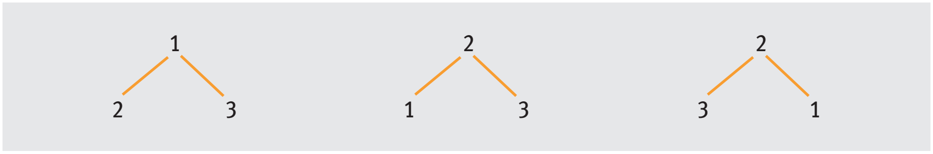
[CHAPTER] 8

SET AND GRAPH Data Structures

8.1 Sets

We have completed our study of two of the four data structures introduced in Section 6.1—the linear (Chapter 6) and hierarchical (Chapter 7). In this chapter, we investigate the two remaining groupings in the taxonomy—sets and graphs. Sets are introduced in Sections 8.1 and 8.2; graphs are discussed in Section 8.3.

In the data structures you have studied so far, the position of elements is an essential property of the collection. For example, consider the following three binary trees:



The trees are different, not because of the elements they contain—they all hold the three integers 1, 2, and 3—but because of differences in the *location* of the values (in other words, root versus leaf, right child versus left child). However, in a set we do not care about an element's exact position within the collection.

Formally, a **set** is a collection of objects with the following two properties:

- There are no duplicates.
- The order of objects within the collection is immaterial.

Because of this location-independent property, the following two sets S' and S":

$$S' = \{1, 7, 13, -8, 105, 99\}$$
$$S'' = \{7, 99, 13, -8, 105, 1\}$$

are considered identical, even though the elements appear in different order.[1]

This is the fundamental difference between sets and the linear and hierarchical structures you studied previously. The latter two groups required an explicit ordering of elements. In the linear structures of Chapter 6, these ordering relationships were called first, last, successor, and predecessor. So, for example, if the two sets S' and S" were linear structures, then the successor of 7 would be 13 in S' but 99 in S". The same applies to the hierarchical structures of Chapter 7, whose ordering relationships were termed parent, child, and sibling. In a set structure, none of these positional mappings exist. Checking for the successor of 7 in S' or the first element of S" is meaningless. The entries in a set are unrelated to each other, except by their common membership within the overall collection.

[1] As mentioned in Chapter 6, a set that permits duplicate elements is called a bag or multiset. These structures are not supported in the Java Collection Framework and are not discussed here.

8.1.1 Operations on Sets

The operations on sets are well known, and it is easy to describe a standard interface that includes virtually all the important operations. (They have been studied since the 1880s as part of a branch of mathematics called **naïve set theory**, developed by the German logician and mathematician Georg Cantor.) This standardization differs markedly from the structures you studied in earlier chapters, such as lists and binary trees. Those structures had a large number of possible operations, and determining which ones to include was highly application dependent. The specifications for a `Set` interface are shown in Figure 8-1.

```
/**
 * An interface for a set, which is a collection of objects that
 * contains no duplicates and has no order.
 */
public interface Set<T> {
    /**
     * Add the given element to this set.
     *
     * Preconditions:
     *   This set is not full and e is not null.
     *
     * Postconditions:
     *   e is contained in this set.
     *
     * @param e the element to add.
     * @return true if e was added to the set.
     */
    public boolean add( T e );

    /**
     * Remove the given element from this set.
     *
     * Preconditions:
     *   This set is not empty and e is not null.
     *
     * Postconditions:
     *   e is not contained in this set.
     *
     * @param e the element to remove from this set.
     * @return true if e was removed from the set.
     */
    public boolean remove( T e );
```

continued

```
    /**
     * Add to this set all elements of the given set.
     *
     * Preconditions:
     *   s is not null.
     *
     * Postconditions:
     *   This set contains the union of this set and s.
     *
     * @param s the set whose elements are to be added.
     */
    public void union( Set<T> s );

    /**
     * Remove from this set all elements not in the given set.
     *
     * Preconditions:
     *   s is not null.
     *
     * Postconditions:
     *   This set contains the intersection of this set and s.
     *
     * @param s the set to intersect with.
     */
    public void intersection( Set<T> s );

    /**
     * Remove all elements in this set that are members of the given
     * set (i.e., retain those elements not in the given set).
     *
     * Preconditions:
     *   s is not null.
     *
     * Postconditions:
     *   This set contains the difference of this set and s.
     *
     * @param s the set to 'subtract' from this set.
     */
    public void difference( Set<T> s );

    /**
     * Determine whether this set is empty.
     *
     * Preconditions:
     *   None
     *
```

continued

```
     * Postconditions:
     *   The set is unchanged.
     *
     * @return true if this set is empty.
     */
    public boolean isEmpty();

    /**
     * Determine if an object is a member of this set.
     *
     * Preconditions:
     *   e is not null.
     *
     * Postconditions:
     *   The set is unchanged.
     *
     * @param e the object to check for set membership.
     * @return true if the given element is a member of this set.
     */
    public boolean contains( T e );

    /**
     * Determine if this set is a subset of a given set.
     *
     * Preconditions:
     *   s is not null.
     *
     * Postconditions:
     *   The set is unchanged.
     *
     * @param s the set to be tested for membership.
     * @return true if this set is a subset of the given set.
     */
    public boolean subset( Set<T> s );

    /**
     * Return an array that contains the elements in this set.
     *
     * Preconditions:
     *   None
     *
     * Postconditions:
     *   The set is unchanged.
     *
     * @return an array representation of this set.
     */
    public T[] toArray();
```

continued

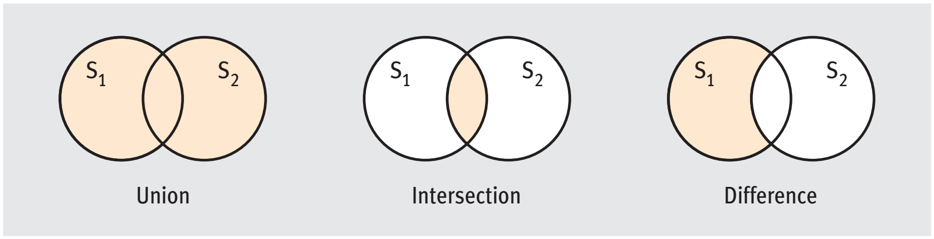
```
    /**
     * Determine the size of this set.
     *
     * Preconditions:
     *   None
     *
     * Postconditions:
     *   The set is unchanged.
     *
     * @return the number of elements in this set.
     */
    public int size();

} // Set
```

[FIGURE 8-1] A `Set` interface

The interface of Figure 8-1 contains five mutator methods. The Boolean method `add(e)` places a new element `e` into the set if it is not already there, and returns true. Because duplicates are not permitted, the method makes no change to the collection and simply returns false if `e` is already a member of the set. This is not considered an error, but simply a fundamental characteristic of sets. The `remove(e)` method removes the element `e` from the set if it is there and returns true. If `e` is not a member of the set, then nothing is done, and the method returns false. Again, this is not treated as an error.

Three mutator methods combine elements of two sets to produce a new set. The **union** operator, written S1 ∪ S2, produces a set whose members include all objects that are members of *either* set S1 or set S2 or both, eliminating any duplicates. The **intersection** operator, written S1 ∩ S2, produces a new set whose members are all objects that belong to *both* set S1 and set S2, again eliminating duplicates. Finally, the **difference** operator, written S1 − S2, constructs a set whose members include only objects that are members of set S1, but not members of set S2.

The behavior of these three mutator methods is diagrammed in Figure 8-2 using a notation called a **Venn diagram**. The shaded areas of each diagram represent the elements included in the newly constructed set.



[FIGURE 8-2] Venn diagrams of three basic set methods

The three observer methods in the interface of Figure 8-1 are `isEmpty()`, `contains()`, and `subset()`. The `isEmpty()` method returns true if this set is the **empty set** (in other words, it has no members), and false otherwise. (The empty set is written {}.) The method `contains(e)` returns true if `e` is a member of this set and false otherwise. Finally, `subset(S)` returns true if every element in set S is also a member of this set; otherwise, it returns false. You can use this method to determine if sets S1 and S2 are equal. Simply evaluate `S1.subset(S2)` and `S2.subset(S1)`. If both return true, then S1 is equal to S2, meaning they contain exactly the same elements in their collection.

The following table lists example behaviors of the methods included in the `Set` interface of Figure 8-1. Each example assumes the following starting values:

S = {1, 2, 3, 4} T = {3, 4, 5, 6} U = {} V = {2, 3}

METHOD	RESULT	METHOD RETURN VALUE
`S.add(6)`	S = {1, 2, 3, 4, 6}	true
`S.add(4)`	S = {1, 2, 3, 4}	false
`T.remove(3)`	T = {4, 5, 6}	true
`T.remove(2)`	T = {3, 4, 5, 6}	false
`S.union(T)`	S = {1, 2, 3, 4, 5, 6}	void
`S.intersection(T)`	S = {3, 4}	void
`S.difference(T)`	S = {1, 2}	void
`T.difference(S)`	T = {5, 6}	void
`T.isEmpty()`	No change in T	false
`U.isEmpty()`	No change in U	true
`T.contains(2)`	No change in T	false
`T.contains(3)`	No change in T	true
`S.subset(V)`	No change in S	true
`S.subset(T)`	No change in S	false

We have included two helpful utility methods in the `Set` interface of Figure 8-1. If $n > 0$, the `toArray()`method takes the n elements contained in the set and stores them linearly in an n-element array. (Because order is immaterial in a set, the placement order of array elements is also immaterial.) The `size()` method returns the total number of elements in the set, often called the **cardinality** of the set.

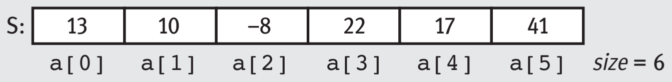
8.1.2 Implementation of Sets

Two widely used techniques for implementing sets are arrays and linked lists. This is similar to the two choices for implementing lists in Section 6.2.3 and for implementing binary trees in Section 7.3.

We can store the elements of a set S in an unordered, one-dimensional array structure `a`, created as follows:

```
private final int MAX = ... ; // Maximum number of elements
private Object a[MAX] = new Object[MAX];
private int size;    // The actual number of elements in a[]
```

Using these declarations, the six-element set S = $\{10, 13, 41, -8, 22, 17\}$ might be stored into array `a` as follows, assuming that MAX $\geq$ 6:



Note that the order of array elements does not need to match the order in which elements were inserted into the set, because order is immaterial.

When we use the array implementation described earlier, the methods in the `Set` interface of Figure 8-1 generally translate into sequential searches of the array. For example, to determine if element `e` is contained in set S, we search array `a` from beginning to end until either `e` is found or we come to the end of the array. Similarly, to add an element `e` to set S, we first search the array to see if `e` is present. If it is, we do nothing. If `e` is not present, we store `e` at the end of the array, assuming that the array is not full. (If it is, we have either caused an error or we must resize the array using the techniques described in Chapters 6 and 7):

```
a[size] = e;
size++;
```

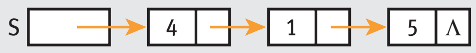
The `remove()` method determines if `e` is present and, if so, removes it. The method then moves all array elements that follow `e` up one position and decrements `size` by 1. Thus, in our array-based implementation, the three methods `add`, `remove`, and `contains` are all O(n), where n is the number of elements in set S.

Figure 8-3 shows the array-based implementation of the intersection method `S1.intersection(S2)` described in the previous section. For each element `e` in S1, we search S2 to see if `e` is present. If it is, we keep the element in S1; if not, we remove it. If S1 and S2 are both unordered, and each contains n elements, then the intersection algorithm of Figure 8-3 is $O(n^2)$. (We can do better if both S1 and S2 are sorted, as shown in Exercise 3 at the end of the chapter.) Using approaches similar to those in Figure 8-3, it is easy to see that the union, difference, and subset operators also require $O(n^2)$ time to complete.

```
public void intersection( Set S2 ) {
    for ( int i = 0; i < size; i++ ) {
        if ( !S2.contains( a[ i ] ) ) {
            remove( a[ i ] );
            i--;  // Don't skip the next element
        }
    }
}
```

[FIGURE 8-3] The array implementation of set intersection

The second approach to implementing a set is to represent it as a singly linked list of the type described in Section 6.2. Using a list representation, the set S = {1, 5, 4} might look like the following:



Because order is immaterial, we must again search the entire linked list to carry out the methods in the `Set` interface of Figure 8-1. For this reason, `subset()`, `intersection()`, `union()`, and `difference()` are still $O(n^2)$, where n is the number of elements in each set, while the `contains()` and `remove()` functions remain $O(n)$.

At first it may seem that the `add(e)` method can be completed in O(1) time, because it attaches a new element `e` to the head of a linked list; as we showed in Chapter 6, this can be done in constant time. However, remember that duplicates are not allowed in a set, so before adding the new element `e`, we must first search the list to ensure it is not already there—making this an $O(n)$ operation as well. The following table summarizes the complexity of the 10 methods in Figure 8-1. These complexities hold whether the chosen implementation is an array or a linked list.

METHOD	COMPLEXITY
`add`	O(n)
`remove`	O(n)
`union`	O(n^2)
`intersection`	O(n^2)
`difference`	O(n^2)
`isEmpty`	O(1)
`contains`	O(n)
`subset`	O(n^2)
`toArray`	O(n)
`size`	O(1)

You have another alternative for implementing a set besides using arrays and linked lists. Under specialized conditions, this alternative can produce significant improvements over the first two methods.

If we make two assumptions about the nature of the set's **base type**—that is, the data type of the objects contained in the set—we can use an extremely efficient representation called a **bit vector**. The two assumptions are: (1) the total number, n, of values that belong to this base type is relatively small, and (2) there is a 1:1 function f that maps the n elements of the base type to the integer values $[0 \ldots n - 1]$:

$f: e \rightarrow I$ where: e is an element of the base type of a set with cardinality N
I is an integer in the range [0 . . . N – 1]

such that if $e_1 \neq e_2$, $f(e_1) \neq f(e_2)$.

Examples of base types that satisfy these two assumptions are the Java primitive types character and Boolean, and a small subrange of integers.

To implement a set S of cardinality n, we first create a Boolean array `a` of length n. Each element i of the array, true or false, indicates whether the unique element `e` that maps to location $i = f($`e`$)$ is a member of set S.

For example, given the following declarations:

```
private final int N = ... ;    // The size of the base type
private boolean a[] = new boolean[N];
```

and given a function *f* that maps elements of the base type to locations 0 … $n - 1$ in array `a`, there is a highly efficient way to implement the set methods described in the previous section. To insert a new element `e` into set S, just set array element `a`[*f*(`e`)] to true. (If it was true to begin with, it remains true, so there are no duplicate elements.) To remove element `e` from set S, set `a`[*f*(`e`)] to false. To determine whether `e` is contained within S, examine array location `a`[*f*(`e`)]. If it is true, `e` is a member of set S; if it is false, `e` is not a member. These are all O(1) operations.

As an example of the bit vector implementation, let's create a set whose base type is the days of the week: "monday", "tuesday", …, "sunday". Because a week contains seven days, one possible declaration for `s2`, the bit vector implementation of this set, is the following:

```
private boolean s2[] = new boolean[7];  // s2 is a bit
                                        // vector
```

The function *f* would perform the following mappings:

f("monday") = 0
f("tuesday") = 1
…
f("sunday") = 6

Given these values, the bit vector representation of the set {"monday", "wednesday", "friday"} looks like this:

true	false	true	false	true	false	false
s2[0]	s2[1]	s2[2]	s2[3]	s2[4]	s2[5]	s2[6]

To determine whether "tuesday" is a member of `s2`, which is just the method `contains("tuesday")` in Figure 8-1, we examine array element `s2[f("tuesday")] = s2[1]`, which currently is false, indicating that "tuesday" is not a member. To delete the element "friday", we set `s2[f("friday")] = s2[4]` to false.

As a second example of the use of bit vectors, let's assume that the base type of our set is the 26 lowercase alphabetic values ['a'... 'z'], which have Unicode values [97 ... 122]. The declarations for the bit vector implementation of this set are:

```
private boolean s3[] = new boolean[26];
```

and the mapping function *f* would be:

```
f(e) = Character.getNumericValue(e) - 97;
```

where `getNumericValue(e)` is the Java library function that returns the integer value of the Unicode character `e`.

Using this bit vector approach, you can implement the four set methods `subset()`, `intersection()`, `union()`, and `difference()` in $O(n)$ time, where n is the size of the two sets. For example, the complexity of the intersection operator in Figure 8-3 is $O(n^2)$, but we can do much better using bit vectors. We use a logical **and** (&&) between corresponding elements of the two bit vector arrays. If both locations are true, the result is true; otherwise, the result is false:

```
// The intersection of set S2 and this set, where
// both are implemented as bit vectors s of size N
for (int i = 0; i < N; i++)
     s[i] = (s[i] && S2.s[i]);
return;
```

This is a more efficient $O(n)$ implementation of intersection than the array or linked list techniques described earlier, both of which required $O(n^2)$ comparisons. Similarly, the set union method can be completed in $O(n)$ time by using a logical **or** (||) between correspon-ding elements of the two bit vector arrays.

Figure 8-4 compares the efficiency of the array and linked list methods with that of the bit vector implementation. In every case, except for `isEmpty()`, the bit vector delivers equal or more efficient behavior. (The `isEmpty()` method is $O(n)$ because it must separately test all n elements of the bit vector to see if they are false. We could reduce it to $O(1)$ if we kept an auxiliary state variable `size` that kept track of the number of true values contained in the bit vector.)

METHOD	ARRAY/LIST COMPLEXITY	BIT VECTOR COMPLEXITY
`add`	$O(n)$	$O(1)$
`remove`	$O(n)$	$O(1)$
`union`	$O(n^2)$	$O(n)$
`intersection`	$O(n^2)$	$O(n)$
`difference`	$O(n^2)$	$O(n)$
`isEmpty`	$O(1)$	$O(n)$
`contains`	$O(n)$	$O(1)$
`subset`	$O(n^2)$	$O(n)$
`toArray`	$O(n)$	$O(n)$
`size`	$O(1)$	$O(1)$

[FIGURE 8-4] Complexities of different set implementations

However, while highly efficient, bit vectors have limited usage because of their assumptions—namely, the cardinality of the set's base type is small. Sets often draw their components from extremely large base types, such as **int**, which has 4 billion elements. This can limit the practical use of bit vectors as a way to implement sets.

The Java library includes the class `BitSet`, which is a bit vector implementation of a set. Instead of using a Boolean array to store the true or false values, `BitSet` uses an array of *bits*, with each individual bit representing false (binary 0) or true (binary 1). The constructor for the class, `BitSet(int nbits)`, allows you to create a bit vector of size `nbits` that is indexed by the nonnegative integer values 0 to `nbits` − 1. The mapping function *f* is simply the identify function $f: i \rightarrow i$, where i is in the range [0 ... `nbits` − 1]. Once you have created the bit vector, you can implement all the methods described in Figure 8-1.

A. M. TURING (1912–1954)

Alan M. Turing, British mathematician and logician, is widely considered the father of modern computer science. Turing was born in London in 1912. His mathematical genius quickly became evident; at the age of 16, he began studying and critiquing the works of Einstein and others. He did his undergraduate work at Cambridge and was made a fellow of King's College for the brilliance of his undergraduate thesis. Soon after graduation he wrote a world-famous paper, "On Computable Numbers, with an Application to the *Entscheidungsproblem*" (the decidability problem). In the paper he described a model of computation, now called a *Turing machine*, that solves any mathematical problem that can be formulated as an algorithm. To this day, Turing machines are a central topic of study in theoretical computer science. Turing received his PhD from Princeton University in 1938.

When World War II broke out, Turing was recruited by the British government to decipher German military codes, including the famous Enigma code used by the German Navy. He moved to Bletchley Park, a secret code-breaking institute, and within a few weeks had helped construct an electromechanical device, called the *Bombe*, that cracked the Enigma code and contributed significantly to the Allied victory. For his invaluable wartime service, Turing was awarded the Order of the British Empire.

Following the war, Turing became interested in automatic computation and did an enormous amount of pioneering work in the emerging field of computing. In 1946 he wrote a landmark paper containing the first complete design of a stored-program computer. His theoretical ideas were realized in 1949 with the construction of the Manchester Mark I, one of the earliest digital computers. He also studied philosophy and computer intelligence. He contributed to the creation of artificial intelligence by devising the *Turing Test*—a way to investigate the question "Can machines think?" He is also said to have written the first chess-playing program, although computers of that era were not powerful enough to implement his ideas. Turing did pioneering work in many other areas of computing, including set theory, cryptanalysis, algorithms, and mathematical biology.

Turing's life did not end happily. He was homosexual at a time when such behavior was viewed as a crime against humanity. In 1952 he was arrested by British police and charged with gross indecency. He lost his security clearance, much of his professional work, and was shunned by many colleagues. He was also required to undergo medical treatments that, at the time, were believed to "cure" homosexuality. Instead they produced painful and unpleasant side effects that eventually caused him to commit suicide

continued

by eating an apple laced with cyanide. A highly successful play detailing his life and tragic death, entitled “Breaking the Code,” opened in London in 1986 and moved to Broadway the following year.

For his pioneering work, the ACM, the professional society of computer science, named its most prestigious award the **A. M. Turing Award**. The honor is considered the Nobel Prize of computing.

8.2 Maps

8.2.1 Definition and Operations

Sets are an important concept in formal mathematics, and are studied extensively in the branch of mathematics called *set theory*. However, the set structure described in Section 8.1 is not widely used, and the `intersection()`, `union()`, and `difference()` methods in Figure 8-2 do not occur very often in computer science applications. (Nevertheless, we show an interesting use of sets when developing the minimum spanning tree algorithm in Section 8.3.2.3.)

That does not mean, however, that sets have no role in software development. On the contrary, you can refine the definitions in Section 8.1 to produce an important variation of the set that is widely used in computer science. This new structure is called a **key-access table**, more commonly referred to as a **map**.

Let’s make the following three modifications to our original definition of a set:

1 The elements of the set are ordered pairs (k_i, v_i), where k_i is the **key field** and v_i is the **value field**. These ordered pairs are often referred to as **2-tuples**, or just **tuples**. There is no limit to the number of tuples in a set, although it must be finite, of course. Thus, our new view of set S, which we call a **map**, is:

$$S = \{ (k_0, v_0), (k_1, v_1), \ldots, (k_n, v_n) \} \qquad n < \infty$$

2 The k_i are of type `keyType`, and the v_i are of type `valueType`. Neither `keyType` nor `valueType` is limited to primitive types; they can be any objects. The k_i must be unique within the set, but the v_i need not be. In addition, the k_i are considered *immutable*, and their value should not be changed once they have been added to the map. The v_i may change as often as desired.

3 We are no longer interested in the traditional set methods of `union()`, `intersection()`, `difference()`, and `subset()` in Figure 8-1. Instead, we will focus on the `put`, `remove`, and `get` methods. These methods place new tuples into the map, remove tuples from the map, and locate a specific tuple within the map. As we will see, however, these methods can be defined in terms of the classical set methods of `union`, `intersection`, and `difference`.

The `put()` method takes a map `S`, a key `k`, and a value `v`. It adds the tuple `(k,v)` to `S` if `k` is not the key field of any tuple currently contained in `S`; otherwise, `S` is unchanged.[2] Essentially, `put(k,v)` performs the following operation, in which the wildcard symbol (*) matches anything:

```
if there is not a tuple of the form (k,*)
    S = S ∪ {(k,v)}
else
    do nothing
```

The `remove()` method takes a map `S` and a key `k`. If there is a tuple in `S` of the form `(k, *)`, where the asterisk again is a wildcard that represents any value field, then this tuple is removed from `S`. If there is no tuple of the form `(k, *)`, then `S` is unchanged. `Remove(k)` is equivalent to the following set operation:

```
S = S - {(k, *)}    // Where '*' matches any value
```

Finally, `get()` takes a map `S` and a key value `k`. If there is a tuple anywhere in `S` of the form `(k,v)`, then this method returns the corresponding value field `v`; otherwise, it returns **null**, indicating there was no tuple of the form `(k,*)` in `S`. Essentially, the operator `v = get(k)` carries out the following operation:

```
if there is a tuple of the form (k, *)
     // getValueField returns the second
     // element of the tuple
     val = getValueField((k,v));
     return val;
else
     return null;
```

[2] You can also define the `put` operator so that if the tuple already exists in the map, it updates the `v` field of the existing tuple.

The map structure represents a collection in which all data values are associated with a unique identifier called the **key** and are stored as a 2-tuple (key, value). All retrievals are done using the associated key rather than location within the structure. Because the key used to retrieve a value is stored within the tuple itself, the position of the tuple within the map is immaterial. In addition, because all keys are unique, every tuple is unique; there are no duplicates. Together, these properties demonstrate that the map structure satisfies the definition of a set given in the previous section.

However, instead of having limited use, a map is an extremely useful data structure. This type of "keyed access" models a number of data-related problems encountered frequently in computer science. For example, all of the following are common data collections:

KEY FIELD	VALUE FIELDS
Student ID No.	(Name, Major, GPA, Year)
Social Security No.	(Name, Address, Occupation)
Part No.	(Part Name, Supplier, Amount in Hand)
Confirmation No.	(Flight No., Seat No., Departure Time)
License Plate No.	(Owner, Make, Year, Color, Fees Paid)
Process No.	(Process Name, Process State, Owner, Resources Used)

Typically, these collections require the user to provide a key, and we either retrieve the value field(s) associated with that key or discover that the key is not present. This is exactly what happens when someone retrieves your student data, tax records, or flight booking information based on a unique identification number that you provide.

Figure 8-5 is an interface for the key-access data structure just described. It includes the three methods `put()`, `remove()`, and `get()` described earlier in this section. Two additional utility methods, `isEmpty()` and `size()`, have the same function as in Figure 8-1. The interface also includes methods called `getKeys()` and `getValues()`, which copy the key fields and value fields, respectively, from the map into a list.

```
/**
 * An interface for the table data structure, also called a map
 */
public interface Table<K,V> {
    /**
     * Puts the given value into the table, indexed by the given key.
     *
```

continued

```
     * Preconditions:
     *   Key is not null
     *   Value is not null
     *
     * Postconditions:
     *   The given value has been associated with the given key.
     *
     * @param key the key to refer to the given value by.
     * @param value the data to associate with the given key.
     */
    public void put( K key, V value );

    /**
     * Remove the object associated with the given key, if any.
     *
     * Preconditions:
     *   Key is not null
     *
     * Postconditions:
     *   The element (if any) identified by key has been removed.
     *
     * @param key the key to remove.
     */
    public void remove( K key );

    /**
     * Return the object associated with the given key.
     *
     * Preconditions:
     *   Key is not null
     *
     * Postconditions:
     *   The table is unchanged.
     *
     * @param key the key to look up in the table.
     *
     * @return the object associated with the key or null if the key
     *         is not in the table.
     */
    public V get( K key );

    /**
     * Determine whether the table is empty.
     *
     * Preconditions:
     *   None
     *
```

continued

```
     * Postconditions:
     *   The table is unchanged.
     *
     * @return true if the table is empty.
     */
    public boolean isEmpty();

    /**
     * Determine the size of the table.
     *
     * Preconditions:
     *   None
     *
     * Postconditions:
     *   The table is unchanged.
     *
     * @return the number of elements in the table.
     */
    public int size();

    /**
     * Determine the cardinality of the table.
     *
     * Preconditions:
     *   None
     *
     * Postconditions:
     *   The table is unchanged.
     *
     * @return a number one greater than the maximum domain element.
     */
    public int cardinality();

    /**
     * Get all the keys in this table.
     *
     * Preconditions:
     *   None
     *
     * Postconditions:
     *   The table is unchanged.
     *
     * @return a list containing all of the keys in the table.
     */
    public List<K> getKeys();
```

continued

```
    /**
     * Get all values stored in this table. There is no correlation
     * between the order of the elements returned by this method and
     * the getKeys() method.
     *
     * Preconditions:
     *   None
     *
     * Postconditions:
     *   The table is unchanged.
     *
     * @return a list containing all of the values in the table.
     */
    public List<V> getValues();

} // Table
```

[FIGURE 8-5] Interface specification for a map

In the next section, you will learn how to implement this structure and provide additional examples of its use.

8.2.2 Implementation Using Arrays and Linked Lists

There are a number of straightforward ways to implement the `Map` interface of Figure 8-5. As we will see, however, these simple approaches often produce inefficient and unacceptable performance.

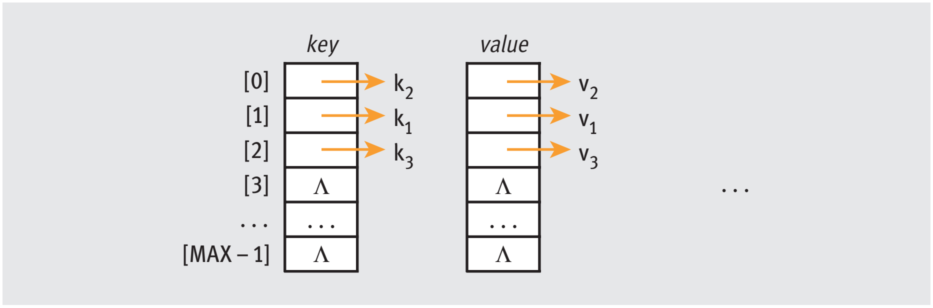
The most obvious approach is to implement the map using two parallel, one-dimensional arrays called `key` and `value`. Each row of the `key` array holds a key value, while the corresponding row of the `value` array holds its associated value field. Thus, the two elements `key[i]` and `value[i]` together represent a single tuple.[3]

Declarations to create this parallel array-based implementation are shown in the following code. It uses data types called `K` and `T`, which are the parameterized types defined in the Table interface in Figure 8-5:

```
    private final int MAX = ... ;
    private K key[] = (K[])new Object[MAX];
    private T value[] = (T[])new Object[MAX];
```

[3] The value field may contain multiple items rather than a single object, like the examples shown on the previous page. In that case, the `value` array may need to be a multidimensional structure. To keep our discussions simple, we will assume that the value field contains a single object.

A map S = $\{(k_1,v_1), (k_2,v_2), (k_3,v_3)\}$ might look like the diagram shown in Figure 8-6. (The order of tuples in the array is not the same, but the order of values in a map is immaterial, so it should not make any difference.)



[FIGURE 8-6] A map implemented using two parallel arrays

We have two choices for maintaining information in these two arrays. The first choice is to keep the tuples sorted in order by key field k_i.

This assumes that objects of type `KeyType` *implement the* `Comparable` *interface, which implies that they can be compared to each other and put into a natural ordering.*

For example, if we are storing tuples in ascending order, then the key values in Figure 8-6 must satisfy the relationship $k_2 < k_1 < k_3$. If we choose to keep keys in sorted order, then we may need to move entries up or down to add or remove a tuple. This would be necessary, for example, to insert a new tuple (k_4,v_4) into the two arrays where $k_2 < k_4 < k_1$. This insertion requires us to move both (k_1,v_1) and (k_3,v_3) down one slot to make room for the new tuple, which takes O(n) time. We saw the same situation with the array implementation of a list in Section 6.3.2.1. Keeping the two arrays sorted by key allows the `get` method to use an efficient O(log n) binary search when it attempts to locate a key and retrieve its associated value.

What if we keep the tuples in random, unsorted order and simply place new tuples at the end of the arrays? Unfortunately, this does not help, because we must check whether the key is already in the array before we can store a new value. Thus, insertion is still O(n). If the keys are in no particular order, both `remove()` and `get()` require a sequential search to locate a specific tuple, which also takes O(n) time.

A linked list implementation does not fare any better. Insertions are O(n) because of the time required to check for duplicates. The `remove()` and `get()` methods also require O(n) time, as they must traverse the list to locate the correct tuple. A linked list also requires additional memory space for the reference fields in each node.

The performance of the basic `Map` methods using unsorted arrays, sorted arrays, and linked lists is summarized in Figure 8-7a.

METHOD	RUN-TIME PERFORMANCE UNSORTED ARRAY	SORTED ARRAY	LINKED LIST
`put`	O(n)	O(n)	O(n)
`remove`	O(n)	O(n)	O(n)
`get`	O(n)	O($\log n$)	O(n)

[FIGURE 8-7a] Performance of array and linked list implementations of a map

None of these implementations appears to work very well. All operations are O(n), except the logarithmic time of the `get` methods on a sorted array, which is enabled by the efficient binary search algorithm of Figure 5-5.

Another possibility is to use the binary search tree from Section 7.4. We could use the key field of the `(k,v)` tuples to order the tree. If the tree is reasonably well balanced, then insertions, deletions, and removals can all be completed in O($\log n$) time, as shown in Figure 7-19b. However, remember that the worst-case performance of a binary search tree is O(n), so we cannot be guaranteed of achieving logarithmic performance (see Figure 7-19a) unless we use one of the balanced search trees discussed in Section 7.5, such as the red-black tree. Then we can be sure that all map methods run in O($\log n$) time, as shown in Figure 8-7b.

METHOD	PERFORMANCE
`put`	O($\log n$)
`remove`	O($\log n$)
`get`	O($\log n$)

[FIGURE 8-7b] Performance of balanced binary search tree implementations of a map

This is a significant improvement over the array and linked list implementations, whose performance was summarized in Figure 8-7a.

So, which approach should we choose to implement the `Map` interface of Figure 8-5? Surprisingly, the answer is "None of the above!" An alternative implementation usually demonstrates superior performance to all the previous methods. Its performance with the three fundamental methods for maps (at least theoretically) is:

METHOD	PERFORMANCE
`put`	O(1)
`remove`	O(1)
`get`	O(1)

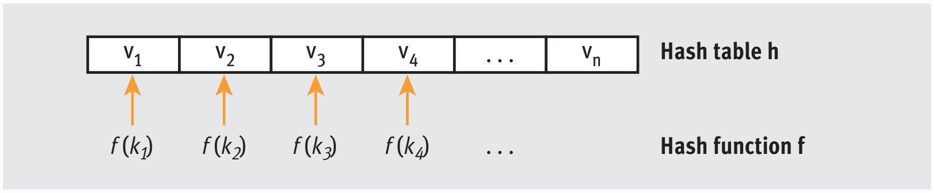
You can't do any better than that. This new technique, called **hashing**, is the most popular method for building and maintaining a map data structure.

8.2.3 Hashing

Hashing works by transforming a key field `k` into a number called a **hash value** in the range $[0 \ldots n - 1]$. It then uses this hash value as an index to determine where in an n-element array `h`, called a **hash table**, we should store the value field. That is, given a `(k,v)` tuple, we apply a function f, called a **hash function**, to the key `k` to obtain an index $i = f(k)$, and we store the value field `v` into location `h[i]`.

To retrieve the value associated with key `k` from `h`, we compute $i = f(k)$ and examine the contents of location `h[i]`. If it contains a non-null value, then the tuple is in the table, and we return the contents of `h[i]`, which must be the value field associated with this key. If `h[i]` is empty, then the tuple `(k,v)` is not in the table, and we return **null** to indicate its absence. (This assumes that every element of the hash table `h` has been initialized to the "empty" state.) Also note that only the value is stored in the array. The key field is only used by the hash function f to locate the value field in `h`.

The logical structure of a hash table is diagrammed in Figure 8-8.



[FIGURE 8-8] Logical structure of a hash table

One of the most important aspects of hashing is the selection of a **hashing function** *f* that maps keys to array locations. As an example, assume that we are using people's names as key values. We can associate a number with each letter; for example, 'a' and 'A' are assigned the value 1, 'b' and 'B' are assigned the value 2, and so on. If the k letters in a name are designated $c_1, c_2, \ldots, c_k$, and the function `intValue(letter)` maps 'a', …, 'z' and 'A', …, 'Z' to their associated integer value 1, 2, …, 26, then a possible hashing function *f* is given by the following:

$$f = [\sum_{i=1}^{k} \texttt{intValue}\ (\texttt{c}_{\texttt{i}})]\ \%\ n \qquad \text{where } n = \text{the hash table size and \% is the modulo function}$$

To see how this hashing function operates, assume a hash table size $n = 50$, key values that are names, and value fields that contain information about that person. Each name field must hash to an integer value in the range [0 ... 49] that corresponds to the location in the hash table where information about that person is stored. Figure 8-9a shows the application of the preceding hash function *f* to the name "Smith".

LETTER		VALUE
'S'	=	19
'm'	=	13
'i'	=	9
't'	=	20
'h'	=	8
Total	=	69
Hash value = (69 % 50) = 19		

[FIGURE 8-9a] Computation of the hash value for the key "Smith"

The name "Smith" hashes to location 19, so information about the person named "Smith" is stored in location `h[19]` of hash table `h`. To retrieve the information associated with the key "Smith", we again hash on the key, obtain the integer value 19, and go directly to location `h[19]`. This either contains the desired information or is marked "empty," in which case we know that "Smith" is not in the table.

If hashing always worked as described, inserting a (k,v) tuple into hash table h, the map method called put would be an O(1) operation:

```
put(k,v):     i  = f(k);      // Hash on the key field
                              // to get an array index
              h[i]  = v;      // Store the value field in
                              // that position of the array
              return;
```

In a retrieval from a hash table, the map method get would also be O(1):

```
v = get(k):   i  = h(k);           // Hash on the key field to
                                   // get an array index
              if h[i] is empty
                   return null;    // The (k,v) tuple is
                                   // not in the table
              else
                   return h[i];    // The (k,v) tuple is
                                   // in table. Return v.
```

Comparing the performance of these two algorithms with the values in Figures 8-7a and 8-7b shows that a hash-based implementation of a map is far superior to either the array or linked list implementation in Section 8.2.2. Unfortunately, the analysis is not that simple, and hashing does not always work as well as described.

A **perfect hashing function** f is one with the property that if x and y are two distinct keys, then $f(x) \neq f(y)$. However, perfect hash functions are rare, and even very good hash functions produce the condition that for some $x \neq y$, $f(x) = f(y)$. When two distinct keys hash to the same location, it is called a **collision**. For example, using the hash function described earlier, the name "Posen" would hash to the same location as the key "Smith", as shown in Figure 8-9b.

LETTER		VALUE
'P'	=	16
'o'	=	15
's'	=	19
'e'	=	5
'n'	=	14
Total	=	69
Hash value = (69 % 50) = 19, the same value as "Smith"		

[FIGURE 8-9b] Computation of the hash value for the key "Posen"

When we go to location 19 to store the information about Ms. Posen, we find the slot already filled with information about Mr. Smith. The algorithms described previously no longer work.

Collisions are unavoidable because in most cases the number of keys is very large, much larger than the hash table size. For example, if keys are people's names, and we assume that last names can contain up to 12 letters, then the number of possible keys is 26^{12}—a tremendously large value (1 followed by 17 zeroes). It would be impossible to create a table large enough to guarantee no collisions. Besides, even if you could allocate that much space, it would be very wasteful because the overwhelming majority of slots would be unused.

So, how do we address the collision problem and turn hashing into a reasonably efficient technique? We must do two things. First, even though collisions are unavoidable, we must minimize their occurrence, and second, we must develop algorithms to deal with the inevitable collisions that do occur.

To minimize the number of collisions, we must select the best possible hashing function, in which *best possible* is defined as a function that scatters the keys most uniformly over the n slots $[0 \ldots n - 1]$ of the hash table. That is, if we randomly select a large number, M, of keys, then an approximately equal number, M/n, should hash to each of the n slots in the hash table. A uniform scattering balances the number of keys that hash to a given location i, thus reducing the total number of collisions. (This is why hashing functions are also called **scatter functions**.) An example of a very poor hashing function would be the following:

$f(\text{key}) = [2 * \textbf{int}(\text{key})] \% n$ (where the table size n is an even number)

This function would map a set of keys only to even-numbered locations 0, 2, 4, 6, … in the table, completely ignoring the odd-numbered slots. This would effectively double the total number of collisions and seriously impair the performance of our algorithms.

TESTING A HYPOTHESIS

Computer science majors are usually required to take a number of mathematics courses, including discrete math, linear algebra, and statistics. Although students frequently use discrete math and linear algebra in classes, they often wonder why statistics is required. The following example answers the question.

We defined a good hashing function as one that scatters keys uniformly across all rows of the hash table. How close to "uniform" must we be to say that a function *f* is behaving well? For example, if we hash 100 keys into a table with five slots, we would ideally expect $100 \div 5 = 20$ keys to hash to each slot. What if we tested a function *f* and obtained the results shown in the Test 1 column?

Entry	Ideal	Test 1	Test 2	Test 3
0	20	22	25	24
1	20	20	18	11
2	20	18	14	30
3	20	19	19	14
4	20	21	24	21

The results look good, with all values close to the ideal of 20. Seeing this, you would probably conclude that the hashing function is scattering well. However, what if you got the results in the Test 2 column? You may not be sure what to do because the totals are further from the ideal. Is this difference due to chance? Is the hashing function biased? It is not clear what to conclude. If you got the values in the Test 3 column, you would be skeptical and might even consider rejecting the function on the basis of bias. But, how can you support your rejection with better evidence than a simple impression that the results don't look good?

The answers to these questions come from the field of statistics—specifically, a technique called **hypothesis testing**. For example, we might assert that "The hashing function *f* is an excellent function. Any deviation from the ideal is due to chance, not bias." This is called a **null hypothesis**, and statistical tests allow us to determine the probability of getting the observed distribution, assuming the null hypothesis is valid. For example, if your function is truly unbiased, distributions such as Test 2 occur about once every three times, while distributions such as Test 3 occur only one time in 300. This kind of quantitative information allows you to make informed decisions about what to do. In this

continued

example, we would do more testing and probably keep the functions that produced the results in the Test 1 and Test 2 columns, but reject the function that produced the results in the Test 3 column.

So, the next time you struggle with a t-test or agonize over a chi-square, remember that these statistical skills will be very useful during your career in computing.

The study of the mathematical properties of hashing functions is complex and well beyond the scope of this text. Courses in function theory and numerical analysis discuss this subject in greater detail. Here, we simply state that one effective type of hashing function for typical hash table sizes and keys is the **multiplicative congruency method**. This method first casts the key field to an integer, regardless of its original type, and then computes the following value for a hash table with array indices in the range $[0 \ldots n - 1]$.

```
f(key) = [(a * int(key)) % N]  //  Where N, the table size,
                               //  and a are large prime
                               //  numbers
```

The function *f* will, for many choices of `a` and `N`, produce reasonably well-scattered integer values in the range $[0 \ldots n - 1]$. (However, it is still important to test *f* to ensure it works well for your specific choices of `a` and `N`.)

The designers of Java understood the importance of hashing and the difficulty of designing and validating good hashing functions. Therefore, they included the following method in class `Object`:

```
public int hashCode();    // Converts an object into a
                          // random integer value that can
                          // be used to store values in a
                          // hash table
```

Because `Object` is the root class of Java, every object in the language inherits this method. It can be used to implement the hashing techniques described in this chapter. So, given our earlier example of people's names and a 50-element hash table, another way to implement a hashing function is:

```
String name;        // A person's name
...
// Hash on name using the inherited function
// hashCode(). Then reduce the result to the range
// [0 ... 49] using the % (modulus) operator
int i = (name.hashCode()) % 50;
```

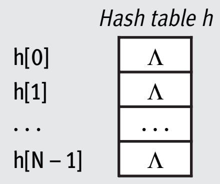
The remainder of this section assumes the existence of a hash function *f* that has been shown to scatter key values well, regardless of whether it is a Java library function or one of your own design.

However, even with a well-designed function *f*, we will still encounter the condition of two or more distinct keys hashing to the same location. To deal with these inevitable collisions, we explore two methods in this section—open addressing and chaining.

8.2.3.1 Open Addressing

With **open addressing**, the `(k,v)` tuples are stored in the *n*-element hash table array itself. Each row of the array holds a single tuple—an object containing the two state variables `key` and `value`.

We first initialize all hash table entries to **null**, representing the empty state or not in use:



Now, assume that we want to implement the `put()` method that inserts a new tuple `(k,v)` into the preceding hash table. First, compute the value $i = f(k)$. If location `h[i]` is not empty, then this slot is already occupied, and we have a collision. The open addressing algorithm searches sequentially through the hash table for an empty location. That is, if row i is occupied, the algorithm searches locations $i + 1, i + 2, i + 3, \ldots$ looking for the first empty slot to store `(k,v)`. This search is done *modulo* the table size n so that, after looking at the last item in the table, `h[N-1]`, it wraps around and continues its search from `h[0]`. The `put()` method terminates when we find an empty location and successfully store this new tuple, or we return to location i. In the latter case, the table is full and the insertion operation fails.

For example, assume a hash table with six slots numbered 0 to 5. We want to store four tuples, with keys k_1, k_2, k_3, and k_4, in just that order. When we apply our hash function f to these four keys, assume the following results:

$$f(k_1) \rightarrow 3$$
$$f(k_2) \rightarrow 5$$
$$f(k_3) \rightarrow 3$$
$$f(k_4) \rightarrow 4$$

The first two key values, k_1 and k_2, hash to empty locations, so their tuples are stored in slots 3 and 5 of the table, respectively. Key k_3 hashes to location 3, which is occupied, so we begin a sequential search to find an empty slot. The immediately succeeding location, slot 4, is available, so (k_3,v_3) is stored there. Finally, key k_4 hashes to location 4, which is also occupied. So is the subsequent location, slot 5. However, the next slot, 0, is empty, so the tuple (k_4,v_4) is placed there. After inserting these four tuples, the contents of the hash table are:

Hash table h

h[0]	(k_4, v_4)
h[1]	Λ
h[2]	Λ
h[3]	(k_1, v_1)
h[4]	(k_3, v_3)
h[5]	(k_2, v_2)

The `get` method works in a similar fashion. To retrieve the value field of the tuple whose key field is `k`, compute $i = f(k)$. If location `h[i]` == **null**, then the tuple is not in the table, and we are finished. If `h[i]` is non-null, examine the key field of the tuple referenced by `h[i]` to see if it equals the desired key `k`. If it does not, then this slot is occupied by a different tuple. Now we search sequentially through the rows of `h` until we either find the desired key `k` and return its associated value field, `v`, or we cycle through the entire table and return to location `i`. If the latter is true, the table is full and the key field `k` is not present. The other possibility is to encounter a **null** entry during our search. This means that the key `k` is not in the table; if it were, it would be stored in this location.

Figure 8-10 is a class that implements the hashing and open addressing algorithms just described. It uses the resources of a class called `Tuple` that defines the structure of an individual tuple in the hash table.

```
/**
 * A hash table that uses open addressing. Javadoc comments
 * for methods specified in the Table interface have been omitted.
 *
 * This code assumes that the preconditions stated in the comments are
 * true when a method is invoked and therefore does not check the
 * preconditions.
 */
public class OpenAddressingHashTable<K,V> implements Table<K,V> {
    // Safety constants for the domain
    public static final int CARDINALITY = 27;
    public static final int ADDR_STEP = 17;
    private Tuple<K,V> table[];  // The hash table
    private int size;            // Number of entries in the table

    /**
     * Create a new hash table.
     */
    public OpenAddressingHashTable() {
      // Note that the cast is necessary because you cannot
      // create generic arrays in Java.  This statement will
      // generate a compiler warning.
      table = (Tuple<K,V>[]) new Tuple[ CARDINALITY ];
      size = 0;
    }

    public void put( K key, V value ) {
      int pos = key.hashCode();

      // Find an open position
      while ( table[ pos ] != null ) {
          pos = ( pos + ADDR_STEP ) % table.length;
      }

      // Store the data in the table
      table[ pos ] = new Tuple<K,V>( key, value );
      size = size + 1;
    }

    public void remove( K key ) {
      int loc = find( key );  // Look for the key
```

continued

```
    // If the key is in the table, remove it
    if ( loc != -1 ) {
        table[ loc ] = null;
        size = size - 1;
    }
  }

  /**
   * Return the location in the table where the tuple with the
   * specified key is found.  A -1 is returned if the key cannot
   * be found in the table.
   *
   * @param key the key to search the table for.
   *
   * @return the location of the key or -1 if the key is not found.
   */
  private int find( K key ) {
    int probe = key.hashCode();
    int loc = -1;

    // Check the next open address until we find the key or have
    // checked the entire array
    for ( int i = 0; i < table.length && loc == -1; i = i + 1) {
        Tuple<K,V> curTuple = table[ probe ];

        // Is this the tuple we are looking for?
        if ( curTuple != null && curTuple.getKey().equals( key ) ) {
          loc = probe;
        }
        else {
          // Calculate the next table location to probe
          probe = ( probe + ADDR_STEP ) % table.length;
        }
    }

    return loc;
  }

  public V get( K key ) {
    int loc = find( key );  // Look for the key in the table
    V retVal = null;

    // If found retrieve the data
    if ( loc != -1 ) {
        retVal = table[ loc ].getValue();
    }
```

continued

```
    return retVal;
  }

  public boolean isEmpty() {
    return size == 0;
  }
  public int size() {
    return size;
  }

  public int cardinality() {
    return CARDINALITY;
  }

  public List<K> getKeys() {
    List<K> keys = new LinkedList<K>();

    // Populate the list with the keys in the table
    for ( int i = 0; i < table.length; i = i + 1 ) {
        if ( table[ i ] != null ) {
          keys.add( table[ i ].getKey() );
        }
    }

    return keys;
  }

  public List<V> getValues() {
    List<V> values = new LinkedList<V>();

    // Populate the list with the values in the table
    for ( int i = 0; i < table.length; i = i + 1 ) {
        if ( table[ i ] != null ) {
            values.add( table[ i ].getValue() );
        }
    }

    return values;
  }

  /**
   * Return a string representation of the hash table.
   *
   * @return a string representation of the hash table.
   */
  public String toString() {
    StringBuffer hashAsString = new StringBuffer( "" );
```

continued

```
        for ( int i = 0; i < table.length; i = i + 1 ) {
            hashAsString.append( "h[" + i + "]==" );

            if ( table[ i ] != null ) {
              hashAsString.append( table[ i ] );
            }

            hashAsString.append( "\n" );
        }

        return hashAsString.toString();
      }

  } // OpenAddressingHashTable
```

[FIGURE 8-10] Implementation of hashing using open addressing

One problem with open addressing is that it frequently produces long chains of occupied cells followed by long chains of empty cells. When a collision occurs, we search sequentially for an available slot. If location i is occupied, the new value is inserted into location $i + 1$, assuming it is empty. Now, if a key hashes to *either* location i or $i + 1$, it is placed in position $i + 2$, assuming that slot is empty. Therefore, the probability of a value being stored in location $i + 2$ is greater than that of being stored in some other cell, and we begin to build chains of occupied slots. These chains degrade performance because progressively longer searches are required to find an empty location. If the empty slots were more evenly distributed throughout the table, searches would be shorter because an empty slot terminates the retrieval operation.

Thus, a small but important modification is needed to open addressing. When a collision occurs, we do not search the table in increments of 1, but in increments of c, where $c > 1$ and is relatively prime with (i.e., shares no common factor with) the hash table size n. This is the case in Figure 8-10, in which the increment size is the constant labeled ADDR_STEP. For example, assume that $n = 10$ and ADDR_STEP = 3 (actually, it is 17 in Figure 8-10). If our key originally hashes to location 5, and that position is occupied, then the slots of the hash table are searched in the order 5, 8, 1, 4, 7, 0, 3, 6, 9, 2, rather than 5, 6, 7, 8, 9, 0, 1, ..., as in a traditional sequential search. This technique does a better job of scattering tuples throughout the hash table and distributing empty slots more evenly, which helps improve the performance of open addressing.

Let's analyze the time required to retrieve a value from a hash table using open addressing. In the `get` method in Figure 8-10, we see that in the best case (no collisions), retrieval requires only a single comparison to locate the desired key, and `get` is an O(1) algorithm. This is the theoretically optimum performance described at the beginning of the chapter. In the worst case, when the table is full, we may have to search the entire n-element hash table to discover whether the key is present. In this case, hashing degenerates to a worst-case performance of $O(n)$ because it is essentially a sequential search of an array.

In the average case, the number of comparisons required for a successful retrieval using open addressing depends not on n, the size of the table, but on how many tuples are stored in the hash table. For example, a sparsely occupied table has few collisions, and we usually go directly to the desired key or to an empty slot, which tells us the key is not present. As the number of elements stored in the table increases, the chances of a collision increase. Therefore, the efficiency of retrieval from a hash table using open addressing depends on a value α, the **load factor**, which is defined as follows:

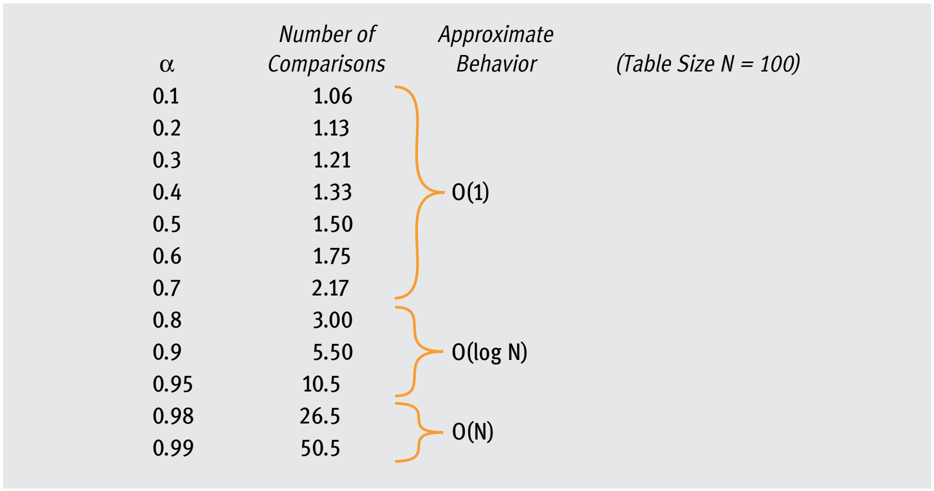
$$\alpha = \text{number of entries in the hash table / hash table size} \qquad (0 \le \alpha \le 1)$$

The computation of the average number of comparisons required for hash table retrieval is quite complex. (The derivation of these formulas can be found in the classic computer science text by Donald Knuth, *The Art of Programming*, Volume 3, referenced in the Challenge Work Exercise at the end of Chapter 7.) The following formula is Knuth's approximation for the average number of comparisons, C, needed for a successful search of a hash table with load factor α using open addressing:

$$C = \frac{1}{2} \times \left[1 + \frac{1}{1-\alpha} \right]$$

For example, if our hash table is half full ($\alpha = 0.5$), the formula says that an average of $1/2 * (1 + 2) = 1.5$ comparisons are required to locate a specific key and retrieve the associated data. Compare this to either a sequential or binary search of a table with 100,000 entries, which require an average of 50,000 ($n/2$) and 17 ($\log_2 n$) searches, respectively. Even if our table is 80 percent full ($\alpha = 0.8$), hashing and open addressing require only about $[1/2 * (1 + 5)] = 3$ searches, still a vast improvement. However, be aware that this performance increase comes at the expense of extra memory. If the hash table size is $n = 100{,}000$, then at 50 percent occupancy, we are leaving 50,000 slots unused. At 80 percent occupancy, 20,000 slots must be left empty. Hashing and open addressing are a reasonable strategy only if there are adequate memory resources and we are willing to leave a portion of our hash table vacant.

We can observe this behavior more clearly by determining exactly how many comparisons are required to retrieve a value from a hash table of size $n = 100$ with load factors ranging from $\alpha = 0.1$ to $\alpha = 0.99$. The results are summarized in Figure 8-11.



α	*Number of Comparisons*	*Approximate Behavior*	*(Table Size N = 100)*
0.1	1.06		
0.2	1.13		
0.3	1.21		
0.4	1.33	O(1)	
0.5	1.50		
0.6	1.75		
0.7	2.17		
0.8	3.00		
0.9	5.50	O(log N)	
0.95	10.5		
0.98	26.5	O(N)	
0.99	50.5		

[FIGURE 8-11] Performance of hashing using open addressing with $n = 100$

Figure 8-11 shows that in a hash table with a load factor α of approximately $0.0 \le \alpha \le 0.8$, the number of searches needed to retrieve a value grows slowly, and hashing displays roughly O(1) performance. In the range $0.8 < \alpha \le 0.95$, the rate of growth in the number of comparisons increases markedly. The performance is now comparable to an $O(\log n)$ binary search technique that needs an average of $\log_2 100 \approx 7$ comparisons to locate a key. Finally, for α in the range $0.95 < \alpha \le 0.99$, performance has degraded significantly, and retrieval requires roughly the same number of comparisons as a sequential search, namely $n/2 = 50$ comparisons. Thus, to get optimal behavior from hashing and open addressing, we must keep the value of the load factor at $\alpha < 0.8$. This means that roughly 20 percent of our hash table should remain empty.

Aside from the extra memory required to obtain reasonable performance, open addressing suffers from another problem—deletions. Assume that we have inserted the key values 10, 15, and 20, in that order, into our hash table, and they all originally hashed to location 1. Also, assume that our increment size is 1, so that we search sequentially for an empty slot if the original slot is full. Here is the result:

The following illustration does not include the value field.

Location	*Key*
0	Λ
1	10
2	15
3	20
4	Λ
. . .	

(where Λ represents “empty”)

If we delete the value 15 stored in table slot 2, and then naively set the slot value to **null**, we end up with the following:

Location	*Key*
0	Λ
1	10
2	Λ
3	20
4	Λ
. . .	

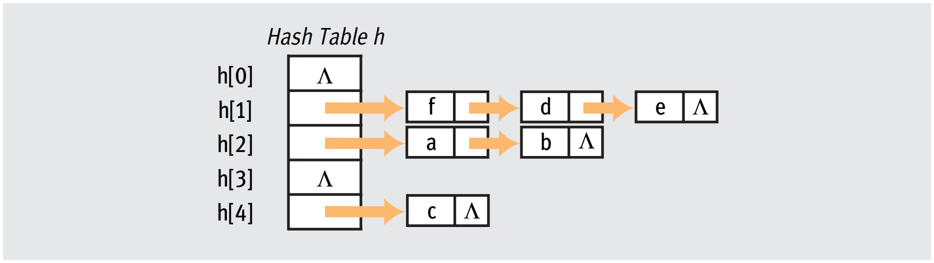
If we then attempt to retrieve the key 20, we encounter a fatal problem. We hash to location 1, see that it is occupied by another key (10), and begin the sequential search of the table from that point. However, slot 2 is empty, and we will incorrectly conclude that 20 is not in the table; if it were, it would have been stored in slot 2. The error occurred because slot 2 was occupied during the original insertion of the key value 20.

With open addressing, deletions are problematic. When we do a deletion, we must indicate that the space was previously occupied, although it is now empty. We can do this by using a different type of empty marker, such as the asterisk symbol. Then, when doing a retrieval, we can terminate the operation when we encounter a “true” empty marker (Λ), but we must continue searching if we encounter the * symbol, which means the space is empty but was previously occupied. If our hash table is highly dynamic and has many deletions, the table soon fills with * markers rather than Λ, and performance suffers as the average search length increases. For this reason, as well as the space required to keep a portion of the table empty, an alternate method of handling collisions is more popular.

8.2.3.2 Chaining

The second approach to collision resolution, called **chaining**, is quite different from the open addressing method. The n elements of the hash table `h` are no longer used to store the actual `(k,v)` tuples themselves, but only a reference to a linked list of all tuples that hash to this location. That is, element `h[i]` is the head of a linked list containing all tuples `(k,v)` such that $i = f(k)$.

For example, assume that our hash table has length 5. If the keys 'a' and 'b' hash to 2, 'c' hashes to 4, and 'd', 'e', and 'f' hash to 1, then the hash table looks like the diagram in Figure 8-12, disregarding the value field.



[FIGURE 8-12] Example of chaining to handle collisions

The use of chaining simplifies the insertion of new tuples into the table. We no longer need to search the entire table to locate where a tuple should be stored. Instead, we add the new `(k,v)` tuple to the head of the linked list referenced by the value `h[i]`, where $i = f(key)$, assuming it is not already there. (If it is there, then we do nothing and return.) To retrieve a value, we again do not have to search the entire hash table. We need to do a sequential search of the linked list pointed at by `h[i]`, where $i = f(key)$.

A class that implements chaining-based hashing algorithms is shown in Figure 8-13. It uses the `hashCode()` method inherited by all Java objects.

```
/**
 * An implementation of a HashTable using chaining for collision
 * resolution. Javadoc comments for methods specified in the
 * table interface have been omitted.
 *
 * This code assumes that the preconditions stated in the
 * comments are true when a method is invoked and therefore
 * does not check the preconditions.
 */
```

continued

```
public class ChainingHashTable<K,V> implements Table<K,V> {
    private List<Tuple<K,V>> hashTable[];  // The hash table
    int size;                              // Size of the table

    /**
     * Create a new hash table.
     *
     * @param cardinality the number of elements in the domain.
     */
    public ChainingHashTable( int cardinality ) {
      // Note that the cast is necessary because you cannot
      // create generic arrays in Java.  This statement will
      // generate a compiler warning.
      hashTable =
        (LinkedList<Tuple<K,V>>[])new LinkedList[ cardinality ];
      size = 0;
    }

    public void put( K key, V value ) {
      int bucket = key.hashCode();

      // Do we need to create a new list for this bucket?
      if ( hashTable[ bucket ] == null ) {
          hashTable[ bucket ] = new LinkedList<Tuple<K,V>>();
      }

      hashTable[ bucket ].add( new Tuple<K,V>( key, value ) );
      size = size + 1;
    }

    public void remove( K key ) {
      int bucket = key.hashCode();
      List<Tuple<K,V>> chain = hashTable[ bucket ];
      boolean found = false;

      // Is there a chain to search?
      if ( chain != null ) {
          // Step through the chain until we fall off the end or
          // find the tuple to delete
          chain.first();
          while ( !found && chain.isOnList() ) {
            // If this tuple has the key we are looking for,
            // delete it and stop the loop
            if ( chain.get().getKey().equals( key ) ) {
                chain.remove();
                found = true;
            }
```

continued

```
                else {
                    chain.next();
                }
            }
        }
    }

    public V get(K key) {
        int bucket = key.hashCode();
        List<Tuple<K,V>> chain = hashTable[ bucket ];
        V retVal = null;

        // Is there a chain to search?
        if ( chain != null ) {
            // Step through the chain until we find the element or
            // run out of list.
            for ( chain.first();
                retVal == null && chain.isOnList();
                chain.next() ) {

              // If this tuple has the key we are looking for,
              // extract the value
              if ( chain.get().getKey().equals( key ) ) {
                  retVal = chain.get().getValue();
              }
            }
        }

        return retVal;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public int size() {
        return size;
    }

    public int cardinality() {
        return hashTable.length;
    }

    public List<K> getKeys() {
        List<Tuple<K,V>> chain;
        List<K> keys = new LinkedList<K>();
```

continued

```
    // Go through each chain and create a list that
    // contains all of the keys in the table
    for ( int i = 0; i < hashTable.length; i++ ) {
        if ( hashTable[ i ] != null ) {
          chain = hashTable[ i ];

          for (chain.first(); chain.isOnList(); chain.next()) {
              keys.add( chain.get().getKey() );
          }
        }
    }

    return keys;
  }

  public List<V> getValues() {
    List<Tuple<K,V>> chain;
    List<V> values = new LinkedList<V>();

    // Go through each chain and create a list that
    // contains all of the keys in the table
    for ( int i = 0; i < hashTable.length; i++ ) {
        if ( hashTable[ i ] != null) {
          chain = hashTable[ i ];

          for (chain.first(); chain.isOnList(); chain.next()) {
              values.add( chain.get().getValue() );
          }
        }
    }

    return values;
  }

  /**
   * Return a string representation of this hash table.
   *
   * @return a string representation of this hash table.
   */
  public String toString() {
    StringBuffer hashAsString = new StringBuffer( "" );
    List chain = null;

    for ( int i = 0; i < hashTable.length; i = i + 1 ) {
        hashAsString.append( "h[" + i + "]==" );
```

continued

```
            if ( hashTable[ i ] != null ) {
              chain = hashTable[ i ];

              for (chain.first(); chain.isOnList(); chain.next()) {
                  hashAsString.append( " " + chain.get() );
              }
            }

            hashAsString.append( "\n" );
        }

        return hashAsString.toString();
      }

  } // ChainingHashTable
```

[FIGURE 8-13] Implementation of hashing using chaining

One nice characteristic of chaining is that you can store more than n keys in the hash table, where n is the table size. We saw this in Figure 8-12, where we stored the six keys, 'a', . . . , 'f' in a hash table of size $n = 5$. With open addressing, we were limited to a maximum of n entries, at which time the table became full, and no more insertions were possible.

On average, each linked list in the hash table will have length $\alpha = M/n$, where M is the total number of tuples stored in the table and n is the hash table size. The time needed to insert a new `(k,v)` tuple into our table using chaining is the sum of the following three steps:

1. Accessing the reference value stored in location `h[i]`, where $i = f(k)$
2. Searching the entire list pointed at by the head pointer to see if the key being inserted is already there (remember, no duplicates are allowed)
3. If it is not there, inserting the new tuple at the head of the list

Steps 1 and 3 take O(1) constant time, while Step 2 takes time proportional to $\alpha = M/n$, the average length of each linked list, assuming the key is not already there. Thus, insertion into a chained hash table takes $(2 + \alpha)$ steps if the key is not in the table. Similarly, retrieval takes one access to the head of the linked list followed by a search of an average of half the linked list before we find the desired key. So, retrieval from a chained hash table requires, on average, $(1 + \alpha/2)$ operations.

Knowing an approximate value for `M`, the number of tuples we will store, we can adjust the hash table size n to provide an acceptable level of performance with a minimum of wasted space. For example, if we plan to store about `M` $= 1000$ tuples in our hash table using chaining, then a table size of $n = 10$ produces linked lists with an average length of 100, requiring about $(1 + 100/2) = 51$ comparisons per retrieval—rather poor. A table size of $n = 50$

produces linked lists of length $1000/50 = 20$ and an average of $(1 + 20/2) = 11$ operations per retrieval, comparable to binary search, which needs $\log_2 1000 \approx 10$. Finally, a table size of $n = 500$ produces an average of two comparisons per retrieval, which is roughly comparable to open addressing with a load factor $\alpha = 0.65$, as shown in Figure 8-11.

Again, note that these performance improvements are achieved at the expense of extra memory. This time the space is needed for the reference values contained in both the hash table and the nodes of the linked lists. For example, assume that 1000 keys are stored in the table, `M` (disregard the value field for now); n, the table size, is 500; and each integer and each reference value occupy four bytes. Let's determine the total amount of memory required to store all the keys.

The chaining technique requires 500 reference values for the hash table itself plus memory space for 1000 linked list nodes, each containing a single key and a single reference. This is a total of $(500 \times 4 + 1000 \times 8) = 10{,}000$ bytes of storage. By way of comparison, open addressing with $\alpha = 0.65$ produces about the same level of performance and requires a table size of $1000/0.65 = 1538$ entries. Each table entry holds a single 4-byte value for a total space requirement of $1538 \times 4 = 6152$ bytes, about 38 percent less space. However, both of these hashing methods require more space than either the sequential or binary search, which is just the 4000 bytes needed to store the key values themselves. This is a good example of what computer scientists like to call the **time-space trade-off**, in which the use of extra memory can potentially reduce the execution time needed to obtain a solution.

To conclude our discussion of hashing, let's analyze the use of a hash table to implement a 60,000-word English dictionary. These dictionaries come as part of modern word-processing packages to provide spell-checking services. A dictionary is an excellent candidate for hashing.

The two typical operations of a dictionary object are:

- Adding new words (so users can customize the dictionary to their needs)
- Checking whether a word is in the dictionary to determine if it is correctly spelled

These operations are performed by the `put()` and `get()`methods in the `Map` interface of Figure 8-5. Also, the lookup operation must be completed very quickly, because it must be carried out once for every word in the document. For large text files, this could encompass hundreds of thousands of words. Users would not be happy if they had to wait too long for the spell-checking software to complete. Thus, the increased speed you can achieve via hashing could be very important to the success of this software package.

If we use the open addressing hashing method to store the dictionary, we know that performance is highly dependent on the load factor α. For $\alpha > 0.9$, performance can degrade significantly. For $\alpha > 0.98$, hashing will begin to approximate linear behavior, which is unacceptable for meeting performance specifications.

Let's assume that we allocate 12 bytes of space for each word in our dictionary—allowing for a maximum of 12 characters per word. Keeping 10 percent of the table empty

($\alpha = 0.9$) means that our 60,000-entry hash table must include about 6600 unused slots ($66{,}600 \times 0.9 = 59{,}940$), which would require $6600 \times 12 = 79$ KB of extra storage beyond what is needed for the 60,000 words themselves. Keeping the table 5 percent empty ($\alpha = 0.95$) requires only an extra 38 KB of memory. Finally, if we can afford to keep the table 20 percent empty, we get much better performance, but at the price of an extra 180 KB to store our hash table.

Another potential problem with open addressing is that deletions must be handled in a special way. As mentioned earlier, if a user removes a word from the dictionary, we must mark that slot with an entry that indicates the slot was once occupied. A large number of deletions could slow down the retrieval operation. However, words are rarely removed from a dictionary, so this should not be a problem.

Instead of open addressing, we could use a chaining scheme with a hash table array of n references and about $60{,}000/n$ words per linked list. Now our word lookup consists of one access to the n-element hash table, followed by a search of about half the ($60{,}000/n$) words that hash to this location. For example, if we want to locate words in an average of six comparisons, then we want the following relationship to hold:

$$1 + \alpha/2 = 6, \text{ or}$$
$$\alpha = 10$$

Because $\alpha = M/n$, and we know that $M = 60{,}000$, we can solve for n:

$$\alpha = M/n$$
$$10 = 60{,}000/n$$
$$n = 6000$$

and we must set n, our hash table size, to approximately 6000.

Now, in addition to the node storage required for the 60,000 words themselves, we need to allocate space for 66,000 references—the 6000 references in the hash table plus the 60,000 references contained in the nodes of the linked lists. If we assume that a reference occupies 4 bytes, then chaining requires an extra $66{,}000 \times 4 = 264$ KB of memory, significantly more than was required by open addressing and $\alpha = 0.9$ or 0.95. If we want better performance, we can increase the size of the hash table, thus decreasing the average length of the linked lists. For example, setting $n = 30{,}000$ reduces the average number of comparisons to 2—one look into the hash table and a search of about half of a linked list of length 2. However, this requires storage for 90,000 reference values (30,000 in the hash table + 60,000 in the list nodes), which requires $90{,}000 \times 4 = 360$ KB of additional memory.

Finally, let's compare hashing with the binary search algorithm presented in Chapter 5 and shown in Figure 5-5. If we keep the 60,000 words of our dictionary in sorted order in an array, we can do a binary search without requiring any extra space for either pointers or empty table locations. However, the average number of comparisons required to locate a word will be about ($\log_2 60{,}000$) = 16, significantly more than either hashing method.

Figure 8-14 summarizes the memory demands and performance characteristics of all the dictionary implementation methods we discussed.

TECHNIQUE	EXTRA MEMORY SPACE REQUIRED	APPROXIMATE NUMBER OF COMPARISONS
Binary search	0	16
Open addressing, $\alpha = 0.95$	38 KB	10.5
Open addressing, $\alpha = 0.9$	79 KB	5.5
Open addressing, $\alpha = 0.8$	180 KB	3
Chaining, $n = 6000$	264 KB	6
Chaining, $n = 30{,}000$	360 KB	2

[FIGURE 8-14] Comparison of dictionary implementation techniques

This example clearly demonstrates the advantages of hashing. By using additional memory, either for empty table slots or reference values, we reduced the average number of comparisons needed to locate a word in the dictionary from 16 in the binary search to between 2 and 10.5, up to an eightfold improvement. Such an improvement may be the difference between a profitable software package and one that is rarely used. An agonizing 15-second delay to spell-check a document using binary search could be reduced to two seconds using hashing with the appropriate parameters. As you can see, selecting good data structures and algorithms is crucial to the success of modern software development.

8.3 Graphs

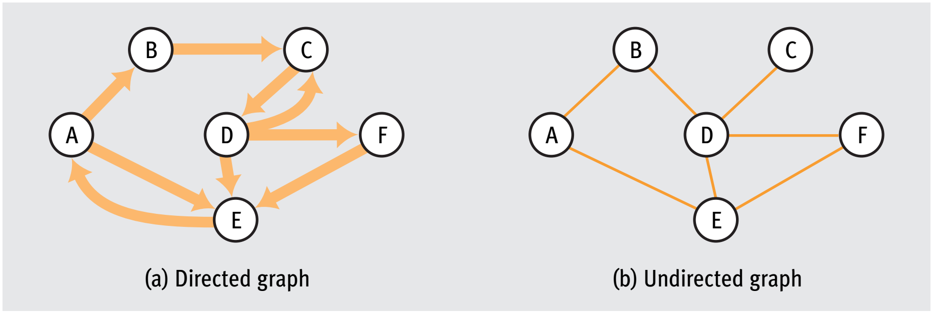
8.3.1 Introduction and Definitions

A graph is the most general and powerful of the three data representations (linear, hierarchical, and graphs) in which the order of elements is a factor. Both the linear structures of Chapter 6 and the hierarchical structures of Chapter 7 are simply graphs in which the number or type of connection between nodes has been restricted. In mathematics, the study of lists, trees, and graphs are lumped together into a single course called *graph theory*. However, in computer science it is common to study these structures separately because of differences in performance and implementation techniques.

Formally, a **graph** is a data structure in which the elements of the collection can be related to an arbitrary number of other elements. That is, a graph is a *(many:many)* data structure in which each element can have an arbitrary number of successors and predecessors.

A graph consists of information units called **nodes** or **vertices**. We will refer to the set of nodes as $N = \{n_1, n_2, \ldots, n_k\}$. For example, in Figures 8-15a and 8-15b, the six nodes are labeled A, B, C, D, E, and F. We may store any type of object in the information field of a node. However, for simplicity, and to focus on the algorithms themselves, in this section we limit the information in a node to either a single character or a single digit.

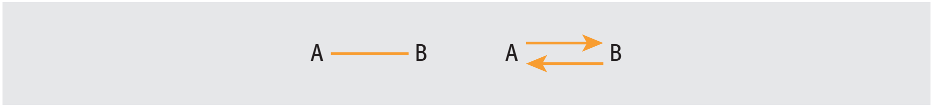
There are two types of graphs, as shown in Figure 8-15.



[FIGURE 8-15] The two fundamental types of graphs

The nodes of a graph are connected by a set of links called **edges**. An edge connecting node n_i to node n_j is usually written as $<n_i, n_j>$, and the set of all edges is written as $E = \{<n_i, n_j>, <n_k, n_l>, \ldots\}$. If edges have a direction, then the graph is termed a **directed graph**, as diagrammed in Figure 8-15a. If $<n_i, n_j>$ is an edge in a directed graph, then n_i is called the **tail** and n_j is called the **head** of the edge. We also say that node n_i is **adjacent to** or **incident to** node n_j.

If no direction is associated with an edge, the structure is called an **undirected graph**, and the presence of an edge implies the existence of a link in both directions. This structure is shown in Figure 8-15b. The following two graphs are equivalent and represent the set of edges $E = \{<A,B>, <B,A>\}$.



However, the following two graphs are not equivalent. The one on the left is an undirected graph that contains an edge connecting A to B and B to A. The one on the right is a directed graph that contains an edge connecting A to B but not B to A.

$$\begin{tikzcd} A \arrow[r, dash] & B & A \arrow[r] & B \end{tikzcd}$$

There can only be a single edge connecting one node to another. That is, the set of edges E cannot have the same entry <A, B> more than once. (If it did, then E would no longer be a set, as duplicates are not allowed.) The following structure, in which three distinct edges connect the same two nodes, is called a **multigraph**:

$$\begin{tikzcd} A \arrow[r, shift left=2] \arrow[r] \arrow[r, shift right=2] & B \end{tikzcd}$$

In graph theory, these types of structures are often not treated. They will not be discussed further in this chapter.

In addition to being directed or undirected, edges can also be **weighted** or **unweighted**. A weighted edge <A, B> has an associated scalar value *W* and is written as <A, B, W>. *W* represents a numerical measure of the cost of using this edge to move from A to B. For example:

<A, B, 8>: $A \xrightarrow{8} B$ Directed, weighted edge

$A \overset{8}{\text{———}} B$ Undirected, weighted edge

The meaning of this notation is that traveling from A to B along this edge involves a cost of eight units. This cost could represent dollars, time, distance, effort, or any other measure of the consumption of resources.

If the edges of the graph are unweighted, the cost of traversing an edge is identical for all edges. In an unweighted graph, the concept of weight is usually not relevant, and we simply set the cost associated with each edge to a constant value, typically 0 or 1.

Figure 8-16 shows the definitions for two classes `Edge` and `Vertex` that define these two basic graph components. The `Edge` class includes state variables for the starting and ending vertices of this edge and its cost. The `Vertex` class includes instance variables for the information stored in the vertex and a table of all edges that start at this vertex.

```
/**
 * An edge in a graph.
 */
public class Edge implements Comparable<Edge> {
    private Vertex source;   // Where the edge begins
    private Vertex dest;     // Where the edge ends
    private int cost;        // The cost to traverse this edge

    /**
     * Create an edge.
     *
     * @param theSource starting vertex.
     * @param theDest ending vertex.
     * @param theCost cost associated with this edge.
     */
    public Edge( Vertex theSource, Vertex theDest, int theCost ) {
      source = theSource;
      dest = theDest;
      cost = theCost;
    }

    /**
     * Create an edge without a cost.
     *
     * @param theSource starting vertex.
     * @param theDest ending vertex.
     */
    public Edge( Vertex theSource, Vertex theDest ) {
      this( theSource, theDest, 0 );
    }

    /**
     * Get the source vertex.
     *
     * @return the source vertex.
     */
    public Vertex getSource() {
      return source;
    }

    /**
     * Get the destination vertex.
     *
     * @return the destination vertex.
     */
    public Vertex getDest() {
      return dest;
    }
```

continued

```
/**
 * Get the cost associated with this edge.
 *
 * @return the cost associated with this edge.
 */
public int getCost() {
  return cost;
}

/**
 * Return a String representation of this edge.
 *
 * @return a String of the form "(startVertex->endVertex:cost)"
 */
public String toString() {
  return
      "(" + source.getName() + "->" +
      dest.getName() + ":" + cost + ")";
}

/**
 * Determine if this edge is equal to another object.
 *
 * @param other the object to compare this edge to.
 *
 * @return true if this edge is equal to the given object.
 */
public boolean equals( Object other ) {
  boolean retVal = false;

  if ( other instanceof Edge ) {
      Edge otherEdge = (Edge)other;

      retVal =
        source.equals( otherEdge.source ) &&
        dest.equals( otherEdge.dest ) &&
        cost == otherEdge.cost;
  }

  return retVal;
}

/**
 * Compare this edge to another edge.  Edges are compared based on
 * their cost.  If the costs are the same the edges will be ordered
 * based on the names of the vertices that they connect.
 *
```

continued

```
     * @param other the edge to compare this edge to.
     */
    public int compareTo( Edge other ) {
      // Compare costs by computing their difference.  If positive
      // the cost of the edge is greater, if 0 the costs are the same,
      // if negative the cost of this edge is less.
      int retVal = cost - other.cost;

      // If the costs are the same, break the tie by looking
      // at the alphabetical ordering of the vertices.
      // In this way, compareTo() will be consistent with equals().
      if ( retVal == 0 ) {
          // Compare the sources
          retVal = source.getName().compareTo(other.source.getName());

          // If the sources are the same, compare the destinations
          if ( retVal == 0 ) {
            retVal = dest.getName().compareTo( other.dest.getName() );
          }
      }

      return retVal;
    }

} // Edge


/**
 * A class representing a named vertex in a graph.
 */
public class Vertex {
    // The cardinality of the table used to track edges
    private static final int TABLE_CARDINALITY = 31;

    private String name;                // Name of the vertex
    private boolean tag;                // Flag
    private Table<String, Edge> edges;  // Edges starting at
                                        //   this vertex

    /**
     * Create a new vertex.
     *
     * @param theIdent a unique identifier for this vertex.
     */
```

continued

```
public Vertex( String theName ) {
  name = theName;
  tag = false;
  edges = new ChainingHashTable<String,Edge>( TABLE_CARDINALITY );
}

/**
 * Get the name of this vertex.
 *
 * Preconditions:
 *     None
 *
 * Postconditions:
 *     The vertex is unchanged.
 *
 * @return the name of this vertex.
 */
public String getName() {
  return name;
}

/**
 * Get the tag associated with this vertex.
 *
 * Preconditions:
 *     None
 *
 * Postconditions:
 *     The vertex is unchanged.
 *
 * @return the state of the tag associated with this vertex.
 */
public boolean getTag() {
  return tag;
}

/**
 * Set the tag associated with this vertex.
 *
 * Preconditions:
 *     None
 *
 * Postconditions:
 *     The flag has the given value.
 *
 * @param newTag the tag's new value.
 */
```

continued

```
public void setTag( boolean newTag ) {
  tag = newTag;
}

/**
 * Add an edge to this vertex. If the edge already exists,
 * it will be replaced by the new edge.
 *
 * Preconditions:
 *     The edge is not null.
 *     The source of the edge is this vertex.
 *
 * Postconditions:
 *     This vertex has the specified edge.
 *
 * @param theEdge the edge to add to this vertex.
 */
public void addEdge( Edge theEdge ) {
  edges.put( theEdge.getDest().getName(), theEdge );
}

/**
 * Get the edge that starts at this vertex and ends at the
 * vertex with the given name.
 *
 * Preconditions:
 *   The name is not null.
 *
 * Postconditions:
 *   The vertex is not changed.
 *
 * @param name the name of the destination vertex.
 *
 * @return a reference to the edge that leads from this vertex
 *         to the specified vertex or null if the edge does
 *         not exist.
 */
public Edge getEdge( String name ) {
  return edges.get( name );
}

/**
 * Determine if the vertex with the given name is a neighbor of
 * this vertex.
 *
```

continued

```
     * Preconditions:
     *   The name is not null.
     *
     * Postconditions:
     *   The vertex is not changed.
     *
     * @param name the name of the vertex.
     *
     * @return true if one of the neighbors of this vertex has
     *         the specified name and false otherwise.
     */
    public boolean isNeighbor( String name ) {
      return edges.get( name ) != null;
    }

    /**
     * Get the neighbors of this vertex.
     *
     * Preconditions:
     *   None.
     *
     * Postconditions:
     *   The vertex is unchanged.
     *
     * @return a list containing all of the vertices that
     *         are neighbors of this vertex.
     */
    public List<Vertex> getNeighbors() {
      List<Edge> theEdges = edges.getValues();
      List<Vertex> retVal = new LinkedList<Vertex>();

      for ( theEdges.first(); theEdges.isOnList(); theEdges.next() ) {
          retVal.add( theEdges.get().getDest() );
      }

      return retVal;
    }

    /**
     * Get the edges that start from this vertex.
     *
     * Preconditions:
     *   None.
     *
     * Postconditions:
     *   The vertex is unchanged.
     *
```

continued

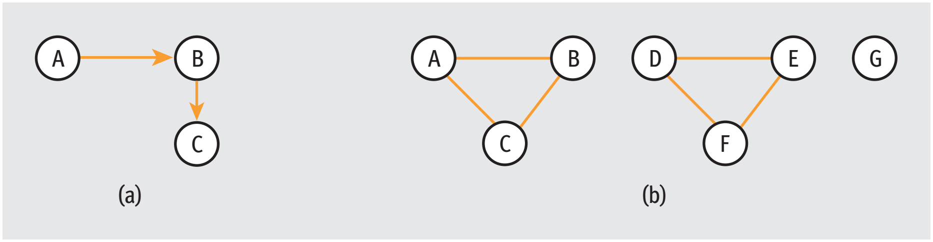
```
     * @return a list containing all of the edges that start from
     *         this vertex.
     */
    public List<Edge> getEdges() {
      List<Edge> theEdges = edges.getValues();
      List<Edge> retVal = new LinkedList<Edge>();

      for ( theEdges.first(); theEdges.isOnList(); theEdges.next() ) {
          retVal.add( theEdges.get() );
      }

      return retVal;
    }

    /**
     * Return a String representation of this vertex.
     *
     * @return a String representation of this vertex.
     */
    public String toString() {
      StringBuffer vertexAsString = new StringBuffer( "" );
      List<Edge> theEdges = edges.getValues();

      vertexAsString.append( name + " (tag=" + tag +")\n" );

      for ( theEdges.first(); theEdges.isOnList(); theEdges.next() ) {
          vertexAsString.append( "   " + theEdges.get() + "\n" );
      }

      return vertexAsString.toString();
    }

} // Vertex
```

[FIGURE 8-16] The `Edge` and `Vertex` classes of a graph

A **path** is a finite sequence of nodes $n_1 n_2 n_3 \ldots n_k$, such that each pair of nodes n_i, n_{i+1}, $i = 1, \ldots k - 1$ is connected by an edge $< n_i, n_{i+1} >$. For example, in Figure 8-15a the sequence ABCD is a path, but ABE is not because no edge connects nodes B and E. The sequence BCDCD is a path that contains repeated nodes. Usually we are only interested in paths in which all nodes are distinct. As we learned in Chapter 7, this is called a **simple path**. When we use the word *path* by itself, we usually mean a simple path.

A very important type of path is the **cycle**. This is a simple path $n_1 n_2 n_3 \ldots n_k$, exactly as defined earlier, but with the added requirement that $n_1 = n_k$. Informally, a cycle is a path that begins at a node, visits any number of other nodes in the graph exactly once (except the first one), and then ends up back at the originating node. In Figure 8-15a, AEA is a cycle, as is EABCDE. The cycle ABCDFEA is called a **Hamiltonian cycle**, which means that it visits *every* node in the graph exactly once.

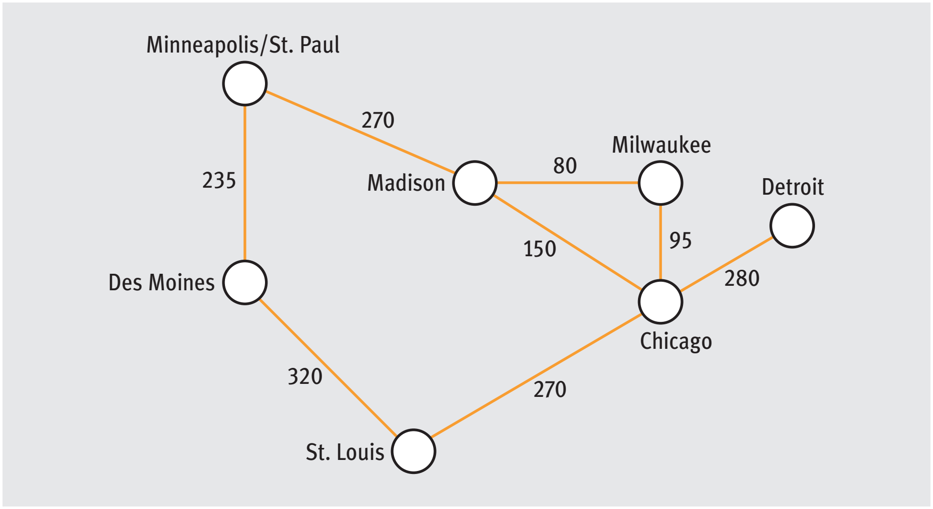
A graph is said to be **connected** if, for every pair of nodes n_i and n_j, there is a path from n_i to n_j. Both graphs in Figure 8-15 are connected, but neither structure in Figure 8-17 is connected.



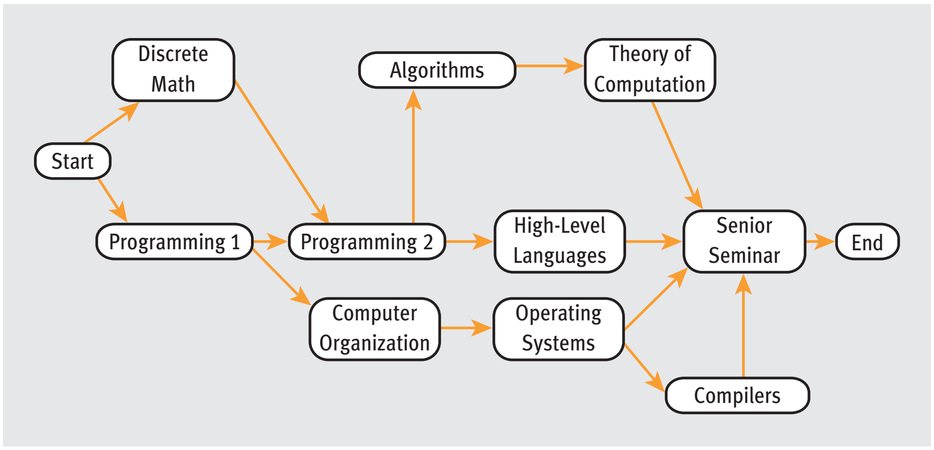
[FIGURE 8-17] Examples of unconnected graphs

The directed graph in Figure 8-17a does not contain paths from B to A, from C to B, or from C to A. The seven-node undirected graph of Figure 8-17b contains three sets of nodes, {A,B,C}, {D,E,F}, and {G}, that are connected among themselves, but are disconnected from each other. These sets are called **connected components**, or **connected subgraphs**.

Graphs occur often in real life. For example, a road map of interstate highway connections between various cities is an excellent example of an undirected graph, because all interstate highways are two-way. We could also add weights to each edge to indicate the distance in miles or driving time in hours between cities, producing a weighted undirected graph:

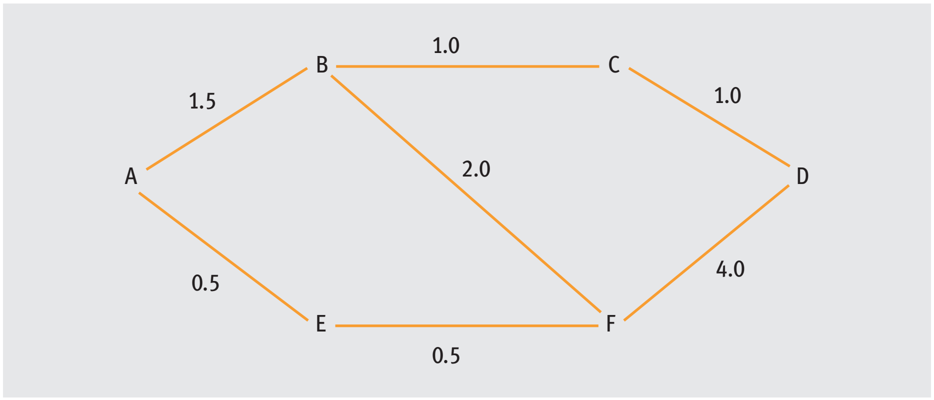


The sequence of courses required to complete a computer science degree can also be represented as a graph. This time it is a directed graph, with the direction of the edge implying the specific order in which courses must be completed.



The preceding graph is interesting in that it contains no cycles. (That is, unless the student has to repeat a course!) This type of structure is called a **directed acyclic graph**, usually abbreviated DAG.

An important example of the use of graphs in computer science is routing in a network such as the Internet. In the Internet, computers are interconnected via high-speed, point-to-point communication links such as phone lines or fiber-optic cables. We use a graph-based representation of the network to find the optimal route from one node to another, and to find backup routes in case of node or line outages. For example, the following diagram shows a six-node network connected via seven communication links. The number next to each link is its line speed in millions of bits/second.



We can use any of four paths to route messages from node A to node D: ABCD, AEFD, ABFD, and AEFBCD. At each node, the network software must create a graph representation of the network and use it to determine the fastest route between itself and any destination. This determination is made every time you send e-mail or view a Web page. (We develop a solution to this *shortest path problem* in Section 8.3.2.4.)

8.3.2 Operations on Graphs

8.3.2.1 Basic Insertion Operations

The two fundamental operations on graphs are adding a new vertex and adding a new edge. An interface that contains these methods, along with utility methods for checking whether a graph contains a particular vertex or edge, is shown in Figure 8-18.

```
/**
 * An interface for a graph. Every vertex in the graph is identified
 * by a unique name.
 */
public interface Graph {
    /**
     * Add a vertex to the graph if the graph does not contain a
     * vertex with the same name.
     *
     * Preconditions:
     *     The name is not null.
     *
     * Postconditions:
     *     The graph contains a vertex with the given name.
     *
     * @param name the name of the vertex.
     */
    public void addVertex( String name );

    /**
     * Add an edge to the graph if the graph does not contain an edge
     * whose source and destination vertices are the same as those
     * specified.
     *
     * Preconditions:
     *     The source and destination vertices are in the graph.
     *
     * Postconditions:
     *     The specified edge is in the graph.
```

continued

```
     *
     * @param source the name of the source vertex.
     * @param dest the name of the destination vertex.
     * @param cost the cost to traverse this edge.
     */
    public void addEdge( String source, String dest, int cost );

    /**
     * Add an edge to the graph. If the graph already contains this
     * edge, the old data will be replaced by the new data.
     *
     * Preconditions:
     *     The source and destination vertices are in the graph.
     *
     * Postconditions:
     *     The specified edge is in the graph.
     *
     * @param source the name of the source vertex.
     * @param dest the name of the destination vertex.
     */
    public void addEdge( String source, String dest );

    /**
     * Determine if the graph contains a vertex with the given name.
     *
     * Preconditions:
     *     The name is not null.
     *
     * Postconditions:
     *     The graph is unchanged.
     *
     * @param name the name of the vertex.
     *
     * @return true if the graph contains a vertex with the given name.
     */
    public boolean hasVertex( String name );

    /**
     * Determine whether there is an edge between the given vertices.
     *
     * Preconditions:
     *     None.
     *
     * Postconditions:
     *     The graph is unchanged.
     *
```

continued

```
     * @param source the name of the source vertex.
     * @param destination the name of the destination vertex.
     *
     * @return true if there is an edge between the vertices.
     */
    public boolean hasEdge( String source, String dest );

    /**
     * Get the vertex with the specified name.
     *
     * Preconditions:
     *   None
     *
     * Postconditions:
     *   The graph is unchanged.
     *
     * @param name the name of the vertex.
     *
     * @return a reference to the vertex with the specified name and
     *         null if no such vertex exists.
     */
    public Vertex getVertex( String name );

    /**
     * Get the edge that connects the specified vertices.
     *
     * Preconditions:
     *   None
     *
     * Postconditions:
     *   The graph is unchanged.
     *
     * @param source the name of the source vertex.
     * @param dest the name of the destination vertex.
     *
     * @return a reference to the edge that connects the specified
     *         vertices and null if the edge does not exist.
     */
    public Edge getEdge( String source, String dest );

    /**
     * Get all of the vertices in the graph.
     *
     * Preconditions:
     *   None.
     *
```

continued

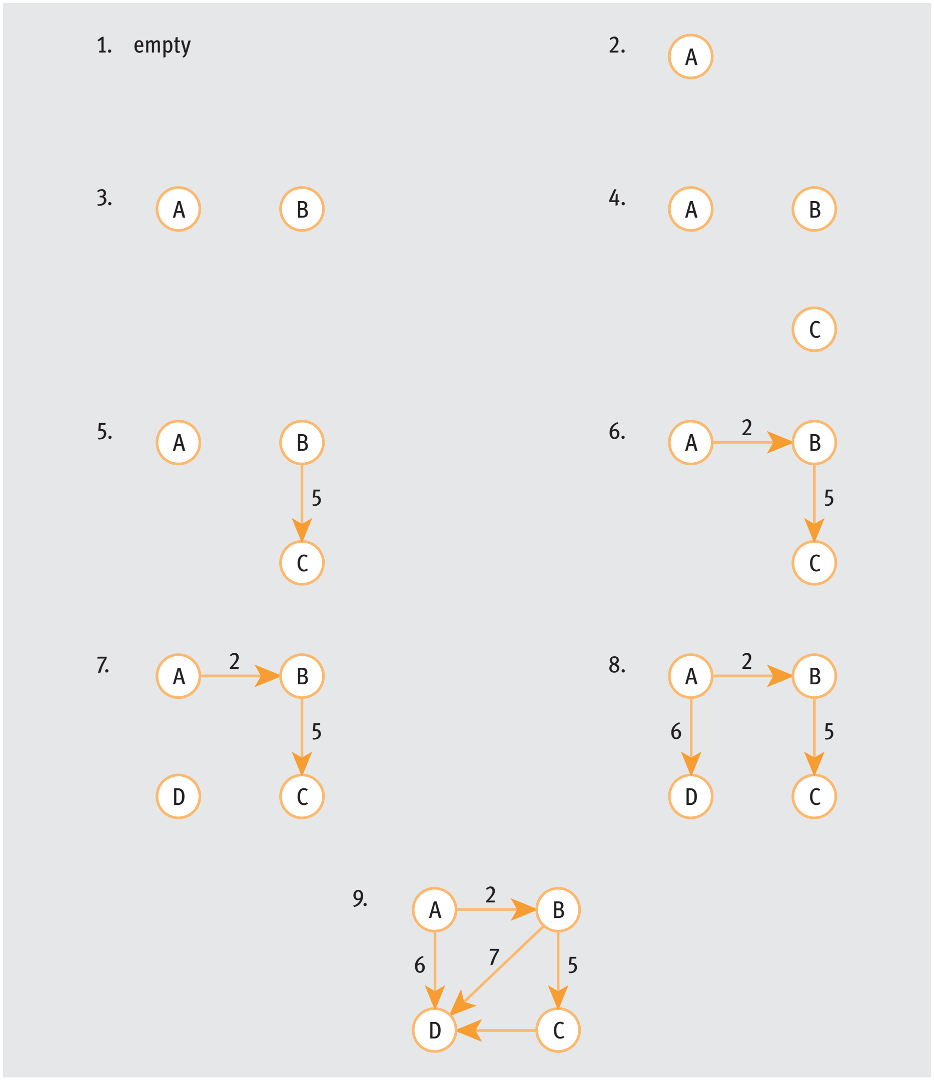
```
     * Postconditions:
     *   The graph is unchanged.
     *
     * @return a list containing all the vertices in the graph.
     */
    public List<Vertex> getVertices();

    /**
     * Get all of the edges in the graph.
     *
     * Preconditions:
     *   None.
     *
     * Postconditions:
     *   The graph is unchanged.
     *
     * @return a list containing all the edges in the graph.
     */
    public List<Edge> getEdges();

} // Graph
```

[FIGURE 8-18] Specifications for a `Graph` interface

Using these "building block" operations, we can construct any arbitrary graph. For example, assuming that G is an empty graph object and A, B, C, and D are the names of our vertices, then the following sequence of eight methods:

```
G.addVertex(A);     // Add a new node A
G.addVertex(B);     // Add a new node B
G.addVertex(C);     // Add a new node C
G.addEdge(B, C, 5); // Connect nodes B and C by an edge of
                    // weight 5
G.addEdge(A, B, 2); // Connect nodes A and B by an edge of
                    // weight 2
G.addVertex(D);     // Now add a new node D
G.addEdge(A, D, 6); // Add link A to it with an edge of
                    // weight 6
G.addEdge(B, D, 7); // Add link B to it with an edge of
                    // weight 7
```

produces the following sequence of graph structures:



Furthermore, the methods in Figure 8-18 can be used to construct both undirected and unweighted graphs. For an undirected graph, whenever you add an edge from A to B, you also add an edge from B to A with the same weight:

```
G.addEdge(A, B, 5);  // To create an undirected graph
G.addEdge(B, A, 5);
```

To create an unweighted graph, just omit the cost parameter:

```
G.addEdge(A, B);     // To create an unweighted edge
```

The cost of the edge from A to B will be set to 0.

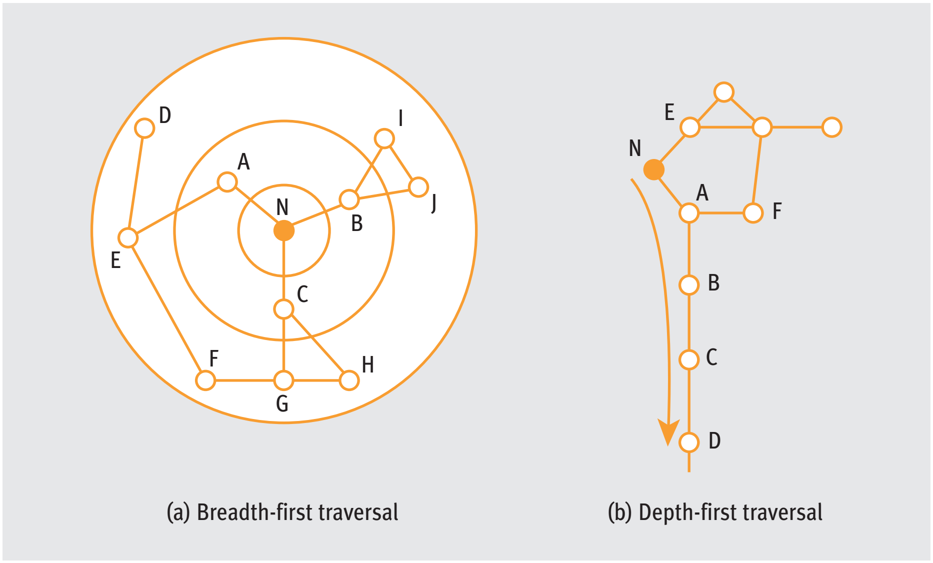
The methods in the interface of Figure 8-18 are the basic building blocks of graphs. However, the interesting operations on graphs are not low-level edge and vertex insertions, but the higher-level operations using these building blocks. We look at some of these operations in the next three sections.

8.3.2.2 Graph Traversal

One of the most important high-level operations on graphs is **traversal**, in which we visit every node in the graph exactly once; to *visit* means to carry out some local processing operation on the node. This operation is identical to the tree traversal algorithms presented in Section 7.3.2. Graph traversal forms the underlying basis of many other graph operations. For example, to print the contents of every node's information field, we traverse the graph, outputting the value in each node as it is visited. To locate a specific key, we again traverse the graph, this time comparing the data stored in the node being visited to our desired key.

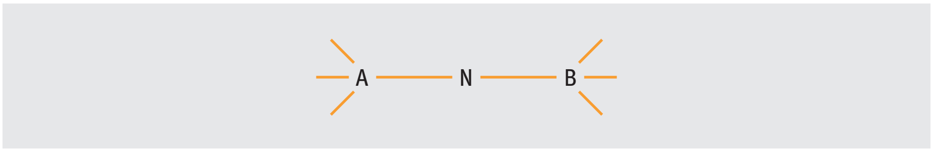
The two basic techniques for traversing a graph are the **breadth-first** and **depth-first** methods. Informally, a breadth-first search means that we visit all the neighbors of a given node before visiting nodes that are not neighbors. Node B is defined as a **neighbor** of node A if there is an edge connecting A to B. A breadth-first traversal can be viewed as a series of expanding concentric circles in which we visit the closer nodes in circle i before moving on to visit more distant nodes in circle $i + 1$. This view is diagrammed in Figure 8-19a. We first visit the three neighbors of the shaded node N—A, B, and C—before moving on to non-neighbor nodes such as D, E, F, G, H, I, and J.

A depth-first search behaves quite differently. It follows a specific path in the graph as far as it leads. Only when the path has been exhausted (by a dead end or cycling back to an earlier node) do we back up and begin to search another path. For example, in Figure 8-19b we would continue to follow the path ABCD… as far as it led, even though there are closer nodes, such as E or F.



[FIGURE 8-19] Types of graph traversal

We must address two issues before we can flesh out the traversal algorithms diagrammed in Figure 8-19 more completely. First, a node N may be the neighbor of two or more nodes, as in the following:



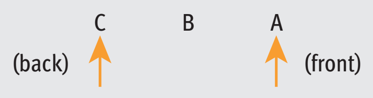
Node N may first have been visited as a neighbor of A. Later, when we visit node B, we do not want to visit its neighbor N again. We solve this problem by using the concept of **marking** a node. Each vertex N will keep a **status value**, also called a **tag**, which can have one of the following three values:

- *Visited*—This node has been visited, and its processing has been completed.
- *Waiting*—A neighbor of this node has been visited, and this node is currently in line waiting to be visited.
- *Not Visited*—Neither this node nor any of its neighbors has been visited yet.

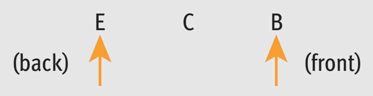
All nodes in the graph start in the not-visited state. When one of its neighbors is visited, a node goes into the waiting state and is placed in a collection of all waiting nodes. When its turn finally comes, it is processed and marked as being in the visited state. By checking the current value of the tag field, we prevent a node from being visited more than once.

Let's examine how to implement the breadth-first search diagrammed in Figure 8-19a. First, we must decide on the proper structure to store the collection of nodes in the waiting state. A little thought should convince you that this collection should be a First In First Out queue structure of the type described in Section 6.4: we want to visit nodes that are already in the collection before we visit the newly added nodes.

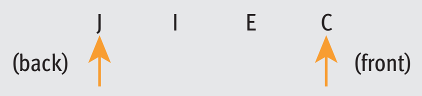
A breadth-first search of the graph in Figure 8-19a begins by marking and enqueueing N, the shaded node. Next, we dequeue N, visit it, mark it as visited, and then queue up N's three neighbors A, B, and C. We also mark them as being in the waiting state. The collection of waiting nodes will now look like this, assuming we add nodes to the queue in alphabetical order:



After finishing with node N and marking it as visited, we remove the node at the front of the queue, A, visit it, and queue up its neighbors (only E, because N has already been visited). The waiting line now holds:



Notice that the new entry E has been placed at the end of the line because we must visit all three neighbors of node N before moving on to their neighbors. This is the characteristic that produces the breadth-first search property. We now remove node B, visit it, and queue up its two neighbors I and J, producing:



Each time we visit a node, we queue only the neighbors whose status is not visited. If their status is visited, they have already been processed. If their status is waiting, the node is already in line, and there is no need to add it to the queue again.

A breadth-first traversal method called `BFSTraverse`, which uses the marking scheme just described, is shown in Figure 8-20. Vertices are marked using the instance variable `tag` contained in every `Vertex` object (see Figure 8-16). The queue of waiting nodes is called `waiting`, and is implemented using the `LinkedQueue` class in Figure 6-39. The visitation of each `Vertex` object is carried out by the `visit()` method of the callback object `cb` passed as a parameter.

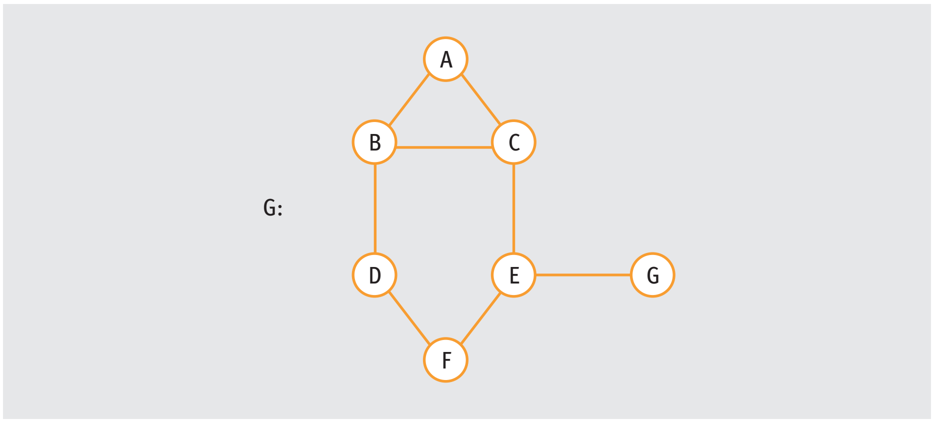
```
/**
 * Perform a breadth-first traversal of the vertices in
 * the given graph starting at the specified node. The
 * callback object is used to process each node as it
 * appears in the traversal.
 *
 * @param g the graph to traverse
 * @param start the vertex where the traversal will start
 * @param cb the object that processes vertices.
 */
public void BFSTraverse( Graph g, Vertex start, Callback<Vertex> cb ) {
    Queue<Vertex> waiting =
        new LinkedQueue<Vertex>();  // Queue of waiting vertices

    // Ensure the graph has no marked vertices
    List<Vertex> vertices = g.getVertices();
    for (vertices.first(); vertices.isOnList(); vertices.next() ) {
        vertices.get().setTag( false );
    }

    // Put the start vertex in the waiting queue
    waiting.enqueue( start );

    // While there are waiting vertices
    while ( !waiting.empty() ) {
        // Get the next vertex
        Vertex curVertex = waiting.front();
        waiting.dequeue();

        // If this vertex hasn't been processed yet
        if ( !curVertex.getTag() ) {
            cb.visit( curVertex );   // Process the vertex
            curVertex.setTag(true);  // Mark it as visited
```

continued

```
                // Put its unmarked neighbors into the work queue
                List<Vertex> neighbors = curVertex.getNeighbors();
                for ( neighbors.first();
                      neighbors.isOnList();
                      neighbors.next() ) {
                    Vertex cur = neighbors.get();
                    if ( !cur.getTag() ) {
                        waiting.enqueue( cur );
                    }
                }
            }
        }
    }
```

[FIGURE 8-20] Breadth-first graph traversal method

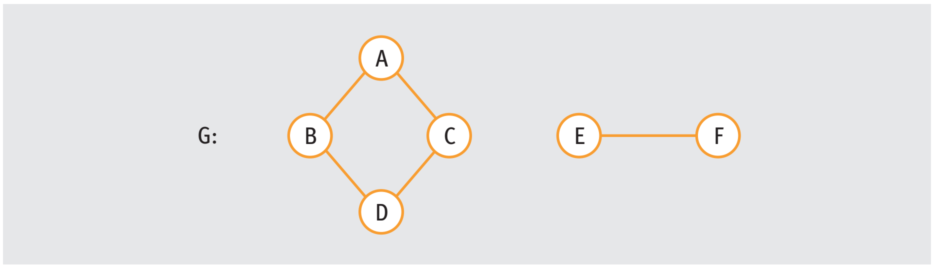
When the algorithm of Figure 8-20 is applied to the following graph G:



one possible traversal of G, beginning from node A, would be ABCDEFG, assuming nodes are queued in alphabetical order. The exact order in which nodes at the same level of the graph are processed depends on the order the vertices are returned by the `getNeighbors()` method.

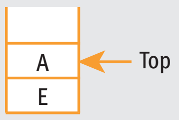
One use of this breadth-first traversal algorithm is to determine whether an undirected graph G is connected—that is, determining if there is a path from node *i* to node *j* for all *i* and *j*. If the breadth-first search of Figure 8-20 is applied to a connected undirected graph G, then the traversal algorithm will ultimately mark every node as visited, and G will have been shown to be connected. However, if one or more vertices remain marked as not visited, then the graph is disconnected, and some nodes are unreachable from our starting node.

We could perform a second traversal operation starting at any of the unmarked nodes to produce a set of **connected subgraphs**. For example, a traversal of the following six-node graph beginning at node A:

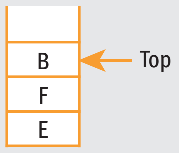


would mark nodes {A, B, C, D} as visited, while leaving E and F marked as not visited. A second breadth-first traversal beginning at either of the unmarked nodes would produce the set {E, F}. These are the two connected subgraphs of the preceding graph G.

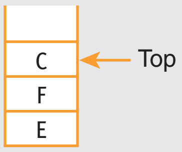
A depth-first graph traversal is similar to the breadth-first search just described, except that the collection of nodes in the waiting state now must be a Last In First Out structure (in other words, a stack). In Figure 8-19b, we can see how a depth-first traversal would work, beginning with the shaded node N. We would first visit node N, and then stack up its two neighbors A and E (again, we assume that nodes are stacked in strict alphabetical order):



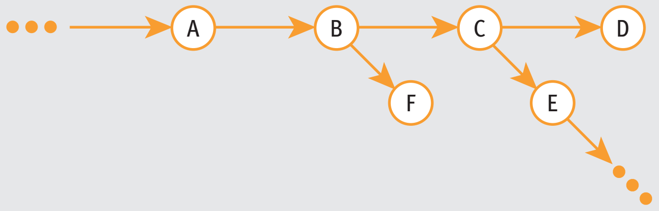
We now pop the stack, returning node A, visit that node, and then stack the two neighbors of that node, B and F. The stack now holds:



Notice that the most recently added nodes are on top of the stack, and will be the ones visited next. We pop node B, visit it, and then stack its neighbor C.



Because we are using a stack instead of a queue, we next visit the nodes that were added most recently, which causes us to continue following the current path wherever it leads. We keep following the path ABCD... in a depth-first fashion until there are no more unmarked nodes on the path. We then pop the stack and start following another path in a depth-first fashion. The new path we follow begins at the last node visited in the current path. Given the following graph:



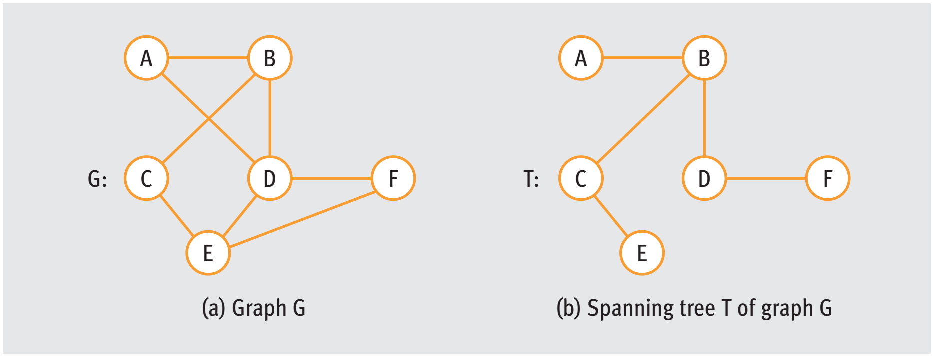
after following path ABCD to its end at D, we back up to the previous node, C, and begin following the path ABCE... as far as it takes us.

We leave the implementation of the depth-first graph traversal algorithm as an exercise for the reader.

8.3.2.3 Minimum Spanning Trees

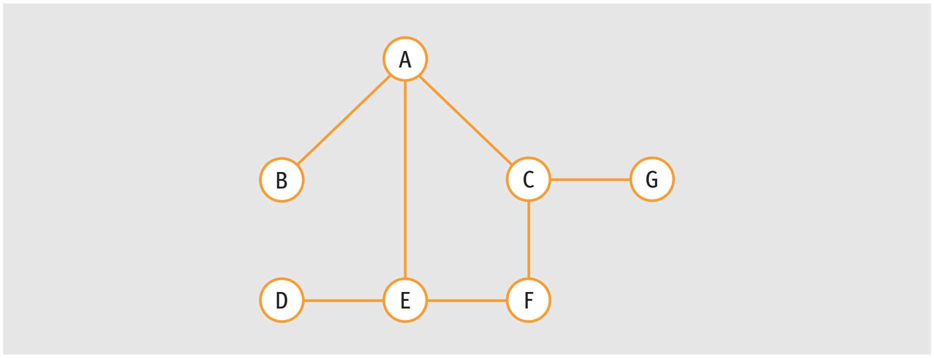
Every connected graph G has a subset of edges that connects all its nodes and forms a tree. This is called a **spanning tree** of the graph G. Spanning trees are important because they contain the minimum number of edges, $n - 1$, that leave an n-node graph in a fully connected state. Think of the process of creating a spanning tree as one of discarding extraneous edges from a graph; the edges are extraneous in the sense that a path between these nodes already exists using other edges.

Figure 8-21b shows a spanning tree T of the connected graph G in Figure 8-21a. Notice that G contains six nodes and eight edges. The spanning tree T contains the same six nodes, but only five edges, and it is still connected.

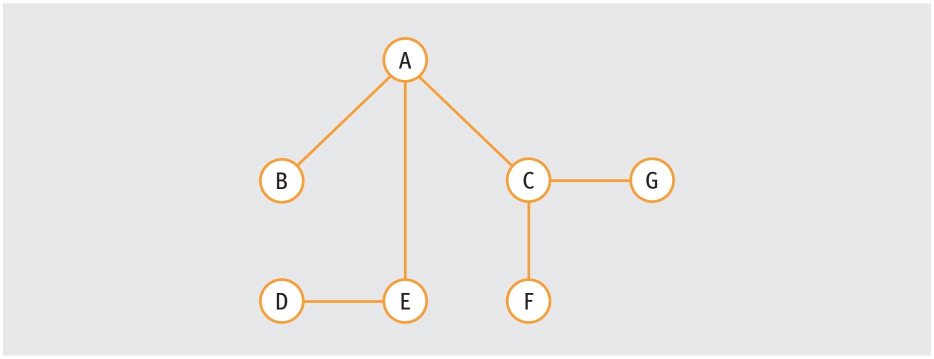


[FIGURE 8-21] Example of a spanning tree of a graph

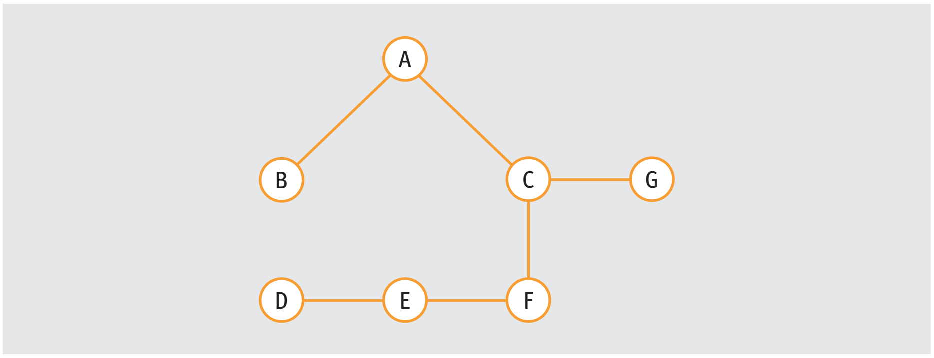
If a graph is unweighted, as in Figure 8-21a, there is a simple way to generate a spanning tree. The graph traversal algorithms of Section 8.3.2.2 visit every node in a graph exactly once. If there is a cycle, and we return to a node via a different edge, we disregard it and do not visit it a second time. Thus, if we simply save the edges used to traverse the graph in either a breadth-first or depth-first traversal, this set of edges forms a spanning tree of G. For example, consider the following graph:



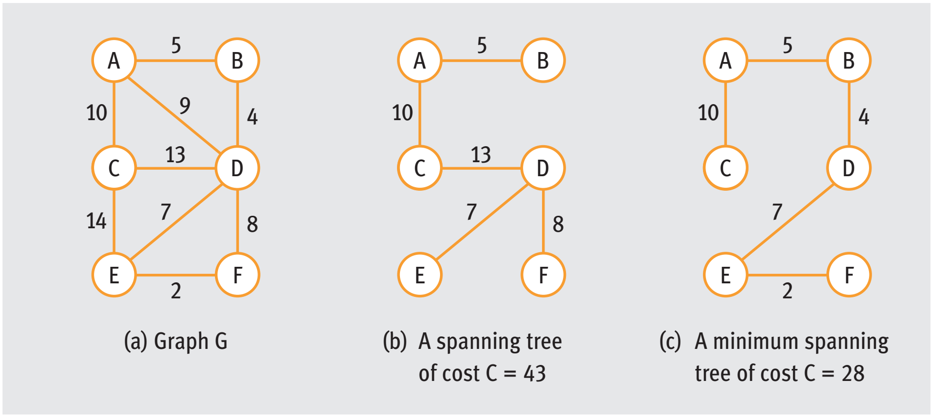
A breadth-first search, beginning at node A and queuing nodes in strict alphabetical order, visits nodes in the order ABCEFGD and uses the six edges <A,B>, <A,C>, <A,E>, <C,F>, <C,G>, and <D,E>. The spanning tree that results, called a **breadth-first spanning tree**, is shown in the following diagram:



A spanning tree generated by a depth-first traversal is called a **depth-first spanning tree**, appropriately enough. For the same seven-node graph just shown, a depth-first traversal of G beginning at A produces the following spanning tree:



The previous diagrams show only two of the many spanning trees you can construct from a single graph. The problem becomes more interesting when the edges of the graph are weighted. In an unweighted graph, we do not care which spanning tree we produce, because they all contain n nodes and $(n - 1)$ edges of equal weight. This is not true for a weighted graph—we want to produce the spanning tree that has the lowest value for the sum of the costs of all edges. This particular spanning tree is called the **minimum spanning tree** (MST). Figure 8-22 shows an example of a weighted graph, an arbitrary spanning tree of the graph, and the minimum spanning tree for the graph.


A
B
C
D
E
F
5
10
9
4
13
14
7
8
2
(a) Graph G
A
B
C
D
E
F
5
10
13
7
8
(b) A spanning tree of cost C = 43
A
B
C
D
E
F
5
10
4
7
2
(c) A minimum spanning tree of cost C = 28

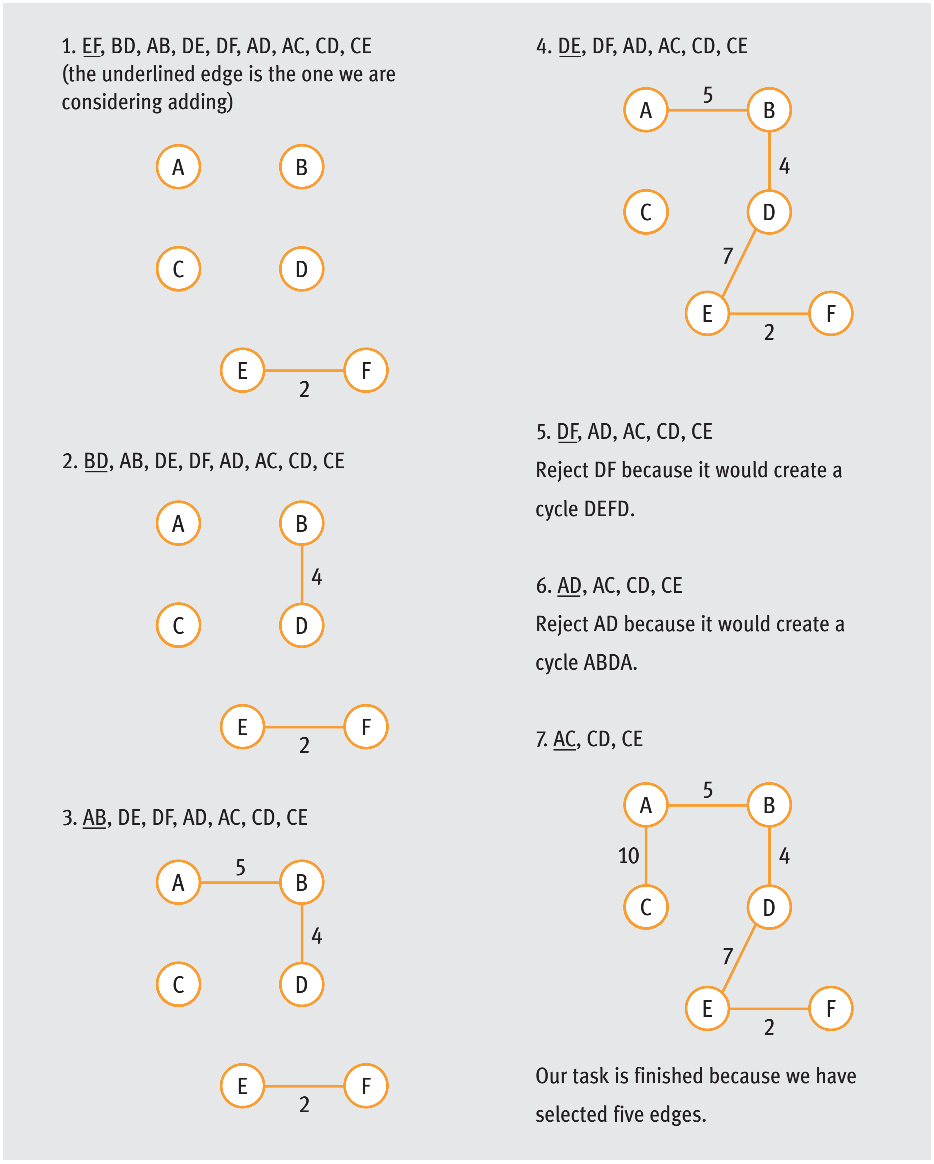
[FIGURE 8-22] Example of a minimum spanning tree of a weighted graph

Minimum spanning trees occur in many real-life applications because they represent the least costly way to connect the nodes in a graph. For example, referring to the road map in Section 8.3.1, how can we interconnect the seven cities using the least mileage? Given *n* host computers, how can we ensure that every host can communicate with every other host using the fewest and cheapest links? The answer to both questions is to construct a minimum spanning tree.

The most well-known technique for building an MST is **Kruskal's algorithm**. It is a **greedy technique**, which always selects the locally optimal choice at each step, regardless of whether it is a globally optimal choice. Essentially, a greedy algorithm always grabs the biggest or best thing available.

In Kruskal's algorithm, we keep adding edges to the spanning tree in order of lowest cost. We first order the set of edges E into increasing order of their cost. From this sorted set E of edges, we select the edge with the lowest cost. We keep this edge in the tree if it does not create a cycle, and we reject it if it does. We repeat the process until we have included $(n - 1)$ edges. If these $n - 1$ edges do not form a cycle, then by definition they must form a tree that connects the root to all n nodes in the graph.

Figure 8-23 outlines the seven steps involved in constructing the minimal spanning tree of Figure 8-22c using Kruskal's algorithm. It begins with the nine edges of the graph ordered by increasing cost.



1. EF, BD, AB, DE, DF, AD, AC, CD, CE (the underlined edge is the one we are considering adding)

2. BD, AB, DE, DF, AD, AC, CD, CE

3. AB, DE, DF, AD, AC, CD, CE

4. DE, DF, AD, AC, CD, CE

5. DF, AD, AC, CD, CE
Reject DF because it would create a cycle DEFD.

6. AD, AC, CD, CE
Reject AD because it would create a cycle ABDA.

7. AC, CD, CE

Our task is finished because we have selected five edges.

[FIGURE 8-23] Using Kruskal's algorithm to produce a minimum spanning tree

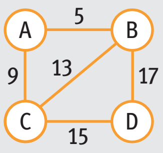
We terminate the algorithm after the seven steps shown in Figure 8-23 because we have added five edges to our six-node graph. If there are no cycles, these edges produce a spanning tree. Because we selected the least-cost edge at each decision point, the tree produced must have minimal cost.

The real trick to implementing Kruskal's algorithm is determining whether adding an edge to the spanning tree creates a cycle. That is, how do we make the decisions shown in Steps 5 and 6 of Figure 8-23, in which we reject an edge because it creates a cycle?

An elegant implementation of this decision uses the set data structure discussed in Section 8.1. We simply maintain sets of connected subgraphs; that is, sets of nodes connected to each other but not to nodes in other sets.

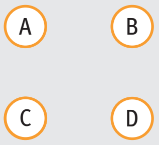
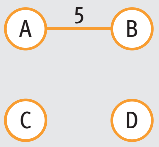
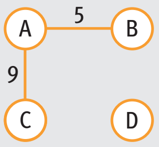
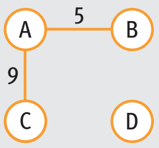
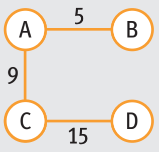
For example, given the two sets A = $\{n_1, n_2, \ldots\}$ and B = $\{n_3, n_4, \ldots\}$, all nodes in set A are connected to each other, and all nodes in set B are connected to each other, but no node in set A is connected to a node in set B, and no node in set B is connected to a node in set A. When the algorithm begins, every node is in its own set because there are no edges in the spanning tree. If we maintain these sets of connected subgraphs as we insert edges, it is easy to determine whether adding an edge to the tree creates a cycle. If an edge connects a node in set A to a node in set B (or vice versa), it cannot create a cycle, because these nodes were previously unconnected. Because all nodes in sets A and B are now connected, we reset A to (A ∪ B), discard set B, and repeat the process until we have a single set that contains all n nodes in the graph. If, on the other hand, our selected edge connects two nodes located in the same set, we know that this produces a cycle. There must already be a path that connects these nodes because they are in the same set.

Figure 8-24 shows how this cycle determination technique works when building an MST using Kruskal's algorithm.



Original graph

Ordered set of edges	
<A, B>	5
<A, C>	9
<B, C>	13
<C, D>	15
<B, D>	17

	MST	Sets	Edge being considered for addition	
1.		{A} {B} {C} {D}	<A, B>	Include, since it connects two nodes in distinct sets
2.		{A, B} {C} {D}	<A, C>	Include, since it connects two nodes in distinct sets
3.		{A, B, C} {D}	<B, C>	Reject, since it connects nodes in the same set and would create a cycle
4.		{A, B, C} {D}	<C, D>	Include, since it connects two nodes in distinct sets
		{A, B, C, D}	Finished	

[FIGURE 8-24] Using sets to detect cycles in Kruskal's algorithm

A program to implement Kruskal's algorithm, including the use of sets to detect cycles, is shown in Figure 8-25. It uses a number of the data structures discussed in this chapter and previous chapters. For example, it uses a `Set` called `curSet` to hold the names of vertices contained within a single connected subgraph. It then creates a linked list named `vertexSets` to link all of the connected subgraphs. Finally, it uses a `Heap` structure called `edges` to hold our collection of `Edge` objects so we can efficiently access (via `getSmallest()`) the edge with the lowest cost. This is a good example of the importance of knowing a wide range of data structures to create elegant and efficient solutions.

```
/**
 * Use Kruskal's algorithm to create a minimum spanning tree
 * for the given graph. The MST is returned in the form of a graph.
 *
 * @param g the graph from which to generate the MST
 *
 * @return a graph containing the vertices of g and the
 *         edges necessary to form a minimum spanning tree.
 */
public Graph kruskalMST( Graph g ) {
    //Where the MST will be stored
    Graph mst = new LinkedGraph();

    // All the vertices in the graph
    List<Vertex> vertices = g.getVertices();

    // List of vertex sets
    List<Set<String>> vertexSets = new LinkedList();

    // Add the vertices in the original graph to the MST
    // and create the vertex sets at the same time
    for ( vertices.first(); vertices.isOnList(); vertices.next() ) {
        String curName = vertices.get().getName();
        Set<String> curSet = null; //new ArrayBasedSet<Vertex>();

        // Add the name of the current vertex to its set, and then
        // add the set to the list that contains the vertex sets
        curSet.add( curName );
        vertexSets.add( curSet );

        // Add the current vertex to the MST graph
        mst.addVertex( curName );
    }

    // Put the edges into a heap, which effectively sorts them
    Heap<Edge> edges = new ArrayBasedHeap<Edge>();
    List<Edge> allEdges = g.getEdges();
```

continued

```
    for ( allEdges.first(); allEdges.isOnList(); allEdges.next() ) {
        edges.insertHeapNode( allEdges.get() );
    }

    // Setup is complete—run the algorithm

    // There is more than one set left in the list vertex sets
    while ( vertexSets.size() > 1 ) {
        // Get the smallest edge
        Edge cur = edges.getSmallest();

        // Find the sets where these vertices are located
        int sourcePos = findSet(vertexSets, cur.getSource().getName());
        int destPos = findSet( vertexSets, cur.getDest().getName() );

        // If the sets are different - add the edge to the MST
        if ( sourcePos != destPos ) {
            Set<String> sourceSet = vertexSets.get( sourcePos );
            Set<String> destSet = vertexSets.get( destPos );

            // Add the edge to the MST
            mst.addEdge( cur.getSource().getName(),
                         cur.getDest().getName(),
                         cur.getCost() );

            // Merge the sets
            sourceSet.union( destSet );
            vertexSets.remove( destPos );
        }
    }

    // The MST is in this graph - return it
    return mst;
}

/**
 * Return the position of the first set in the list that
 * contains the specified name.
 *
 * @param vertexSets a list of sets to search.
 * @param name the name being searched for.
 *
 * @return the position of the first set in the list that contains
 *         the name or -1 if the name cannot be found.
 */
private int findSet( List<Set<String>> vertexSets, String name ) {
    int retVal = -1;
```

continued

```
        // Step through the list and examine each set.  Stop when you
        // find a set with the name or we fall off the list
        for (int i = 0; retVal == -1 && i < vertexSets.size(); i = i + 1) {
            Set<String> curSet = vertexSets.get( i );

            // Does the current set contain the name we are looking for?
            if ( curSet.contains( name ) ) {
                retVal = i;
            }
        }

        // Return the position of the set
        return retVal;
    }
```

[FIGURE 8-25] Kruskal's algorithm for finding minimum spanning trees

8.3.2.4 Shortest Path

The final high-level graph operation we will investigate is the **shortest path** problem, which we mentioned earlier with regard to network routing.

In this problem, we are given a weighted graph G and any two nodes in G: n_i, the **source**, and n_j, the **destination**. We want to identify the path from n_i to n_j with the minimal value for the sum of the weights of all edges in the path. That is, we want to determine the path P = $\{n_i, n_i + 1, n_i + 2, \ldots n_j - 1, n_j\}$ that has the following property:

$$\min \left[\sum_{k=i}^{j-1} \texttt{getCost}\ (< n_k, n_{k+1} >)\right] \quad \texttt{getCost(e)} \text{ returns the cost of edge e}$$

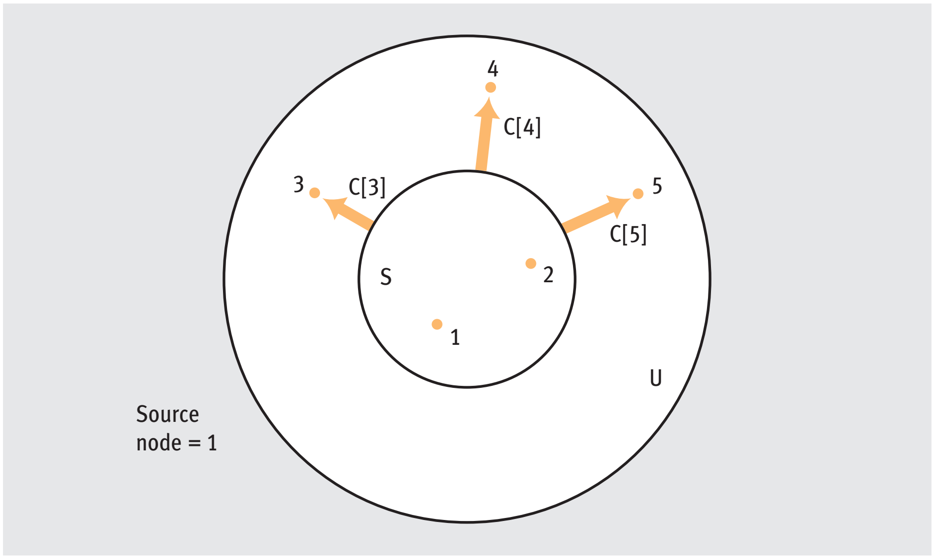
If the graph is not connected, and there is no path from n_i to n_j, the algorithm should return the special value ∞.

Note that we are explicitly concerned about finding the *shortest* path, not simply any path. To determine the latter, we could build a spanning tree rooted at the source node. If the source and destination nodes were connected, then the destination would be a node in the spanning tree, and there would be a unique path from the root to that node. However, we cannot guarantee that this path will have minimal weight. In Figure 8-22a, the shortest path from C to F is CEF, with a cost of 16 units (14 + 2). However, the minimal spanning tree of Figure 8-22c produces the path CABDEF, with a total cost of 28 units, almost double the least-cost path.

Determining shortest paths is an important, real-world problem when managing large-scale computer networks such as the Internet. When an e-mail message or a Web page is transmitted from one machine to another, the network must determine the path. Ideally, it should be as close as possible to the shortest path. Thus, Internet routing is nothing more than the problem of determining shortest paths between two computers in a connected network; it uses techniques similar to those we will describe. (For a discussion of differences between the shortest-path routing algorithm used for the Internet and the algorithm described in the upcoming pages, see the feature at the end of this section, “Theory vs. Practice”.)

The algorithm we will use to determine the shortest path was developed by Professor Edsger Dijkstra, and is appropriately called **Dijkstra’s algorithm**. (We read about Dijkstra and his many contributions to computer science in Section 6.3.2.) Instead of determining the shortest path from a single source node n_i to a single destination node n_j, Dijkstra’s algorithm determines the cost of the shortest path from a source node n_i to *all* other nodes in the graph.

The algorithm operates by dividing the n nodes of the graph G into two disjoint sets—a set S that contains nodes for which we have determined the shortest path, and a set U, which contains nodes for which we have not yet determined the shortest path. These two sets are initialized to the values $S = \{n_i\}$, the source node (because the shortest path to yourself always has cost 0), and U = {all nodes in G except the source node n_i}. We also maintain a data structure called the **cost list**, written as C[j], which represents the cost of reaching node j, $j \in U$, only going through nodes currently in S. Informally, C[j] represents the cost of reaching out from the current set of shortest-path nodes and visiting a node for which we have not yet determined the shortest path. Figure 8-26 shows the relationship between S, U, and C[j]. At the point in the algorithm diagrammed in Figure 8-26, we have determined the shortest path from source node 1 to nodes 1 and 2, but have not yet determined the shortest path from source node 1 to nodes 3, 4, or 5. The cost to reach nodes 3, 4, or 5 from source node 1, traversing only nodes 1 and/or 2, is C[3], C[4], and C[5], respectively.



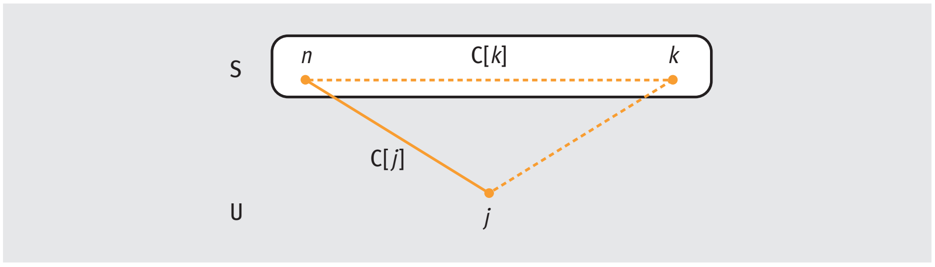
[FIGURE 8-26] Basic concepts in Dijkstra's algorithm

The basic idea behind Dijkstra's algorithm is to use a greedy technique similar to Kruskal's algorithm. At each step we select a single node n_i from the set U, determine the shortest path to it, and place it in S. We repeat this operation until all nodes are in set S, and set U is empty. The algorithm then terminates.

At each step, the new node we select to include in S is the one in U that costs the least to reach from nodes currently in set S. This is the node *k*, $k \in$ U with the minimum value of C[*k*].

Step 1: Determine the minimal value C[*k*] for $k \in$ U.
Remove node *k* from the set U.
Place node *k* in set S.

After finishing Step 1, we update the cost list C, because we have added a new node *k* to S.

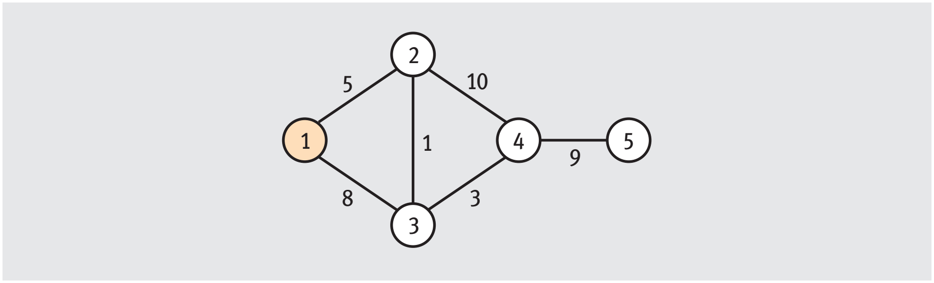


Previously, the cost of reaching node j, $j \in U$, from only nodes in S was $C[j]$—the solid line in the preceding diagram. Now that we have added node k to S, it may be cheaper to go from node n to our new node k and then directly out to j—the dotted lines in the preceding diagram. The cost of this new path will be $[C[k] + \texttt{EdgeCost}(k, j)]$, where $\texttt{EdgeCost}(k, j)$ is the cost of the edge that connects nodes k and j, or infinity if no such edge exists. We are essentially determining whether the addition of node k to set S has created a new shortest path to other nodes outside of S. Thus, we can express the next step in the algorithm as:

Step 2: $C[j] = \min(C[j], C[k] + \texttt{EdgeCost}(k,j))$ for all $j \in U$

This two-step process iterates until all nodes are in S, and U is empty.

Figure 8-27 shows the application of this updating process to a five-node graph G, using node 1 as the source node.



Initial values: $S = \{1\}$ $U = \{2, 3, 4, 5\}$
$C_2 = 5$ $C_3 = 8$ $C_4 = \infty$ $C_5 = \infty$

(Note: C_4 and C_5 are set to ∞ because there is no direct link to these two nodes from node 1.)

Iteration 1: Step 1. Select the minimal value of C.
$C_2 = 5$ (this is the smallest value of all the C_i)
Move the node (i.e., node 2) into S.
$S = \{1, 2\}$ $U = \{3, 4, 5\}$

Step 2. Recompute the C values for all nodes still in set U:
$C_3 = \min(C_3, C_2 + \texttt{EdgeCost}(2,3))$
$= \min(8, 5 + 1)$
$= 6$ (This is reduced from 8 because we can now reach node 3 via the route $1 \rightarrow 2 \rightarrow 3$.)

continued

$C_4 = \min(\infty, 5 + 10)$
$= 15$ (Notice this has been reduced from ∞)

$C_5 = \min(\infty, 5 + \infty)$
$= \infty$ (It is still infinite because there is no direct link to node 5 from either nodes 1 or 2.)

Iteration 2: Step 1. Select the minimal value of C.
$C_3 = 6$ (the smallest value of the remaining C_i)
Move the node into S.
$S = \{1, 2, 3\}$ $U = \{4, 5\}$

Step 2. Recompute the C values for all nodes still in set U:
$C_4 = \min(15, 6 + 3)$
$= 9$ (We have lowered it a second time because we can now reach node 4 via $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$.)

$C_5 = \min(\infty, 6 + \infty)$
$= \infty$

Iteration 3: Step 1. Select the minimal value of C.
$C_4 = 9$ (the smallest value of the remaining C_i)
Move the node into S.
$S = \{1, 2, 3, 4\}$ $U = \{5\}$

Step 2. Recompute the C values for all nodes still in set U:
$C_5 = \min(\infty, 9 + 9)$
$= 18$ (we have reduced it from infinity)

Iteration 4: Step 1. Select the minimal value of C.
$C_5 = 18$ (the smallest value of the remaining C_i)
Move the node into S.
$S = \{1, 2, 3, 4, 5\}$ $U = \{\}$

Step 2. The set U is now empty, so we are finished.

[FIGURE 8-27] Example of Dijkstra's algorithm

At the end of the operations shown in Figure 8-27, we determined the cost of the shortest path from source node 1 to all other nodes in the graph. They are the most recent values in the cost list C.

NODE	SHORTEST PATH COST C[j]
1	0
2	5
3	6
4	9
5	18

Figure 8-28 shows a method called `dijkstraSP` that implements the shortest path technique just described. Rather than returning the actual path, the algorithm of Figure 8-28 returns an array that contains the *cost* of the shortest path from the source node to all other nodes. We leave it as a reader exercise to make the necessary changes so the algorithm returns a *list* of the actual nodes contained in the shortest path.

```
/**
 * Perform Dijkstra's shortest path algorithm on the given graph,
 * starting at the given vertex.
 *
 * @param g the graph to traverse.
 * @param name the name of the vertex where the traversal starts.
 *
 * @return an array containing vertex path costs.
 */
public int[] dijkstraSP( Graph g, String name ) {
    // The vertices for which the shortest path is not known
    Set<String> u = new ArrayBasedSet<String>();

    // The vertices for which the shortest path is known
    Set<String> s = new ArrayBasedSet<String>();

    // Put the vertices in an array to make things easier
    List<Vertex> vertices = g.getVertices();
    Vertex v[] = new Vertex[ vertices.size() ];
    for ( int i = 0; i < vertices.size(); i++ ) {
        v[ i ] = vertices.get( i );
    }

    Vertex start = g.getVertex( name );  // The starting vertex
    int c[] = new int[ v.length ];       // The lowest costs so far
    Edge curEdge = null;                 // Temporary edge
```

continued

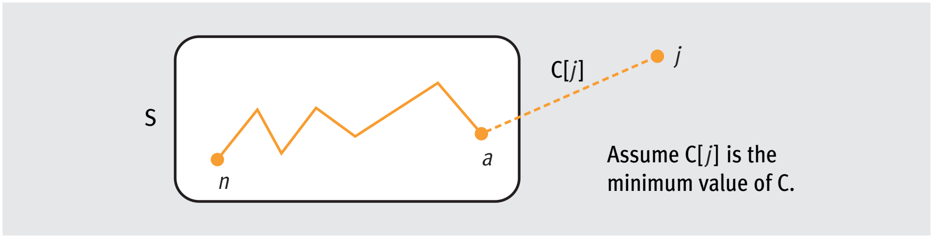
```
// Use a heap to sort the v matrix by name
Heap<String> names = new ArrayBasedHeap<String>();
for ( int i = 0; i < v.length; i = i + 1 ) {
    names.insertHeapNode( v[ i ].getName() );
}
for ( int i = 0; !names.empty(); i = i + 1 ) {
    v[ i ] = g.getVertex( names.getSmallest() );
}

s.add( name );  // We "know" the shortest path to the source

// For each vertex, compute the starting cost
for ( int i = 0; i < v.length; i = i + 1 ) {
    // If this isn't the start node
    if ( !v[ i ].getName().equals( name ) ) {
        // Put it in the unknown set
        u.add( v[ i ].getName() );

        // Compute the initial cost to reach this vertex
        curEdge = start.getEdge( v[ i ].getName() );

        if ( curEdge != null ) {
            c[ i ] = curEdge.getCost();
        }
        else {
            // This vertex is currently unreachable
            c[ i ] = Integer.MAX_VALUE;
        }
    }
    else {
        // It costs 0 to get to the start vertex
        c[ i ] = 0;
    }
}

// Setup is complete—run the algorithm

while ( !u.isEmpty() ) {
    // Find the position of the lowest-cost unknown node
    int min = Integer.MAX_VALUE;
    int minPos = -1;

    for ( int i = 0; minPos == -1 && i < c.length; i = i + 1 ) {
        if ( c[ i ] < min && u.contains( v[ i ].getName() ) ) {
            min = c[ i ];
            minPos = i;
        }
    }
```

continued

```
            // We know the shortest path to the vertex
            s.add( v[ minPos ].getName() );
            u.remove( v[ minPos ].getName() );

            // Update the costs based
            for ( int i = 0; i < c.length; i = i + 1 ) {
                // Get the edge between the new shortest path and the
                // current node in the array
                curEdge = v[ minPos ].getEdge( v[ i ].getName() );

                // If there is an edge
                if ( curEdge != null ) {
                    // If going through the new node is better than
                    // what has been seen update the cost
                    if ( c[ i ] > c[ minPos ] + curEdge.getCost() ) {
                        c[ i ] = c[ minPos ] + curEdge.getCost();
                    }
                }
            }
        }

        return c;
    }
```

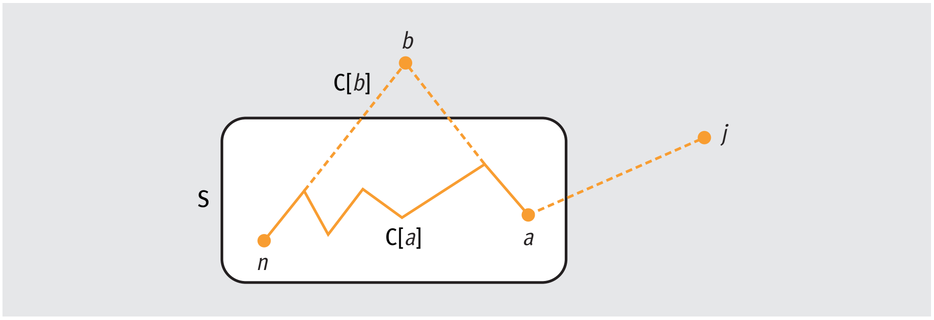
[FIGURE 8-28] Dijkstra's shortest path algorithm

A greedy algorithm does not always produce the optimal solution. To prove that Dijkstra's algorithm does find the shortest path, we must examine the following diagram:



Assume that C[*j*] is the minimal value contained in cost list C. Assume also that the shortest path to node *j* from the source node *n* wanders around nodes inside S, eventually reaching some node *a*. The cost of this wandering is C[*a*]. From there the path goes out from S directly to node *j*. We claim that this path, which we write *n* … *a* … *j*, is the lowest-cost path.

By definition, this path is the lowest-cost path containing only nodes in S, as this is the definition of C, and we selected the minimum entry C[j]. If it is not the overall shortest path to j, then there must be at least one node $b \notin$ S on the other path:



Now the proposed shortest path from n to j wanders around nodes in S, but eventually it will go outside of S to some node $b \in$ U. The cost of going from n to b via nodes within S is C[b]. However, because node $a \in$ S and node $b \notin$ S, we know that C[a] $<$ C[b] because nodes are being added to S in strict increasing order of their cost in list C. Therefore, the cost of path n ... a must be less than the cost of the path n ... b ... a, and the path n ... a ... j must be lower in cost than the path n … b … a … j. Thus, Dijkstra's algorithm does produce the shortest path.

It is easy to show that Dijkstra's algorithm is O(n^2), where n is the number of nodes in the graph. In each iteration of the algorithm, we move one node from set U to set S. Originally, there are ($n - 1$) nodes in U, so repeating the algorithm until U is empty requires O(n) steps. During each iteration, we must locate the minimum value in the cost list C and recompute all values in C. The length of C is originally ($n - 1$), and decreases by one after each iteration. The number of steps required to find a minimum of n elements and to recompute a new value for n elements is obviously O(n), resulting in an O(n^2) algorithm.

THEORY VS. PRACTICE

We have just described Dijkstra's algorithm, which finds the optimal path between any two nodes in a graph in O(n^2) time. This algorithm is used to route messages on the Internet. However, there are significant differences between the Internet implementation of Dijkstra's shortest path algorithm and our theoretical description of it in this chapter. A number of reasons account for this difference:

- *The dynamic nature of the Internet*—In our examples we had complete knowledge of the graph—nodes, links, and weights—and this information did not change while we were solving a problem. However, the Internet is enormously dynamic, and keeping a complete snapshot of the network is impossible; it changes every second. New nodes and links are added or upgraded, while others fail and are removed. Our overall picture of the network is outdated the minute it is created, and our algorithm will compute with old, invalid information.
- *The error-prone nature of the Internet*—To discover the structure of the network, nodes exchange information. They send routing control messages to other nodes in the network to share information they have collected about delays along their links. Unfortunately, like any other network message, routing control messages can be garbled or lost and never reach their intended destination. In our theoretical description of the shortest path algorithm, we never had to worry about losing our data. But it can happen in the Internet, and we must be able to deal with it.
- *The enormous scale of the Internet*—As of early 2006, the Internet had about four hundred million (4×10^8) hosts. Therefore, an algorithm that runs in O(n^2) time, which in theory sounds reasonable, would require each node to carry out $(4 \times 10^8)^2 = 1.6 \times 10^{17}$ computations. Most systems cannot do this in a reasonable amount of time. (A typical desktop system can do 2 billion computations/second, which would require five years to complete the all-pairs shortest path computations we have described.)

As you can see, there is a gap between textbook theory and practical solution. That is why computer science requires a study of theoretical, mathematical skills, such as discovering and analyzing algorithms, as well as applied engineering skills, such as getting

continued

algorithms to work successfully and efficiently in an actual environment. Although the fundamental basis of Internet routing is Dijkstra's shortest path algorithm, a number of assumptions, modifications, and simplifications were required to make it work in the real world. Fortunately, software developers were able to overcome these problems, and the actual Internet routing algorithm—called OSPF, for Open Shortest Path First—works well in the massive, dynamic, and non-fault-tolerant environment called the Internet.

8.3.3 Implementation of Graphs

8.3.3.1 The Adjacency Matrix Representation

Two popular techniques for representing graph structures are the **adjacency matrix** and the **adjacency list**. This section looks at the first of these two approaches.

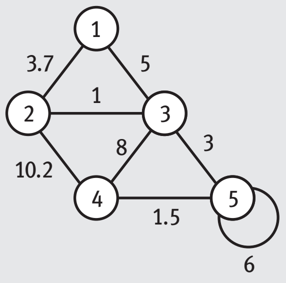
Assume that the n nodes in graph G are uniquely numbered $0, \ldots, n - 1$. An adjacency matrix represents the graph using an $n \times n$ two-dimensional array M of Boolean values, as shown in the following:

```
private final int MAX = … ; // The matrix size
private boolean M[][] = new boolean[MAX][MAX];
private int n;         // Actual number of nodes in M. 0 ≤ n ≤ MAX
```

If $M(x, y) =$ true, there is a directed edge from node x to node y, $0 \leq x, y < n$. If $M(x, y) =$ false, there is no edge. The amount of space needed to represent G is $O(MAX^2)$.

If the graph is undirected, then $M(x, y) = M(y, x)$ for all x, y, and we need to store only the upper or lower triangular portion of the adjacency matrix. If the edges are weighted, we can implement the graph as a two-dimensional array of real numbers rather than Boolean operators, with $M[i, j]$ representing the numerical weight of the edge $<i, j>$.

Figure 8-29 is an example of an adjacency matrix representation of an undirected, weighted graph. The symbol ∞ stands for some extremely large positive number, and represents infinite cost. It is used to encode the absence of an edge between two nodes.



	1	2	3	4	5
1	∞				
2	3.7	∞			
3	5.0	1.0	∞		
4	∞	10.2	8.0	∞	
5	∞	∞	3.0	1.5	6.0

[FIGURE 8-29] Adjacency matrix representation of an undirected, weighted graph

An adjacency matrix representation makes sense only if a graph is **dense**; that is, if it contains a large percentage of the n^2 edges that can theoretically exist in a graph with n vertices. However, if a graph is **sparse** and contains something closer to the minimum number of edges in a connected graph, namely $(n - 1)$, the adjacency matrix will be quite empty, with only about $1/n$ of its cells having values other than ∞. For example, in a directed graph with 30 nodes, the 30×30 adjacency matrix representation would require 900 cells; however, the graph could be connected with as few as 29 edges, leaving 871 cells unused. If the graph were 1000×1000, the adjacency matrix would require one million cells even though as few as 999 might be occupied.

The basic operations of inserting and deleting an edge are particularly easy to implement using the adjacency matrix representation because of the random access property of arrays. To add an edge from node i to node j, we simply say:

$M[i,j] = \text{true}$ (or $M[i,j] = r$ for some real value r if the graph is weighted), which takes O(1) time

Deletion of an edge is virtually identical:

$M[i,j] = \text{false}$ (or $M[i,j] = \infty$ if it is a weighted graph), which is also O(1)

Adding a new node to a graph involves setting all the entries in both row n and column n of the matrix to false or ∞, and then incrementing the value of n, the total number of nodes in G. Because of the fixed-size restriction on arrays, this operation can fail if $n = \text{MAX}$. In that case, there is no room for another node, and we must resize the array and copy all the values in the existing array into the new structure.

When deleting a node, we must not only mark node *i* as deleted, we must also locate all edges connected to node *i* and remove them from the matrix. This involves traversing both row *i* and column *i* of M, looking for any edge that either begins or ends at node *i*, and setting it to false (or to ∞ if the graph is weighted):

```
// Deleting node i
for (k = 0; k < n; k++)     {
      M(i, k) = false;
      M(k, i) = false;
}
```

This is an O(*n*) operation.

8.3.3.2 The Adjacency List Representation

A potentially more space-efficient method for representing graphs is the **adjacency list**. With this technique we first create an *n*-element, one-dimensional array A[MAX], where MAX is the maximum number of nodes in the graph. This is called the **node list**. Each element A[*i*] in the node list is the head of a linked list of all the neighbors of node *i*—that is, all nodes connected to node *i* by an edge in the graph. This is called a **neighbor list**; each entry in it is called a **neighbor node**. For example, if the neighbor list beginning at the entry A[3] includes the value 5, there is a directed edge from node 3 to node 5. If the edge is undirected, the neighbor list will have a separate entry pointed at by A[5] that includes the value 3.

Referring back to the five-node graph in Figure 8-29, the adjacency list representation of G, excluding the weight field, would be:

NODE	NEIGHBOR LIST
1	2 → 3
2	1 → 3 → 4
3	1 → 2 → 4 → 5
4	2 → 3 → 5
5	3 → 4 → 5

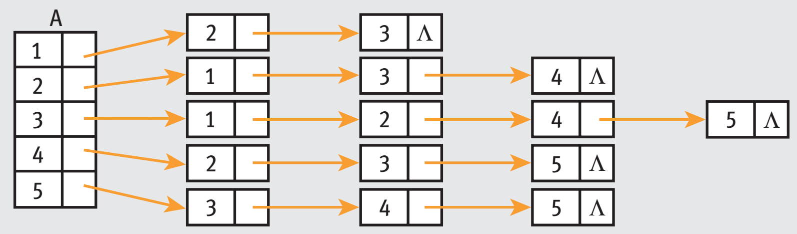
It is easy to include the concept of weighting in an adjacency list. Simply make the elements in the neighbor list 2-tuples of the form (node number, weight). Now the weighted graph G of Figure 8-29 is represented as follows:

Node	*Neighbor List*
1	(2, 3.7) → (3, 5.0)
2	(1, 3.7) → (3, 1.0) → (4, 10.2)
3	(1, 5.0) → (2, 1.0) → (4, 8.0) → (5, 3.0)
4	(2, 10.2) → (3, 8.0) → (5, 1.5)
5	(3, 3.0) → (4, 1.5) → (5, 6.0)

An adjacency list can be implemented using an array of references to objects of type `Neighbor`. Each `Neighbor` object contains a node identifier, a weight field (optional), and a reference to another node:

```
private final int MAX = … ;     // Maximum number of nodes
     public class Neighbor     {
          private int nodeID;        // The node number
          private float edgeWeight; // Weight associated
                                     // with this edge
          private Neighbor next;     // Next node in the
                                     // neighbor list
…         // Public accessors and mutators here
}
          private Neighbor A[MAX];  // The node list
```

Using these declarations, the directed graph of Figure 8-29 is represented as shown in Figure 8-30.



[FIGURE 8-30] Internal representation of graphs using adjacency lists

In a directed, unweighted graph containing n nodes and E edges, there is one reference for each of the entries in the node list, and one integer (the node ID) and one reference (next) for each edge E—two if the edges are undirected. If both an **int** and a reference require four bytes, then the amount of space needed for the adjacency list representation of a directed, unweighted graph G is:

$$\begin{aligned} & \text{space for the node list} + \text{space for the neighbor list nodes} \\ = \quad & (4 * n) + (4 + 4) * \mathrm{E} = 4n + 8\mathrm{E} = \\ = \quad & \mathrm{O}(n + \mathrm{E}) \end{aligned}$$

This compares favorably with the $\mathrm{O}(n^2)$ cells required by the adjacency matrix technique.

Figure 8-31 compares the memory space needed by these two techniques to represent a directed, unweighted graph G with $n = 50$ nodes, as the number of edges E varies from 49 to 2500, the minimum and maximum for a connected graph. The table does not include any space required by a weight field, and it assumes that four bytes are needed to store both a node identifier and a reference variable.

	SPACE REQUIRED (IN BYTES) FOR A GRAPH WITH n = 50 NODES		
	EDGES	ADJACENCY MATRIX	ADJACENCY LIST
(minimum)	49	10,000	592
	100	10,000	1000
	250	10,000	2200
	500	10,000	4200
	1000	10,000	8200
	1200	10,000	9800
	1500	10,000	12,200
	2000	10,000	16,200
(maximum)	2500	10,000	20,200

[FIGURE 8-31] Comparison of the space needs of two graph representation techniques

For graphs with less than approximately 1200 edges, the adjacency list is a more compact representation. The smallest connected graph, E = 49, has a space savings of 94 percent, an enormous reduction. As the graph becomes denser, this space savings decreases, and the adjacency matrix technique eventually becomes superior because we no longer need the large number of references required by the adjacency list representation. Although the cutoff point will change for different graph sizes and graph types, the basic conclusions about the space efficiency of these two techniques remain the same.

Assume that our node list is an array that can be accessed in O(1) time. The operations of adding a new edge and removing an existing edge both need to search the linked list of neighbor nodes referenced by that element in the node list. (When adding, we need to search the list to make sure that the edge is not already there.) On average, each neighbor list contains E/*n* nodes, and the insert and delete edge operations O(E/*n*) use the adjacency list method.

To delete a node *i*, we first set A[*i*] to Λ. This eliminates every edge for which node *i* is the head node of the edge. However, we must also delete every edge for which node *i* is the tail. This involves a search of every neighbor list in the table, which requires O(E) time, where E is the total number of edges in the graph. For example, to delete node 1 in Figure 8-30, we not only must set A[1] to Λ, we must also remove the entries for (2, 1) and (3, 1) contained in the neighbor lists of nodes 2 and 3, respectively.

Finally, to delete edge <*m*, *n*> from a graph, we must search the neighbor list referenced by A[*m*] looking for an entry that contains the node identifier *n*. This requires O(E/*n*) time.

Figure 8-32 summarizes the time complexities of basic node and edge insertions and deletions using the adjacency matrix and adjacency list representations.

OPERATION	ADJACENCY MATRIX	ADJACENCY LIST
Insert node	O(1)	O(E/*n*)
Insert edge	O(1)	O(E/*n*)
Delete node	O(*n*)	O(E)
Delete edge	O(1)	O(E/*n*)

[FIGURE 8-32] Time complexities of insertions and deletions

In a sparse graph where $E \approx n$, the two methods are generally comparable in running time. Given the space savings that can accrue from the adjacency list method, it becomes the technique of choice. However, as the graph gets denser, $E \approx n^2$, and the adjacency list implementation performs more poorly than the adjacency matrix method and uses more space. In this situation, the adjacency matrix method is definitely the technique of choice.

Analyzing the running times of graph algorithms often depends on which implementation technique is used to represent the graph internally. For example, the breadth-first traversal algorithm in Figure 8-20 has an outer **while** loop that is executed *n* times, because we must visit all *n* vertices in the graph before we finish. Within this outer loop, the critical operation is an inner loop that locates all the neighbors of a given vertex *v* and checks their status. If we are using adjacency matrices to represent a graph, we must examine all *n* elements in row v of the adjacency matrix looking for every occurrence of the value true. This takes $O(n)$ time. Coupled with the n repetitions of the outer loop, the complexity of a breadth-first graph traversal algorithm implemented with adjacency matrices is $O(n^2)$.

Using adjacency lists, the number of edges in each of the n linked lists is approximately E/n. Because we must search this list each time we execute the outer loop of the algorithm in Figure 8-19, the complexity of a breadth-first traversal using adjacency lists is $n * (E/n) = O(E)$. If the graph is dense, then $E \approx n^2$, and the two methods take roughly the same time. In sparse graphs, $E \approx n$ and the complexity is closer to $O(n)$, a significant improvement.

8.4 Summary

This chapter completes our discussion of the four data structure classifications introduced in Chapter 6—linear, hierarchical, sets, and graphs. Of course, the subject of data structures is enormous, and a number of advanced topics were not covered, including frequency-ordered lists, indexed lists, AVL trees, bags, and multigraphs. You will study many of these structures in future computer science courses.

An important trend in modern software development is the rapid growth of data structure libraries. For many of the data structures we have studied, it is unnecessary to develop your own code from scratch. Instead, you can import and reuse existing code from a standard library. As we have mentioned several times, code reuse is one of the most important techniques for increasing software productivity. When it comes to the topic of data structures, a software developer should first ask not how to build them, but where to find them.

In Java, this library of standard data structures is called the **Java Collection Framework**; the next chapter introduces you to this important set of packages and classes. Many of the data structures discussed in Chapters 6, 7, and 8 already exist, and are immediately available to the Java developer. The next chapter also explains how to extend the Java Collection Framework using inheritance and polymorphism to construct new data structures that are not yet included in the library. This type of code reuse and class extension are fundamental characteristics of modern software development.

EXERCISES

1 Given the following four sets:

A = {5, 10, 15, 20} C = {6, 8, 10}

B = {1, 2, 3, 4, 5} D = {}

What is the result of performing each of the following set methods on the preceding sets?

a $A \cup B$

b $A \cup C$

c $A \cup D$

d $A \cap B$

e $A \cap D$

f $B \cap C$

g $A - B$

h $B - A$

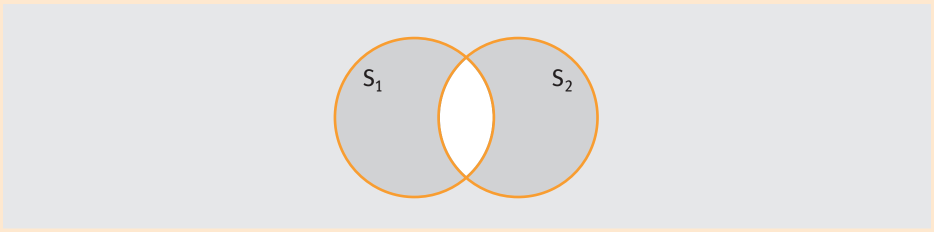
i `D.isEmpty()`

j `C.contains(6)`

k `D.contains(0)`

l `A.subset(B)`

2 Define the calling sequence and pre- and postconditions for a new set method called `nIntersection`, for nonintersection. This operation produces a new set containing only elements that are either in set S_1 or in set S_2, but not in both. (In graph theory, this operation is called the **symmetric difference**.) Using the Venn diagrams of Section 8.1.1, this operation produces a new set containing only the values in the shaded area:



Was this operation really necessary? That is, could it have been defined in terms of the methods contained in the `Set` interface of Figure 8-1?

3 How would you revise the intersection method in Figure 8-3 if the elements of the two sets S_1 and S_2 were integer objects sorted into ascending order? What is the time complexity of this newly revised method?

4 Assume that our set is implemented as a linked list using the following declarations:

```
public class SetNode     {  // The class representing the
                            // nodes in the linked list
   private Object info;
   private SetNode next;
}

SetNode   head;             // Pointer to the first
                            // element of this set
```

In addition, assume that the class includes the standard accessor and mutator methods, `getInfo()`, `getNext()`, `setInfo()`, and `setNext()`. Implement the `union` and `difference` methods contained in the `Set` interface of Figure 8-1.

5 Assume that we use a bit vector implementation for sets of the base type [0..10]. Give the declaration for the bit vector array itself, and show the internal representation of each of the following sets using this array.

a {2, 4, 6, 8}

b {0, 1, 2, 3, 4, 6, 7, 8, 9, 10}

c {}

d {1, 2}

6 Write the bit vector implementation of the `Set` interface in Figure 8-1. Include the implementation of all 10 methods in this interface.

7 Using the same hashing function shown in Figures 8-9a and b, where would the information on "Tymann" be stored in the hash table? What about "Schneider"?

8 The keys that you want to store in a 50-element hash table are strings of exactly five characters in length: $c_1c_2c_3c_4c_5$. Assume you are using the following hashing function:

$$f = [\sum_{i=1}^{5} (a * \texttt{intValue}\ (c_i))]\ \%\ 50$$

where `intValue` returns the Unicode value of the character c_i

Select a large prime value for a and test how well your hashing function works on a selection of 1000 random five-letter sequences. Does it scatter reasonably well over the 50 slots in your hash table? If not, choose another value for a and see if it works better. What happens when you choose $a = 2$? (If you have had an elementary course in statistics, you can use the chi-square Goodness of Fit test to formally test the scattering ability of function f. If you do not know this test, just "eyeball" the results.)

9 **a** Given a hash table of size $n = 8$, with indices running from 0 to 7, show where the following keys would be stored using hashing, open addressing, and a step size of $c = 1$ (that is, if there is a collision search sequentially for the next available slot). Assume that the hash function is just the ordinal position of the letter in the alphabet modulo 8—in other words, f('a') = 0, f('b') = 1, … , f('h') = 7, f('i') = 0, etc.

'd', 'e', 'l', 't', 'g', 'h', 'q'

b Repeat the same operations, but this time use a step size of $c = 3$.

c Why must the step size c be relatively prime with the table size n? Show what happens in Exercise 9b if you select a step size of $c = 4$.

10 Assume a hash table size $n = 50{,}000$. After how many insertion operations (with no deletions) will retrieval using hashing and open addressing display about the same performance, in terms of the worst-case number of comparisons, as binary search? How about sequential search?

11 A major problem with open addressing is the issue of deletions. We cannot simply delete entries in a hash table that uses open addressing for collision resolution.

a Show exactly what happens with a 10-element integer hash table `h`, open addressing, and the following hash function:

$f(key) = (key \% 10)$

when we execute the following five operations, one right after the other:

- Insert the value 10.
- Insert the value 20.
- Insert the value 30.
- Delete the value 20.
- Retrieve the value 30.

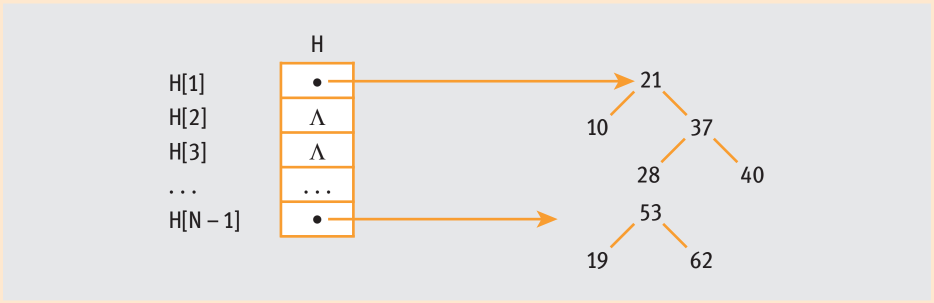
b Is there any way we can solve the problem? If so, describe your solution and implement the modified delete operation you designed.

12 Using the data and hash function from Exercise 9a and a table size of $n = 8$, show what the hash table looks like after insertion of the seven letters if we use the chaining technique.

13 Assume that we are using chaining and want to store approximately 1000 keys in our hash table. How large should we make the hash table so that the average number of comparisons needed to retrieve any given key is approximately 4? How much extra space is required for the table as well as the 1000 keys, compared with keeping the 1000 keys in an array? Disregard the value field, and assume that each object requires 4 bytes.

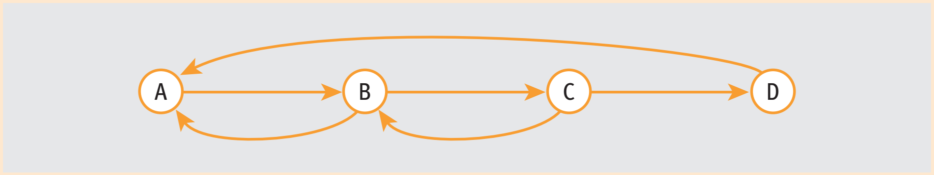
14 Given the dictionary analysis of Figure 8-14, what would be the average number of comparisons to perform a retrieval if we were using chaining and could increase the size of our hash table to $n = 60{,}000$? How much extra memory is required? Do you think this approach is beneficial? Explain why or why not.

15 Implement a variation of the chaining method in which each entry in the hash table is a reference to the root of a binary search tree, as discussed in Section 7.4. This binary search tree contains all of the (*key*, *value*) entries that hashed to this location. The structure of a hash table H would look like the following:



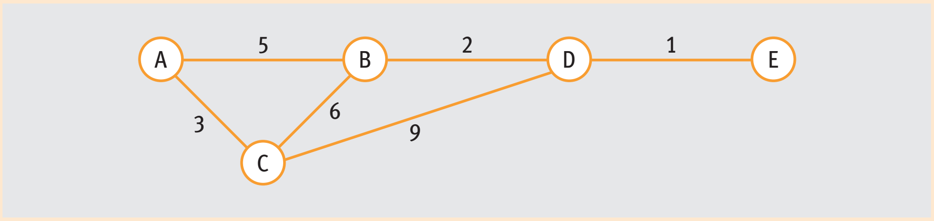
Design and implement a Java class that implements the `Map` interface of Figure 8-5 using this variation of the chaining method.

16 Given the following unweighted, directed graph:



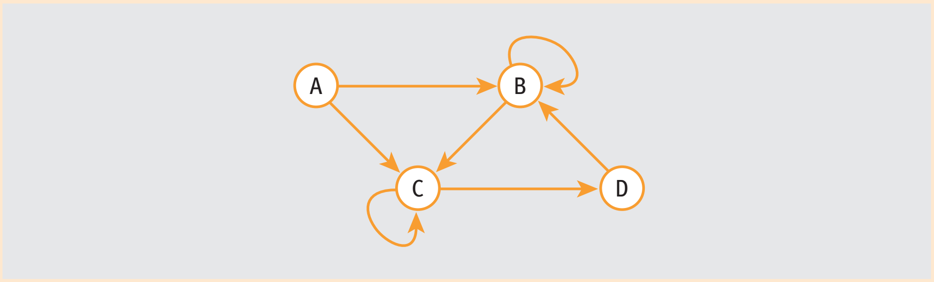
list its nodes, edges, and simple cycles. Why is the weighted, directed graph considered the most general of the different graph types?

17 Examine the following weighted, undirected graph:



Identify all the simple paths from A to E. What is the shortest path?

18 Is the following graph connected or disconnected? Explain your answer.



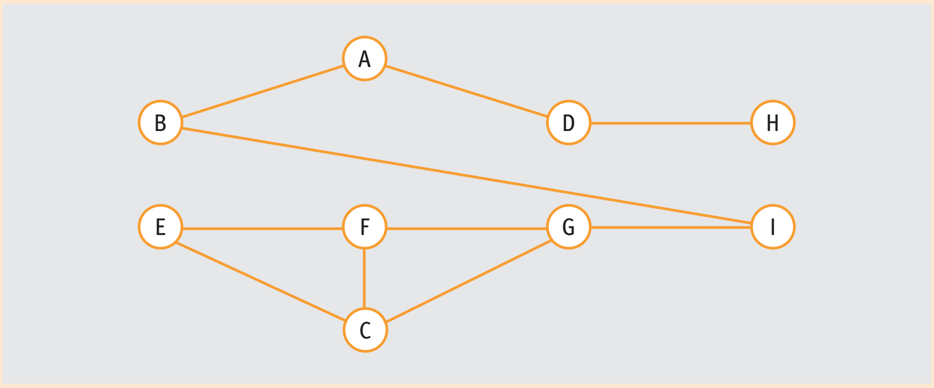
19 What graph structure results from the following sequence of basic operations, assuming that G is initially empty?

```
G.addVertex(A)
G.addVertex(B)
G.addVertex(C)
G.addVertex(D)
G.addEdge(A, B, 1)
G.addEdge(B, C, 1)
G.addEdge(C, D, 2)
```

20 What is wrong with the following pair of graph operations?

```
G.addEdge(A, B, 1)
G.addEdge(A, B, 5)
```

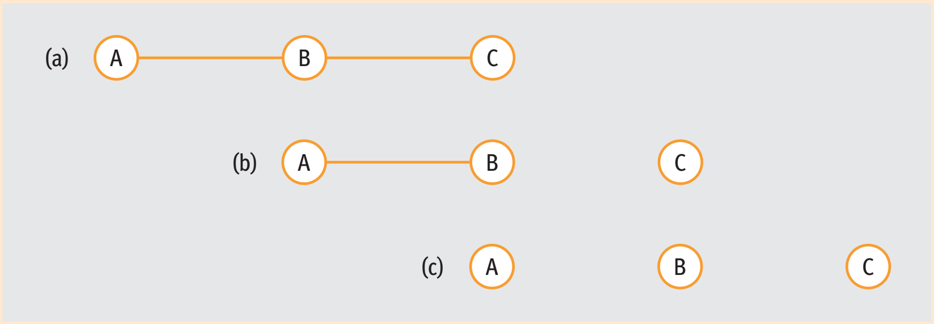
21 **a** In exactly what order would the nodes in the following graph be visited using a breadth-first traversal beginning at node C?



What if we began from node I? (Assume that nodes are queued in strict alphabetical order.)

b Repeat Exercise 21a, but this time use a depth-first traversal.

22 Design and implement an instance method that determines, for a given graph G, the total number of connected components within G. For example, given the following three graphs:



your method would output the values 1, 2, and 3, respectively. You can use the breadth-first traversal method of Figure 8-20 in your solution.

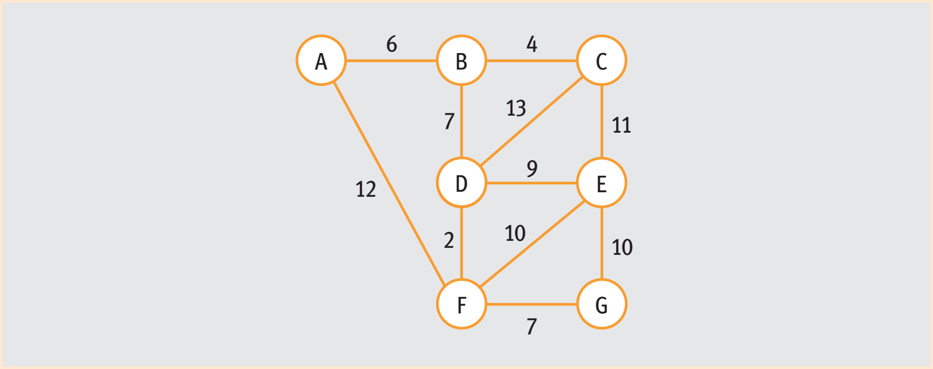
23 Design and implement an instance method that, for a given graph G, performs a depth-first traversal of the nodes of G.

24 Using the nine-node graph diagrammed in Exercise 21, determine the following:

a A breadth-first spanning tree rooted at node B

b A depth-first spanning tree rooted at node C

25 a Given the following weighted, undirected graph G:



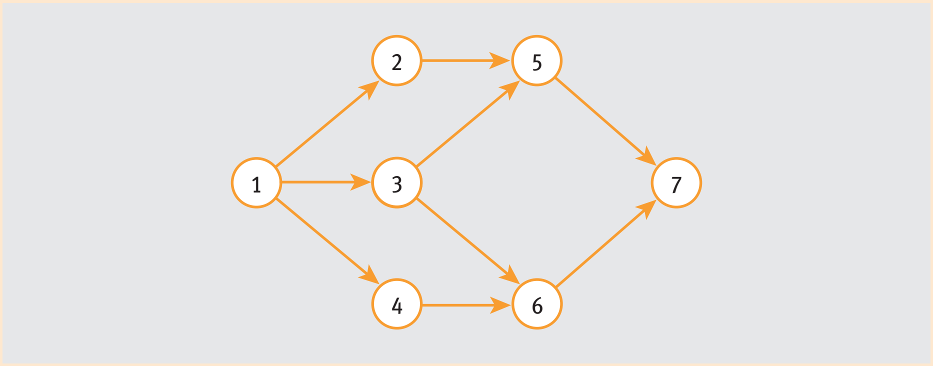
Find its minimum spanning tree using Kruskal's algorithm (described in Section 8.3.2.3).

b Determine the time complexity of Kruskal's algorithm in terms of the number of nodes, n, and the number of edges, E.

26 Using the weighted graph of Exercise 25, what is the shortest path from node A to node E? From node B to node G? Determine the cost of the shortest path from node A to all other nodes using Dijkstra's algorithm.

27 Modify Dijkstra's algorithm in Figure 8-28 so that it prints the edges contained in the shortest path as well as the cost of that path.

28 Given the following directed acyclic graph (DAG):



Show its representation using:

a An adjacency matrix

b An adjacency list

Determine how much space is required for each method. For this example, state which of the two is more space efficient. Assume that all stored values (integers and references) require four bytes.

29 Write a method `addEdge(i,j,W)` that inserts the edge $<i,j,W>$ into a graph G. Assume that G is represented using the adjacency list technique described in Section 8.3.3.2.

CHALLENGE WORK EXERCISES

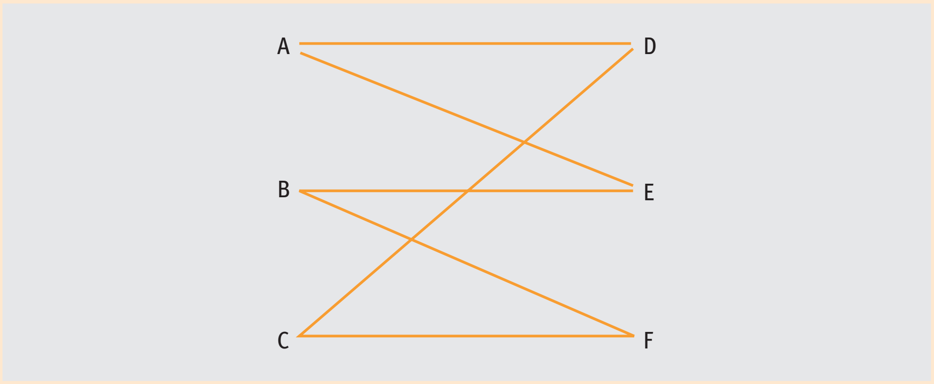
Graphs are the most important of the three ordered data structure groups, which is why they have been studied by mathematicians for hundreds of years, dating back to the Swiss mathematician Leonard Euler in the 1730s. Today, graph theory is an active and popular branch of mathematical research.

In this chapter, we introduced only a small number of interesting graph problems:

- Graph traversal (Section 8.3.2.2)
- Minimum spanning trees (Section 8.3.2.3)
- Shortest paths (Section 8.3.2.4)

However, there are many other interesting graph-related problems with important applications in such fields as transportation, engineering, manufacturing, and communications. Consider the following problems:

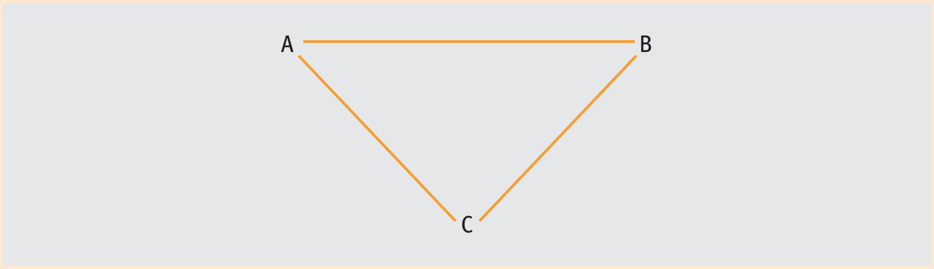
1 Determining if a graph is **bipartite**. A graph is bipartite if its vertices can be divided into two disjoint sets so that no two vertices in the same set are connected by an edge. For example, the following graph:



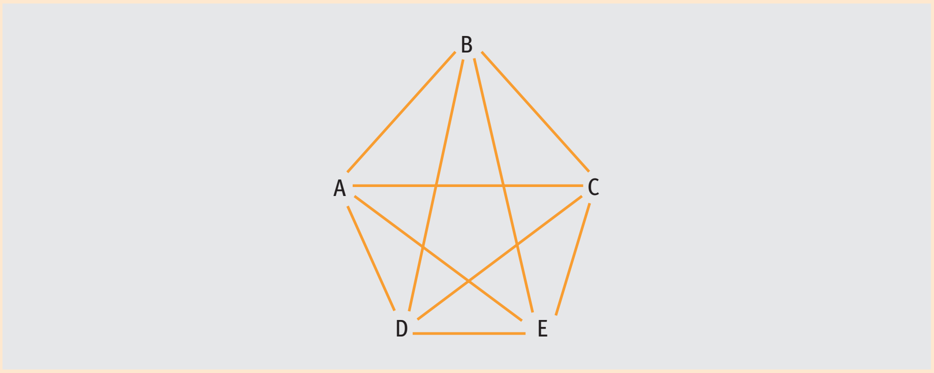
can be partitioned into the two sets S1 = {A, B, C} and S2 = {D, E, F} so that every edge begins with a node in one set but terminates at a node in the other set. Thus, this graph is bipartite. Bipartite graphs are widely used to solve *matching problems*. For example, A, B, and C may be students, while D, E, and F are courses with a registration limit. There is an edge between a student and a course if the student wants to register for the course.

Using this graph representation, develop an algorithm to obtain the maximum number of students registered for each class, given course limits, while shutting out the fewest number of students from a class they wanted to take.

2 Determining if a graph is **planar**. A planar graph can be drawn on a two-dimensional surface so that none of its edges cross each other. For example, the following graph is planar:

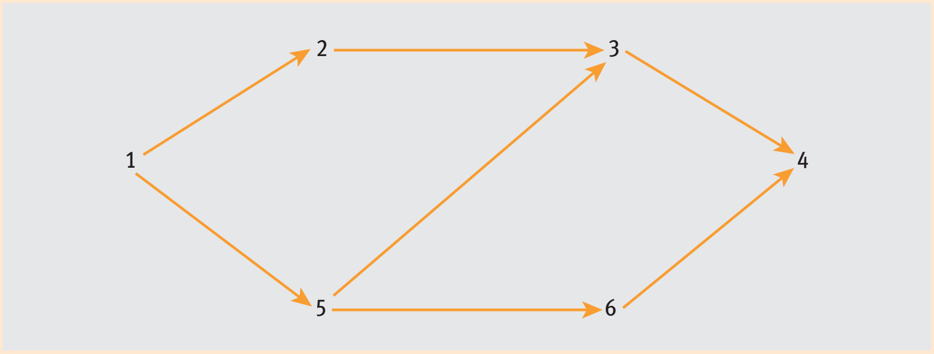


However, no matter how you try to redraw the edges in the following graph:



it cannot be done without edge crossings. Planarity is a very important property in *circuit design*; you must design a circuit board without any crossings, because the crossing of two wires can cause a short circuit.

3 Finding the **topological sort** of a directed acyclic graph (DAG). A topological sort is a linear ordering of nodes [n_1, n_2, n_3, …] so that if there is an edge in the DAG of the form $<n_i, n_j>$, then n_i comes before n_j in the linear ordering. For example, given the following DAG:



There are a number of different topological sorts, including:

1, 2, 5, 3, 6, 4
1, 5, 2, 6, 3, 4
1, 2, 5, 6, 3, 4

Topological sorting is very important in applications that deal with *scheduling*. For example, the preceding DAG could represent the six steps necessary to complete a task. A topological sort can tell you an order in which you can meaningfully perform these steps.

Investigate these three graph problems and learn about the algorithms that solve them. Then implement the algorithms and test them on actual graph structures.

For more information on graph structures and problems on graphs, consult the following sources:

- *http://en.wikipedia.org/wiki/Graph_theory*
- Fred Buckley, Martin Lewinter, *A Friendly Introduction to Graph Theory*, Prentice Hall Publishing Co., 2002, ISBN 0-13-0669490
- G. Chartrand, et al., *Introduction to Graph Theory*, McGraw-Hill, 2004, ISBN 0-07-2948620