Example 7.3 [External Data Structures] Although this book focuses primarily on data structures that reside in internal memory—internal memory data structures—some data structures such as the B-tree of Section 16.4.3 are intended for data that reside on a disk. These data structures require pointers that cannot be Java references because these pointers refer to addresses on a disk. ■

Although these three examples involve data that are to be stored on a disk, some internal memory applications also benefit from the use of simulated pointers. One such application—the union-find problem—is developed in Sections 7.7 and 12.9.2. Another application—bipartite covers—is developed in Section 18.3.4.

7.2 SIMULATING POINTERS

We describe how to simulate pointers in internal memory. The techniques readily extend to disk storage. We implement linked lists using an array of nodes and simulate Java references by integers that are indexes into this array. Our linked data structures reside within this array. Therefore, to back up our data structures, for example, we need merely back up the contents of each node as it appears from left to right in the node array. To recover, we read back the node contents in left-to-right order and reestablish the node array.

Suppose we use an array `node` whose data type is `SimulatedNode` (Program 7.1). The type `SimulatedNode` differs from the data type `ChainNode` only in the data type of the member `next`—in `ChainNode` the member `next` is a Java reference, whereas in `SimulatedNode`, `next` is an `int`.

```
class SimulatedNode
{
   // package visible data members
   Object element;
   int next;

   // package visible constructors
   SimulatedNode() {}

   SimulatedNode(int next)
      {this.next = next;}
}
```

Program 7.1 Data type of nodes using simulated pointers

The nodes `node[0]`, `node[1]`, $\cdots$, `node[numberOfNodes-1]` are available for our linked data structures. We will refer to `node[i]` as node `i`, and we will use the integer -1 to play the role of `null`.

We may construct a simulated chain (i.e., a singly linked list of nodes that uses integer pointers; the last node's pointer is -1) `c` that consists of nodes 5, 2, and 7 (in that order) in the following way:

- set `c.firstNode` $= 5$ (pointer to first node on the simulated chain `c` is of type `int`)
- set `node[5].next` $= 2$ (pointer to second node on simulated chain)
- set `node[2].next` $= 7$ (pointer to next node)
- set `node[7].next` $= -1$ (indicating that node 7 is the last node on the chain).

Figure 7.1(a) shows a 10-node memory array `node` with our three-node simulated chain. The shaded nodes are the in-use nodes, and the unshaded ones are the free nodes. Simulated chain drawings such as the one in Figure 7.1(a) are hard to follow because they do not visually display the linear ordering of the chain nodes. We therefore draw simulated chains the same way as when Java references are used (Figure 7.1(b)); the nodes are drawn in their left-to-right list order, and simulated pointers are drawn as arrows.

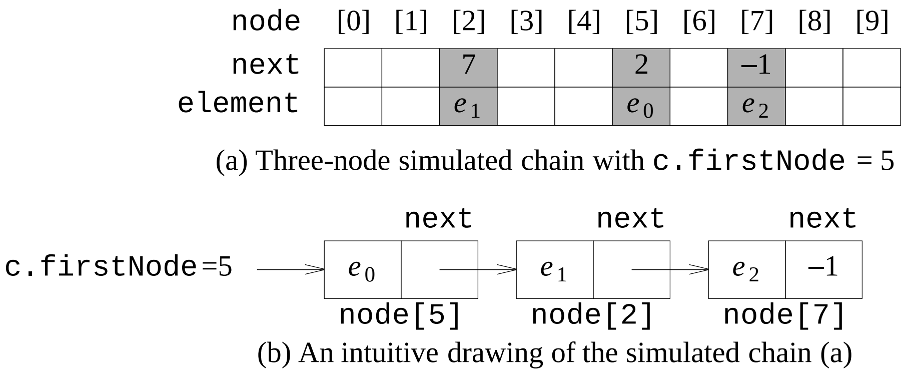

Figure 7.1 Simulated chain with three nodes

7.3 MEMORY MANAGEMENT

When we use simulated pointers, we must find a way to do some of the things that Java does for us when references are used as pointers. For example, we must be able to allocate a node for use (i.e., do the equivalent of the `new` method of Java) and reclaim previously allocated nodes that are no longer being used. A **memory management system** provides for three tasks—create a collection of nodes, allocate a node as needed, and reclaim nodes that are no longer in use.

Although memory management for nodes with different sizes is rather complex, memory management for nodes of the same size is fairly simple. The node array of Figure 7.1(a), for example, deals with fixed-size nodes (their data type is `SimulatedNode`, and each instance of this data type is 8 bytes long). We limit our development of methods to allocate and reclaim memory to a simple scheme for nodes of a fixed size.

The scheme we will develop deviates philosophically from that used by Java in the way memory is reclaimed; while Java relies on garbage collection, our scheme will require the user to explicitly reclaim memory that he/she no longer needs. This explicit reclaiming of memory is done by invoking a memory deallocation method.

In our simple memory management scheme, nodes that are not in use (nodes 0, 1, 3, 4, 6, 8, and 9 in Figure 7.1(a)) are kept in a **storage pool** from where they may be allocated for use as needed. Initially, this pool contains all nodes `node[0:numberOfNodes-1]` that are to be managed. `allocateNode` takes a node out of this pool and makes this node available to the user; the user may free a node that was in use by invoking the method `deallocateNode`, which puts a node back into the pool. The method `allocateNode` is functionally equivalent to the `new` method of Java. Since Java reclaims dynamically allocated storage that is no longer in use by a garbage collection process, Java does not have a method equivalent to `deallocateNode`.

Even though a general simulated-pointer system includes methods to initialize the storage pool and to allocate and deallocate nodes, many applications of simulated pointers need none or only some of these three methods. For example, our applications of simulated pointers in Sections 7.7 and 12.9.2 require none of these methods.

We will develop two classes `SimulatedSpace1` and `SimulatedSpace2` that implement memory management schemes for the case of equal size nodes. In `SimulatedSpace1`, all free nodes are on a simulated chain; in `SimulatedSpace2`, only free nodes that have been used at least once are on this chain. This difference in the organization of the storage pool mainly influences the time it takes to initialize the pool.

7.3.1 The Class `SimulatedSpace1`

In this implementation of a memory management system, the storage pool is set up as a simulated chain of nodes (as in Figure 7.2). This chain is called the **available space list**. It contains all nodes that are currently free. `firstNode` is a variable of type `int` that points to the first node on this chain. When a node is deallocated, it is added to the chain of free nodes; when a node is to be allocated for use, a node is removed from the free node chain. Additions to and removals from the chain of free nodes are made at the front.

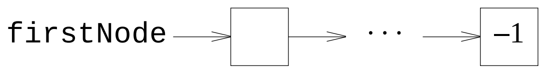

Figure 7.2 Available space list

The class `SimulatedSpace1` has the data members `firstNode` (integer pointer to first node on available space list) and `node` (array of type `SimulatedNode`). The class also has a constructor method that allocates the array `node` and initializes the available space list, the method `allocateNode` that allocates the first node on the available space list (in case the available space list is empty, `allocateNode` first doubles the array size and puts the additional nodes created by this doubling on to the available space list), and the method `deallocateNode` that returns a previously allocated node to the available space list.

Since all nodes are initially free, the available space list contains `numberOfNodes` nodes at the time it is created. The class contructor (Program 7.2) initializes the available space list. Programs 7.3 and 7.4 perform the `allocateNode` and `deallocateNode` operations.

```
public SimulatedSpace1(int numberOfNodes)
{
   node = new SimulatedNode [numberOfNodes];

   // create nodes and link into a chain
   for (int i = 0; i < numberOfNodes - 1; i++)
      node[i] = new SimulatedNode(i + 1);

   // last node of array and chain
   node[numberOfNodes - 1] = new SimulatedNode(-1);
   // firstNode has the default initial value 0
}
```

Program 7.2 Initialize available space list

The time complexity of the constructor is $O($`numberOfNodes`$)$. The complexity of `allocateNode` is $\Theta(1)$ except when it becomes necessary to double the size of the array `node`. When array doubling is necessary, the complexity of `allocateNode` is $O($`node.length`$)$. From the analysis that resulted in Theorem 5.1, it follows that

```
public int allocateNode(Object element, int next)
{// Allocate a free node and set its fields.
   if (firstNode == -1)
   {   // double number of nodes
       node = (SimulatedNode []) ChangeArrayLength.
              changeLength1D(node, 2 * node.length);

       // create and link new nodes
       for (int i = node.length / 2;
            i < node.length - 1; i++)
          node[i] = new SimulatedNode(i + 1);
       node[node.length - 1] = new SimulatedNode(-1);

       firstNode = node.length / 2;
   }

   int i = firstNode;         // allocate first node
   firstNode = node[i].next;  // firstNode points to
                              // next free node
   node[i].element = element;
   node[i].next = next;
   return i;
}
```

Program 7.3 Allocate a node using simulated pointers

```
public void deallocateNode(int i)
{// Free node i.
   // make i first node on free space list
   node[i].next = firstNode;
   firstNode = i;

   // remove element reference so that space can be garbage collected
   node[i].element = null;
}
```

Program 7.4 Deallocate a node with simulated pointers the time required to allocate n nodes is $\Theta(n)$. The complexity of deallocateNode is $\Theta(1)$.

7.3.2 The Class SimulatedSpace2

We can reduce the run time of the constructor (Program 7.2) by eliminating the for loop and invoking new only when a new node is actually needed. We maintain two integer variables first1 and first2. first1 is such that we have yet to invoke new for node[i], first1 $\leq$ i $<$ node.length; first2 points to the first node in a chain of nodes that have been used at least once and are currently not in use. Initially, first1 $=$ 0 and first2 $=$ -1. Whenever a node is deallocated, it is put onto the chain pointed at by first2. When a new node is needed, we allocate the first node from this chain provided this chain is not empty. Otherwise, we invoke new and construct a node for position first1 of the array node and make this new node available for use. The code for SimulatedSpace2 is available from the Web site.

7.3.3 Evaluation of Simulated Memory Management

We make the following observations:

- When the single list scheme is in use, simulated chains can be built by explicitly setting the link field only in the last node because the appropriate link values are already present in the remaining nodes (see Figure 7.2). This advantage can also be incorporated into the dual available space list scheme by writing a method getNodes(n) that provides a chain with n nodes on it. This method will explicitly set links only when nodes are taken from the first available space list.

- An entire chain of nodes can be freed efficiently using either of these schemes. For instance, if we know the front f and end e of a chain, all nodes on it are freed by the following statements:

  ```
  node[e].next = firstNode;
  firstNode = f;
  ```

  When the nodes of a simulated chain are deallocated in this way, garbage collection of the memory used by the objects referenced by the element fields of the nodes is not enabled. To enable garbage collection of this space, we must set the element fields of the freed nodes to null.

- If theList is a simulated circular list, then all nodes on it are deallocated in $\Theta(1)$ time by using Program 7.5. S is the simulated space used by the circular lists. Figure 7.3 shows the link changes that take place. Note that in this code `theList.firstNode` points to the first node in the circular list `theList`, `secondNode` points to the second node of `theList`, and `firstNode` points the first node of the available space list (a single space list is assumed).

```
public void deallocateCircular(SimulatedCircularList theList)
{// Deallocate all nodes in the circular list theList.
   if (theList.firstNode != -1)
   {// theList is not empty
      int secondNode = S.node[theList.firstNode].next;
      S.node[theList.firstNode].next = firstNode;
      firstNode = secondNode;
      theList.firstNode = -1;  // theList is now empty
   }
}
```

Program 7.5 Deallocate a circular list

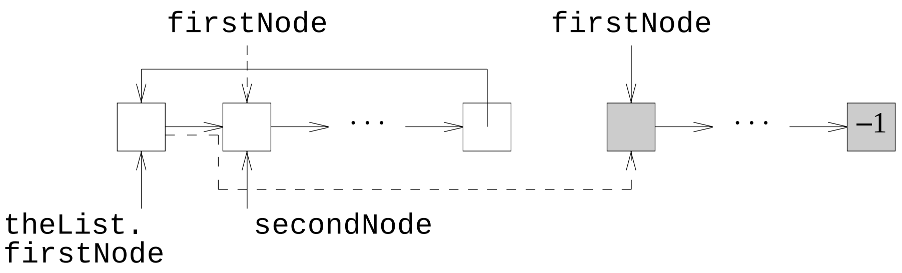

Figure 7.3 Deallocating a circular list

7.4 COMPARISON WITH GARBAGE COLLECTION

The use of garbage collection, as is done in Java, to reclaim free storage and the use of a method such as `deallocateNode` represent two fundamentally different approaches to memory management. In the garbage collection method, memory that was in use but has subsequently become free is returned to the storage pool by a program called the **garbage collector**. Garbage collectors typically perform two tasks—identify free memory (this task is called **marking**) and return the identified free memory to the storage pool.

A third task—garbage compaction—is often associated with garbage collection. In garbage compaction data in memory that is in use are relocated so that the in-use