4.6 BEYOND SIMPLE LINKED LISTS

Arrays Versus Linked Lists and Doubly Linked Lists

Many ADTs can be implemented with either arrays or linked lists. Certainly, the bag and the sequence ADT could each be implemented with either approach.

Which approach is better? There is no absolute answer, but there are certain operations that are better performed by arrays and others where linked lists are preferable. This section provides some guidelines.

Arrays Are Better at Random Access. The term **random access** refers to examining or changing an arbitrary element that is specified by its position in a list. For example: *What is the 42nd element in the list*? Or another example: *Change the element at position 1066 to a 7*. These are constant time operations for an array. But in a linked list, a search for the i^{th} element must begin at the head and will take $O(i)$ time. Sometimes there are ways to speed up the process, but even improvements remain linear time in the worst case.

If an ADT makes significant use of random-access operations, then an array is better than a linked list.

Linked Lists Are Better at Additions/Removals at a Cursor. Our sequence ADT maintains a *cursor* that refers to a "current element." Typically, a cursor moves through a list one element at a time without jumping around to random locations. If all operations occur at the cursor, then a linked list is preferable to an array. In particular, additions and removals at a cursor generally are linear time for an array (since elements that are after the cursor must *all* be shifted up or back to a new index in the array). But these operations are constant time operations for a linked list. Also remember that effective additions and removals in a linked list generally require maintaining both a cursor and a *precursor* (which refers to the node before the cursor).

If an ADT's operations take place at a cursor, then a linked list is better than an array.

Resizing Can Be Inefficient for an Array. A collection class that uses an array generally provides a method to allow a programmer to adjust the capacity as needed. But changing the capacity of an array can be inefficient. The new memory must be allocated and initialized, and the elements are then copied from the old memory to the new memory. If a program can predict the necessary capacity ahead of time, then capacity is not a big problem since the object can be given sufficient capacity from the outset. But sometimes the eventual capacity is unknown, and a program must continually adjust the capacity. In this situation, a linked list has advantages. When a linked list grows, it grows one node at a time, and there is no need to copy elements from old memory to new memory.

If an ADT is frequently adjusting its capacity, then a linked list may be better than an array.

Doubly Linked Lists Are Better for a Two-Way Cursor. Sometimes list operations require a cursor that can move forward and backward through a list—a kind of **two-way cursor**. This situation calls for a **doubly linked list**, which is like an ordinary linked list except that each node contains two references: one linking to the next node and one linking to the previous node. An example of a doubly linked list of integers is shown in Figure 4.19 (in the margin) along with references to its head and tail. Here is the start of a declaration for a doubly linked list with integer data:

```
public class IntTwoWayNode
{
   private int data;
   private IntTwoWayNode backlink;
   private IntTwoWayNode forelink;
   ...
```

The `backlink` refers to the previous node, and the `forelink` refers to the next node in the list.

If an ADT's operations take place at a two-way cursor, then a doubly linked list is the best implementation.

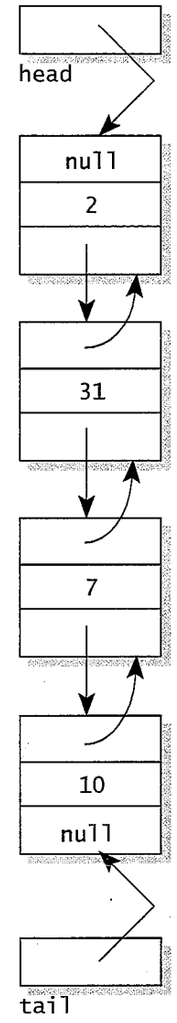

FIGURE 4.19
Doubly Linked List

Dummy Nodes

Many linked list operations require special treatment when the operation occurs on an empty list or the operation occurs at the head or tail node. The special treatment requires extra programming that results in more opportunity for programming errors. Because of this, programmers often create an extra node at the beginning of each list (called a **dummy head node**) and an extra node at the end of each list (called a **dummy tail node**). The data in these two dummy nodes is not part of the list. The nodes are present only to make it easier to program certain operations. Dummy nodes may be used with ordinary (singly linked) lists or with doubly linked lists.

A list with a dummy head node and a dummy tail node has another advantage: The references to the head and tail are set up just once (using the dummy nodes). After they are set up, the head and tail will never need to be changed, never need to refer to some other node. On the other hand, without dummy nodes, the head or tail will need to refer to new nodes whenever a node is inserted or removed at the front or back of the list.

For example, consider the problem of removing a node from a doubly linked list that is maintained with references to both the head and the tail, as in Figure 4.19. If there are no dummy nodes, then there are many cases to consider:

1. Suppose there is only one node on the list, and we are removing it. Then both the head and the tail must be set to null.

2. Suppose we are removing the head node from a list with more than one node. In this case, we must reset the head reference variable to

refer to the second node and also set the backward link of this new head node to null.

3. Suppose we are removing the tail node from a list with more than one node. In this case, we must reset the tail reference variable to refer to the next-to-last node and also set the forward link of this new tail to null.

4. Suppose we are removing a node from the middle of a list (neither the head nor the tail). In this case, we can set up two references: `before` (which refers to the node before the node we're removing) and `after` (which refers to the node after the node we're removing). Then, assuming we have methods to set the forward and backward links:

```
before.setForwardLink(after);
after.setBackwardLink(before);
```

There are lots of cases to consider and lots of cases to maybe go wrong. But if we have a dummy head node and a dummy tail node, then these dummy nodes are never removed, and, from the four cases, only Case 4 is possible. The Case 4 is also nice because it never needs to change head or tail reference variables.

If you are programming a linked list from scratch (with no pre-existing methods to manipulate the list), then consider maintaining both a dummy head node and a dummy tail node. In this case, methods for searching the list may require extra care (to not search the dummy nodes), but both insertion and removal are significantly easier to program.

Making the Decision

Your decision on what kind of implementation to use is based on your knowledge of which operations occur in the ADT, which operations you expect to be performed most often, and whether you expect your arrays to require frequent capacity changes. Figure 4.20 summarizes these considerations.

FIGURE 4.20 Guidelines for Choosing Between an Array and a Linked List

Frequent random access operations	Use an array.
Operations occur at a cursor	Use a linked list.
Operations occur at a two-way cursor	Use a doubly linked list.
Frequent capacity changes	A linked list avoids resizing inefficiency.
	Consider using a dummy head and tail.

Self-Test Exercises for Section 4.6

38. What underlying data structure is quickest for random access?
39. What underlying data structure is quickest for additions/removals at a cursor?
40. What underlying data structure is best if a cursor must move both forward and backward?
41. What is the typical worst-case time analysis for changing the capacity of a collection class that is implemented with an array?
42. For the `IntTwoWayNode` declaration on page 233: Implement a method to remove a node from its list. The node that activates the method is the node to be removed. Assume that there are public methods to get or set the forward or backward link of a node.

CHAPTER SUMMARY

- A *linked list* consists of nodes; each *node* contains some data and a link to the next node in the list. The link part of the final node contains the null reference.
- Typically, a linked list is accessed through a *head reference* that refers to the *head node* (i.e., the first node). Sometimes a linked list is accessed elsewhere, such as through the *tail reference* that refers to the last node.
- You should be familiar with the methods of our node class, which provides fundamental operations to manipulate linked lists. These operations follow basic patterns that every programmer uses.
- Our linked lists can be used to implement ADTs. Such an ADT will have one or more private instance variables that are references to nodes in a linked list. The methods of the ADT will use the node methods to manipulate the linked list.
- You have seen two ADTs implemented with linked lists: a bag and a sequence. You will see more in the chapters that follow.
- ADTs often can be implemented in many different ways, such as by using an array or using a linked list. In general, arrays are better at *random access*; linked lists are better at additions/removals at a *cursor*.
- A *doubly linked list* has nodes with two references: one to the next node and one to the previous node. Doubly linked lists are a good choice for supporting a cursor that moves forward and backward.
- Including dummy head and tail nodes on a linked list will simplify the programming for insertion and removal of nodes.