LinkedLists.mkr Page 445 Wednesday, April 8, 1998 1:17 PM

16.3 Doubly Linked Lists and Circular Linked Lists

As mentioned in Section 16.2, the singly linked list does not efficiently support some important operations. For instance, although it is easy to go to the front of the list, it is time-consuming to go to the end. Although we can easily advance via `advance`, implementing `retreat` cannot be done efficiently with only a `next` reference. In some applications, this might be critical. For instance, when designing a text editor, we can maintain the internal image of the file as a linked list of lines. We certainly want to be able to move up just as easily as down, to insert both before and after a line rather than just after, and to be able to get to the last line quickly. A moment's thought suggests that to implement this efficiently, we should have each node maintain two references: one to the next node in the list and one to the previous node. Then, to make everything symmetric, we should have not only a header but also a tail. This is the so-called *doubly linked list*. Figure 16.16 shows the doubly linked list representing `a`, `b`. Each node now has two references (`next` and `prev`), and searching and moving can easily be performed in both directions. Obviously, there are some important changes.

Doubly linked lists allow bidirectional traversal by storing two references per node.

First, an empty list now consists of a `head` and `tail` connected together, as shown in Figure 16.17 (page 446). Notice, by the way, that `head.prev` and `tail.next` are not needed in the algorithms and are not even initialized. The test for emptiness is now

Symmetry demands that we use both a `head` and a `tail` and that we support roughly twice as many operations.

```
head.next == tail
```

or

```
tail.prev == head
```

We no longer use `null` to decide if an advance has taken us past the end of the list. Instead, we have gone past the end if `current` is either `head` or `tail` (recall that we can go in either direction). `retreat` can be implemented by

When we advance past the end of the list, we now hit the `tail` node instead of `null`.

```
current = current.prev;
```

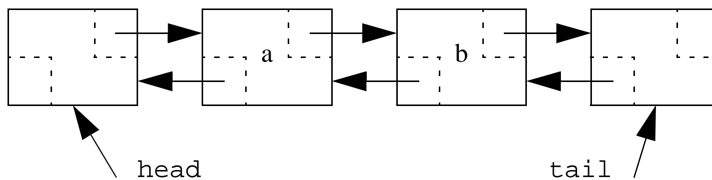

Figure 16.16 Doubly linked list