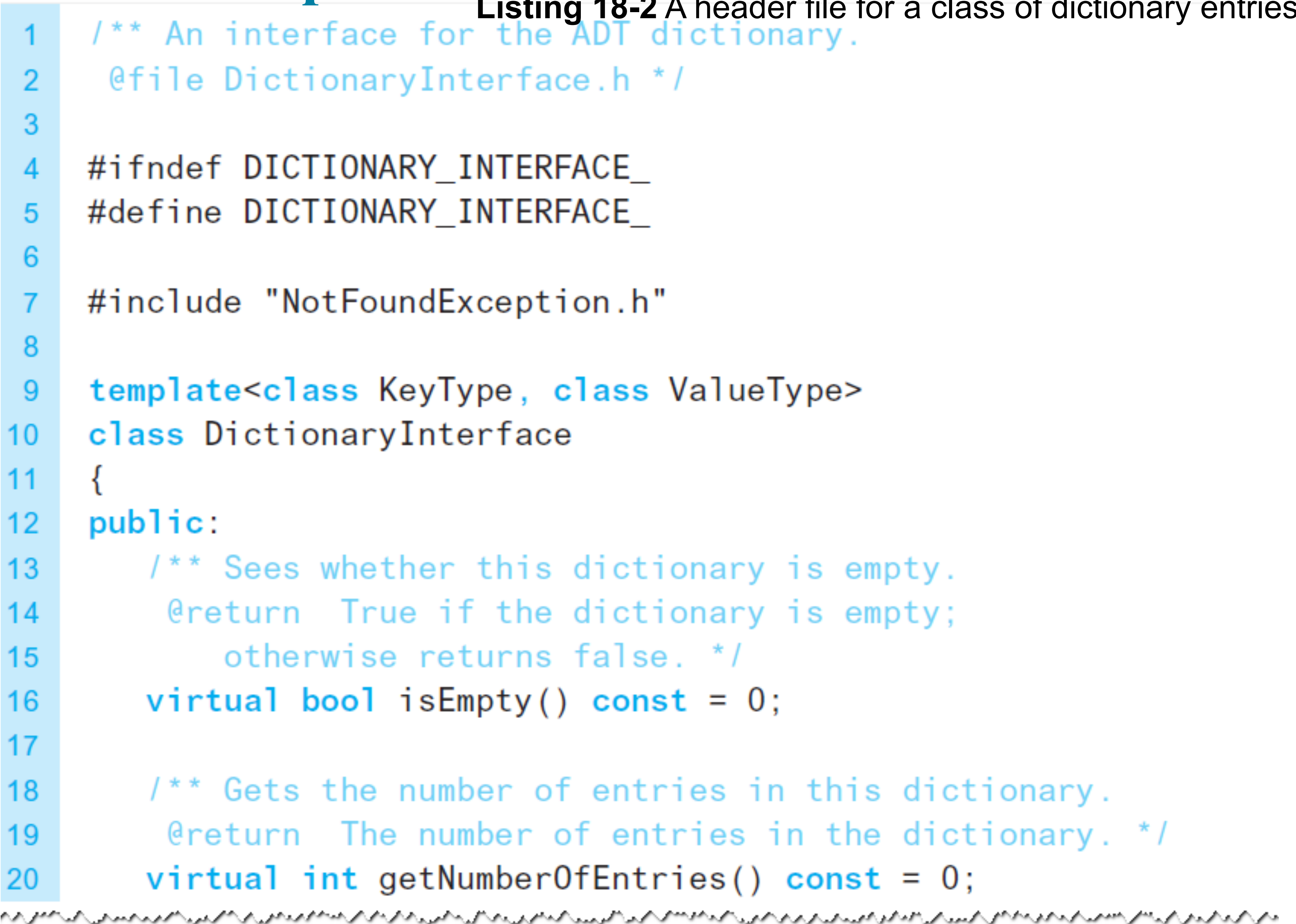
- Categories of linear implementations
  - Sorted by search key array-based
  - Sorted by search key link-based
  - Unsorted array-based
  - Unsorted link-based

Figure 18-3 A dictionary entry



Possible Implementations (2 of 9)

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



```
/** An interface for the ADT dictionary.
 @file DictionaryInterface.h */

#ifndef DICTIONARY_INTERFACE_
#define DICTIONARY_INTERFACE_

#include "NotFoundException.h"

template<class KeyType, class ValueType>
class DictionaryInterface
{
public:
   /** Sees whether this dictionary is empty.
    @return  True if the dictionary is empty;
       otherwise returns false. */
   virtual bool isEmpty() const = 0;

   /** Gets the number of entries in this dictionary.
    @return  The number of entries in the dictionary. */
   virtual int getNumberOfEntries() const = 0;
```

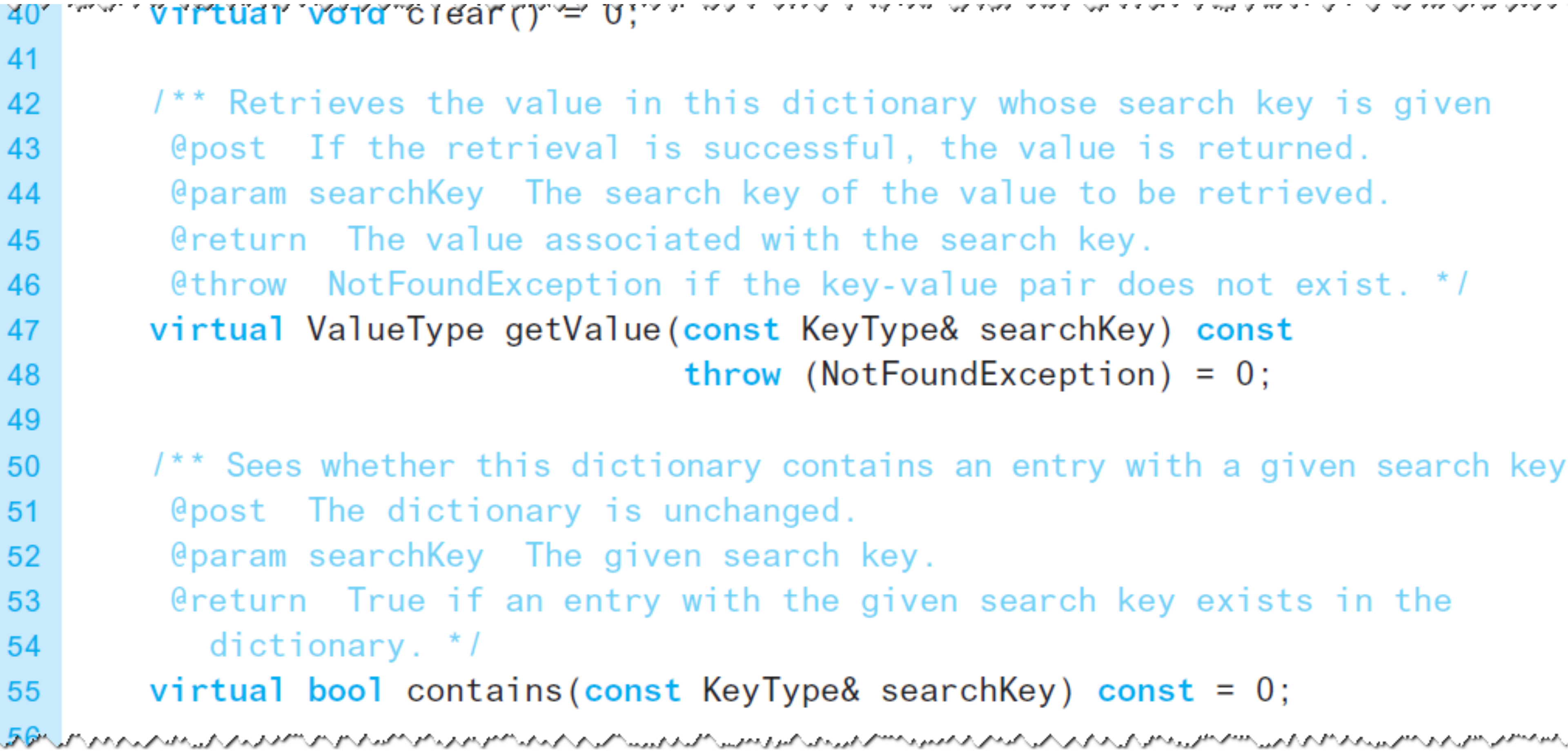
Possible Implementations (3 of 9)

Listing 18-2 [Continued]

```
   /** Adds a new search key and associated value to this dictionary.
    @pre  The new search key differs from all search keys presently
        in the dictionary.
    @post  If the addition is successful, the new key-value pair is in its
        proper position within the dictionary.
    @param searchKey  The search key associated with the value to be added.
    @param newValue  The value to be added.
    @return  True if the entry was successfully added, or false if not. */
   virtual bool add(const KeyType& searchKey, const ValueType& newValue) = 0;

   /** Removes a key-value pair from this dictionary.
    @post  If the entry whose search key equals searchKey existed in the
      dictionary, the entry was removed.
    @param searchKey  The search key of the entry to be removed.
    @return  True if the entry was successfully removed, or false if not. */
   virtual bool remove(const KeyType& searchKey) = 0;

   /** Removes all entries from this dictionary. */
   virtual void clear() = 0;
```

Possible Implementations (4 of 9)

Listing 18-2 [Continued]



```
    virtual void clear() = 0;

    /** Retrieves the value in this dictionary whose search key is given
     @post  If the retrieval is successful, the value is returned.
     @param searchKey  The search key of the value to be retrieved.
     @return  The value associated with the search key.
     @throw  NotFoundException if the key-value pair does not exist. */
    virtual ValueType getValue(const KeyType& searchKey) const
                               throw (NotFoundException) = 0;

    /** Sees whether this dictionary contains an entry with a given search key
     @post  The dictionary is unchanged.
     @param searchKey  The given search key.
     @return  True if an entry with the given search key exists in the
       dictionary. */
    virtual bool contains(const KeyType& searchKey) const = 0;
```

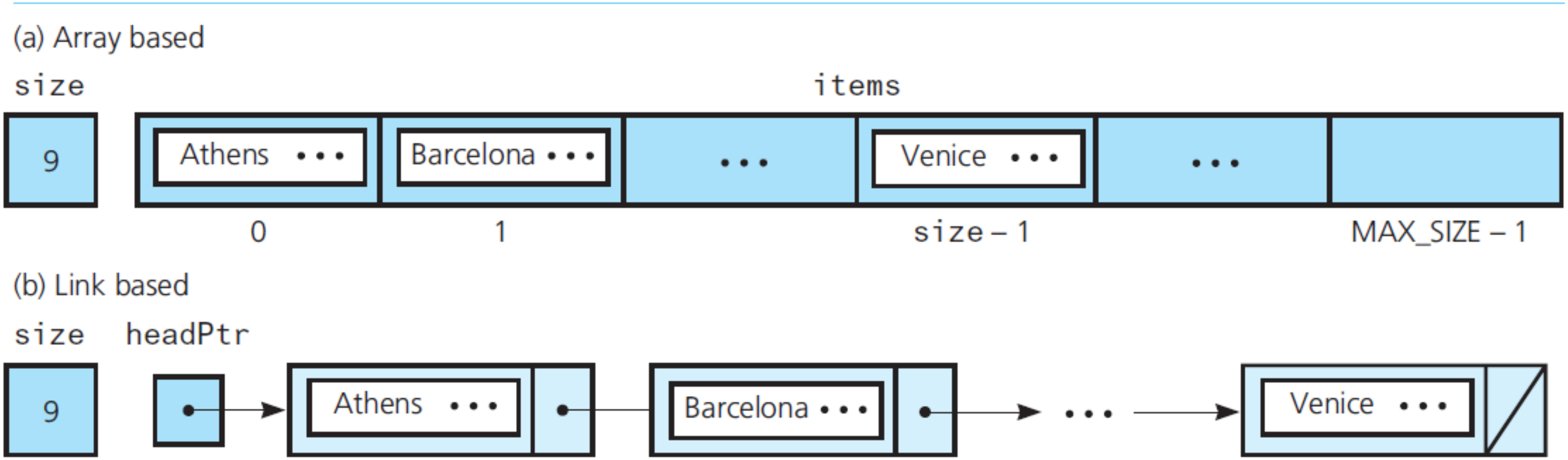
Possible Implementations (5 of 9)

Listing 18-2 [Continued]

```
   virtual bool contains(const KeyType& searchKey) const = 0;

   /** Traverses this dictionary and calls a given client function once
       for each entry.
    @post  The given function's action occurs once for each entry in the
       dictionary and possibly alters the entry.
    @param visit  A client function. */
   virtual void traverse(void visit(ValueType&)) const = 0;
   /** Destroys this dictionary and frees its assigned memory. */

   virtual ~DictionaryInterface(){ }
}; // end DictionaryInterface
#endif
```

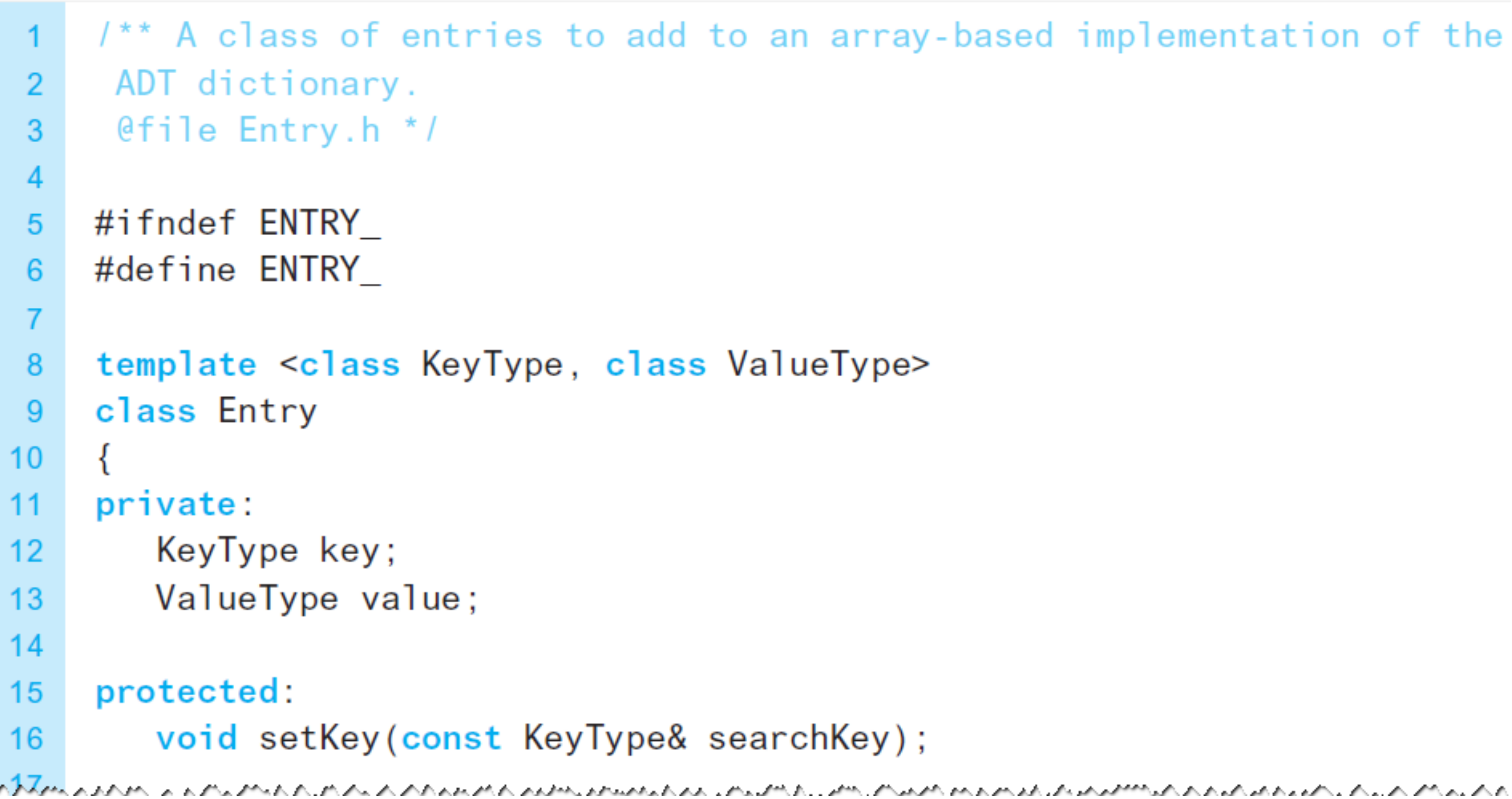
Possible Implementations (6 of 9)

Figure 18-4 Data members for two sorted linear implementations of the ADT dictionary for the data in Figure 18-1 (see slide 2)



Possible Implementations (7 of 9)

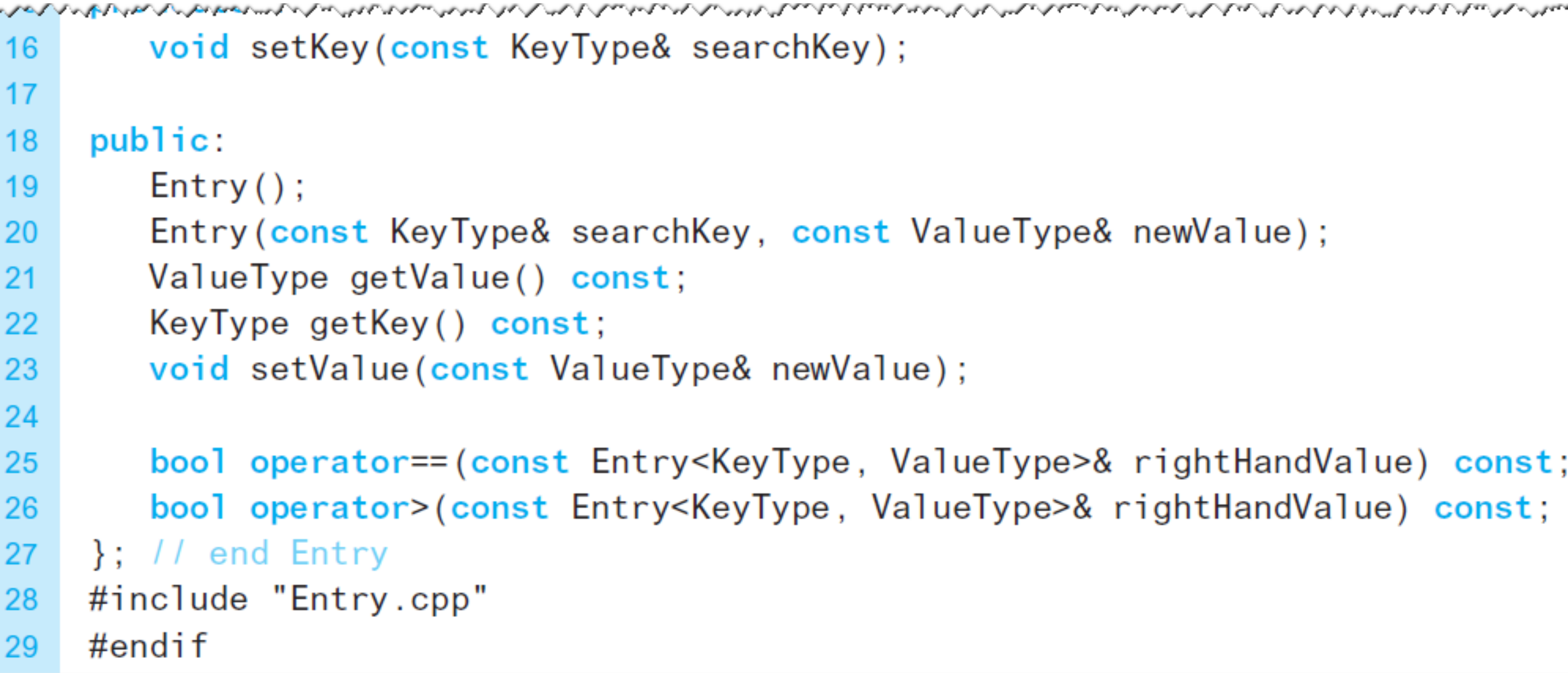
Listing 18-2 [Continued]



```
/** A class of entries to add to an array-based implementation of the
 ADT dictionary.
 @file Entry.h */

#ifndef ENTRY_
#define ENTRY_

template <class KeyType, class ValueType>
class Entry
{
private:
   KeyType key;
   ValueType value;

protected:
   void setKey(const KeyType& searchKey);
```

Possible Implementations (8 of 9)

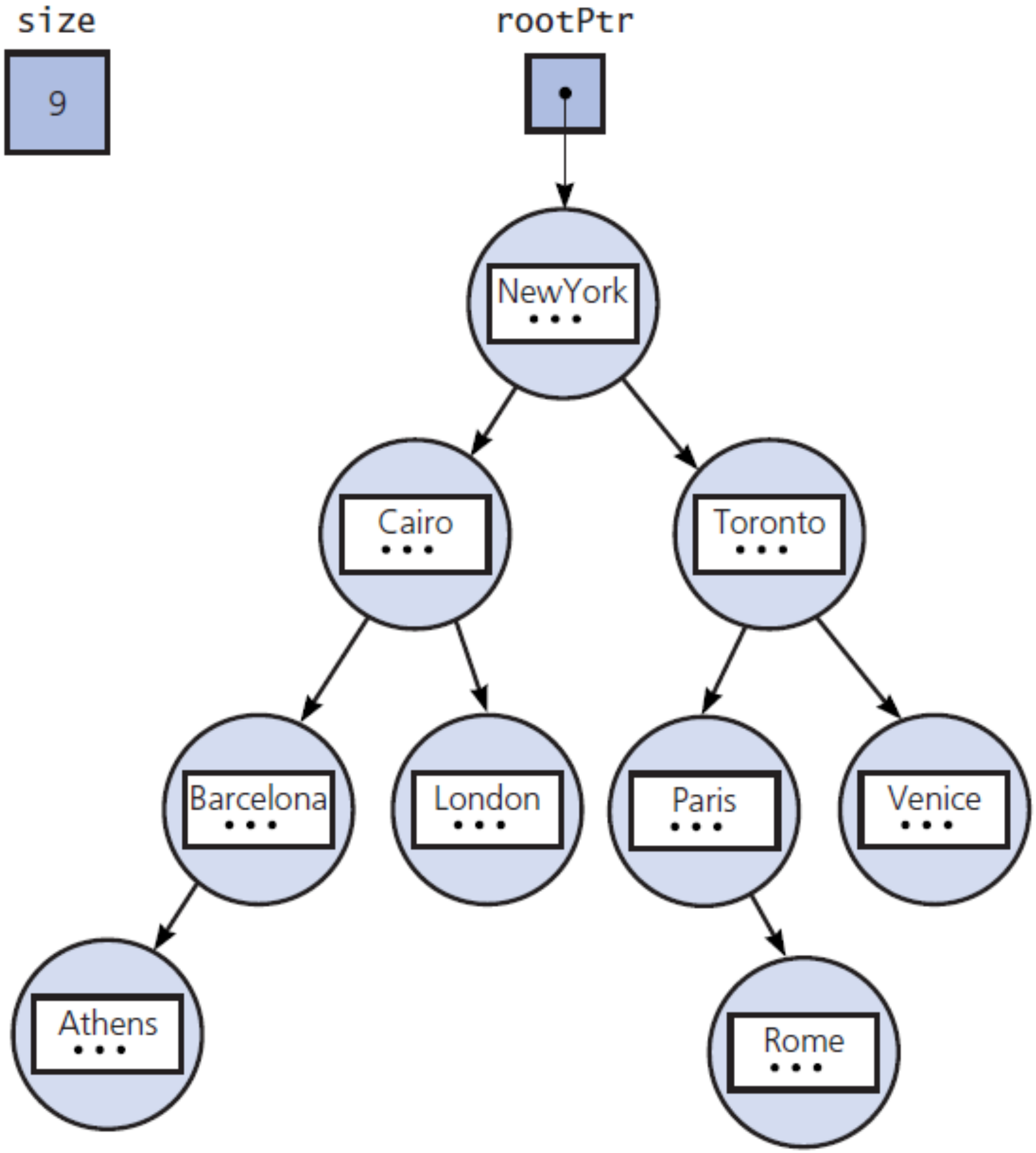
Listing 18-2 [Continued]



```
    void setKey(const KeyType& searchKey);

public:
    Entry();
    Entry(const KeyType& searchKey, const ValueType& newValue);
    ValueType getValue() const;
    KeyType getKey() const;
    void setValue(const ValueType& newValue);

    bool operator==(const Entry<KeyType, ValueType>& rightHandValue) const;
    bool operator>(const Entry<KeyType, ValueType>& rightHandValue) const;
}; // end Entry
#include "Entry.cpp"
#endif
```

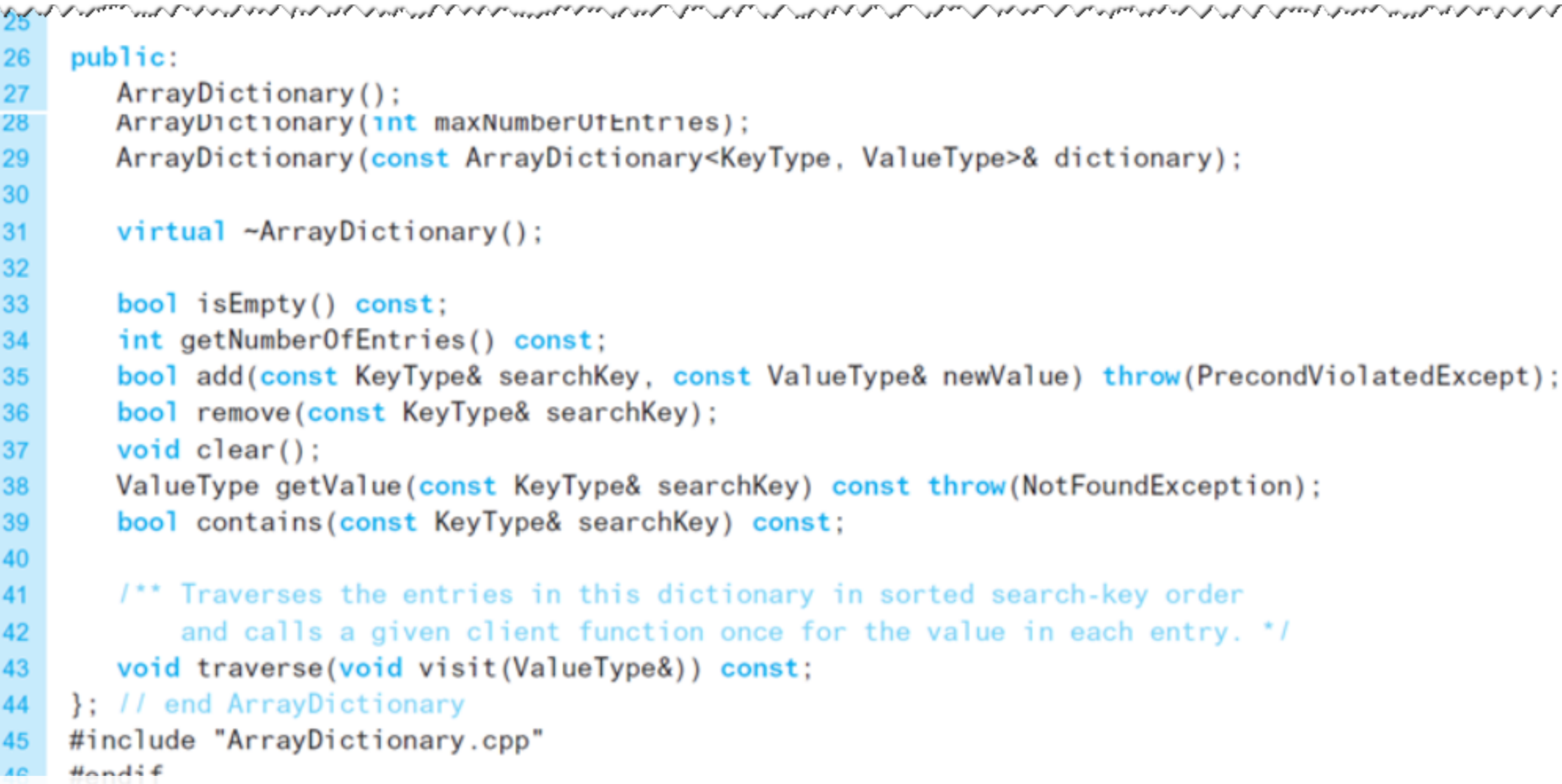
Possible Implementations (9 of 9)

Figure 18-5 The data members for a binary search tree implementation of the ADT dictionary for the data in Figure 18-1 (see slide 2)



Sorted Array-Based Implementation of ADT Dictionary

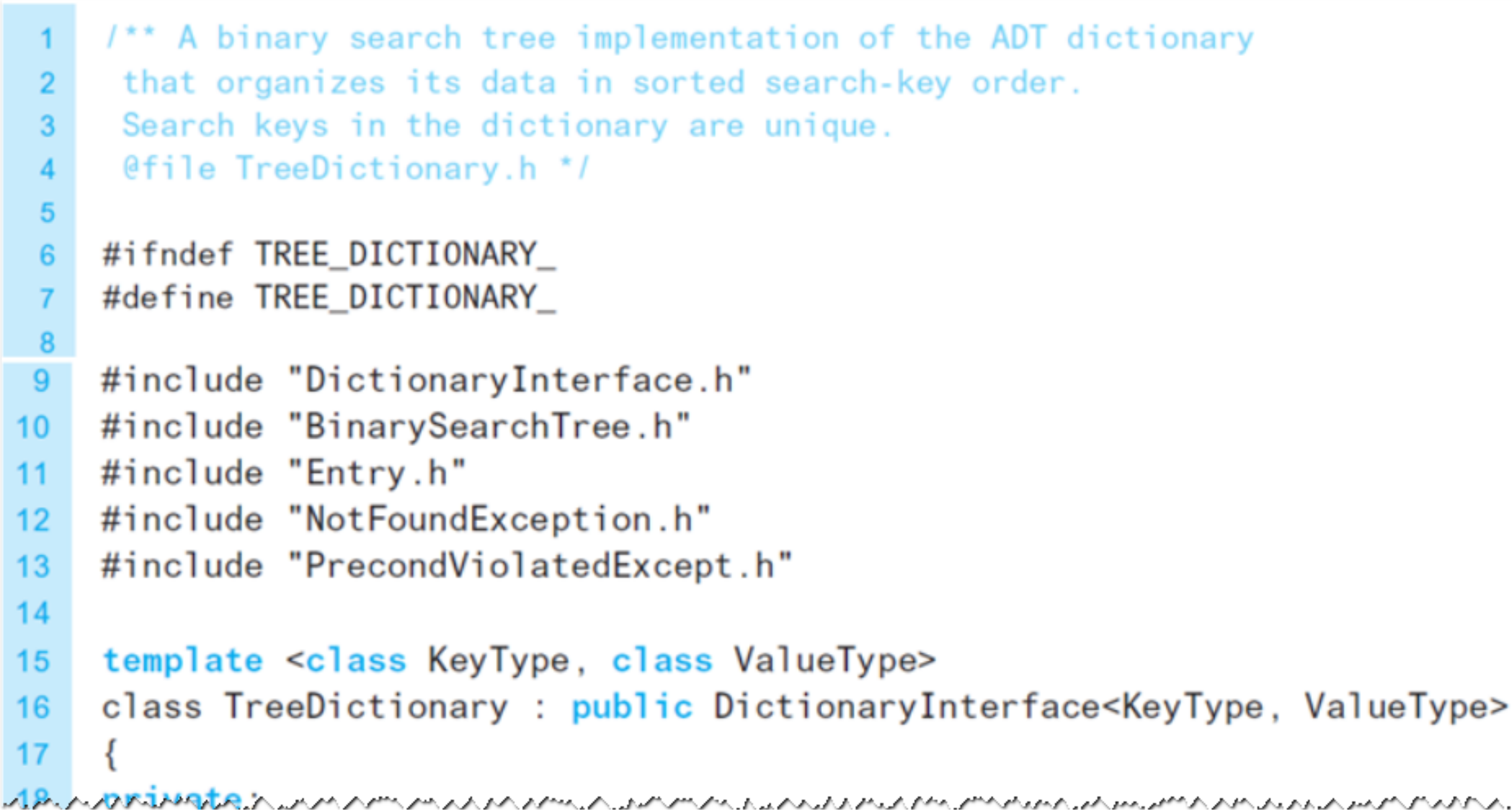
Listing 18-3 A header file for the class **ArrayDictionary**



```
public:
   ArrayDictionary();
   ArrayDictionary(int maxNumberOfEntries);
   ArrayDictionary(const ArrayDictionary<KeyType, ValueType>& dictionary);

   virtual ~ArrayDictionary();

   bool isEmpty() const;
   int getNumberOfEntries() const;
   bool add(const KeyType& searchKey, const ValueType& newValue) throw(PrecondViolatedExcept);
   bool remove(const KeyType& searchKey);
   void clear();
   ValueType getValue(const KeyType& searchKey) const throw(NotFoundException);
   bool contains(const KeyType& searchKey) const;

   /** Traverses the entries in this dictionary in sorted search-key order
       and calls a given client function once for the value in each entry. */
   void traverse(void visit(ValueType&)) const;
}; // end ArrayDictionary
#include "ArrayDictionary.cpp"
#endif
```

Binary Search Tree Implementation of the ADT Dictionary (1 of 3)

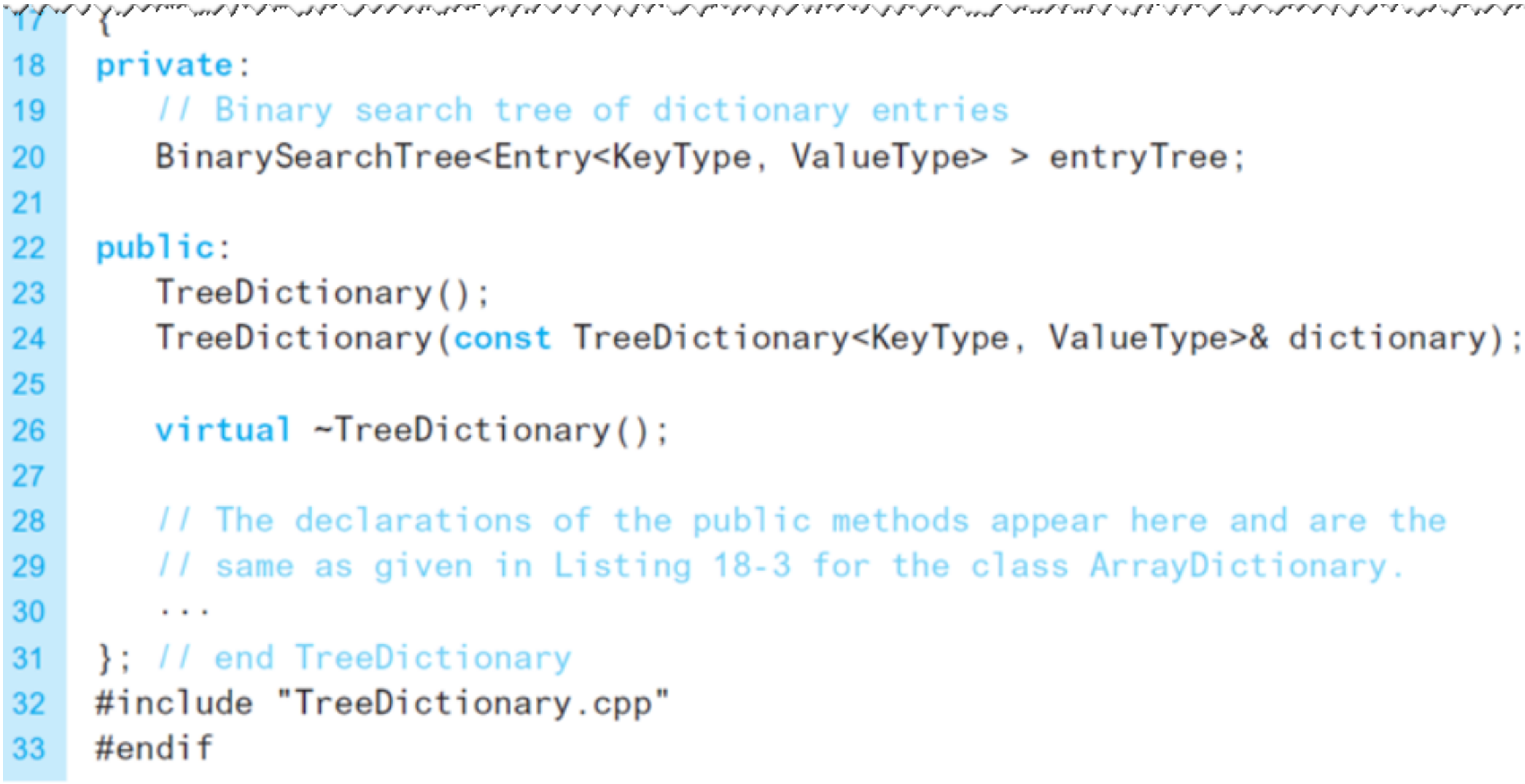
Listing 18-4 A header file for the class **TreeDictionary**



```
/** A binary search tree implementation of the ADT dictionary
 that organizes its data in sorted search-key order.
 Search keys in the dictionary are unique.
 @file TreeDictionary.h */

#ifndef TREE_DICTIONARY_
#define TREE_DICTIONARY_

#include "DictionaryInterface.h"
#include "BinarySearchTree.h"
#include "Entry.h"
#include "NotFoundException.h"
#include "PrecondViolatedExcept.h"

template <class KeyType, class ValueType>
class TreeDictionary : public DictionaryInterface<KeyType, ValueType>
{
private:
```

Binary Search Tree Implementation of the ADT Dictionary (2 of 3)

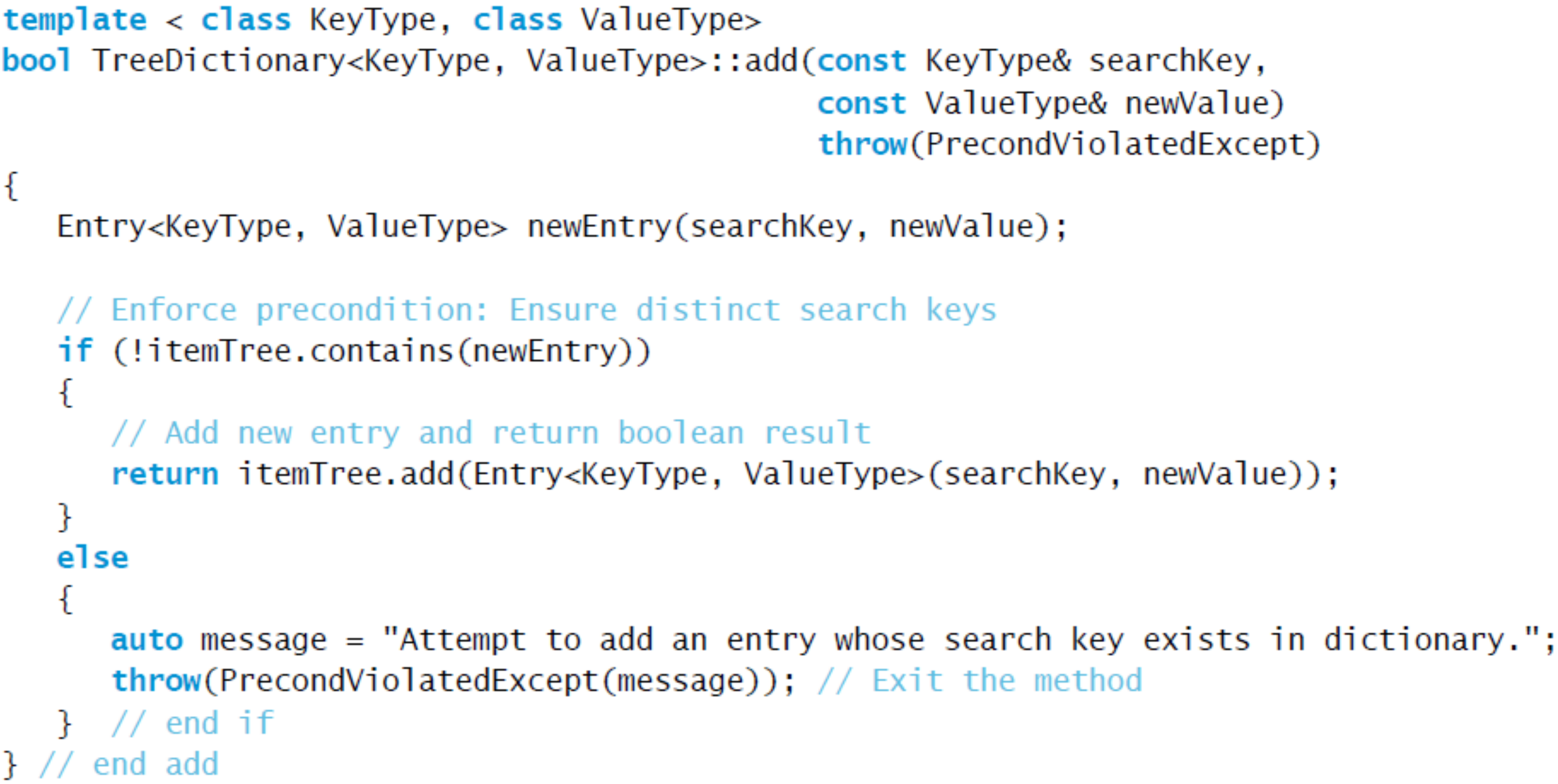
Listing 18-4 [Continued]



```
{
private:
   // Binary search tree of dictionary entries
   BinarySearchTree<Entry<KeyType, ValueType> > entryTree;

public:
   TreeDictionary();
   TreeDictionary(const TreeDictionary<KeyType, ValueType>& dictionary);

   virtual ~TreeDictionary();

   // The declarations of the public methods appear here and are the
   // same as given in Listing 18-3 for the class ArrayDictionary.
   . . .
}; // end TreeDictionary
#include "TreeDictionary.cpp"
#endif
```

Binary Search Tree Implementation of the ADT Dictionary (3 of 3)

Method **add** which prevents duplicate keys.



```
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
```

Selecting an Implementation

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

Three Scenarios (1 of 5)

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

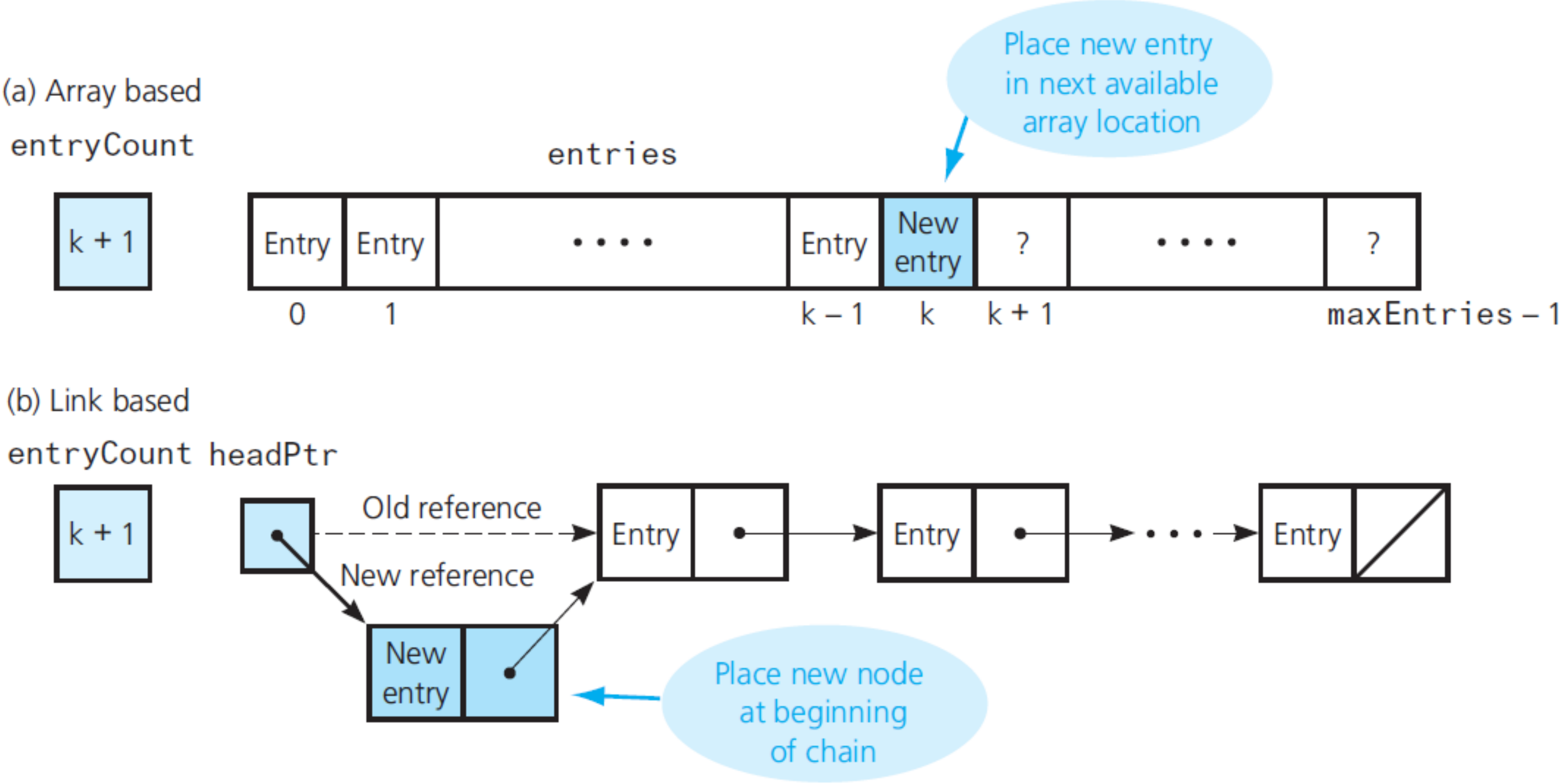
Three Scenarios (2 of 5)

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

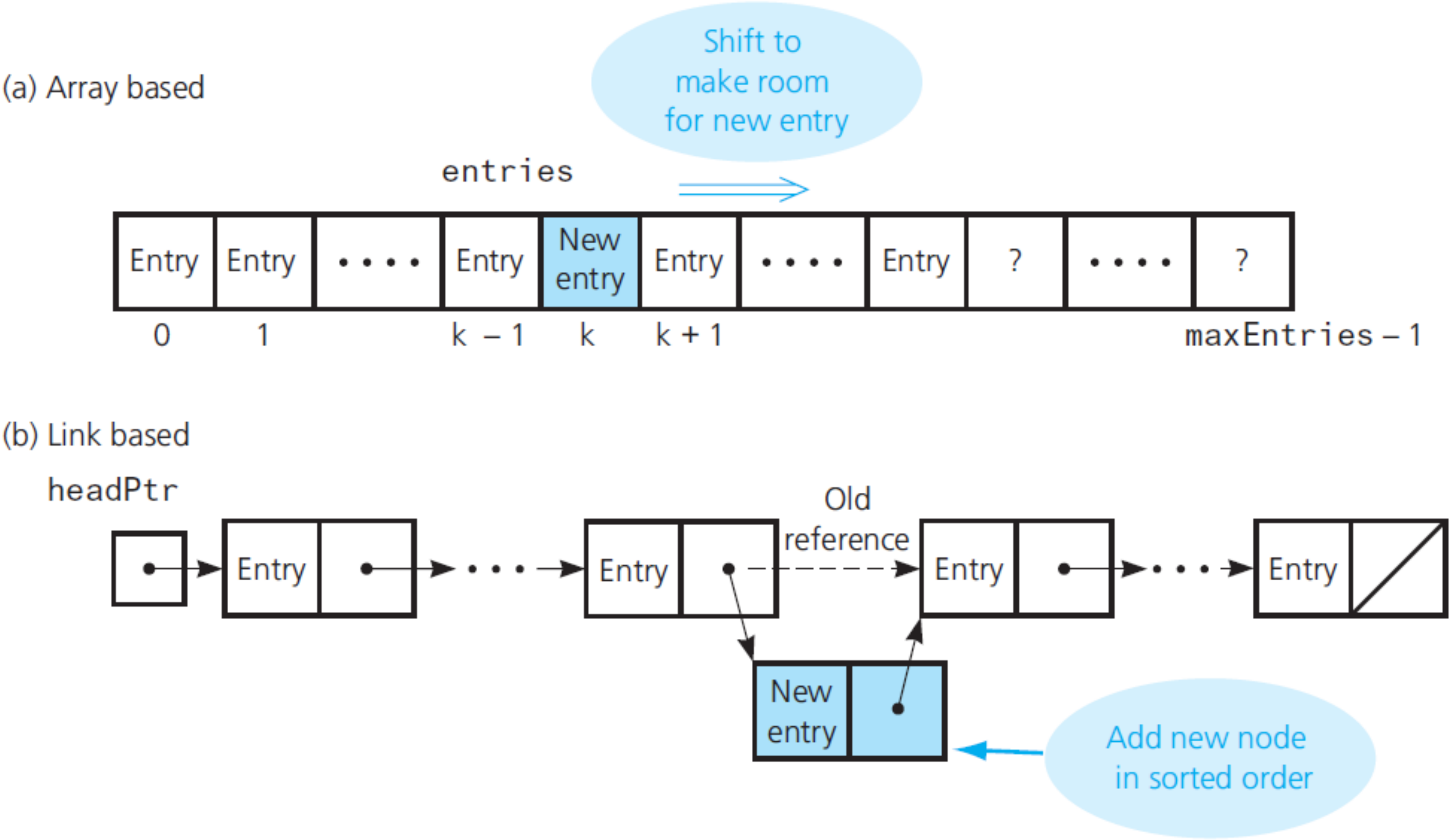
Three Scenarios (3 of 5)

Figure 18-6 Addition for unsorted linear implementations



Three Scenarios (4 of 5)

Figure 18-7 Addition for sorted linear implementations



Three Scenarios (5 of 5)

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

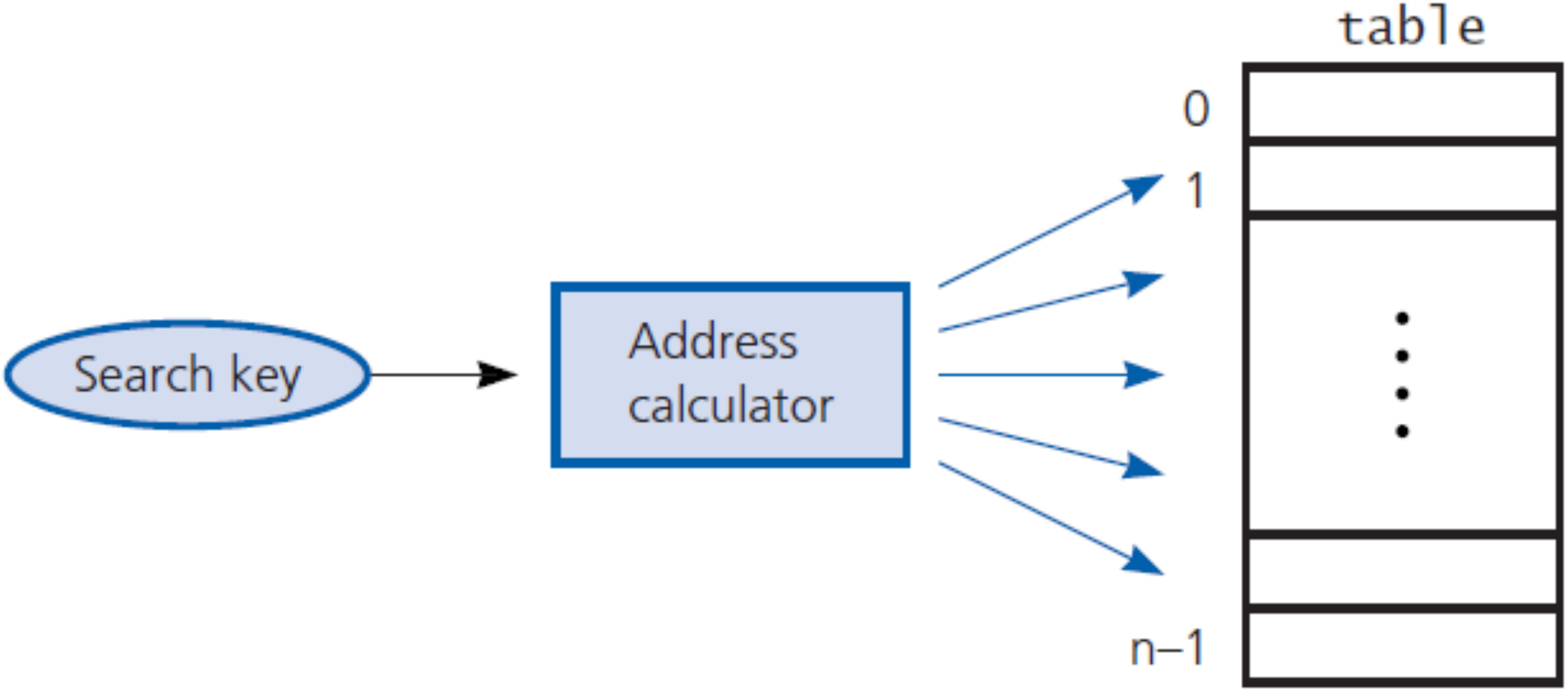
	Addition	**Removal**	**Retrieval**	**Traversal**
Unsorted array-based	O(1)	O(*n*)	O(*n*)	O(*n*)
Unsorted link-based	O(1)	O(*n*)	O(*n*)	O(*n*)
Sorted array-based	O(*n*)	O(*n*)	O(log *n*)	O(*n*)
Sorted link-based	O(*n*)	O(*n*)	O(*n*)	O(*n*)
Binary search tree	O(log *n*)	O(log *n*)	O(log *n*)	O(*n*)

Hashing as a Dictionary Implementation (1 of 3)

- 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 (2 of 3)

Figure 18-9 Address calculator



Hashing as a Dictionary Implementation (3 of 3)

- 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

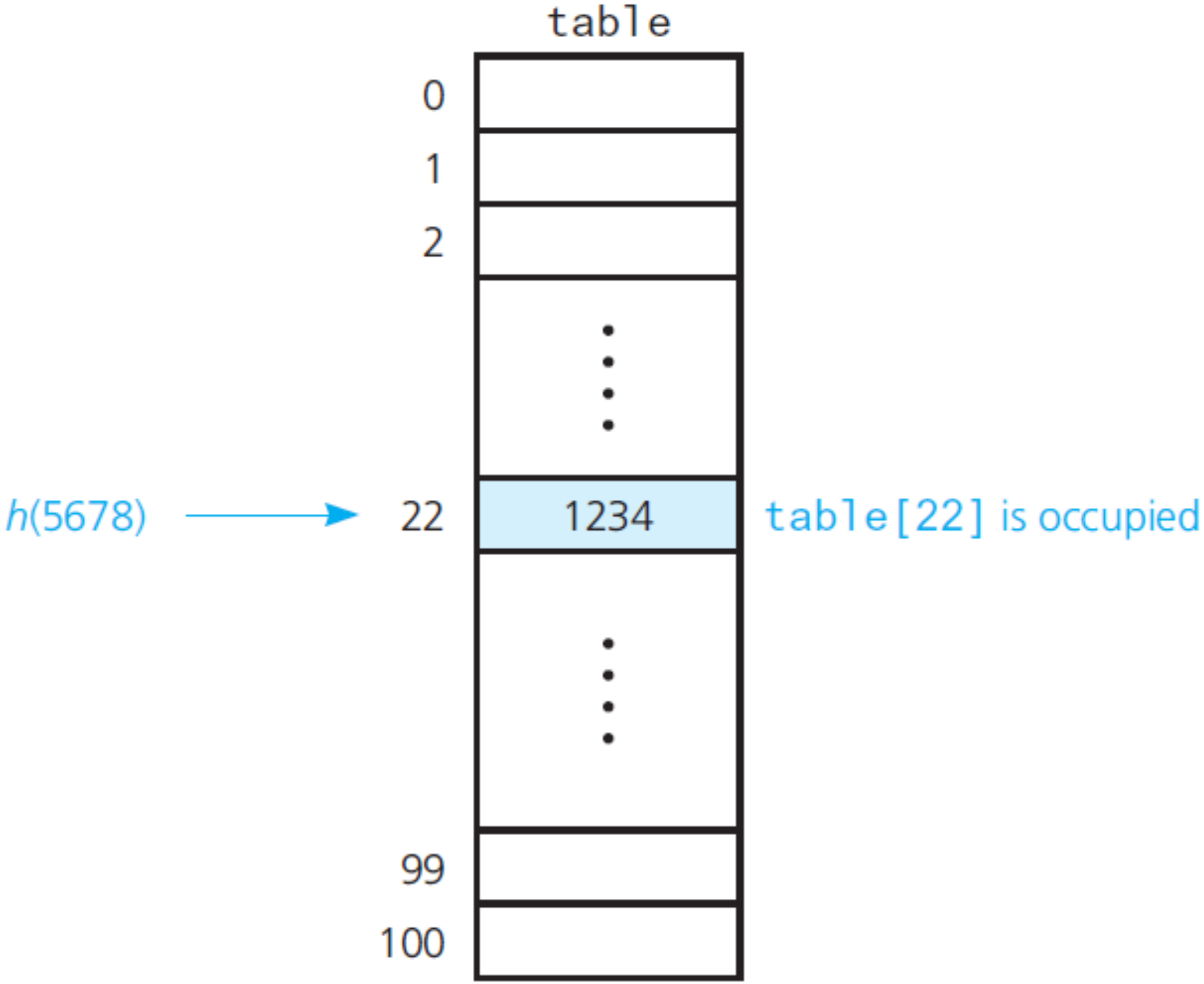
Hash Functions

- 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 $h(x) = x \bmod \textbf{tableSize}$
  - Convert character string to an integer – use ASCII values

Resolving Collisions with Open Addressing

(1 of 7)

Figure 18-10 A collision



Resolving Collisions with Open Addressing

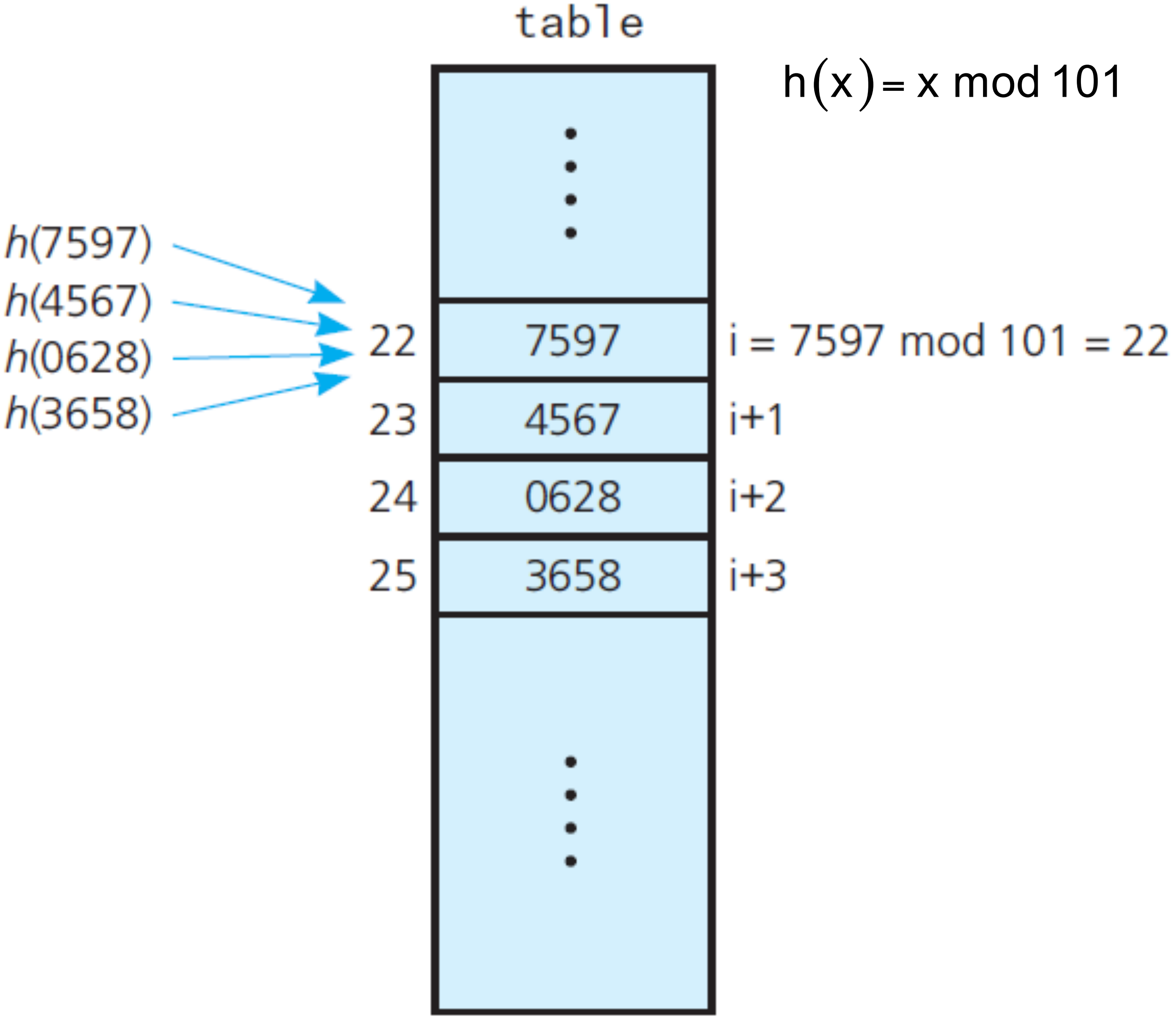
(2 of 7)

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

Resolving Collisions with Open Addressing

(3 of 7)

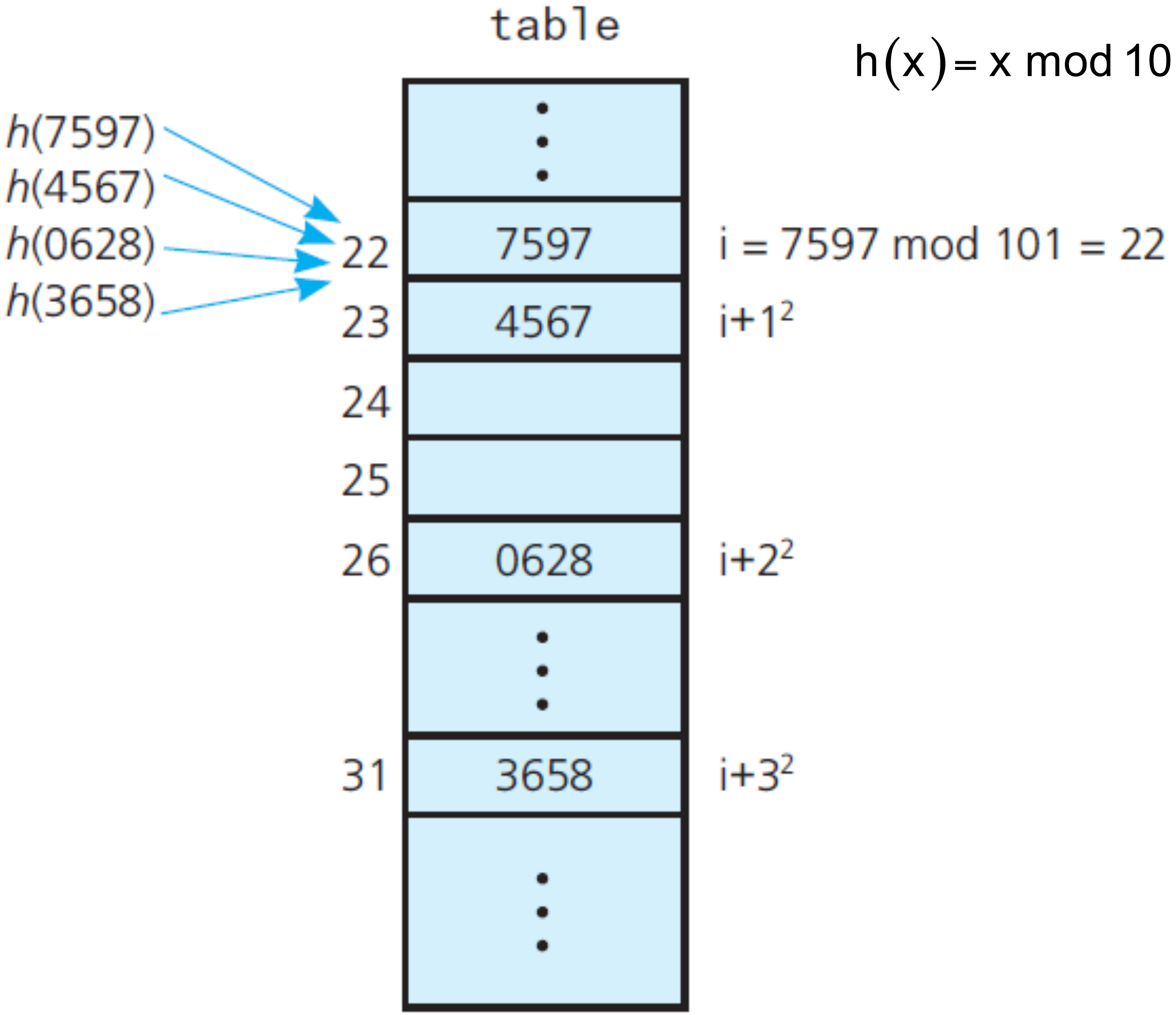
Figure 18-11 Linear probing with $h(x) = x \bmod 101$



Resolving Collisions with Open Addressing

(4 of 7)

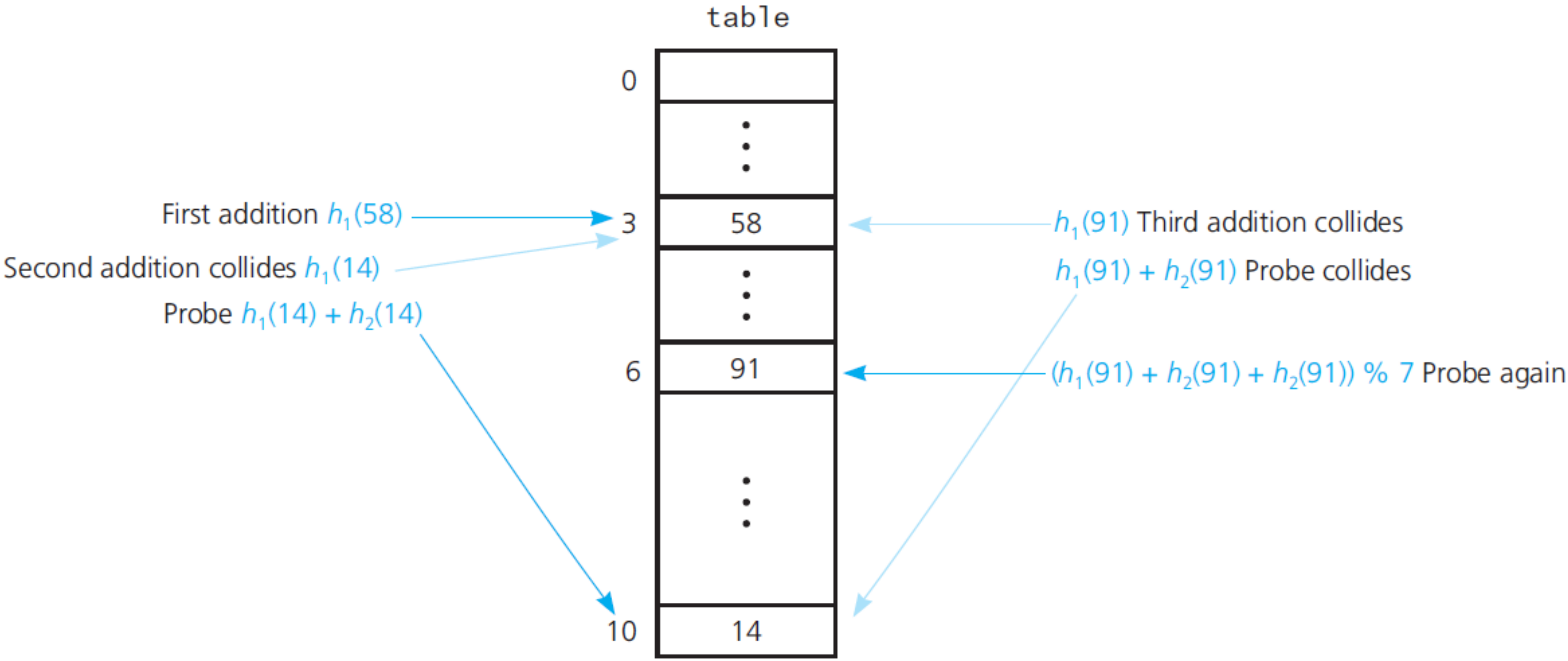
Figure 18-12 Quadratic probing with



Resolving Collisions with Open Addressing

(5 of 7)

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



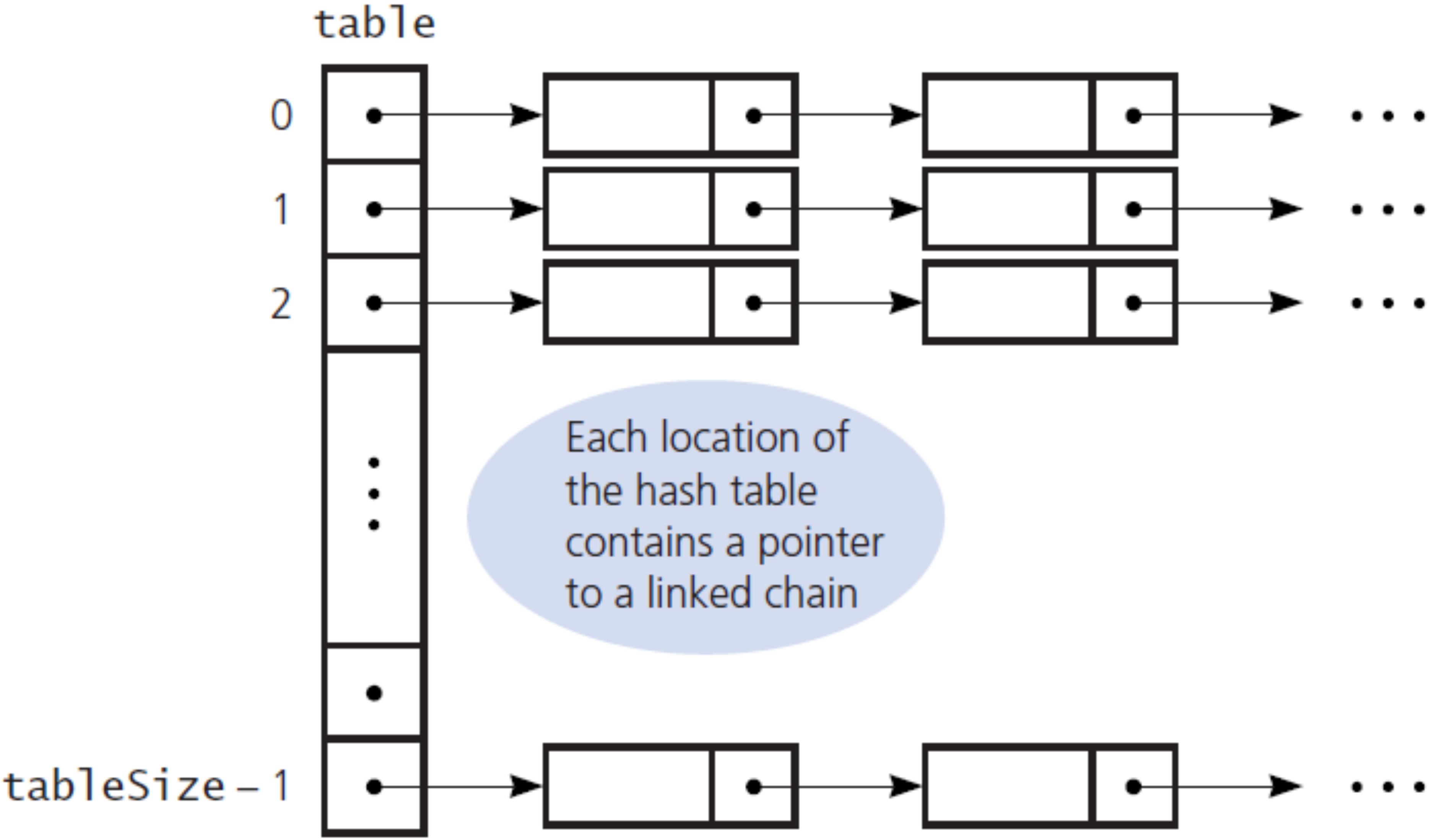
Resolving Collisions with Open Addressing

(6 of 7)

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

Resolving Collisions with Open Addressing

(7 of 7)



Each location of the hash table contains a pointer to a linked chain

Figure 18-14 Separate chaining

The Efficiency of Hashing (1 of 6)

- 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 ful

The Efficiency of Hashing (2 of 6)

- Linear probing – average number of comparisons

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

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

The Efficiency of Hashing (3 of 6)

- Quadratic probing and double hashing – average number of comparisons

$$\frac{-\log_e(1-\alpha)}{\alpha} \quad \text{for a successful search, and}$$

$$\frac{1}{1-\alpha} \quad \text{for an unsuccessful search}$$

The Efficiency of Hashing (4 of 6)

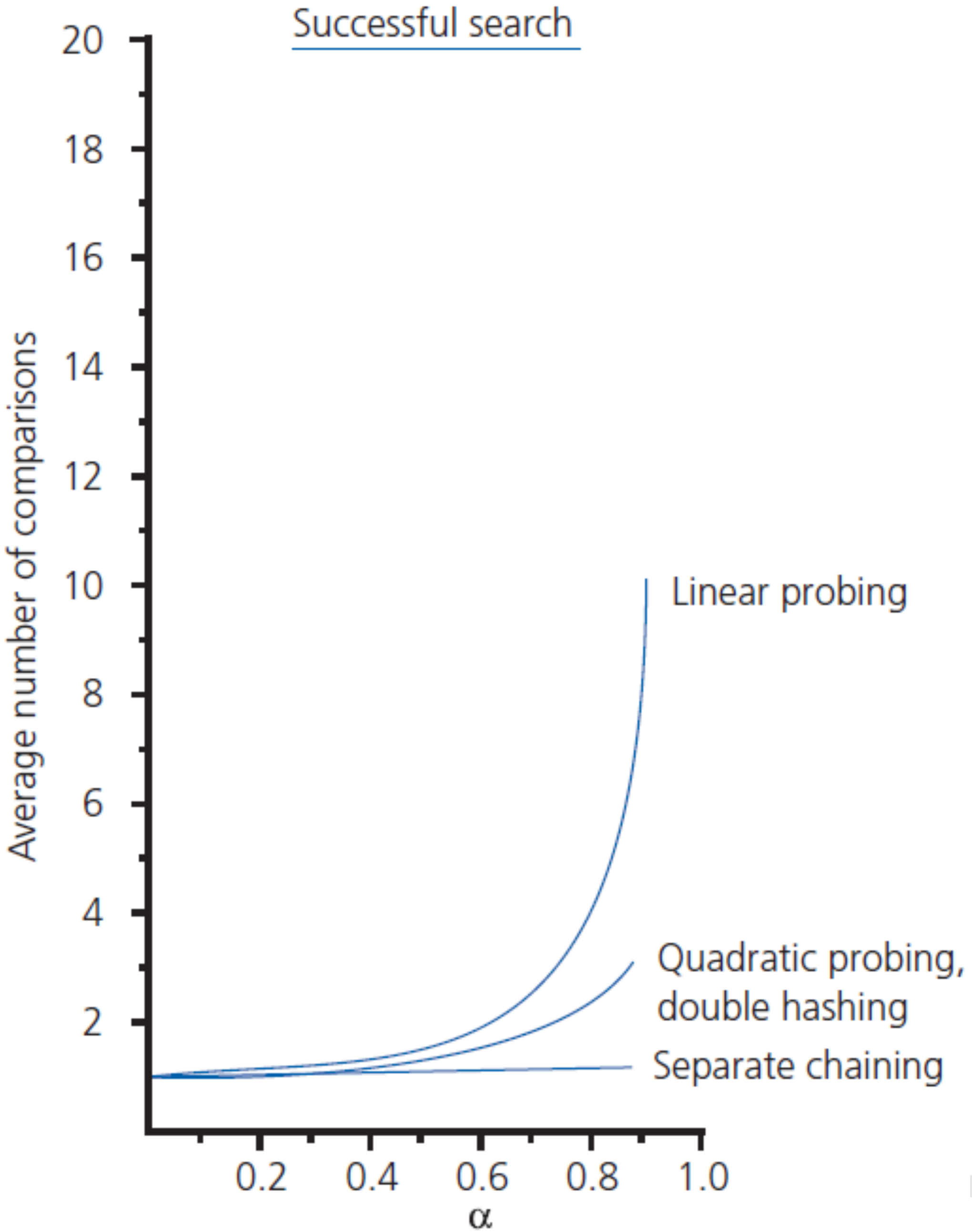
- Efficiency of the retrieval and removal operations under the separate-chaining approach

$1+\frac{\alpha}{2}$ for a successful search, and

α for an unsuccessful search

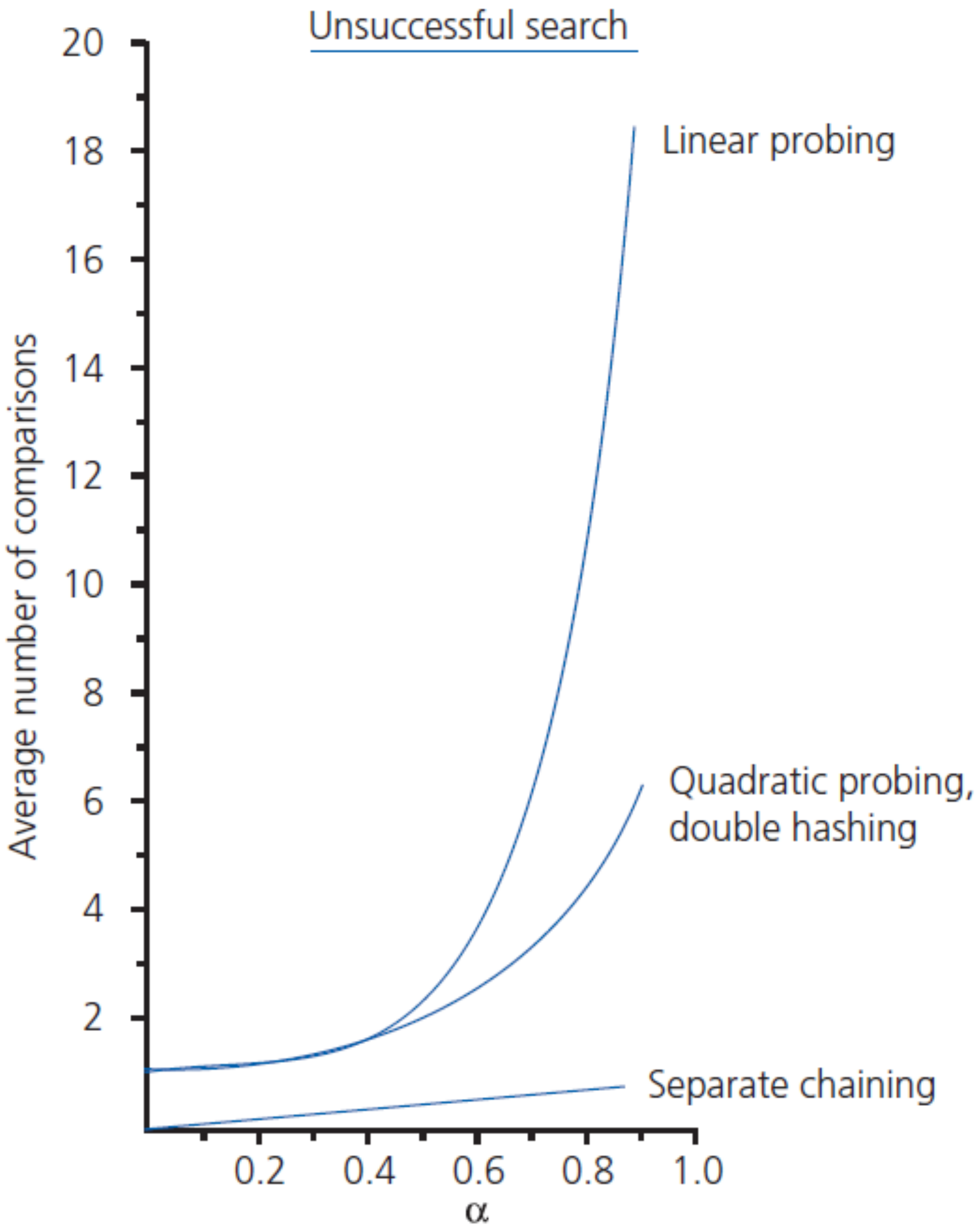
The Efficiency of Hashing (5 of 6)

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



The Efficiency of Hashing (6 of 6)

Figure 18-15 [Continued]



What Constitutes a Good Hash Function?

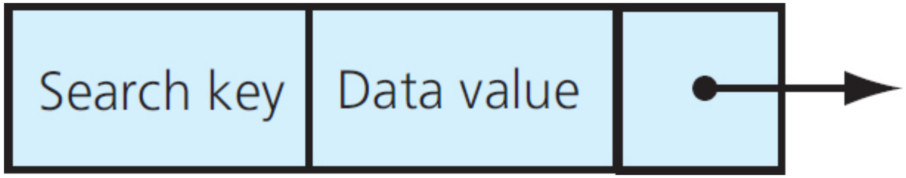
- Is hash function easy and fast to compute?
- Does hash function scatter data evenly throughout hash table?
- How well does hash function scatter random data?
- How well does hash function scatter **non**-random data?

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

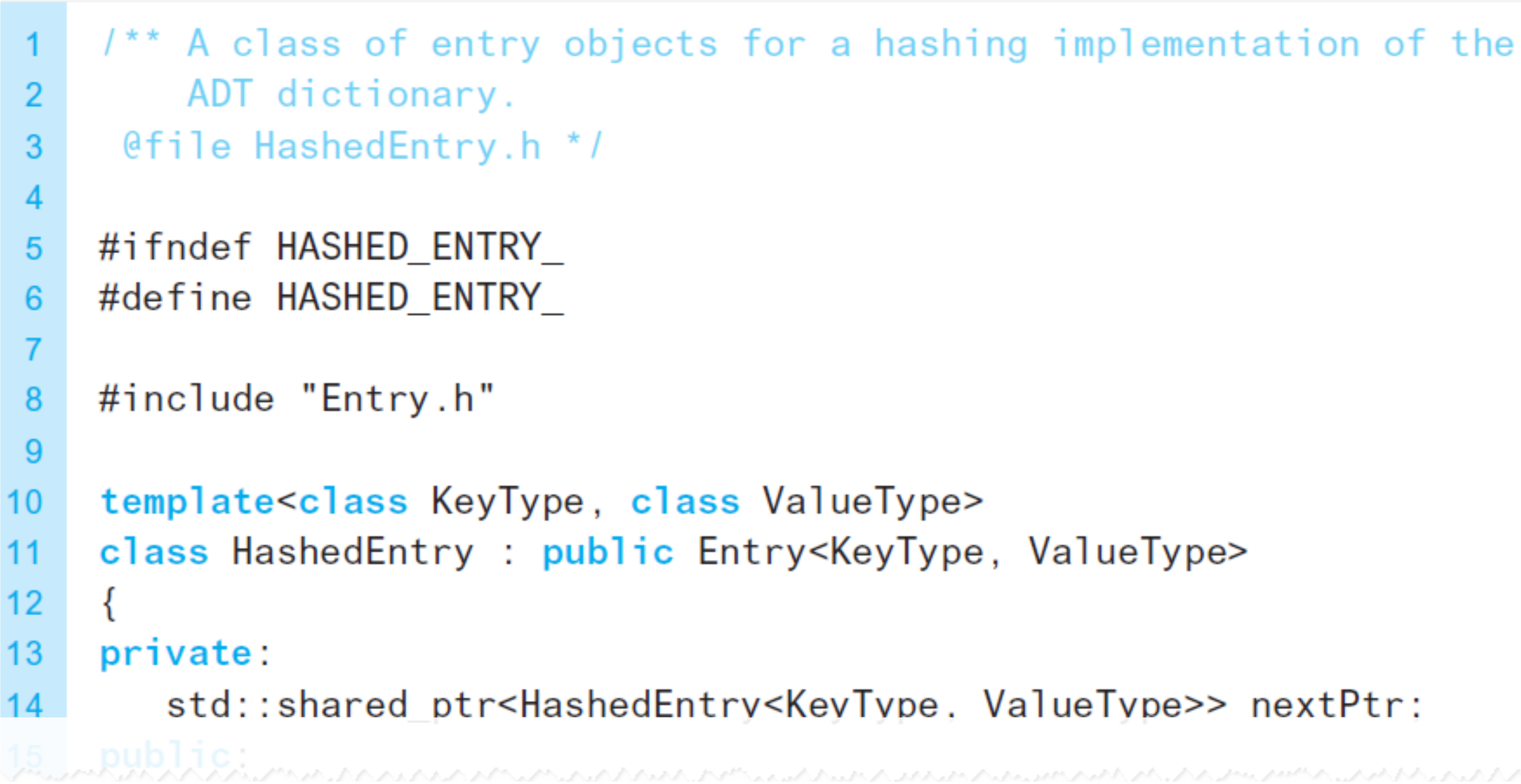
Using Hashing, Separate Chaining to Implement ADT Dictionary (1 of 3)

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



Using Hashing, Separate Chaining to Implement ADT Dictionary (2 of 3)

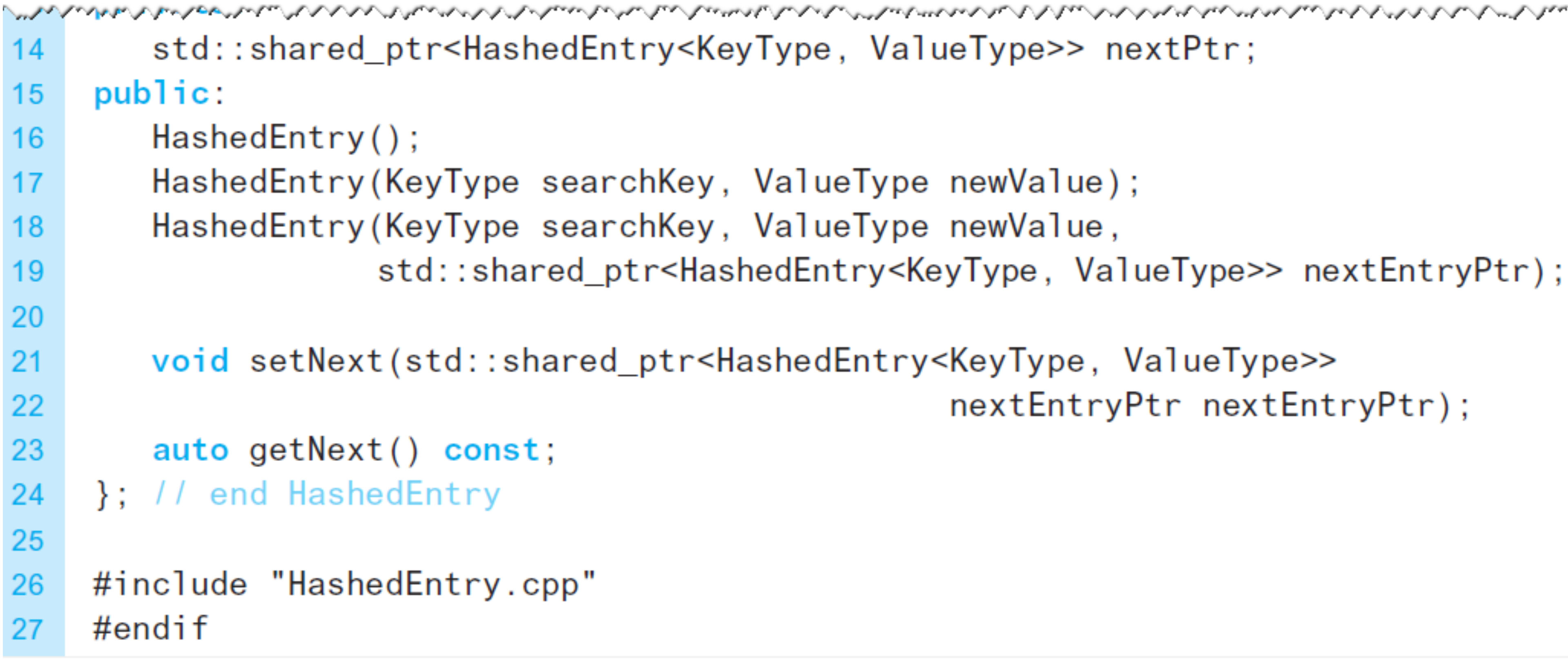
Listing 18-5 The class **HashedEntry**



```
/** A class of entry objects for a hashing implementation of the
    ADT dictionary.
 @file HashedEntry.h */

#ifndef HASHED_ENTRY_
#define HASHED_ENTRY_

#include "Entry.h"

template<class KeyType, class ValueType>
class HashedEntry : public Entry<KeyType, ValueType>
{
private:
   std::shared_ptr<HashedEntry<KeyType, ValueType>> nextPtr;
public:
```

Using Hashing, Separate Chaining to Implement ADT Dictionary (3 of 3)

Listing 18-5 [Continued]



```
   std::shared_ptr<HashedEntry<KeyType, ValueType>> nextPtr;
public:
   HashedEntry();
   HashedEntry(KeyType searchKey, ValueType newValue);
   HashedEntry(KeyType searchKey, ValueType newValue,
               std::shared_ptr<HashedEntry<KeyType, ValueType>> nextEntryPtr);

   void setNext(std::shared_ptr<HashedEntry<KeyType, ValueType>>
                                             nextEntryPtr nextEntryPtr);
   auto getNext() const;
}; // end HashedEntry

#include "HashedEntry.cpp"
#endif
```