`deallocateNode` are invoked 1,000,000 times each. But `new` is invoked only 50 times. The time savings from the 999,950 calls to `new` that are not made plus the savings from the significantly reduced garbage collection effort are more than the cost of the 1,000,000 calls to each of `allocateNode` and `deallocateNode`.

The described modifications do not always result in a reduction in run time. For example, suppose we make 1,000,000 inserts, follow these with 1,000,000 removes, and then terminate the program. The original version of `Chain` makes 1,000,000 calls to `new`; the modified version makes 1,000,000 calls to each of the methods `new`, `allocateNode`, and `deallocateNode`.

The performance enhancement strategy just described also may be applied to the linked classes developed in later chapters.

EXERCISE

6. Develop the class `ManagedChain` that includes all the public instance methods of the class `Chain` (Program 6.2). The class `ManagedChain` includes class methods to allocate and deallocate nodes. The storage pool is initially empty; simulated pointers are not used. The allocate method `allocateNode` allocates a node from the storage pool whenever the storage pool is not empty. When the storage pool is empty, it gets a node by invoking the Java method `new`. The `remove` method should return the freed node to the storage pool, and the `add` method should invoke `allocateNode` rather than `new`.

   (a) Test the correctness of your code.

   (b) Comment on the relative merits of using the classes `ManagedChain` and `Chain` to represent a collection of chains.

7.7 APPLICATION—UNION-FIND PROBLEM

7.7.1 Equivalence Classes

Suppose we have a set $U = 1, 2, \cdots, n$ of n elements and a set $R = (i_1, j_1), (i_2, j_2), \cdots, (i_r, j_r)$ of r relations. The relation R is an **equivalence relation** iff the following conditions are true:

- $(a, a) \in R$ for all $a \in U$ (the relation is reflexive).
- $(a, b) \in R$ iff $(b, a) \in R$ (the relation is symmetric).
- $(a, b) \in R$ and $(b, c) \in$ R imply that $(a, c) \in R$ (the relation is transitive).

Often when we specify an equivalence relation R, we omit some of the pairs in R. The omitted pairs may be obtained by applying the reflexive, symmetric, and transitive properties of an equivalence relation.

Example 7.4 Suppose $n = 14$ and $R = \{(1,11), (7,11), (2,12), (12,8), (11,12), (3,13), (4,13), (13,14), (14,9), (5,14), (6,10)\}$. We have omitted all pairs of the form (a, a) because these pairs are implied by the reflexive property. Similarly, we have omitted all symmetric pairs. Since $(1,11) \in R$, the symmetric property requires $(11,1) \in R$. Other omitted pairs are obtained by applying the transitive property. For example, (7,11) and (11,12) imply (7,12). ■

Two elements a and b are equivalent if $(a, b) \in R$. An **equivalence class** is defined to be a maximal set of equivalent elements. *Maximal* means that no element outside the class is equivalent to an element in the class. Since it is not possible for an element to be in more than one equivalence class, an equivalence relation partitions the universe U into disjoint classes.

Example 7.5 Consider the equivalence relation of Example 7.4. Since elements 1 and 11, and 11 and 12 are equivalent, elements 1, 11, and 12 are equivalent. They are therefore in the same class. These three elements do not, however, form an equivalence class, as they are equivalent to other elements (e.g., 7). So {1, 11, 12} is not a maximal set of equivalent elements. The set {1, 2, 7, 8, 11, 12} is an equivalence class. The relation R defines two other equivalence classes: {3, 4, 5, 9, 13, 14} and {6, 10}. Notice that the three equivalence classes are disjoint. ■

In the **offline equivalence class** problem, we are given n and R and we need to determine the equivalence classes. From the definition of an equivalence class, it follows that each element is in exactly one equivalence class. In the **online equivalence class** problem, we begin with n elements, each in a separate equivalence class. We are to process a sequence of the operations: (1) `combine(a,b)` $\cdots$ combines the equivalence classes that contain elements `a` and `b` into a single class and (2) `find(theElement)` $\cdots$ determines the class that currently contains element `theElement`. The purpose of the find operation is to determine whether two elements are in the same class. Hence the find operation is to be implemented to return the same answer for elements in the same class and different answers for elements in different classes.

We can write the combine operation in terms of two `find`s and a `union` that actually takes two different classes and makes one. So `combine(a,b)` is equivalent to

```
classA = find(a);
classB = find(b);
if (classA != classB)
   union(classA, classB);
```

Notice that with the find and union operations, we can add new relations to R. For instance, to add the relation (a, b), we determine whether a and b are already in the same class. If they are, then the new relation is redundant. If they aren't, then we perform a `union` on the two classes that contain a and b.

In this section we are concerned primarily with the online equivalence problem, which is more commonly known as the **union-find** problem. Although the solutions developed in this section are rather simple, they are not the most efficient. Faster solutions are developed in Section 12.9.2. A fast solution for the offline equivalence problem is developed in Section 9.5.5.

7.7.2 Applications

The following examples show how a machine-scheduling problem and a circuit-wiring problem may be modeled as online equivalence class problems. A version of the circuit wiring problem may be modeled as an offline equivalence class problem.

Example 7.6 A certain factory has a single machine that is to perform n tasks. Task i has an integer release time r_i and an integer deadline d_i. The completion of each task requires one unit of time on this machine. A **feasible schedule** is an assignment of tasks to time slots on the machine such that task `i` is assigned to a time slot between its release time and deadline and no slot has more than one task assigned to it.

Consider the following four tasks:

Task	A	B	C	D
Release time	0	0	1	2
Deadline	4	4	2	3

Tasks A and B are released at time 0, task C is released at time 1, and task D is released at time 2. The following task-to-slot assignment is a feasible schedule: do task A from 0 to 1; task C from 1 to 2; task D from 2 to 3; and task B from 3 to 4 (see Figure 7.6).

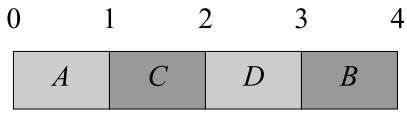

Figure 7.6 A schedule for four tasks

An intuitively appealing method to construct a schedule is

1. Sort the tasks into nonincreasing order of release time.
2. Consider the tasks in this nonincreasing order. For each task determine the free slot nearest to, but not after, its deadline. If this free slot is before the task's release time, fail. Otherwise, assign the task to this slot.

Exercise 9 asks you to prove that the strategy just described fails to find a feasible schedule only when such a schedule does not exist.

The online equivalence class problem can be used to implement step (2). For this step, let d denote the latest deadline of any task. The usable time slots are of the form "from $i-1$ to i" where $1 \leq i \leq d$. We will refer to these usable slots as slots 1 through d. For any slot a, define $near(a)$ as the largest i such that $i \leq a$ and slot i is free. If no such i exists, define $near(a) = near(0) = 0$. Two slots a and b are in the same equivalence class iff $near(a) = near(b)$.

Prior to the scheduling of any task, $near(a) = a$ for all slots, and each slot is in a separate equivalence class. When slot a is assigned a task in step (2), $near$ changes for all slots b with $near(b) = a$. For these slots the new value of $near$ is $near(a-1)$. Hence when slot a is assigned a task, we need to perform a `union` on the equivalence classes that currently contain slots a and $a-1$. If with each equivalence class e we retain, in $nearest[e]$, the value of $near$ of its members, then $near(a)$ is given by $nearest[\texttt{find}(a)]$. (Assume that the equivalence class name is taken to be whatever the `find` operation returns.) ■

Example 7.7 [From Wires to Nets] An electronic circuit consists of components, pins, and wires. Figure 7.7 shows a circuit with the three components A, B, and C. Each wire connects a pair of pins. Two pins a and b are **electrically equivalent** iff they are either connected by a wire or there is a sequence i_1, i_2, ... i_k of pins such that a, i_1; i_1, i_2; i_2, i_3; $\cdots$; i_{k-1}, i_k; and i_k, b are all connected by wires. A **net** is a maximal set of electrically equivalent pins. *Maximal* means that no pin outside the net is electrically equivalent to a pin in the net.

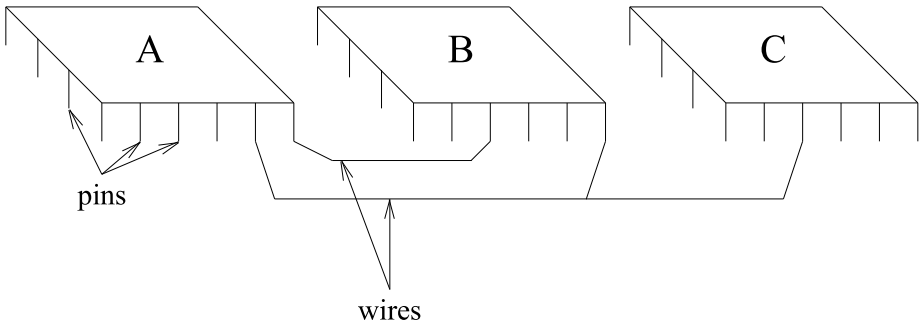

Figure 7.7 A three-chip circuit on a printed circuit board

Consider the circuit shown in Figure 7.8. In this figure only the pins and wires have been shown. The 14 pins are numbered 1 through 14. Each wire may be described by the two pins that it connects. For instance, the wire connecting pins

1 and 11 is described by the pair (1,11), which is equivalent to the pair (11,1). The set of wires is {(1,11), (7,11), (2,12), (12,8), (11,12), (3,13), (4,13), (13,14), (14,9), (5,14), (6,10)}. The nets are {1, 2, 7, 8, 11, 12}, {3, 4, 5, 9, 13, 14} and {6, 10}.

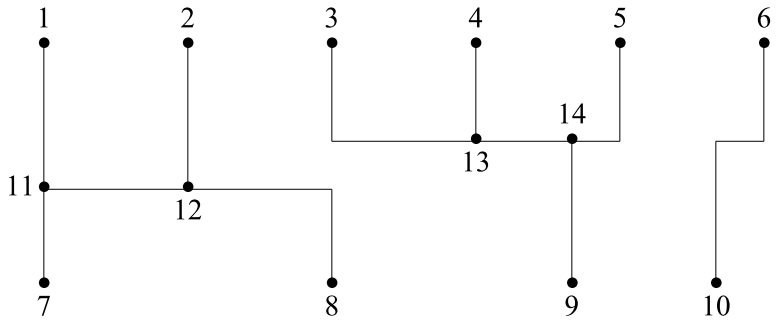

Figure 7.8 Circuit with pins and wires shown

In the **offline net finding problem**, we are given the pins and wires and are to determine the nets. This problem is modeled by the offline equivalence problem with each pin being a member of U and each wire a member of R.

In the **online** version we begin with a collection of pins and no wires and are to perform a sequence of operations of the form (1) add a wire to connect pins a and b and (2) find the net that contains pin a. The purpose of the find operation is to determine whether two pins are in the same net or in different nets. This version of the net problem may be modeled by the online equivalence class problem. Initially, there are no wires, and we have $R = \phi$. The net find operation corresponds to the equivalence class `find` operation and adding a new wire (a, b) corresponds to `combine`(a, b), which is equivalent to `union(find(a), find(b))`. ■

7.7.3 First Union-Find Solution

A simple solution to the online equivalence class problem is to use an array `equivClass` and let `equivClass[i]` be the class that currently contains element `i`. The methods to initialize, union, and find take the form given in Program 7.7. `n` is the number of elements. `n` and `equivClass` are both assumed to be class (i.e., static) data members. To unite two different classes, we arbitrarily pick one of these classes and change the `equivClass` values of all elements in this class to correspond to the `equivClass` values of the elements of the other class. Note that the inputs to `union` are `equivClass` values (i.e., the results of a `find` operation) and not element indexes. Even though `union` works correctly when a redundant union (i.e., one in which `classA = classB`), we make the assumption that redundant unions are not performed. The `initialize` and `union` methods have complexity Θ(`n`) (we

assume that `new` does not throw an exception when invoked by `initialize`), and the complexity of `find` is $\Theta(1)$. From Examples 7.6 and 7.7, we see that in any application of these methods, we will perform one initialization, u unions, and f finds. The time needed for all of these operations is $\Theta(\texttt{n+}u\texttt{*n+}f) = \Theta(u\texttt{*n+}f)$.

```
public class UnionFindFirstSolution
{
   static int [] equivClass;
   static int n;  // number of elements

   /** initialize numberOfElements classes with one element each */
   static void initialize(int numberOfElements)
   {
      n = numberOfElements;
      equivClass = new int [n + 1];
      for (int e = 1; e <= n; e++)
         equivClass[e] = e;
   }

   /** unite the classes classA and classB */
   static void union(int classA, int classB)
   {// assume classA != classB
      for (int k = 1; k <= n; k++)
         if (equivClass[k] == classB)
            equivClass[k] = classA;
   }

   /** find the class that contains theElement */
   static int find(int theElement)
      {return equivClass[theElement];}
}
```

Program 7.7 Union-find methods using arrays

7.7.4 Second Union-Find Solution

The time complexity of the union operation can be reduced by keeping a chain for each equivalence class because now we can find all elements in a given equivalence class by going down the chain for that class, rather than by examining all `equivClass` values. In fact, if each equivalence class knows its size, we can choose to change the `equivClass` values of the smaller equivalence class and perform the

union operation even faster. By using simulated pointers, we get quick access to the node that represents element `e`. We adopt the following conventions:

- `EquivNode` is a class with data members `equivClass`, `size`, and `next`. Program 7.8 gives the code for this class.

```
class EquivNode
{
   int equivClass;   // element class identifier
   int size;         // size of class
   int next;         // pointer to next element in class

   /** constructor */
   EquivNode(int theClass, int theSize)
   {
      equivClass = theClass;
      size = theSize;
      // next has the default value 0
   }
}
```

Program 7.8 The class `EquivNode`

- An array `node[1:n]` of type `EquivNode` is used to represent the `n` elements together with the equivalence class chains.
- `node[e].equivClass` is both the value to be returned by `find(e)` and a pointer to the first node in the chain for the equivalence class `node[e].equivClass`.
- `node[e].size` is defined only if `e` is the first node on a chain. In this case `node[e].size` is the number of nodes on the chain that begins at `node[e]`.
- `node[e].next` gives the next node on the chain that contains node `e`. Since the nodes in use are numbered 1 through `n`, a `null` pointer can be simulated by 0 rather than by -1.

Program 7.9 gives the new code for `initialize`, `union`, and `find`.

Since an equivalence class is of size $O(\mathtt{n})$, the complexity of the union operation is $O(\mathtt{n})$ when chains are used. The complexity of the initialization and find operations remain $O(\mathtt{n})$ and $\Theta(1)$, respectively. To determine the complexity of performing one initialization and a sequence of u unions and f finds, we will use the following lemma.

```
public class UnionFindSecondSolution
{
   static EquivNode [] node; // array of nodes
   static int n;             // number of elements

   /** initialize numberOfElements classes with one element each */
   static void initialize(int numberOfElements)
   {
      n = numberOfElements;
      node = new equivNode [n + 1];

      for (int e = 1; e <= n; e++)
         // node[e] is initialized so that its equivClass is e,
         // size is 1, and next is 0
         node[e] = new equivNode(e,1);
   }

   /** unite the classes classA and classB */
   static void union(int classA, int classB)
   {// assume classA != classB
      // make classA smaller class
      if (node[classA].size > node[classB].size)
      {// swap classA and classB
         int t = classA;
         classA = classB;
         classB = t;
      }

      //  change equivClass values of smaller class
      int k;
      for (k = classA; node[k].next != 0; k = node[k].next)
         node[k].equivClass = classB;
      node[k].equivClass = classB; // last node in chain

      // insert chain classA after first node in chain classB
      // and update new chain size
      node[classB].size += node[classA].size;
      node[k].next = node[classB].next;
      node[classB].next = classA;
   }
```

Program 7.9 Union-find methods using chains (continues)

```
   static int find(int theElement)
      {return node[theElement].equivClass;}
}
```

Program 7.9 Union-find methods using chains (concluded)

Lemma 7.1 *If we start with n classes that have one element each and perform u nonredundant unions, then*

1. *No class has more than $u+1$ elements.*
2. *At least $n-2u$ singleton classes remain.*
3. *$u < n$.*

Proof See Exercise 7. ■

The complexity of the initialize and f finds is $O(\mathtt{n+}f)$. For the u nonredundant unions, we note that the cost of each union is Θ(size of smaller class). During the union elements are moved from the smaller class to the bigger one. The complexity of a single union is O(number of elements moved), and the complexity of all u unions is O(total number of element moves). Following a union operation, each element that is moved to a new class ends up in a class whose size is at least twice that of the class the element was in before the union operation (because elements move from an initially smaller class into an initially bigger class). Therefore, since at the end no class has more than $u+1$ elements (Lemma 7.1(1)), no element can be moved more than $\log_2(u+1)$ times during the u unions. Furthermore, from Lemma 7.1(2), at most $2u$ elements can move (because the elements left in singleton classes have never moved). So the total number of element moves cannot exceed $2u\log_2(u+1)$. As a result, the time needed to perform the u unions is $O(u\log u)$. The complexity of the initialization and the sequence of u unions and f finds is therefore $O(\mathtt{n+}u\log u\mathtt{+}f)$.

EXERCISES

7. Prove Lemma 7.1.
8. Write a Java program for the online net finding problem of Example 7.7. Model the problem as the online equivalence class problem and use the chain method. Test the correctness of your program.
9. Prove that the strategy outlined in Example 7.6 fails to find a feasible schedule only when such a schedule does not exist.
10. Compare the run-time performance of Programs 7.7 and 7.9.