- Do you plan to search the data? If so, how? If you want to examine elements in order of insertion, you want a list. If you intend to search for arbitrary elements, a set may be better. If you need to find an object, given partial information about it (for example, a PIN or ID number, to find a user's bank account object), a map may be best.
- In what order should the elements be stored? Lists hold their elements in order of insertion, whereas the tree collections (`TreeSet` and `TreeMap`) order elements by their natural ordering. If order doesn't matter, you may want a hash table collection such as a `HashSet`.

Chapter Summary

A collection is an object that stores a group of other objects. Examples of collections are `ArrayList`, `HashSet`, and `TreeMap`. Collections are used to structure, organize, and search data.

A linked list is a collection that's similar to an `ArrayList` but that is implemented internally by storing each element in a small container object called a node. Linked lists can perform certain operations faster than array lists, such as adding data to and removing data from the front or middle of the list.

An iterator is an object that keeps track of the current position in a list and expedites the examination of its elements in sequential order. Linked lists are often used with iterators for increased efficiency.

An abstract data type (ADT) is a specification of the operations that can be performed on data. Two examples of ADTs are `List` and `Set`. ADTs in the Java Collections Framework are represented as interfaces (e.g., the `List` interface, which is implemented both by `LinkedList` and by `ArrayList`).

A set is a collection that doesn't allow duplicates. Sets generally can be searched very quickly to see whether they contain a particular element value. The `Set` interface represents sets.

There are two major set classes in Java: `TreeSet` and `HashSet`. A `TreeSet` holds `Comparable` data in a sorted order; a `HashSet` can hold any data and can be searched faster, but its elements are stored in an unpredictable order.

A map is a collection that associates key objects with value objects. Maps are used to create relationships of association between pieces of data, such as a person's name and phone number.

There are two major map classes in Java: `TreeMap` and `HashMap`. A `TreeMap` holds `Comparable` keys in a sorted order; a `HashMap` can hold any data as its keys and performs value lookups faster, but its keys are stored in an unpredictable order.

Self-Check Problems

Section 11.1: Lists

1. When should you use a `LinkedList` instead of an `ArrayList`?
2. Would a `LinkedList` or an `ArrayList` perform better when run on the following code? Why?

```
public static int min(List<Integer> list) {
    int min = list.get(0);
    for (int i = 1; i < list.size(); i++) {
```

```
            if (list.get(i) < min) {
                min = list.get(i);
            }
        }
        return min;
    }
```

3. What is an iterator? Why are iterators often used with linked lists?

4. Write a piece of code that counts the number of duplicate elements in a linked list, that is, the number of elements whose values are repeated at an earlier index in the list. Assume that all duplicates in the list occur consecutively. For example, the list `[1, 1, 3, 5, 5, 5, 5, 7, 7, 11]` contains five duplicates: one duplicate of element value `1`, three duplicates of element value `5`, and one duplicate of element value `7`.

5. Write a piece of code that inserts a `String` into an ordered linked list of `Strings`, maintaining sorted order. For example, for the list `["Alpha", "Baker", "Foxtrot", "Tango", "Whiskey"]`, inserting `"Charlie"` in order would produce the list `["Alpha", "Baker", "Charlie", "Foxtrot", "Tango", "Whiskey"]`.

6. Write a method called `removeAll` that accepts a linked list of integers as a parameter and removes all occurrences of a particular value. You must preserve the original relative order of the remaining elements of the list. For example, the call `removeAll(list, 3)` would change the list `[3, 9, 4, 2, 3, 8, 17, 4, 3, 18, 2, 3]` to `[9, 4, 2, 8, 17, 4, 18, 2]`.

7. Write a method called `wrapHalf` that accepts a linked list of integers as a parameter and moves the first half of the list to the back of the list. If the list contains an odd number of elements, the smaller half is wrapped (in other words, for a list of size N, the middle element, N/2, becomes the first element in all cases). For example, calling `wrapHalf` on the list `[1, 2, 3, 4, 5, 6]`, would change that list into `[4, 5, 6, 1, 2, 3]`. For the list `[5, 6, 7, 8, 9]`, the result would be `[7, 8, 9, 5, 6]`.

8. What is an abstract data type (ADT)? What ADT does a linked list implement?

9. Self-Check Problem 4 asked you to write code that would count the duplicates in a linked list. Rewrite your code as a method called `countDuplicates` that will allow either an `ArrayList` or a `LinkedList` to be passed as the parameter.

Section 11.2: Sets

10. A `List` has every method that a `Set` has, and more. So why would you use a `Set` rather than a `List`?

11. When should you use a `TreeSet`, and when should you use a `HashSet`?

12. A `Set` doesn't have the `get` and `set` methods that an `ArrayList` has. How do you examine every element of a `Set`?

13. What elements are contained in the following set after this code executes?

```
Set<Integer> set = new HashSet<Integer>();
set.add(74);
set.add(12);
set.add(74);
set.add(74);
set.add(43);
set.remove(74);
set.remove(999);
set.remove(43);
```

```
set.add(32);
set.add(12);
set.add(9);
set.add(999);
```

14. How do you perform a union operation on two sets? An intersection? Try to give an answer that doesn't require any loops.

15. Write the output produced when the following method is passed each of the following lists:

```
public static void mystery(List<String> list) {
    Set<String> result = new TreeSet<String>();
    for (String element : list) {
        if (element.compareTo(list.get(0)) < 0) {
            result.add(element);
        } else {
            result.clear();
        }
    }
    System.out.println(result);
}
```

a. `[marty, stuart, helene, jessica, amanda]`
b. `[sara, caitlin, janette, zack, riley]`
c. `[zorah, alex, tyler, roy, roy, charlie, phil, charlie, tyler]`

Section 11.3: Maps

16. Write the code to declare a `Map` that associates people's names with their ages. Add mappings for your own name and age, as well as those of a few friends or relatives.

17. A `Map` doesn't have the `get` and `set` methods that an `ArrayList` has. It doesn't even have an iterator method like a `Set` does, nor can you use a for-each loop on it directly. How do you examine every key (or every value) of a `Map`?

18. What keys and values are contained in the following map after this code executes?

```
Map<Integer, String> map = new HashMap<Integer, String>();
map.put(8, "Eight");
map.put(41, "Forty-one");
map.put(8, "Ocho");
map.put(18, "Eighteen");
map.put(50, "Fifty");
map.put(132, "OneThreeTwo");
map.put(28, "Twenty-eight");
map.put(79, "Seventy-nine");
map.remove(41);
map.remove(28);
map.remove("Eight");
map.put(50, "Forty-one");
map.put(28, "18");
map.remove(18);
```

19. Write the output produced when the following method is passed each of the following maps:

```
public static void mystery(Map<String, String> map) {
    Map<String, String> result = new TreeMap<String, String>();
    for (String key : map.keySet()) {
        if (key.compareTo(map.get(key)) < 0) {
            result.put(key, map.get(key));
        } else {
            result.put(map.get(key), key);
        }
    }
    System.out.println(result);
}
```

a. `{two=deux, five=cinq, one=un, three=trois, four=quatre}`
b. `{skate=board, drive=car, program=computer, play=computer}`
c. `{siskel=ebert, girl=boy, H=T, ready=begin, first=last, begin=end}`
d. `{cotton=shirt, tree=violin, seed=tree, light=tree, rain=cotton}`

20. Write the output produced when the following method is passed each of the following maps:

```
public static void mystery(Map<String, String> m) {
    Set<String> s = new TreeSet<String>();
    for (String key : m.keySet()) {
        if (!m.get(key).equals(key)) {
            s.add(m.get(key));
        } else {
            s.remove(m.get(key));
        }
    }
    System.out.println(s);
}
```

a. `{sheep=wool, house=brick, cast=plaster, wool=wool}`
b. `{ball=blue, winkie=yellow, corn=yellow, grass=green, emerald=green}`
c. `{pumpkin=peach, corn=apple, apple=apple, pie=fruit, peach=peach}`
d. `{lab=ipl, lion=cat, corgi=dog, cat=cat, emu=animal, nyan=cat}`

21. Write the map returned when the following method is passed the following maps:

```
public Map<String, String> mystery(Map<String, Integer> map1,
                                   Map<Integer, String> map2) {
    Map<String, String> result = new TreeMap<String, String>();
    for (String s1 : map1.keySet()) {
        if (map2.containsKey(map1.get(s1))) {
            result.put(s1, map2.get(map1.get(s1)));
        }
    }
```

```
    return result;
}
```

a. `map1: {bar=1, baz=2, foo=3, mumble=4},`
   `map2: {1=earth, 2=wind, 3=air, 4=fire}`
b. `map1: {five=105, four=104, one=101, six=106, three=103, two=102},`
   `map2: {99=uno, 101=dos, 103=tres, 105=quatro}`
c. `map1: {a=42, b=9, c=7, d=15, e=11, f=24, g=7},`
   `map2: {1=four, 3=score, 5=and, 7=seven, 9=years, 11=ago}`

22. Modify the `WordCount` program so that it prints the most frequently occurring words sorted by number of occurrences. To do this, write code at the end of the program to create a reverse map from counts to words that is based on the original map. Assume that no two words of interest occur the exact same number of times.

Exercises

1. Modify the `Sieve` program developed in Section 11.1 to make two optimizations. First, instead of storing all integers up to the maximum in the `numbers` list, store only 2 and all odd numbers from 3 upward. Second, write code to ensure that if the first number in the `numbers` list ever reaches the square root of the maximum, all remaining values from the `numbers` list are moved into the `primes` list. (Why is this a valid operation?)

2. Write a method called `alternate` that accepts two `Lists` as its parameters and returns a new `List` containing alternating elements from the two lists, in the following order:

- First element from first list
- First element from second list
- Second element from first list
- Second element from second list
- Third element from first list
- Third element from second list
- . . .

If the lists do not contain the same number of elements, the remaining elements from the longer list should be placed consecutively at the end. For example, for a first list of `[1, 2, 3, 4, 5]` and a second list of `[6, 7, 8, 9, 10, 11, 12]`, a call of `alternate(list1, list2)` should return a list containing `[1, 6, 2, 7, 3, 8, 4, 9, 5, 10, 11, 12]`.

3. Write a method called `removeInRange` that accepts four parameters: a `LinkedList`, an element value, a starting index, and an ending index. The method's behavior is to remove all occurrences of the given element that appear in the list between the starting index (inclusive) and the ending index (exclusive). Other values and occurrences of the given value that appear outside the given index range are not affected.

For example, for the list `[0, 0, 2, 0, 4, 0, 6, 0, 8, 0, 10, 0, 12, 0, 14, 0, 16]`, a call of `removeInRange(list, 0, 5, 13)` should produce the list `[0, 0, 2, 0, 4, 6, 8, 10, 12, 0, 14, 0, 16]`. Notice that the zeros located at indexes between 5 inclusive and 13 exclusive in the original list (before any modifications were made) have been removed.

4. Write a method called `partition` that accepts a list of integers and an integer value `E` as its parameter, and rearranges (partitions) the list so that all the elements with values less than `E` occur before all elements with values greater than `E`. The exact order of the elements is unimportant, so long as all elements less than `E` appear before all elements greater than `E`. For example, for the linked list `[15, 1, 6, 12, -3, 4, 8, 21, 2, 30, -1, 9]`, one acceptable ordering of

the list after a call of `partition(list, 5)` would be `[-1, 1, 2, 4, -3, 12, 8, 21, 6, 30, 15, 9]`. You may assume that the list contains no duplicates and does not contain the element value `E`.

5. Write a method called `sortAndRemoveDuplicates` that accepts a list of integers as its parameter and rearranges the list's elements into sorted ascending order, as well as removing all duplicate values from the list. For example, the list `[7, 4, -9, 4, 15, 8, 27, 7, 11, -5, 32, -9, -9]` would become `[-9, -5, 4, 7, 8, 11, 15, 27, 32]` after a call to your method. Use a `Set` as part of your solution.

6. Write a method `countUnique` that accepts a list of integers as a parameter and returns the number of unique integer values in the list. Use a set as auxiliary storage to help you solve this problem. For example, if a list contains the values `[3, 7, 3, -1, 2, 3, 7, 2, 15, 15]`, your method should return `5`. The empty list contains `0` unique values.

7. Write a method `countCommon` that accepts two lists of integers as parameters and returns the number of unique integers that occur in both lists. Use one or more sets as storage to help you solve this problem. For example, if one list contains the values `[3, 7, 3, -1, 2, 3, 7, 2, 15, 15]` and the other list contains the values `[-5, 15, 2, -1, 7, 15, 36]`, your method should return `4` because the elements `-1, 2, 7,` and `15` occur in both lists.

8. Write a method `maxLength` that accepts a set of strings as a parameter and that returns the length of the longest string in the list. If your method is passed an empty set, it should return `0`.

9. Write a method `hasOdd` that accepts a set of integers as a parameter and returns `true` if the set contains at least one odd integer and `false` otherwise. If passed the empty set, your method should return `false`.

10. Write a method `removeEvenLength` that accepts a set of strings as a parameter and that removes all of the strings of even length from the set.

11. Write a method called `symmetricSetDifference` that accepts two `Sets` as parameters and returns a new `Set` containing their symmetric set difference (that is, the set of elements contained in either of the two sets but not in both). For example, the symmetric difference between the sets `[1, 4, 7, 9]` and `[2, 4, 5, 6, 7]` is `[1, 2, 5, 6, 9]`.

12. Write a method `contains3` that accepts a list of strings as a parameter and returns `true` if any single string occurs at least 3 times in the list, and `false` otherwise. Use a map as auxiliary storage.

13. Write a method `isUnique` that accepts a map whose keys and values are strings as a parameter and returns `true` if no two keys map to the same value (and `false` if any two or more keys do map to the same value). For example, if the map contains the following key/value pairs, your method would return `true`: `{Marty=Stepp, Stuart=Reges, Jessica=Miller, Amanda=Camp, Hal=Perkins}`. But calling it on the following map would return `false`, because of two mappings for `Perkins` and `Reges`: `{Kendrick=Perkins, Stuart=Reges, Jessica=Miller, Bruce=Reges, Hal=Perkins}`.

14. Write a method `intersect` that accepts two maps whose keys are strings and whose values are integers as parameters and returns a new map containing only the key/value pairs that exist in both of the parameter maps. In order for a key/value pair to be included in your result, not only do both maps need to contain a mapping for that key, but they need to map it to the same value. For example, if the two maps passed are `{Janet=87, Logan=62, Whitaker=46, Alyssa=100, Stefanie=80, Jeff=88, Kim=52, Sylvia=95}` and `{Logan=62, Kim=52, Whitaker=52, Jeff=88, Stefanie=80, Brian=60, Lisa=83, Sylvia=87}`, your method would return the following new map (the order of the key/value pairs does not matter): `{Logan=62, Stefanie=80, Jeff=88, Kim=52}`.

15. Write a method `maxOccurrences` that accepts a list of integers as a parameter and returns the number of times the most frequently occurring integer (the "mode") occurs in the list. Solve this problem using a map as auxiliary storage. If the list is empty, return `0`.

16. Write a method called `is1to1` that accepts a map whose keys and values are strings as its parameter and returns `true` if no two keys map to the same value. For example, `{Marty=206-9024, Hawking=123-4567, Smith=949-0504,`

`Newton=123–4567}` should return `false`, but `{Marty=206–9024, Hawking=555–1234, Smith=949–0504, Newton=123–4567}` should return `true`. The empty map is considered 1-to-1 and returns `true`.

17. Write a method called `subMap` that accepts two maps from strings to strings as its parameters and returns `true` if every key in the first map is also contained in the second map and maps to the same value in the second map. For example, `{Smith=949–0504, Marty=206–9024}` is a submap of `{Marty=206–9024, Hawking=123–4567, Smith=949–0504, Newton=123–4567}`. The empty map is a submap of every map.

18. Write a method called `reverse` that accepts a map from strings to strings as a parameter and returns a new map that is the reverse of the original. The reverse of a map is a new map that uses the values from the original as its keys and the keys from the original as its values. Since a map's values need not be unique but its keys must be, you should have each value map to a set of keys. In other words, if the original map maps keys of type *K* to values of type *V*, the new map should map keys of type *V* to values that are `Sets` containing elements of type *K*. For example, the map `{42=Marty, 81=Sue, 17=Ed, 31=Dave, 56=Ed, 3=Marty, 29=Ed}` has a reverse of `{Marty=[42, 3], Sue=[81], Ed=[17, 56, 29], Dave=[31]}`. (The order of the keys and values does not matter.)

19. Write a method called `rarest` that accepts a map whose keys are strings and whose values are integers as a parameter and returns the integer value that occurs the fewest times in the map. If there is a tie, return the smaller integer value. If the map is empty, throw an exception.

20. Write a modified version of the `Vocabulary` program developed in Chapter 10 that uses sets rather than `ArrayLists` to store its words. (The program will be noticeably shorter and will run faster!)

Programming Projects

1. Write a program that computes the edit distance (also called the Levenshtein distance, for its creator Vladimir Levenshtein) between two words. The edit distance between two strings is the minimum number of operations that are needed to transform one string into the other. For this program, an operation is a substitution of a single character, such as from "brisk" to "brick". The edit distance between the words "dog" and "cat" is 3, following the chain of "dot", "cot", and "cat" to transform "dog" into "cat". When you compute the edit distance between two words, each intermediate word must be an actual valid word. Edit distances are useful in applications that need to determine how similar two strings are, such as spelling checkers.

   Read your input from a dictionary text file. From this file, compute a map from every word to its immediate neighbors, that is, the words that have an edit distance of 1 from it. Once this map is built, you can walk it to find paths from one word to another.

   A good way to process paths to walk the neighbor map is to use a linked list of words to visit, starting with the beginning word, such as "dog". Your algorithm should repeatedly remove the front word of the list and add all of its neighbors to the end of the list, until the ending word (such as "cat") is found or until the list becomes empty, which indicates that no path exists between the two words.

2. Write a program that solves the classic "stable marriage" problem. This problem deals with a group of men and a group of women. The program tries to pair them up so as to generate as many stable marriages as possible. A set of marriages is unstable if you can find a man and a woman who would rather be married to each other than to their current spouses (in which case the two would be inclined to divorce their spouses and marry each other).

   The input file for the program will list all of the men, one per line, followed by a blank line, followed by all of the women, one per line. The men and women are numbered according to their positions in the input file (the first man is #1, the second man is #2, and so on; the first woman is #1, the second woman is #2, and so on). Each input line (except for the blank line separating men from women) lists the person's name, followed by a colon, followed by a

list of integers. These integers are the marriage partner preferences of this particular person. For example, see the following input line in the men's section:

```
Joe: 10 8 35 9 20 22 33 6 29 7 32 16 18 25
```

This line indicates that the person is named "Joe" and that his first choice for marriage is woman #10, his second choice is woman #8, and so on. Any women not listed are considered unacceptable to Joe.

The stable marriage problem is solved by the following algorithm:

```
assign each person to be free.
while (some man M with a nonempty preference list is free) {
    W = first woman on M's list.

    if (some man P is engaged to W) {
        assign P to be free.
    }
    assign M and W to be engaged to each other.

    for (each successor Q of M who is on W's list) {
        delete W from Q's preference list.
        delete Q from W's preference list.
    }
}
```

Consider the following input:

```
Man 1: 4 1 2 3
Man 2: 2 3 1 4
Man 3: 2 4 3 1
Man 4: 3 1 4 2
Woman 1: 4 1 3 2
Woman 2: 1 3 2 4
Woman 3: 1 2 3 4
Woman 4: 4 1 3 2
```

The following is a stable marriage solution for this input:

```
Man 1 and Woman 4
Man 3 and Woman 2
Man 2 and Woman 3
Man 4 and Woman 1
```

3. Write a program that solves the classic "random writer" problem. This problem deals with reading input files of text and examining the frequencies of characters. On the basis of those frequencies, you can generate randomized output that appears to match the writing style of the original document. The longer the chains you link together, the more accurate the random text will sound. For example, level 4 random text (text with chains of 4 letters long) generated from *Tom Sawyer* might look like this: "en themself, Mr. Welshman, but him awoke, the balmy shore. I'll give him that he couple overy because in the slated snufflindeed structure's kind was rath. She said that the wound the door a fever eyes that WITH him." Level 10 random text from the same source might look like this: "you understanding that they don't come around in the cave should get the word beauteous was over-fondled, and that together and decided that he might as we used to do—it's nobby fun. I'll learn you." Search the Internet for "Random Writer" to learn more about this problem, such as the specification posed by computer scientist Joseph Zachary.

The program produces the following output for *Moby-Dick:*

```
This program displays the most
frequently occurring words from
the book Moby-Dick.

a occurs 4509 times.
and occurs 6138 times.
his occurs 2451 times.
in occurs 3975 times.
of occurs 6405 times.
that occurs 2705 times.
the occurs 13991 times.
to occurs 4433 times.
```

Collection Overview

We've discussed three major abstract data types in this chapter: lists, sets, and maps. It's important to understand the differences between them and to know when each should be used. Table 11.6 summarizes the different types of collections and their pros and cons.

When you approach a new programming problem involving data, you can ask yourself questions to decide what type of collection is most appropriate. Here are some examples:

- What are you going to do with the data? Do you intend to add and remove many elements, search the data many times, or connect this data to other data?

Table 11.6 Comparison of Lists, Sets, and Maps

ADT	Implementations	Description/Strengths	Weaknesses	Example Usages
`List`	`ArrayList`, `LinkedList`	A sequence of elements arranged in order of insertion	Slow to search, slow to add/remove arbitrary elements	List of accounts; prime numbers; the lines of a file
`Set`	`HashSet`, `TreeSet`	A set of unique elements that can be searched quickly	Does not have indexes; user cannot retrieve arbitrary elements	Unique words in a book; lottery ticket numbers
`Map`	`HashMap`, `TreeMap`	A group of associations between pairs of "key" and "value" objects	Not a general-purpose collection; cannot easily map backward from a value to its key	Word counting; phone book creation