Class 08 - Dictionaries

Important & simple.

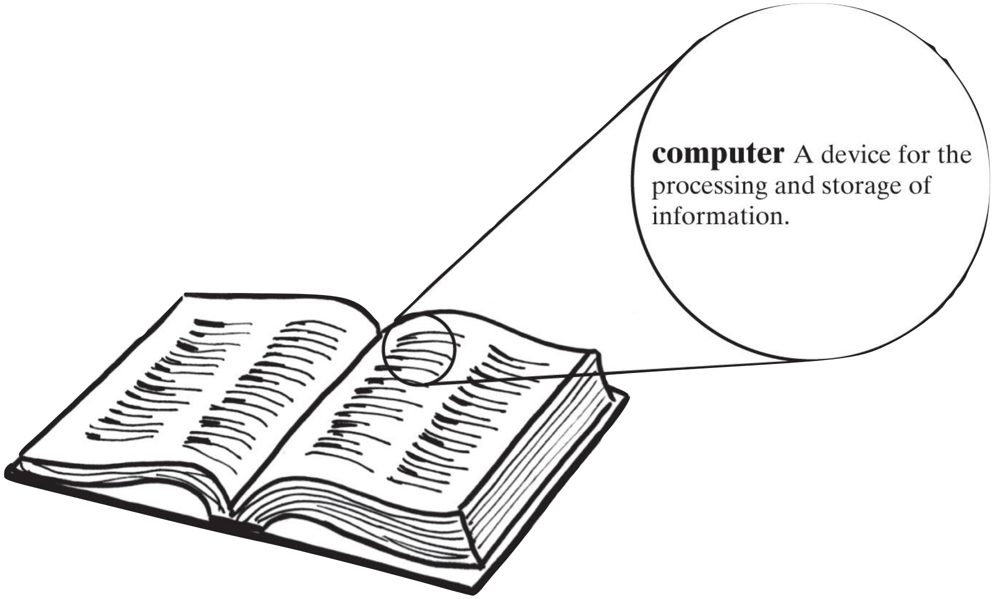
key & value.

can be any object.

CSIS 3475 Data Structures and Algorithms

Dictionaries

- When you want to look up …
  - The meaning of a word
  - An address
  - A phone number
  - A contact on your phone
- These can be implemented in an ADT Dictionary


computer A device for the processing and storage of information.

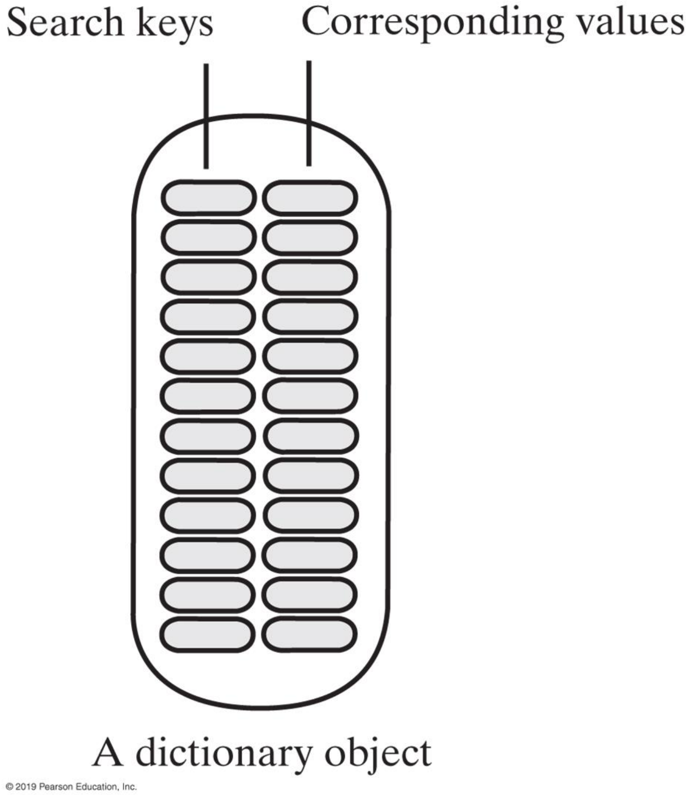
© 2019 Pearson Education, Inc.

Specifications for the ADT Dictionary

- Synonyms for ADT Dictionary
  - Map
  - Table
  - Associative array
- An entry in the dictionary contains
  - Keyword, search key
  - Value

Specifications for the ADT Dictionary

- Dictionary Data
  - *Collection of pairs* (k, v) of objects k and v,
    - k is the search key
    - v is the corresponding value
  - *Number of pairs* in the collection



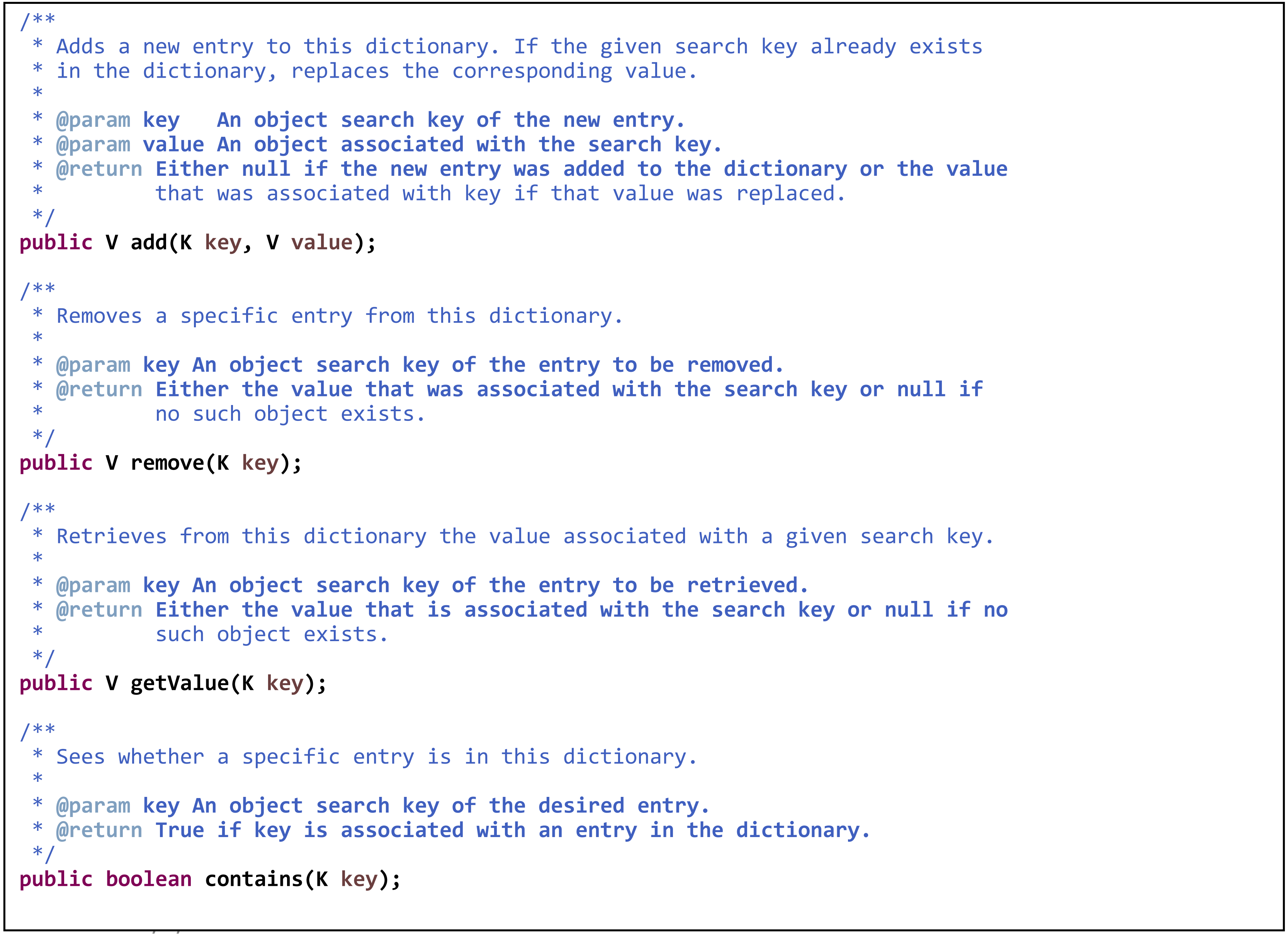
© 2019 Pearson Education, Inc.

FIGURE 20-2 An instance of the ADT dictionary has search keys paired with corresponding values

Specifications for the ADT Dictionary

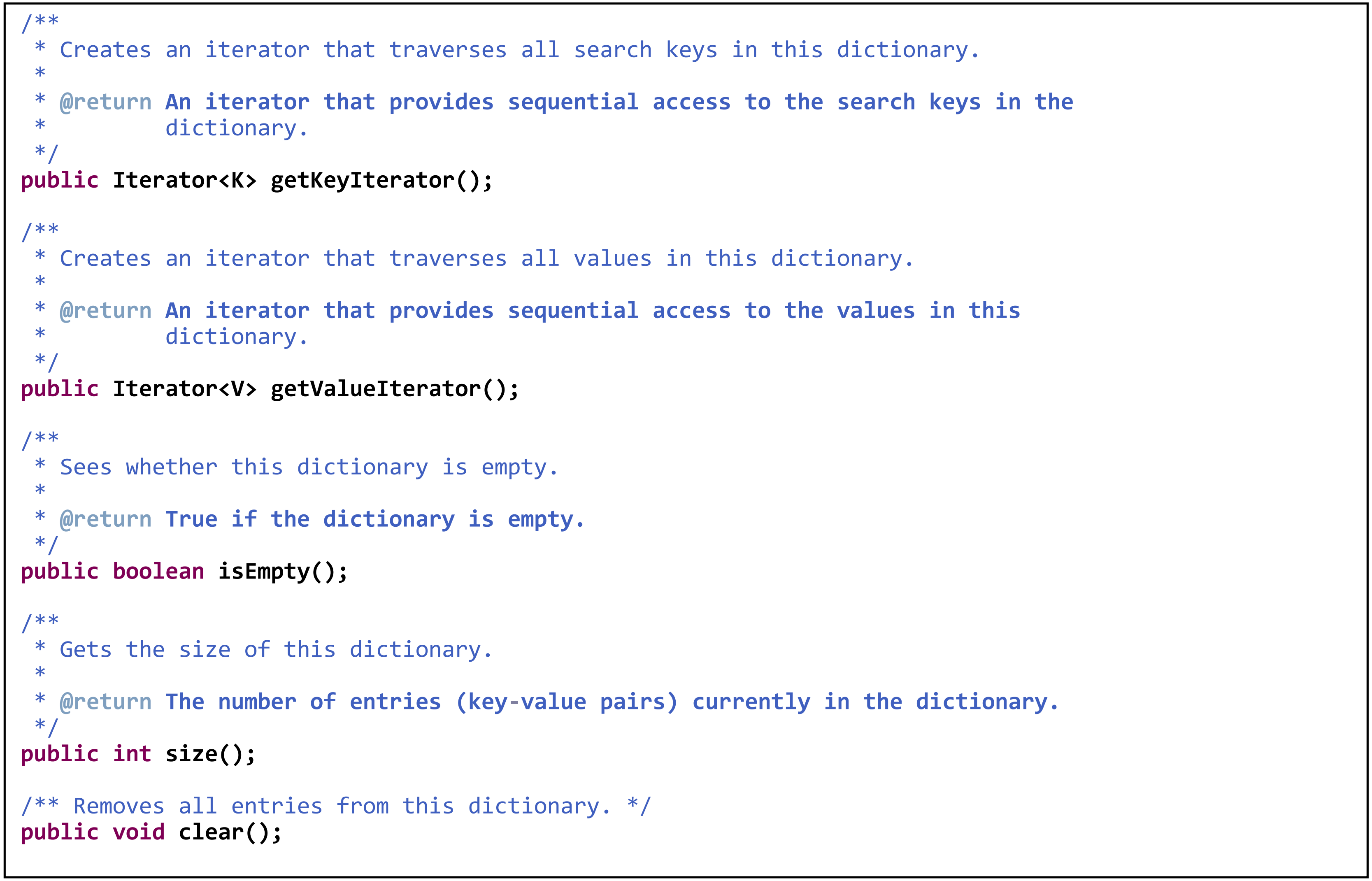
- Operations
  - `add(key, value)`
  - `remove(key)`
  - `getValue(key)` – look up in Dictionary
  - `contains(key)`
  - `getKeyIterator()`
  - `getValueIterator()`
  - `isEmpty()`
  - `size()`
  - `clear()`

Dictionary Interface



```
/**
 * Adds a new entry to this dictionary. If the given search key already exists
 * in the dictionary, replaces the corresponding value.
 *
 * @param key   An object search key of the new entry.
 * @param value An object associated with the search key.
 * @return Either null if the new entry was added to the dictionary or the value
 *         that was associated with key if that value was replaced.
 */
public V add(K key, V value);

/**
 * Removes a specific entry from this dictionary.
 *
 * @param key An object search key of the entry to be removed.
 * @return Either the value that was associated with the search key or null if
 *         no such object exists.
 */
public V remove(K key);

/**
 * Retrieves from this dictionary the value associated with a given search key.
 *
 * @param key An object search key of the entry to be retrieved.
 * @return Either the value that is associated with the search key or null if no
 *         such object exists.
 */
public V getValue(K key);

/**
 * Sees whether a specific entry is in this dictionary.
 *
 * @param key An object search key of the desired entry.
 * @return True if key is associated with an entry in the dictionary.
 */
public boolean contains(K key);
```

Dictionary Interface



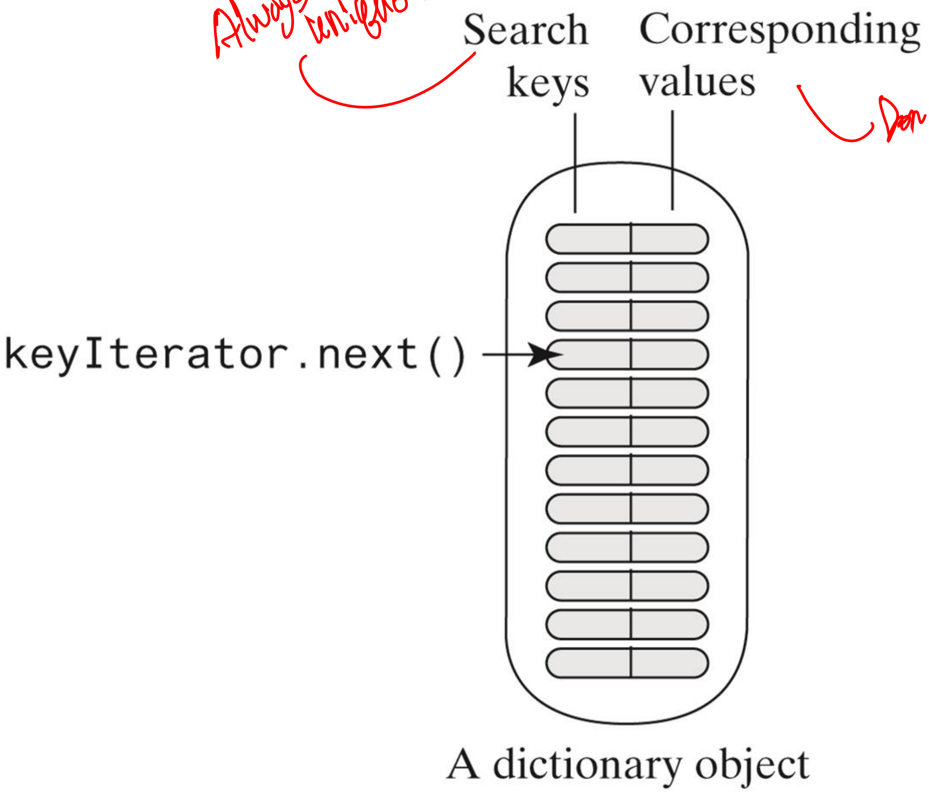
```java
/**
 * Creates an iterator that traverses all search keys in this dictionary.
 *
 * @return An iterator that provides sequential access to the search keys in the
 *         dictionary.
 */
public Iterator<K> getKeyIterator();

/**
 * Creates an iterator that traverses all values in this dictionary.
 *
 * @return An iterator that provides sequential access to the values in this
 *         dictionary.
 */
public Iterator<V> getValueIterator();

/**
 * Sees whether this dictionary is empty.
 *
 * @return True if the dictionary is empty.
 */
public boolean isEmpty();

/**
 * Gets the size of this dictionary.
 *
 * @return The number of entries (key-value pairs) currently in the dictionary.
 */
public int size();

/** Removes all entries from this dictionary. */
public void clear();
```

Dictionary Iterators

- Options for dictionary iterators
  - Can use each of these iterators either separately or together to traverse:
    - All search keys in a dictionary without traversing values
    - All values without traversing search keys
    - All search keys and all values in parallel

```
Iterator<String> keyIterator = dataBase.getKeyIterator();
Iterator<Student> valueIterator = dataBase.getValueIterator();
```

Traversing a dictionary’s keys separately



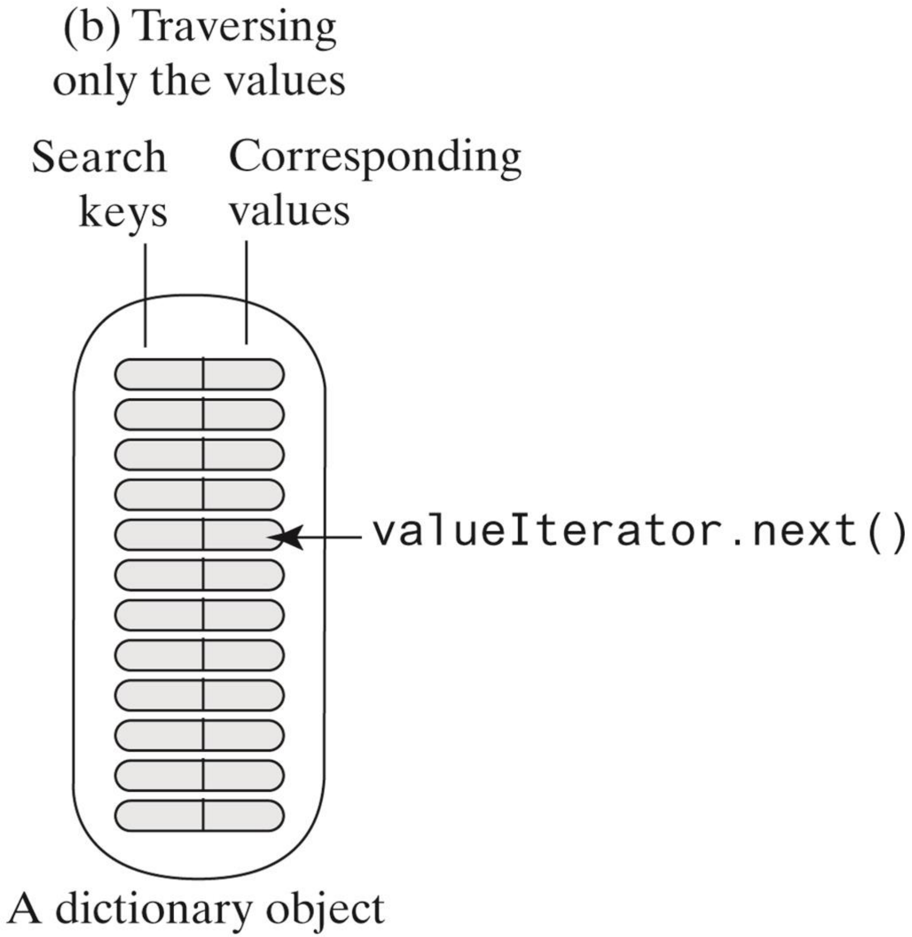
(a) Traversing only the search keys

Always unique!

Don't have to be unique!

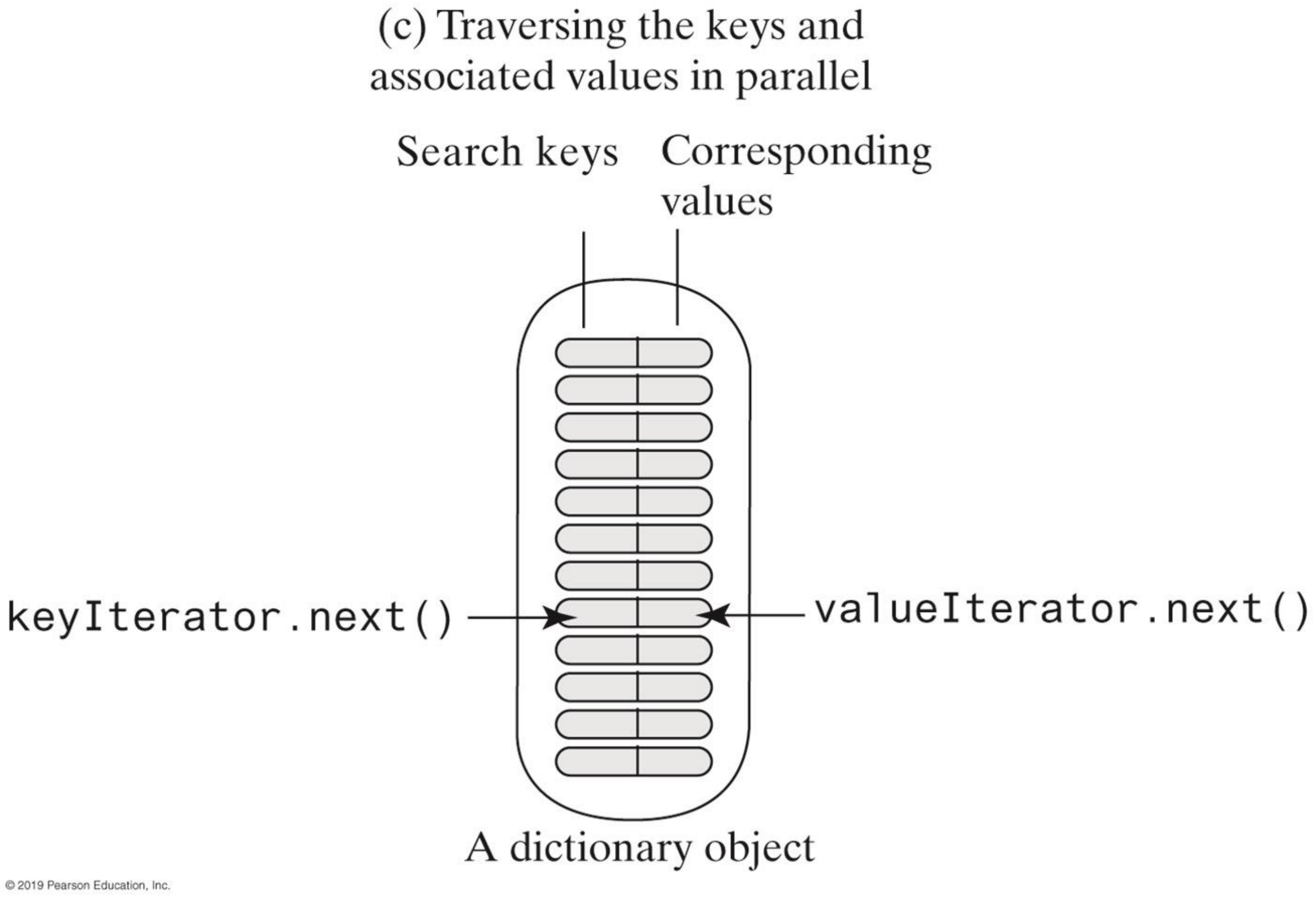
© 2019 Pearson Education, Inc.

Traversing a dictionary’s values separately



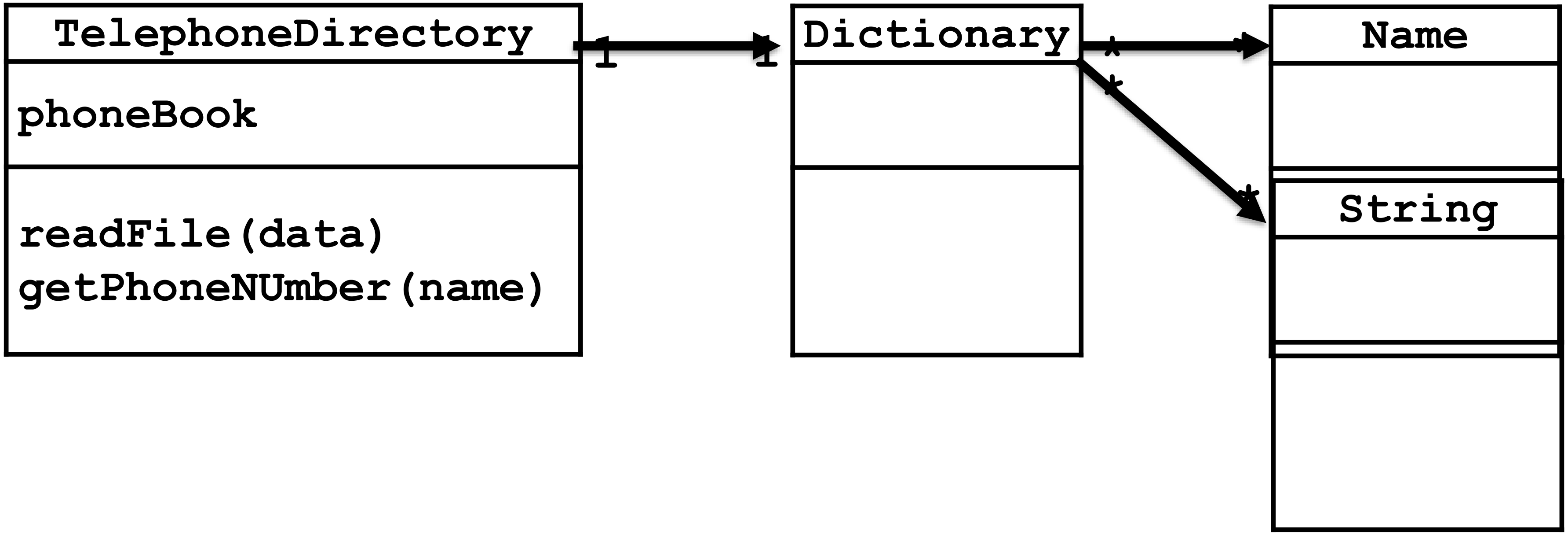
© 2019 Pearson Education, Inc.

Traverse keys/values in parallel



Using a Dictionary

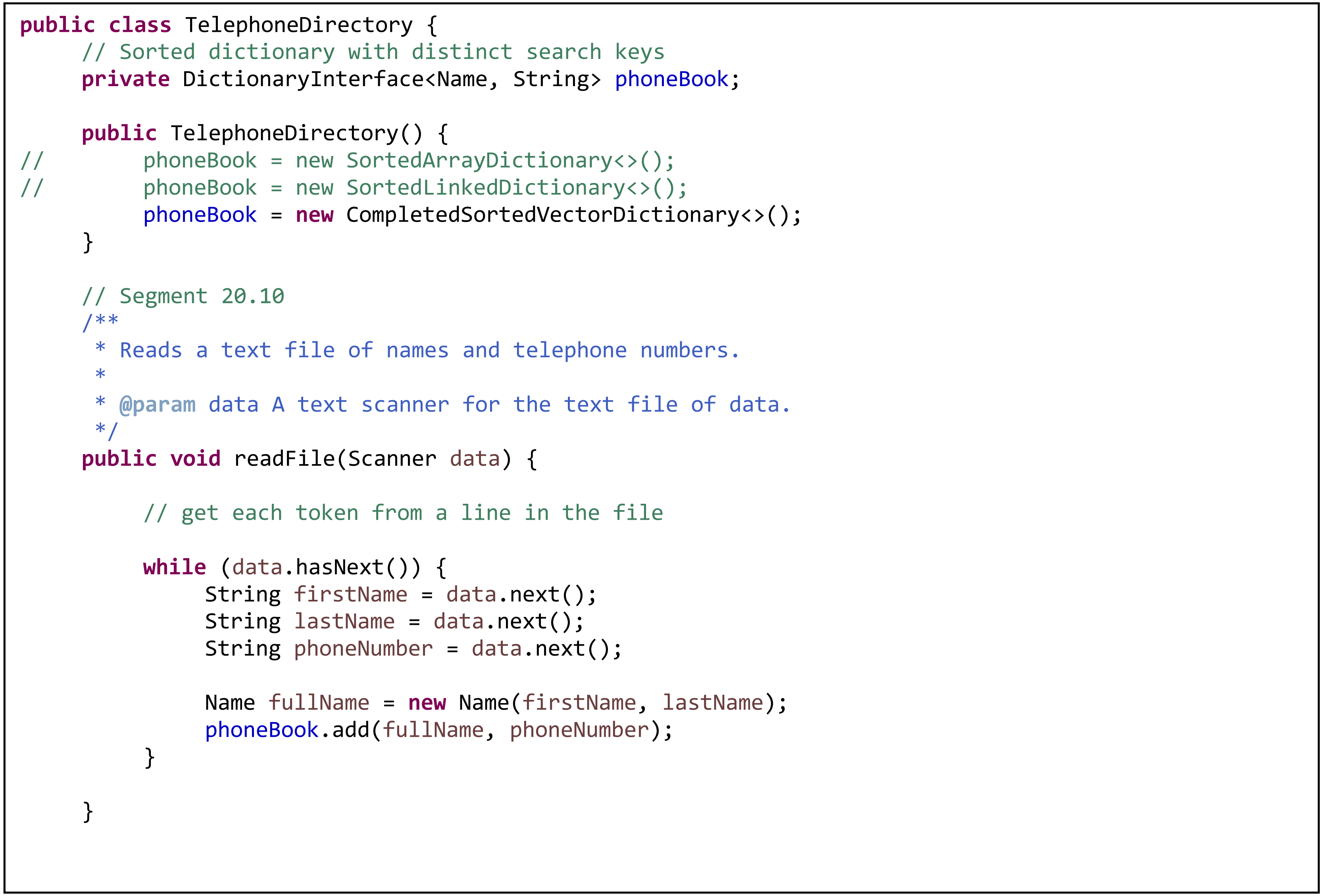
A class diagram for a telephone directory



Telephone Directory Example

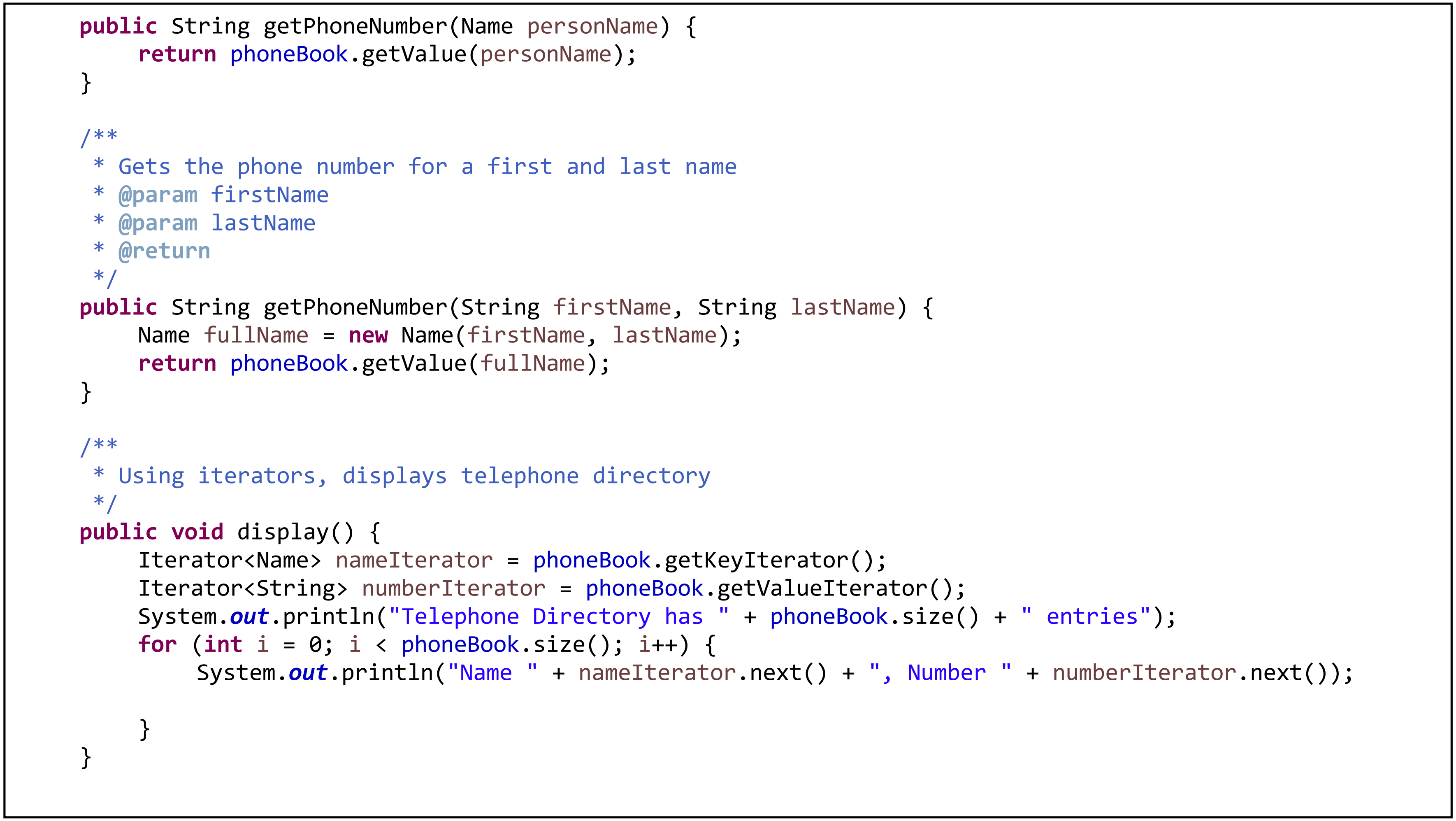
- Telephone directory consist of names and phone numbers.
  - In this example, the file TelephoneDirectoryData.txt has first and last name and a phone number
- TelephoneDirectory class
  - readFile() – reads names and numbers from the file and stores into a sorted directory
  - other methods look up the phone number given a name
  - Uses SortedArrayDictionary but can use any sorted dictionary.
- TelephoneDirectoryDemo
  - reads from the file
  - prompts the user for a name, then looks up the number and returns it

TelephoneDirectory readFile()



```java
public class TelephoneDirectory {
    // Sorted dictionary with distinct search keys
    private DictionaryInterface<Name, String> phoneBook;

    public TelephoneDirectory() {
//      phoneBook = new SortedArrayDictionary<>();
//      phoneBook = new SortedLinkedDictionary<>();
        phoneBook = new CompletedSortedVectorDictionary<>();
    }

    // Segment 20.10
    /**
     * Reads a text file of names and telephone numbers.
     *
     * @param data A text scanner for the text file of data.
     */
    public void readFile(Scanner data) {

        // get each token from a line in the file

        while (data.hasNext()) {
            String firstName = data.next();
            String lastName = data.next();
            String phoneNumber = data.next();

            Name fullName = new Name(firstName, lastName);
            phoneBook.add(fullName, phoneNumber);
        }

    }
```

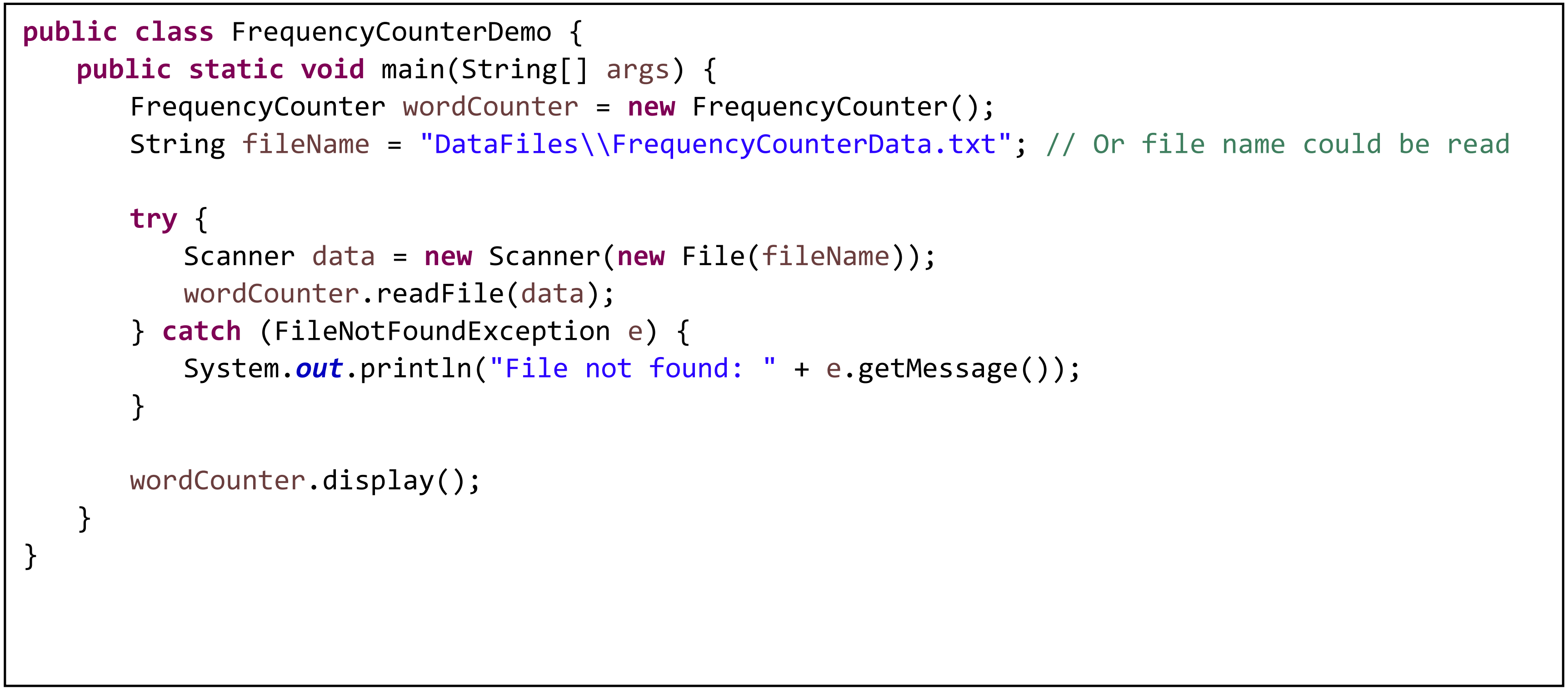
TelephoneDirectory – getPhoneNumber(), display()



```
    public String getPhoneNumber(Name personName) {
        return phoneBook.getValue(personName);
    }

    /**
     * Gets the phone number for a first and last name
     * @param firstName
     * @param lastName
     * @return
     */
    public String getPhoneNumber(String firstName, String lastName) {
        Name fullName = new Name(firstName, lastName);
        return phoneBook.getValue(fullName);
    }

    /**
     * Using iterators, displays telephone directory
     */
    public void display() {
        Iterator<Name> nameIterator = phoneBook.getKeyIterator();
        Iterator<String> numberIterator = phoneBook.getValueIterator();
        System.out.println("Telephone Directory has " + phoneBook.size() + " entries");
        for (int i = 0; i < phoneBook.size(); i++) {
            System.out.println("Name " + nameIterator.next() + ", Number " + numberIterator.next());

        }
    }
```

Word Frequency Counter Example

- Read a file and count the frequency of each word.
- FrequencyCounter



```java
public class FrequencyCounterDemo {
    public static void main(String[] args) {
        FrequencyCounter wordCounter = new FrequencyCounter();
        String fileName = "DataFiles\\FrequencyCounterData.txt"; // Or file name could be read

        try {
            Scanner data = new Scanner(new File(fileName));
            wordCounter.readFile(data);
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e.getMessage());
        }

        wordCounter.display();
    }
}
```

Frequency Counter Class – readFile()

```
public void readFile(Scanner data) {
    // this means any whitespace, to allow for tokenization
    data.useDelimiter("\\W+");

    // convert each token to lower case, and see if it is in the word table
    // if not, add it with a value of 1, otherwise simply update the frequency

    while (data.hasNext()) {
        String nextWord = data.next();
        nextWord = nextWord.toLowerCase();
        Integer frequency = wordTable.getValue(nextWord);

        if (frequency == null) { // Add new word to table
            wordTable.add(nextWord, Integer.valueOf(1));
        } else { // Increment count of existing word; replace wordTable entry
            frequency++;
            wordTable.add(nextWord, frequency);
        }
    }

    data.close();
}
```

FrequencyCounter – display()

- Notice use of iterators
  - One each for key and value
- Alternative is to iterate through the keys and get each value
  - Performance issue

```
    public void display() {
        Iterator<String> keyIterator = wordTable.getKeyIterator();
        Iterator<Integer> valueIterator = wordTable.getValueIterator();

        while (keyIterator.hasNext()) {

            // iterate through each key, then each value, as they match up

            System.out.println(keyIterator.next() + " " + valueIterator.next());

            // less efficient, get each key, then look up the value

//          String key = keyIterator.next();
//          System.out.println(key + " " + wordTable.getValue(key));
        }
    }
```

Concordance Example

- Concordance provides location of a word (like an index)
- Read in a file
- For each word, put in a dictionary consisting of the word and a list of line numbers the word is on.
- Dictionary<String, ArrayList> is needed. Note that a value can be a list.

```java
public class ConcordanceDemo {
    public static void main(String[] args) {

//      System.out.println("Current relative path is: " + System.getProperty("user.dir"));
        Concordance wordIndex = new Concordance();

        String fileName = "ConcordanceData.txt"; // could be read
        try {
            Scanner textReader = new Scanner(new File(fileName));
            wordIndex.readFile(textReader);
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e.getMessage());
        }

        System.out.println("Here is the concordance for the text read from the data file:");
        wordIndex.display();

        System.out.println("\nTest getLineNumbers(\"learning\")");

        ListWithIteratorInterface<Integer> lineList = wordIndex.getLineNumbers("learning");
        Iterator<Integer> listIterator = lineList.getIterator();
        while (listIterator.hasNext()) {
            System.out.print(listIterator.next() + " ");
        } // end while
        System.out.println();

        System.out.println("\n\nDone!");
    } // end main
} // end Driver
```

Concordance class readFile()

- For each line, scan the line and tokenize it (each word is separated by white space.
- Once we have a word, look it up in the dictionary.
- If it doesn’t exist, add it and a blank line list
- Now add the line number to the line list.

```
public void readFile(Scanner data) {
    int lineNumber = 1;

    while (data.hasNext()) {
        String line = data.nextLine();
        line = line.toLowerCase();

        // read tokens delimited by white space

        Scanner lineProcessor = new Scanner(line);
        lineProcessor.useDelimiter("\\W+");

        while (lineProcessor.hasNext()) {
            String nextWord = lineProcessor.next();
            ListWithIteratorInterface<Integer> lineList = wordTable.getValue(nextWord);

            if (lineList == null) {
                // Create new list for new word; add list and word to index
                lineList = new CompletedAListWithIterator<Integer>();
                wordTable.add(nextWord, lineList);
            } // end if

            // Add line number to end of list so list is sorted
            lineList.add(lineNumber);
        }
        lineProcessor.close();
        lineNumber++;
    }

    data.close();
}
```

Concordance class – display()

- Iterate through the words (keys) and values (line lists).
- Display each word, and then iterate through the line list

```
    public void display() {
        Iterator<String> keyIterator = wordTable.getKeyIterator();
        Iterator<ListWithIteratorInterface<Integer>> valueIterator = wordTable.getValueIterator();
        while (keyIterator.hasNext()) {
            // Display the word
            System.out.print(keyIterator.next() + " ");

            // Get line numbers and iterator
            ListWithIteratorInterface<Integer> lineList = valueIterator.next();
            Iterator<Integer> lineListIterator = lineList.getIterator();

            // Display line numbers
            while (lineListIterator.hasNext()) {
                System.out.print(lineListIterator.next() + " ");
            }

            System.out.println();
        }
    }

    public ListWithIteratorInterface<Integer> getLineNumbers(String word) {
        return wordTable.getValue(word);
    }
```

Java Class Library: The Interface Map

- Method headers for a selection of methods in Map
- Highlighted methods differ from our method implementations.

```
public V put(K key, V value);
public V remove (Object key);
public V get(Object key);
public boolean containsKey(Object key);
public boolean containsKey(Object value);
public Set<K> keySet();
public Collection<V> values();
public boolean isEmpty();
public int size();
public void clear();
```

you can modify.

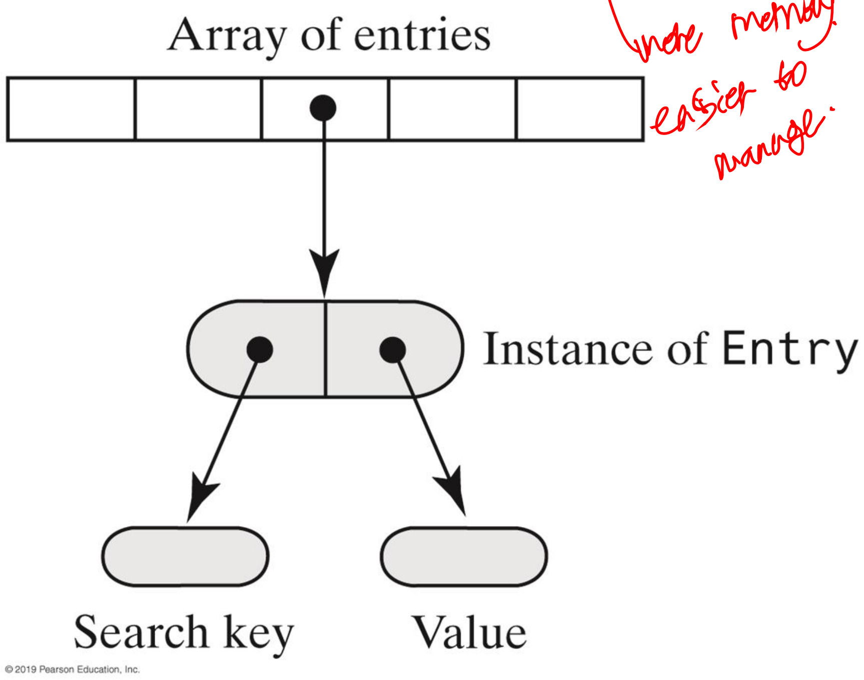
Array-Based Dictionaries

This could be interupted. while process is going.

- Two possible ways to use arrays to represent the entries in a dictionary

(a) An array of objects that encapsulate each search key and corresponding value

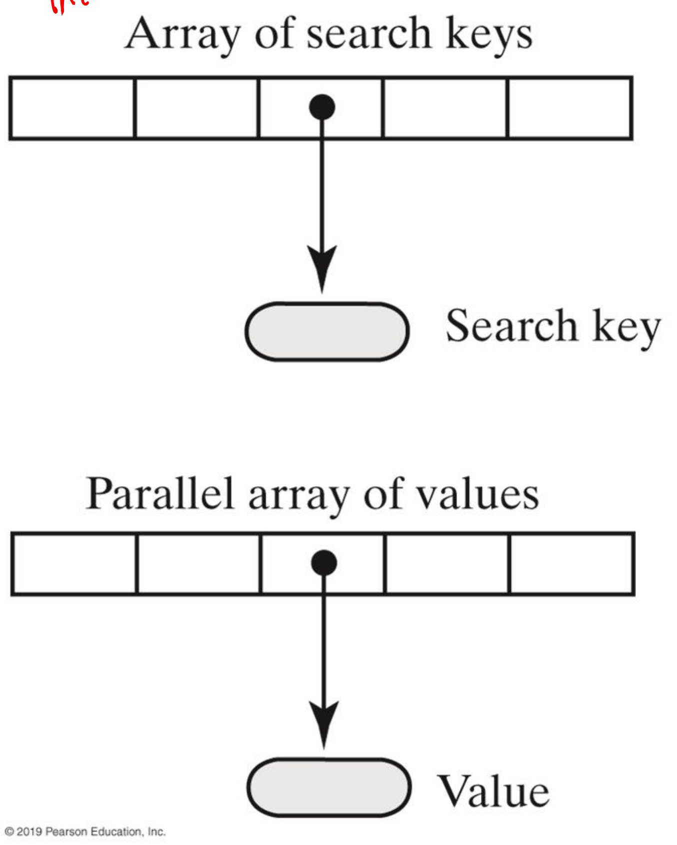
more memory easier to manage.



© 2019 Pearson Education, Inc.

(b) Two arrays in parallel, one of search keys and one of values

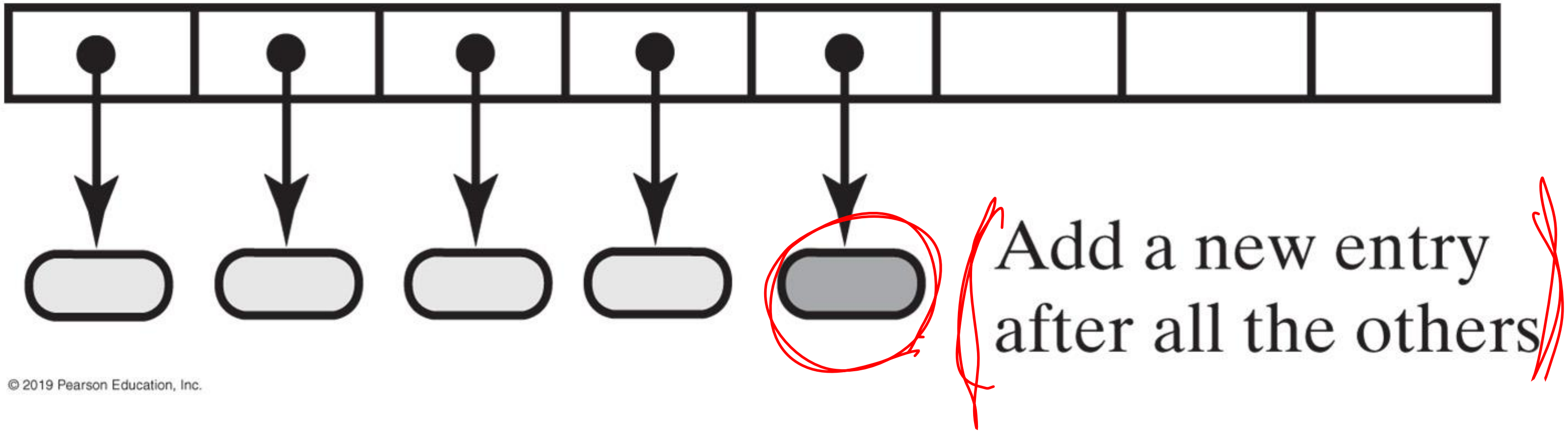
This one is faster.



© 2019 Pearson Education, Inc.

Unsorted Array-Based Implementations

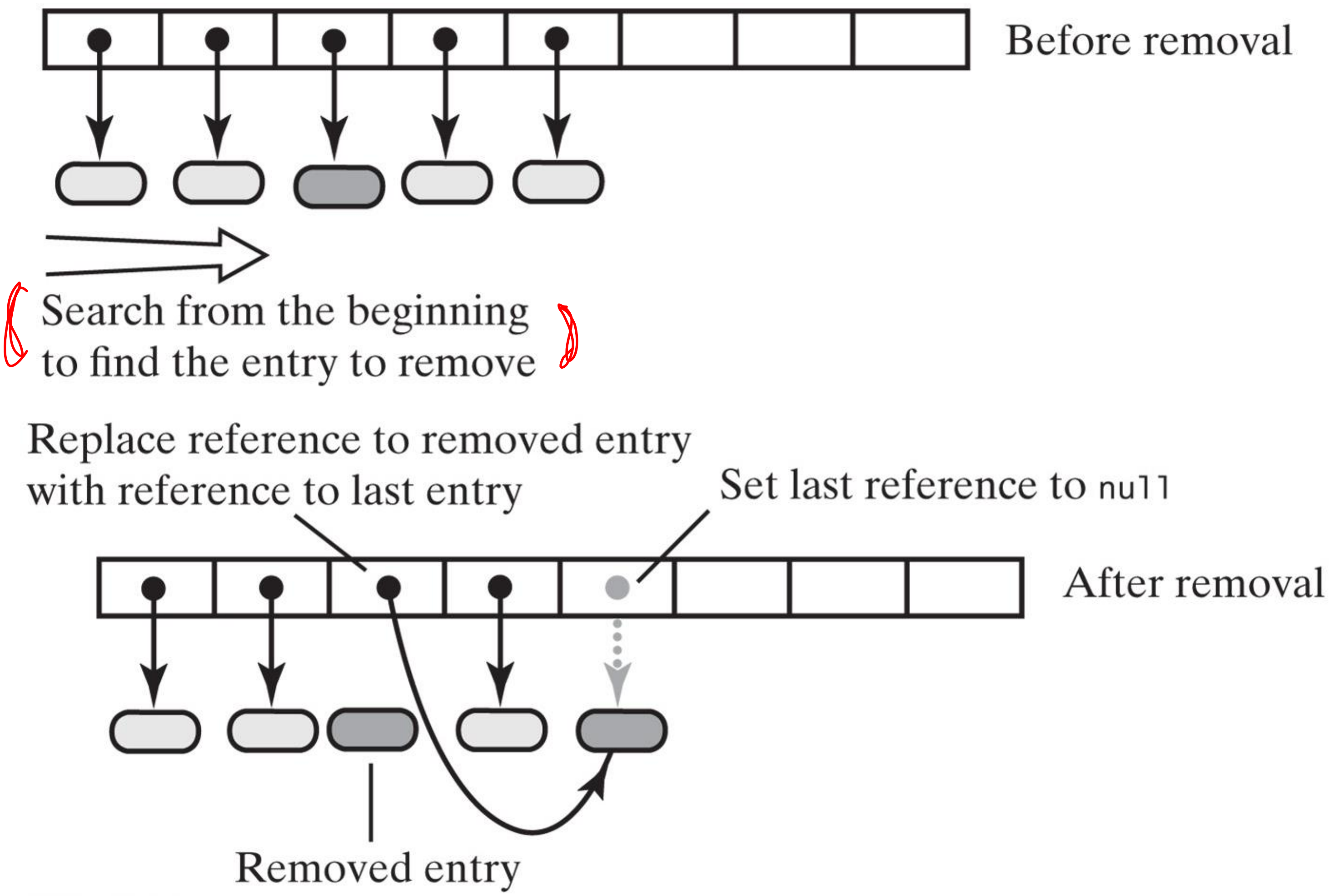
- Adding a new entry to an unsorted array-based dictionary



Add a new entry after all the others

© 2019 Pearson Education, Inc.

Removing an entry from an unsorted array-based dictionary


Before removal
Search from the beginning to find the entry to remove
Replace reference to removed entry with reference to last entry
Set last reference to null
After removal
Removed entry

© 2019 Pearson Education, Inc.

Array-based implementations

- Build an dictionary using an array
  - Requirements
    - Unique keys
    - Array elements have no 'holes' – contiguous
- Three ways to implement
  - ArrayDictionary class – uses internal code
  - VectorDictionary class – uses Java library Vector class internally
  - ListDictionary – uses ListInterface classes AList or LList

ListDictionary

K: where Entry to compare

You have to compare keys.

- Uses ListInterface for internal dictionary
- Implements using AList or LList

for SortedList

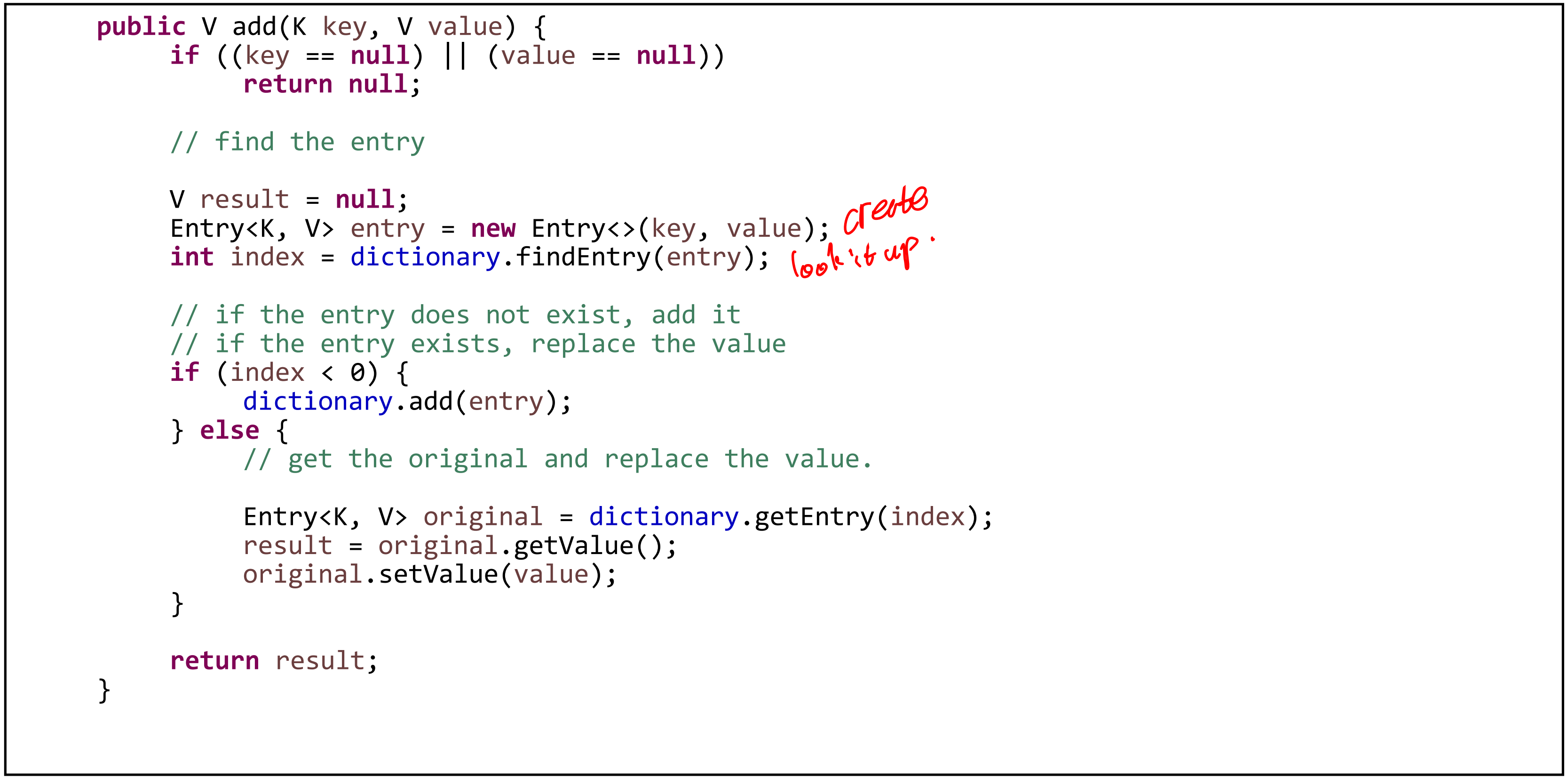
```
public class CompletedListDictionary<K extends Comparable<? super K>, V>
     implements DictionaryInterface<K, V> {

     private final static int DEFAULT_CAPACITY = 25;

     private ListInterface<Entry<K, V>> dictionary;

     public CompletedListDictionary() {
          this(DEFAULT_CAPACITY); // Call next constructor
     }

     public CompletedListDictionary(int initialCapacity) {

          dictionary = new CompletedAList<>(initialCapacity);
//        dictionary = new CompletedLList<>();
     }
```

wildcard

©Michael Hrybyk and Pearson Education NOT TO BE REDISTRIBUTED

ListDictionary – add()

- uses findEntry() and add() from ListInterface
- If entry not found, add it, otherwise replace the value



```
public V add(K key, V value) {
    if ((key == null) || (value == null))
        return null;

    // find the entry

    V result = null;
    Entry<K, V> entry = new Entry<>(key, value); create
    int index = dictionary.findEntry(entry); look it up.

    // if the entry does not exist, add it
    // if the entry exists, replace the value
    if (index < 0) {
        dictionary.add(entry);
    } else {
        // get the original and replace the value.

        Entry<K, V> original = dictionary.getEntry(index);
        result = original.getValue();
        original.setValue(value);
    }

    return result;
}
```

ListDictionary - remove

- findEntry(), and if found, use ListInterface remove()

```
    public V remove(K key) {
        if (key == null)
            return null;

        V result = null;

        // find the entry
        Entry<K, V>“entry“= new Entry<>(key, null); Create.
        int index = dictionary.findEntry(entry);

        // if it exists, remove it
        //   cannot use removeEntry() as the value will not be returned

        if (index >= 0) {
            Entry<K, V> original = dictionary.remove(index);
            result = original.getValue();
        }

        return result;
    }
```

ListDictionary – getValue(), contains()

```
    public V getValue(K key) {
        if (key == null)
            return null;

        V result = null;

        // find the entry

        Entry<K, V> entry = new Entry<>(key, null);
        int index = dictionary.findEntry(entry);

        // if it exists, get the value associated with the key

        if (index >= 0) {
            result = dictionary.getEntry(index).getValue();
        }

        return result;
    }
    public boolean contains(K key) {
        if (key == null)
            return false;

        Entry<K, V> entry = new Entry<>(key, null);

        return dictionary.findEntry(entry) >= 0;
    }
```

ListDictionary – iterators

```
    private class KeyIterator implements Iterator<K> {
        private int currentIndex;

        private KeyIterator() {
            currentIndex = 0;
        }

        public boolean hasNext() {
            return currentIndex < dictionary.size();
        }

        public K next() {
            K result = null;

            if (hasNext()) {
                result = dictionary.getEntry(currentIndex).getKey();
                currentIndex++;
            } else {
                throw new NoSuchElementException();
            }

            return result;
        }

        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
```

ListDictionary – other methods

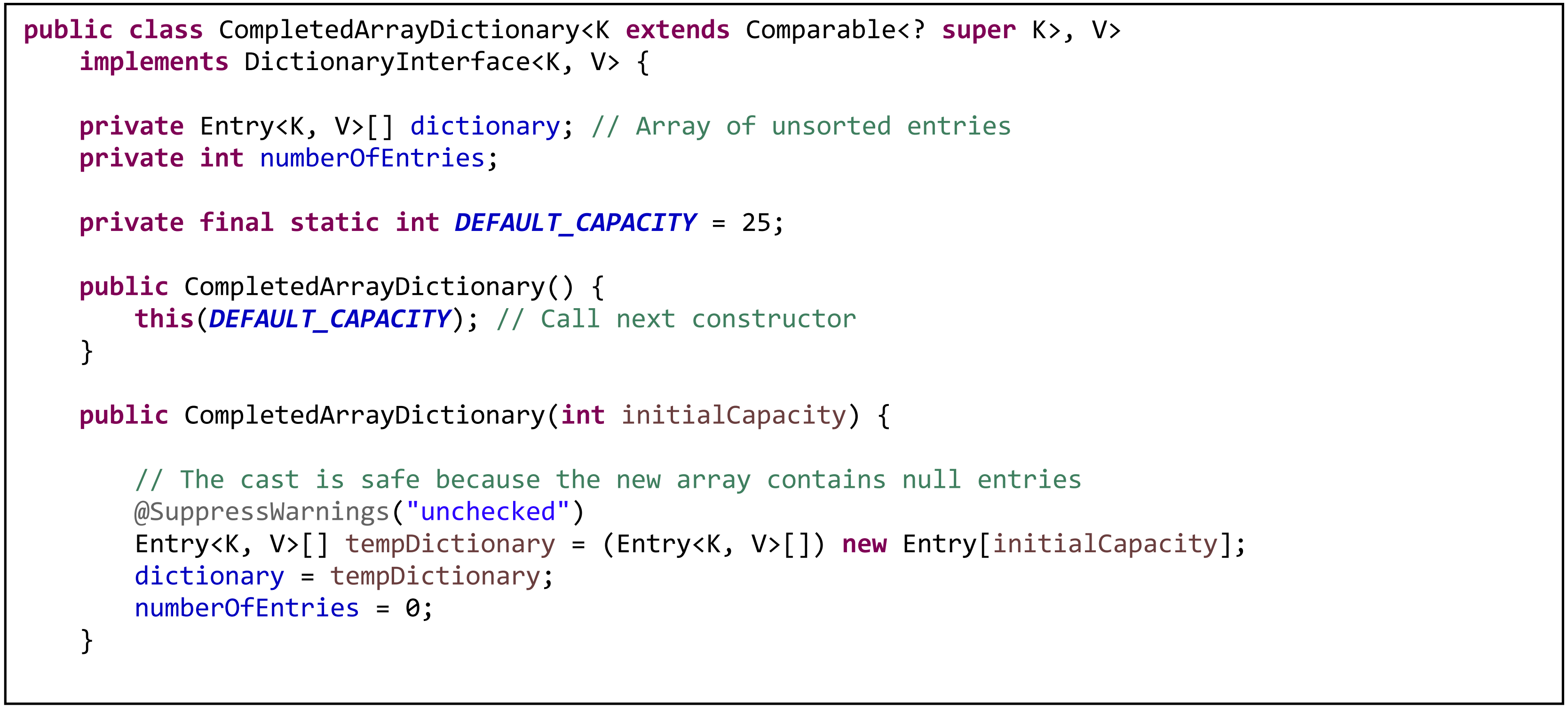
- All other methods simply use the internal dictionary object methods.
  - EG, dictionary.size()
- No need to manage numberOfEntries
  - Done in the internal dictionary object
- ValueIterator is basically the save as KeyIterator
- This scheme fosters reuse

VectorDictionary

- Similar to ListDictionary
- Uses internal Java library array of Vector objects
- Uses Vector methods

ArrayDictionary

- Uses internal code
- Uses an internal array of Entry<K,V>, a set of key/value pairs
  - Use of generics, so the key and value can be anything



```java
public class CompletedArrayDictionary<K extends Comparable<? super K>, V>
    implements DictionaryInterface<K, V> {

    private Entry<K, V>[] dictionary; // Array of unsorted entries
    private int numberOfEntries;

    private final static int DEFAULT_CAPACITY = 25;

    public CompletedArrayDictionary() {
        this(DEFAULT_CAPACITY); // Call next constructor
    }

    public CompletedArrayDictionary(int initialCapacity) {

        // The cast is safe because the new array contains null entries
        @SuppressWarnings("unchecked")
        Entry<K, V>[] tempDictionary = (Entry<K, V>[]) new Entry[initialCapacity];
        dictionary = tempDictionary;
        numberOfEntries = 0;
    }
```

Entry class

- Place to keep the key and the value.
- Use of generics
- Implements Comparable for use in sorted dictionaries
- Provides compareTo() and equals() methods that only compare keys, not values

```java
public class Entry<K extends Comparable<? super K>, V>
    implements Comparable<Entry<K,V>> {

    private K key;
    private V value;

    public Entry(K searchKey, V dataValue) {
        key = searchKey;
        value = dataValue;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }

    public void setValue(V dataValue) {
        value = dataValue;
    }

```

Entry – compareTo()

- only compares keys not values

```
public int compareTo(Entry<K, V> obj) {
    // if only looking for the key, a new entry
    //  with a value of null will have to be used

    // if this is the same object, then we are equal
    if (this == obj)
        return 0;

    // if the object we are comparing is null,
    // then we are higher
    if(obj == null)
        return 1;

    Entry<K, V> other = (Entry<K, V>) obj;

    // if we are null, we are lower
    // if both are null, then return equals

    if (key == null) {
        // null is always lower
        if (other.key != null)
            return -1;
        else return 0;
    }

    // this is a repeat of above for safety

    if(other.key == null)
        return 1;

    // done accounting for nulls, simply return compareTo()

    return key.compareTo(other.key);
}
```

Entry – equals()

- only compares keys, not values

```
    public boolean equals(Object obj) {

        // if only looking for the key, a new entry
        //  with a value of null will have to be used

        // this code is autogenerated by eclipse
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;

        // simply compare the keys for equality
        //  we don't really care about the value
        //  as keys must be unique
        @SuppressWarnings("unchecked")
        Entry<K, V> other = (Entry<K, V>) obj;
        if (key == null) {
            if (other.key != null)
                return false;
        } else if (!key.equals(other.key))
            return false;

        return true;
    }
```

ArrayDictionary – add()

- Find the key
- If it doesn't exist, add it
- If it does, simply replace the value

```
    public V add(K key, V value) {
        if ((key == null) || (value == null))
            return null;
        else {
            V result = null;
            int keyIndex = locateIndex(key); // key cannot be null

            if (keyIndex < numberOfEntries) {
                // Key found, return and replace entry's value
                result = dictionary[keyIndex].getValue(); // Get old value
                dictionary[keyIndex].setValue(value); // Replace value
            }
            else // Key not found; add new entry to dictionary
            {
                // Add at end of array
                dictionary[numberOfEntries] = new Entry<>(key, value);
                numberOfEntries++;
            }

            return result;
        }
    }
```

if does not find, it locates last!

ArrayDictionary – locateIndex()

- Iterate through the array until the key is found

```
/**
 * Returns the array index of the entry that contains key, or
 *    returns numberOfEntries if no such entry exists.
 * Precondition: key is not null.
 * @return
 */
private int locateIndex(K key) {
    // Sequential search
    int index = 0;
    while ((index < numberOfEntries) && !key.equals(dictionary[index].getKey()))
        index++;

    return index;
}
```

Unsorted Array-Based Implementations

Algorithm to describe the remove operation.

```
Algorithm remove(key)
// Removes an entry from the dictionary, given its search key, and returns its value.
// If no such entry exists in the dictionary, returns null.
result = null
Search the array for an entry containing key
if (an entry containing key is found in the array)
{
        result = value currently associated with key
        Replace the entry with the last entry in the array
        Set array element containing last entry to null
        Decrement the size of the dictionary
}
// Else result is null
return result
```

ArrayDictionary – remove()

- Find the key and remove it
- Same as AList

```java
public V remove(K key) {
    V result = null;
    int keyIndex = locateIndex(key);

    if (keyIndex < numberOfEntries) {
        // Key found; remove entry and return its value
        result = dictionary[keyIndex].getValue();
        // Replace removed entry with last entry
        dictionary[keyIndex] = dictionary[numberOfEntries - 1];
        dictionary[numberOfEntries - 1] = null;
        numberOfEntries--;
    }

    return result;
}
```

ArrayDictionary – other methods

- Need getIterator() for Key and Value
- Other methods like clear(), getSize(), getValue() are straightforward.

ArrayDictionary - iterators

- Keep a current index
- next() increments the index and returns the key
- KeyIterator is below, Value will work the same way (needs a class)

```
private class KeyIterator implements Iterator<K> {
     private int currentIndex;

     private KeyIterator() {
          currentIndex = 0;
     }

     public boolean hasNext() {
          return currentIndex < numberOfEntries;
     }

     public K next() {
          K result = null;

          if (hasNext()) {
               Entry<K, V> currentEntry = dictionary[currentIndex];
               result = currentEntry.getKey();
               currentIndex++;
          } else {
               throw new NoSuchElementException();
          }

          return result;
     }
}
```

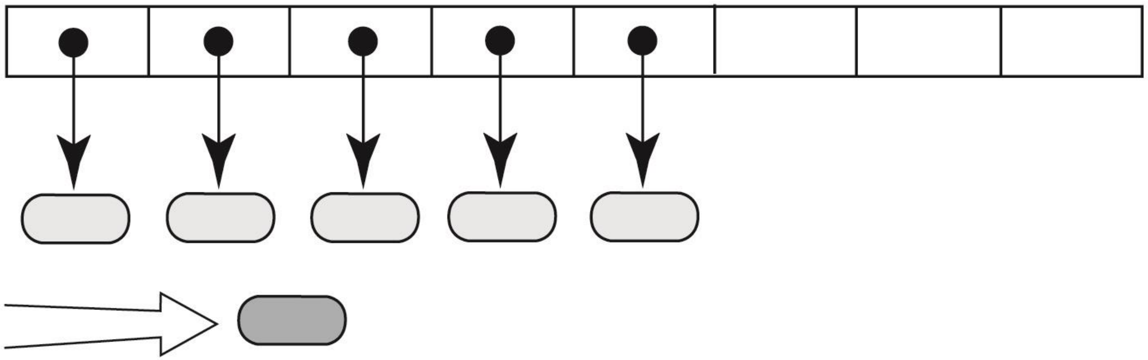
Unsorted Array-Based Implementations

- For this implementation, worst-case efficiencies of the operations are:
  - Addition: $O(n)$
  - Removal: $O(n)$
  - Retrieval: $O(n)$
  - Traversal: $O(n)$

Sorted Array-Based Implementations

- Adding an entry to a sorted array-based dictionary

(a) Locate where to add an entry

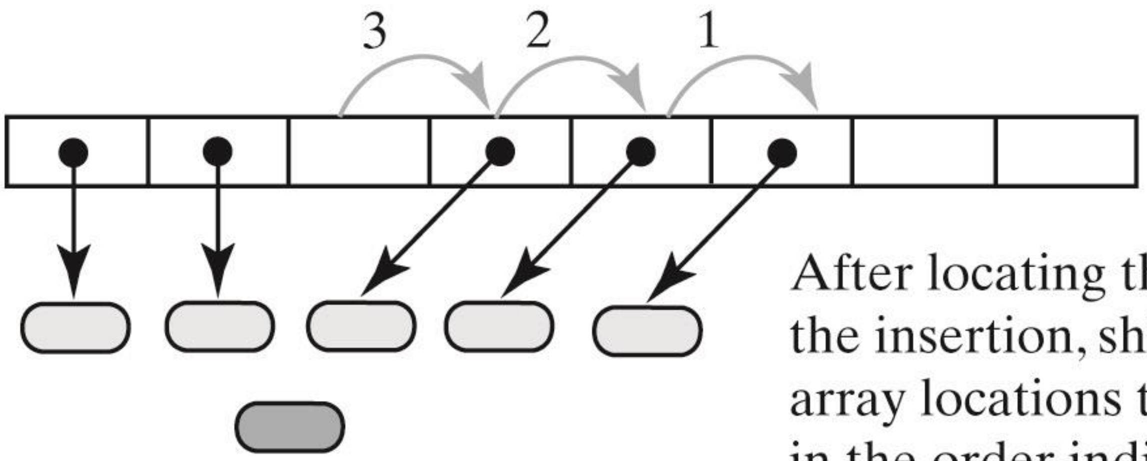


Search from the beginning to find the correct position for a new entry

Sorted Array-Based Implementations

- Adding an entry to a sorted array-based dictionary

(b) Make room for the new entry

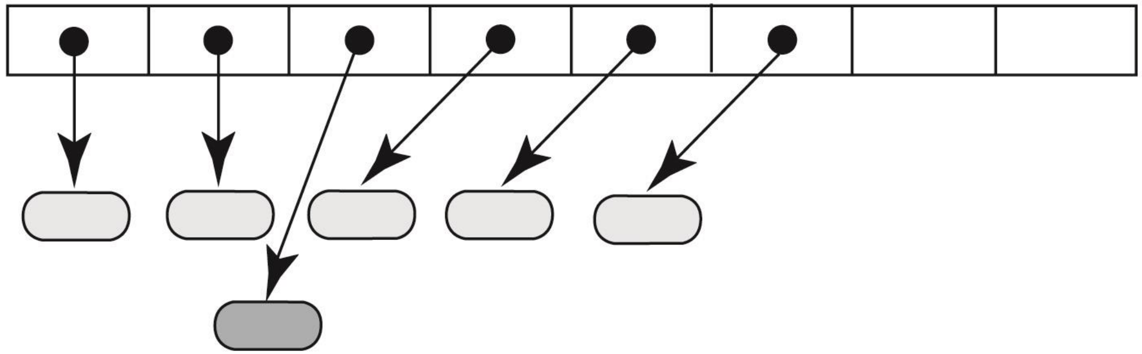


After locating the correct position for the insertion, shift the contents of subsequent array locations toward the end of the array in the order indicated

Sorted Array-Based Implementations

- Adding an entry to a sorted array-based dictionary

(c) Complete the insertion



© 2019 Pearson Education, Inc.

Sorted Array-based implementations

- Three ways to implement
  - SortedArrayDictionary class – uses internal code
  - SortedVectorDictionary class – uses Java library Vector object internally
  - SortedListDictionary class – uses ListInterface object SortedAList
    - same code as ListDictionary

SortedArrayDictionary

- Constructor uses Entry[] justl like in unsorted case

```
public class CompletedSortedArrayDictionary<K extends Comparable<? super K>, V>
   implements DictionaryInterface<K, V> {

   private Entry<K, V>[] dictionary; // Array of entries sorted by search key
   private int numberOfEntries;

   private final static int DEFAULT_CAPACITY = 25; // 6 is for testing

   public CompletedSortedArrayDictionary() {
      this(DEFAULT_CAPACITY);
   } // end default constructor

   public CompletedSortedArrayDictionary(int initialCapacity) {

      // The cast is safe because the new array contains null entries
      @SuppressWarnings("unchecked")
      Entry<K, V>[] tempDictionary = (Entry<K, V>[]) new Entry[initialCapacity];
      dictionary = tempDictionary;
      numberOfEntries = 0;
   }
```

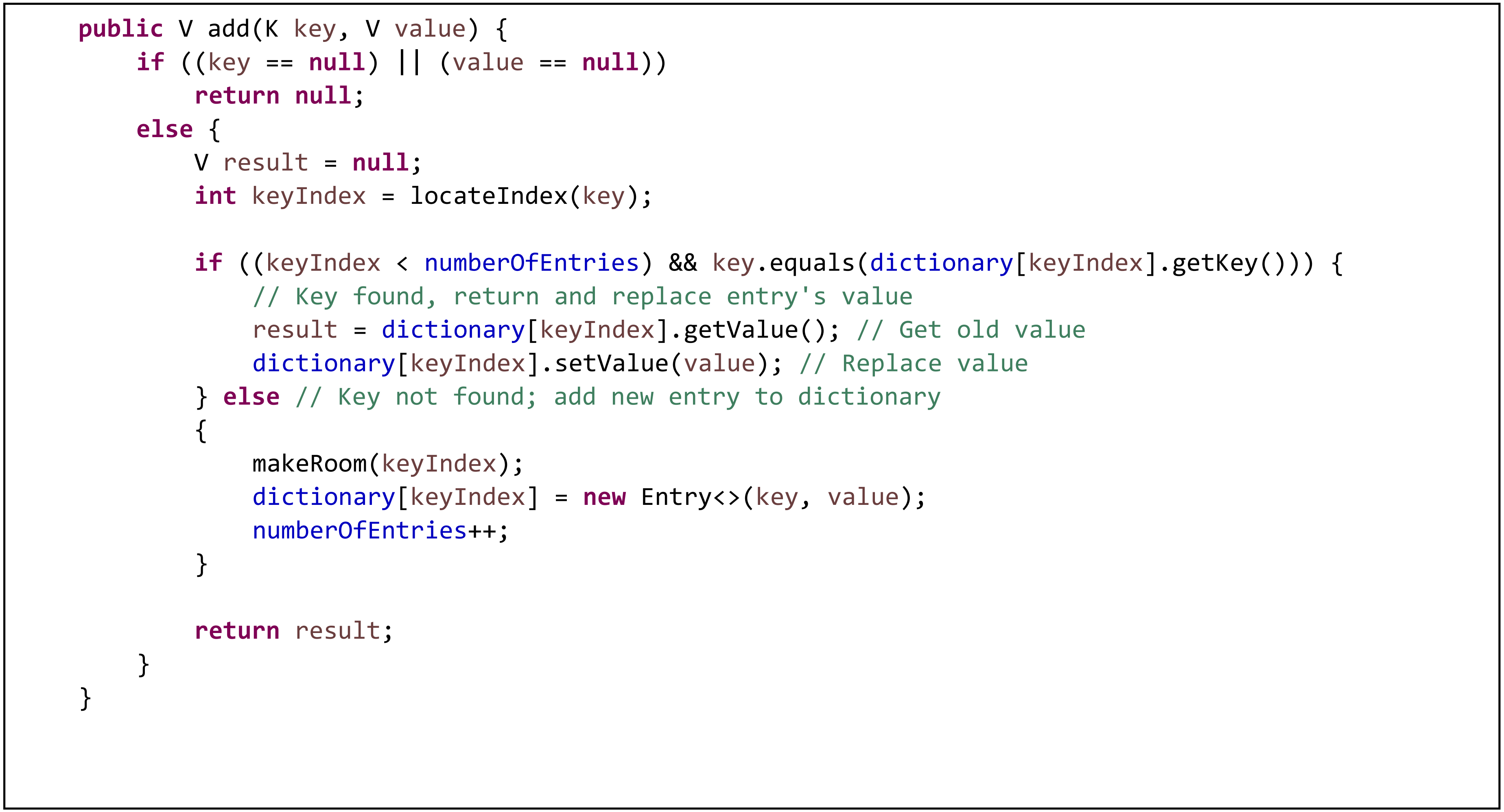
Sorted Array-Based Dictionary

Algorithm for adding an entry

```
Algorithm add(key, value)
// Adds a new key-value entry to the dictionary and returns null. If key already exists
// in the dictionary, returns the corresponding value and replaces it with value.
If either key or value is null, or array is full throw an exception
result = null
Search the array until you either find an entry containing key or locate the point where it should be
if (an entry containing key is found in the array)
{
    result = value currently associated with key
    Replace key's associated value with value
}
else // Insert new entry
{
    Make room in the array for a new entry at the index indicated by the previous search
    Insert a new entry containing key and value into the vacated location of the array
    Increment the size of the dictionary

}
return result
```

SortedArrayDictionary – add()

- Need to be able to insert a key in a position
- Use of makeRoom()



```
public V add(K key, V value) {
    if ((key == null) || (value == null))
        return null;
    else {
        V result = null;
        int keyIndex = locateIndex(key);

        if ((keyIndex < numberOfEntries) && key.equals(dictionary[keyIndex].getKey())) {
            // Key found, return and replace entry's value
            result = dictionary[keyIndex].getValue(); // Get old value
            dictionary[keyIndex].setValue(value); // Replace value
        } else // Key not found; add new entry to dictionary
        {
            makeRoom(keyIndex);
            dictionary[keyIndex] = new Entry<>(key, value);
            numberOfEntries++;
        }

        return result;
    }
}
```

SortedArrayDictionary – locateIndex(), getValue()

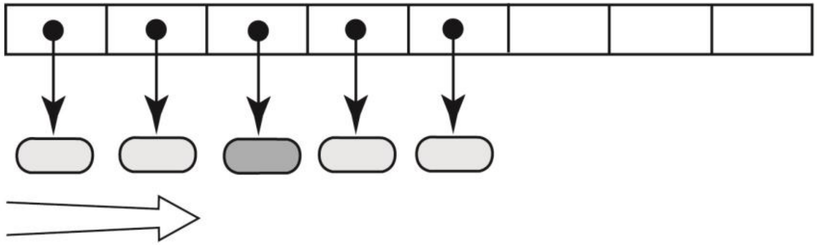
- locateIndex() - uses of compareTo(), which allows placing a new entry in the spot right before the current one

```
    private int locateIndex(K key) {
        // Search until an entry is found containing key or
        // pass the point where it should be
        int index = 0;
        while ((index < numberOfEntries) && key.compareTo(dictionary[index].getKey()) > 0) {
            index++;
        }

        return index;
    }

    public V getValue(K key) {
        V result = null;
        int keyIndex = locateIndex(key);

        // position is found, if key matches, return the value
        if ((keyIndex < numberOfEntries) && key.equals(dictionary[keyIndex].getKey())) {
            result = dictionary[keyIndex].getValue(); // Key found; return value
        }

        return result;
    }
```

Sorted Array-Based Dictionary

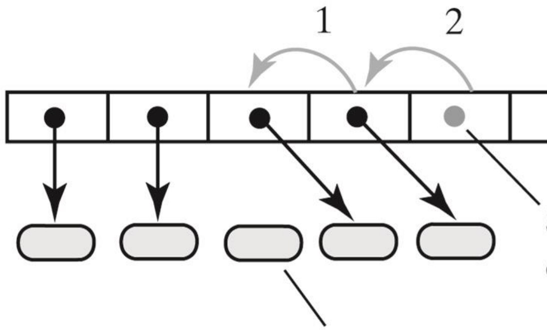
- Removing an entry from a sorted array-based dictionary

(a) Locate entry to remove



Search from the beginning to find the entry to remove

(b) Shift entries toward the one to remove



Set reference to `null`, after its original value is shifted

To remove this entry, shift the contents of subsequent array locations toward the beginning of the array in the order indicated

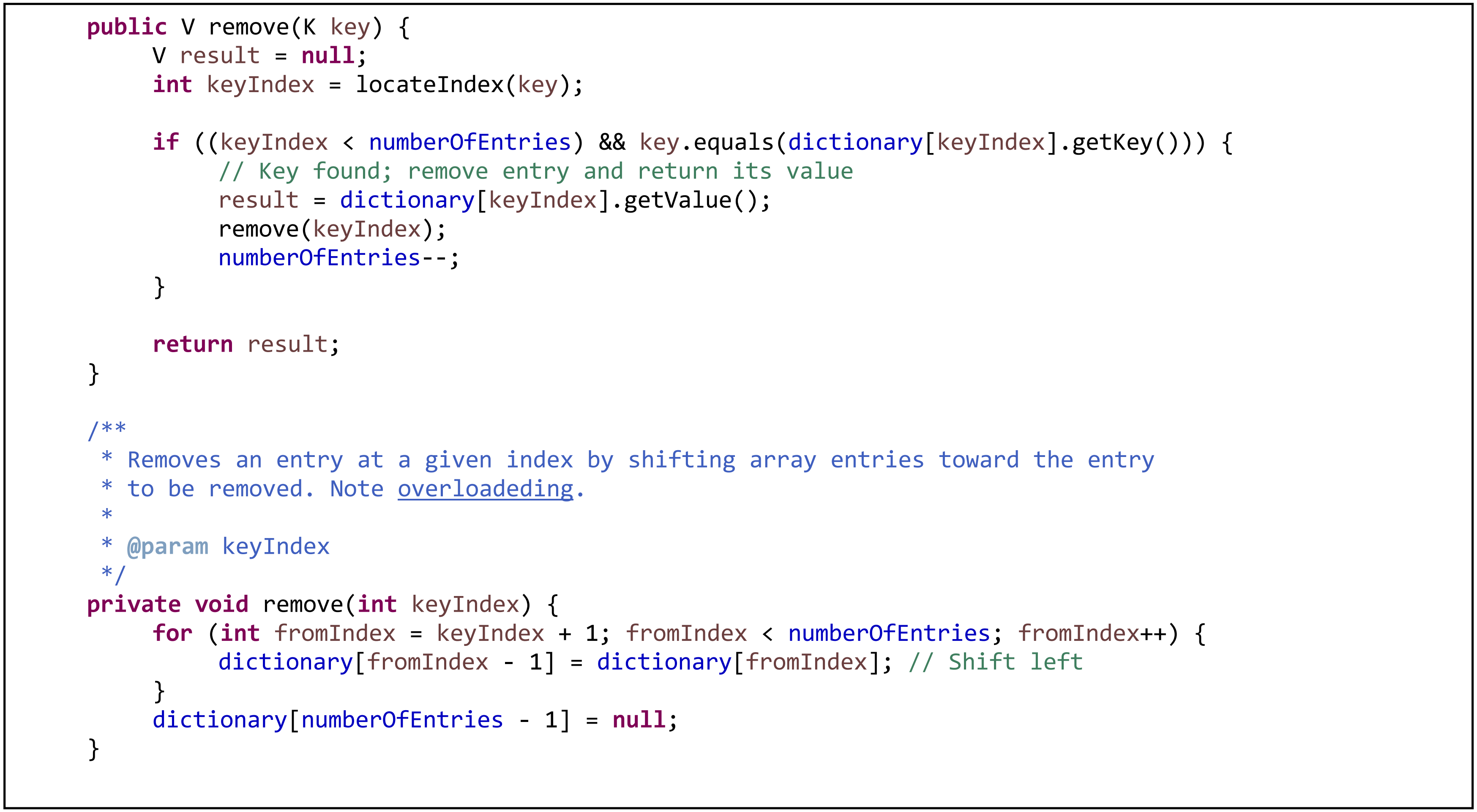
Sorted Array-Based Dictionary

Algorithm that describes the `remove` operation

```
Algorithm remove(key)
// Removes an entry from the dictionary, given its search key, and returns its value.
// If no such entry exists in the dictionary, returns null.
result = null
Search the array for an entry containing key
if (an entry containing key is found in the array)
{
        result = value currently associated with key
        Shift any entries that are after the located one to the next lower
            position in the array
        Set array element that had contained last entry to null
        Decrement the size of the dictionary
}
return result
```

SortedArrayDictionary – remove()

- Needs to remove an entry from the array by shifting down



```java
public V remove(K key) {
    V result = null;
    int keyIndex = locateIndex(key);

    if ((keyIndex < numberOfEntries) && key.equals(dictionary[keyIndex].getKey())) {
        // Key found; remove entry and return its value
        result = dictionary[keyIndex].getValue();
        remove(keyIndex);
        numberOfEntries--;
    }

    return result;
}

/**
 * Removes an entry at a given index by shifting array entries toward the entry
 * to be removed. Note overloadeding.
 *
 * @param keyIndex
 */
private void remove(int keyIndex) {
    for (int fromIndex = keyIndex + 1; fromIndex < numberOfEntries; fromIndex++) {
        dictionary[fromIndex - 1] = dictionary[fromIndex]; // Shift left
    }
    dictionary[numberOfEntries - 1] = null;
}
```

SortedArrayDictionary – other methods

- All other methods are exactly the same as ArrayDictionary
- Iterators are also the same

Efficiency of sorted array-based dictionary

- When **`locateIndex`** uses a binary search in the sorted array-based implementation, the worst-case efficiencies are:
  - Addition: O(n)
  - Removal: O(n)
  - Retrieval: O(log n)
  - Traversal: O(n)

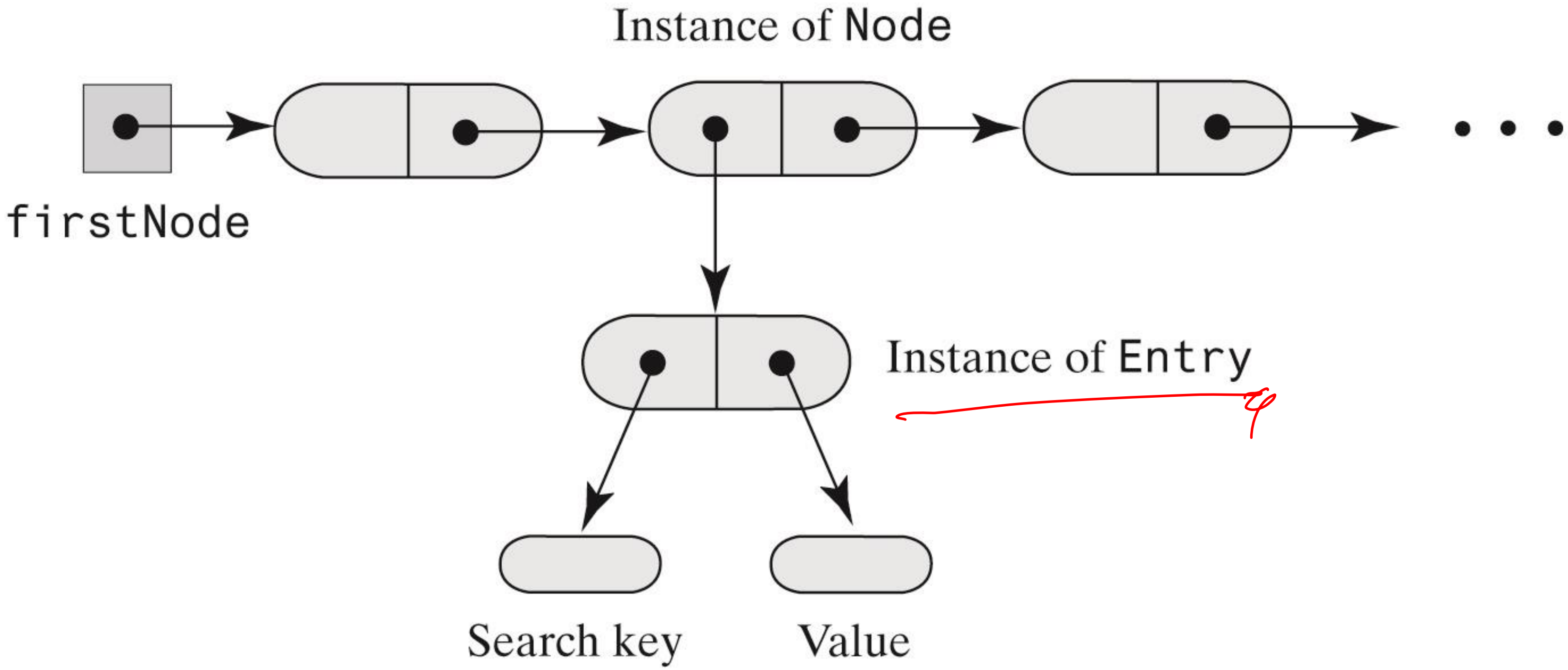
Linked Dictionary implementations

- Ways to implement
  - LinkedDictionary class – uses internal code
  - ListDictionary – uses internal ListInterface object LList
  - Can also used Java library Collections that use the List abstract class (not done here)

Linked Dictionary Implementations

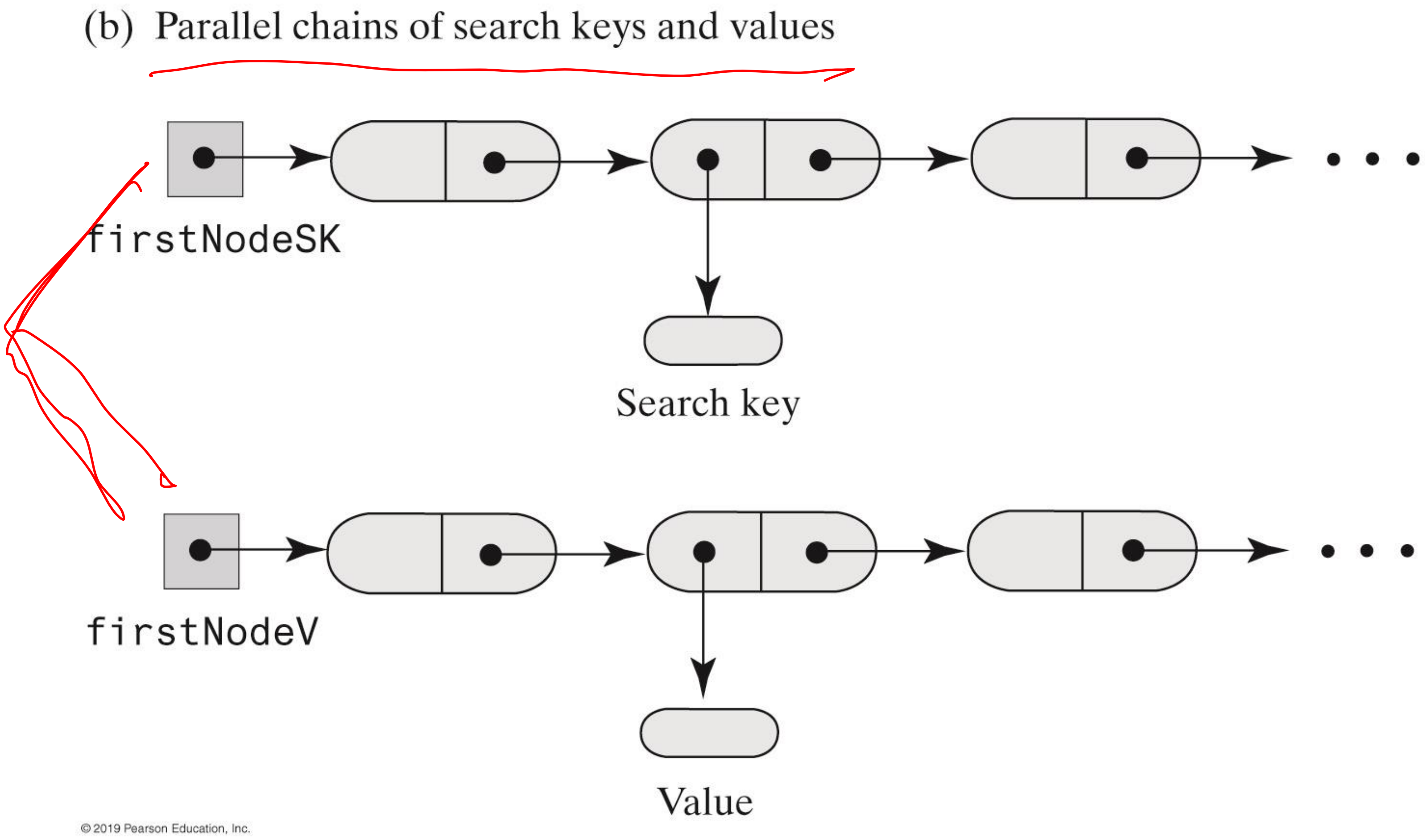
- Representing the entries in a dictionary
- This will be used in examples
- Use of Entry object that contains a key and value

(a) A chain of nodes that each reference an entry object



Linked Dictionary Implementations

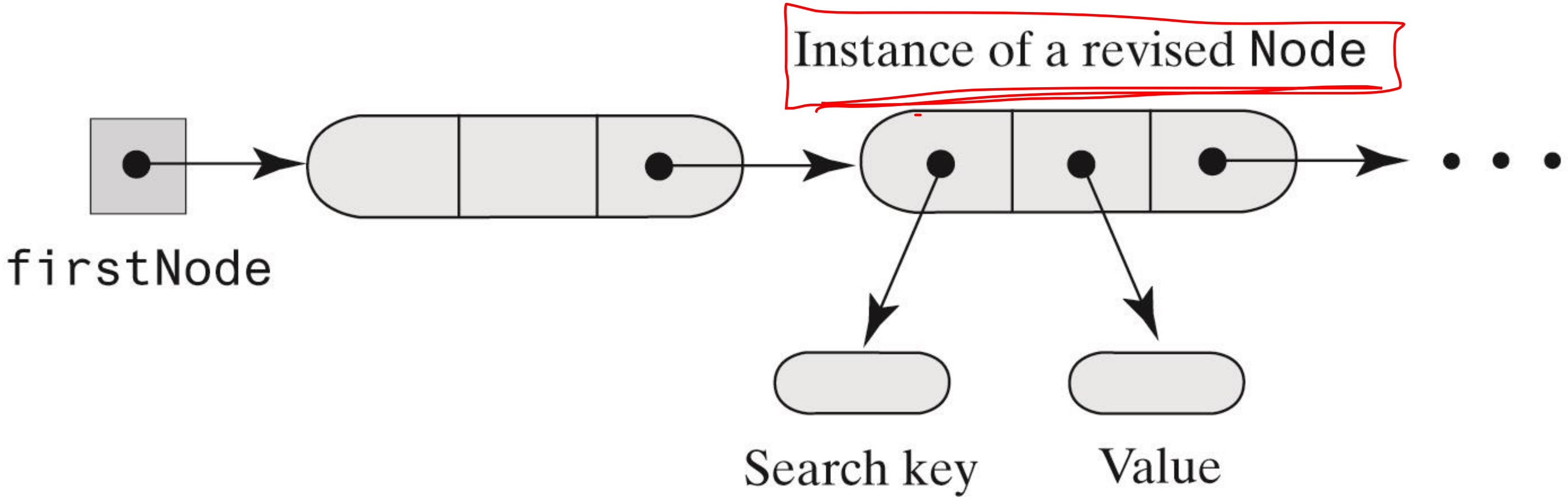
- Representing the entries in a dictionary


(b) Parallel chains of search keys and values
firstNodeSK
Search key
firstNodeV
Value
© 2019 Pearson Education, Inc.

Linked Dictionary Implementations

- Representing the entries in a dictionary

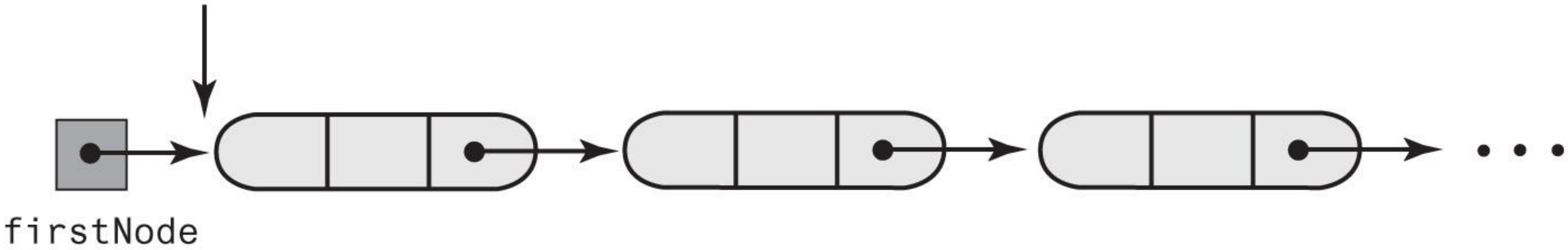
(c) A chain of nodes that each reference a search key and a value



Linked Dictionary Implementations

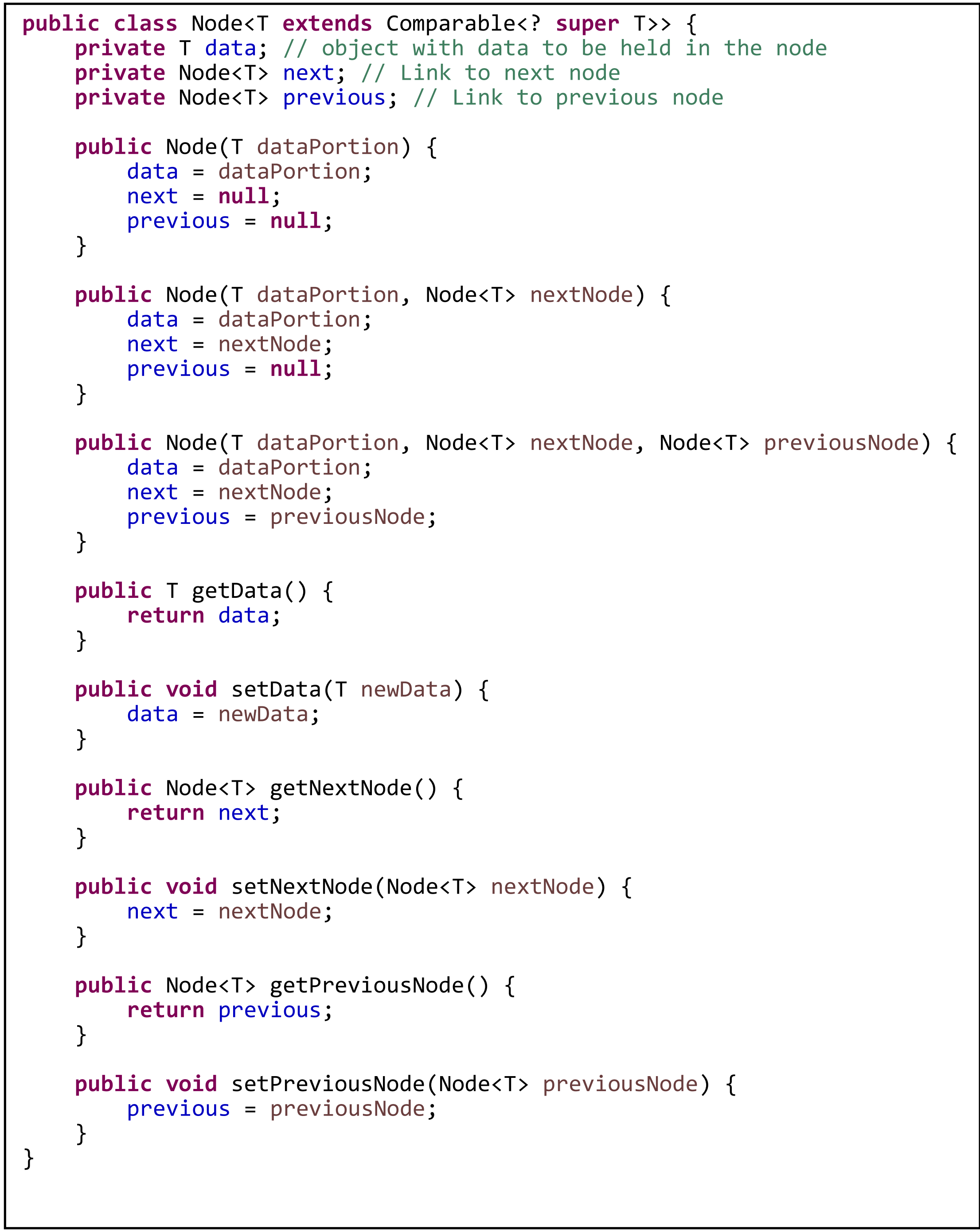
- Adding to an unsorted linked dictionary

Insert a new node at the beginning of the chain



Node class

- Used to store Entry objects
- `Node<Entry<K, V>> firstNode`
- Node is doubly linked with next and previous defined
- Examples will only use next, demonstrating use of single forward link.



```
public class Node<T extends Comparable<? super T>> {
    private T data; // object with data to be held in the node
    private Node<T> next; // Link to next node
    private Node<T> previous; // Link to previous node

    public Node(T dataPortion) {
        data = dataPortion;
        next = null;
        previous = null;
    }

    public Node(T dataPortion, Node<T> nextNode) {
        data = dataPortion;
        next = nextNode;
        previous = null;
    }

    public Node(T dataPortion, Node<T> nextNode, Node<T> previousNode) {
        data = dataPortion;
        next = nextNode;
        previous = previousNode;
    }

    public T getData() {
        return data;
    }

    public void setData(T newData) {
        data = newData;
    }

    public Node<T> getNextNode() {
        return next;
    }

    public void setNextNode(Node<T> nextNode) {
        next = nextNode;
    }

    public Node<T> getPreviousNode() {
        return previous;
    }

    public void setPreviousNode(Node<T> previousNode) {
        previous = previousNode;
    }
}
```

LinkedDictionary - constructors

- Uses Nodes with an Entry obect.
- firstNode is head of list

```
public class CompletedLinkedDictionary<K extends Comparable<? super K>, V>
    implements DictionaryInterface<K, V> {

    private Node<Entry<K,V>> firstNode; // Reference to first node of chain
    private int numberOfEntries;

    public CompletedLinkedDictionary() {
        initializeDataFields();
    }

    /**
     * Initializes the class's data fields to indicate an empty list.
     */
    private void initializeDataFields() {
        firstNode = null;
        numberOfEntries = 0;
    }
```

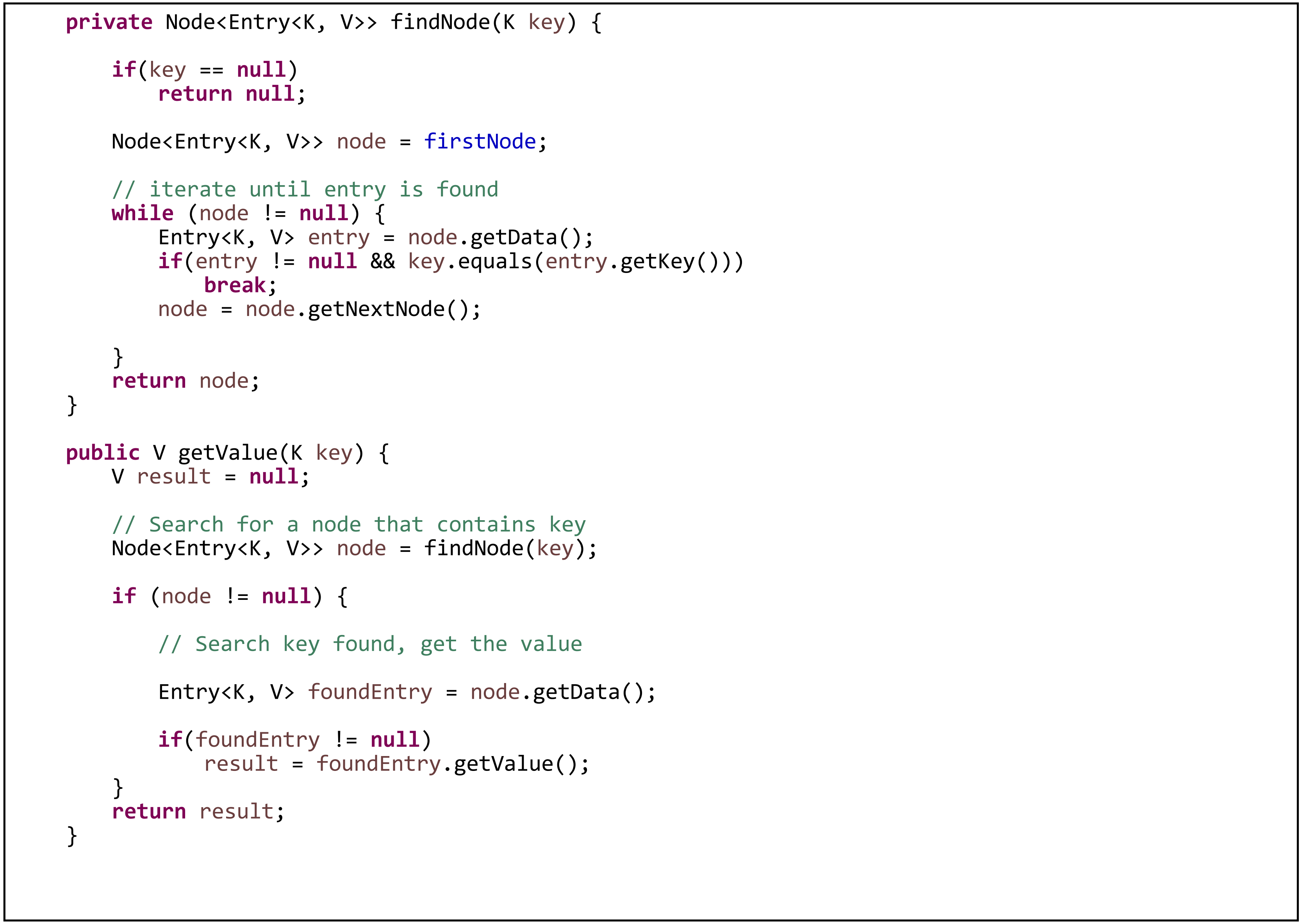
LinkedDictionary – add()

```
    public V add(K key, V value) {
        if ((key == null) || (value == null))
            return null;
        else {
            V result = null;

            Entry<K, V> entry = new Entry<>(key, value);

            // Search chain for a node containing key
            Node<Entry<K, V>> node = findNode(key);

            if (node == null) {
                // Key not in dictionary; add new node at beginning of chain
                Node<Entry<K, V>> newNode = new Node<>(entry);
                newNode.setNextNode(firstNode);
                firstNode = newNode;
                numberOfEntries++;
            } else {
                // Key in dictionary; replace corresponding value
                Entry<K, V> original = node.getData();
                result = original.getValue(); // Get ready to return removed entry
                original.setValue(value); // Replace value
            }

            return result;
        }
    }
```

LinkedDictionary – remove()

```
    public V remove(K key) {
        V result = null; // Return value

        if (!isEmpty()) {
            // Search chain for a node containing key;
            // save reference to preceding node

            Entry<K, V> entry = new Entry<>(key, null);
            Node<Entry<K, V>> currentNode = firstNode;
            Node<Entry<K, V>> nodeBefore = null;

            while ((currentNode != null) && !entry.equals(currentNode.getData())) {
                nodeBefore = currentNode;
                currentNode = currentNode.getNextNode();
            }

            if (currentNode != null) {
                // node found; remove it

                // Node after the one to be removed
                Node<Entry<K, V>> nodeAfter = currentNode.getNextNode();

                // if there was no preceding node, then the current node is actually firstNode
                //  so set the firstNode to the one following

                if (nodeBefore == null)
                    firstNode = nodeAfter;
                else
                    nodeBefore.setNextNode(nodeAfter); // Disconnect the node to be removed

                result = currentNode.getData().getValue();
                numberOfEntries--;
            }
        }

        return result;
    }
```

LinkedDictionary – findNode(), getValue()



```java
    private Node<Entry<K, V>> findNode(K key) {

        if(key == null)
            return null;

        Node<Entry<K, V>> node = firstNode;

        // iterate until entry is found
        while (node != null) {
            Entry<K, V> entry = node.getData();
            if(entry != null && key.equals(entry.getKey()))
                break;
            node = node.getNextNode();

        }
        return node;
    }

    public V getValue(K key) {
        V result = null;

        // Search for a node that contains key
        Node<Entry<K, V>> node = findNode(key);

        if (node != null) {

            // Search key found, get the value

            Entry<K, V> foundEntry = node.getData();

            if(foundEntry != null)
                result = foundEntry.getValue();
        }
        return result;
    }
```

LinkedDictionary – iterators and other methods

- Key and Value iterators keep track of next node
- other methods are very simple

```
private class KeyIterator implements Iterator<K> {
    Node<Entry<K, V>> nextNode;

    private KeyIterator() {
        nextNode = firstNode;
    }

    public boolean hasNext() {
        return nextNode != null;
    }

    public K next() {
        K result;

        if (hasNext()) {
            result = nextNode.getData().getKey();
            nextNode = nextNode.getNextNode();
        } else {
            throw new NoSuchElementException();
        }

        return result;
    }
}
```

Sorted Linked Dictionary implementations

- Ways to implement
  - o SortedLinkedDictionary class – uses internal code
  - o SortedListDictionary – uses internal ListInterface object SortedLList
    - same code as ListDictionary
  - o Can also used Java library Collections that use the List abstract class (not done here)

SortedLinkedDictionary - constructor

- Uses public Node and Entry classes
- firstNode is the head of the list

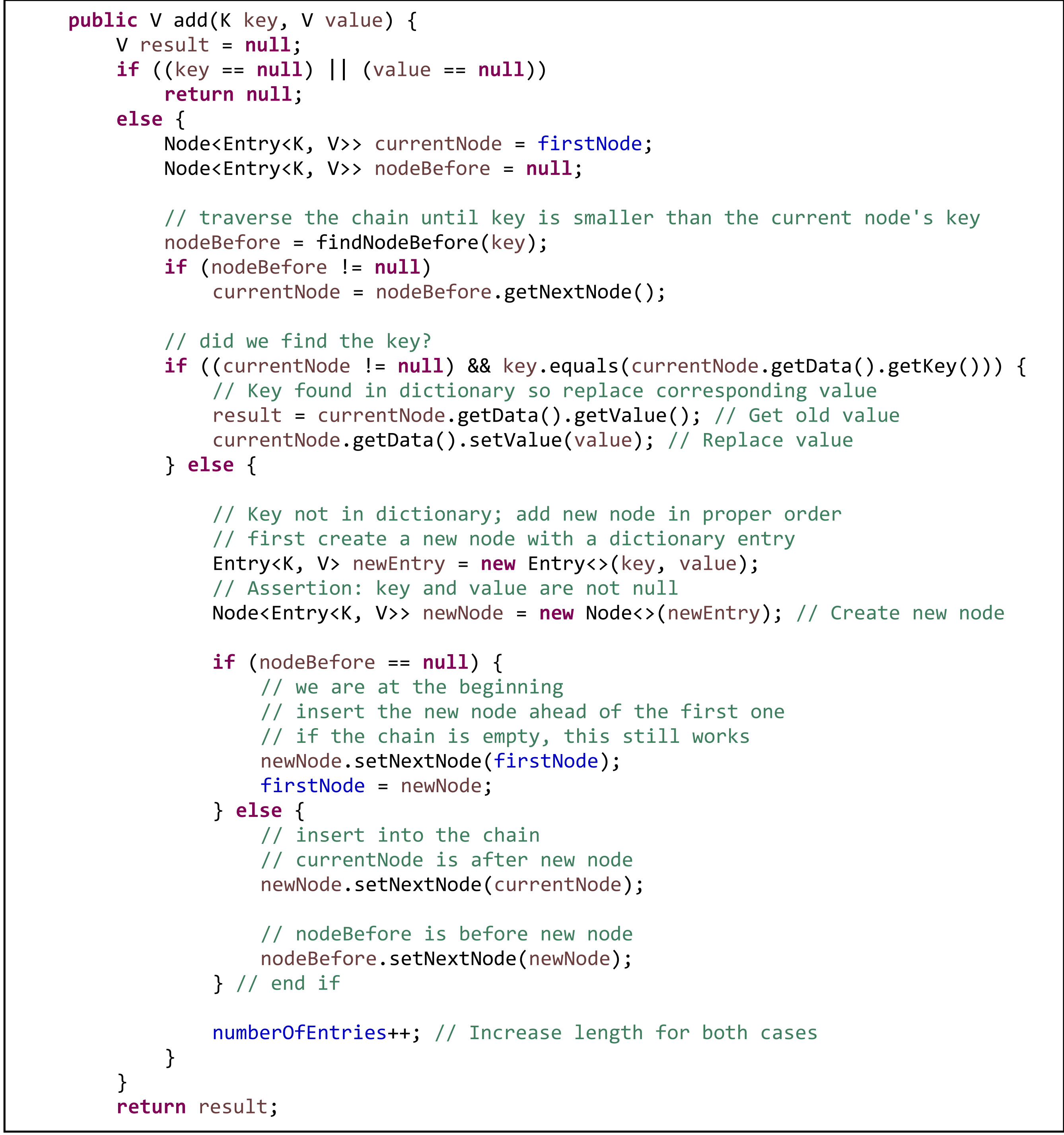
```
public class CompletedSortedLinkedDictionary<K extends Comparable<? super K>, V>
    implements DictionaryInterface<K, V> {

    private Node<Entry<K, V>> firstNode; // Reference to first node of chain
    private int numberOfEntries;

    public CompletedSortedLinkedDictionary() {
        initializeDataFields();
    }

    /**
     * Initializes the class's data fields to indicate an empty list.
     */
    private void initializeDataFields() {
        firstNode = null;
        numberOfEntries = 0;
    }
```

Sorted Linked Dictionary

- Algorithm for adding new entry to sorted linked dictionary

```
Algorithm add(key, value)
// Adds a new key-value entry to the dictionary and returns null. If key already exists
// in the dictionary, returns the corresponding value and replaces that value with value.
If either key or value is null, throw an exception
result = null
Search the chain until either you find a node containing key or you pass the point where it should be
if (a node containing key is found in the chain)
{
    result = value currently associated with key
    Replace key's associated value with value
}
else
{
    Allocate a new node containing key and value
    if (the chain is empty or the new entry belongs at the beginning of the chain)
        Add the new node to the beginning of the chain
    else
        Insert the new node before the last node that was examined during the search Increment the size of the dictionary
}
return result
```

SortedLinkedDictionary – add()



```
public V add(K key, V value) {
    V result = null;
    if ((key == null) || (value == null))
        return null;
    else {
        Node<Entry<K, V>> currentNode = firstNode;
        Node<Entry<K, V>> nodeBefore = null;

        // traverse the chain until key is smaller than the current node's key
        nodeBefore = findNodeBefore(key);
        if (nodeBefore != null)
            currentNode = nodeBefore.getNextNode();

        // did we find the key?
        if ((currentNode != null) && key.equals(currentNode.getData().getKey())) {
            // Key found in dictionary so replace corresponding value
            result = currentNode.getData().getValue(); // Get old value
            currentNode.getData().setValue(value); // Replace value
        } else {

            // Key not in dictionary; add new node in proper order
            // first create a new node with a dictionary entry
            Entry<K, V> newEntry = new Entry<>(key, value);
            // Assertion: key and value are not null
            Node<Entry<K, V>> newNode = new Node<>(newEntry); // Create new node

            if (nodeBefore == null) {
                // we are at the beginning
                // insert the new node ahead of the first one
                // if the chain is empty, this still works
                newNode.setNextNode(firstNode);
                firstNode = newNode;
            } else {
                // insert into the chain
                // currentNode is after new node
                newNode.setNextNode(currentNode);

                // nodeBefore is before new node
                nodeBefore.setNextNode(newNode);
            } // end if

            numberOfEntries++; // Increase length for both cases
        }
    }
    return result;
```

SortedLinkedDictionary – remove()

```
public V remove(K key) {
     if (isEmpty() || key == null)
          return null;

     // find the node before the key
     // to remove, connect nodeBefore to nodeAfter

     Node<Entry<K, V>> currentNode = firstNode;
     Node<Entry<K, V>> nodeBefore = null;

     // look for the previous node to the one with the key
     // then get the next node which should contain the key

     // if the key matches, then disconnect the node

     // this would be easier with a doubly linked list
     // we would just find the node, then link previous to next

     nodeBefore = findNodeBefore(key);

     if (nodeBefore != null)
          currentNode = nodeBefore.getNextNode();

     // if the current node is null, it means the
     // list was traversed to the end without finding the key

     if (currentNode == null)
          return null;

     // determine if the key matches the current node's key

     if (!key.equals(currentNode.getData().getKey()))
          return null;

     // if not at the beginning of the list, disconnect the node to remove

     if (currentNode == firstNode)
          firstNode = firstNode.getNextNode();
     else {
          Node<Entry<K, V>> nodeAfter = currentNode.getNextNode();
          nodeBefore.setNextNode(nodeAfter);
     }

     // save the old value to return to caller
     V result = currentNode.getData().getValue();
     numberOfEntries--;

     return result;
}
```

SortedLinkedDictionary – findNodeBefore()

- Using singly-linked list, to insert, need to get the previous node
    - previous node will provide link to current node
- Would be easier if doubly-linked list was used

```java
private Node<Entry<K, V>> findNodeBefore(K key) {
    if (isEmpty())
        return null;

    Node<Entry<K, V>> currentNode = firstNode;
    Node<Entry<K, V>> nodeBefore = null;

    // iterate until the key is less than or equal to the current node's key
    while (currentNode != null) {
        Entry<K, V> entry = currentNode.getData();
        if (entry != null && key.compareTo(entry.getKey()) <= 0)
            break;
        nodeBefore = currentNode;
        currentNode = currentNode.getNextNode();
    }

    // if not found, it will be null
    // if less than the first node, null will be returned as well

    return nodeBefore;
}
```

SortedLinkedDictionary – other methods

- All other methods are exactly the same as in LinkedDictionary

An Unsorted Linked Dictionary

- Efficiency of an unsorted linked dictionary:
  - The worst-case efficiencies of the operations.
    - Addition: O(n)
    - Removal: O(n)
    - Retrieval: O(n)
    - Traversal: O(n)

Sorted Linked Dictionary

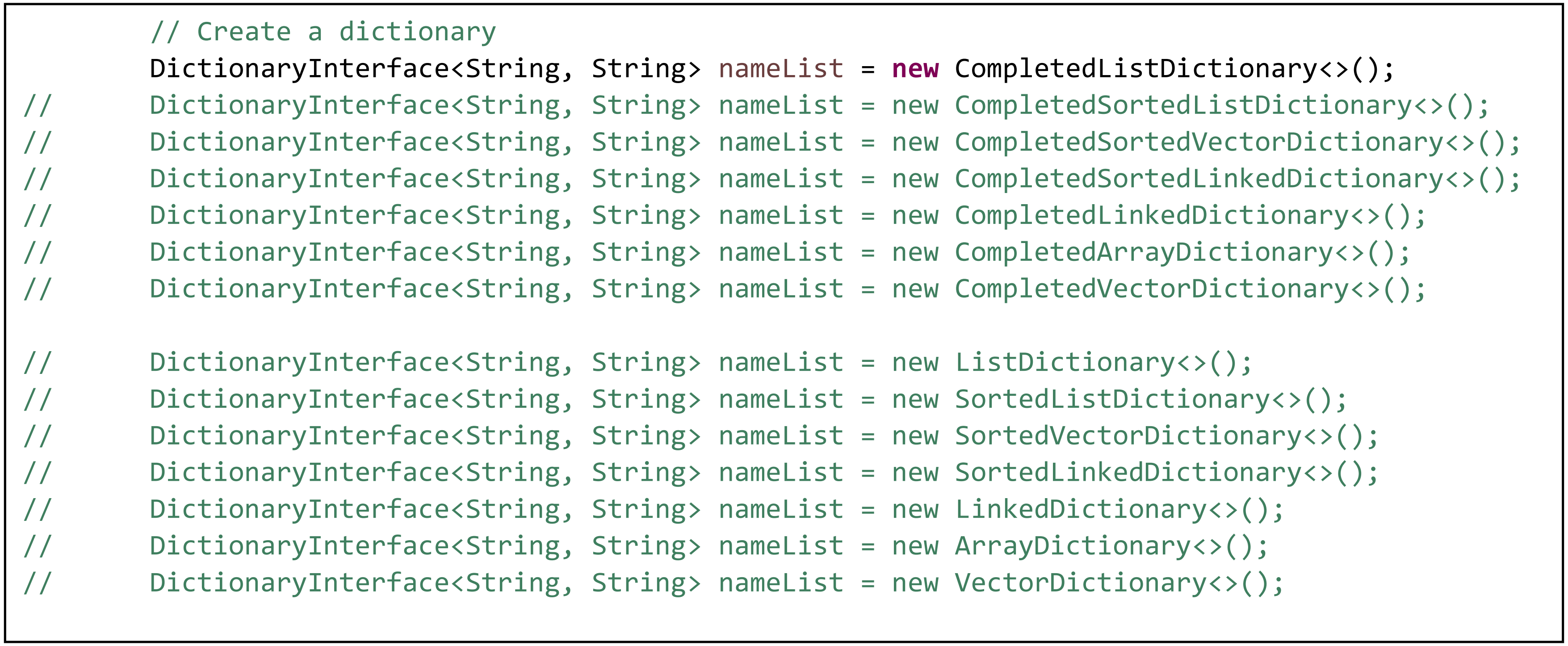
- Efficiency of a sorted linked dictionary:
  - The worst-case efficiencies of the operations.
    - Addition: $O(n)$
    - Removal: $O(n)$
    - Retrieval: $O(n)$
    - Traversal: $O(n)$

Implementation Comparison

Operation	Array-based Unsorted	Array-based Sorted	Linked Unsorted	Linked Sorted
`Addition`	O(*n*)	O(*n*)	O(*n*)	O(*n*)
`Removal`	O(*n*)	O(*n*)	O(*n*)	O(*n*)
`Retrieval`	O(*n*)	O(log *n*)	O(*n*)	O(*n*)
`Traversal`	O(*n*)	O(*n*)	O(*n*)	O(*n*)

Testing - DictionaryTestDriver

- Finish creating the Dictionary classes
- Uncomment the class to test
- Output should always be the same, although sorted dictionaries will have sorted output
- Other examples can be likewise modified for testing
  - TelephoneDirectory, Concordance, FrequencyCounter



```
        // Create a dictionary
        DictionaryInterface<String, String> nameList = new CompletedListDictionary<>();
//      DictionaryInterface<String, String> nameList = new CompletedSortedListDictionary<>();
//      DictionaryInterface<String, String> nameList = new CompletedSortedVectorDictionary<>();
//      DictionaryInterface<String, String> nameList = new CompletedSortedLinkedDictionary<>();
//      DictionaryInterface<String, String> nameList = new CompletedLinkedDictionary<>();
//      DictionaryInterface<String, String> nameList = new CompletedArrayDictionary<>();
//      DictionaryInterface<String, String> nameList = new CompletedVectorDictionary<>();

//      DictionaryInterface<String, String> nameList = new ListDictionary<>();
//      DictionaryInterface<String, String> nameList = new SortedListDictionary<>();
//      DictionaryInterface<String, String> nameList = new SortedVectorDictionary<>();
//      DictionaryInterface<String, String> nameList = new SortedLinkedDictionary<>();
//      DictionaryInterface<String, String> nameList = new LinkedDictionary<>();
//      DictionaryInterface<String, String> nameList = new ArrayDictionary<>();
//      DictionaryInterface<String, String> nameList = new VectorDictionary<>();
```