Chapter 12-3, Lecture Notes

Data with Multiple Organizations

1. Many applications require a data organization that simultaneously support several different data-management tasks.
   1.1 One simple example involves waiting list of customers, that is, a queue of customer records. In addition requiring the standard queue operations iSEmpty, enqueue, dequeue, and getFront, suppose that the application frequently requires a listing of the customer records in the queue.
   1.2 This listing is more useful if the records appear sorted by customer name.
   1.3 You thus need a traverse operation that visits the customer records in sorted order.
      1.3.1 This scenario presents an interesting problem.
      1.3.2 If you simply store the customer records in a queue, they will not be sorted by name. If you just store the records in sorted order, you will be unable to process the customers on a first-come, first-served basis.
      1.3.3 The problem requires you to organize the data in two different ways.

   1.4 One solution is to maintain two independent data structures, one organized to support the sorted traversal and the other organized to support queue operations.
      1.4.1 Figure 12-51 depicts a sorted linked list of customer reco and a pointer-based implementation of the queue.
      1.4.2 The pointer-based data structures are a good choice because they do not require a good estimate the maximum number of customer records that must be stored.
      1.4.3 One disadvantage of this scheme is the space needed to store copies of each customer record.
      1.4.4 In addition, not all of the required operations are supported as efficiently as possible.
      1.4.5 How well does this scheme support required operations?
   1.5 The operations that only retrieve data-sorted traverse and getFront are easy to perform.
      1.5.1 You can obtain a sorted listing of customer records by traversing the sorted linked list, and you can perform the queue getFront operation by inspecting the record at the front of the queue.
      1.5.2 The operations, enqueue and dequeue are, however, more difficult to perform because must modify the data.

      The enqueue operation has two steps:

      1. Insert a copy of the new customer record at the back of the queue. step requires only a few pointer changes.

      2. Insert a copy of the new customer record into its proper position in sorted linked list. This step requires a traversal of the sorted linked list.

Similarly, the dequeue operation has two steps:

1. Delete the customer at the front of the queue, but retain a copy of the name for the next step. This step requires only a few pointer changes.

2. Search the sorted linked list for the name just removed from the queue, and delete from the list the customer record containing this name. This step requires a traversal of the sorted linked list.

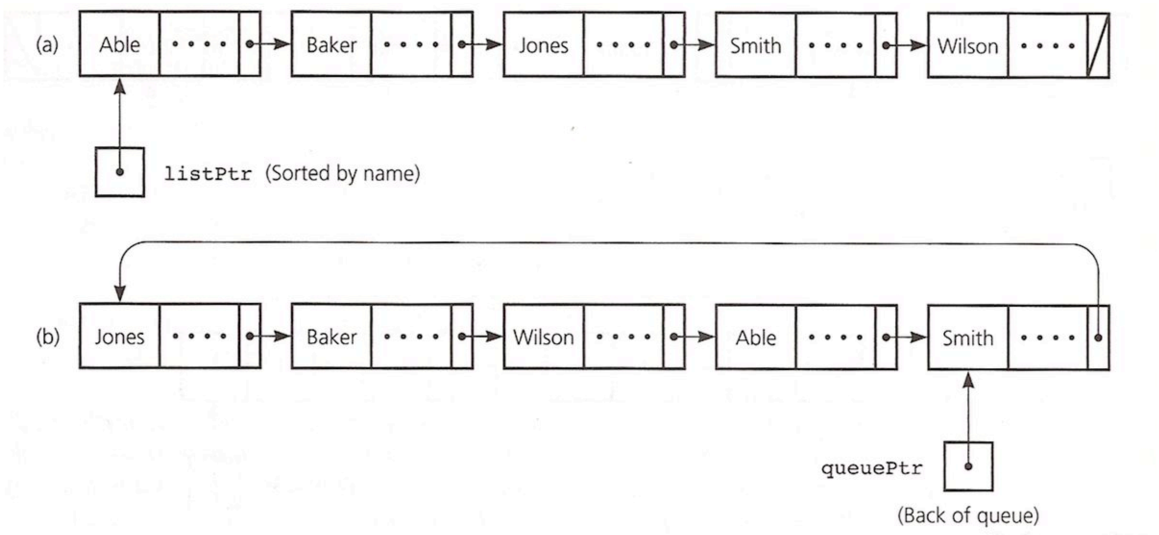

FIGURE 12-51

Independent data structures: (a) a sorted linked list; (b) a pointer-based queue

1.6 The scheme does efficiently support the traverse and getFront operations, but enqueue and dequeue require a traversal of the sorted linked list (whereas in a queue alone enqueue and dequeue require only a small, constant number of steps).
   1.6.1 Improve on this scheme?
   1.6.2 One possibility is to store the customer records in a binary search tree rather than a sorted linked list.
   1.6.3 This approach would allow you to perform the second steps of the enqueue and dequeue operations much more efficiently.
   1.6.4 While the binary search tree strategy is certainly an improvement over the original scheme, the enqueue and dequeue operations would still require significantly more work than they would for a normal queue.

1.7 A different kind of scheme, one that supports the dequeue operation almost as efficiently as if you were maintaining only a queue, is possible by allowing the data structures to communicate with each other.

1.7.1 This concept is demonstrated here first with a sorted linked list and a queue, and then with more-complex structures, such as a binary search tree.

1.7.2 In the data structure shown in Figure 12-52, the sorted linked list still contains customer records, but the queue now contains only pointers to customer records.

1.7.3 That is, each entry of the queue points to the record in the sorted linked list for the customer at the given queue position.

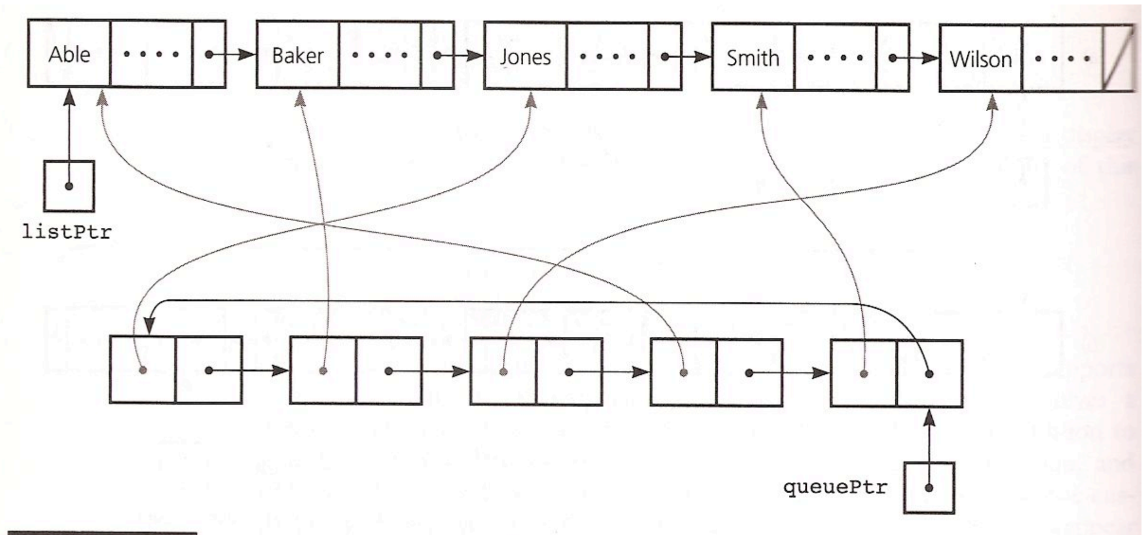

FIGURE 12-52

A queue pointing into a sorted linked list

1.7.4 An obvious advantage of storing only pointers in the queue is that the storage requirements are reduced, since a pointer is likely to be much smaller than a customer record.

1.7.5 This scheme also significantly improves the efficiency of the dequeue operation.

1.7.6 The efficiency of the traverse, getFront, and enqueue operations does not differ significantly from that of the original scheme that Figure 12-51 depicts.

1.7.7 You still perform the traverse operation by traversing the sorted linked list. However, you perform getFront and enqueue as follows:

```
getFront(out queueFront:ItemType)

    Let p be the pointer stored at the front of the queue (p points to the node,
    which is in the sorted linked list, that contains the record for the customer
    at the front of the queue)

    queueFront = item in the node to which p points
```

```
enqueue(in newItem:ItemType)

    Find the proper position for newItem in the sorted linked list
    Insert a node that contains newItem into this position

    Insert a pointer to the new node at the back of the queue
```

The real benefit of the new scheme is in the implementation of the dequeue operation:

```
dequeue()

    Delete the item at the front of the queue and retain its value p (p points to
    the node that contains the customer record to be deleted)

    Delete from the sorted linked list the node to which p points
```

1.8 Because the front of the queue contains a pointer to the customer record R that you want to delete, there is no need to search the sorted linked list.
   1.8.1 You have a pointer to the appropriate record, and all you need to do is delete it.
   1.8.2 There is one big problem, however.
      1.8.2.1 Because you are able to go directly to R without traversing the linked list from its beginning, you have no trailing pointer to the record that precedes R on the list!
      1.8.2.2 Recall that you must have a trailing pointer to delete the record.
      1.8.2.3 As the scheme now stands, the only way to obtain the trailing pointer is to traverse the linked list from its beginning, but this requirement negates the advantage gained by having the queue point into the linked list.
      1.8.2.4 Can solve this problem by replacing the singly linked list in Figure 12-52 with a doubly linked list, as shown in Figure 12-53.
   1.8.3 To summarize, you have seen a fairly good scheme for implementing the queue operations plus a sorted traversal.
   1.8.4 The only operation whose efficiency you might improve significantly is enqueue, since you still must traverse the linked list to find the proper place to insert a new customer record.

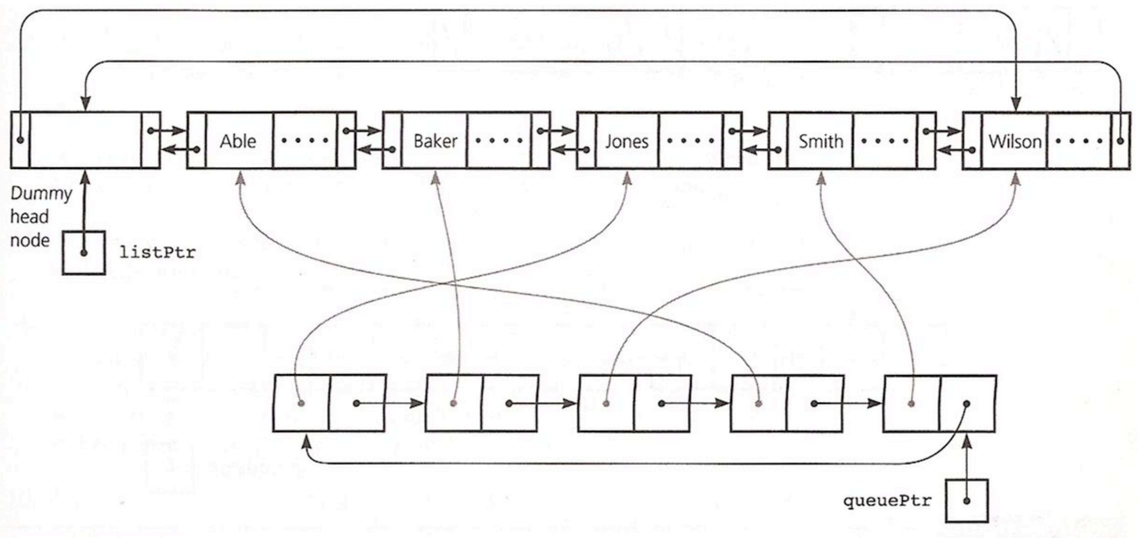

FIGURE 12-53
A queue pointing into a doubly linked list

1.9 The choice to store the customer records in a linear linked list was made to simplify the discussion.

- 1.9.1 A more efficient scheme has the queue point. into a binary search tree rather than a linked list.
- 1.9.2 This data structure allows you to perform the enqueue operation in logarithmic time, assuming that the tree remains balanced. To support the dequeue operation efficiently, however, you need a doubly linked tree.
- 1.9.3 That is, each node in the tree must point to its parent so that you can easily delete the node to which the front of the queue points.
- 1.9.4 Figure 12-54 illustrates this data structure; its implementation, which is somewhat difficult, is the subject of Programming Problem 9.
- 1.9.5 In general, you can impose several different organizations at the same time on a set of data

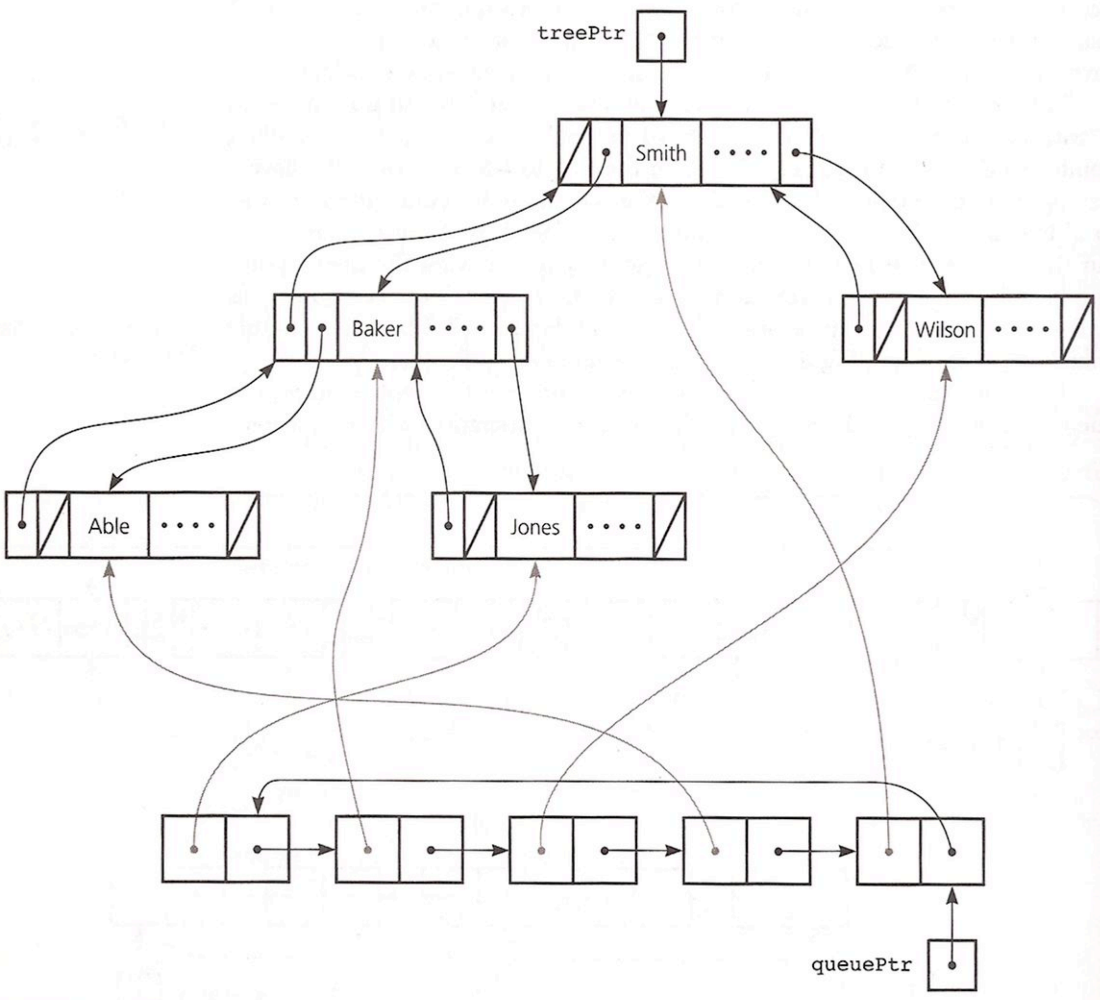

FIGURE 12-54

A queue pointing into a doubly linked binary search tree