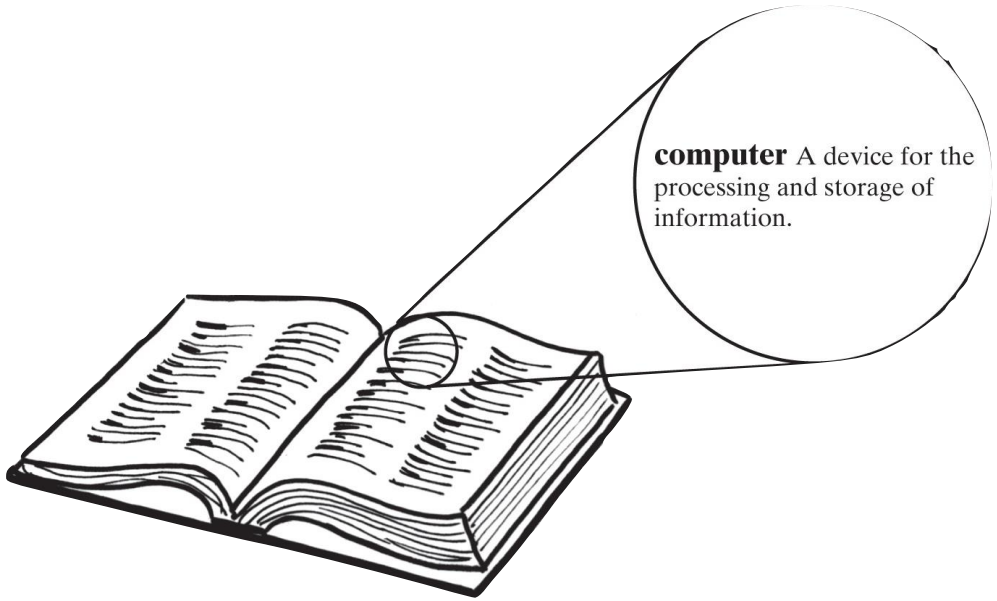
Class 08 - Dictionaries

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



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 collectic

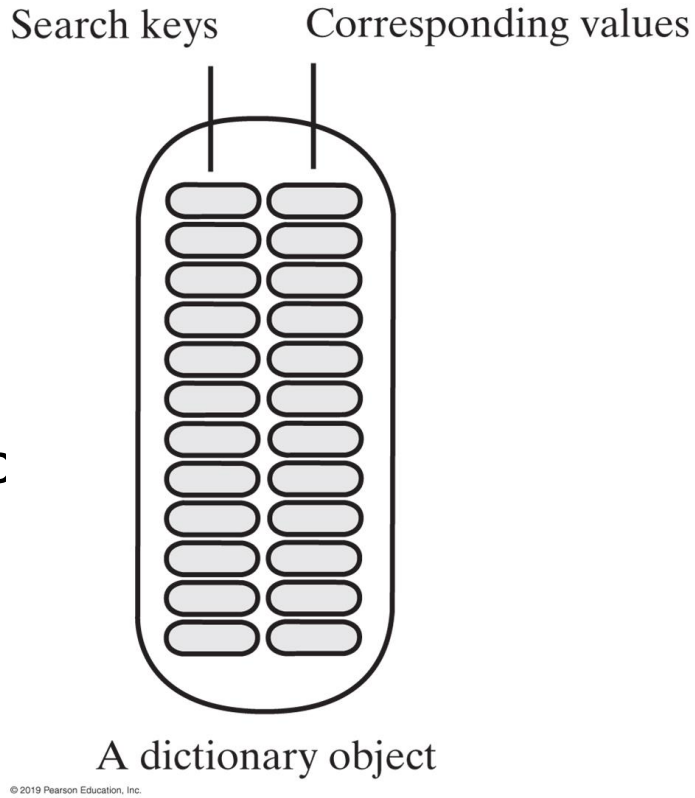


FIGURE 20-2 An instance of the ADT dictionary has search keys paired with corresponding values

Specifications for the ADT Dictionary

- Operations
  - `add(key, value)`
  - `remove(key)`
  - `getValue(key)`
  - `contains(key)`
  - `getKeyIterator()`
  - `getValueIterator()`
  - `isEmpty()`
  - `getSize()`
  - `clear()`

Dictionary Interface

```java
/**
 * 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

```
/**
 * 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 getSize();

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

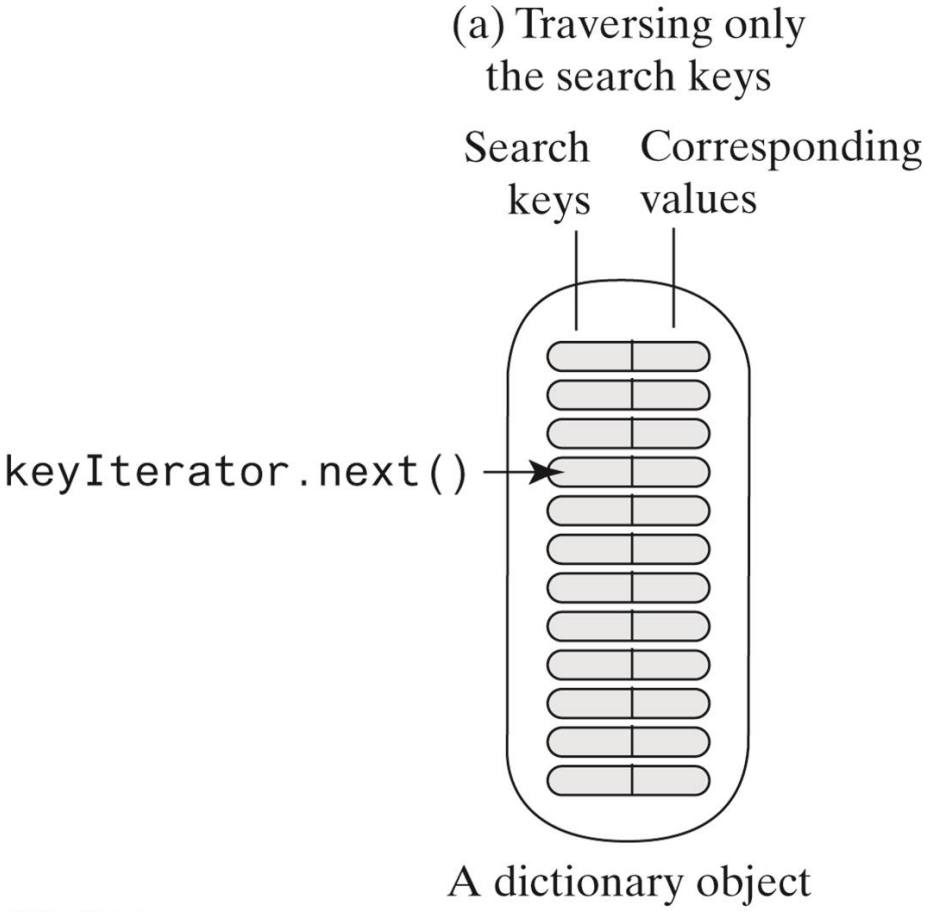
Dictionary Iterators

- 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();
```

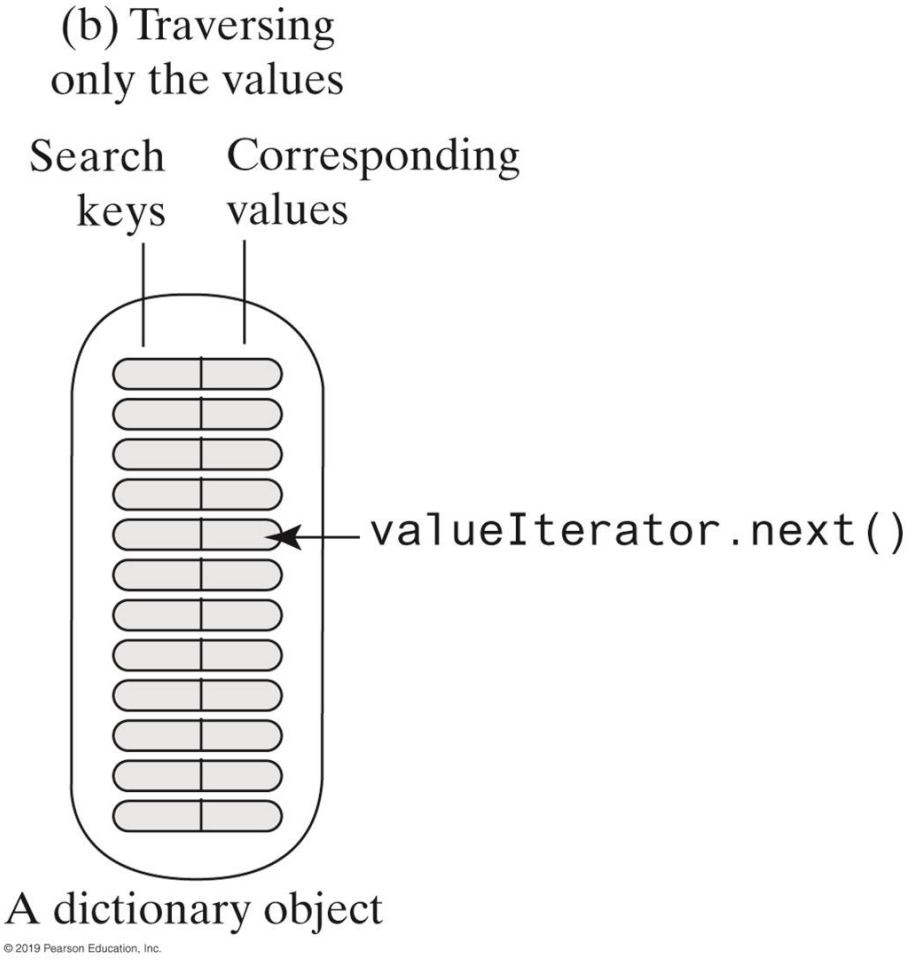
Dictionary Iterators

FIGURE 20-3a Traversing a dictionary's keys separately



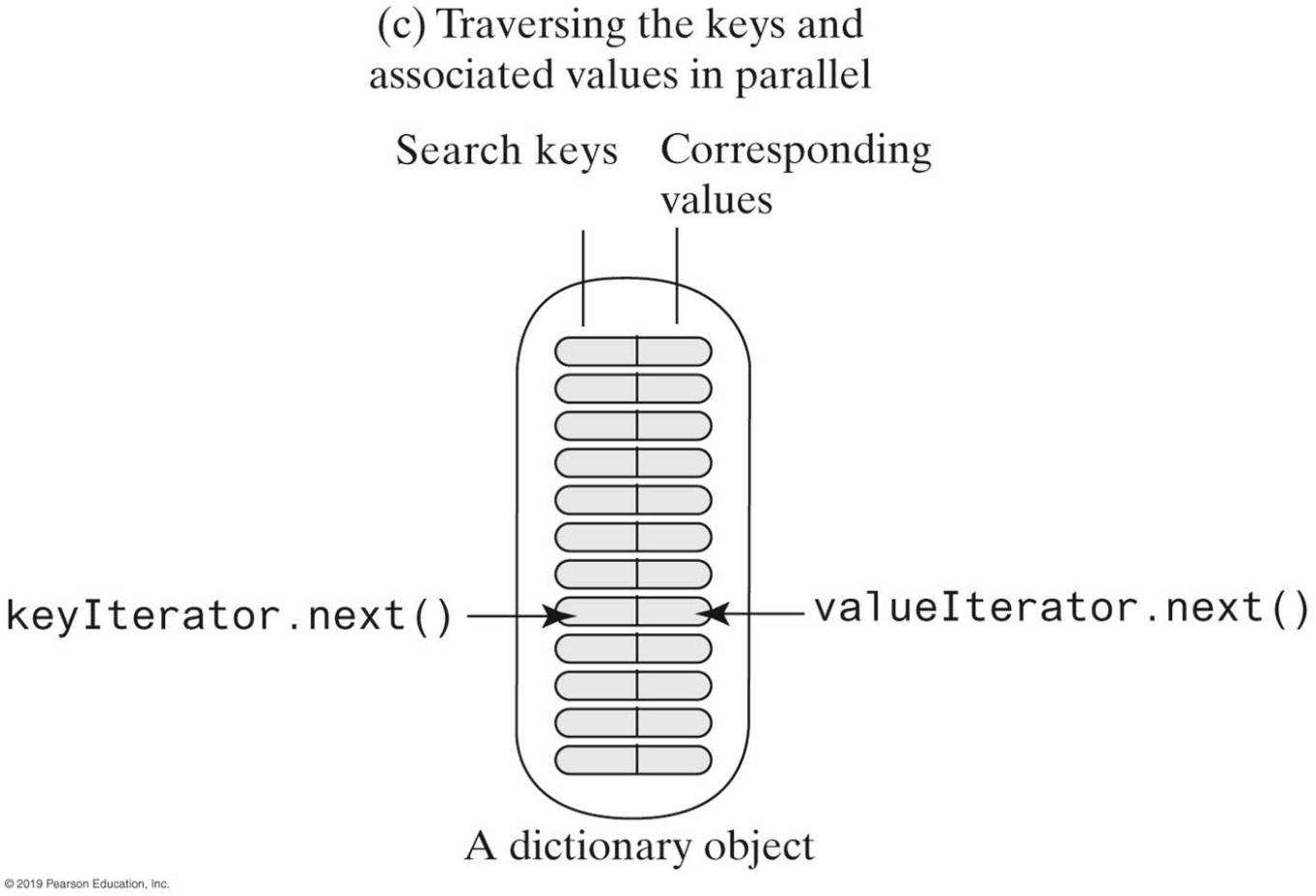
Dictionary Iterators

FIGURE 20-3a Traversing a dictionary's values separately



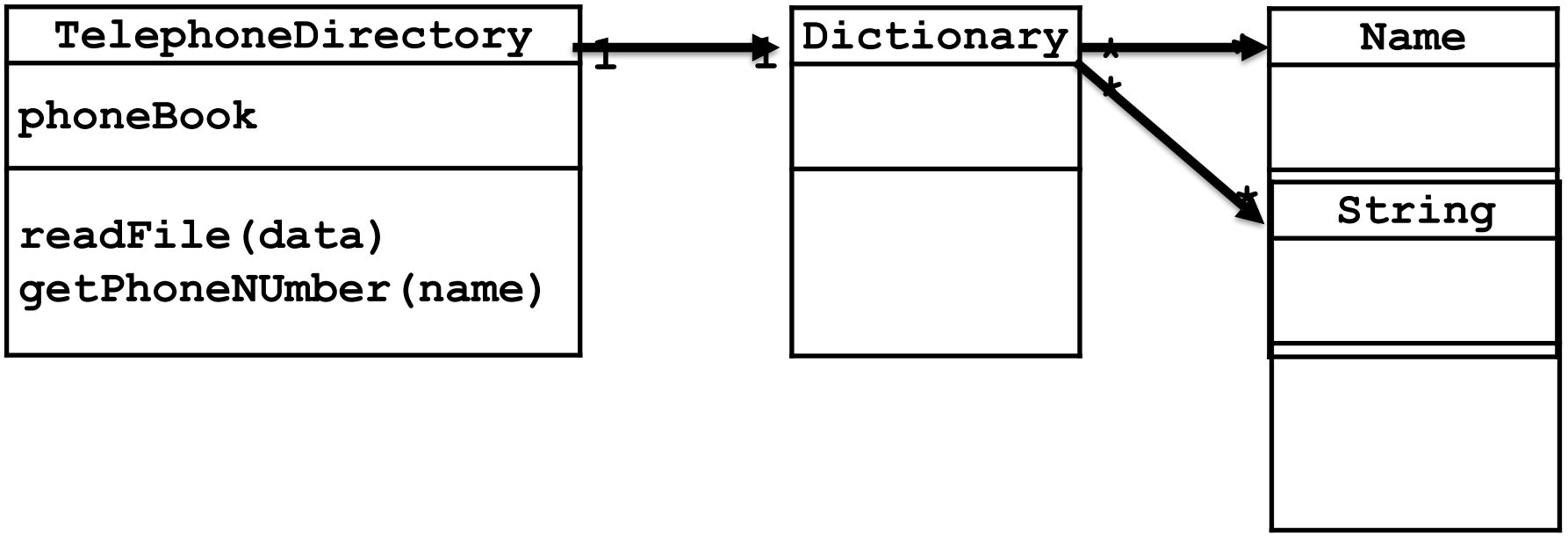
Dictionary Iterators

FIGURE 20-3a Two iterators that traverse a dictionary's keys and values in parallel



Using a Dictionary

FIGURE 20-4 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

Word Frequency Counter Example

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

```
public class FrequencyCounterDemo
{
   public static void main(String[] args)
   {
      FrequencyCounter wordCounter = new FrequencyCounter();
      String fileName = "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();
   }  // end main
}
```

Frequency Counter Class – readFile()

```
/**
 * A class that counts the number of times each word occurs in a document.
 */
public class FrequencyCounter {
    // dictionary to store words and their frequency
    private DictionaryInterface<String, Integer> wordTable;

    public FrequencyCounter() {
        wordTable = new SortedLinkedDictionary<>();
//      wordTable = new SortedArrayDictionary<>();
//      wordTable = new SortedVectorDictionary<>();

    } // end default constructor

    // 20.16
    /**
     * Reads a text file of words and counts their frequencies of occurrence.
     *
     * @param data A text scanner for the text file of data.
     */
    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);
            } // end if
        } // end while

        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));
        } // end while
    } // end display
```

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.

```
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();

        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 ArrayListWithListIterator<>();
                wordTable.add(nextWord, lineList);
            } // end if

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

    data.close();
} // end readFile
```

Concordance class

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

```
/** Displays words and the lines in which they occur. */
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() + " ");
        } // end while

        System.out.println();
    } // end while
} // end display


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

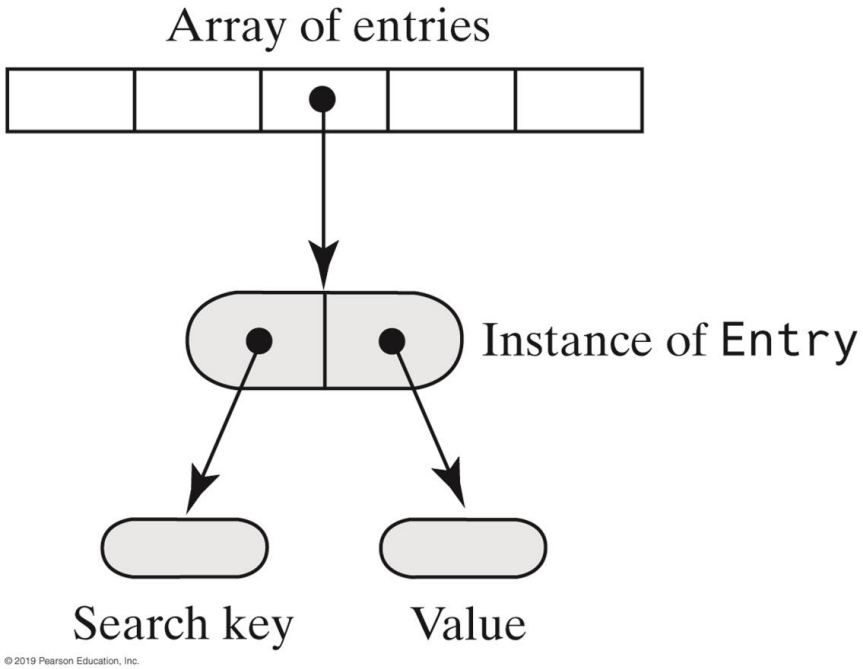
Java Class Library: The Interface `Map`

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

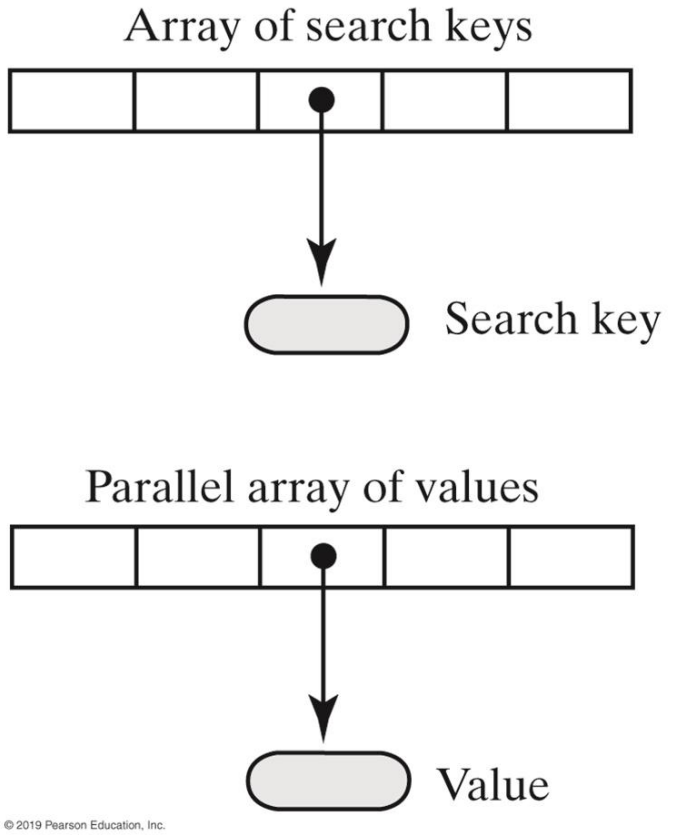
```
public V put(K key, V value);
public V remove (Object key);
public V get(Object key);
public boolean containsKey(Object key);
public boolean containsKey(Object value);
public Set<K> keySet();
public Collection<V> values();
public boolean isEmpty();
public int size();
public void clear();
```

Array-Based Dictionaries

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

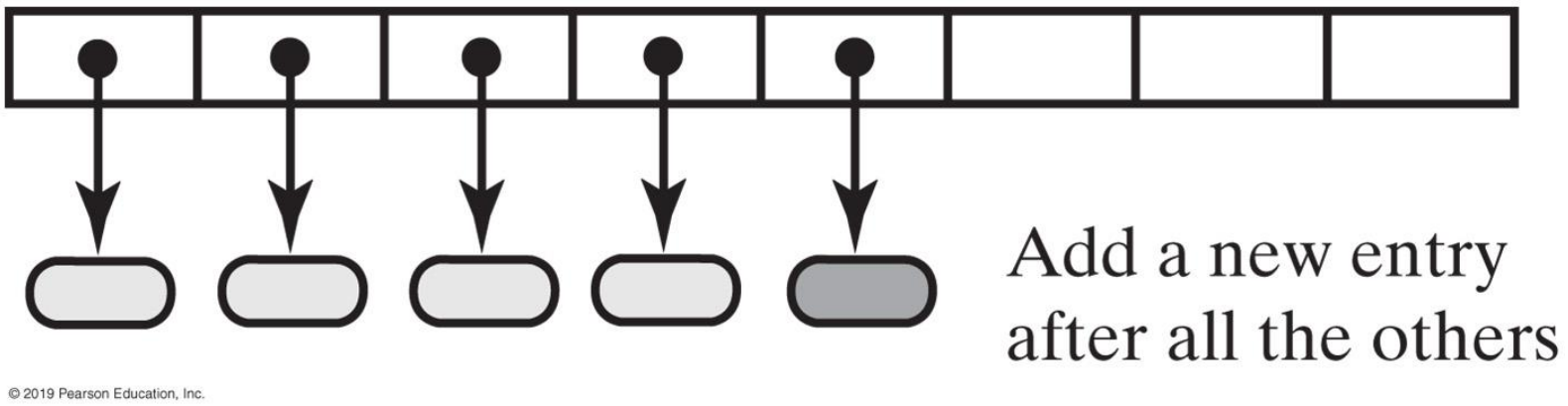


(b) Two arrays in parallel, one of search keys and one of values



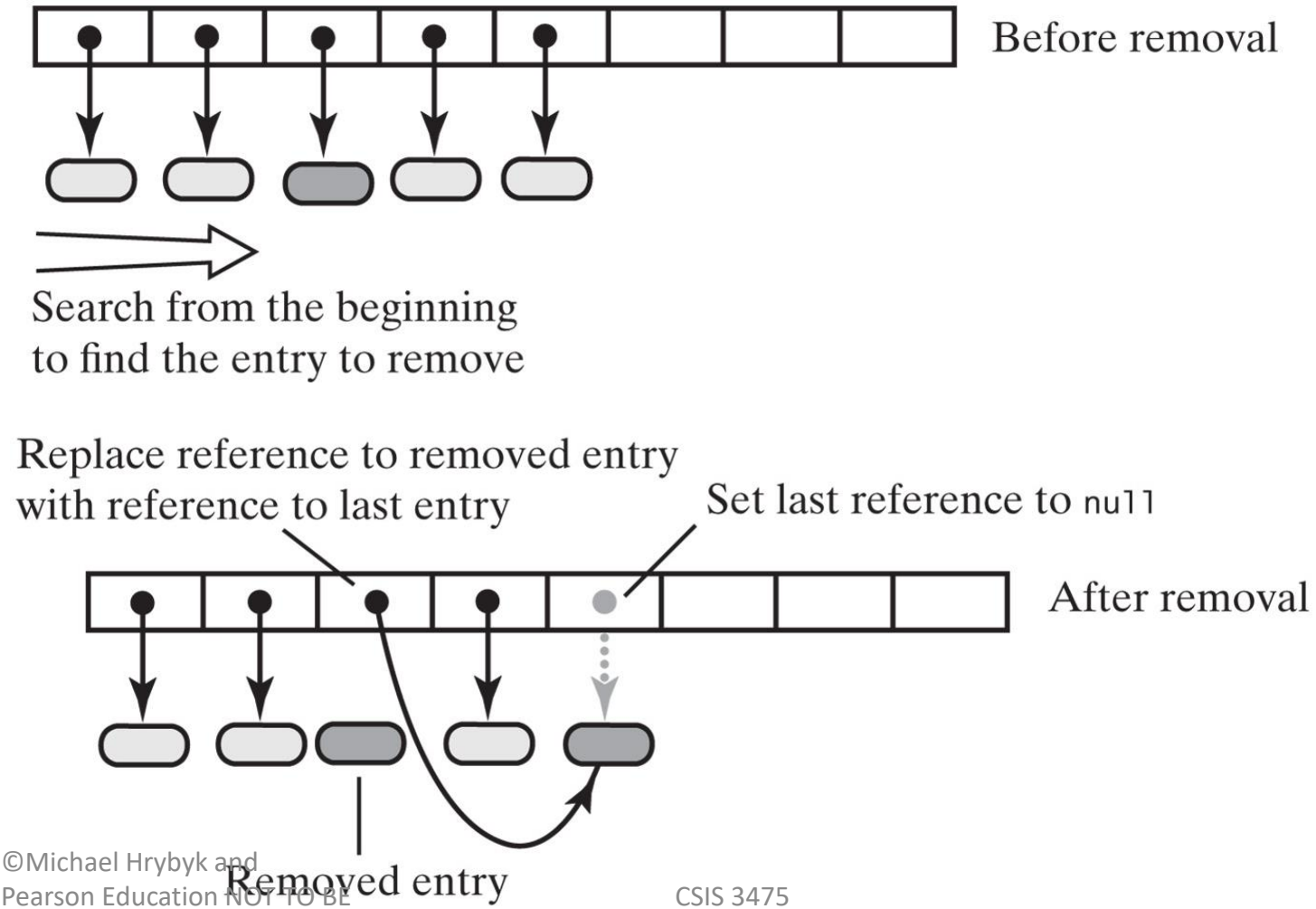
Unsorted Array-Based Implementations

- FIGURE 21-2 Adding a new entry to an unsorted array-based dictionary



Unsorted Array-Based Implementations

- FIGURE 21-3 Removing an entry from an unsorted array-based dictionary



ArrayDictionary

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

```
public class ArrayDictionary<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; // 6 is for testing

    public ArrayDictionary() {
        this(DEFAULT_CAPACITY); // Call next constructor
    } // end default constructor

    public ArrayDictionary(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;
    } // end constructor
```

Entry class

- Place to keep the key and the value.
- Very simple
- Use of generics (note use of different letters, S, T)

```
private class Entry<S, T> {
    private S key;
    private T value;

    private Entry(S searchKey, T dataValue) {
        key = searchKey;
        value = dataValue;
    } // end constructor

    private S getKey() {
        return key;
    } // end getKey

    private T getValue() {
        return value;
    } // end getValue

    private void setValue(T dataValue) {
        value = dataValue;
    } // end setValue

    // No setKey method
} // end Entry
```

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++;
        } // end if

        return result;
    } // end if
} // end add
```

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.
private int locateIndex(K key) {
    // Sequential search
    int index = 0;
    while ((index < numberOfEntries) && !key.equals(dictionary[index].getKey()))
        index++;

    return index;
} // end locateIndex
```

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

```
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;
} // end remove
```

ArrayDictionary – other methods

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

ArrayDictionary - iterators

- Keep a current index
- Only implement 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;
    } // end default constructor

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

    public K next() {
        K result = null;

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

        return result;
    } // end next

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

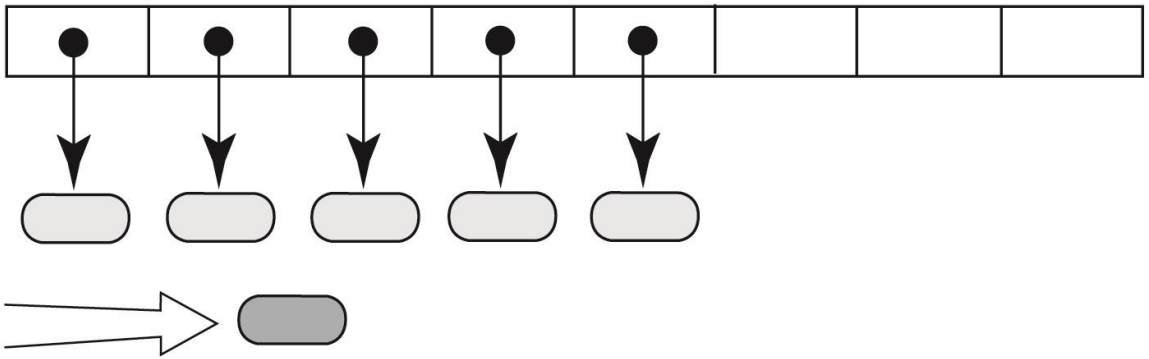
Unsorted Array-Based Implementations

- 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

- FIGURE 21-4a Adding an entry to a sorted array-based dictionary

(a) Locate where to add an entry

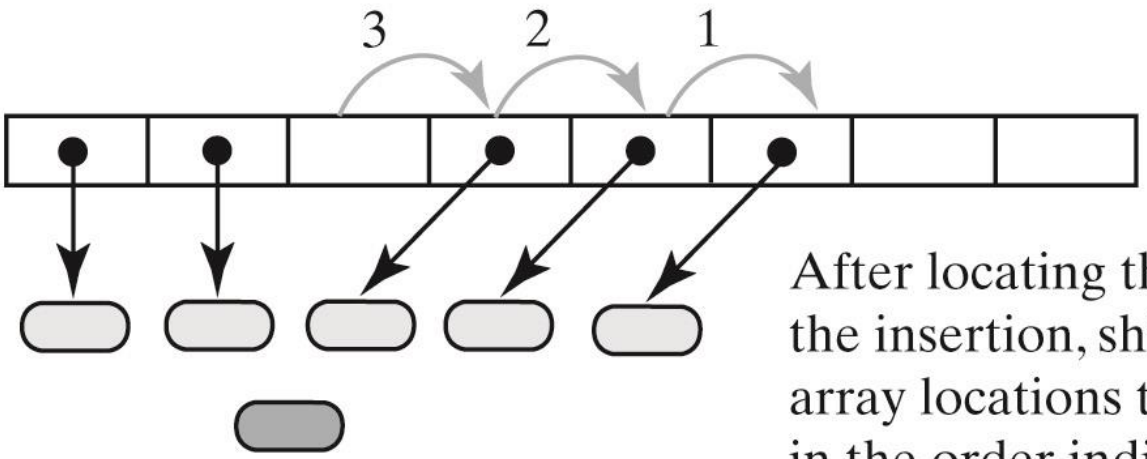


Search from the beginning to find the correct position for a new entry

Sorted Array-Based Implementations

- FIGURE 21-4b Adding an entry to a sorted array-based dictionary

(b) Make room for the new entry

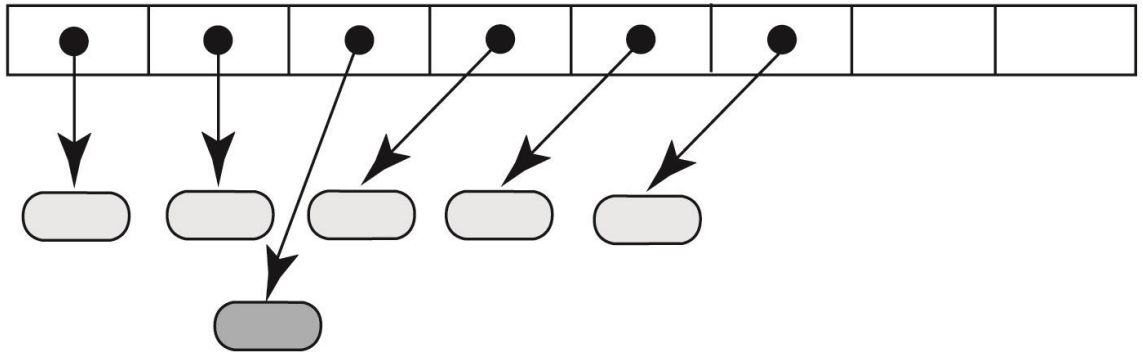


After locating the correct position for the insertion, shift the contents of subsequent array locations toward the end of the array in the order indicated

Sorted Array-Based Implementations

- FIGURE 21-4c Adding an entry to a sorted array-based dictionary

(c) Complete the insertion



SortedArrayDictionary

- Constructor uses Entry[] justl like in unsorted case

```
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * A class that implements the ADT dictionary by using a resizable array. The
 * dictionary is sorted and has distinct search keys. Search keys and associated
 * values are not null.
 *
 * @author Frank M. Carrano
 * @author Timothy M. Henry
 * @version 5.0
 */
public class SortedArrayDictionary<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 SortedArrayDictionary() {
                          this(DEFAULT_CAPACITY);
             } // end default constructor

             public SortedArrayDictionary(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 (just like a sorted list)
- Use of makeRoom()

```
public V add(K key, V value) {
            if ((key == null) || (value == null))
                        throw new IllegalArgumentException("Cannot add null to a dictionary.");
            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 and set
                                    dictionary[keyIndex].setValue(value); // Replace value
                        } else // Key not found; add new entry to dictionary
                        {
                                    makeRoom(keyIndex);
                                    dictionary[keyIndex] = new Entry<>(key, value);
                                    numberOfEntries++;
                        } // end if

                        return result;
            } // end if
}
// Makes room for a new entry at a given index by shifting
// array entries towards the end of the array.
// Precondition: keyIndex is valid; list length is old length.
private void makeRoom(int keyIndex) {
            // Move each entry to next higher position beginning at end of list
            // and continuing until item at newPosition is moved
            for (int index = numberOfEntries - 1; index >= keyIndex; index--) {
                        dictionary[index + 1] = dictionary[index]; // Shift right
            } // end for
} // end makeRoom
```

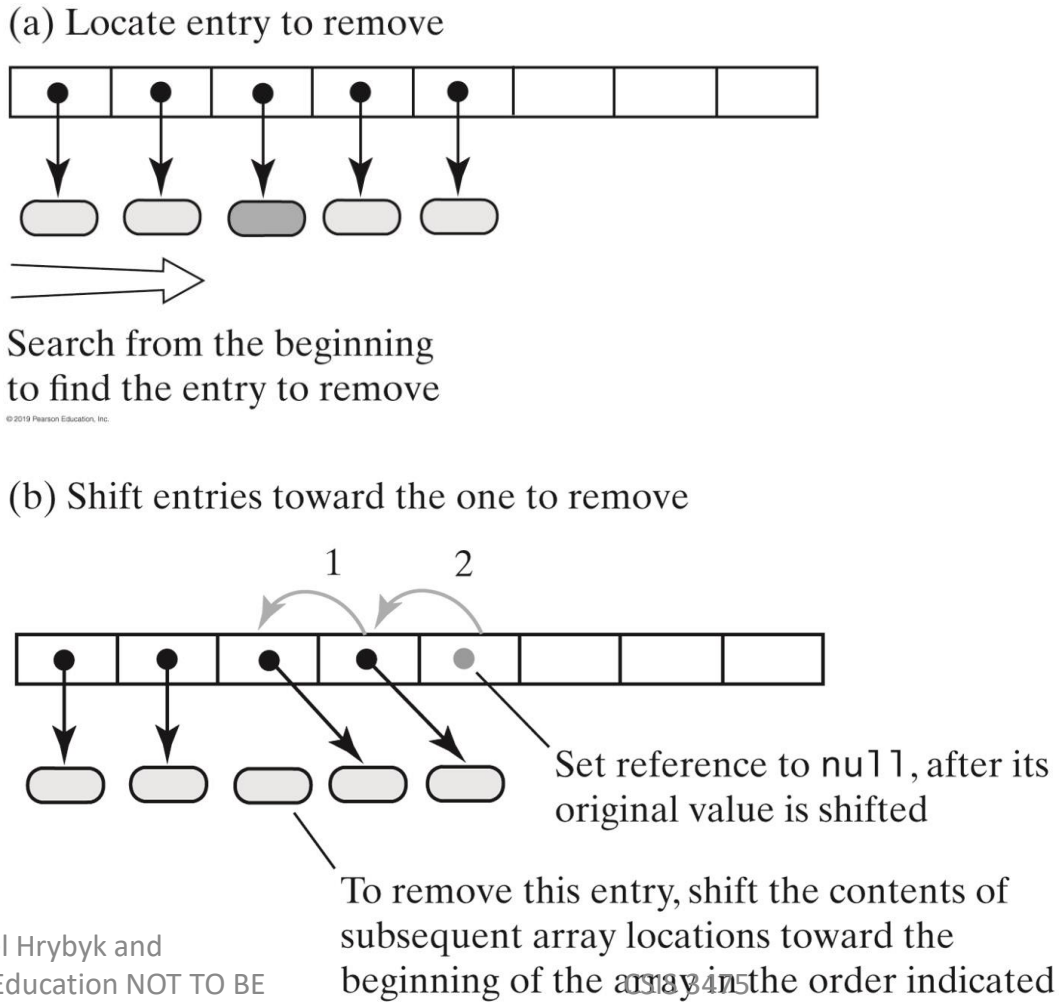
SortedArrayDictionary – locatedIndex()

- Notice use of compareTo(), which lets us place a new entry in the spot right before the current one

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

                    return index;
        }
```

Sorted Array-Based Dictionary

- FIGURE 21-5 Removing an entry from a sorted array-based dictionary



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
- Like removeGap() in ArrayList

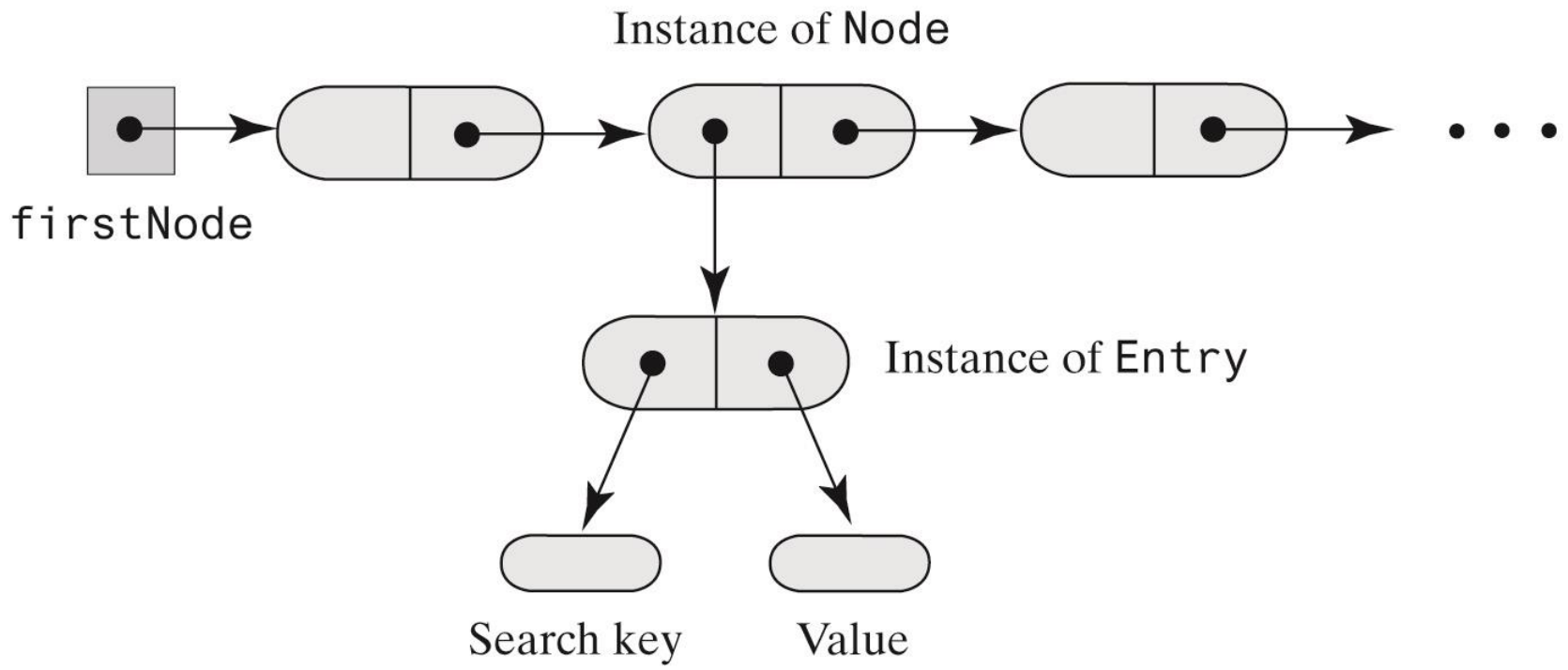
```
public V remove(K key) {
            V result = null;
            int keyIndex = locateIndex(key);

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

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

Sorted Array-Based Dictionary

- Efficiency of sorted array-based dictionary.
- When **`locateIndex`** uses a binary search in the sorted array-based implementation, the worst-case efficiencies are:
  - Addition: $\mathrm{O}(n)$
  - Removal: $\mathrm{O}(n)$
  - Retrieval: $\mathrm{O}(\log n)$
  - Traversal: $\mathrm{O}(n)$

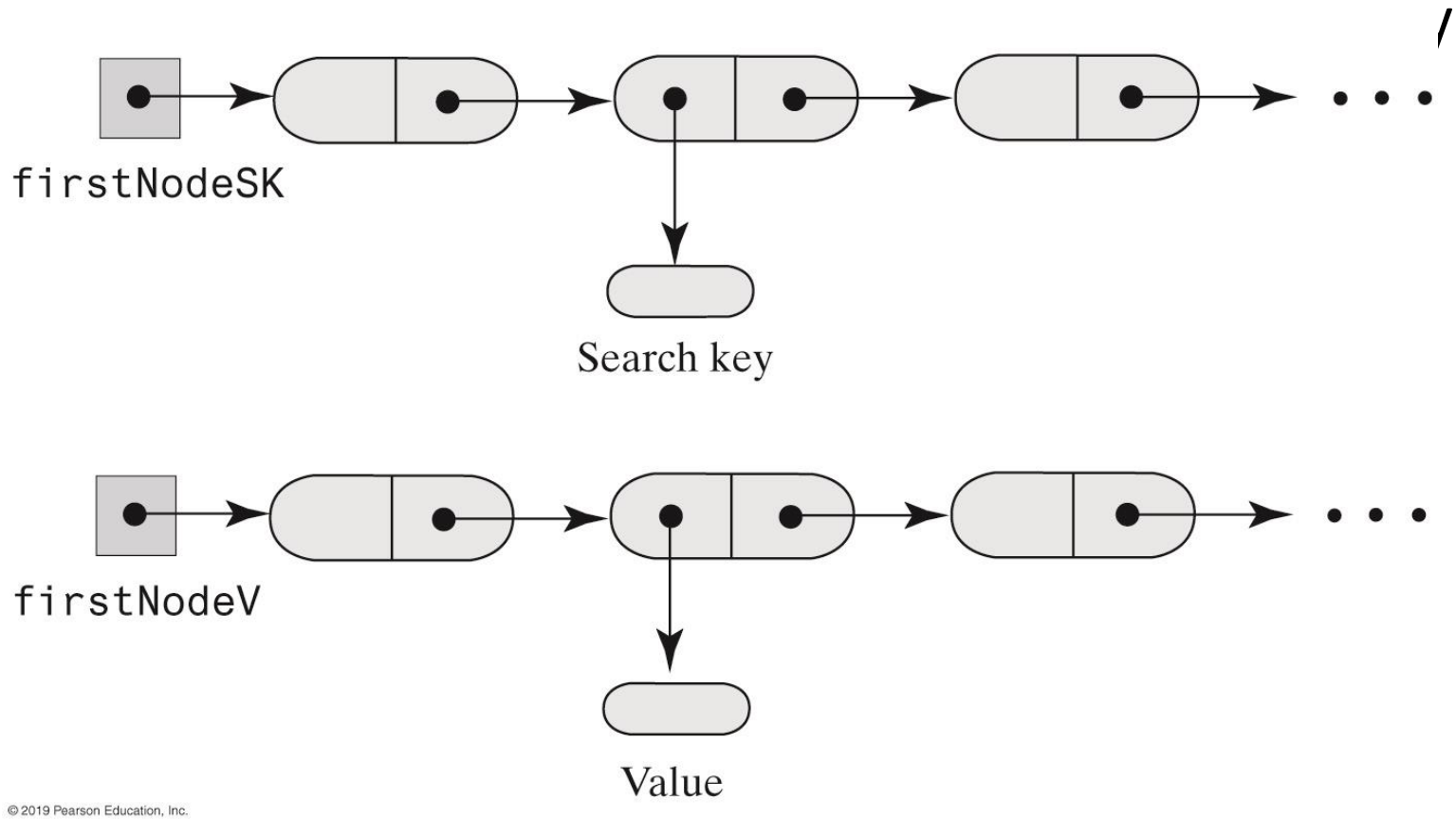
Linked Dictionary Implementations

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



Linked Dictionary Implementations

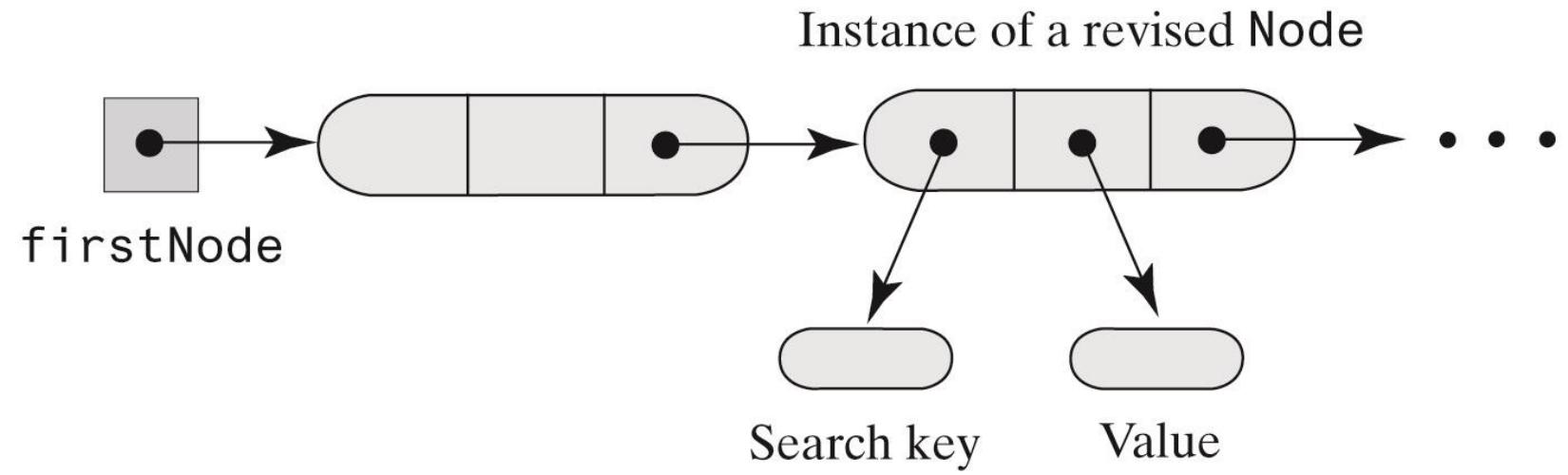
(b) Parallel chains of search keys and values



Linked Dictionary Implementations

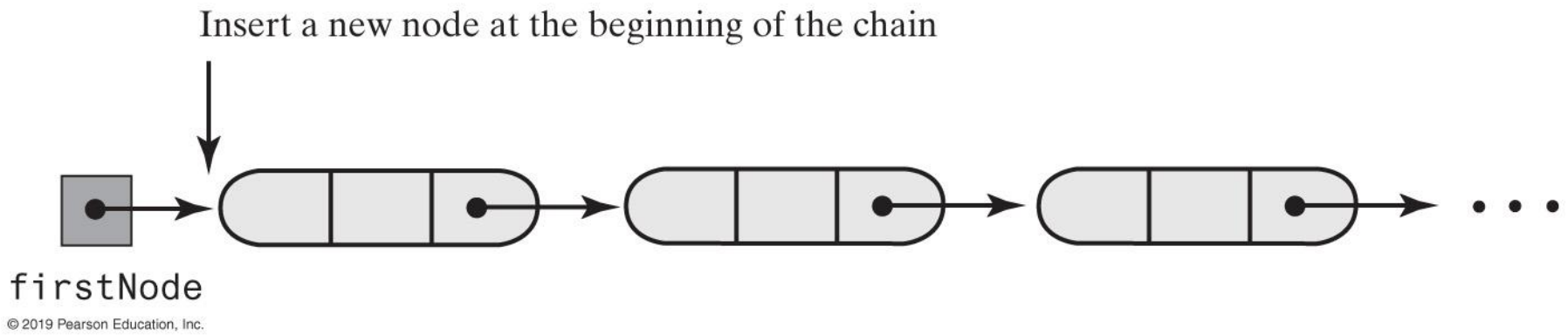
- FIGURE 21-6c Representing the entries in a dictionary

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



Linked Dictionary Implementations

- FIGURE 21-7 Adding to an unsorted linked dictionary



An Unsorted Linked Dictionary

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

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))
		throw new IllegalArgumentException("Cannot add null to a dictionary.");
	else {
		// Search chain until you either find a node containing key
		// or locate where it should be
		Node currentNode = firstNode;
		Node nodeBefore = null;

		while ((currentNode != null) && (key.compareTo(currentNode.getKey()) > 0)) {
			nodeBefore = currentNode;
			currentNode = currentNode.getNextNode();
		} // end while

		if ((currentNode != null) && key.equals(currentNode.getKey())) {
			// Key in dictionary; replace corresponding value
			result = currentNode.getValue(); // Get old value
			currentNode.setValue(value); // Replace value
		} else // Key not in dictionary; add new node in proper order
		{
			// Assertion: key and value are not null
			Node newNode = new Node(key, value); // Create new node

			if (nodeBefore == null) { // Add at beginning (includes empty chain)
				newNode.setNextNode(firstNode);
				firstNode = newNode;
			} else // Add elsewhere in non-empty chain
			{
				newNode.setNextNode(currentNode); // currentNode is after new node
				nodeBefore.setNextNode(newNode); // nodeBefore is before new node
			} // end if

			numberOfEntries++; // Increase length for both cases
		} // end if
	} // end if

	return result;
} // end add
```

SortedLinkedDictionary – remove()

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

     if (!isEmpty()) {
          // Find node before the one that contains or could contain key
          Node currentNode = firstNode;
          Node nodeBefore = null;

          while ((currentNode != null) && (key.compareTo(currentNode.getKey()) > 0)) {
               nodeBefore = currentNode;
               currentNode = currentNode.getNextNode();
          } // end while

          if ((currentNode != null) && key.equals(currentNode.getKey())) {
               Node nodeAfter = currentNode.getNextNode(); // Node after the one to be removed

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

               result = currentNode.getValue(); // Get ready to return removed entry
               numberOfEntries--; // Decrease length for both cases
          } // end if
     } // end if

     return result;
} // end remove
```

SortedLinkedDictionary – Node class

- Similar to Node for lists, but need it to store key and value
- Could have used original Node and had data be an Entry

```
private class Node {
                private K key;
                private V value;
                private Node next;

                private Node(K searchKey, V dataValue) {
                                key = searchKey;
                                value = dataValue;
                                next = null;
                } // end constructor

                private Node(K searchKey, V dataValue, Node nextNode) {
                                key = searchKey;
                                value = dataValue;
                                next = nextNode;
                } // end constructor

                private K getKey() {
                                return key;
                } // end getKey

                private V getValue() {
                                return value;
                } // end getValue

                // no setKey!!

                private void setValue(V newValue) {
                                value = newValue;
                } // end setValue

                private Node getNextNode() {
                                return next;
                } // end getNextNode

                private void setNextNode(Node nextNode) {
                                next = nextNode;
                } // end setNextNode
} // end Node
```

Sorted Linked Dictionary

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

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`	$O(n)$	$O(n)$	$O(n)$	$O(n)$