CHAPTER 7

The Set Data Model

The set is the most fundamental data model of mathematics. Every concept in mathematics, from trees to real numbers, is expressible as a special kind of set. In this book, we have seen sets in the guise of events in a probability space. The dictionary abstract data type is a kind of set, on which particular operations — *insert*, *delete*, and *lookup* — are performed. Thus, it should not be surprising that sets are also a fundamental model of computer science. In this chapter, we learn the basic definitions concerning sets and then consider algorithms for efficiently implementing set operations.

7.1 What This Chapter Is About

This chapter covers the following topics:

- The basic definitions of set theory and the principal operations on sets (Sections 7.2–7.3).
- The three most common data structures used to implement sets: linked lists, characteristic vectors, and hash tables. We compare these data structures with respect to their relative efficiency in supporting various operations on sets (Sections 7.4–7.6).
- Relations and functions as sets of pairs (Section 7.7).
- Data structures for representing relations and functions (Sections 7.8–7.9).
- Special kinds of binary relations, such as partial orders and equivalence relations (Section 7.10).
- Infinite sets (Section 7.11).

7.2 Basic Definitions

In mathematics, the term "set" is not defined explicitly. Rather, like terms such as "point" and "line" in geometry, the term set is defined by its properties. Specifically, there is a notion of *membership* that makes sense only for sets. When S is a set and x is anything, we can ask the question, "Is x a member of set S?" The set S then consists of all those elements x for which x is a member of S. The following points summarize some important notations for talking about sets.

1. The expression $x \in S$ means that the element x is a member of the set S.

2. If $x_1, x_2, \ldots, x_n$ are all the members of set S, then we can write

   $$S = \{x_1, x_2, \ldots, x_n\}$$

   Here, each of the x's must be distinct; we cannot repeat an element twice in a set. However, the order in which the members of a set are listed is arbitrary.

Empty set

3. The empty set, denoted $\emptyset$, is the set that has no members. That is, $x \in \emptyset$ is false, no matter what x is.

◆ **Example 7.1.** Let $S = \{1, 3, 6\}$; that is, let S be the set that has the integers 1, 3, and 6, and nothing else, as members. We can say $1 \in S$, $3 \in S$, and $6 \in S$. However, the statement $2 \in S$ is false, as is the statement that any other thing is a member of S.

Sets can also have other sets as members. For example, let $T = \{\{1, 2\}, 3, \emptyset\}$. Then T has three members. First is the set $\{1, 2\}$, that is, the set with 1 and 2 as is sole members. Second is the integer 3. Third is the empty set. The following are true statements: $\{1, 2\} \in T$, $3 \in T$, and $\emptyset \in T$. However, $1 \in T$ is false. That is, the fact that 1 is a member of a member of T does not mean that 1 is a member of T itself. ◆

Atoms

In formal set theory, there really is nothing but sets. However, in our informal set theory, and in data structures and algorithms based on sets, it is convenient to assume the existence of certain *atoms,* which are elements that are not sets. An atom can be a member of a set, but nothing can be a member of an atom. It is important to remember that the empty set, like the atoms, has no members. However, the empty set is a set rather than an atom.

We shall generally assume that integers and lowercase letters denote atoms. When talking about data structures, it is often convenient to use complex data types as the types of atoms. Thus, atoms may be structures or arrays, and not be very "atomic" at all.

Definition of Sets by Abstraction

Enumeration of the members of a set is not the only way we may define sets. Often, it is more convenient to start with some set S and some property of elements P, and define the set of those elements in S that have property P. The notation for this operation, which is called *abstraction,* is

Sets and Lists

Although our notation for a list, such as $(x_1, x_2, \ldots, x_n)$, and our notation for a set, $\{x_1, x_2, \ldots, x_n\}$, look very much alike, there are important differences. First, the order of elements in a set is irrelevant. The set we write as $\{1, 2\}$ could just as well be written $\{2, 1\}$. In contrast, the list $(1, 2)$ is not the same as the list $(2, 1)$.

Second, a list may have repetitions. For example, the list $(1, 2, 2)$ has three elements; the first is 1, the second is 2, and the third is also 2. However, the set notation $\{1, 2, 2\}$ makes no sense. We cannot have an element, such as 2, occur as a member of a set more than once. If this notation means anything, it is the same as $\{1, 2\}$ or $\{2, 1\}$ — that is, the set with 1 and 2 as members, and no other members.

Multiset or bag

Sometimes we speak of a *multiset* or *bag,* which is a set whose elements are allowed to have a multiplicity greater than 1. For example, we could speak of the multiset that contains 1 once and 2 twice. Multisets are not the same as lists, because they still have no order associated with their elements.

$$\{x \mid x \in S \text{ and } P(x)\}$$

or "the set of elements x in S such that x has property P."

Set former

The preceding expression is called a *set former.* The variable x in the set former is local to the expression, and we could just as well have written

$$\{y \mid y \in S \text{ and } P(y)\}$$

to describe the same set.

◆ **Example 7.2.** Let S be the set $\{1, 3, 6\}$ from Example 7.1. Let $P(x)$ be the property "x is odd." Then

$$\{x \mid x \in S \text{ and } x \text{ is odd }\}$$

is another way of defining the set $\{1, 3\}$. That is, we accept the elements 1 and 3 from S because they are odd, but we reject 6 because it is not odd.

As another example, consider the set $T = \{\{1, 2\}, 3, \emptyset\}$ from Example 7.1. Then

$$\{A \mid A \in T \text{ and } A \text{ is a set }\}$$

denotes the set $\{\{1, 2\}, \emptyset\}$. ◆

Equality of Sets

We must not confuse what a set *is* with how it is represented. Two sets are *equal*, that is, they are really the same set, if they have exactly the same members. Thus, most sets have many different representations, including those that explicitly enumerate their elements in some order and representations that use abstraction.

◆ **Example 7.3.** The set $\{1, 2\}$ is the set that has exactly the elements 1 and 2 as members. We can present these elements in either order, so $\{1, 2\} = \{2, 1\}$. There are also many ways to express this set by abstraction. For example,

$$\{x \mid x \in \{1, 2, 3\} \text{ and } x < 3\}$$

is equal to the set $\{1, 2\}$. ✥

Infinite Sets

It is comforting to assume that sets are *finite* — that is, that there is some particular integer n such that the set at hand has exactly n members. For example, the set $\{1, 3, 6\}$ has three members. However, some sets are *infinite,* meaning there is no integer that is the number of elements in the set. We are familiar with infinite sets such as

1. **N**, the set of nonnegative integers
2. **Z**, the set of nonnegative and negative integers
3. **R**, the set of real numbers
4. **C**, the set of complex numbers

From these sets, we can create other infinite sets by abstraction.

✥ **Example 7.4.** The set former

$$\{x \mid x \in \mathbf{Z} \text{ and } x < 3\}$$

stands for the set of all the negative integers, plus 0, 1, and 2. The set former

$$\{x \mid x \in \mathbf{Z} \text{ and } \sqrt{x} \in \mathbf{Z}\}$$

represents the set of integers that are perfect squares, that is, $\{0, 1, 4, 9, 16, \dots\}$.

For a third example, let $P(x)$ be the property that x is prime (i.e., $x > 1$ and x has no divisors except 1 and x itself). Then the set of primes is denoted

$$\{x \mid x \in \mathbf{N} \text{ and } P(x)\}$$

This expression denotes the infinite set $\{2, 3, 5, 7, 11, \dots\}$. ✥

There are some subtle and interesting properties of infinite sets. We shall take up the matter again in Section 7.11.

EXERCISES

7.2.1: What are the members of the set $\{\{a, b\}, \{a\}, \{b, c\}\}$?

7.2.2: Write set-former expressions for the following:

a) The set of integers greater than 1000.
b) The set of even integers.

7.2.3: Find two different representations for the following sets, one using abstraction, the other not.

a) $\{a, b, c\}$.
b) $\{0, 1, 5\}$.

Russell's Paradox

One might wonder why the operation of abstraction requires that we designate some other set from which the elements of the new set must come. Why can't we just use an expression like $\{x \mid P(x)\}$, for example,

$$\{x \mid x \text{ is blue }\}$$

to define the set of all blue things? The reason is that allowing such a general way to define sets gets us into a logical inconsistency discovered by Bertrand Russell and called *Russell's paradox.* We may have met this paradox informally when we heard about the town where the barber shaves everyone who doesn't shave himself, and then were asked whether the barber shaves himself. If he does, then he doesn't, and if he doesn't, he does. The way out of this anomaly is to realize that the statement "shaves everyone who doesn't shave himself," while it looks reasonable, actually makes no formal sense.

To understand Russell's paradox concerning sets, suppose we could define sets of the form $\{x \mid P(x)\}$ for any property P. Then let $P(x)$ be the property "x is not a member of x." That is, let P be true of a set x if x is not a member of itself. Let S be the set

$$S = \{x \mid x \text{ is not a member of } x\}$$

Now we can ask, "Is set S a member of itself?"

Case 1: Suppose that S is not a member of S. Then $P(S)$ is true, and so S is a member of the set $\{x \mid x \text{ is not a member of } x\}$. But that set is S, and so by assuming that S is not a member of itself, we prove that S is indeed a member of itself. Thus, it cannot be that S is not a member of itself.

Case 2: Suppose that S is a member of itself. Then S is not a member of

$$\{x \mid x \text{ is not a member of } x\}$$

But again, that set is S, and so we conclude that S is not a member of itself.

Thus, when we start by assuming that $P(S)$ is false, we prove that it is true, and when we start by assuming that $P(S)$ is true, we wind up proving that it is false. Since we arrive at a contradiction either way, we are forced to blame the notation. That is, the real problem is that it makes no sense to define the set S as we did.

Another interesting consequence of Russell's paradox is that it makes no sense to suppose there is a "set of all elements." If there were such a "universal set" — say U — then we could speak of

Universal set

$$\{x \mid x \in U \text{ and } x \text{ is not a member of } x\}$$

and we would again have Russell's paradox. We would then be forced to give up abstraction altogether, and that operation is far too useful to drop.

7.3 Operations on Sets

There are special operations that are commonly performed on sets, such as union and intersection. You are probably familiar with many of them, but we shall review the most important operations here. In the next sections we discuss some implementations of these operations.

Union, Intersection, and Difference

Perhaps the most common ways to combine sets are with the following three operations:

1. The *union* of two sets S and T, denoted $S \cup T$, is the set containing the elements that are in S or T, or both.
2. The *intersection* of sets S and T, written $S \cap T$, is the set containing the elements that are in both S and T.
3. The *difference* of sets S and T, denoted $S - T$, is the set containing those elements that are in S but not in T.

✦ **Example 7.5.** Let S be the set $\{1, 2, 3\}$ and T the set $\{3, 4, 5\}$. Then

$$S \cup T = \{1, 2, 3, 4, 5\},\ S \cap T = \{3\},\ \text{and}\ S - T = \{1, 2\}$$

That is, $S \cup T$ contains all the elements appearing in either S or T. Although 3 appears in both S and T, there is, of course, only one occurrence of 3 in $S \cup T$, because elements cannot appear more than once in a set. $S \cap T$ contains only 3, because no other element appears in both S and T. Finally, $S - T$ contains 1 and 2, because these appear in S and do not appear in T. The element 3 is not present in $S - T$, because although it appears in S, it also appears in T. ✦

When the sets S and T are events in a probability space, the union, intersection, and difference have a natural meaning. $S \cup T$ is the event that either S or T (or both) occurs. $S \cap T$ is the event that both S and T occur. $S - T$ is the event that S, but not T occurs. However, if S is the set that is the entire probability space, then $S - T$ is the event "T does not occur," that is, the complement of T.

Venn Diagrams

It is often helpful to see operations involving sets as pictures called *Venn diagrams.* Figure 7.1 is a Venn diagram showing two sets, S and T, each of which is represented by an ellipse. The two ellipses divide the plane into four regions, which we have numbered 1 to 4.

1. Region 1 represents those elements that are in neither S nor T.
2. Region 2 represents $S - T$, those elements that are in S but not in T.
3. Region 3 represents $S \cap T$, those elements that are in both S and T.
4. Region 4 represents $T - S$, those elements that are in T but not in S.
5. Regions 2, 3, and 4 combined represent $S \cup T$, those elements that are in S or T, or both.

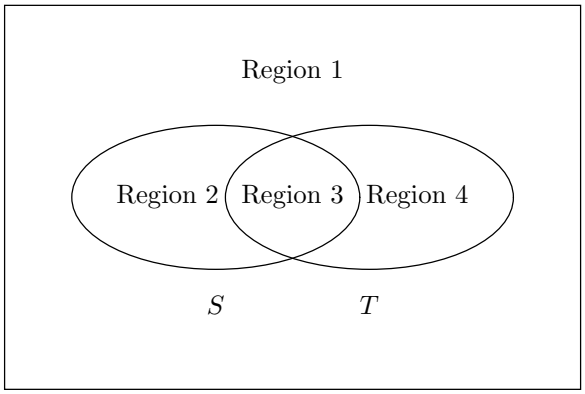

Fig. 7.1. Regions representing Venn diagrams for the basic set operations.

What Is an Algebra?

We may think that the term "algebra" refers to solving word problems, finding roots of polynomials, and other matters covered in a high school algebra course. To a mathematician, however, the term algebra refers to any sort of system in which there are operands and operators from which one builds expressions. For an algebra to be interesting and useful, it usually has special constants and *laws* that allow us to transform one expression into another "equivalent" expression.

The most familiar algebra is that in which operands are integers, reals, or perhaps complex numbers — or variables representing values from one of these classes — and the operators are the ordinary arithmetic operators: addition, multiplication, subtraction, and division. The constants 0 and 1 are special and satisfy laws like $x + 0 = x$. In manipulating arithmetic expressions, we use laws such as the distributive law, which lets us replace any expression of the form $a \times b + a \times c$ by an equivalent expression $a \times (b + c)$. Notice that by making this transformation, we reduce the number of arithmetic operations by 1. Often the purpose of algebraic manipulation of expressions, such as this one, is to find an equivalent expression whose evaluation takes less time than the evaluation of the original.

Throughout this book, we shall meet various kinds of algebras. Section 8.7 introduces relational algebra, a generalization of the algebra of sets that we discuss here. Section 10.5 talks about the algebra of *regular expressions* for describing patterns of character strings. Section 12.8 introduces the reader to the *Boolean* algebra of logic.

While we have suggested that Region 1 in Fig. 7.1 has finite extent, we should remember that this region represents everything outside S and T. Thus, this region is *not* a set. If it were, we could take its union with S and T to get the "universal set," which we know by Russell's paradox does not exist. Nevertheless, it is often convenient to draw as a region the elements that are not in any of the sets represented explicitly in the Venn diagram, as we did in Fig. 7.1.

Algebraic Laws For Union, Intersection, and Difference

Mirroring the algebra of arithmetic operations such as $+$ and $*$, one can define an

Equivalent expressions

algebra of sets in which the operators are union, intersection, and difference and the operands are sets or variables denoting sets. Once we allow ourselves to build up complicated expressions like $R \cup \big((S \cap T) - U\big)$, we can ask whether two expressions are *equivalent,* that is, whether they always denote the same set regardless of what sets we substitute for the operands that are variables. By substituting one expression for an equivalent expression, sometimes we can simplify expressions involving sets so that they may be evaluated more efficiently.

In what follows, we shall list the most important *algebraic laws* — that is, statements asserting that one expression is equivalent to another — for union, intersection, and difference of sets. The symbol $\equiv$ is used to denote equivalence of expressions.

In many of these algebraic laws, there is an analogy between union, intersection, and difference of sets, on one hand, and addition, multiplication, and subtraction of integers on the other hand. We shall, however, point out those laws that do not have analogs for ordinary arithmetic.

a) *The commutative law of union*: $(S \cup T) \equiv (T \cup S)$. That is, it does not matter which of two sets appears first in a union. The reason this law holds is simple. The element x is in $S \cup T$ if x is in S or if x is in T, or both. That is exactly the condition under which x is in $T \cup S$.

b) *The associative law of union*: $\big(S \cup (T \cup R)\big) \equiv \big((S \cup T) \cup R\big)$. That is, the union of three sets can be written either by first taking the union of the first two or the last two; in either case, the result will be the same. We can justify this law as we did the commutative law, by arguing that an element is in the set on the left if and only if it is in the set on the right. The intuitive reason is that both sets contain exactly those elements that are in either S, T, or R, or any two or three of them.

The commutative and associative laws of union together tell us that we can take the union of a collection of sets in any order. The result will always be the same set of elements, namely those elements that are in one or more of the sets. The argument is like the one we presented for addition, which is another commutative and associative operation, in Section 2.4. There, we showed that all ways to group a sum led to the same result.

c) *The commutative law of intersection*: $(S \cap T) \equiv (T \cap S)$. Intuitively, an element x is in the sets $S \cap T$ and $T \cap S$ under exactly the same circumstances: when x is in S and x is in T.

d) *The associative law of intersection*: $\big(S \cap (T \cap R)\big) \equiv \big((S \cap T) \cap R\big)$. Intuitively, x is in either of these sets exactly when x is in all three of S, T, and R. Like addition or union, the intersection of any collection of sets may be grouped as we choose, and the result will be the same; in particular, it will be the set of elements in all the sets.

e) *Distributive law of intersection over union*: Just as we know that multiplication distributes over addition — that is, $a \times (b + c) = a \times b + a \times c$ — the law

$$\big(S \cap (T \cup R)\big) \equiv \big((S \cap T) \cup (S \cap R)\big)$$

holds for sets. Intuitively, an element x is in each of these sets exactly when x is in S and also in at least one of T and R. Similarly, by the commutativity of

union and intersection, we can distribute intersections from the right, as

$$\big((T \cup R) \cap S\big) \equiv \big((T \cap S) \cup (R \cap S)\big)$$

f) *Distributive law of union over intersection*: Similarly,

$$\big(S \cup (T \cap R)\big) \equiv \big((S \cup T) \cap (S \cup R)\big)$$

holds. Both the left and right sides are sets that contain an element x exactly when x is either in S, or is in both T and R. Notice that the analogous law of arithmetic, where union is replaced by addition and intersection by multiplication, is false. That is, $a+b\times c$ is generally not equal to $(a+b)\times(a+c)$. Here is one of several places where the analogy between set operations and arithmetic operations breaks down. However, as in (e), we can use the commutativity of union to get the equivalent law

$$\big((T \cap R) \cup S\big) \equiv \big((T \cup S) \cap (R \cup S)\big)$$

✦ **Example 7.6.** Let $S = \{1,2,3\}$, $T = \{3,4,5\}$, and $R = \{1,4,6\}$. Then

$$\begin{array}{rcl} S \cup (T \cap R) & = & \{1,2,3\} \cup (\{3,4,5\} \cap \{1,4,6\}) \\ & = & \{1,2,3\} \cup \{4\} \\ & = & \{1,2,3,4\} \end{array}$$

On the other hand,

$$\begin{array}{rcl} (S \cup T) \cap (S \cup R) & = & (\{1,2,3\} \cup \{3,4,5\}) \cap (\{1,2,3\} \cup \{1,4,6\}) \\ & = & \{1,2,3,4,5\} \cap \{1,2,3,4,6\} \\ & = & \{1,2,3,4\} \end{array}$$

Thus, the distributive law of union over intersection holds in this case. That doesn't prove that the law holds in general, of course, but the intuitive argument we gave with rule (f) should be convincing. ✦

g) *Associative law of union and difference*: $\big(S-(T \cup R)\big) \equiv \big((S-T)-R\big)$. Both sides contain an element x exactly when x is in S but in neither T nor R. Notice that this law is analogous to the arithmetic law $a-(b+c) = (a-b)-c$.

h) *Distributive law of difference over union*: $\big((S \cup T)-R\big) \equiv \big((S-R) \cup (T-R)\big)$. In justification, an element x is in either set when it is not in R, but is in either S or T, or both. Here is another point at which the analogy with addition and subtraction breaks down; it is not true that $(a+b)-c = (a-c)+(b-c)$, unless $c = 0$.

i) *The empty set is the identity for union.* That is, $(S \cup \emptyset) \equiv S$, and by commutativity of union, $(\emptyset \cup S) \equiv S$. Informally, an element x can be in $S \cup \emptyset$ only when x is in S, since x cannot be in $\emptyset$.

Note that there is no identity for intersection. We might imagine that the set of "all elements" could serve as the identity for intersection, since the intersection of a set S with this "set" would surely be S. However, as mentioned in connection with Russell's paradox, there cannot be a "set of all elements."

Idempotence

j) *Idempotence of union.* An operator is said to be *idempotent* if, when applied to two copies of the same value, the result is that value. We see that $(S \cup S) \equiv S$. That is, an element x is in $S \cup S$ exactly when it is in S. Again the analogy with arithmetic fails, since $a + a$ is generally not equal to a.

k) *Idempotence of intersection.* Similarly, we have $(S \cap S) \equiv S$.

There are a number of laws relating the empty set to operations besides union. We list them here.

l) $(S - S) \equiv \emptyset$.

m) $(\emptyset - S) \equiv \emptyset$.

n) $(\emptyset \cap S) \equiv \emptyset$, and by commutativity of intersection, $(S \cap \emptyset) \equiv \emptyset$.

Proving Equivalences by Venn Diagrams

Figure 7.2 illustrates the distributive law for intersection over union by a Venn diagram. This diagram shows three sets, S, T, and R, which divide the plane into eight regions, numbered 1 through 8. These regions correspond to the eight possible relationships (in or out) that an element can have with the three sets.

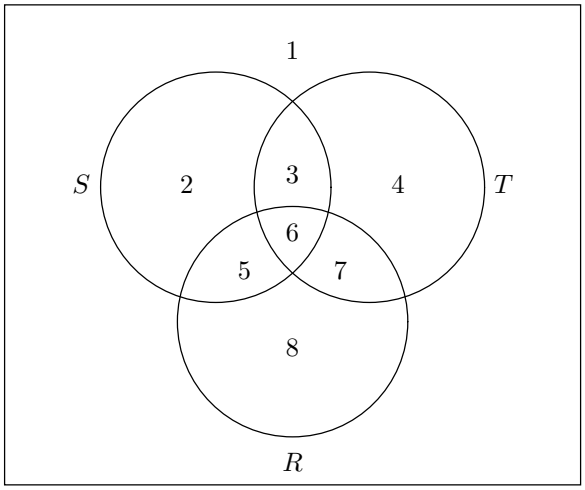

Fig. 7.2. Venn diagram showing the distributive law of intersection over union: $S \cap (T \cup R)$ consists of regions 3, 5, and 6, as does $(S \cap T) \cup (S \cap R)$.

We can use the diagram to help us keep track of the values of various subexpressions. For instance, $T \cup R$ is regions 3, 4, 5, 6, 7, and 8. Since S is regions 2, 3, 5, and 6, it follows that $S \cap (T \cup R)$ is regions 3, 5, and 6. Similarly, $S \cap T$ is regions 3 and 6, while $S \cap R$ is regions 5 and 6. It follows that $(S \cap T) \cup (S \cap R)$ is the same regions 3, 5, and 6, proving that

$$\big(S \cap (T \cup R)\big) \equiv \big((S \cap T) \cup (S \cap R)\big)$$

In general, we can prove an equivalence by considering one representative element from each region and checking that it is either in the set described by both sides of the equivalence or in neither of those sets. This method is very close to the truth-table method by which we prove algebraic laws for propositional logic in Chapter 12.

Proving Equivalences by Applying Transformations

Another way to prove two expressions equivalent is by turning one into the other using one or more of the algebraic laws we have already seen. We shall give a more formal treatment of how expressions are manipulated in Chapter 12, but for the present, let us observe that we can

1. Substitute any expression for any variable in an equivalence, provided that we substitute for all occurrences of that variable. The equivalence remains true.
2. Substitute, for a subexpression E in some equivalence, an expression F that is known to be equivalent to E. The equivalence remains true.

In addition, we can write any of the equivalences that were stated as laws and assume that equivalence is true.

✦ **Example 7.7.** We shall prove the equivalence $\big(S - (S \cup R)\big) \equiv \emptyset$. Let us start with law (g), the associative law for union and difference, which is

$$\big(S - (T \cup R)\big) \equiv \big((S - T) - R\big)$$

We substitute S for each of the two occurrences of T to get a new equivalence:

$$\big(S - (S \cup R)\big) \equiv \big((S - S) - R\big)$$

By law (l), $(S - S) \equiv \emptyset$. Thus, we may substitute $\emptyset$ for $(S - S)$ above to get:

$$\big(S - (S \cup R)\big) \equiv (\emptyset - R)$$

Law (m), with R substituted for S says that $\emptyset - R \equiv \emptyset$. We may thus substitute $\emptyset$ for $\emptyset - R$ and conclude that $\big(S - (S \cup R)\big) \equiv \emptyset$. ✦

The Subset Relationship

There is a family of comparison operators among sets that is analogous to the comparisons among numbers. If S and T are sets, we say that $S \subseteq T$ if every member of S is also a member of T. We can express this in words several ways: "S is a subset of T," "T is a superset of S," "S is contained in T," or "T contains S."

Containment of sets

We say that $S \subset T$, if $S \subseteq T$, and there is at least one element of T that is not also a member of S. The $S \subset T$ relationship can be read "S is a proper subset of T," "T is a proper superset of S," "S is properly contained in T," or "T properly contains S."

Proper subset

As with "less than," we can reverse the direction of the comparison; $S \supset T$ is synonymous with $T \subset S$, and $S \supseteq T$ is synonymous with $T \subseteq S$.

✦ **Example 7.8.** The following comparisons are all true:

1. $\{1,2\} \subseteq \{1,2,3\}$
2. $\{1,2\} \subset \{1,2,3\}$
3. $\{1,2\} \subseteq \{1,2\}$

Note that a set is always a subset of itself but a set is never a proper subset of itself, so that $\{1,2\} \subset \{1,2\}$ is false. ✦

There are a number of algebraic laws involving the subset operator and the other operators that we have already seen. We list some of them here.

o) $\emptyset \subseteq S$ for any set S.

p) If $S \subseteq T$, then

i) $(S \cup T) \equiv T$,

ii) $(S \cap T) \equiv S$, and

iii) $(S - T) \equiv \emptyset$.

Proving Equivalences by Showing Containments

Two sets S and T are equal if and only if $S \subseteq T$ and $T \subseteq S$; that is, each is a subset of the other. For if every element in S is an element of T and vice versa, then S and T have exactly the same members and thus are equal. Conversely, if S and T have exactly the same members, then surely $S \subseteq T$ and $T \subseteq S$ are true. This rule is analogous to the arithmetic rule that $a = b$ if and only if both $a \leq b$ and $b \leq a$ are true.

We can show the equivalence of two expressions E and F by showing that the set denoted by each is contained in the set denoted by the other. That is, we

1. Consider an arbitrary element x in E and prove that it is also in F, and then
2. Consider an arbitrary element x in F and prove that it is also in E.

Note that both proofs are necessary in order to prove that $E \equiv F$.

	STEP	REASON
1)	x is in $S - (T \cup R)$	Given
2)	x is in S	Definition of $-$ and (1)
3)	x is not in $T \cup R$	Definition of $-$ and (1)
4)	x is not in T	Definition of $\cup$ and (3)
5)	x is not in R	Definition of $\cup$ and (3)
6)	x is in $S - T$	Definition of $-$ with (2) and (4)
7)	x is in $(S - T) - R$	Definition of $-$ with (6) and (5)

Fig. 7.3. Proof of one half of the associative law for union and difference.

✦ **Example 7.9.** Let us prove the associative law for union and difference,

$$\big(S - (T \cup R)\big) \equiv \big((S - T) - R\big)$$

We start by assuming that x is in the expression on the left. The sequence of steps is shown in Fig. 7.3. Note that in steps (4) and (5), we use the definition of union backwards. That is, (3) tells us that x is not in $T \cup R$. If x were in T, (3) would be wrong, and so we can conclude that x is not in T. Similarly, x is not in R.

	STEP	REASON
1)	x is in $(S - T) - R$	Given
2)	x is in $S - T$	Definition of $-$ and (1)
3)	x is not in R	Definition of $-$ and (1)
4)	x is in S	Definition of $-$ and (2)
5)	x is not in T	Definition of $-$ and (2)
6)	x is not in $T \cup R$	Definition of $\cup$ with (3) and (5)
7)	x is in $S - (T \cup R)$	Definition of $-$ with (4) and (6)

Fig. 7.4. Second half of the proof of the associative law for union and difference.

We are not done; we must now start by assuming that x is in $(S - T) - R$ and show that it is in $S - (T \cup R)$. The steps are shown in Fig. 7.4. ✥

✥ **Example 7.10.** As another example, let us prove part of (p), the rule that if $S \subseteq T$, then $S \cup T \equiv T$. We begin by assuming that x is in $S \cup T$. We know by the definition of union that either

1. x is in S or
2. x is in T.

In case (1), since $S \subseteq T$ is assumed, we know that x is in T. In case (2), we immediately see that x is in T. Thus, in either case x is in T, and we have completed the first half of the proof, the statement that $(S \cup T) \subseteq T$.

Now let us assume that x is in T. Then x is in $S \cup T$ by the definition of union. Thus, $T \subseteq (S \cup T)$, which is the second half of the proof. We conclude that if $S \subseteq T$ then $(S \cup T) \equiv T$. ✥

The Power Set of a Set

If S is any set, the *power set* of S is the set of subsets of S. We shall use $\mathbf{P}(S)$ to denote the power set of S, although the notation 2^S is also used.

✥ **Example 7.11.** Let $S = \{1, 2, 3\}$. Then

$$\mathbf{P}(S) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

Singleton set

That is, $\mathbf{P}(S)$ is a set with eight members; each member is itself a set. The empty set is in $\mathbf{P}(S)$, since surely $\emptyset \subseteq S$. The *singletons* — sets with one member of S, namely, $\{1\}$, $\{2\}$, and $\{3\}$ — are in $\mathbf{P}(S)$. Likewise, the three sets with two of the three members of S are in $\mathbf{P}(S)$, and S itself is a member of $\mathbf{P}(S)$.

As another example, $\mathbf{P}(\emptyset) = \{\emptyset\}$ since $\emptyset \subseteq S$, but for no set S besides the empty set is $S \subseteq \emptyset$. Note that $\{\emptyset\}$, the set containing the empty set, is not the same as the empty set. In particular, the former has a member, namely $\emptyset$, while the empty set has no members. ✥

The Size of Power Sets

If S has n members, then $\mathbf{P}(S)$ has 2^n members. In Example 7.11 we saw that a set of three members has a power set of $2^3 = 8$ members. Also, $2^0 = 1$, and we saw that the empty set, which contains zero elements, has a power set of one element.

Let $S = \{a_1, a_2, \ldots, a_n\}$, where $a_1, a_2, \ldots, a_n$ are any n elements. We shall now prove by induction on n that $\mathbf{P}(S)$ has 2^n members.

BASIS. If $n = 0$, then S is $\emptyset$. We have already observed that $\mathbf{P}(\emptyset)$ has one member. Since $2^0 = 1$, we have proved the basis.

INDUCTION. Suppose that when $S = \{a_1, a_2, \ldots, a_n\}$, $\mathbf{P}(S)$ has 2^n members. Let a_{n+1} be a new element, and let $T = S \cup \{a_{n+1}\}$, a set of $n + 1$ members. Now a subset of T either does not have or does have a_{n+1} as a member. Let us consider these two cases in turn.

1. The subsets of T that do not include a_{n+1} are also subsets of S, and therefore in $\mathbf{P}(S)$. By the inductive hypothesis, there are exactly 2^n such sets.
2. If R is a subset of T that includes a_{n+1}, let $Q = R - \{a_{n+1}\}$; that is, Q is R with a_{n+1} removed. Then Q is a subset of S. By the inductive hypothesis, there are exactly 2^n possible sets Q, and each one corresponds to a unique set R, which is $Q \cup \{a_{n+1}\}$.

We conclude that there are exactly 2×2^n, or 2^{n+1}, subsets of T, half that are subsets of S, and half that are formed from a subset of S by including a_{n+1}. Thus, the inductive step is proved; given that any set S of n elements has 2^n subsets, we have shown that any set T of $n + 1$ elements has 2^{n+1} subsets.

EXERCISES

7.3.1: In Fig. 7.2, we showed two expressions for the set of regions $\{3, 5, 6\}$. However, each of the regions can be represented by expressions involving S, T, and R and the operators union, intersection, and difference. Write two different expressions for each of the following:

a) Region 6 alone
b) Regions 2 and 4 together
c) Regions 2, 4, and 8 together

7.3.2: Use Venn diagrams to show the following algebraic laws. For each subexpression involved in the equivalence, indicate the set of regions it represents.

a) $\big(S \cup (T \cap R)\big) \equiv \big((S \cup T) \cap (S \cup R)\big)$
b) $\big((S \cup T) - R\big) \equiv \big((S - R) \cup (T - R)\big)$
c) $\big(S - (T \cup R)\big) \equiv \big((S - T) - R\big)$

7.3.3: Show each of the equivalences from Exercise 7.3.2 by showing containment of each side in the other.

7.3.4: Assuming $S \subseteq T$, prove the following by showing that each side of the equivalence is a subset of the other:

a) $(S \cap T) \equiv S$
b) $(S - T) \equiv \emptyset$

7.3.5*: Into how many regions does a Venn diagram with n sets divide the plane, assuming that no set is a subset of any other? Suppose that of the n sets there is one that is a subset of one other, but there are no other containments. Then some regions would be empty. For example, in Fig. 7.1, if $S \subseteq T$, then region 2 would be empty, because there is no element that is in S but not in T. In general, how many nonempty regions would there be?

7.3.6: Prove that if $S \subseteq T$, then $\mathbf{P}(S) \subseteq \mathbf{P}(T)$.

7.3.7*: In C we can represent a set S whose members are sets by a linked list whose elements are the headers for lists; each such list represents a set that is one of the members of S. Write a C program that takes a list of elements representing a set (i.e., a list in which all the elements are distinct) and returns the power set of the given set. What is the running time of your program? *Hint*: Use the inductive proof that there are 2^n members in the power set of a set of n elements to devise a recursive algorithm that creates the power set. If you are clever, you can use the same list as part of several sets, to avoid copying the lists that represent members of the power set, thus saving both time and space.

7.3.8: Show that

a) $\mathbf{P}(S) \cup \mathbf{P}(T) \subseteq \mathbf{P}(S \cup T)$
b) $\mathbf{P}(S \cap T) \subseteq \mathbf{P}(S) \cap \mathbf{P}(T)$

Are either (a) or (b) true if containment is replaced by equivalence?

7.3.9: What is $\mathbf{P}(\mathbf{P}(\mathbf{P}(\emptyset)))$?

7.3.10*: If we apply the power-set operator n times, starting with $\emptyset$, how many members does the resulting set have? For an example, Exercise 7.3.9 is the case $n = 3$.

7.4 List Implementation of Sets

We have already seen, in Section 6.4, how to implement the dictionary operations *insert*, *delete*, and *lookup* using a linked-list data structure. We also observed there that the expected running time of these operations is $O(n)$ if the set has n elements. This running time is not as good as the $O(\log n)$ average time taken for the dictionary operations using a balanced binary search tree data structure, as in Section 5.8. On the other hand, as we shall see in Section 7.6, a linked-list

representation of dictionaries plays an essential role in the hash-table data structure for dictionaries, which is generally faster than the binary search tree.

Union, Intersection, and Difference

The basic set operations such as union can profit from the use of linked lists as a data structure, although the proper techniques are somewhat different from what we use for the dictionary operations. In particular, sorting the lists significantly improvess the running time for union, intersection, and difference. As we saw in Section 6.4, sorting makes only a small improvement in the running time of dictionary operations.

To begin, let us see what problems arise when we represent sets by unsorted lists. In this case, we must compare each element of each set with each element of the other. Thus, to take the union, intersection, or difference of sets of size n and m requires $O(mn)$ time. For example, to create a list U that represents the union of two sets S and T, we may start by copying the list for S onto the initially empty list U. Then we examine each element of T and see whether it is in S. If not, we add the element to U. The idea is sketched in Fig. 7.5.

```
(1)  copy S to U;
(2)  for (each x in T)
(3)      if (!lookup(x, S))
(4)          insert(x, U);
```

Fig. 7.5. Pseudocode sketch of the algorithm for taking the union of sets represented by unsorted lists.

Suppose S has n members and T has m members. The operation in line (1), copying S to U, can easily be accomplished in $O(n)$ time. The lookup of line (3) takes $O(n)$ time. We only execute the insertion of line (4) if we know from line (3) that x is not in S. Since x can only appear once on the list for T, we know that x is not yet in U. Therefore, it is safe to place x at the front of U's list, and line (4) can be accomplished in $O(1)$ time. The for-loop of lines (2) through (4) is iterated m times, and its body takes time $O(n)$. Thus, the time for lines (2) to (4) is $O(mn)$, which dominates the $O(n)$ for line (1).

There are similar algorithms for intersection and difference, each taking $O(mn)$ time. We leave these algorithms for the reader to design.

Union, Intersection, and Difference Using Sorted Lists

We can perform unions, intersections, and set differences much faster when the lists representing the sets are sorted. In fact, we shall see that it pays to sort the lists before performing these operations, even if the lists are not initially sorted. For example, consider the computation of $S \cup T$, where S and T are represented by sorted lists. The process is similar to the merge algorithm of Section 2.8. One difference is that when there is a tie for smallest between the elements currently at the fronts of the two lists, we make only one copy of the element, rather than two copies as we must for merge. The other difference is that we cannot simply remove elements from the lists for S and T for the union, since we should not destroy S or

T while creating their union. Instead, we must make copies of all elements to form the union.

We assume that the types `LIST` and `CELL` are defined as before, by the macro

```
DefCell(int, CELL, LIST);
```

The function `setUnion` is shown in Fig. 7.6. It makes use of an auxiliary function $assemble(x, L, M)$ that creates a new cell at line (1), places element x in that cell at line (2), and calls `setUnion` at line (3) to take the union of the lists L and M. Then `assemble` returns a cell for x followed by the list that results from applying `setUnion` to L and M. Note that the functions `assemble` and `setUnion` are mutually recursive; each calls the other.

Function `setUnion` selects the least element from its two given sorted lists and passes to `assemble` the chosen element and the remainders of the two lists. There are six cases for `setUnion`, depending on whether or not one of its lists is `NULL`, and if not, which of the two elements at the heads of the lists precedes the other.

1. If both lists are `NULL`, `setUnion` simply returns `NULL`, ending the recursion. This case is lines (5) and (6) of Fig. 7.6.
2. If L is `NULL` and M is not, then at lines (7) and (8) we assemble the union by taking the first element from M, followed by the "union" of the `NULL` list with the tail of M. Note that, in this case, successive calls to `setUnion` result in M being copied.
3. If M is `NULL` but L is not, then at lines (9) and (10) we do the opposite, assembling the answer from the first element of L and the tail of L.
4. If the first elements of L and M are the same, then at lines (11) and (12) we assemble the answer from one copy of this element, referred to as `L->element`, and the tails of L and M.
5. If the first element of L precedes that of M, then at lines (13) and (14) we assemble the answer from this smallest element, the tail of L, and the entire list M.
6. Symmetrically, at lines (15) and (16), if M has the smallest element, then we assemble the answer from that element, the entire list L, and the tail of M.

✦ **Example 7.12.** Suppose S is $\{1, 3, 6\}$ and T is $\{5, 3\}$. The sorted lists representing these sets are $L = (1, 3, 6)$ and $M = (3, 5)$. We call $setUnion(L, M)$ to take the union. Since the first element of L, which is 1, precedes the first element of M, which is 3, case (5) applies, and we assemble the answer from 1, the tail of L, which we shall call $L_1 = (3, 6)$, and M. Function $assemble(1, L_1, M)$ calls $setUnion(L_1, M)$ at line (3), and the result is the list with first element 1 and tail equal to whatever the union is.

This call to `setUnion` is case (4), where the two leading elements are equal; both are 3 here. Thus, we assemble the union from one copy of element 3 and the tails of the lists L_1 and M. These tails are L_2, consisting of only the element 6, and M_1, consisting of only the element 5. The next call is $setUnion(L_2, M_1)$, which is an instance of case (6). We thus add 5 to the union and call $setUnion(L_2, \texttt{NULL})$. That is case (3), generating 6 for the union and calling $setUnion(\texttt{NULL}, \texttt{NULL})$. Here, we

```
     LIST setUnion(LIST L, LIST M);
     LIST assemble(int x, LIST L, LIST M);

     /* assemble produces a list whose head element is x and
        whose tail is the union of lists L and M */

     LIST assemble(int x, LIST L, LIST M)
     {
         LIST first;

(1)      first = (LIST) malloc(sizeof(struct CELL));
(2)      first->element = x;
(3)      first->next = setUnion(L, M);
(4)      return first;
     }

     /* setUnion returns a list that is the union of L and M */

     LIST setUnion(LIST L, LIST M)
     {
(5)      if (L == NULL && M == NULL)
(6)          return NULL;
(7)      else if (L == NULL) /* M cannot be NULL here */
(8)          return assemble(M->element, NULL, M->next);
(9)      else if (M == NULL) /* L cannot be NULL here */
(10)         return assemble(L->element, L->next, NULL);
         /* if we reach here, neither L nor M can be NULL */
(11)     else if (L->element == M->element)
(12)         return assemble(L->element, L->next, M->next);
(13)     else if (L->element < M->element)
(14)         return assemble(L->element, L->next, M);
(15)     else /* here, M->element < L->element */
(16)         return assemble(M->element, L, M->next);
     }
```

Fig. 7.6. Computing the union of sets represented by sorted lists.

have case (1), and the recursion ends. The result of the initial call to `setUnion` is the list $(1, 3, 5, 6)$. Figure 7.7 shows in detail the sequence of calls and returns made on this example data. ✦

Notice that the list generated by `setUnion` always comes out in sorted order. We can see why the algorithm works, by observing that whichever case applies, each element in lists L or M is either copied to the output, by becoming the first parameter in a call to `assemble`, or remains on the lists that are passed as parameters in the recursive call to `setUnion`.

Running Time of Union

If we call `setUnion` on sets with n and m elements, respectively, then the time taken

```
call setUnion((1,3,6),(3,5))
    call assemble(1,(3,6),(3,5))
        call setUnion((3,6),(3,5))
            call assemble(3,(6),(5))
                call setUnion((6),(5))
                    call assemble(5,(6),NULL)
                        call setUnion((6),NULL)
                            call assemble(6,NULL,NULL)
                                call setUnion(NULL,NULL)
                                return NULL
                            return (6)
                        return (6)
                    return (5,6)
                return (5,6)
            return (3,5,6)
        return (3,5,6)
    return (1,3,5,6)
return (1,3,5,6)
```

Fig. 7.7. Sequence of calls and returns for Example 7.12.

Big-Oh for Functions of More Than One Variable

As we pointed out in Section 6.9, the notion of big-oh, which we defined only for functions of one variable, can be applied naturally to functions of more than one variable. We say that $f(x_1, \ldots, x_k)$ is $O\big(g(x_1, \ldots, x_k)\big)$ if there are constants c and $a_1, \ldots, a_k$ such that whenever $x_i \geq a_i$ for all $i = 1, \ldots, k$, it is the case that $f(x_1, \ldots, x_k) \leq cg(x_1, \ldots, x_k)$. In particular, note that even though $m+n$ is greater than mn when one of m and n is 0 and the other is greater than 0, we can still say that $m + n$ is $O(mn)$, by choosing constants c, a_1, and a_2 all equal to 1.

by `setUnion` is $O(m+n)$. To see why, note that calls to `assemble` spend $O(1)$ time creating a cell for the output list and then calling `setUnion` on the remaining lists. Thus, the calls to `assemble` in Fig. 7.6 can be thought of as costing $O(1)$ time plus the time for a call to `setUnion` on lists the sum of whose lengths is either one less than that of L and M, or in case (4), two less. Further, all the work in `setUnion`, exclusive of the call to `assemble`, takes $O(1)$ time.

It follows that when `setUnion` is called on lists of total length $m+n$, it will result in at most $m + n$ recursive calls to `setUnion` and an equal number to `assemble`. Each takes $O(1)$ time, exclusive of the time taken by the recursive call. Thus, the time to take the union is $O(m + n)$, that is, proportional to the sum of the sizes of the sets.

This time is less than that of the $O(mn)$ time needed to take the union of sets represented by unsorted lists. In fact, if the lists for our sets are not sorted, we can sort them in $O(n \log n + m \log m)$ time, and then take the union of the sorted lists. Since $n \log n$ dominates n and $m \log m$ dominates m, we can express the total

cost of sorting and taking the union as $O(n \log n + m \log m)$. That expression can be greater than $O(mn)$, but is less whenever n and m are close in value — that is, whenever the sets are approximately the same size. Thus, it usually makes sense to sort before taking the union.

Intersection and Difference

The idea in the algorithm for union outlined in Fig. 7.6 works for intersections and differences of sets as well: when the sets are represented by sorted lists, intersection and difference are also performed in linear time. For intersection, we want to copy an element to the output only if it appears on both lists, as in case (4). If either list is `NULL`, we cannot have any elements in the intersection, and so cases (1), (2), and (3) can be replaced by a step that returns `NULL`. In case (4), we copy the element at the heads of the lists to the intersection. In cases (5) and (6), where the heads of the lists are different, the smaller cannot appear on both lists, and so we do not add anything to the intersection but pop the smaller off its list and take the intersection of the remainders.

To see why that makes sense, suppose, for example, that a is at the head of list L, that b is at the head of list M, and that $a < b$. Then a cannot appear on the sorted list M, and so we can rule out the possibility that a is on both lists. However, b can appear on list L somewhere after a, so that we may still be able to use b from M. Thus, we need to take the intersection of the tail of L with the entire list M. Conversely, if b were less than a, we would take the intersection of L with the tail of M. C code to compute the intersection is shown in Fig. 7.8. It is also necessary to modify `assemble` to call `intersection` instead of `setUnion`. We leave this change as well as a program to compute the difference of sorted lists as exercises.

```
LIST intersection(LIST L, LIST M)
{
    if (L == NULL || M == NULL)
        return NULL;
    else if (L->element == M->element)
        return assemble(L->element, L->next, M->next);
    else if (L->element < M->element)
        return intersection(L->next, M);
    else /* here, M->element < L->element */
        return intersection(L, M->next);
}
```

Fig. 7.8. Computing the intersection of sets represented by sorted lists. A new version of `assemble` is required.

EXERCISES

7.4.1: Write C programs for taking the (a) union, (b) intersection, and (c) difference of sets represented by unsorted lists.

7.4.2: Modify the program of Fig. 7.6 so that it takes the (a) intersection and (b) difference of sets represented by sorted lists.

7.4.3: The functions `assemble` and `setUnion` from Fig. 7.6 leave the lists whose union they take intact; that is, they make copies of elements rather than use the cells of the given lists themselves. Can you simplify the program by allowing it to destroy the given lists as it takes their union?

7.4.4*: Prove by induction on the sum of the lengths of the lists given as parameters that `setUnion` from Fig. 7.6 returns the union of the given lists.

Symmetric difference

7.4.5*: The *symmetric difference* of two sets S and T is $(S - T) \cup (T - S)$, that is, the elements that are in exactly one of S and T. Write a program to take the symmetric difference of two sets that are represented by sorted lists. Your program should make one pass through the lists, like Fig. 7.6, rather than call routines for union and difference.

7.4.6*: We analyzed the program of Fig. 7.6 informally by arguing that if the total of the lengths of the lists was n, there were $O(n)$ calls to `setUnion` and `assemble` and each call took $O(1)$ time plus whatever time the recursive call took. We can formalize this argument by letting $T_U(n)$ be the running time for `setUnion` and $T_A(n)$ be the running time of `assemble` on lists of total length n. Write recursive rules defining T_U and T_A in terms of each other. Substitute to eliminate T_A, and set up a conventional recurrence for T_U. Solve that recurrence. Does it show that `setUnion` takes $O(n)$ time?

7.5 Characteristic-Vector Implementation of Sets

Frequently, the sets we encounter are each subsets of some small set U, which we shall refer to as the "universal set."[1] For example, a hand of cards is a subset of the set of all 52 cards. When the sets with which we are concerned are each subsets of some small set U, there is a representation of sets that is much more efficient than the list representation discussed in the previous section. We order the elements of U in some way so that each element of U can be associated with a unique "position," which is an integer from 0 up to $n - 1$, where n is the number of elements in U.

Then, given a set S that is contained in U, we can represent S by a *characteristic vector* of 0's and 1's, such that for each element x of U, if x is in S, the position corresponding to x has a 1, and if x is not in S, then that position has a 0.

Example 7.13. Let U be the set of cards. We may order the cards any way we choose, but one reasonable scheme is to order them by suits: clubs, then diamonds, then hearts, then spades. Then, within a suit, we order the cards ace, 2, 3, ..., 10, jack, queen, king. For instance, the position of the ace of clubs is 0, the king of clubs is 12, the ace of diamonds is 13, and the jack of spades is 49. A royal flush in hearts is represented by the characteristic vector

0000000000000000000000000010000000011110000000000000

[1] Of course U cannot be a true universal set, or set of all sets, which we argued does not exist because of Russell's paradox.

The first 1, in position 26, represents the ace of hearts; and the other four 1's, in positions 35 through 38, represent the 10, jack, queen, and king of hearts.

The set of all clubs is represented by

11111111111110000000000000000000000000000000000000000

and the set of all picture cards is represented by

00000000001110000000000111000000000111000000000111

✥

Array Implementation of Sets

We can represent characteristic vectors for subsets of a universal set of n elements using Boolean arrays of the following type:

```
typedef BOOLEAN USET[n];
```

The type `BOOLEAN` is as described in Section 1.6. To insert the element corresponding to position i into a set S declared to be of type `USET`, we have only to execute

```
S[i] = TRUE;
```

Similarly, to delete the element corresponding to position i from S, we do

```
S[i] = FALSE;
```

If we want to look up this element, we have only to return the value $S[i]$, which tells us whether the ith element is present in S or not.

Note that each of the dictionary operations *insert*, *delete*, and *lookup* thus takes $O(1)$ time, when sets are represented by characteristic vectors. The only disadvantage to this technique is that all sets must be subsets of some universal set U. Moreover, the universal set must be small; otherwise, the length of the arrays becomes so large that we cannot store them conveniently. In fact, since we shall normally have to initialize all elements of the array for a set S to `TRUE` or `FALSE`, the initialization of any subset of U (even $\emptyset$) must take time proportional to the size of U. If U had a large number of elements, the time to initialize a set could dominate the cost of all other operations.

To form the union of two sets that are subsets of a common universal set of n elements, and that are represented by characteristic vectors S and T, we define another characteristic vector R to be the bitwise `OR` of the characteristic vectors S and T:

`R[i] = S[i] || T[i]`, for $0 \leq i < n$

Similarly, we can make R represent the intersection of S and T by taking the bitwise `AND` of S and T:

`R[i] = S[i] && T[i]`, for $0 \leq i < n$

Finally, we can make R represent the set difference $S - T$ as follows:

`R[i] = S[i] && !T[i], for` $0 \leq i < n$

The arrays representing characteristic vectors and the Boolean operations on them can be implemented using the bitwise operators of C if we define the type `BOOLEAN` appropriately. However, the code is machine specific, and so we shall not present any details here. A portable (but more space consuming) implementation of characteristic vectors can be accomplished with arrays of `int`'s of the appropriate size, and this is the definition of `BOOLEAN` that we have assumed.

◆ **Example 7.14.** Let us consider sets of apple varieties. Our universal set will consist of the six varieties listed in Fig. 7.9; the order of their listing indicates their position in characteristic vectors.

	VARIETY	COLOR	RIPENS
0)	Delicious	red	late
1)	Granny Smith	green	early
2)	Gravenstein	red	early
3)	Jonathan	red	early
4)	McIntosh	red	late
5)	Pippin	green	late

Fig. 7.9. Characteristics of some apple varieties.

The set of red apples is represented by the characteristic vector

$$Red = 101110$$

and the set of early apples is represented by

$$Early = 011100$$

Thus, the set of apples that are either red or early, that is, $Red \cup Early$, is represented by the characteristic vector 111110. Note that this vector has a 1 in those positions where either the vector for Red, that is, 101110, or the vector for $Early$, that is, 011100, or both, have a 1.

We can find the characteristic vector for $Red \cap Early$, the set of apples that are both red and early, by placing a 1 in those positions where both 101110 and 011100 have 1. The resulting vector is 001100, representing the set of apples {Gravenstein, Jonathan}. The set of apples that are red but not early, that is,

$$Red - Early$$

is represented by the vector 100010. The set is {Delicious, McIntosh}. ◆

Notice that the time to perform union, intersection, and difference using characteristic vectors is proportional to the length of the vectors. That length is not directly related to the size of the sets, but is equal to the size of the universal set. If the sets have a reasonable fraction of the elements in the universal set, then the time for union, intersection, and difference is proportional to the sizes of the sets involved. That is better than the $O(n \log n)$ time for sorted lists, and much better than the $O(n^2)$ time for unsorted lists. However, the drawback of characteristic

vectors is that, should the sets be much smaller than the universal set, the running time of these operations can be far greater than the sizes of the sets involved.

EXERCISES

7.5.1: Give the characteristic vectors of the following sets of cards. For convenience, you can use 0^k to represent k consecutive 0's and 1^k for k consecutive 1's.

a) The cards found in a pinochle deck

b) The red cards

c) The one-eyed jacks and the suicide king

7.5.2: Using bitwise operators, write C programs to compute the (a) union and (b) difference of two sets of cards, the first represented by words $a1$ and $a2$, the second represented by $b1$ and $b2$.

7.5.3*: Suppose we wanted to represent a bag (multiset) whose elements were contained in some small universal set U. How could we generalize the characteristic-vector method of representation to bags? Show how to perform (a) *insert*, (b) *delete*, and (c) *lookup* on bags represented this way. Note that bag *lookup*(x) returns the number of times x appears in the bag.

7.6 Hashing

The characteristic-vector representation of dictionaries, when it can be used, allows us to access directly the place where an element is represented, that is, to access the position in the array that is indexed by the value of the element. However, as we mentioned, we cannot allow our universal set to be too large, or the array will be too long to fit in the available memory of the computer. Even if it did fit, the time to initialize the array would be prohibitive. For example, suppose we wanted to store a real dictionary of words in the English language, and also suppose we were willing to ignore words longer than 10 letters. We would still have $26^{10} + 26^9 + \cdots + 26$ possible words, or over 10^{14} words. Each of these possible words would require a position in the array.

At any time, however, there are only about a million words in the English language, so that only one out of 100 million of the array entries would be `TRUE`. Perhaps we could collapse the array, so that many possible words could share an entry. For example, suppose we assigned the first 100 million possible words to the first cell of the array, the next 100 million possibilities to the second cell, and so on, up to the millionth cell. There are two problems with this arrangement:

1. It is no longer enough just to put `TRUE` in a cell, because we won't know which of the 100 million possible words are actually present in the dictionary, or if in fact more than one word in any one group is present.

2. If, for example, the first 100 million possible words include all the short words, then we would expect many more than the average number of words from the dictionary to fall into this group of possible words. Note that our arrangement has as many cells of the array as there are words in the dictionary, and so we expect the average cell to represent one word; but surely there are in English many thousands of words in the first group, which would include all the words of up to five letters, and some of the six-letter words.

To solve problem (1), we need to list, in each cell of the array, all the words in its group that are present in the dictionary. That is, the array cell becomes the header of a linked list with these words. To solve problem (2), we need to be careful how we assign potential words to groups. We must distribute elements among groups so that it is unlikely (although never impossible) that there will be many elements in a single group. Note that if there are a large number of elements in a group, and we represent groups by linked lists, then lookup will be very slow for members of a large group.

The Hash Table Data Structure

We have now evolved from the characteristic vector — a valuable data structure that is of limited use — to a structure called a *hash table* that is useful for any dictionary whatsoever, and for many other purposes as well.[2] The speed of the hash table for the dictionary operations can be made $O(1)$ on the average, independent of the size of the dictionary, and independent of the size of the universal set from which the dictionary is drawn. A picture of a hash table appears in Fig. 7.10. However, we show the list for only one group, that to which x belongs.

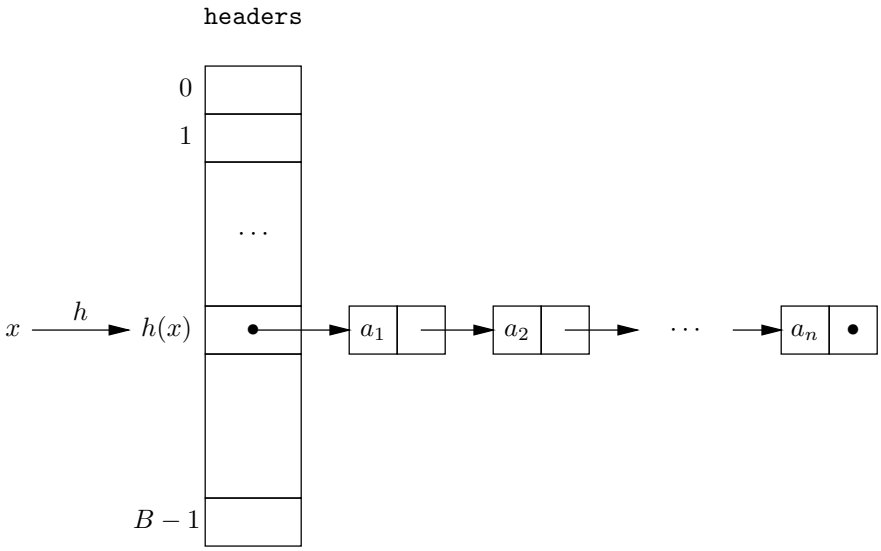

Fig. 7.10. A hash table.

[2] Although in situations where a characteristic vector is feasible, we would normally prefer that representation over any other.

Bucket

There is a *hash function* that takes an element x as argument and produces an integer value between 0 and $B-1$, where B is the number of *buckets* in the hash table. The value $h(x)$ is the bucket in which we place the element x. Thus, the buckets correspond to the "groups" of words that we talked about in the preceding informal discussion, and the hash function is used to decide to which bucket a given element belongs.

Hash function

The appropriate hash function to use depends on the type of the elements. For example,

1. If elements are integers, we could let $h(x)$ be x `%` B, that is, the remainder when x is divided by B. That number is always in the required range, 0 to $B-1$.

2. If the elements are character strings, we can take an element $x = a_1a_2\cdots a_k$, where each a_i is a character, and compute $y = a_1 + a_2 + \cdots + a_k$, since a `char` in C is a small integer. We then have an integer y that is the sum of the integer equivalents of all the characters in the string x. If we divide y by B and take the remainder, we have a bucket number in the range 0 to $B-1$.

What is important is that the hash function "hashes" the element. That is, h wildly mixes up the buckets into which elements fall, so they tend to fall in approximately equal numbers into all the buckets. This equitable distribution must occur even for a fairly regular pattern of elements, such as consecutive integers or character strings that differ in only one position.

Each bucket consists of a linked list wherein are stored all the elements of the set that are sent by the hash function to that bucket. To find an element x, we compute $h(x)$, which gives us a bucket number. If x is anywhere, it is in that bucket, so that we may search for x by running down the list for that bucket. In effect, the hash table allows us to use the (slow) list representation for sets, but, by dividing the set into B buckets, allows us to search lists that are only $1/B$ as long as the size of the set, on the average. If we make B roughly as large as the set, then buckets will average only one element, and we can find elements in an average of $O(1)$ time, just as for the characteristic-vector representation of sets.

✦ **Example 7.15.** Suppose we wish to store a set of character strings of up to 32 characters, where each string is terminated by the null character. We shall use the hash function outlined in (2) above, with $B = 5$, that is, a five-bucket hash table. To compute the hash value of each element, we sum the integer values of the characters in each string, up to but not including the null character. The following declarations give us the desired types.

```
(1)     #define B 5
(2)     typedef char ETYPE[32];
(3)     DefCell(ETYPE, CELL, LIST);
(4)     typedef LIST HASHTABLE[B];
```

Line (1) defines the constant B to be the number of buckets, 5. Line (2) defines the type `ETYPE` to be arrays of 32 characters. Line (3) is our usual definition of cells and linked lists, but here the element type is `ETYPE`, that is, 32-character arrays. Line (4) defines a hashtable to be an array of B lists. If we then declare

```
HASHTABLE headers;
```

the array `headers` is of the appropriate type to contain the bucket headers for our hash table.

```
int h(ETYPE x)
{
    int i, sum;

    sum = 0;
    for (i = 0; x[i] != '\0'; i++)
        sum += x[i];
    return sum % B;
}
```

Fig. 7.11. A hash function that sums the integer equivalents of characters, assuming `ETYPE` is an array of characters.

Now, we must define the hash function h. The code for this function is shown in Fig. 7.11. The integer equivalent of each of the characters of the string x is summed in the variable `sum`. The last step computes and returns as the value of the hash function h the remainder of this sum when it is divided by the number of buckets B.

Let us consider some examples of words and the buckets into which the function h puts them. We shall enter into the hash table the seven words[3]

```
anyone lived in a pretty how town
```

In order to compute h(`anyone`), we need to understand the integer values of characters. In the usual ASCII code for characters, the lower-case letters have integer values starting at 97 for `a` (that's 1100001 in binary), 98 for `b`, and so on, up to 122 for `z`. The upper-case letters correspond to integers that are 32 less than their lower-case equivalents — that is, from 65 for `A` (1000001 in binary) to 90 for `Z`.

Thus, the integer equivalents for the characters in `anyone` are 97, 110, 121, 111, 110, 101. The sum of these is 650. When we divide by B, which is 5, we get the remainder 0. Thus, `anyone` belongs in bucket 0. The seven words of our example are assigned, by the hash function of Fig. 7.11, to the buckets indicated in Fig. 7.12.

We see that three of the seven words have been assigned to one bucket, number 0. Two words are assigned to bucket 2, and one each to buckets 1 and 4. That is somewhat less even a distribution than would be typical, but with a small number of words and buckets, we should expect anomalies. As the number of words becomes large, they will tend to distribute themselves among the five buckets approximately evenly. The hash table, after insertion of these seven words, is shown in Fig. 7.13. ✥

Implementing the Dictionary Operations by a Hash Table

To insert, delete, or look up an element x in a dictionary that is represented by a hash table, there is a simple three-step process:

---

[3] The words are from a poem of the same name by e. e. cummings. The poem doesn't get any easier to decode. The next line is "with up so floating many bells down."

WORD	SUM	BUCKET
anyone	650	0
lived	532	2
in	215	0
a	97	2
pretty	680	0
how	334	4
town	456	1

Fig. 7.12. Words, their values, and their buckets.

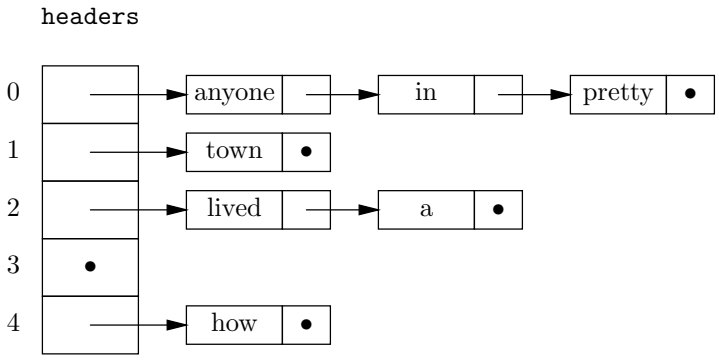

Fig. 7.13. Hash table holding seven elements.

1. Compute the proper bucket, which is $h(x)$.
2. Use the array of header pointers to find the list of elements for the bucket numbered $h(x)$.
3. Perform the operation on this list, just as if the list represented the entire set.

The algorithms in Section 6.4 can be used for the list operations after suitable modifications for the fact that elements here are character strings while in Section 6.4 elements were integers. As an example, we show the complete function for inserting an element into a hash table in Fig. 7.14. You can develop similar functions for `delete` and `lookup` as an exercise.

To understand Fig. 7.14, it helps to notice that the function `bucketInsert` is similar to the function `insert` from Fig. 6.5. At line (1) we test to see whether we have reached the end of the list. If we have, then we create a new cell at line (2). However, at line (3), instead of storing an integer into the newly created cell, we use the function `strcpy` from the standard header file `string.h` to copy the string x into the element field of the cell.

Also, at line (5), to test if x has not yet been found on the list, we use function `strcmp` from `string.h`. That function returns 0 if and only if x and the element in the current cell are equal. Thus, we continue down the list as long as the value of the comparison is nonzero, that is, as long as the current element is not x.

The function `insert` here consists of a single line, in which we call `buck-`

```
     #include <string.h>

     void bucketInsert(ETYPE x, LIST *pL)
     {
(1)      if ((*pL) == NULL) {
(2)          (*pL) = (LIST) malloc(sizeof(struct CELL));
(3)          strcpy((*pL)->element, x);
(4)          (*pL)->next = NULL;
         }
(5)      else if (strcmp((*pL)->element, x)) /* x and element
                     are different */
(6)          bucketInsert(x, &((*pL)->next));
     }

     void insert(ETYPE x, HASHTABLE H)
     {
(7)      bucketInsert(x, &(H[h(x)]));
     }
```

Fig. 7.14. Inserting an element into a hash table.

`etInsert` after first finding the element of the array that is the header for the appropriate bucket, $h(x)$. We assume that the hash function h is defined elsewhere. Also recall that the type `HASHTABLE` means that `H` is an array of pointers to cells (i.e., an array of lists).

◆ **Example 7.16.** Suppose we wish to delete the element `in` from the hash table of Fig. 7.13, assuming the hash function described in Example 7.15. The delete operation is carried out essentially like the function `insert` of Fig. 7.14. We compute h(`in`), which is 0. We thus go to the header for bucket number 0. The second cell on the list for this bucket holds `in`, and we delete that cell. The detailed C program is left as an exercise. ◆

Running Time of Hash Table Operations

As we can see by examining Fig. 7.14, the time taken by the function *insert* to find the header of the appropriate bucket is $O(1)$, assuming that the time to compute $h(x)$ is a constant independent of the number of elements stored in the hash table.[4] To this constant we must add on the average an additional $O(n/B)$ time, if n is the number of elements in the hash table and B is the number of buckets. The reason is that *bucketInsert* will take time proportional to the length of the list, and that length, on the average, must be the total number of elements divided by the number of buckets, or n/B.

An interesting consequence is that if we make B approximately equal to the number of elements in the set — that is, n and B are close — then n/B is about 1

[4] That would be the case for the hash function of Fig. 7.11, or most other hash functions encountered in practice. The time for computing the bucket number may depend on the type of the element — longer strings may require the summation of more integers, for example — but the time is not dependent on the number of elements stored.

and the dictionary operations on a hash table take $O(1)$ time each, on the average, just as when we use a characteristic-vector representation. If we try to do better by making B much larger than n, so that most buckets are empty, it still takes us $O(1)$ time to find the bucket header, and so the running time does not improve significantly once B becomes larger than n.

Restructuring hash tables

We must also consider that in some circumstances it may not be possible to keep B close to n all the time. If the set is growing rapidly, then n increases while B remains fixed, so that ultimately n/B becomes large. It is possible to restructure the hash table by picking a larger value for B and then inserting each of the elements into the new table. It takes $O(n)$ time to do so, but that time is no greater than the $O(n)$ time that must be spent inserting the n elements into the hash table in the first place. (Note that n insertions, at $O(1)$ average time per insertion, require $O(n)$ time in all.)

EXERCISES

7.6.1: Continue filling the hash table of Fig. 7.13 with the words `with up so floating many bells down`.

7.6.2*: Comment on how effective the following hash functions would be at dividing typical sets of English words into buckets of roughly equal size:

a) Use $B = 10$, and let $h(x)$ be the remainder when the length of the word x is divided by 10.

b) Use $B = 128$, and let $h(x)$ be the integer value of the last character of x.

c) Use $B = 10$. Take the sum of the values of the characters in x. Square the result, and take the remainder when divided by 10.

7.6.3: Write C programs for performing (a) *delete* and (b) *lookup* in a hash table, using the same assumptions as for the code in Fig. 7.14.

7.7 Relations and Functions

While we have generally assumed that elements of sets are atomic, in practice it is often useful to give elements some structure. For example, in the previous section we talked about elements that were character strings of length 32. Another important structure for elements is fixed-length lists, which are similar to C structures. Lists used as set elements will be called *tuples*, and each list element is called a *component* of the tuple.

Tuple, component

The number of components a tuple has is called its *arity.* For example, (a, b) is a tuple of arity 2; its first component is a and its second component is b. A tuple of arity k is also called a *k-tuple*.

Arity: unary, binary

A set of elements, each of which is a tuple of the same arity, — say, k — is called a *relation.* The arity of this relation is k. A tuple or relation of arity 1 is *unary.* If the arity is 2, it is *binary,* and in general, if the arity is k, then the tuple or relation is *k-ary.*

✦ **Example 7.17.** The relation $R = \{(1,2),\ (1,3),\ (2,2)\}$ is a relation of arity 2, or a binary relation. Its members are $(1,2)$, $(1,3)$, and $(2,2)$, each of which is a tuple of arity 2. ✦

In this section, we shall consider primarily binary relations. There are also many important applications of nonbinary relations, especially in representing and manipulating tabular data (as in relational databases). We shall discuss this topic extensively in Chapter 8.

Cartesian Products

Before studying binary relations formally, we need to define another operator on sets. Let A and B be two sets. Then the *product* of A and B, denoted $A \times B$, is defined as the set of pairs in which the first component is chosen from A and the second component from B. That is,

$$A \times B = \{(a,b) \mid a \in A \text{ and } b \in B\}$$

The product is sometimes called the *Cartesian* product, after the French mathematician René Descartes.

✦ **Example 7.18.** Recall that $\mathbf{Z}$ is the conventional symbol for the set of all integers. Thus, $\mathbf{Z} \times \mathbf{Z}$ stands for the set of pairs of integers.

As another example, if A is the two-element set $\{1,2\}$ and B is the three-element set $\{a,b,c\}$, then $A \times B$ is the six-element set

$$\{(1,a),\ (1,b),\ (1,c),\ (2,a),\ (2,b),\ (2,c)\}$$

Note that the product of sets is aptly named, because if A and B are finite sets, then the number of elements in $A \times B$ is the product of the number of elements in A and the number of elements in B. ✦

Cartesian Product of More Than Two Sets

Unlike the arithmetic product, the Cartesian product does not have the common properties of commutativity or associativity. It is easy to find examples where

$$A \times B \neq B \times A$$

disproving commutativity. The associative law does not even make sense, because $(A \times B) \times C$ would have as members pairs like $\big((a,b),c\big)$, while members of $A \times (B \times C)$ would be pairs of the form $\big(a,(b,c)\big)$.

k-way product of sets

Since we shall need on several occasions to talk about sets of tuples with more than two components, we need to extend the product notation to a k-way product. We let $A_1 \times A_2 \times \cdots \times A_k$ stand for the *product* of sets $A_1, A_2, \ldots, A_k$, that is, the set of k-tuples $(a_1, a_2, \ldots, a_k)$ such that $a_1 \in A_1$, $a_2 \in A_2, \ldots$, and $a_k \in A_k$.

✦ **Example 7.19.** $\mathbf{Z} \times \mathbf{Z} \times \mathbf{Z}$ represents the set of triples of integers (i, j, k) — it contains, for example, the triple $(1, 2, 3)$. This three-way product should not be confused with $(\mathbf{Z} \times \mathbf{Z}) \times \mathbf{Z}$, which represents pairs like $\big((1, 2), 3\big)$, or $\mathbf{Z} \times (\mathbf{Z} \times \mathbf{Z})$, which represents pairs like $\big(1, (2, 3)\big)$.

On the other hand, note that all three product expressions can be represented by structures consisting of three integer fields. The distinction is in how one interprets the structures of this type. Thus, we often feel free to "confuse" parenthesized and unparenthesized product expressions. Similarly, the three C type declarations

```
struct {int f1; int f2; int f3;};
struct {struct {int f1; int f2;}; int f3;};
struct {int f1; struct {int f2; int f3;};};
```

would all be stored in a similar way — only the notation for accessing fields would differ. ✦

Binary Relations

A binary relation R is a set of pairs that is a subset of the product of two sets A and B. If a relation R is a subset of $A \times B$, we say that R is *from A to B*. We call A the *domain* and B the *range* of the relation. If B is the same as A, we say that R is a relation *on A* or "on the domain" A.

Domain, range

✦ **Example 7.20.** The arithmetic relation $<$ on integers is a subset of $\mathbf{Z} \times \mathbf{Z}$, consisting of those pairs (a, b) such that a is less than b. Thus, the symbol $<$ may be regarded as the name of the set

$$\{(a, b) \mid (a, b) \in \mathbf{Z} \times \mathbf{Z}, \text{ and } a \text{ is less than } b\}$$

We then use $a < b$ as a shorthand for "$(a, b) \in <$," or "(a, b) is a member of the relation $<$." The other arithmetic relations on integers, such as $>$ or $\leq$, can be defined similarly, as can the arithmetic comparisons on real numbers.

For another example, consider the relation R from Example 7.17. Its domain and range are uncertain. We know that 1 and 2 must be in the domain, because these integers appear as first components of tuples in R. Similarly, we know that the range of R must include 2 and 3. However, we could regard R as a relation from $\{1, 2\}$ to $\{2, 3\}$, or as a relation from $\mathbf{Z}$ to $\mathbf{Z}$, as two examples among an infinity of choices. ✦

Infix Notation for Relations

As we suggested in Example 7.20, it is common to use an infix notation for binary relations, so that a relation like $<$, which is really a set of pairs, can be written between the components of pairs in the relation. That is why we commonly find expressions like $1 < 2$ and $4 \geq 4$, rather than the more pedantic "$(1, 2) \in <$" or "$(4, 4) \in \geq$."

✦ **Example 7.21.** The same notation can be used for arbitrary binary relations. For instance, the relation R from Example 7.17 can be written as the three "facts" $1R2$, $1R3$, and $2R2$. ✦

Declared and Current Domains and Ranges

The second part of Example 7.20 underscores the point that we cannot tell the domain or range of a relation just by looking at it. Surely the set of elements appearing in first components must be a subset of the domain, and the set of elements that occur in second components must be a subset of the range. However, there could be other elements in the domain or range.

The difference is not important when a relation does not change. However, we shall see in Sections 7.8 and 7.9, and also in Chapter 8, that relations whose values change are very important. For example, we might speak of a relation whose domain is the students in a class, and whose range is integers, representing total scores on homework. Before the class starts, there are no pairs in this relation. After the first assignment is graded, there is one pair for each student. As time goes on, students drop the class or are added to the class, and scores increase.

We could define the domain of this relation to be the set of all students registered at the university and the range to be the set of integers. Surely, at any time, the value of the relation is a subset of the Cartesian product of these two sets. On the other hand, at any time, the relation has a *current domain* and a *current range*, which are the sets of elements appearing in first and second components, respectively, of the pairs in the relation. When we need to make a distinction, we can call the domain and range the *declared* domain and range. The current domain and range will always be a subset of the declared domain and range, respectively.

Current domain, range

Declared domain, range

Graphs for Binary Relations

We can represent a relation R whose domain is A and whose range is B by a graph. We draw a node for each element that is in A and/or B. If aRb, then we draw an arrow ("arc") from a to b. (General graphs are discussed in more detail in Chapter 9.)

◆ **Example 7.22.** The graph for the relation R from Example 7.17 is shown in Fig. 7.15. It has nodes for the elements 1, 2, and 3. Since $1R2$, there is an arc from node 1 to node 2. Since $1R3$, there is an arc from 1 to 3, and since $2R2$, there is an arc from node 2 to itself. There are no other arcs, because there are no other pairs in R. ◆

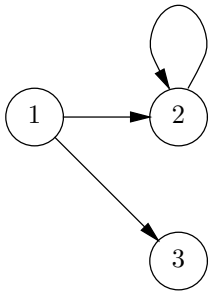

Fig. 7.15. Graph for the relation $\{(1,2),\ (1,3),\ (2,2)\}$.

Functions

Suppose a relation R, from domain A to range B, has the property that for every member a of A there is at most one element b in B such that aRb. Then R is said to be a *partial function from domain A to range B.*

Partial function

If for every member a of A there is exactly one element b in B such that aRb, then R is said to be a *total function* from A to B. The difference between a partial function and a total function is that a partial function can be undefined on some elements of its domain; for example, for some a in A, there may be no b in B such that aRb. We shall use the term "function" to refer to the more general notion of a partial function, but whenever the distinction between a partial function and a total function is important, we shall use the word "partial."

Total function

There a common notation used to describe functions. We often write $R(a) = b$ if b is the unique element such that aRb.

✦ **Example 7.23.** Let S be the total function from $\mathbf{Z}$ to $\mathbf{Z}$ given by

$$\{(a, b) \mid b = a^2\}$$

that is, the set of pairs of integers whose second component is the square of the first component. Then S has such members as $(3, 9)$, $(-4, 16)$, and $(0, 0)$. We can express the fact that S is the squaring function by writing $S(3) = 9$, $S(-4) = 16$, and $S(0) = 0$. ✦

Notice that the set-theoretic notion of a function is not much different from the notion of a function that we encountered in C. That is, suppose `s` is a C function declared as

```
int s(int a)
{
    return a*a;
}
```

that takes an integer and returns its square. We usually think of `s(a)` as being the same function as $S(a)$, although the former is a way to compute squares and the latter only defines the operation of squaring abstractly. Also note that in practice `s(a)` is always a partial function, since there are many values of `a` for which `s(a)` will not return an integer because of the finiteness of computer arithmetic.

C has functions that take more than one parameter. A C function `f` that takes two integer parameters `a` and `b`, returning an integer, is a function from $\mathbf{Z} \times \mathbf{Z}$ to $\mathbf{Z}$. Similarly, if the two parameters are of types that make them belong to sets A and B, respectively, and `f` returns a member of type C, then `f` is a function from $A \times B$ to C. More generally, if `f` takes k parameters — say, from sets $A_1, A_2, \ldots, A_k$, respectively — and returns a member of set B, then we say that `f` is a function from $A_1 \times A_2 \times \cdots \times A_k$ to B.

For example, we can regard the function `lookup(x,L)` from Section 6.4 as a function from $\mathbf{Z} \times L$ to $\{\texttt{TRUE}, \texttt{FALSE}\}$. Here, L is the set of linked lists of integers.

Formally, a function from domain $A_1 \times \cdots \times A_k$ to range B is a set of pairs of the form $\big((a_1, \ldots, a_k), b\big)$, where each a_i is in set A_i and b is in B. Notice that the first element of the pair is itself a k-tuple. For example, the function `lookup(x,L)` discussed above can be thought of as the set of pairs $\big((x, L), t\big)$, where x is an

The Many Notations for Functions

A function F from, say, $A \times B$ to C is technically a subset of $(A \times B) \times C$. A typical pair in the function F would thus have the form $\big((a, b), c\big)$, where a, b, and c are members of A, B, and C, respectively. Using the special notation for functions, we can write $F(a, b) = c$.

We can also view F as a relation from $A \times B$ to C, since every function is a relation. Using the infix notation for relations, the fact that $((a, b), c)$ is in F could also be written $(a, b)Fc$.

When we extend the Cartesian product to more than two sets, we may wish to remove parentheses from product expressions. Thus, we might identify $(A \times B) \times C$ with the technically inequivalent expression $A \times B \times C$. In that case, a typical member of F could be written (a, b, c). If we stored F as a set of such triples, we would have to remember that the first two components together make up the domain element and the third component is the range element.

integer, L is a list of integers, and t is either `TRUE` or `FALSE`, depending on whether x is or is not on the list L. We can think of a function, whether written in C or as formally defined in set theory, as a box that takes a value from the domain set and produces a value from the range set, as suggested in Fig. 7.16 for the function `lookup`.

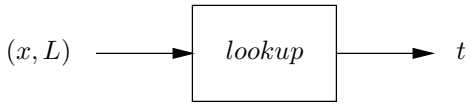

Fig. 7.16. A function associates elements from the domain with unique elements from the range.

One-to-One Correspondences

Let F be a partial function from domain A to range B with the following properties:

1. For every element a in A, there is an element b in B such that $F(a) = b$.
2. For every b in B, there is some a in A such that $F(a) = b$.
3. For no b in B are there two elements a_1 and a_2 in A such that $F(a_1)$ and $F(a_2)$ are both b.

Then F is said to be a *one-to-one correspondence* from A to B. The term *bijection* is also used for a one-to-one correspondence.

Property (1) says that F is a total function from A to B. Property (2) is the condition of being *onto*: F is a total function from A onto B. Some mathematicians use the term *surjection* for a total function that is onto.

Surjection

Properties (2) and (3) together say that F behaves like a total function from B to A. A total function with property (3) is sometimes called an *injection*.

Injection

A one-to-one correspondence is basically a total function in both directions, but it is important to observe that whether F is a one-to-one correspondence depends not only on the pairs in F, but on the declared domain and range. For example, we could take any one-to-one correspondence from A to B and change the domain by

adding to A some new element e not mentioned in F. F would not be a one-to-one correspondence from $A \cup \{e\}$ to B.

◆ **Example 7.24.** The squaring function S from $\mathbf{Z}$ to $\mathbf{Z}$ of Example 7.23 is not a one-to-one correspondence. It does satisfy property (1), since for every integer i there is some integer, namely, i^2, such that $S(i) = i^2$. However, it fails to satisfy (2), since there are some b's in $\mathbf{Z}$ — in particular all the negative integers — that are not $S(a)$ for any a. S also fails to satisfy (3), since there are many examples of two distinct a's for which $S(a)$ equals the same b. For instance, $S(3) = 9$ and $S(-3) = 9$.

For an example of a one-to-one correspondence, consider the total function P from $\mathbf{Z}$ to $\mathbf{Z}$ defined by $P(a) = a+1$. That is, P adds 1 to any integer. For instance, $P(5) = 6$, and $P(-5) = -4$. An alternative way to look at the situation is that P consists of the tuples

$$\{ \ldots, (-2,-1), (-1,0), (0,1), (1,2), \ldots \}$$

or that it has the graph of Fig. 7.17.

We claim that P is a one-to-one correspondence from integers to integers. First, it is a partial function, since when we add 1 to an integer a we get the unique integer $a+1$. It satisfies property (1), since for every integer a, there is some integer $a+1$, which is $P(a)$. Property (2) is also satisfied, since for every integer b there is some integer, namely, $b-1$, such that $P(b-1) = b$. Finally, property (3) is satisfied, because for an integer b there cannot be two distinct integers such that when you add 1 to either, the result is b. ◆

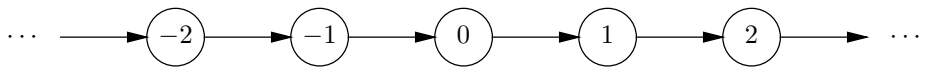

Fig. 7.17. Graph for the relation that is the function $P(a) = a + 1$.

A one-to-one correspondence from A to B is a way of establishing a unique association between the elements of A and B. For example, if we clap our hands together, the left and right thumbs touch, the left and right index fingers touch, and so on. We can think of this association between the set of fingers on the left hand and the fingers on the right hand as a one-to-one correspondence F, defined by $F(\text{“left thumb”}) = \text{“right thumb”}$, $F(\text{“left index finger”}) = \text{“right index finger”}$, and so on. We could similarly think of the association as the inverse function, from the right hand to the left. In general, a one-to-one correspondence from A to B can be inverted by switching the order of components in its pairs, to become a one-to-one correspondence from B to A.

A consequence of the existence of this one-to-one correspondence between hands is that the number of fingers on each hand is the same. That seems a natural and intuitive notion; two sets have the same number of elements exactly when there is a one-to-one correspondence from one set to the other. However, we shall see in Section 7.11 that when sets are infinite, there are some surprising conclusions we are forced to draw from this definition of “same number of elements.”

EXERCISES

7.7.1: Give an example of sets A and B for which $A \times B$ is not the same as $B \times A$.

7.7.2: Let R be the relation defined by aRb, bRc, cRd, aRc, and bRd.

a) Draw the graph of R.
b) Is R a function?
c) Name two possible domains for R; name two possible ranges.
d) What is the smallest set S such that R is a relation on S (i.e., the domain and the range can both be S)?

7.7.3: Let T be a tree and let S be the set of nodes of T. Let R be the "child-parent" relation; that is, cRp if and only if c is a child of p. Answer the following, and justify your answers:

a) Is R a partial function, no matter what tree T is?
b) Is R a total function from S to S no matter what T is?
c) Can R ever be a one-to-one correspondence (i.e., for some tree T)?
d) What does the graph for R look like?

7.7.4: Let R be the relation on the set of integers $\{1, 2, \ldots, 10\}$ defined by aRb if a and b are distinct and have a common divisor other than 1. For example, $2R4$ and $6R9$, but not $2R3$.

a) Draw the graph for R.
b) Is R a function? Why or why not?

7.7.5*: Although we observed that $S = (A \times B) \times C$ and $T = A \times (B \times C)$ are not the same set, we can show that they are "essentially the same" by exhibiting a natural one-to-one correspondence between them. For each $\big((a, b), c\big)$ in S, let

$$F\Big(\big((a, b), c\big)\Big) = \big(a, (b, c)\big)$$

Show that F is a one-to-one correspondence from S to T.

7.7.6: What do the three statements $F(10) = 20$, $10F20$, and $(10, 20) \in F$ have in common?

Inverse relation

7.7.7*: The *inverse* of a relation R is the set of pairs (b, a) such that (a, b) is in R.

a) Explain how to get the graph of the inverse of R from the graph for R.
b) If R is a total function, is the inverse of R necessarily a function? What if R is a one-to-one correspondence?

7.7.8: Show that a relation is a one-to-one correspondence if and only if it is a total function and its inverse is also a total function.

7.8 Implementing Functions as Data

In a programming language, functions are usually implemented by code, but when their domain is small, they can be implemented using techniques quite similar to the ones we used for sets. In this section we shall discuss the use of linked lists, characteristic vectors, and hash tables to implement finite functions.

Functions as Programs and Functions as Data

While we drew a strong analogy in Section 7.7 between the abstract notion of a function and a function as implemented in C, we should also be aware of an important difference. If F is a C function and x a member of its domain set, then F tells us how to compute the value $F(x)$. The same program works for any value x.

However, when we represent functions as data, we require, first of all, that the function consists of a finite set of pairs. Second, it is normal that the pairs are essentially unpredictable. That is, there is no convenient way to compute, given x, the value $F(x)$. The best we can do is create a table giving each of the pairs

$$(a_1, b_1),\ (a_2, b_2), \ldots, (a_n, b_n)$$

such that $F(a_i) = b_i$. Such a function is effectively data, rather than a program, even though we could, in principle, create a program that could store the table as part of itself and from the internal table look up $F(x)$, given x. However, a more productive approach is to store the table separately as data and look up values by a general-purpose algorithm that will work for any such function.

Operations on Functions

The operations we most commonly perform on functions are similar to those for dictionaries. Suppose F is a function from domain set A to range set B. Then we may

1. *Insert* a new pair (a, b), such that $F(a) = b$. The only nuance is that, since F must be a function, should there already be a pair (a, c) for any c, this pair must be replaced by (a, b).
2. *Delete* the value associated with $F(a)$. Here, we need to give only the domain value a. If there is any b such that $F(a) = b$, the pair (a, b) is removed from the set. If there is no such pair, then no change is made.
3. *Lookup* the value associated with $F(a)$; that is, given domain value a, we return the value b such that $F(a) = b$. If there is no such pair (a, b) in the set, then we return some special value warning that $F(a)$ is undefined.

✦ **Example 7.25.** Suppose F consists of the pairs $\{(3, 9),\ (-4, 16),\ (0, 0)\}$; that is, $F(3) = 9$; $F(-4) = 16$, and $F(0) = 0$. Then $lookup(3)$ returns 9, and $lookup(2)$ returns a value indicating that no value is defined for $F(2)$. If F is the "squaring" function, the value -1 might be used to indicate a missing value, since -1 is not the true square of any integer.

The operation $delete(3)$ removes the pair $(3, 9)$, while $delete(2)$ has no effect. If we execute $insert(5, 25)$, the pair $(5, 25)$ is added to the set F, or equivalently, we now have $F(5) = 25$. If we execute $insert(3, 10)$, the old pair $(3, 9)$ is removed from F, and the new pair $(3, 10)$ is added to F, so that now $F(3) = 10$. ✦

Linked-List Representation of Functions

A function, being a set of pairs, can be stored in a linked list just like any other set. It is useful to define cells with three fields, one for the domain value, one for the range value, and one for a next-cell pointer. For example, we could define cells as

```
typedef struct CELL *LIST;
struct CELL {
    DTYPE domain;
    RTYPE range;
    LIST next;
};
```

where `DTYPE` is the type for domain elements and `RTYPE` is the type for range elements. Then a function is represented by a pointer to (the first cell of) a linked list.

The function in Fig. 7.18 performs the operation $insert(a, b, L)$, assuming that `DTYPE` and `RTYPE` are both arrays of 32 characters. We search for a cell containing a in the `domain` field. If found, we set its `range` field to b. If we reach the end of the list, we create a new cell and store (a, b) therein. Otherwise, we test whether the cell has domain element a. If so, we change the range value to b, and we are done. If the domain has a value other than a, then we recursively insert into the tail of the list.

```
typedef char DTYPE[32], RTYPE[32];

void insert(DTYPE a, RTYPE b, LIST *pL)
{
    if ((*pL) == NULL) {/* at end of list */
        (*pL) = (LIST) malloc(sizeof(struct CELL));
        strcpy((*pL)->domain, a);
        strcpy((*pL)->range, b);
        (*pL)->next = NULL;
    }
    else if (!strcmp(a, (*pL)->domain)) /* a = domain element;
            change F(a) */
        strcpy((*pL)->range, b);
    else /* domain element is not a */
        insert(a, b, &((*pL)->next));
};
```

Fig. 7.18. Inserting a new fact into a function represented as a linked list.

If the function F has n pairs, then *insert* takes $O(n)$ time on the average. Likewise, the analogous *delete* and *lookup* functions for a function represented as a linked list require $O(n)$ time on the average.

Vector Representation of Functions

Suppose the declared domain is the integers 0 through $DNUM - 1$, or it can be regarded as such, perhaps by being an enumeration type. Then we can use a generalization of a characteristic vector to represent functions. Define a type `FUNCT` for the characteristic vector as

```
typedef RTYPE FUNCT[DNUM];
```

Here it is essential that either the function be total or that `RTYPE` contain a value that we can interpret as "no value."

✦ **Example 7.26.** Suppose we want to store information about apples, like the harvest information of Fig. 7.9, but we now want to give the actual month of harvest, rather than the binary choice early/late. We can associate an integer constant with each element in the domain and range by defining the enumeration types

```
enum APPLES {Delicious, GrannySmith, Jonathan, McIntosh,
             Gravenstein, Pippin};
enum MONTHS {Unknown, Jan, Feb, Mar, Apr, May, Jun, Jul, Aug,
             Sep, Oct, Nov, Dec};
```

This declaration associates 0 with the identifier `Delicious`, 1 with `GrannySmith`, and so on. It also associates 0 with `Unknown`, 1 with `Jan`, and so on. The identifier `Unknown` indicates that the harvest month is not known. We can now declare an array

```
int Harvest[6];
```

with the intention that the array `Harvest` represents the set of pairs in Fig. 7.19. Then the array `Harvest` appears as in Fig. 7.20, where, for example, the entry `Harvest[Delicious] = Oct` means `Harvest[0] = 10`. ✦

Apple	Harvest Month
Delicious	Oct
Granny Smith	Aug
Jonathan	Sep
McIntosh	Oct
Gravenstein	Sep
Pippin	Nov

Fig. 7.19. Harvest months of apples.

Hash-Table Representation of Functions

We can store the pairs belonging to a function in a hash table. The crucial point is that we apply the hash function only to the domain element to determine the bucket of the pair. The cells in the linked lists forming the buckets have one field for the domain element, another for the corresponding range element, and a third to link one cell to the next on the list. An example should make the technique clear.

Delicious	Oct
GrannySmith	Aug
Jonathan	Sep
McIntosh	Oct
Gravenstein	Sep
Pippin	Nov

Fig. 7.20. The array `Harvest`.

◆ **Example 7.27.** Let us use the same data about apples that appeared in Example 7.26, except now we shall use the actual names rather than integers to represent the domain. To represent the function `Harvest`, we shall use a hash table with five buckets. We shall define `APPLES` to be 32-character arrays, while `MONTHS` is an enumeration as in Example 7.26. The buckets are linked lists with field `variety` for a domain element of type `APPLES`, field `harvested` for a range element of type `int` (a month), and a link field `next` to the next element of the list.

We shall use a hash function h similar to that shown in Fig. 7.11 of Section 7.6. Of course, h is applied to domain elements only — that is, to character strings of length 32, consisting of the name of an apple variety.

Now, we can define the type `HASHTABLE` as an array of `B LIST`'s. B is the number of buckets, which we have taken to be 5. All these declarations appear in the beginning of Fig. 7.22. We may then declare a hash table `Harvest` to represent the desired function.

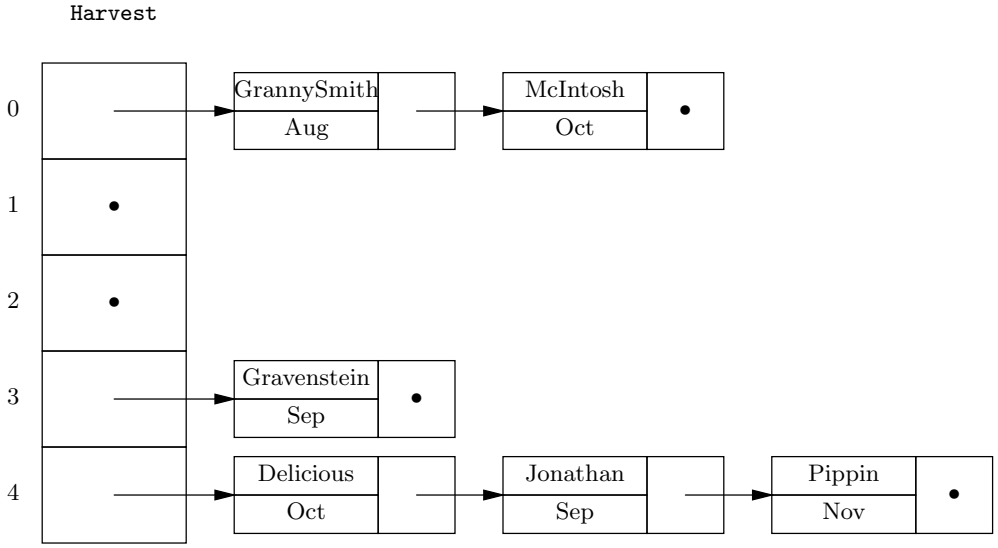

Fig. 7.21. Apples and their harvest months stored in a hash table.

After inserting the six apple varieties listed in Fig. 7.19, the arrangement of cells within buckets is shown in Fig. 7.21. For example, the word `Delicious` yields the sum 929 if we add up the integer values of the nine characters. Since the remainder

when 929 is divided by 5 is 4, the Delicious apple belongs in bucket 4. The cell for Delicious has that string in the `variety` field, the month `Oct` in the `harvested` field, and a pointer to the next cell of the bucket. ✦

```
#include <string.h>
#define B 5

typedef char APPLES[32];
enum MONTHS {Unknown, Jan, Feb, Mar, Apr, May, Jun, Jul, Aug,
            Sep, Oct, Nov, Dec};
typedef struct CELL *LIST;
struct CELL {
    APPLES variety;
    int harvested;
    LIST next;
};
typedef LIST HASHTABLE[B];

int lookupBucket(APPLES a, LIST L)
{
    if (L == NULL)
        return Unknown;
    if (!strcmp(a, L->variety)) /* found */
        return L->harvested;
    else /* a not found; examine tail */
        return lookupBucket(a, L->next);
}

int lookup(APPLES a, HASHTABLE H)
{
    return lookupBucket(a, H[h(a)]);
}
```

Fig. 7.22. Lookup for a function represented by a hash table.

Operations on Functions Represented by a Hash Table

Each of the operations *insert*, *delete*, and *lookup* start with a domain value that we hash to find a bucket. To insert the pair (a, b), we find the bucket $h(a)$ and search its list. The action is then the same as the function to insert a function pair into a list, given in Fig. 7.18.

To execute *delete*(a), we find bucket $h(a)$, search for a cell with domain value a, and delete that cell from the list, when and if it is found. The *lookup*(a) operation is executed by again hashing a and searching the bucket $h(a)$ for a cell with domain value a. If such a cell is found, the associated range value is returned.

For example, the function `lookup(a, H)` is shown in Fig. 7.22. The function `lookupBucket(a, L)` runs down the list L for a bucket and returns the value

harvested(a)

Vectors versus Hash Tables

There is a fundamental difference in the way we viewed the information about apples in Examples 7.26 and 7.27. In the characteristic-vector approach, apple varieties were a fixed set, which became an enumerated type. There is no way, while a C program is running, to change the set of apple names, and it is meaningless to perform a lookup with a name that is not part of our enumerated set.

On the other hand, when we set the same function up as a hash table, we treated the apple names as character strings, rather than members of an enumerated type. As a consequence, it is possible to modify the set of names while the program is running — say, in response to some input data about new apple varieties. It makes sense for a lookup to be performed for a variety not in the hash table, and we had to make provisions, by the addition of a "month" `Unknown`, for the possibility that we would look up a variety that was not mentioned in our table. Thus, the hash table offers increased flexibility over the characteristic vector, at some cost in speed.

that is, the month in which apple variety a is harvested. If the month is undefined, it returns the value `Unknown`.

Efficiency of Operations on Functions

The times required for the operations on functions for the three representations we have discussed here are the same as for the operations of the same names on dictionaries. That is, if the function consists of n pairs, then the linked-list representation requires $O(n)$ time per operation on the average. The characteristic-vector approach requires only $O(1)$ time per operation, but, as for dictionaries, it can be used only if the domain type is of limited size. The hash table with B buckets offers average time per operation of $O(n/B)$. If it is possible to make B close to n, then $O(1)$ time per operation, on the average, can be achieved.

EXERCISES

7.8.1: Write functions that perform (a) *delete* and (b) *lookup* on functions represented by linked lists, analogous to the *insert* function of Fig. 7.18.

7.8.2: Write functions that perform (a) *insert*, (b) *delete*, and (c) *lookup* on a function represented by a vector, that is, an array of `RTYPE`'s indexed by integers representing `DTYPE`'s.

7.8.3: Write functions that perform (a) *insert* and (b) *delete* on functions represented by hash tables, analogous to the *lookup* function of Fig. 7.22.

7.8.4: A binary search tree can also be used to represent functions as data. Define appropriate data structures for a binary search tree to hold the apple information in Fig. 7.19, and implement (a) *insert*, (b) *delete*, and (c) *lookup* using these structures.

7.8.5: Design an information retrieval system to keep track of information about at bats and hits for baseball players. Your system should accept triples of the form

```
Ruth 5 2
```

to indicate that Ruth in 5 at bats got 2 hits. The entry for Ruth should be updated appropriately. You should also be able to query the number of at bats and hits for any player. Implement your system so that the functions *insert* and *lookup* will work on any data structure as long as they use the proper subroutines and types.

7.9 Implementing Binary Relations

The implementation of binary relations differs in some details from the implementation of functions. Recall that both binary relations and functions are sets of pairs, but a function has for each domain element a at most one pair of the form (a, b) for any b. In contrast, a binary relation can have any number of range elements associated with a given domain element a.

In this section, we shall first consider the meaning of insertion, deletion, and lookup for binary relations. Then we see how the three implementations we have been using — linked lists, characteristic vectors, and hash tables — generalize to binary relations. In Chapter 8, we shall discuss implementation of relations with more than two components. Frequently, data structures for such relations are built from the structures for functions and binary relations.

Operations on Binary Relations

When we insert a pair (a, b) into a binary relation R, we do not have to concern ourselves with whether or not there already is a pair (a, c) in R, for some $c \neq b$, as we do when we insert (a, b) into a function. The reason, of course, is that there is no limit on the number of pairs in R that can have the domain value a. Thus, we shall simply insert the pair (a, b) into R as we would insert an element into any set.

Likewise, deletion of a pair (a, b) from a relation is similar to deletion of an element from a set: we look for the pair and remove it if it is present.

The *lookup* operation can be defined in several ways. For example, we could take a pair (a, b) and ask whether this pair is in R. However, if we interpret *lookup* thus, along with the *insert* and *delete* operations we just defined, a relation behaves like any dictionary. The fact that the elements being operated upon are pairs, rather than atomic, is a minor detail; it just affects the type of elements in the dictionary.

However, it is often useful to define *lookup* to take a domain element a and return all the range elements b such that (a, b) is in the binary relation R. This interpretation of *lookup* gives us an abstract data type that is somewhat different from the dictionary, and it has certain uses that are distinct from those of the dictionary ADT.

◆ **Example 7.28.** Most varieties of plums require one of several other specific varieties for pollination; without the appropriate "pollinizer," the tree cannot bear fruit. A few varieties are "self-fertile": they can serve as their own pollinizer. Figure 7.23 shows a binary relation on the set of plum varieties. A pair (a, b) in this relation means that variety b is a pollinizer for variety a.

Inserting a pair into this table corresponds to asserting that one variety is a pollinizer for another. For example, if a new variety is developed, we might enter into the relation facts about which varieties pollinize the new variety, and which it

VARIETY	POLLINIZER
Beauty	Santa Rosa
Santa Rosa	Santa Rosa
Burbank	Beauty
Burbank	Santa Rosa
Eldorado	Santa Rosa
Eldorado	Wickson
Wickson	Santa Rosa
Wickson	Beauty

Fig. 7.23. Pollinizers for certain plum varieties.

More General Operations on Relations

We may want more information than the three operations *insert*, *delete*, and *lookup* can provide when applied to the plum varieties of Example 7.28. For example, we may want to ask "What varieties does Santa Rosa pollinate?" or "Does Eldorado pollinate Beauty?" Some data structures, such as a linked list, allow us to answer questions like these as fast as we can perform the three basic dictionary operations, if for no other reason than that linked lists are not very efficient for any of these operations.

A hash table based on domain elements does not help answer questions in which we are given a range element and must find all the associated domain elements — for instance, "What varieties does Santa Rosa pollinate?" We could, of course, base the hash function on range elements, but then we could not answer easily questions like "What pollinates Burbank?" We could base the hash function on a combination of the domain and range values, but then we couldn't answer either type of query efficiently; we could only answer easily questions like "Does Eldorado pollinate Beauty?"

There are ways to get questions of all these types answered efficiently. We shall have to wait, however, until the next chapter, on the relational model, to learn the techniques.

can pollinize. Deletion of a pair corresponds to a retraction of the assertion that one variety can pollinize another.

The lookup operation we defined takes a variety a as argument, looks in the first column for all pairs having the value a, and returns the set of associated range values. That is, we ask, "What varieties can pollinize variety a?" This question seems to be the one we are most likely to ask about the information in this table, because when we plant a plum tree, we must make sure that, if it is not self-fertile, then there is a pollinizer nearby. For instance, if we invoke *lookup*(Burbank), we expect the answer {Beauty, Santa Rosa}. ✦

Linked-List Implementation of Binary Relations

We can link the pairs of a relation in a list if we like. The cells of this list consist

of a domain element, a range element, and a pointer to the next cell, just like the cells for functions. Insertion and deletion are carried out as for ordinary sets, as discussed in Section 6.4. The only nuance is that equality of set members is determined by comparing both the field holding the domain element and the field holding the range element.

Lookup is a somewhat different operation from the operations of the same name we have encountered previously. We must go down the list, looking for cells with a particular domain value a, and we must assemble a list of the associated range values. An example will show the mechanics of the *lookup* operation on linked lists.

◆ **Example 7.29.** Suppose we want to implement the plum relation of Example 7.28 as a linked list. We could define the type `PVARIETY` as a character string of length 32; and cells, whose type we shall call `RCELL` (relation cell), can be defined by a structure:

```
typedef char PVARIETY[32];
typedef struct RCELL *RLIST;
struct RCELL {
    PVARIETY variety;
    PVARIETY pollinizer;
    RLIST next;
};
```

We also need a cell containing one plum variety and a pointer to the next cell, in order to build a list of the pollinizers of a given variety, and thus to answer a *lookup* query. This type we shall call `PCELL`, and we define

```
typedef struct PCELL *PLIST;
struct PCELL {
    PVARIETY pollinizer;
    PLIST next;
};
```

We can then define *lookup* by the function in Fig. 7.24.

The function *lookup* takes a domain element a and a pointer to the first cell of a linked list of pairs as arguments. We perform the *lookup*(a) operation on a relation R by calling `lookup(a,L)`, where L is a pointer to the first cell on the linked list representing relation R. Lines (1) and (2) are simple. If the list is empty, we return `NULL`, since surely there are no pairs with first component a in an empty list.

The hard case occurs when a is found in the domain field, called `variety`, in the first cell of the list. This case is detected at line (3) and handled by lines (4) through (7). We create at line (4) a new cell of type `PCELL`, which becomes the first cell on the list of `PCELL`'s that we shall return. Line (5) copies the associated range value into this new cell. Then at line (6) we call `lookup` recursively on the tail of the list L. The return value from this call, which is a pointer to the first cell on the resulting list (`NULL` if the list is empty), becomes the `next` field of the cell we created at line (4). Then at line (7) we return a pointer to the newly created cell, which holds one range value and is linked to cells holding other range values for domain value a, if any exist.

The last case occurs when the desired domain value a is not found in the first cell of the list L. Then we just call *lookup* on the tail of the list, at line (8), and

```
    PLIST lookup(PVARIETY a, RLIST L)
    {
        PLIST P;

(1)     if (L == NULL)
(2)         return NULL;
(3)     else if (!strcmp(L->variety, a)) /* L->variety == a */ {
(4)         P = (PLIST) malloc(sizeof(struct PCELL));
(5)         strcpy(P->pollinizer, L->pollinizer);
(6)         P->next = lookup(a, L->next);
(7)         return P;
        }
        else /* a not the domain value of current pair */
(8)         return lookup(a, L->next);
    }
```

Fig. 7.24. Lookup in a binary relation represented by a linked list.

return whatever that call returns. ✦

A Characteristic-Vector Approach

For sets and for functions, we saw that we could create an array indexed by elements of a "universal" set and place appropriate values in the array. For sets, the appropriate array values are `TRUE` and `FALSE`, and for functions they are those values that can appear in the range, plus (usually) a special value that means "none."

For binary relations, we can index an array by members of a small declared domain, just as we did for functions. However, we cannot use a single value as an array element, because a relation can have any number of range values for a given domain value. The best we can do is to use as an array element the header of a linked list that contains all the range values associated with a given domain value.

✦ **Example 7.30.** Let us redo the plum example using this organization. As was pointed out in the last section, when we use a characteristic-vector style, we must fix the set of values, in the domain at least; there is no such constraint for linked-list or hash-table representations. Thus, we must redeclare the `PVARIETY` type to be an enumerated type:

```
enum PVARIETY {Beauty, SantaRosa, Burbank, Eldorado, Wickson};
```

We can continue to use the `PCELL` type for lists of varieties, as defined in Example 7.29. Then we may define the array

```
PLIST Pollinizers[5];
```

That is, the array representing the relation of Fig. 7.23 is indexed by the varieties mentioned in that figure, and the value associated with each variety is a pointer to the first cell on its list of pollinizers. Figure 7.25 shows the pairs of Fig. 7.23 represented in this way. ✦

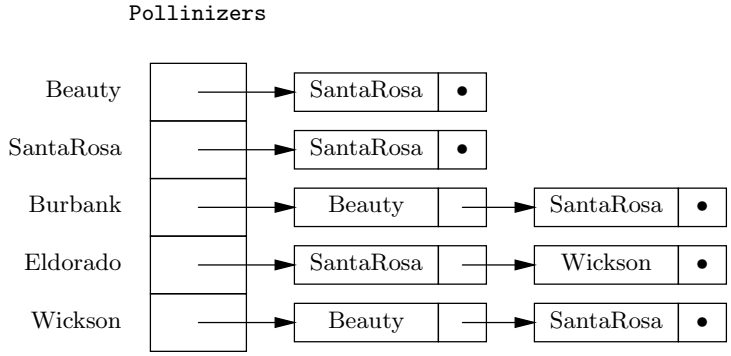

Fig. 7.25. Characteristic-vector representation of the pollinizers relation.

Insertion and deletion of pairs is performed by going to the appropriate array element and thence to the linked list. At that point, insertion in or deletion from the list is performed normally. For example, if we determined that Wickson cannot adequately pollinate Eldorado, we could execute the operation

delete(Eldorado, Wickson)

The header of the list for Eldorado is found in `Pollinizers[Eldorado]`, and from there we go down the list until we find a cell holding Wickson and delete it.

Lookup is trivial; we have only to return the pointer found in the appropriate array entry. For example, to answer the query `lookup(Burbank, Pollinizers)`, we simply return the list `Pollinizers[Burbank]`.

Hash-Table Implementation of Binary Relations

We may store a given binary relation R in a hash table, using a hash function that depends only on the first component of a pair. That is, the pair (a, b) will be placed in bucket $h(a)$, where h is the hash function. Note that this arrangement is exactly the same as that for a function; the only difference is that for a binary relation a bucket may contain more than one pair with a given value a as the first component, whereas for a function, it could never contain more than one such pair.

To insert the pair (a, b), we compute $h(a)$ and examine the bucket with that number to be sure that (a, b) is not already there. If it is not, we append (a, b) to the end of the list for that bucket. To delete (a, b), we go to the bucket $h(a)$, search for this pair, and remove it from the list if it is there.

To execute *lookup*(a), we again find the bucket $h(a)$ and go down the list for this bucket, collecting all the b's that appear in cells with first component a. The *lookup* function of Fig. 7.24, which we wrote for a linked list, applies equally well to the list that forms one bucket of a hash table.

Running Time of Operations on a Binary Relation

The performance of the three representations for binary relations is not much different from the performance of the same structures on functions or dictionaries. Consider first the list representation. While we have not written the functions for *insert* and *delete*, we should be able to visualize that these functions will run down the entire list, searching for the target pair, and stop upon finding it. On a list of

length n, this search takes $O(n)$ average time, since we must scan the entire list if the pair is not found and, on the average, half the list if it is found.

For *lookup*, an examination of Fig. 7.24 should convince us that this function takes $O(1)$ time plus a recursive call on the tail of a list. We thus make n calls if the list is of length n, for a total time of $O(n)$.

Now consider the generalized characteristic vector. The operation *lookup*(a) is easiest. We go to the array element indexed by a, and there we find our answer, a list of all the b's such that (a, b) is in the relation. We don't even have to examine the elements or copy them. Thus, *lookup* takes $O(1)$ time when characteristic vectors are used.

On the other hand, *insert* and *delete* are less simple. To insert (a, b), we can go to the array element indexed by a easily enough, but we must search the entire list to make sure that (a, b) is not already there.[5] That requires an amount of time proportional to the average length of a list, that is, to the average number of range values associated with a given domain value. We shall call this parameter m. Another way to look at m is that it is n, the total number of pairs in the relation, divided by the number of different domain values. If we assume that any list is as likely to be searched as any other, then we require $O(m)$ time on the average to perform an *insert* or a *delete*.

Finally, let us consider the hash table. If there are n pairs in our relation and B buckets, we expect there to be an average of n/B pairs per bucket. However, the parameter m must be figured in as well. If there are n/m different domain values, then at most n/m buckets can be nonempty, since the bucket for a pair is determined only by the domain value. Thus, m is a lower bound on the average size of a bucket, regardless of B. Since n/B is also a lower bound, the time to perform one of the three operations is $O\big(\max(m, n/B)\big)$.

✦ **Example 7.31.** Suppose there is a relation of 1000 pairs, distributed among 100 domain values. Then the typical domain value has 10 associated range values; that is, $m = 10$. If we use 1000 buckets — that is, $B = 1000$ — then m is greater than n/B, which is 1, and we expect the average bucket that we might actually search (because its number is $h(a)$ for some domain value a that appears in the relation) to have about 10 pairs. In fact, it will have on the average slightly more, because by coincidence, the same bucket could be $h(a_1)$ and $h(a_2)$ for different domain values a_1 and a_2. If we choose $B = 100$, then $m = n/B = 10$, and we would again expect each bucket we might search to have about 10 elements. As just mentioned, the actual number is slightly more because of coincidences, where two or more domain values hash to the same bucket. ✦

EXERCISES

7.9.1: Using the data types from Example 7.29, write a function that takes a pollinizer value b and a list of variety-pollinizer pairs, and returns a list of the varieties that are pollinized by b.

---

[5] We could insert the pair without regard for whether it is already present, but that would have both the advantages and disadvantages of the list representation discussed in Section 6.4, where we allowed duplicates.

"Dictionary Operations" on Functions and Relations

A set of pairs might be thought of as a set, as a function, or as a relation. For each of these cases, we have defined operations *insert*, *delete*, and *lookup* suitably. These operations differ in form. Most of the time, the operation takes both the domain and range element of the pair. However, sometimes only the domain element is used as an argument. The table below summarizes the differences in the use of these three operations.

	Set of Pairs	Function	Relation
Insert	Domain and Range	Domain and Range	Domain and Range
Delete	Domain and Range	Domain only	Domain and Range
Lookup	Domain and Range	Domain only	Domain only

7.9.2: Write (a) *insert* and (b) *delete* routines for variety-pollinizer pairs using the assumptions of Example 7.29.

7.9.3: Write (a) *insert*, (b) *delete*, and (c) *lookup* functions for a relation represented by the vector data structure of Example 7.30. When inserting, do not forget to check for an identical pair already in the relation.

7.9.4: Design a hash-table data structure to represent the pollinizer relation that forms the primary example of this section. Write functions for the operations *insert*, *delete*, and *lookup*.

7.9.5*: Prove that the function `lookup` of Fig. 7.24 works correctly, by showing by induction on the length of list L that `lookup` returns a list of all the elements b such that the pair (a, b) is on the list L.

7.9.6*: Design a data structure that allows $O(1)$ average time to perform each of the operations *insert*, *delete*, *lookup*, and *inverseLookup*. The latter operation takes a range element and finds the associated domain elements.

7.9.7: In this section and the previous, we defined some new abstract data types that had operations we called *insert*, *delete*, and *lookup*. However, these operations were defined slightly differently from the operations of the same name on dictionaries. Make a table for the ADT's DICTIONARY, FUNCTION (as discussed in Section 7.8), and RELATION (as discussed in this section) and indicate the possible abstract implementations and the data structures that support them. For each, indicate the running time of each operation.

7.10 Some Special Properties of Binary Relations

In this section we shall consider some of the special properties that certain useful binary relations have. We begin by defining some basic properties: transitivity, reflexivity, symmetry, and antisymmetry. These are combined to form common types of binary relations: partial orders, total orders, and equivalence relations.

Transitivity

Let R be a binary relation on the domain D. We say that the relation R is *transitive* if whenever aRb and bRc are true, aRc is also true. Figure 7.26 illustrates the transitivity property as it appears in the graph of a relation. Whenever the dotted arrows from a to b and from b to c appear in the diagram, for some particular a, b, and c, then the solid arrow from a to c must also be in the diagram. It is important to remember that transitivity, like the other properties to be defined in this section, pertains to the relation as a whole. It is not enough that the property be satisfied for three particular domain elements; it must be satisfied for all triples a, b, and c in the declared domain D.

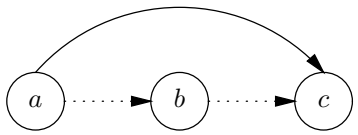

Fig. 7.26. Transitivity condition requires that if both the arcs aRb and bRc are present in the graph of a relation, then so is the arc aRc.

◆ **Example 7.32.** Consider the relation $<$ on $\mathbf{Z}$, the set of integers. That is, $<$ is the set of pairs of integers (a, b) such that a is less than b. The relation $<$ is transitive, because if $a < b$ and $b < c$, we know that $a < c$. Similarly, the relations $\leq$, $>$, and $\geq$ on integers are transitive. These four comparison relations are likewise transitive on the set of real numbers.

However, consider the relation $\neq$ on the integers (or the reals for that matter). This relation is not transitive. For instance, let a and c both be 3, and let b be 5. Then $a \neq b$ and $b \neq c$ are both true. If the relation were transitive, we would have $a \neq c$. But that says $3 \neq 3$, which is wrong. We conclude that $\neq$ is not transitive.

Transitivity of subset

For another example of a transitive relation, consider $\subseteq$, the subset relation. We might like to consider the relation as being the set of all pairs of sets (S, T) such that $S \subseteq T$, but to imagine that there is such a set would lead us to Russell's paradox again. However, suppose we have a "universal" set U. We can let $\subseteq_U$ be the set of pairs of sets

$$\{(S, T) \mid S \subseteq T \text{ and } T \subseteq U\}$$

Then $\subseteq_U$ is a relation on $\mathbf{P}(U)$, the power set of U, and we can think of $\subseteq_U$ as the subset relation.

For instance, let $U = \{1, 2\}$. Then $\subseteq_{\{1,2\}}$ consists of the nine (S, T)-pairs shown in Fig. 7.27. Thus, $\subseteq_U$ contains exactly those pairs such that the first component is a subset (not necessarily proper) of the second component and both are subsets of $\{1, 2\}$.

It is easy to check that $\subseteq_U$ is transitive, no matter what the universal set U is. If $A \subseteq B$ and $B \subseteq C$, then it must be that $A \subseteq C$. The reason is that for every x in A, we know that x is in B, because $A \subseteq B$. Since x is in B, we know that x is in C, because $B \subseteq C$. Thus, every element of A is an element of C. Therefore, $A \subseteq C$. ◆

S	T
$\emptyset$	$\emptyset$
$\emptyset$	$\{1\}$
$\emptyset$	$\{2\}$
$\emptyset$	$\{1,2\}$
$\{1\}$	$\{1\}$
$\{1\}$	$\{1,2\}$
$\{2\}$	$\{2\}$
$\{2\}$	$\{1,2\}$
$\{1,2\}$	$\{1,2\}$

Fig. 7.27. The pairs in the relation $\subseteq_{\{1,2\}}$.

Reflexivity

Some binary relations R have the property that for every element a in the declared domain, R has the pair (a,a); that is, aRa. If so, we say that R is *reflexive.* Figure 7.28 suggests that the graph of a reflexive relation has a loop on every element of its declared domain. The graph may have other arrows in addition to the loops. However, it is not sufficient that there be loops for the elements of the current domain; there must be one for each element of the declared domain.

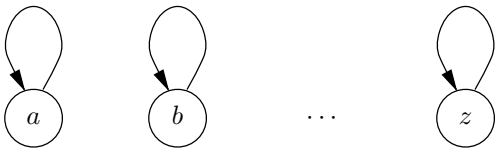

Fig. 7.28. A reflexive relation R has xRx for every x in its declared domain.

◆ **Example 7.33.** The relation $\geq$ on the reals is reflexive. For each real number a, we have $a \geq a$. Similarly, $\leq$ is reflexive, and both these relations are also reflexive on the integers. However, $<$ and $>$ are not reflexive, since $a < a$ and $a > a$ are each false for at least one value of a; in fact, they are both false for all a.

The subset relations $\subseteq_U$ defined in Example 7.32 are also reflexive, since $A \subseteq A$ for any set A. However, the similarly defined relations $\subset_U$ that contain the pair (S,T) if $T \subseteq U$ and $S \subset T$ — that is, S is a proper subset of T — are not reflexive. The reason is that $A \subset A$ is false for some A (in fact, for all A). ◆

Symmetry and Antisymmetry

Inverse relation

Let R be a binary relation. As defined in Exercise 7.7.7, the *inverse* of R is the set of pairs of R with the components reversed. That is, the inverse of R, denoted R^{-1}, is

$$\{(b, a) \mid (a, b) \in R\}$$

For example, $>$ is the inverse of $<$, since $a > b$ exactly when $b < a$. Likewise, $\geq$ is the inverse of $\leq$.

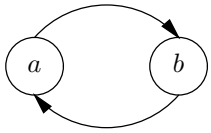

Fig. 7.29. Symmetry requires that if aRb, then bRa as well.

We say that R is *symmetric* if it is its own inverse. That is, R is symmetric if, whenever aRb, we also have bRa. Figure 7.29 suggests what symmetry looks like in the graph of a relation. Whenever the forward arc is present, the backward arc must also be present.

We say that R is *antisymmetric* if aRb and bRa are both true only when $a = b$. Note that it is not necessary that aRa be true for any particular a in an antisymmetric relation. However, an antisymmetric relation can be reflexive. Figure 7.30 shows how the antisymmetry condition relates to graphs of relations.

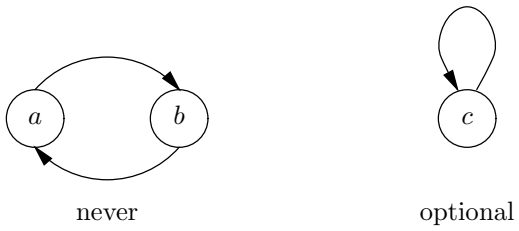

Fig. 7.30. An antisymmetric relation cannot have a cycle involving two elements, but loops on a single element are permitted.

◆ **Example 7.34.** The relation $\leq$ on integers or reals is antisymmetric, because if $a \leq b$ and $b \leq a$, it must be that $a = b$. The relation $<$ is also antisymmetric, because under no circumstances are $a < b$ and $b < a$ both true. Similarly, $\geq$ and $>$ are antisymmetric, as are the subset relations $\subseteq_U$ that we discussed in Example 7.32.

However, note that $\leq$ is not symmetric. For example, $3 \leq 5$, but $5 \leq 3$ is false. Likewise, none of the other relations mentioned in the previous paragraph is symmetric.

An example of a symmetric relation is $\neq$ on the integers. That is, if $a \neq b$, then surely $b \neq a$. ◆

Pitfalls in Property Definitions

As we have pointed out, the definition of a property is a general condition, one that applies to all elements of the domain. For example, in order for a relation R on declared domain D to be reflexive, we need to have aRa for every $a \in D$. It is not sufficient for aRa to be true for one a, nor does it make sense to say that a relation is reflexive for some elements and not others. If there is even one a in D for which aRa is false, then R is not reflexive. (Thus, reflexivity may depend on the domain, as well as on the relation R.)

Also, a condition like transitivity — "if aRb and bRc then aRc" — is of the form "if A then B." Remember that we can satisfy such a statement either by making B true or by making A false. Thus, for a given triple a, b, and c, the transitivity condition is satisfied whenever aRb is false, or whenever bRc is false, or whenever aRc is true. As an extreme example, the empty relation is transitive, symmetric, and antisymmetric, because the "if" condition is never satisfied. However, the empty relation is not reflexive, unless the declared domain is $\emptyset$.

Partial Orders and Total Orders

A *partial order* is a transitive and antisymmetric binary relation. A relation is said to be a *total order* if in addition to being transitive and antisymmetric, it makes every pair of elements in the domain *comparable*. That is to say, if R is a total order, and if a and b are any two elements in its domain, then either aRb or bRa is true. Note that every total order is reflexive, because we may let a and b be the same element, whereupon the comparability requirement tells us that aRa.

Comparable elements

✦ **Example 7.35.** The arithmetic comparisons $\leq$ and $\geq$ on integers or reals are total orders and therefore are also partial orders. Notice that for any a and b, either $a \leq b$ or $b \leq a$, but both are true exactly when $a = b$.

The comparisons $<$ and $>$ are partial orders but not total orders. While they are antisymmetric, they are not reflexive; that is, neither $a < a$ nor $a > a$ is true.

The subset relations $\subseteq_U$ and $\subset_U$ on 2^U for some universal set U are partial orders. We already observed that they are transitive and antisymmetric. These relations are not total orders, however, as long as U has at least two members, since then there are incomparable elements. For example, let $U = \{1, 2\}$. Then $\{1\}$ and $\{2\}$ are subsets of U, but neither is a subset of the other. ✦

One can view a total order R as a linear sequence of elements, as suggested in Fig. 7.31, where whenever aRb for distinct elements a and b, a appears to the left of b along the line. For example, if R is $\leq$ on the integers, then the elements along the line would be $\ldots, -2, -1, 0, 1, 2, \ldots$. If R is $\leq$ on the reals, then the points correspond to the points along the real line, as if the line were an infinite ruler; the real number x is found x units to the right of the 0 mark if x is nonnegative, and $-x$ units to the left of the zero mark if x is negative.

If R is a partial order but not a total order, we can also draw the elements of the domain in such a way that if aRb, then a is to the left of b. However, because there may be some incomparable elements, we cannot necessarily draw the elements

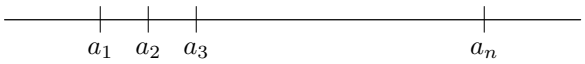

Fig. 7.31. Picture of a total order on a_1, a_2, $a_3, \ldots, a_n$.

in one line so that the relation R means "to the left."

◆ **Example 7.36.** Figure 7.32 represents the partial order $\subseteq_{\{1,2,3\}}$. We have drawn the relation as a *reduced graph,* in which we have omitted arcs that can be inferred by transitivity. That is, $S \subseteq_{\{1,2,3\}} T$ if either

Reduced graph

1. $S = T$,
2. There is an arc from S to T, or
3. There is a path of two or more arcs leading from S to T.

For example, we know that $\emptyset \subseteq_{\{1,2,3\}} \{1,3\}$, because of the path from $\emptyset$ to $\{1\}$ to $\{1,3\}$. ◆

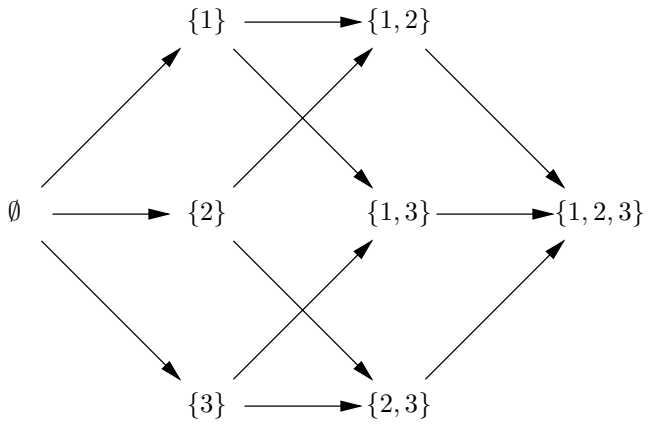

Fig. 7.32. Reduced graph for the partial order $\subseteq_{\{1,2,3\}}$.

Equivalence Relations

An *equivalence relation* is a binary relation that is reflexive, symmetric, and transitive. This kind of relation is quite different from the partial orders and total orders we have met in our previous examples. In fact, a partial order can never be an equivalence relation, except in the trivial cases that the declared domain is empty, or there is only one element a in the declared domain and the relation is $\{(a, a)\}$.

◆ **Example 7.37.** A relation like $\leq$ on integers is not an equivalence relation. Although it is transitive and reflexive, it is not symmetric. If $a \leq b$, we do not have $b \leq a$, except if $a = b$.

For an example that is an equivalence relation, let R consist of those pairs of integers (a, b) such that $a - b$ is an integer multiple of 3. For example $3R9$, since

$3 - 9 = -6 = 3 \times (-2)$. Also, $5R(-4)$, since $5 - (-4) = 9 = 3 \times 3$. However, $(1, 2)$ is not in R (or we can say "$1R2$ is false"), since $1 - 2 = -1$, which is not an integer multiple of 3. We can demonstrate that R is an equivalence relation, as follows:

1. R is reflexive, since aRa for any integer a, because $a - a$ is zero, which is a multiple of 3.
2. R is symmetric. If $a - b$ is a multiple of 3 — say, $3c$ for some integer c — then $b - a$ is $-3c$ and is therefore also an integer multiple of 3.
3. R is transitive. Suppose aRb and bRc. That is, $a - b$ is a multiple of 3, say, $3d$; and $b - c$ is a multiple of 3, say, $3e$. Then

$$a - c = (a - b) + (b - c) = 3d + 3e = 3(d + e)$$

and so $a - c$ is also a multiple of 3. Thus, aRb and bRc imply aRc, which means that R is transitive.

For another example, let S be the set of cities of the world, and let T be the relation defined by aTb if a and b are connected by roads, that is, if it is possible to drive by car from a to b. Thus, the pair (Toronto, New York) is in T, but

(Honolulu, Anchorage)

is not. We claim that T is an equivalence relation.

T is reflexive, since trivially every city is connected to itself. T is symmetric because if a is connected to b, then b is connected to a. T is transitive because if a is connected to b and b is connected to c, then a is connected to c; we can travel from a to c via b, if no shorter route exists. ✥

Equivalence Classes

Another way to view an equivalence relation is that it partitions its domain into *equivalence classes*. If R is an equivalence relation on a domain D, then we can divide D into equivalence classes so that

1. Each domain element is in exactly one equivalence class.
2. If aRb, then a and b are in the same equivalence class.
3. If aRb is false, then a and b are in different equivalence classes.

✥ **Example 7.38.** Consider the relation R of Example 7.37, where aRb when $a - b$ is a multiple of 3. One equivalence class is the set of integers that are exactly divisible by 3, that is, those that leave a remainder of 0 when divided by 3. This class is $\{\ldots, -3, 0, 3, 6, \ldots\}$. A second is the set of integers that leave a remainder of 1 when divided by 3, that is, $\{\ldots, -2, 1, 4, 7, \ldots\}$. The last class is the set of integers that leave a remainder of 2 when divided by 3. This class is $\{\ldots, -1, 2, 5, 8, \ldots\}$. The classes partition the set of integers into three disjoint sets, as suggested by Fig. 7.33.

Notice that when two integers leave the same remainder when divided by 3, then their difference is evenly divided by 3. For instance, $14 = 3 \times 4 + 2$ and $5 = 3 \times 1 + 2$. Thus, $14 - 5 = 3 \times 4 - 3 \times 1 + 2 - 2 = 3 \times 3$. We therefore know that $14R5$. On the other hand, if two integers leave different remainders when divided by

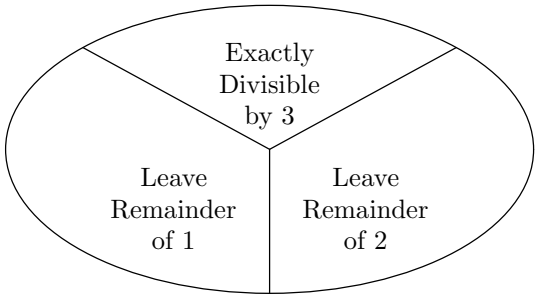

Fig. 7.33. Equivalence classes for the relation on the integers: "Difference is divisible by 3."

3, their difference surely is not evenly divisible by 3. Thus, integers from different classes, like 5 and 7, are not related by R. ❖

To construct the equivalence classes for an equivalence relation R, let $class(a)$ be the set of elements b such that aRb. For example, if our equivalence relation is the one we called R in Example 7.37, then $class(4)$ is the set of integers that leave a remainder of 1 when divided by 3; that is $class(4) = \{\ldots, -2, 1, 4, 7, \ldots\}$.

Notice that if we let a vary over each of the elements of the domain, we typically get the same class many times. In fact, when aRb, then $class(a) = class(b)$. To see why, suppose that c is in $class(a)$. Then aRc, by definition of *class*. Since we are given that aRb, by symmetry it follows that bRa. By transitivity, bRa and aRc imply bRc. But bRc says that c is in $class(b)$. Thus, every element in $class(a)$ is in $class(b)$. Since the same argument tells us that, as long as aRb, every element in $class(b)$ is also in $class(a)$, we conclude that $class(a)$ and $class(b)$ are identical.

However, if $class(a)$ is not the same as $class(b)$, then these classes can have no element in common. Suppose otherwise. Then there must be some c in both $class(a)$ and $class(b)$. By our previous assumption, we know that aRc and bRc. By symmetry, cRb. By transitivity, aRc and cRb imply aRb. But we just showed that whenever aRb is true, $class(a)$ and $class(b)$ are the same. Since we assumed these classes were not the same, we have a contradiction. Therefore, the assumed c in the intersection of $class(a)$ and $class(b)$ cannot exist.

There is one more observation we need to make: every domain element is in some equivalence class. In particular, a is always in $class(a)$, because reflexivity tells us aRa.

We can now conclude that an equivalence relation divides its domain into equivalence classes that are disjoint and that place each element into exactly one class. Example 7.38 illustrated this phenomenon.

Closures of Relations

A common operation on relations is to take a relation that does not have some property and add as few pairs as possible to create a relation that does have that property. The resulting relation is called the *closure* (for that property) of the original relation.

Transitive closure

✦ **Example 7.39.** We discussed reduced graphs in connection with Fig. 7.32. Although we were representing a transitive relation, $\subseteq_{\{1,2,3\}}$, we drew arcs corresponding to only a subset of the pairs in the relation. We can reconstruct the entire relation by applying the transitive law to infer new pairs, until no new pairs can be inferred. For example, we see that there are arcs corresponding to the pairs $(\{1\},\{1,3\})$ and $(\{1,3\},\{1,2,3\})$, and so the transitive law tells us that the pair $(\{1\},\{1,2,3\})$ must also be in the relation. Then this pair, together with the pair $(\emptyset,\{1\})$ tells us that $(\emptyset,\{1,2,3\})$ is in the relation. To these we must add the "reflexive" pairs (S,S), for each set S that is a subset of $\{1,2,3\}$. In this manner, we can reconstruct all the pairs in the relation $\subseteq_{\{1,2,3\}}$. ✦

Topological sorting

Another useful closure operation is *topological sorting,* where we take a partial order and add tuples until it becomes a total order. While the transitive closure of a binary relation is unique, there are frequently several total orders that contain a given partial order. We shall learn in Chapter 9 of a surprisingly efficient algorithm for topological sorting. For the moment, let us consider an example where topological sorting is useful.

✦ **Example 7.40.** It is common to represent a sequence of tasks that must be performed in a manufacturing process by a set of "precedences" that must be obeyed. For a simple example, you must put on your left sock before your left shoe, and your right sock before your right shoe. However, there are no other precedences that must be obeyed. We can represent these precedences by a set consisting of the two pairs $(leftsock, leftshoe)$ and $(rightsock, rightshoe)$. This set is a partial order.

We can extend this relation to six different total orders. One is the total order in which we dress the left foot first; this relation is a set that contains the ten pairs

$(leftsock, leftsock)$ $(leftsock, leftshoe)$ $(leftsock, rightsock)$ $(leftsock, rightshoe)$
$(leftshoe, leftshoe)$ $(leftshoe, rightsock)$ $(leftshoe, rightshoe)$
$(rightsock, rightsock)$ $(rightsock, rightshoe)$
$(rightshoe, rightshoe)$

We can think of this total order as the linear arrangement

$$leftsock \rightarrow leftshoe \rightarrow rightsock \rightarrow rightshoe$$

There is the analogous procedure where we dress the right foot first.

There are four other possible total orders consistent with the original partial order, where we first put on the socks and then the shoes. These are represented by the linear arrangements

$$leftsock \rightarrow rightsock \rightarrow leftshoe \rightarrow rightshoe$$
$$leftsock \rightarrow rightsock \rightarrow rightshoe \rightarrow leftshoe$$
$$rightsock \rightarrow leftsock \rightarrow leftshoe \rightarrow rightshoe$$
$$rightsock \rightarrow leftsock \rightarrow rightshoe \rightarrow leftshoe$$

✦

A third form of closure is to find the smallest equivalence relation containing a given relation. For example, a road map represents a relation consisting of pairs of cities connected by road segments having no intermediate cities. To determine the

road-connected cities, we can apply reflexivity, transitivity, and symmetry to infer those pairs of cities that are connected by some sequence of these elementary roads. This form of closure is called finding the "connected components" in a graph, and an efficient algorithm for the problem will be discussed in Chapter 9.

Connected components

EXERCISES

7.10.1: Give an example of a relation that is reflexive for one declared domain but not reflexive for another declared domain. Remember that for D to be a possible domain for a relation R, D must include every element that appears in a pair of R but it may also include more elements.

7.10.2**: How many pairs are there in the relation $\subseteq_{\{1,2,3\}}$? In general, how many pairs are there in $\subseteq_U$, if U has n elements? *Hint*: Try to guess the function from a few cases like the two-element case (Fig. 7.27) where there are 9 pairs. Then prove your guess correct by induction.

7.10.3: Consider the binary relation R on the domain of four-letter strings defined by sRt if t is formed from the string s by cycling its characters one position left. That is, *abcdRbcda*, where a, b, c, and d are individual letters. Determine whether R is (a) reflexive, (b) symmetric, (c) transitive, (d) a partial order, and/or (e) an equivalence relation. Give a brief argument why, or a counterexample, in each case.

7.10.4: Consider the domain of four-letter strings in Exercise 7.10.3. Let S be the binary relation consisting of R applied 0 or more times. Thus, *abcdSabcd*, *abcdSbcda*, *abcdScdab*, and *abcdSdabc*. Put another way, a string is related by S to any of its rotations. Answer the five questions from Exercise 7.10.3 for the relation S. Again, give justification in each case.

7.10.5*: What is wrong with the following "proof"?

(Non)Theorem: If binary relation R is symmetric and transitive, then R is reflexive.

(Non)Proof: Let x be some member of the domain of R. Pick y such that xRy. By symmetry, yRx. By transitivity, xRy and yRx imply xRx. Since x is an arbitrary member of R's domain, we have shown that xRx for every element in the domain of R, which "proves" that R is reflexive.

7.10.6: Give examples of relations with declared domain $\{1, 2, 3\}$ that are

a) Reflexive and transitive, but not symmetric
b) Reflexive and symmetric, but not transitive
c) Symmetric and transitive, but not reflexive
d) Symmetric and antisymmetric
e) Reflexive, transitive, and a total function
f) Antisymmetric and a one-to-one correspondence

7.10.7*: How many arcs are saved if we use the reduced graph for the relation $\subseteq_U$, where U has n elements, rather than the full graph?

7.10.8: Are (a) $\subseteq_U$ and (b) $\subset_U$ either partial orders or total orders when U has one element? What if U has zero elements?

7.10.9*: Show by induction on n, starting at $n = 1$, that if there is a sequence of n pairs $a_0Ra_1, a_1Ra_2, \ldots, a_{n-1}Ra_n$, and if R is a transitive relation, then a_0Ra_n. That is, show that if there is any path in the graph of a transitive relation, then there is an arc from the beginning of the path to the end.

7.10.10: Find the smallest equivalence relation containing the pairs (a, b), (a, c), (d, e), and (b, f).

7.10.11: Let R be the relation on the set of integers such that aRb if a and b are distinct and have a common divisor other than 1. Determine whether R is (a) reflexive, (b) symmetric, (c) transitive, (d) a partial order, and/or (e) an equivalence relation.

7.10.12: Repeat Exercise 7.10.11 for the relation R_T on the nodes of a particular tree T defined by aR_Tb if and only if a is an ancestor of b in tree T. However, unlike Exercise 7.10.11, your possible answers are "yes," "no," or "it depends on what tree T is."

7.10.13: Repeat Exercise 7.10.12 for relation S_T on the nodes of a particular tree T defined by aS_Tb if and only if a is to the left of b in tree T.

7.11 Infinite Sets

All of the sets that one would implement in a computer program are finite, or limited, in extent; one could not store them in a computer's memory if they were not. Many sets in mathematics, such as the integers or reals, are infinite in extent. These remarks seem intuitively clear, but what distinguishes a finite set from an infinite one?

The distinction between finite and infinite is rather surprising. A finite set is one that does not have the same number of elements as any of its proper subsets. Recall from Section 7.7 that we said we could use the existence of a one-to-one correspondence between two sets to establish that that they are *equipotent*, that is, they have the same number of members.

Equipotent sets

If we take a finite set such as $S = \{1, 2, 3, 4\}$ and any proper subset of it, such as $T = \{1, 2, 3\}$, there is no way to find a one-to-one correspondence between the two sets. For example, we could map 4 of S to 3 of T, 3 of S to 2 of T, and 2 of S to 1 of T, but then we would have no member of T to associate with 1 of S. Any other attempt to build a one-to-one correspondence from S to T must likewise fail.

Your intuition might suggest that the same should hold for any set whatsoever: how could a set have the same number of elements as a set formed by throwing away one or more of its elements? Consider the natural numbers (nonnegative integers) $\mathbf{N}$ and the proper subset of $\mathbf{N}$ formed by throwing away 0; call it $\mathbf{N} - \{0\}$, or $\{1, 2, 3, \ldots\}$. Then consider the one-to-one correspondence F from $\mathbf{N}$ to $\mathbf{N} - \{0\}$ defined by $F(0) = 1$, $F(1) = 2$, and, in general, $F(i) = i + 1$.

Surprisingly, F is a one-to-one correspondence from $\mathbf{N}$ to $\mathbf{N} - \{0\}$. For each i in $\mathbf{N}$, there is at most one j such that $F(i) = j$, so F is a function. In fact, there is exactly one such j, namely $i + 1$, so that condition (1) in the definition of one-to-one correspondence (see Section 7.7) is satisfied. For every j in $\mathbf{N} - \{0\}$ there is some i such that $F(i) = j$, namely, $i = j - 1$. Thus condition (2) in the definition of one-to-one correspondence is satisfied. Finally, there cannot be two

Infinite Hotels

To help you appreciate that there are as many numbers from 0 up as from 1 up, imagine a hotel with an infinite number of rooms, numbered 0, 1, 2, and so on; for any integer, there is a room with that integer as room number. At a certain time, there is a guest in each room. A kangaroo comes to the front desk and asks for a room. The desk clerk says, "We don't see many kangaroos around here." Wait — that's another story. Actually, the desk clerk makes room for the kangaroo as follows. He moves the guest in room 0 to room 1, the guest in room 1 to room 2, and so on. All the old guests still have a room, and now room 0 is vacant, and the kangaroo goes there. The reason this "trick" works is that there are truly the same number of rooms numbered from 1 up as are numbered from 0 up.

distinct numbers i_1 and i_2 in $\mathbf{N}$ such that $F(i_1)$ and $F(i_2)$ are both j, because then $i_1 + 1$ and $i_2 + 1$ would both be j, from which we would conclude that $i_1 = i_2$. We are forced to conclude that F is a one-to-one correspondence between $\mathbf{N}$ and its proper subset $\mathbf{N} - \{0\}$.

Formal Definition of Infinite Sets

The definition accepted by mathematicians of an *infinite set* is one that has a one-to-one correspondence between itself and at least one of its proper subsets. There are more extreme examples of how an infinite set and a proper subset can have a one-to-one correspondence between them.

◆ **Example 7.41.** The set of natural numbers and the set of even natural numbers are equipotent. Let $F(i) = 2i$. Then F is a one-to-one correspondence that maps 0 to 0, 1 to 2, 2 to 4, 3 to 6, and in general, every natural number to a unique natural number, its double.

Similarly, $\mathbf{Z}$ and $\mathbf{N}$ are the same size; that is, there are as many nonnegative and negative integers as nonnegative integers. Let $F(i) = 2i$ for all $i \geq 0$, and let $F(i) = -2i - 1$ for $i < 0$. Then 0 goes to 0, 1 to 2, -1 to 1, 2 to 4, -2 to 3, and so on. Every integer is sent to a unique nonnegative integer, with the negative integers going to odd numbers and the nonnegative integers to even numbers.

Even more surprising, the set of pairs of natural numbers is equinumerous with $\mathbf{N}$ itself. To see how the one-to-one correspondence is constructed, consider Fig. 7.34, which shows the pairs in $\mathbf{N} \times \mathbf{N}$ arranged in an infinite square. We order the pairs according to their sum, and among pairs of equal sum, by order of their first components. This order begins $(0, 0)$, $(0, 1)$, $(1, 0)$, $(0, 2)$, $(1, 1)$, $(2, 0)$, $(0, 3)$, $(1, 2)$, and so on, as suggested by Fig. 7.34.

Now, every pair has a place in the order. The reason is that for any pair (i, j), there are only a finite number of pairs with a smaller sum, and a finite number with the same sum and a smaller value of i. In fact, we can calculate the position of the pair (i, j) in the order; it is $(i + j)(i + j + 1)/2 + i$. That is, our one-to-one correspondence associates the pair (i, j) with the unique natural number $(i + j)(i + j + 1)/2 + i$.

Notice that we have to be careful how we order pairs. Had we ordered them by rows in Fig. 7.34, we would never get to the pairs on the second or higher rows,

	5	15					
↑	4	10	16				
j	3	6	11				
	2	3	7	12			
	1	1	4	8	13		
	0	0	2	5	9	14	
		0	1	2	3	4	5
			i	→			

Fig. 7.34. Ordering pairs of natural numbers.

Every Set Is Either Finite or Infinite

At first glance, it might appear that there are things that are not quite finite and not quite infinite. For example, when we talked about linked lists, we put no limit on the length of a linked list. Yet whenever a linked list is created during the execution of a program, it has a finite length. Thus, we can make the following distinctions:

1. Every linked list is finite in length; that is, it has a finite number of cells.
2. The length of a linked list may be any nonnegative integer, and the set of possible lengths of linked lists is infinite.

because there are an infinite number of pairs on each row. Similarly, ordering by columns would not work. ✦

The formal definition of infinite sets is interesting, but that definition may not meet our intuition of what infinite sets are. For example, one might expect that an infinite set was one that, for every integer n, contained at least n elements. Fortunately, this property can be proved for every set that the formal definition tells us is infinite. The proof is an example of induction.

STATEMENT $S(n)$: If I is an infinite set, then I has a subset with n elements.

BASIS. Let $n = 0$. Surely $\emptyset \subseteq I$.

INDUCTION. Assume $S(n)$ for some $n \geq 0$. We shall prove that I has a subset with $n+1$ elements. By the inductive hypothesis, I has a subset T with n elements. By the formal definition of an infinite set, there is a proper subset $J \subset I$ and a 1-1 correspondence f from I to J. Let a be an element in $I - J$; surely a exists because J is a proper subset.

Consider R, the *image* of T under f, that is, if $T = \{b_1, \ldots, b_n\}$, then $R = \{f(b_1), \ldots, f(b_n)\}$. Since f is 1-1, each of $f(b_1), \ldots, f(b_n)$ are different, so R is of size n. Since f is from I to J, each of the $f(b_k)$'s is in J; that is, $R \subseteq J$. Thus, a

Cardinality of Sets

We defined two sets S and T to be equipotent (equal in size) if there is a one-to-one correspondence from S to T. Equipotence is an equivalence relation on any set of sets, and we leave this point as an exercise. The equivalence class to which a set S belongs is said to be the *cardinality* of S. For example, the empty set belongs to an equivalence class by itself; we can identify this class with cardinality 0. The class containing the set $\{a\}$, where a is any element, is cardinality 1, the class containing the set $\{a, b\}$ is cardinality 2, and so on.

Countable set, aleph-zero

The class containing $\mathbf{N}$ is "the cardinality of the integers," usually given the name *aleph-zero,* and a set in this class is said to be *countable.* The set of real numbers belongs to another equivalence class, often called *the continuum.* There are, in fact, an infinite number of different infinite cardinalities.

cannot be in R. It follows that $R \cup \{a\}$ is a subset of I with $n+1$ elements, proving $S(n+1)$.

Countable and Uncountable Sets

From Example 7.41, we might think that all infinite sets are equipotent. We've seen that $\mathbf{Z}$, the set of integers, and $\mathbf{N}$, the set of nonnegative integers, are the same size, as are some infinite subsets of these that intuitively "seem" smaller than $\mathbf{N}$. Since we saw in Example 7.41 that the pairs of natural numbers are equinumerous with $\mathbf{N}$, it follows that the nonnegative rational numbers are equinumerous with the natural numbers, since a rational is just a pair of natural numbers, its numerator and denominator. Likewise, the (nonnegative and negative) rationals can be shown to be just as numerous as the integers, and therefore as the natural numbers.

Any set S for which there is a one-to-one correspondence from S to $\mathbf{N}$ is said to be *countable.* The use of the term "countable" makes sense, because S must have an element corresponding to 0, an element corresponding to 1, and so on, so that we can "count" the members of S. From what we just said, the integers, the rationals, the even numbers, and the set of pairs of natural numbers are all countable sets. There are many other countable sets, and we leave the discovery of the appropriate one-to-one correspondences as exercises.

However, there are infinite sets that are not countable. In particular, the real numbers are not countable. In fact, we shall show that there are more real numbers between 0 and 1 than there are natural numbers. The crux of the argument is that the real numbers between 0 and 1 can each be represented by a decimal fraction of infinite length. We shall number the positions to the right of the decimal point 0, 1, and so on. If the reals between 0 and 1 are countable, then we can number them, r_0, r_1, and so on. We can then arrange the reals in an infinite square table, as suggested by Fig. 7.35. In our hypothetical listing of the real numbers between 0 and 1, $\pi/10$ is assigned to row zero, 5/9 is assigned to row one, 5/8 is assigned to row two, 4/33 is assigned to row three, and so on.

Diagonalization

However, we can prove that Fig. 7.35 does not really represent a listing of all the reals in the range 0 to 1. Our proof is of a type known as a *diagonalization*, where we use the diagonal of the table to create a value that cannot be in the list of reals. We create a new real number r with decimal representation $.a_0a_1a_2\cdots$.

		POSITIONS							
		0	1	2	3	4	5	6	$\cdots$
	0	3	1	4	1	5	9	2	$\cdots$
REAL	1	5	5	5	5	5	5	5	$\cdots$
NUMBERS	2	6	2	5	0	0	0	0	$\cdots$
$\downarrow$	3	1	2	1	2	1	2	1	$\cdots$
	4								

Fig. 7.35. Hypothetical table of real numbers, assuming that the reals are countable.

The value of the ith digit, a_i, depends on that of the ith diagonal digit, that is, on the value found at the ith position of the ith real. If this value is 0 through 4, we let $a_i = 8$. If the value at the ith diagonal position is 5 through 9, then $a_i = 1$.

✦ **Example 7.42.** Given the part of the table suggested by Fig. 7.35, our real number r begins $.8118\cdots$. To see why, note that the value at position 0 of real 0 is 3, and so $a_0 = 8$. The value at position 1 of real 1 is 5, and so $a_1 = 1$. Continuing, the value at position 2 of real 2 is 5 and the value at position 3 of real 3 is 2, and so the next two digits are 18. ✦

We claim that r does not appear anywhere in the hypothetical list of reals, even though we supposed that all real numbers from 0 to 1 were in the list. Suppose r were r_j, the real number associated with row j. Consider the difference d between r and r_j. We know that a_j, the digit in position j of the decimal expansion of r, was specifically chosen to differ by at least 4 and at most 8 from the digit in the jth position of r_j. Thus, the contribution to d from the jth position is between $4/10^{j+1}$ and $8/10^{j+1}$.

The contribution to d from all positions after the jth is no more than $1/10^{j+1}$, since that would be the difference if one of r and r_j had all 0's there and the other had all 9's. Hence, the contribution to d from all positions j and greater is between $3/10^{j+1}$ and $9/10^{j+1}$.

Finally, in positions before the jth, r and r_j are either the same, in which case the contribution to d from the first $j-1$ positions is 0, or r and r_j differ by at least $1/10^j$. In either case, we see that d cannot be 0. Thus, r and r_j cannot be the same real number.

We conclude that r does not appear in the list of real numbers. Thus, our hypothetical one-to-one correspondence from the nonnegative integers to the reals between 0 and 1 is not one to one. We have shown there is at least one real number in that range, namely r, that is not associated with any integer.

EXERCISES

7.11.1: Show that equipotence is an equivalence relation. *Hint*: The hard part is transitivity, showing that if there is a one-to-one correspondence f from S to T, and a one-to-one correspondence g from T to R, then there is a one-to-one correspondence from S to R. This function is the *composition* of f and g, that is, the function that sends each element x in S to $g\big(f(x)\big)$ in R.

Composition of functions

7.11.2: In the ordering of pairs in Fig. 7.34, what pair is assigned number 100?

7.11.3*: Show that the following sets are countable (have a one-to-one correspondence between them and the natural numbers):

a) The set of perfect squares
b) The set of triples (i, j, k) of natural numbers
c) The set of powers of 2
d) The set of finite sets of natural numbers

7.11.4**: Show that $\mathbf{P}(\mathbf{N})$, the power set of the natural numbers, has the same cardinality as the reals — that is, there is a one-to-one correspondence from $\mathbf{P}(\mathbf{N})$ to the reals between 0 and 1. Note that this conclusion does not contradict Exercise 7.11.3(d), because here we are talking about finite and infinite sets of integers, while there we counted only finite sets. *Hint*: The following construction almost works, but needs to be fixed. Consider the characteristic vector for any set of natural numbers. This vector is an infinite sequence of 0's and 1's. For example, $\{0, 1\}$ has the characteristic vector $1100\cdots$, and the set of odd numbers has the characteristic vector $010101\cdots$. If we put a decimal point in front of a characteristic vector, we have a binary fraction between 0 and 1, which represents a real number. Thus, every set is sent to a real in the range 0 to 1, and every real number in that range can be associated with a set, by turning its binary representation into a characteristic vector. The reason this association is not a one-to-one correspondence is that certain reals have two binary representations. For example, $.11000\cdots$ and $.10111\cdots$ both represent the real number 3/4. However, these sequences as characteristic vectors represent different sets; the first is $\{0, 1\}$ and the second is the set of all integers except 1. You can modify this construction to define a one-to-one correspondence.

7.11.5**: Show that there is a one-to-one correspondence from pairs of reals in the range 0 to 1 to reals in that range. *Hint*: It is not possible to imitate the table of Fig. 7.34 directly. However, we may take a pair of reals, say, (r, s), and combine the infinite decimal fractions for r and s to make a unique new real number t. This number will not be related to r and s by any simple arithmetic expression, but from t, we can recover r and s uniquely. The reader must discover a way to construct the decimal expansion of t from the expansions of r and s.

7.11.6**: Show that whenever a set S contains subsets of all integer sizes $0, 1, \ldots$, then it is an infinite set according to the formal definition of "infinite"; that is, S has a one-to-one correspondence with one of its proper subsets.

7.12 Summary of Chapter 7

You should take away the following points from Chapter 7:

- The concept of a set is fundamental to both mathematics and computer science.
- The common operations on sets such as union, intersection, and difference can be visualized in terms of Venn diagrams.
- Algebraic laws can be used to manipulate and simplify expressions involving sets and operations on sets.

- Linked lists, characteristic vectors, and hash tables provide three basic ways to represent sets. Linked lists offer the greatest flexibility for most set operations but are not always the most efficient. Characteristic vectors provide the greatest speed for certain set operations but can be used only when the universal set is small. Hash tables are often the method of choice, providing both economy of representation and speed of access.
- (Binary) relations are sets of pairs. A function is a relation in which there is at most one tuple with a given first component.
- A one-to-one correspondence between two sets is a function that associates a unique element of the second set with each element of the first, and vice versa.
- There are a number of significant properties of binary relations: reflexivity, transitivity, symmetry, and asymmetry are among the most important.
- Partial orders, total orders, and equivalence relations are important special cases of binary relations.
- Infinite sets are those sets that have a one-to-one correspondence with one of their proper subsets.
- Some infinite sets are "countable," that is, they have a one-to-one correspondence with the integers. Other infinite sets, such as the reals, are not countable.
- The data structures and operations defined on sets and relations in this chapter will be used in many different ways in the remainder of this book.

7.13 Bibliographic Notes for Chapter 7

Halmos [1974] provides a good introduction to set theory. Hashing techniques were first developed in the 1950's, and Peterson [1957] covers the early techniques. Knuth [1973] and Morris [1968] contain additional material on hashing techniques. Reingold [1972] discusses the computational complexity of basic set operations. The theory of infinite sets was developed by Cantor [1915].

Cantor, G. [1915]. "Contributions to the founding of the theory of transfinite numbers," reprinted by Dover Press, New York.

Halmos, P. R. [1974]. *Naive Set Theory*, Springer-Verlag, New York.

Knuth, D. E. [1973]. *The Art of Computer Programming*, Vol. III, *Sorting and Searching*, Addison-Wesley, Reading, Mass.

Morris, R. [1968]. "Scatter storage techniques," *Comm. ACM* **11**:1, pp. 35–44.

Peterson, W. W. [1957]. "Addressing for random access storage," *IBM J. Research and Development* **1**:7, pp. 130–146.

Reingold, E. M. [1972]. "On the optimality of some set algorithms," *J. ACM* **19**:4, pp. 649–659.