Linked Lists

An array is a very useful data structure provided in programming languages. However, it has at least two limitations: (1) its size has to be known at compilation time and (2) the data in the array are separated in computer memory by the same distance, which means that inserting an item inside the array requires shifting other data in this array. This limitation can be overcome by using *linked structures*. A linked structure is a collection of nodes storing data and links to other nodes. In this way, nodes can be located anywhere in memory, and passing from one node of the linked structure to another is accomplished by storing the addresses of other nodes in the linked structure. Although linked structures can be implemented in a variety of ways, the most flexible implementation is by using pointers.

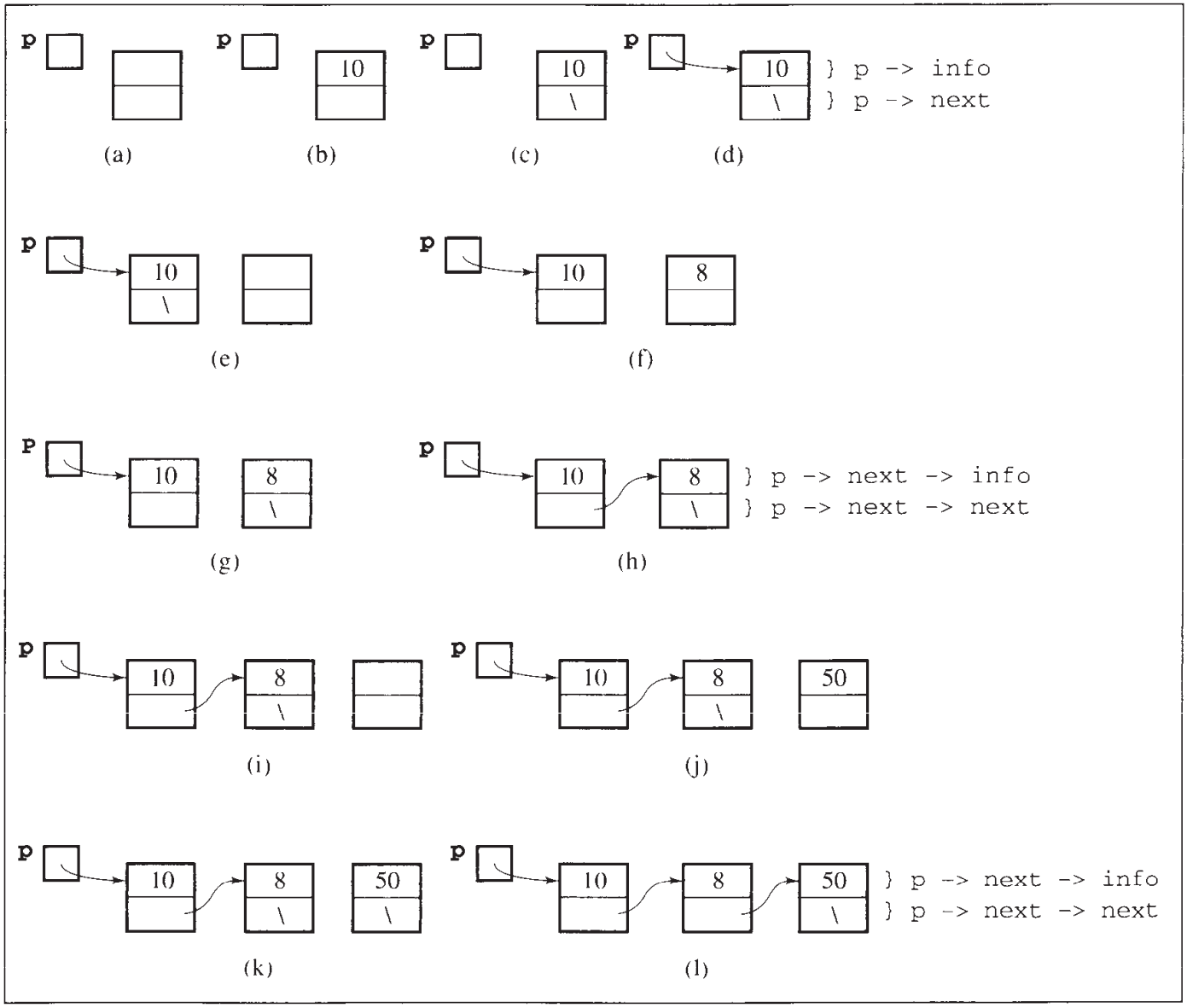
3.1 Singly Linked Lists

If a node contains a data member that is a pointer to another node, then many nodes can be strung together using only one variable to access the entire sequence of nodes. Such a sequence of nodes is the most frequently used implementation of a *linked list*, which is a data structure composed of nodes, each node holding some information and a pointer to another node in the list. If a node has a link only to its successor in this sequence, the list is called a *singly linked list*. An example of such a list is shown in Figure 3.1. Note that only one variable `p` is used to access any node in the list. The last node on the list can be recognized by the null pointer.

Each node in the list in Figure 3.1 is an instance of the following class definition:

```
class IntNode {
public:
    IntNode() {
```

FIGURE 3.1 A singly linked list.




```
        next = 0;
    }
    IntNode(int i, IntNode *in = 0) {
        info = i; next = in;
    }
    int info;
    IntNode *next;
};
```

A node includes two data members: `info` and `next`. The `info` member is used to store information, and this member is important to the user. The `next` member is

used to link together nodes to form a linked list. It is an auxiliary data member used to maintain the linked list. It is indispensable for implementation of the linked list but less important (if at all) from the user's perspective. Note that `IntNode` is defined in terms of itself because one data member, `next`, is a pointer to a node of the same type that is just being defined. This circularity, however, is permitted in C++.

The definition of a node also includes two constructors. The first constructor initializes the `next` pointer to null and leaves the value of `info` unspecified. The second constructor takes two arguments, one to initialize the `info` member and another to initialize the `next` member. The second constructor also covers the case when only one numerical argument is supplied by the user. In this case, `info` is initialized to the argument and `next` to null.

Now, let us create the linked list in Figure 3.11. One way to create this three-node linked list is to first generate the node containing number 10, then the node containing 8, and finally the node containing 50. Each node has to be initialized properly and incorporated into the list. To see this, each step is illustrated in Figure 3.1 separately. First, we execute the declaration and assignment

```
IntNode *p = new IntNode(10);
```

which creates the first node on the list and makes the variable `p` a pointer to this node. This is done in four steps. In the first step, a new `IntNode` is created (Figure 3.1a), in the second step, the `info` member of this node is set to 10 (Figure 3.1b), and in the third step, the node's `next` member is set to `null` (Figure 3.1c). The null pointer is marked with a slash in the pointer data member. Note that the slash in the `next` member is not a slash character. That is, the second and third steps—initialization of data members of the new `IntNode`—are performed by invoking the constructor `IntNode(10)`, which turns into the constructor `IntNode(10,0)`. The fourth step is making `p` a pointer to the newly created node (Figure 3.1d). This pointer is the address of the node, and it is shown as an arrow from the variable `p` to the new node.

The second node is created with the assignment

```
p->next = new IntNode(8);
```

where `p->next` is the `next` member of the node pointed by `p` (Figure 3.1d). As before, four steps are executed:

1. creating a new node (Figure 3.1e),
2. assigning by the constructor number 8 to the `info` member of this node (Figure 3.1f) and
3. `null` to its `next` member (Figure 3.1g), and finally
4. including the new node in the list by making the `next` member of the first node a pointer to the new node (Figure 3.1h).

Note that the data members of nodes pointed to by `p` are accessed using the arrow notation, which is clearer than using a dot notation, as in `(*p).next`.

The linked list is now extended by adding a third node with the assignment

```
p->next->next = new IntNode(50);
```

where p->next->next is the next member of the second node. This cumbersome notation has to be used because the list is accessible only through the variable p.

In processing the third node, four steps are also executed: creating the node (Figure 3.1i), initializing its two data members (Figure 3.1j–k), and then incorporating the node in the list (Figure 3.1l).

Our linked list example illustrates a certain inconvenience in using pointers: The longer the linked list, the longer the chain of nexts to access the nodes at the end of the list. In this example, p->next->next->next allows us to access the next member of the 3rd node on the list. But what if it were the 103rd or, worse, the 1003rd node on the list? Typing 1003 nexts, as in p->next->...->next, would be daunting. If we missed one next in this chain, then a wrong assignment is made. Also, the flexibility of using linked lists is diminished. Therefore, other ways of accessing nodes in linked lists are needed. One way is always to keep two pointers to the linked list: one to the first node and one to the last, as shown in Figure 3.2.

FIGURE 3.2 An implementation of a singly linked list of integers.

```
//************************  intSLLst.h  ***************************
//           singly-linked list class to store integers

#ifndef INT_LINKED_LIST
#define INT_LINKED_LIST

class IntNode {
public:
    int info;
    IntNode *next;
    IntNode(int el, IntNode *ptr = 0) {
        info = el; next = ptr;
    }
};

class IntSLList {
public:
    IntSLList() {
        head = tail = 0;
    }
    ~IntSLList();
    int isEmpty() {
        return head == 0;
    }
    void addToHead(int);
```

FIGURE 3.2 *(continued)*

```
    void addToTail(int);
    int  deleteFromHead(); // delete the head and return its info;
    int  deleteFromTail(); // delete the tail and return its info;
    void deleteNode(int);
    bool isInList(int) const;
private:
    IntNode *head, *tail;
};

#endif

//************************  intSLLst.cpp ***************************

#include <iostream.h>
#include "intSLLst.h"

IntSLList::~IntSLList() {
    for (IntNode *p; !isEmpty(); ) {
        p = head->next;
        delete head;
        head = p;
    }
}
void IntSLList::addToHead(int el) {
    head = new IntNode(el,head);
    if (tail == 0)
       tail = head;
}
void IntSLList::addToTail(int el) {
    if (tail != 0) {    // if list not empty;
         tail->next = new IntNode(el);
         tail = tail->next;
    }
    else head = tail = new IntNode(el);
}
int IntSLList::deleteFromHead() {
    int el = head->info;
    IntNode *tmp = head;
    if (head == tail)   // if only one node in the list;
         head = tail = 0;
```

Continues

FIGURE 3.2 *(continued)*

```
    else head = head->next;
    delete tmp;
    return el;
}
int IntSLList::deleteFromTail() {
    int el = tail->info;
    if (head == tail) { // if only one node in the list;
        delete head;
        head = tail = 0;
    }
    else {              // if more than one node in the list,
        IntNode *tmp;  // find the predecessor of tail;
        for (tmp = head; tmp->next != tail; tmp = tmp->next);
        delete tail;
        tail = tmp;    // the predecessor of tail becomes tail;
        tail->next = 0;
    }
    return el;
}
void IntSLList::deleteNode(int el) {
    if (head != 0)                    // if non-empty list;
        if (head == tail && el == head->info) { // if only one
            delete head;                        // node in the list;
            head = tail = 0;
        }
        else if (el == head->info) {// if more than one node in the list
            IntNode *tmp = head->next;
            head = head->next;
            delete tmp;            // and old head is deleted;
        }
        else {                     // if more than one node in the list
            IntNode *pred, *tmp;
            for (pred = head, tmp = head->next; // and a non-head node
                 tmp != 0 && !(tmp->info == el);// is deleted;
                 pred = pred->next, tmp = tmp->next);
            if (tmp != 0) {
                pred->next = tmp->next;
                if (tmp == tail)
                   tail = pred;
                delete tmp;
            }
```

FIGURE 3.2 *(continued)*

```
        }
}
bool IntSLList::isInList(int el) const {
    IntNode *tmp;
    for (tmp = head; tmp != 0 && !(tmp->info == el); tmp = tmp->next);
    return tmp != 0;
}
```

A singly linked list implementation in Figure 3.2 uses two classes: one class, `IntNode`, for nodes of the list, and another, `IntSLList`, for access to the list. The class `IntSLList` defines two data members, `head` and `tail`, which are pointers to the first and the last nodes of a list. This explains why all members of `IntNode` are declared public. Because particular nodes of the list are accessible through pointers, nodes are made inaccessible to outside objects by declaring `head` and `tail` private so that the information-hiding principle is not really compromised. If some of the members of `IntNode` were declared nonpublic, then classes derived from `IntSLList` could not access them.

An example of a list is shown in Figure 3.3. The list is declared with the statement

```
IntSLList list;
```

The first object in Figure 3.3a is not part of the list; it allows for having access to the list. For simplicity, in subsequent figures, only nodes belonging to the list are shown, the access node is omitted, and the `head` and `tail` members are marked as in Figure 3.3b.

Besides the `head` and `tail` members, the class `IntSLList` also defines member functions that allow us to manipulate the lists. We now look more closely at some basic operations on linked lists presented in Figure 3.2.

3.1.1 Insertion

Adding a node at the beginning of a linked list is performed in four steps.

1. An empty node is created. It is empty in the sense that the program performing insertion does not assign any values to the data members of the node (Figure 3.4a).
2. The node's `info` member is initialized to a particular integer (Figure 3.4b).
3. Because the node is being included at the front of the list, the `next` member becomes a pointer to the first node on the list, that is, the current value of `head` (Figure 3.4c).
4. The new node precedes all the nodes on the list, but this fact has to be reflected in the value of `head`; otherwise, the new node is not accessible. Therefore, `head` is updated to become the pointer to the new node (Figure 3.4d).

FIGURE 3.3 A singly linked list of integers.

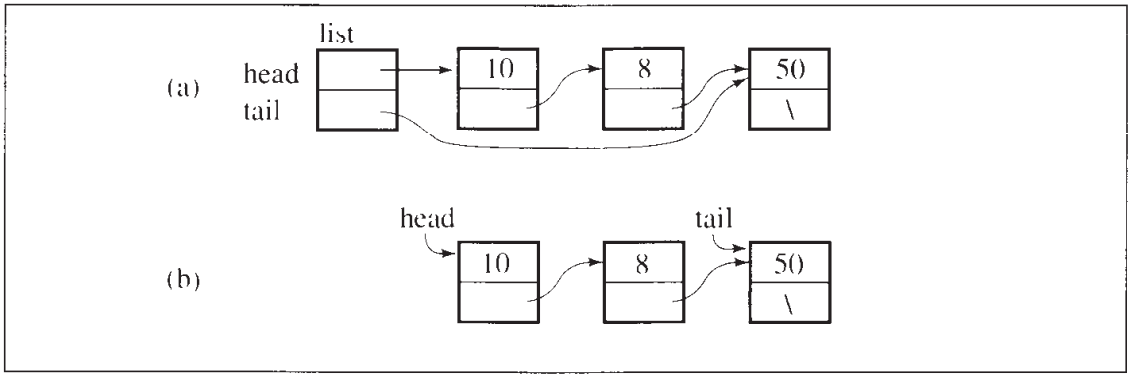
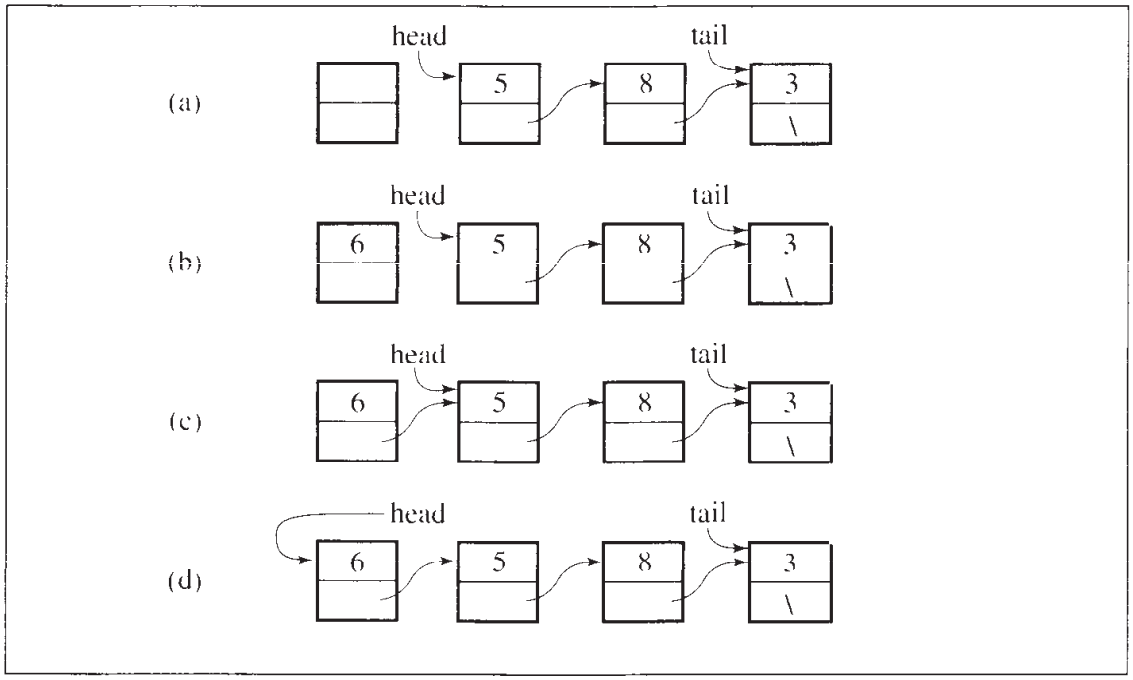


FIGURE 3.4 Inserting a new node at the beginning of a singly linked list.

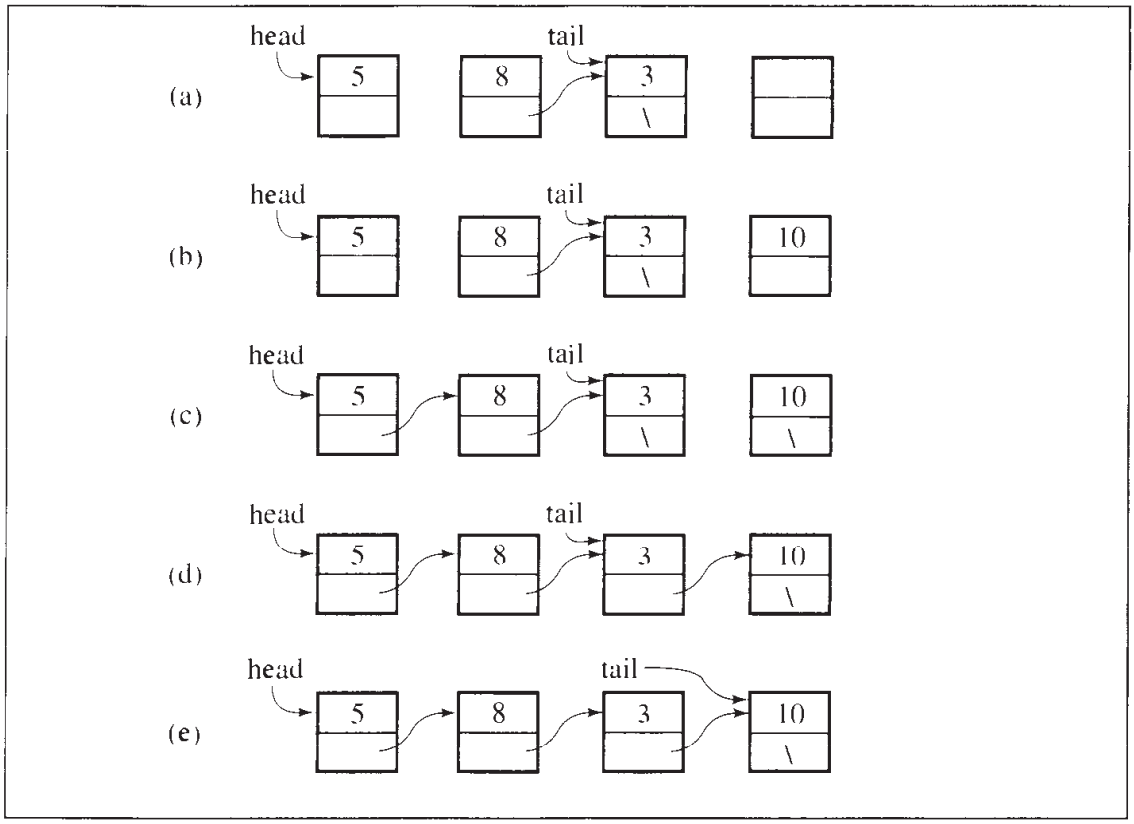


The four steps are executed by the member function `addToHead()` (Figure 3.2). The function executes the first three steps indirectly by calling the constructor `IntNode(el,head)`. The last step is executed directly in the function by assigning the address of the newly created node to `head`.

The member function `addToHead()` singles out one special case, namely, inserting a new node in an empty linked list. In an empty linked list, both `head` and `tail` are null; therefore, both become pointers to the only node of the new list. When inserting in a nonempty list, only `head` needs to be updated.

The process of adding a new node to the end of the list has five steps.

FIGURE 3.5 Inserting a new node at the end of a singly linked list.

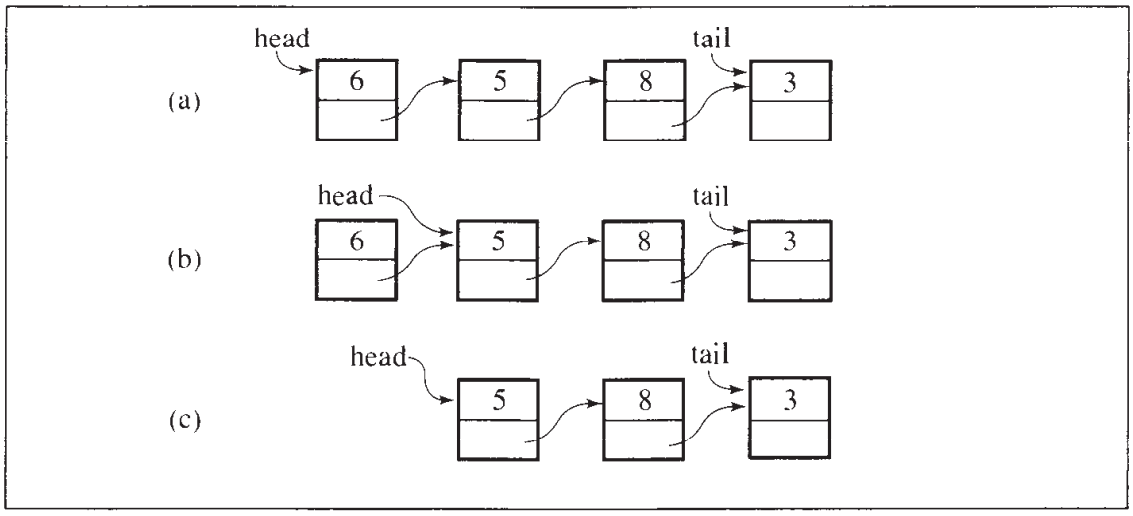


1. An empty node is created (Figure 3.5a).
2. The node's `info` member is initialized to an integer `el` (Figure 3.5b).
3. Because the node is being included at the end of the list, the `next` member is set to null (Figure 3.5c).
4. The node is now included in the list by making the `next` member of the last node of the list a pointer to the newly created node (Figure 3.5d).
5. The new node follows all the nodes of the list, but this fact has to be reflected in the value of `tail`, which now becomes the pointer to the new node (Figure 3.5e).

All these steps are executed in the `if` clause of `addToTail()` (Figure 3.2). The `else` clause of this function is executed only if the linked list is empty. If this case were not included, the program may crash because in the `if` clause we make an assignment to the `next` member of the node referred by `tail`. In the case of an empty linked list, it is a pointer to a nonexisting data member of a nonexisting node.

The process of inserting a new node at the beginning of the list is very similar to the process of inserting a node at the end of the list. This is so because the implementation of `IntSLList` uses two pointer members: `head` and `tail`. For this reason, both `addToHead()` and `addToTail()` can be executed in constant time $O(1)$; that is, regardless of the number of nodes in the list, the number of operations performed

FIGURE 3.6 Deleting a node at the beginning of a singly linked list.



by these two member functions does not exceed some constant number c. Note that because the `head` pointer allows us to have access to a linked list, the `tail` pointer is not indispensable; its only role is to have immediate access to the last node of the list. With this access, a new node can be easily added at the end of the list. If the `tail` pointer were not used, then adding a node at the end of the list would be more complicated because we would first have to reach the last node in order to attach a new node to it. This requires scanning the list and requires $O(n)$ steps to finish; that is, it is linearly proportional to the length of the list. The process of scanning lists is illustrated when discussing deletion of the last node.

3.1.2 Deletion

One deletion operation consists in deleting a node at the beginning of the list and returning the value stored in it. This operation is implemented by the member function `deleteFromHead()`. In this operation, the information from the first node is temporarily stored in a local variable `el`, and then `head` is reset so that what was the second node becomes the first node. In this way, the former first node can be deleted in constant time $O(1)$ (Figure 3.6).

Unlike before, there are now two special cases to consider. One case is when we attempt to remove a node from an empty linked list. If such an attempt is made, the program is very likely to crash, which we don't want to happen. The caller should also know that such an attempt is made to perform certain action. After all, if the caller expects a number to be returned from the call to `deleteFromHead()` and no number can be returned, then the caller may be unable to accomplish some other operations.

There are several ways to approach this problem. One way is to use an `assert` statement:

```
int IntSLList::deleteFromHead() {
    assert(!isEmpty()); // terminate the program if false;
    int el = head->info;
    . . . . . .
    return el;
}
```

The `assert` statement checks the condition `!isEmpty()`, and if the condition is false, the program is aborted. This is a crude solution because from the perspective of the caller the program can be continued even if no number is returned from `deleteFromHead()`.

Another solution is to throw an exception and catch it by the user as in:

```
int IntSLList::deleteFromHead() {
    if (isEmpty())
        throw("Empty");
    int el = head->info;
    . . . . . .
    return el;
}
```

The `throw` clause with the string argument is expected to have a matching `try-catch` clause in the caller (or caller's caller, etc.) also with the string argument which catches the exception as in

```
void f() {
    . . . . . .
    try {
        n = list.deleteFromHead();
        // do something with n;
    } catch(char *s) {
        cerr << "Error: " << s << endl;
    }
    . . . . . .
}
```

This solution gives the caller some control over the abnormal situation without making it lethal to the program as with the use of the `assert` statement. The user is responsible for providing an exception handler in the form of the `try-catch` clause, with the solution appropriate to the particular case. If the clause is not provided, then the program crashes when the exception is thrown. The function `f()` may only print a message that a list is empty when an attempt is made to delete a number from an empty list, another function `g()` may assign a certain value to n in such a case, and yet another function `h()` may find such a situation detrimental to the program and abort the program altogether.

The idea that the user is responsible for providing an action in the case of an exception is also presumed in the implementation given in Figure 3.2. The member function assumes that the list is not empty. To prevent the program from crashing, the

member function `isEmpty()` is added to the `IntSLList` class, and the user should use it as in:

```
if (!list.isEmpty())
     n = list.deleteFromHead();
else do not remove;
```

Note that including a similar `if` statement in `deleteFromHead()` does not solve the problem. Consider this code:

```
int IntSLList::deleteFromHead() {
    if (!isEmpty()) {          // if non-empty list;
         int el = head->info;
         . . . . . .
         return el;
    }
    else return 0;
}
```

If an `if` statement is added, then the `else` clause must also be added; otherwise, the program does not compile because "not all control paths return a value." But now, if 0 is returned, the caller does not know whether the returned 0 is the sign of failure or if it is a literal 0 retrieved from the list. To avoid any confusion, the caller must use an `if` statement to test whether the list is empty before calling `deleteFromHead()`. In this way, one `if` statement would be redundant.

To maintain uniformity in the interpretation of the return value, the last solution can be modified so that instead of returning an integer, the function returns the pointer to an integer:

```
int* IntSLList::deleteFromHead() {
    if (!isEmpty()) {          // if non-empty list;
         int *el = new int(head->info);
         . . . . . .
         return el;
    }
    else return 0;
}
```

where 0 in the `else` clause is the null pointer, not the number 0. In this case, the function call

```
n = *list.deleteFromHead();
```

results in a program crash if `deleteFromHead()` returns the null pointer.

Therefore, a test must be performed by the caller before calling `deleteFromHead()` to check whether `list` is empty or a pointer variable has to be used,

```
int *p = list.deleteFromHead();
```

and then a test is performed after the call to check whether `p` is null or not. In either case, this means that the `if` statement in `deleteFromHead()` is redundant.

The second special case is when the list has only one node to be removed. In this case, the list becomes empty, which requires setting `tail` and `head` to null.

The second deletion operation consists in deleting a node from the end of the list, and it is implemented as the member function `deleteFromTail()`. The problem is that after removing a node, `tail` should refer to the new tail of the list; that is, `tail` has to be moved backward by one node. But moving backward is impossible because there is no direct link from the last node to its predecessor. Hence, this predecessor has to be found by searching from the beginning of the list and stopping right before `tail`. This is accomplished with a temporary variable `tmp` used to scan the list within the `for` loop. The variable `tmp` is initialized to the head of the list, and then in each iteration of the loop, it is advanced to the next node. If the list is as in Figure 3.7a, then `tmp` first refers the head node holding number 6; after executing the assignment `tmp = tmp->next`, `tmp` refers to the second node (Figure 3.7b). After the second iteration and executing the same assignment, `tmp` refers to the third node (Figure 3.7c). Because this node is also the next to last node, the loop is exited, after which the last node is deleted (Figure 3.7d). Because `tail` is now pointing to a nonexisting node, it is immediately set to point to the next to last node currently pointed to by `tmp` (Figure 3.7e). To mark the fact that it is the last node of the list, the `next` member of this node is set to null (Figure 3.7f).

Note that in the `for` loop, a temporary variable is used to scan the list. If the loop were simplified to

```
for ( ; head->next != tail; head = head->next);
```

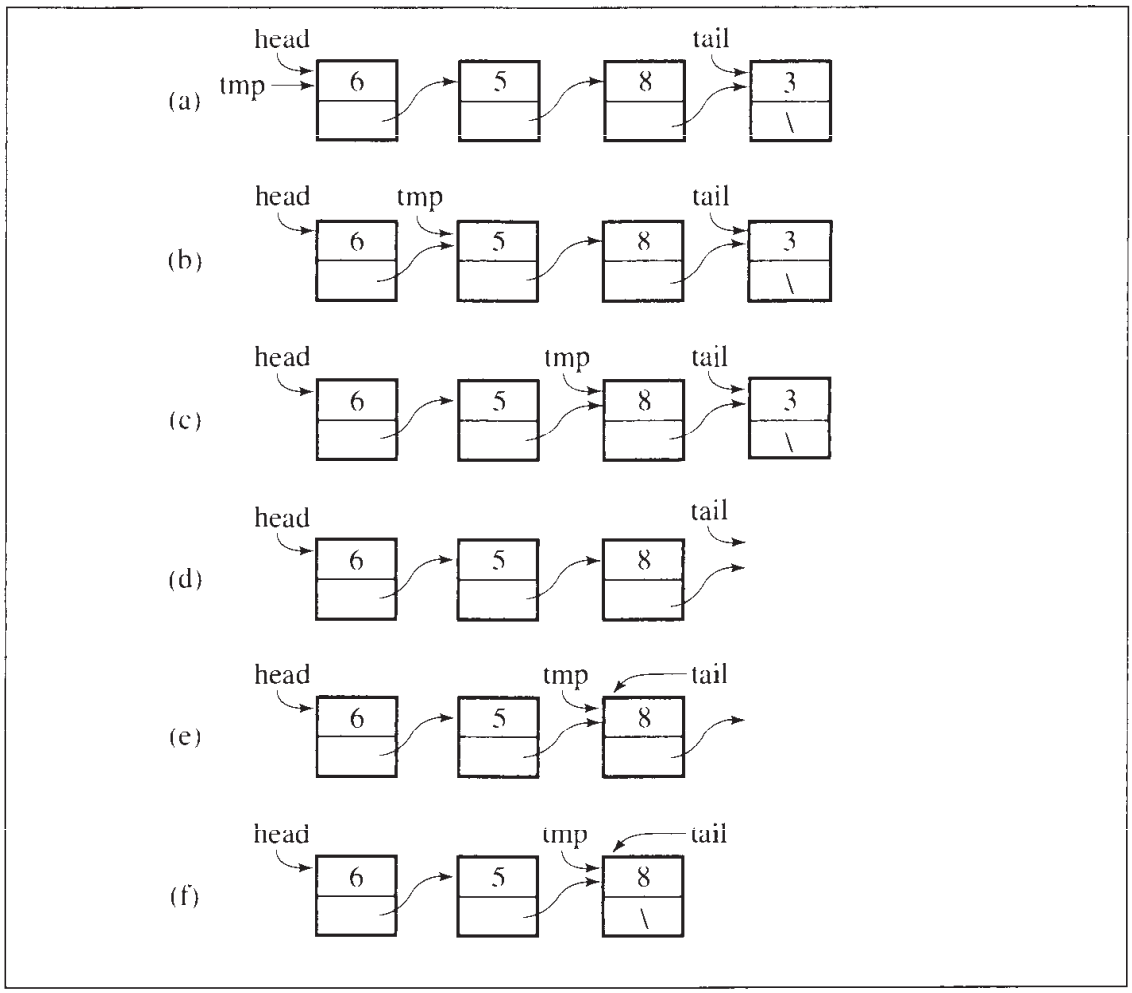
then the list is scanned only once, and the access to the beginning of the list is lost because `head` was permanently updated to the next to last node, which is about to become the last node. It is absolutely critical that, in cases such as this, a temporary variable is used so that the access to the beginning of the list is kept intact.

In removing the last node, the two special cases are the same as in `deleteFromHead()`. If the list is empty, then nothing can be removed, but what should be done in this case is decided in the user program just as in the case of `deleteFromHead()`. The second case is when a single-node list becomes empty after removing its only node, which also requires setting `head` and `tail` to null.

The most time-consuming part in `deleteFromTail()` is finding the next to last node performed by the `for` loop. It is clear that the loop performs $n - 1$ iterations in a list of n nodes, which is the main reason this member function takes $O(n)$ time to delete the last node.

The two discussed deletion operations remove a node from the head or from the tail (that is, always from the same position) and return an integer that happens to be in the node being removed. A different approach is when we want to delete a node that holds a particular integer regardless of the position of this node in the list. It may be right at the beginning, at the end, or anywhere inside the list. Briefly, a node has to be located first and then detached from the list by linking the predecessor of this node directly to its successor. Because we do not know where the node may be, the process of finding and deleting a node with a certain integer is much more complex than the deletion operations discussed so far. The member function `deleteNode()` (Figure 3.2) is an implementation of this process.

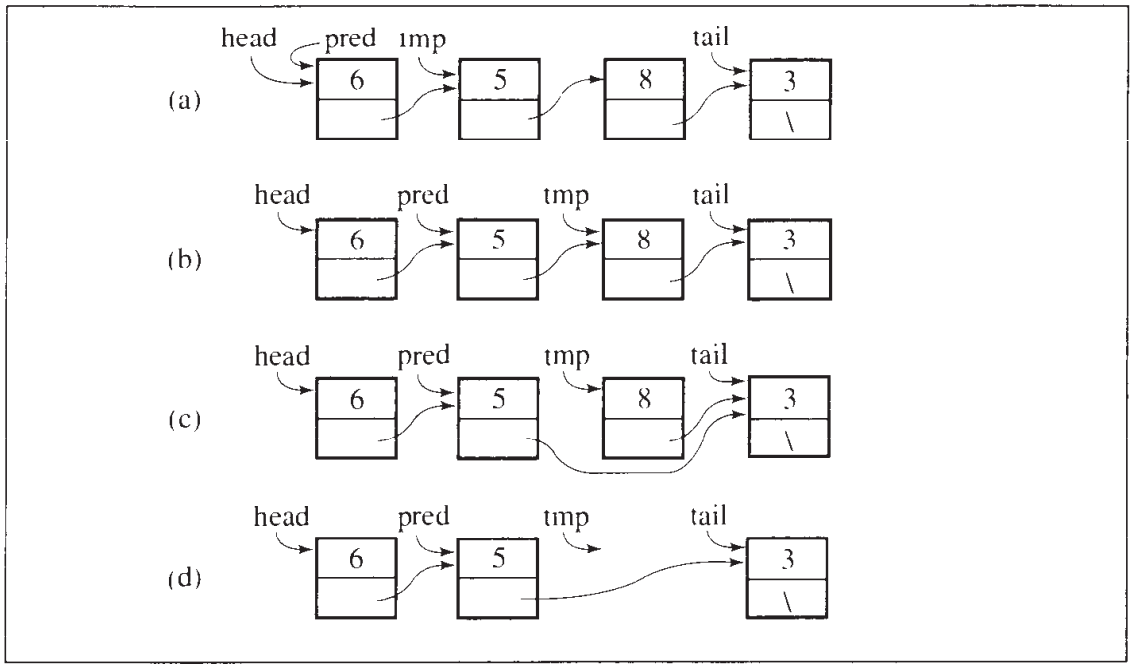
FIGURE 3.7 Deleting a node from the end of a singly linked list.



A node is removed from inside a list by linking its predecessor to its successor. But because the list has only forward links, the predecessor of a node is not reachable from the node. One way to accomplish the task is to find the node to be removed first by scanning the list and then scanning it again to find its predecessor. Another way is presented in `deleteNode()`, as shown in Figure 3.8. Assume that we want to delete a node that holds number 8. The function uses two pointer variables, `pred` and `tmp`, which are initialized in the `for` loop so that they point to the first and second nodes of the list, respectively (Figure 3.8a). Because the node `tmp` has number 5, the first iteration is executed in which both `pred` and `tmp` are advanced to the next nodes (Figure 3.8b). Because the condition of the `for` loop is now true (`tmp` points to the node with 8), the loop is exited and an assignment `pred->next = tmp->next` is executed (Figure 3.8c). This assignment effectively excludes the node with 8 from the list. The node is still accessible from variable `tmp`, and this access is used to return space occupied by this node to the pool of free memory cells by executing `delete` (Figure 3.8d).

The preceding paragraph discusses only one case. Here are the remaining cases:

FIGURE 3.8 Deleting a node from a singly linked list.



1. An attempt to remove a node from an empty list, in which case the function is immediately exited.
2. Deleting the only node from a one-node linked list: Both `head` and `tail` are set to null.
3. Removing the first node of the list with at least two nodes, which requires updating `head`.
4. Removing the last node of the list with at least two nodes, leading to the update of `tail`.
5. An attempt to delete a node with a number that is not in the list: Do nothing.

It is clear that the best case for `deleteNode()` is when the head node is to be deleted, which takes $O(1)$ time to accomplish. The worst case is when the last node needs to be deleted, which reduces `deleteNode()` to `deleteFromTail()` and to its $O(n)$ performance. What is the average case? It depends on how many iterations the `for` loop executes. Assuming that any node on the list has an equal chance to be deleted, the loop performs no iteration if it is the first node, one iteration if it is the second node, . . . , and finally $n - 1$ iterations if it is the last node. For a long sequence of deletions, one deletion requires on the average

$$\frac{0+1+\ldots+(n-1)}{n} = \frac{\frac{(n-1)n}{2}}{n} = \frac{n-1}{2}$$

That is, on the average, `deleteNode()` executes $O(n)$ steps to finish, just like in the worst case.

3.1.3 Search

The insertion and deletion operations modify linked lists. The searching operation scans an existing list to learn whether or not a number is in it. We implement this operation with the Boolean member function `isInList()`. The function uses a temporary variable `tmp` to go through the list starting from the head node. The number stored in each node is compared to the number being sought, and if the two numbers are equal, the loop is exited; otherwise, `tmp` is updated to `tmp->next` so that the next node can be investigated. After reaching the last node and executing the assignment `tmp = tmp->next`, `tmp` becomes null, which is used as an indication that the number `el` is not in the list. That is, if `tmp` is not null, the search was discontinued somewhere inside the list because `el` was found. That is why `isInList()` returns the result of comparison `tmp != null`: If `tmp` is not null, `el` was found and `true` is returned. If `tmp` is null, the search was unsuccessful and `false` is returned.

With reasoning similar to that used to determine the efficiency of `deleteNode()`, `isInList()` takes $O(1)$ time in the best case and $O(n)$ in the worst and average cases.

In the foregoing discussion, the operations on nodes have been stressed. However, a linked list is built for the sake of storing and processing information, not for the sake of itself. Therefore, the approach used in this section is limited in that the list can only store integers. If we wanted a linked list for float numbers or arrays of numbers, then a new class would have to be declared with a new set of member functions, all of them resembling the ones discussed here. However, it is more advantageous to declare such a class only once without deciding in advance what type of data will be stored in it. This can be done very conveniently in C++ with templates. To illustrate the use of templates for list processing, the next section uses them to define lists, although examples of list operations are still limited to lists that store integers.

3.2 Doubly Linked Lists

The member function `deleteFromTail()` indicates a problem inherent to singly linked lists. The nodes in such lists contain only pointers to the successors; therefore, there is no immediate access to the predecessors. For this reason, `deleteFromTail()` was implemented with a loop which allowed us to find the predecessor of `tail`. Although this predecessor is, so to speak, within sight, it is out of reach. We have to scan the entire list to stop right in front of `tail` to delete it. For long lists and for frequent executions of `deleteFromTail()`, this may be an impediment to swift list processing. To avoid this problem, the linked list is redefined so that each node in the list has two pointers, one to the successor and one to the predecessor. A list of this type is called a *doubly linked list* and is illustrated in Figure 3.9. Figure 3.10 contains a fragment of implementation for a generic `DoublyLinkedList` class.

Member functions for processing doubly linked lists are slightly more complicated than their singly linked lists counterparts because there is one more pointer

FIGURE 3.9 A doubly linked list.

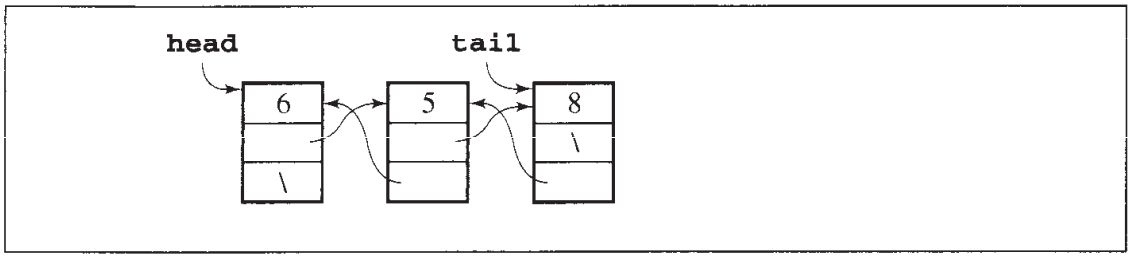


FIGURE 3.10 An implementation of a doubly linked list.

```
#ifndef DOUBLY_LINKED_LIST
#define DOUBLY_LINKED_LIST

#include <iostream.h>

template<class T>
class Node {
public:
    Node() {
        next = prev = 0;
    }
    Node(const T& el, Node *n = 0, Node *p = 0) {
        info = el; next = n; prev = p;
    }
    T info;
    Node *next, *prev;
};

template<class T>
class DoublyLinkedList {
public:
    DoublyLinkedList() {
        head = tail = 0;
    }
    void addToDLLTail(const T&);
    T deleteFromDLLTail();
    . . . . . .
protected:
    Node<T> *head, *tail;
};
```

Continues

FIGURE 3.10 *(continued)*

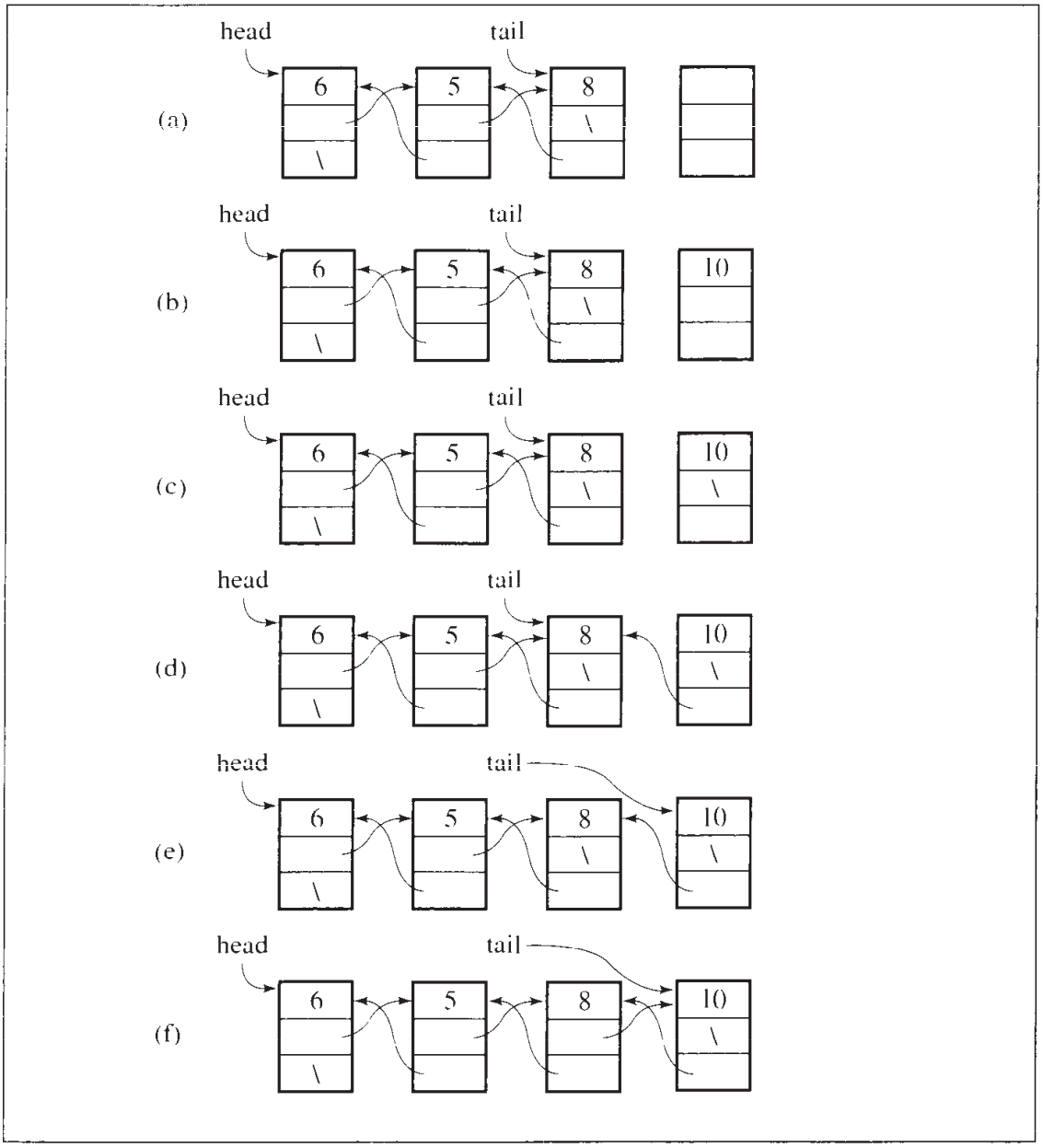
```
template<class T>
void DoublyLinkedList<T>::addToDLLTail(const T& el) {
    if (tail != 0) {
         tail = new Node<T>(el,0,tail);
         tail->prev->next = tail;
    }
    else head = tail = new Node<T>(el);
}
template<class T>
T DoublyLinkedList<T>::deleteFromDLLTail() {
    T el = tail->info;
    if (head == tail) { // if only one node in the list;
         delete head;
         head = tail = 0;
    }
    else {              // if more than one node in the list;
         tail = tail->prev;
         delete tail->next;
         tail->next = 0;
    }
    return el;
}
. . . . . .
#endif
```

member to be maintained. Only two functions are discussed: a function to insert a node at the end of the doubly linked list and a function to remove a node from the end (Figure 3.10).

To add a node to a list, the node has to be created, its data members properly initialized, and then the node needs to be incorporated into the list. Inserting a node at the end of a doubly linked list performed by `addToDLLTail()` is illustrated in Figure 3.11. The process is performed in six steps:

1. A new node is created (Figure 3.11a) and then its three data members are initialized:
2. the `info` member to the number `el` being inserted (Figure 3.11b),
3. the `next` member to null (Figure 3.11c),
4. and the `prev` member to the value of `tail` so that this member points to the last node in the list (Figure 3.11d). But now, the new node should become the last node; therefore,

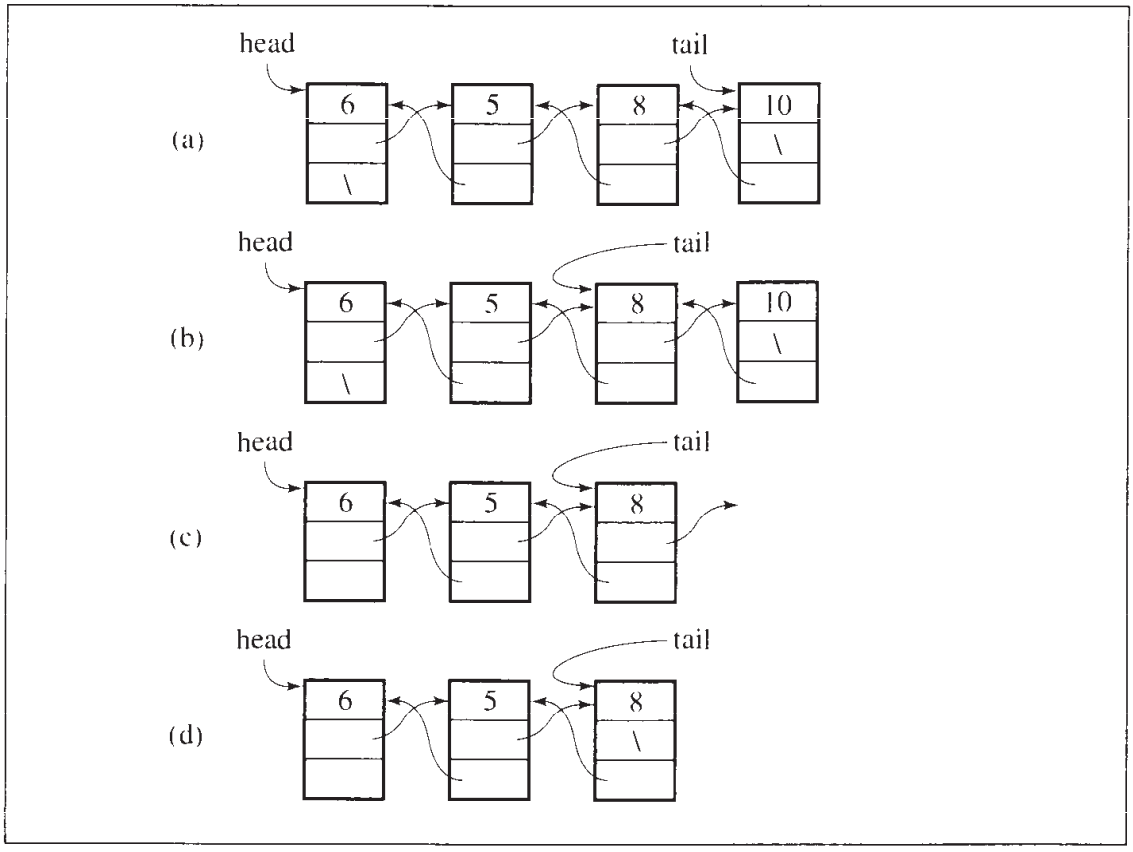
FIGURE 3.11 Adding a new node at the end of a doubly linked list.



5. `tail` is set to point to the new node (Figure 3.11e). But the new node is not yet accessible from its predecessor; to rectify this,
6. the `next` member of the predecessor is set to point to the new node (Figure 3.11f).

A special case concerns the last step. It is assumed in this step that the newly created node has a predecessor, so it accesses its `prev` member. It should be obvious that for an empty linked list, the new node is the only node in the list and that it has no predecessor. In this case, both `head` and `tail` refer to this node, and the sixth step is now setting `head` to point to this node. Note that step four—setting `prev` member to

FIGURE 3.12 Deleting a node from the end of a doubly linked list.



the value of `tail`—is executed properly because for an initially empty list, `tail` is null. Thus, null becomes the value of the `prev` member of the new node.

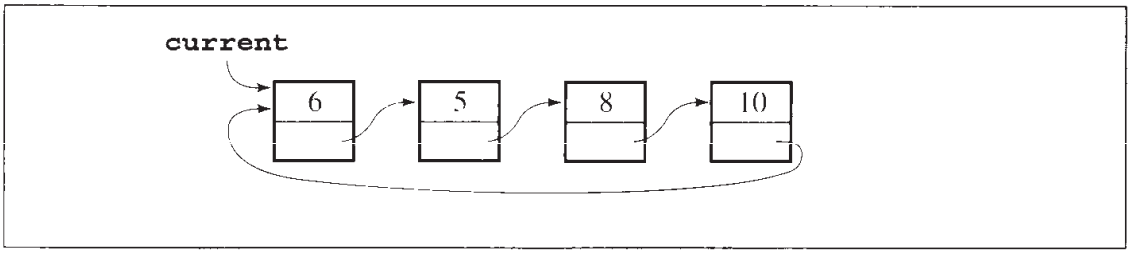
Deleting the last node from the doubly linked list is straightforward because there is direct access from the last node to its predecessor and no loop is needed to remove the last node. When deleting the last node from the list in Figure 3.12a, temporary variable `el` is set to the value in the node, then `tail` is set to its predecessor (Figure 3.12b), and the last and now redundant node is deleted (Figure 3.12c). In this way, the next to last node becomes the last node. The `next` member of the tail node is a dangling reference; therefore, `next` is set to null (Figure 3.12d). The last step is returning the copy of an object stored in the removed node.

An attempt to delete a node from an empty list may result in a program crash. Therefore, the user has to check whether the list is not empty before making an attempt of deleting the last node from it to extract information from the node. As for singly linked list's `deleteFromHead()`, the caller should have an `if` statement

```
if (!list.isEmpty())
     n = list.deleteFromDLLTail();
else do not remove;
```

Another special case is the deletion of the node from a single-node linked list. In this case, both `head` and `tail` are set to null.

FIGURE 3.13 A circular singly linked list.



Because of the immediate accessibility of the last node, both `addToDLLTail()` and `deleteFromDLLTail()` execute in constant time $O(1)$.

Functions for operating at the beginning of the doubly linked list are easily obtained from the two functions just discussed by changing `head` to `tail` and vice versa, changing `next` to `prev` and vice versa, and exchanging the order of parameters when executing `new`.

3.3 Circular Lists

In some situations, a *circular list* is needed in which nodes form a ring: The list is finite and each node has a successor. An example of such a situation is when several processes are using the same resource for the same amount of time, and we have to assure that no process accesses the resource before all other processes did. Therefore, all processes—let their numbers be 6, 5, 8, and 10 as in Figure 3.13—are put on a circular list accessible through the pointer `current`. After one node of the list is accessed and the process number is retrieved from the node to activate this process, `current` moves to the next node so that the next process can be activated the next time.

In an implementation of a circular singly linked list, we can use only one permanent pointer, `tail`, to the list even though operations on the list require access to the tail and its predecessor, the head. To that end, a linear singly linked list discussed in Section 3.1 uses two permanent pointers, `head` and `tail`.

Figure 3.14a shows a sequence of insertions at the front of the circular list, and Figure 3.14b illustrates insertions at the end of the list. As an example of a member function operating on such a list, we present a function to insert a node at the tail of a circular singly linked list in $O(1)$:

```
void addToTail(int el) {
   if (isEmpty()) {
        tail = new IntNode(el);
        tail->next = tail;
   }
   else {
        tail->next = new IntNode(el,tail->next);
        tail = tail->next;
}
```

FIGURE 3.14 Inserting nodes at the front of circular singly linked list (a) and at its end (b).

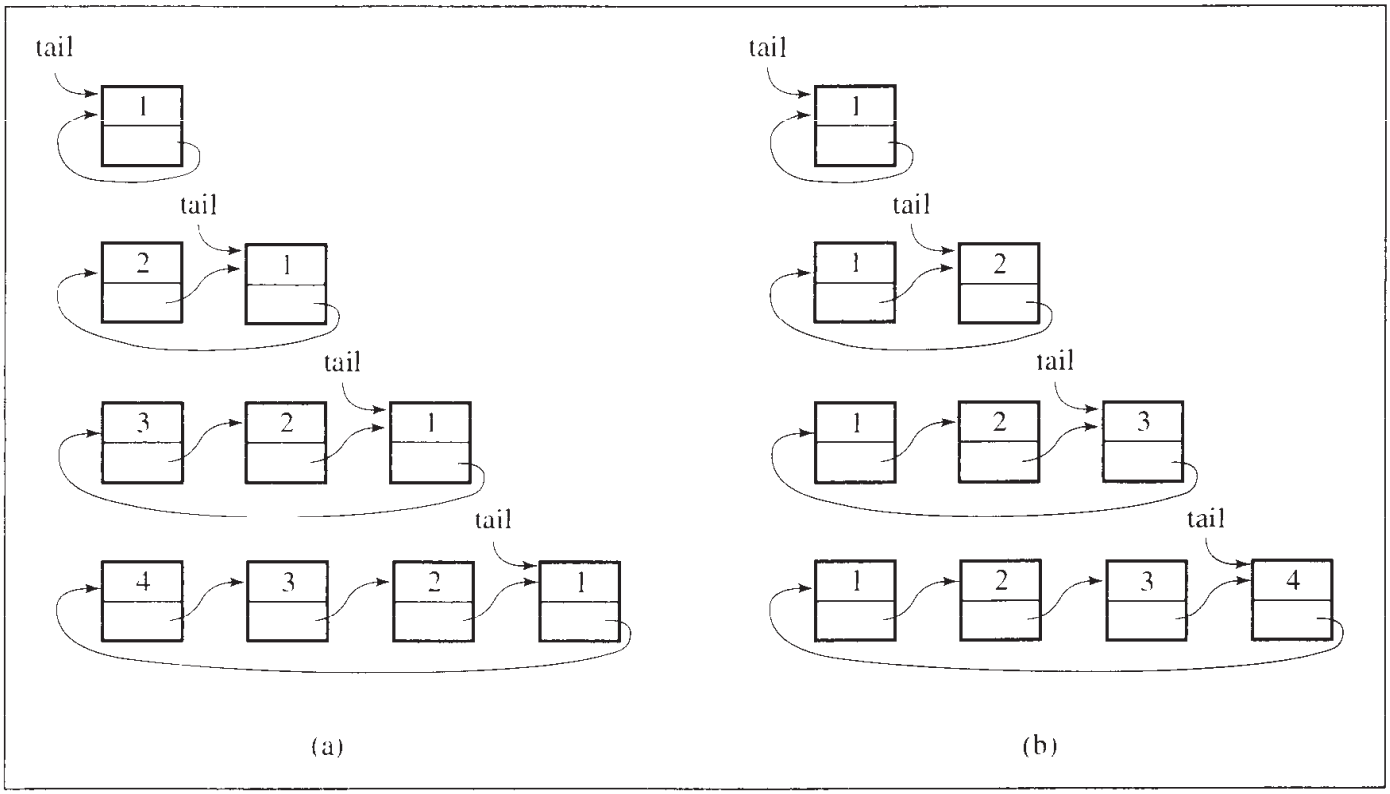
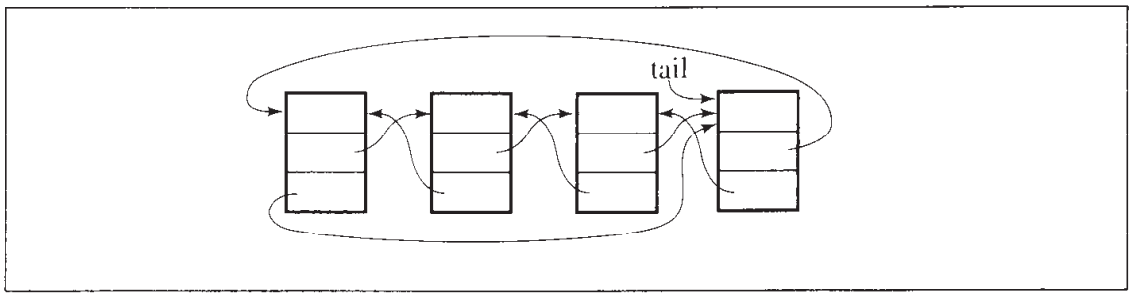
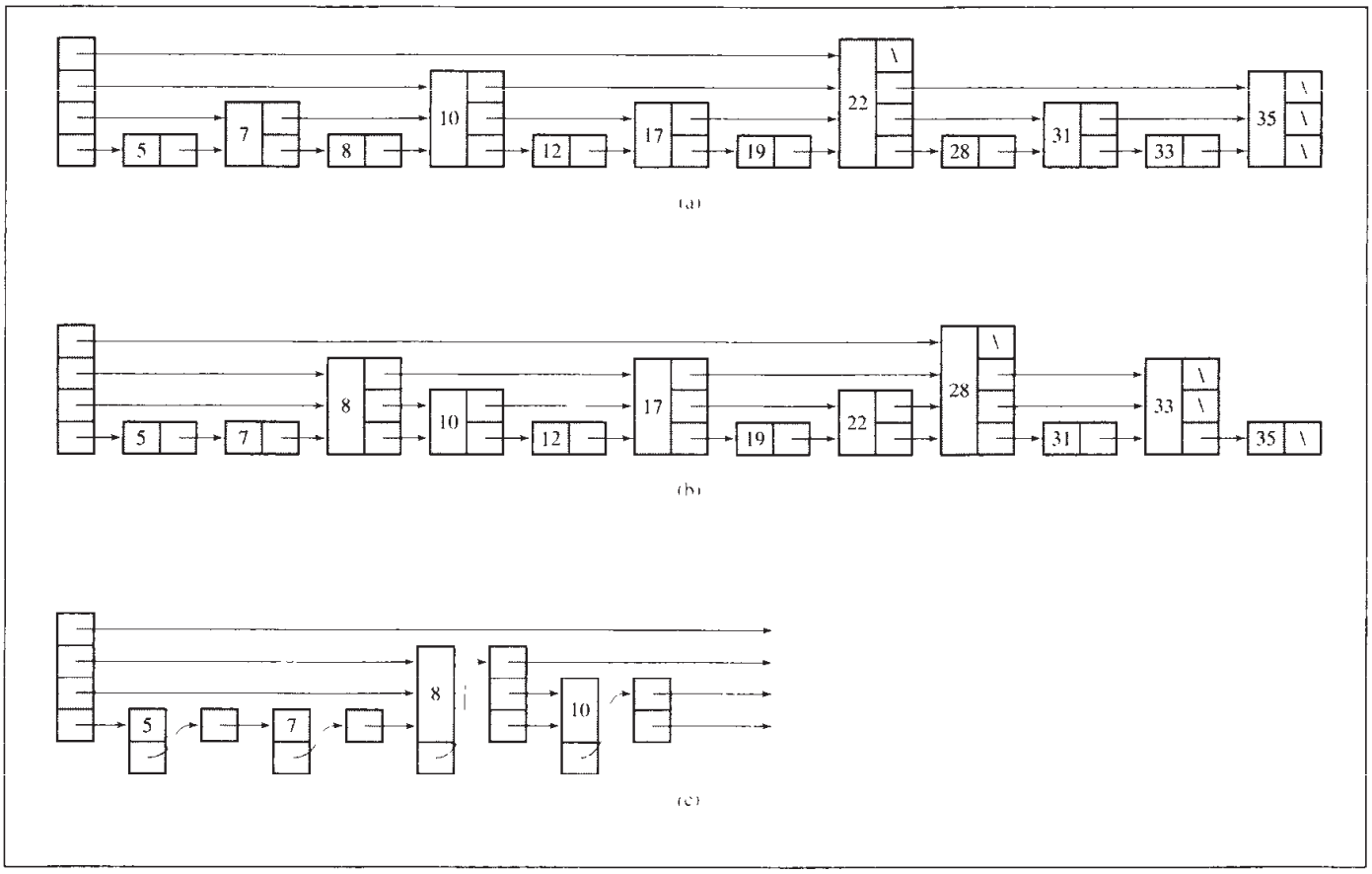


FIGURE 3.15 A circular doubly linked list.



The implementation just presented is not without its problems. A member function for deletion of the tail node requires a loop so that `tail` can be set after deletion to its predecessor. This makes this function delete the tail node in $O(n)$ time. Moreover, processing data in the reverse order (printing, searching, etc.) is not very efficient. To avoid the problem and still be able to insert and delete nodes at the front and at the end of the list without using a loop, a doubly linked circular list can be used. The list forms two rings: one going forward through `next` members and one going backward through `prev` members. Figure 3.15 illustrates such a list accessible through the last node. Deleting the node from the end of the list can be done easily

FIGURE 3.16 A skip list with (a) evenly and (b) unevenly spaced nodes of different levels; (c) the skip list with pointer nodes clearly shown.



10 and then again to 17. The last try is by starting the first-level sublist which begins in node 10; this sublist's first node has 12, the next number is 17, and since there is no lower level, the search is pronounced unsuccessful. Code for the searching procedure is given in Figure 3.17.

Searching appears to be efficient. However, the design of skip lists can lead to very inefficient insertion and deletion procedures. To insert a new element, all nodes following the node just inserted have to be restructured; the number of pointers and the values of pointers have to be changed. In order to retain some of the advantages which skip lists offer with respect to searching and avoid problems with restructuring the lists when inserting and deleting nodes, the requirement on the positions of nodes of different levels is now abandoned and only the requirement on the number of nodes of different levels is kept. For example, the list in Figure 3.16a becomes the list in Figure 3.16b: Both lists have six nodes with only one pointer, three nodes with two pointers, two nodes with three pointers, and one node with four pointers. The new list is searched exactly the same way as the original list. Inserting does not require list restructuring, and nodes are generated so that the distribution of the nodes on different levels is kept adequate. How can this be accomplished?

FIGURE 3.17 An implementation of a skip list.

```
//************************  genSkipL.h  ************************
//                      generic skip list class

const int maxLevel = 4;

template<class T>
class SkipListNode {
public:
    SkipListNode() {
    }
    T key;
    SkipListNode **next;
};

template<class T>
class SkipList {
public:
    SkipList();
    void choosePowers();
    int  chooseLevel();
    T* skipListSearch(const T&);
    void skipListInsert(const T&);
private:
    typedef SkipListNode<T> *nodePtr;
    nodePtr root[maxLevel];
    int powers[maxLevel];
};

template<class T>
SkipList<T>::SkipList() {
    for (int i = 0; i < maxLevel; i++)
        root[i] = 0;
}
template<class T>
void SkipList<T>::choosePowers() {
    powers[maxLevel-1] = (2 << (maxLevel-1)) - 1;  // 2^maxLevel - 1
    for (int i = maxLevel - 2, j = 0; i >= 0; i--, j++)
        powers[i] = powers[i+1] - (2 << j);        // 2^(j+1)
}
template<class T>
int SkipList<T>::chooseLevel() {
```

Continues

FIGURE 3.17 *(continued)*

```
    int i, r = rand() % powers[maxLevel-1] + 1;
    for (i = 1; i < maxLevel; i++)
        if (r < powers[i])
            return i-1; // return a level < the highest level;
    return i-1;         // return the highest level;
}
template<class T>
T* SkipList<T>::skipListSearch(const T& key) {
    nodePtr prev, curr;
    int lvl;                              // find the highest non-null
    for (lvl = maxLevel-1; lvl >= 0 && !root[lvl]; lvl--);  // level;
    prev = curr = root[lvl];
    while (1) {
        if (key == curr->key)                    // success if equal;
            return &curr->key;
        else if (key < curr->key) {              // if smaller, go down
            if (lvl == 0)                        // if possible,
                return 0;
            else if (curr == root[lvl])          // by one level
                curr = root[--lvl];              // starting from the
            else curr = *(prev->next + --lvl); // predecessor which
        }                                        // can be the root;
        else {                                   // if greater,
            prev = curr;                         // go to the next
            if (*(curr->next + lvl) != 0)        // non-null node
                curr = *(curr->next + lvl);      // on the same level
            else {                               // or to a list on a
                                                 // lower level;
                for (lvl--; lvl> = 0 && *(curr->next + lvl)==0; lvl--);
                if (lvl >= 0)
                    curr = *(curr->next + lvl);
                else return 0;
            }
        }
    }
}
template<class T>
void SkipList<T>::skipListInsert(const T& key) {
    nodePtr curr[maxLevel], prev[maxLevel], newNode;
    int lvl, i;
```

FIGURE 3.17 *(continued)*

```
    curr[maxLevel-1] = root[maxLevel-1];
    prev[maxLevel-1] = 0;
    for (lvl = maxLevel - 1; lvl >= 0; lvl--) {
        while (curr[lvl] && curr[lvl]->key < key) { // go to the next
            prev[lvl] = curr[lvl];              // if smaller;
            curr[lvl] = *(curr[lvl]->next + lvl);
        }
        if (curr[lvl] && curr[lvl]->key == key)       // don't include
            return;                                   // duplicates;
        if (lvl > 0)                                  // go one level down
            if (prev[lvl] == 0) {                     // if not the lowest
                curr[lvl-1] = root[lvl-1];            // level, using a link
                prev[lvl-1] = 0;                      // either from the root
            }
            else {                                    // or from the predecessor;
                curr[lvl-1] = *(prev[lvl]->next + lvl-1);
                prev[lvl-1] = prev[lvl];
            }
    }
    lvl = chooseLevel();        // generate randomly level for newNode;
    newNode = new SkipListNode<T>;
    newNode->next = new nodePtr[sizeof(nodePtr) * (lvl+1)];
    newNode->key  = key;
    for (i = 0; i <= lvl; i++) {          // initialize next fields of
        *(newNode->next + i) = curr[i];   // newNode and reset to newNode
        if (prev[i] == 0)                 // either fields of the root
            root[i] = newNode;            // or next fields of newNode's
        else *(prev[i]->next + i) = newNode;    // predecessors;
    }
}
```

Assume that *maxLevel* = 4. For 15 elements, the required number of one-pointer nodes is eight, two-pointer nodes is four, three-pointer nodes is two, and four-pointer nodes is one. Each time a node is inserted, a random number r between 1 and 15 is generated, and if $r < 9$, then a node of level one is inserted. If $r < 13$, a second-level node is inserted, if $r < 15$, it is a third-level node, and if $r = 15$, the node of level four is generated and inserted. If *maxLevel* = 5, then for 31 elements the correspondence between the value of r and the level of node is as follows:

r	Level of Node to Be Inserted
31	5
29–30	4
25–28	3
17–24	2
1–16	1

To determine such a correspondence between *r* and the level of node for any *maxLevel*, the function `choosePowers()` initializes the array `powers[]` by putting lower bounds for each range. For example, for *maxLevel* = 4, the array is [1 9 13 15], and for *maxLevel* = 5, it is [1 17 25 29 31]. `chooseLevel()` uses `powers[]` to determine the level of the node about to be inserted. Figure 3.17 contains the code for `choosePowers()` and `chooseLevel()`. Note that the levels range between 0 and *maxLevel*–1 (and not between 1 and *maxLevel*) so that the array indexes can be used as levels. For example, the first level is level zero.

But we also have to address the question of implementing a node. The easiest way is to make each node have *maxLevel* pointers, but this is wasteful. We need only as many pointers per one node as the level of the node requires. To accomplish this, the `next` member of each node is not a pointer to the next node, but to an array of pointer(s) to the next node(s). The size of this array is determined by the level of the node. The `SkipListNode` and a `SkipList` classes are declared, as in Figure 3.17. In this way, the list in Figure 3.16b is really a list whose first four nodes are shown in Figure 3.16c. Only now can an inserting procedure be implemented, as in Figure 3.17.

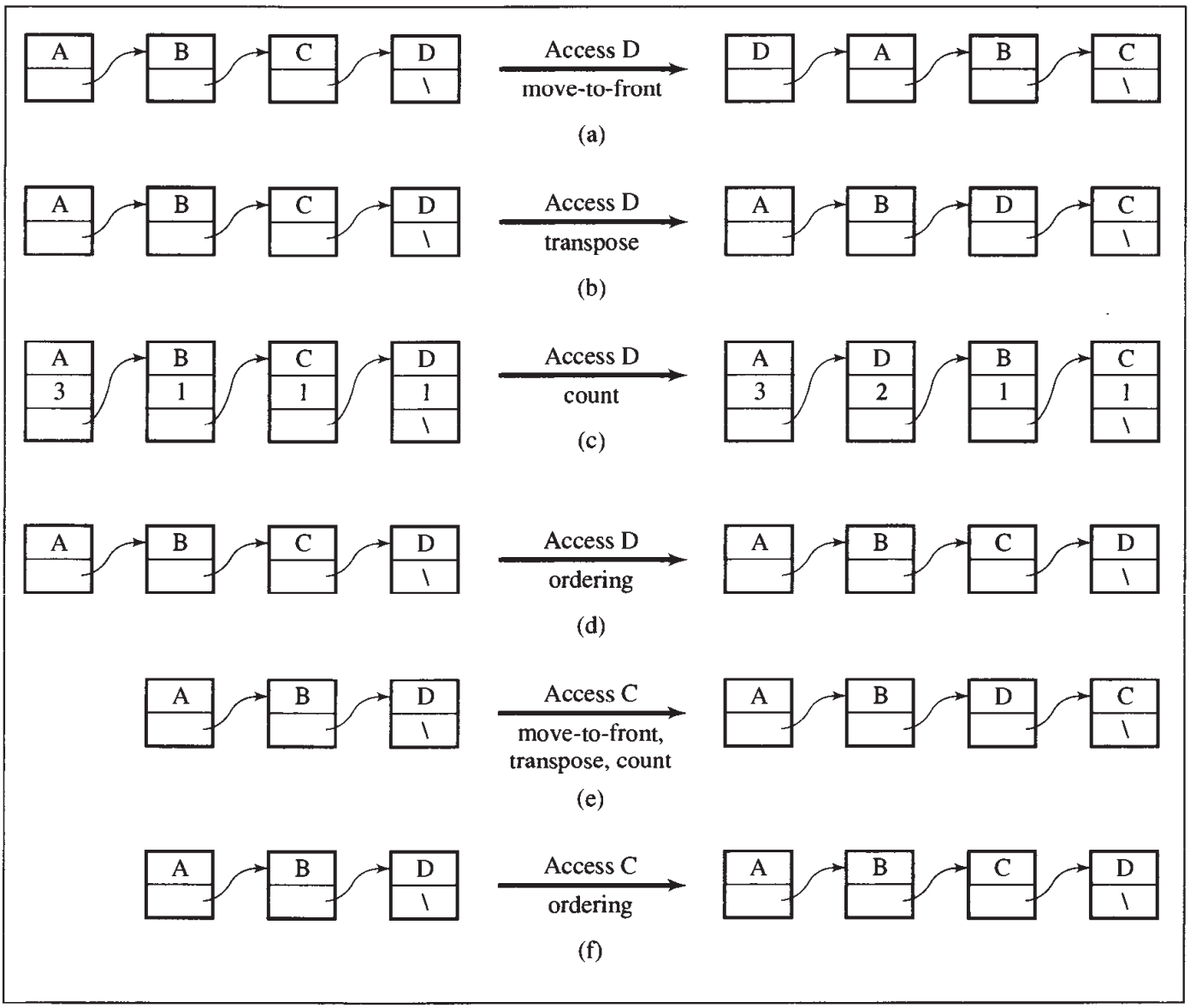
How efficient are skip lists? In the ideal situation, which is exemplified by the list in Figure 3.16a, the search time is $O(\lg n)$. In the worst situation, when all lists are on the same level, the skip list turns into a regular singly linked list, and the search time is $O(n)$. However, the latter situation is unlikely to occur; in the random skip list, the search time is of the same order as the best case, that is, $O(\lg n)$. This is an improvement over the efficiency of search in regular linked lists. It also turns out that skip lists fare extremely well in comparison with more sophisticated data structures, such as self-adjusting trees or AVL trees (cf. Sections 6.7.2, 6.8), and therefore they are a viable alternative to these data structures (see also the table in Figure 3.20).

3.5 Self-Organizing Lists

The introduction of skip lists was motivated by the need to speed up the searching process. Although singly and doubly linked lists require sequential search to locate an element or to see that it is not in the list, we can improve the efficiency of the search by dynamically organizing the list in a certain manner. This organization depends on the configuration of data; thus, the stream of data requires reorganizing the nodes already on the list. There are many different ways to organize the lists, and this section describes four of them.

1. *Move-to-front method.* After the desired element is located, put it at the beginning of the list.

FIGURE 3.18 Accessing an element on a linked list and changes on the list depending on the self-organization technique applied: (a) move-to-front method, (b) transpose method, (c) count method, and (d) ordering method, in particular, alphabetical ordering which leads to no change. In the case when the desired element is not in the list, (e) the first three methods add a new node with this element at the end of the list and (f) the ordering method maintains an order on the list.



2. *Transpose method.* After the desired element is located, swap it with its predecessor unless it is at the head of the list.
3. *Count method.* Order the list by the number of times elements are being accessed.
4. *Ordering method.* Order the list using certain criteria natural for information under scrutiny.

In the first three methods, new information is stored in a node added to the end of the list (Figure 3.18e); in the fourth method, new information is stored in a node inserted somewhere in the list to maintain the order of the list (Figure 3.18f). An

FIGURE 3.19 Processing the stream of data, A C B C D A D A C A C C E E, by different methods of organizing linked lists. Linked lists are presented in an abbreviated form; for example, the transformation shown in Figure 3.18a is abbreviated as transforming list A B C D into list D A B C.

element searched for	plain	move-to-front	transpose	count	ordering
A:	A	A	A	A	A
C:	AC	AC	AC	AC	AC
B:	ACB	ACB	ACB	ACB	ABC
C:	ACB	CAB	CAB	CAB	ABC
D:	ACBD	CABD	CABD	CABD	ABCD
A:	ACBD	ACBD	ACBD	CABD	ABCD
D:	ACBD	DACB	ACDB	DCAB	ABCD
A:	ACBD	ADCB	ACDB	ADCB	ABCD
C:	ACBD	CADB	CADB	CADB	ABCD
A:	ACBD	ACDB	ACDB	ACDB	ABCD
C:	ACBD	CADB	CADB	ACDB	ABCD
C:	ACBD	CADB	CADB	CADB	ABCD
E:	ACBDE	CADBE	CADBE	CADBE	ABCDE
E:	ACBDE	ECADB	CADEB	CAEDB	ABCDE

example of searching for elements in a list organized by these different methods is shown in Figure 3.19.

With the first three methods, we try to locate the elements most likely to be looked for near the beginning of the list, most explicitly with the move-to-front method and most cautiously with the transpose method. The ordering method already uses some properties inherent to the information stored in the list. For example, if we are storing nodes pertaining to people, then the list can be organized alphabetically by the name of the person or the city or in ascending or descending order using, say, birthday or salary. This is particularly advantageous when searching for information which is not in the list, since the search can terminate without scanning the entire list. Searching all the nodes of the list, however, is necessary in such cases using the other three methods. The count method can be subsumed in the category of the ordering methods if frequency is part of the information. In many cases, however, the count itself is an additional piece of information required solely to maintain the list; hence, it may to be considered "natural" to the information at hand.

Analyses of the efficiency of these methods customarily compare their efficiency to that of *optimal static ordering*. With this ordering, all the data are already ordered by the frequency of their occurrence in the body of data so that the list is used only for searching, not for inserting new items. Therefore, this approach requires two passes through the body of data, one to build the list and another to use the list for search alone.

To experimentally measure the efficiency of these methods, the number of all actual comparisons was compared to the maximum number of possible comparisons. The latter number is calculated by adding the lengths of the list at the moment of processing each element. For example, in the table in Figure 3.19, the body of data contains 14 letters, 5 of them being different, which means that 14 letters were processed. The length of the list before processing each letter is recorded, and the result, 0 + 1 + 2 + 3 + 3 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 5 = 46, is used to compare the number of all made comparisons to this combined length. In this way, we know what percentage of the list was scanned during the entire process. For all the list organizing methods except optimal ordering, this combined length is the same; only the number of comparisons can change. For example, when using the move-to-front technique for the data in the table in Figure 3.19, 33 comparisons were made, which is 71.7% when compared to 46. The latter number gives the worst possible case, the combined length of intermediate lists every time all the nodes in the list are looked at. Plain search, with no reorganization, required only 30 comparisons, which is 65.2%.

These samples are in agreement with theoretical analyses which indicate that count and move-to-front methods are, in the long run, at most twice as costly as the optimal static ordering; the transpose method approaches, in the long run, the cost of the move-to-front method. In particular, with amortized analysis, it can be established that the cost of accessing a list element with the move-to-front method is at most twice the cost of accessing this element on the list that uses optimal static ordering.

In a proof of this statement, the concept of inversion is used. For two lists containing the same elements, an inversion is defined to be a pair of elements (x, y) such that on one list x precedes y and on the other list y precedes x. For example, the list (*C, B, D, A*) has four inversions with respect to list (*A, B, C, D*): (*C, A*), (*B, A*), (*D, A*), and (*C, B*). Define the amortized cost to be the sum of actual cost and the difference between the number of inversions before accessing an element and after accessing it,

$$amCost(x) = cost(x) + (inversionsBeforeAccess(x) - inversionsAfterAccess(x))$$

To assess this number, consider an optimal list OL = (*A, B, C, D*) and a move-to-front list MTF = (*C, B, D, A*). The access of elements usually changes the balance of inversions. Let *displaced*(x) be the number of elements preceding x in MTF but following x in OL. For example, *displaced*(*A*) = 3, *displaced*(*B*) = 1, *displaced*(*C*) = 0, and *displaced*(*D*) = 0. If $pos_{MTF}(x)$ is the current position of x in MTF, then $pos_{MTF}(x) - 1 - displaced(x)$ is the number of elements preceding x in both lists. It is easy to see that for *D* this number equals 2, and for the remaining elements it is 0. Now, accessing an element x and moving it to the front of MTF creates $pos_{MTF}(x) - 1 - displaced(x)$ new inversions and removes *displaced*(x) other inversions so that the amortized time to access x is

$$amCost(x) = pos_{MTF}(x) + pos_{MTF}(x) - 1 - displaced(x) - displaced(x) =$$
$$2(pos_{MTF}(x) - displaced(x)) - 1$$

where $cost(x) = pos_{MTF}(x)$. Accessing A transforms MTF = (C, B, D, A) into (A, C, B, D) and $amCost(A) = 2(4 - 3) - 1 = 1$. For B, the new list is (B, C, D, A) and $amCost(B) = 2(2 - 1) - 1 = 1$. For C, the list does not change and $amCost(C) = 2(1 - 0) - 1 = 1$. Finally, for D, the new list is (D, C, B, A) and $amCost(D) = 2(3 - 0) - 1 = 5$. However, the number of common elements preceding x on the two lists cannot exceed the number of all the elements preceding x on OL; therefore, $pos_{MTF}(x) - 1 - displaced(x) \leq pos_{OL}(x) - 1$, so that

$$amCost(x) \leq 2pos_{OL}(x) - 1$$

The amortized cost of accessing an element x in MTF is in excess of $pos_{OL}(x) - 1$ units to its actual cost of access on OL. This excess is used to cover an additional cost of accessing elements in MTF for which $pos_{MTF}(x) > pos_{OL}(x)$, that is, elements that require more accesses on MTF than on OL.

It is important to stress that the amortized costs of single operations are meaningful in the context of sequences of operations. A cost of an isolated operation may seldom equal its amortized cost; however, in a sufficiently long sequence of accesses, each access on the average takes at most $2pos_{OL}(x) - 1$ time.

Figure 3.20 contains sample runs of the self-organizing lists. The first two columns of numbers refer to files containing programs, and the remaining columns refer to files containing English text. Except for alphabetical ordering, all methods improve their efficiency with the size of the file. The move-to-front and count methods are almost the same in their efficiency, and both outperform the transpose, plain, and ordering methods. The poor performance for smaller files is due to the fact that all of the methods are busy including new words to the lists, which requires an exhaustive search of the lists. Later, the methods concentrate on organizing the lists to reduce the number of searches. The table in Figure 3.20 also includes data for a skip list. There is an overwhelming difference between the skip list's efficiency compared to the other methods. However, keep in mind that in the table in Figure 3.20, only comparisons of data are included with no indication of the other operations needed for execution of the analyzed methods. In particular, there is no indication of how many pointers are used and relinked, which, when included, may make the difference between various methods less dramatic.

These sample runs show that for lists of modest size, the linked list suffices. With the increase in the amount of data and in the frequency with which they have to be accessed, more sophisticated methods and data structures need to be used.

3.6 Sparse Tables

In many applications, the choice of a table seems to be the most natural one, but space considerations may preclude this choice. This is particularly true if only a small fraction of the table is actually used. A table of this type is called a *sparse table* since the

FIGURE 3.20 Measuring the efficiency of different methods using formula (number of data comparison)/(combined length) expressed in percentages.

Different Words/ All Words	149/423	550/2847	156/347	609/1510	1163/5866	2013/23065
Optimal	26.4	17.6	28.5	24.5	16.2	10.0
Plain	71.2	56.3	70.3	67.1	51.7	35.4
Move-to-Front	49.5	31.3	61.3	54.5	30.5	18.4
Transpose	69.5	53.3	68.8	66.1	49.4	32.9
Count	51.6	34.0	61.2	54.7	32.0	19.8
Alphabetical Order	45.6	55.7	50.9	48.0	50.4	50.0
Skip List	12.3	5.5	15.1	6.6	4.8	3.8

table is populated sparsely by data and most of its cells are empty. In this case, the table can be replaced by a system of linked lists.

As an example, consider the problem of storing grades for all students in a university for a certain semester. Assume that there are 8000 students and 300 classes. A natural implementation is a two-dimensional array *grades* where student numbers are indexes of the columns and class numbers the indexes of the rows (see Figure 3.21). An association of student names and numbers is represented by the one-dimensional array *students* and an association of class names and numbers by the array *classes*. The names do not have to be ordered. If order is required, then another array can be used where each array element is occupied by a record with two fields, name and number,[1] or the original array can be sorted each time an order is required. This, however, leads to the constant reorganization of *grades* and is not recommended.

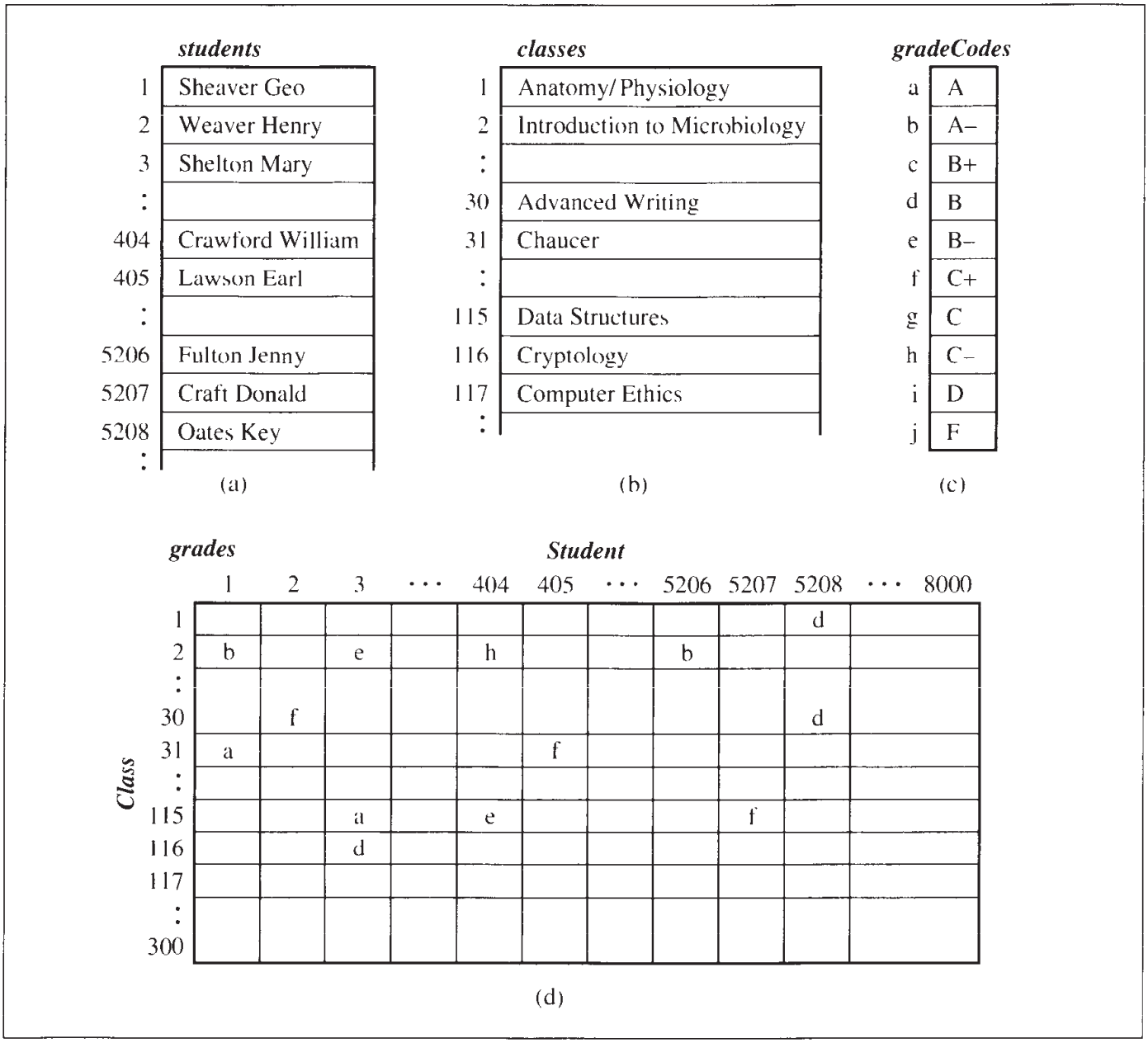
Each cell of *grades* stores a grade obtained by each student after finishing a class. If signed grades such as A–, B+, or C+ are used, then two bytes are require to store each grade. To reduce the table size by one-half, the array *gradeCodes* in Figure 3.21c associates each grade with a letter which requires only one byte of storage.

The entire table (Figure 3.21d) occupies 8000 students · 300 classes · 1 byte = 2.4 million bytes. This table is very large but is sparsely populated by grades. Assuming that, on the average, students take four classes a semester, each column of the table has only four cells occupied by grades, and the rest of the cells, 296 cells or 98.7%, are unoccupied and wasted.

A better solution is to use two 2-dimensional arrays. *classesTaken* represents all the classes taken by every student and *studentsInClasses* represents all students participating in each class (see Figure 3.22). A cell of each table is an object with two data

[1]This is called an *index-inverted table.*

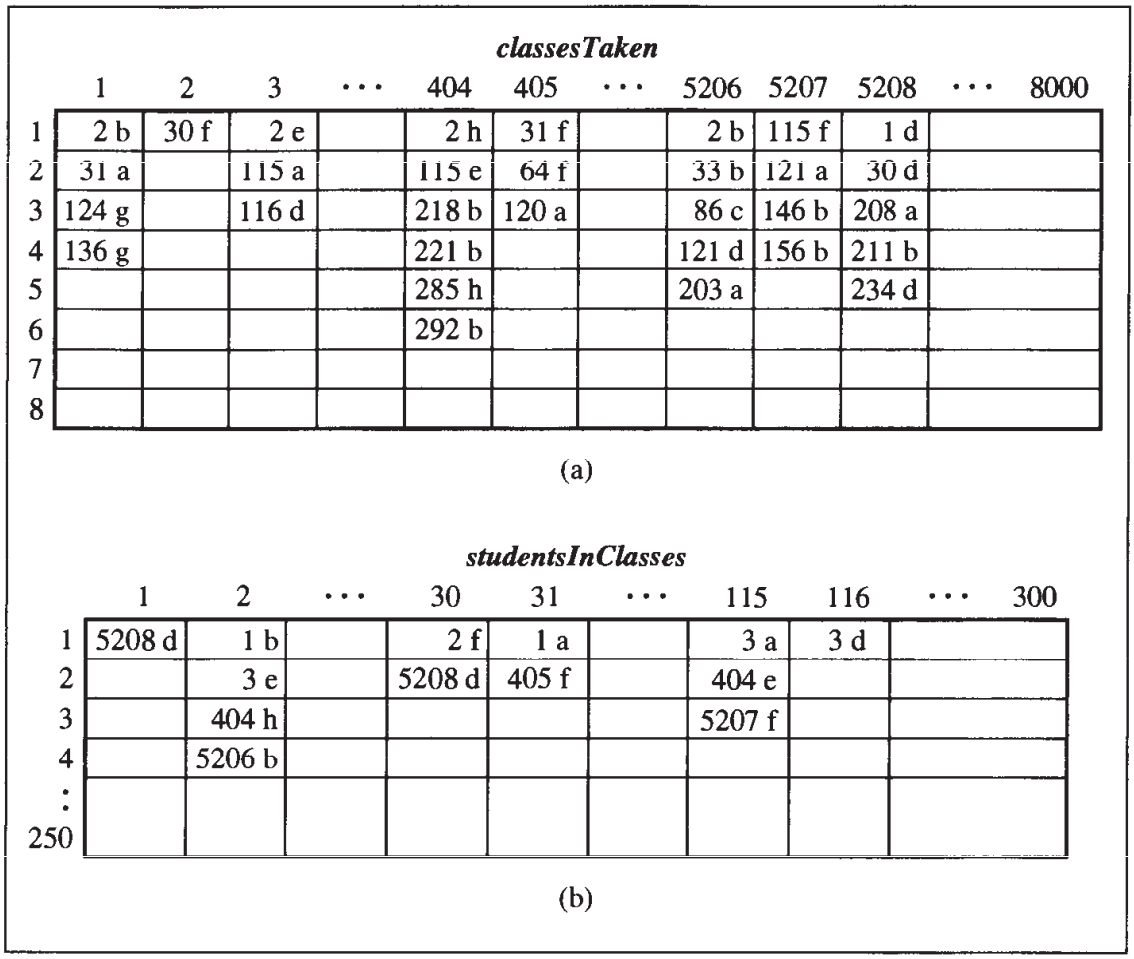
FIGURE 3.21 Arrays and sparse table used for storing student grades.



members: a student or class number and a grade. We assume that a student can take at most eight classes and that there can be at most 250 students signed up for a class. We need two arrays, since with only one array it is very time-consuming to produce lists. For example, if only *classesTaken* is used, then printing a list of all students taking a particular class requires an exhaustive search of *classesTaken*.

Assume that the computer on which this program is being implemented requires two bytes to store an integer. With this new structure, three bytes are needed for each cell. Therefore, *classesTaken* occupies 8000 students · 8 classes · 3 bytes = 192,000 bytes, *studentsInClasses* occupies 300 classes · 250 students · 3 bytes = 225,000 bytes, and both tables require a total of 417,000 bytes, less than one-fifth the number of bytes required for the sparse table in Figure 3.21.

FIGURE 3.22 Two-dimensional arrays for storing student grades.



classesTaken

	1	2	3	...	404	405	...	5206	5207	5208	...	8000
1	2 b	30 f	2 e		2 h	31 f		2 b	115 f	1 d		
2	31 a		115 a		115 e	64 f		33 b	121 a	30 d		
3	124 g		116 d		218 b	120 a		86 c	146 b	208 a		
4	136 g				221 b			121 d	156 b	211 b		
5					285 h			203 a		234 d		
6					292 b							
7												
8												

(a)

studentsInClasses

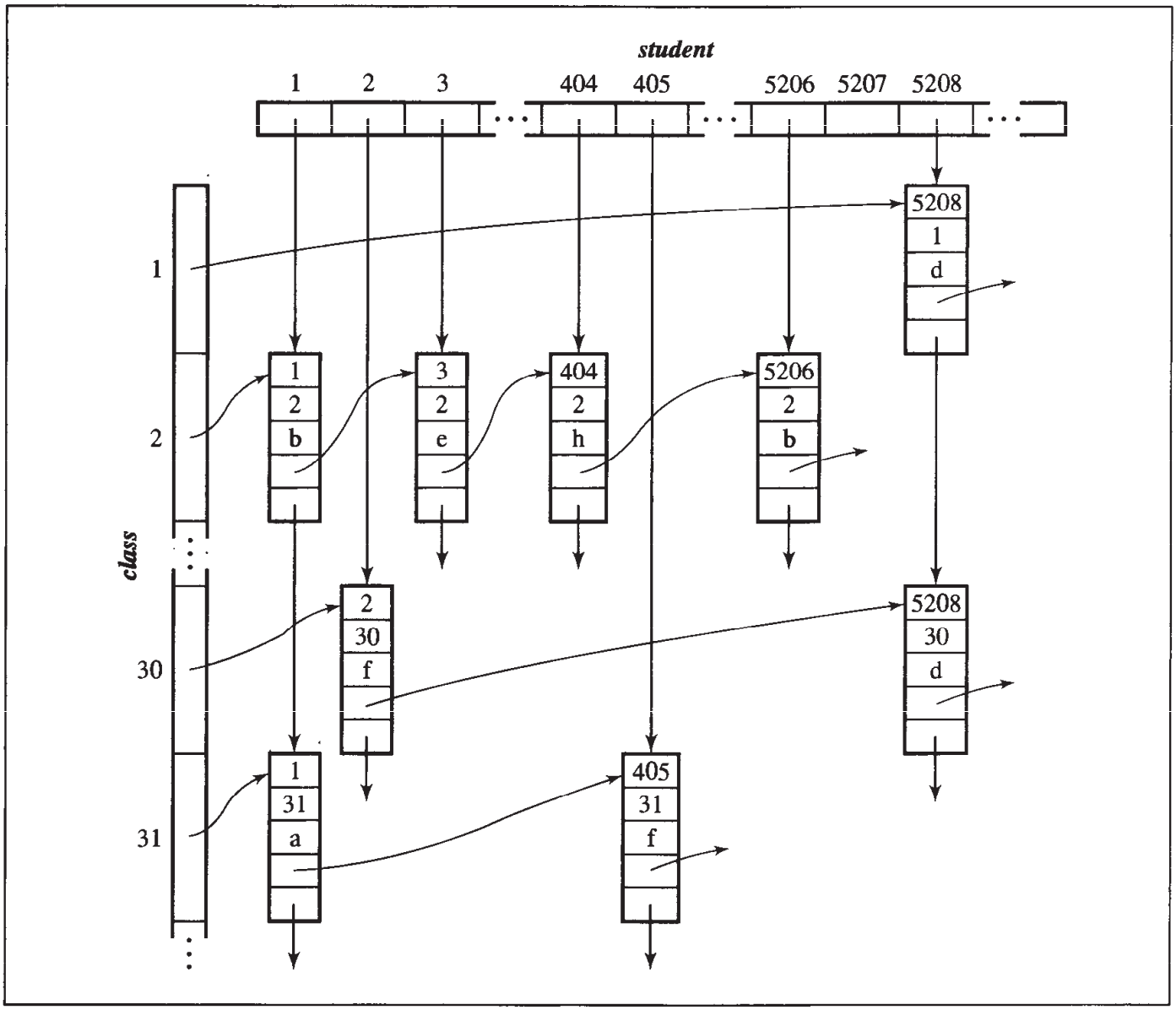
	1	2	...	30	31	...	115	116	...	300
1	5208 d	1 b		2 f	1 a		3 a	3 d		
2		3 e		5208 d	405 f		404 e			
3		404 h					5207 f			
4		5206 b								
:										
250										

(b)

Although this is a much better implementation than before, it still suffers from a wasteful use of space; seldom if ever will both arrays be full since most classes have fewer than 250 students and most students take fewer than eight classes. This structure is also inflexible: If a class can be taken by more than 250 students, a problem occurs which has to be circumvented in an artificial way. One way is to create a nonexistent class which holds students for the overflowing class. Another way is to recompile the program with a new table size, which may not be practical at a future time. Another more flexible solution is needed that uses space frugally.

Two one-dimensional arrays of linked lists can be used as in Figure 3.23. Each cell of the array *class* is a pointer to a linked list of students taking a class, and each cell of the array *student* indicates a linked list of classes taken by a student. The linked lists contain nodes of five data members: student number, class number, grade, a pointer to the next student, and a pointer to the next class. Assuming that each pointer requires only two bytes, one node occupies nine bytes, and the entire structure can be stored in 8000 students · 4 classes (on the average) · 9 bytes = 288,000 bytes, which is approximately 10% of the space required for the first implementation and about 70% of the space of the second. No space is used unnecessarily, there is no restriction imposed on the number of students per class, and the lists of students taking a class can be printed immediately.

FIGURE 3.23 Student grades implemented using linked lists.



3.7 LISTS IN THE STANDARD TEMPLATE LIBRARY

The list sequence container is an implementation of various operations on the nodes of a linked list. The STL implements a list as a generic doubly linked list with pointers to the head and to the tail. An instance of such a list that stores integers is presented in Figure 3.9.

The class `list` can be used in a program only if it is included with the instruction

```
#include <list>
```

The member functions included in the list container are presented in Figure 3.24.

FIGURE 3.24 An alphabetical list of member functions in the class `list`.

Member Function	Action and Return Value
`void assign(first, last)`	remove all the nodes in the list and insert in it the elements from the range indicated by iterators `first` and `last`
`void assign(n, el = T())`	remove all the nodes in the list and insert in it n copies of `el` (if `el` is not provided, a default constructor `T()` is used)
`T& back()`	return the element in the last node of the list
`const T& back() const`	return the element in the first node of the list
`iterator begin()`	return an iterator that references the first node of the list
`const_iterator begin() const`	return a `const` iterator that references the first node of the list
`void clear()`	remove all the nodes in the list
`bool empty() const`	return `true` if the list includes no nodes and false otherwise
`iterator end()`	return an iterator that is past the last node of the list
`const_iterator end() const`	return a `const` iterator that is past the last node of the list
`iterator/void erase(i)`	remove the node referenced by iterator `i`
`iterator/void erase(first, last)`	remove the nodes in the range indicated by iterators `first` and `last`
`T& front()`	return the element in the first node of the list
`const T& front() const`	return the element in the first node of the list
`iterator insert(i, el = T())`	insert `el` before the node referenced by iterator `i` and return iterator referencing the new node
`void insert(i, n, el)`	insert n copies of `el` before the node referenced by iterator `i`
`void insert(i, first, last)`	insert elements from location referenced by `first` to location referenced by `last` before the node referenced by iterator `i`
`list()`	construct an empty list
`list(n, el = T())`	construct a list with n copies of `el` of type `T`
`list(first, last)`	construct a list with the elements from the range indicated by iterators `first` and `last`
`list(lst)`	copy constructor
`size_type max_size() const`	return the maximum number of nodes for the list
`void merge(lst)`	for the sorted current list and `lst`, remove all nodes from `lst` and insert them in sorted order in the current list
`void merge(lst, f)`	for the sorted current list and `lst`, remove all nodes from `lst` and insert them in the current list in the sorted order specified by a two-argument Boolean function `f()`

Continues

FIGURE 3.24 *(continued)*

`void pop_back()`	remove the last node of the list
`void pop_front()`	remove the first node of the list
`void push_front(el)`	insert `el` at the head of the list
`void push_back(el)`	insert `el` at the end of the list
`reverse_iterator rbegin()`	return an iterator that references the last node of the list
`const_reverse_iterator rbegin() const`	return a `const` iterator that references the last node of the list
`void remove(el)`	remove from the list all the nodes that include `el`
`void remove_if(f)`	remove the nodes for which a one-argument Boolean function `f()` returns `true`
`reverse_iterator rend()`	return an iterator that is before the first node of the list
`const_reverse_iterator rend() const`	return a `const` iterator that is before the first node of the list
`void resize(n, el = T())`	make the list have `n` nodes by adding `n - size()` more nodes with element `el` or by discarding overflowing `size() - n` nodes from the end of the list
`size_type size() const`	return the number of nodes in the list
`void sort()`	sort elements of the list in ascending order
`void sort(f)`	sort elements of the list in the order specified by a one-argument Boolean function `f()`
`void splice(i, lst)`	remove the nodes of list `lst` and insert them into the list before the position referenced by iterator `i`
`void splice(i, lst, j)`	remove from list `lst` the node referenced by iterator `j` and insert it into the list before the position referenced by iterator `i`
`void splice(i, lst, first, last)`	remove from list `lst` the nodes in the range indicated by iterators `first` and `last` and insert them into the list before the position referenced by iterator `i`
`void swap(lst)`	swap the content of the list with the content of another list `lst`
`void unique()`	remove duplicate elements from the sorted list
`void unique(f)`	remove duplicate elements from the sorted list where being a duplicate is specified by a two-argument Boolean function `f()`

A new list is generated with the instruction

```
list<T> lst;
```

where `T` can be any data type. If it is a user-defined type, the type must also include a default constructor which is required for initialization of new nodes. Otherwise, the compiler is unable to compile the member functions with arguments initialized by the default constructor. These include one constructor and functions `resize()`, `assign()`, and one version of `insert()`. Note that this problem does not arise when creating a list of pointers to user-defined types, as in

```
list<T*> ptrLst;
```

The working of most of the member functions has already been illustrated in the case of the vector container (see Figure 1.4 and the discussion of these functions in Section 1.8). Vector container has only three member functions not found in the list container (`at()`, `capacity()`, and `reserve()`), but there are a number of list member functions that are not found in the vector container. Examples of their operation are presented in Figure 3.25.

FIGURE **3.25** A program demonstrating the operation of `list` member functions.

```
#include <iostream>
#include <list>
#include <algorithm>

using namespace std;

void main() {
    list<int> lst1;              // lst1 is empty
    list<int> lst2(3,7);         // lst2 = (7 7 7)
    for (int j = 1; j <= 5; j++) // lst1 = (1 2 3 4 5)
        lst1.push_back(j);
    list<int>::iterator i1 = lst1.begin(), i2 = i1, i3;
    i2++; i2++; i2++;
    list<int> lst3(++i1,i2);     // lst3 = (2 3)
    list<int> lst4(lst1);        // lst4 = (1 2 3 4 5)
    i1 = lst4.begin();
    lst4.splice(++i1,lst2);      // lst2 is empty,
                                 // lst4 = (1 7 7 7 2 3 4 5)
    lst2 = lst1;                 // lst2 = (1 2 3 4 5)
    i2 = lst2.begin();
    lst4.splice(i1,lst2,++i2);   // lst2 = (1 3 4 5),
                                 // lst4 = (1 7 7 7 2 2 3 4 5)
    i2 = lst2.begin();
```

Continues

FIGURE 3.25 *(continued)*

```
    i3 = i2;
    lst4.splice(i1,lst2,i2,++i3); // lst2 = (3 4 5),
                                  // lst4 = (1 7 7 7 2 1 2 3 4 5)
    lst4.remove(1);               // lst4 = (7 7 7 2 2 3 4 5)
    lst4.sort();                  // lst4 = (2 2 3 4 5 7 7 7)
    lst4.unique();                // lst4 = (2 3 4 5 7)
    lst1.merge(lst2);             // lst1 = (1 2 3 3 4 4 5 5),
                                  // lst2 is empty
    lst3.reverse();               // lst3 = (3 2)
    lst4.reverse();               // lst4 = (7 5 4 3 2)
    lst3.merge(lst4,greater<int>());     // lst3 = (7 5 4 3 3 2 2),
                                         // lst4 is empty
    lst3.remove_if(bind2nd(not_equal_to<int>(),3));// lst3 = (3 3)
    lst3.unique(not_equal_to<int>());    // lst3 = (3 3)
}
```

3.8 DEQUES IN THE STANDARD TEMPLATE LIBRARY

A *deque* (double-ended queue) is a list that allows for direct access to both ends of the list particularly to insert and delete elements. Hence, a deque can be implemented as a doubly linked list with pointer data members `head` and `tail` as discussed in Section 3.2. Moreover, as pointed out in the previous section, the container `list` uses a doubly linked list already. The STL, however, adds another functionality to the deque, namely, random access to any position of the deque, just as in arrays and vectors. Vectors, as discussed in Section 1.8, have poor performance for insertion and deletion of elements at the front, but these operations are quickly performed for doubly linked lists. This means that the STL deque should combine the behavior of a vector and a list.

The member functions of the STL container `deque` are listed in Figure 3.26. The functions are basically the same as those available for lists, with few exceptions. Deque does not include function `splice()`, which is specific to `list`, and functions `merge()`, `remove()`, `sort()`, and `unique()`, which are also available as algorithms, and `list` only reimplements them as member functions. The most significant difference is the function `at()` (and its equivalent, `operator[]`) that is unavailable in `list`. The latter function is available in vector, and if we compare the set of member functions in `vector` (Figure 1.3) and in `deque`, we see only a few differences. `vector` does not have `pop_front()` and `push_front()`, as does `deque`, but `deque` does not include functions `capacity()` and `reserve()`, which are available in `vector`. A few operations are illustrated in Figure 3.27. Note that for lists

FIGURE 3.26 A list of member functions in the class deque.

Member Function	Operation
`void assign(first, last)`	remove all the elements in the deque and insert in it the elements from the range indicated by iterators `first` and `last`
`void assign(n, el = T())`	remove all the elements in the deque and insert in it n copies of `el`
`T& at(n)`	return the element in position n of the deque
`const T& at(n) const`	return the element in position n of the deque
`T& back()`	return the last element in the deque
`const T& back() const`	return the last element in the deque
`iterator begin()`	return an iterator that references the first element of the deque
`const_iterator begin() const`	return a `const` iterator that references the first element of the deque
`void clear()`	remove all the elements in the deque
`deque()`	construct an empty deque
`deque(n, el = T())`	construct a deque with n copies of `el` of type `T` (if `el` is not provided, a default constructor `T()` is used)
`deque(dq)`	copy constructor
`deque(first, last)`	construct a deque and initialize it with values from the range indicated by iterators `first` and `last`
`bool empty() const`	return `true` if the deque includes no elements and `false` otherwise
`iterator end()`	return an iterator that is past the last element of the deque
`const_iterator end() const`	return a `const` iterator that is past the last element of the deque
`iterator/void erase(i)`	remove the element referenced by iterator `i`
`iterator/void erase(first, last)`	remove the elements in the range indicated by iterators `first` and `last`
`T& front()`	return the first element in the deque
`const T& front() const`	return the first element in the deque
`iterator insert(i, el = T())`	insert `el` before the element indicated by iterator `i` and return iterator referencing the newly inserted element
`void insert(i, n, el)`	insert n copies of `el` before the element referenced by iterator `i`

Continues

FIGURE 3.26 *(continued)*

`void insert(i, first, last)`	insert elements from location referenced by `first` to location referenced by `last` before the element referenced by iterator `i`; `first` and `last` are either `const_iterators` or `const` pointers
`size_type max_size() const`	return the maximum number of elements for the deque
`T& operator[]`	subscript operator
`void pop_back()`	remove the last element of the deque
`void pop_front()`	remove the first element of the deque
`void push_back(el)`	insert `el` at the end of the deque
`void push_front(el)`	insert `el` at the beginning of the deque
`reverse_iterator rbegin()`	return an iterator that references the last element of the deque
`const_reverse_iterator rbegin() const`	return a `const` iterator that references the last element of the deque
`reverse_iterator rend()`	return an iterator that is before the first element of the deque
`void resize(n, el = T())`	make the deque have n positions by adding `n - size()` more positions with element `el` or by discarding overflowing `size() - n` positions from the end of the deque
`reverse_iterator rend()`	return an iterator that is before the first element of the deque
`const_reverse_iterator rend() const`	return a `const` iterator that is before the first element of the deque
`size_type size() const`	return the number of elements in the deque
`void swap(dq)`	swap the content of the deque with the content of another deque `dq`

only autoincrement and autodecrement were possible for iterators, but for deques we can add any number to iterators. For example, `dq1.begin()+1` is legal for deques, but not for lists.

A very interesting aspect of the STL deque is its implementation. Random access can be simulated in doubly linked lists having in the definition of `operator[](int n)` a loop that scans sequentially the list and stops at the *n*th node. The STL implementation solves this problem differently. An STL deque is not implemented as a linked list but as an array of pointers to blocks or arrays of data. The number of blocks changes dynamically depending on storage needs, and the size of the array of pointers changes accordingly. (We encounter a similar approach applied in extendible hashing in Section 10.5.1.)

FIGURE 3.27 A program demonstrating the operation of deque member functions.

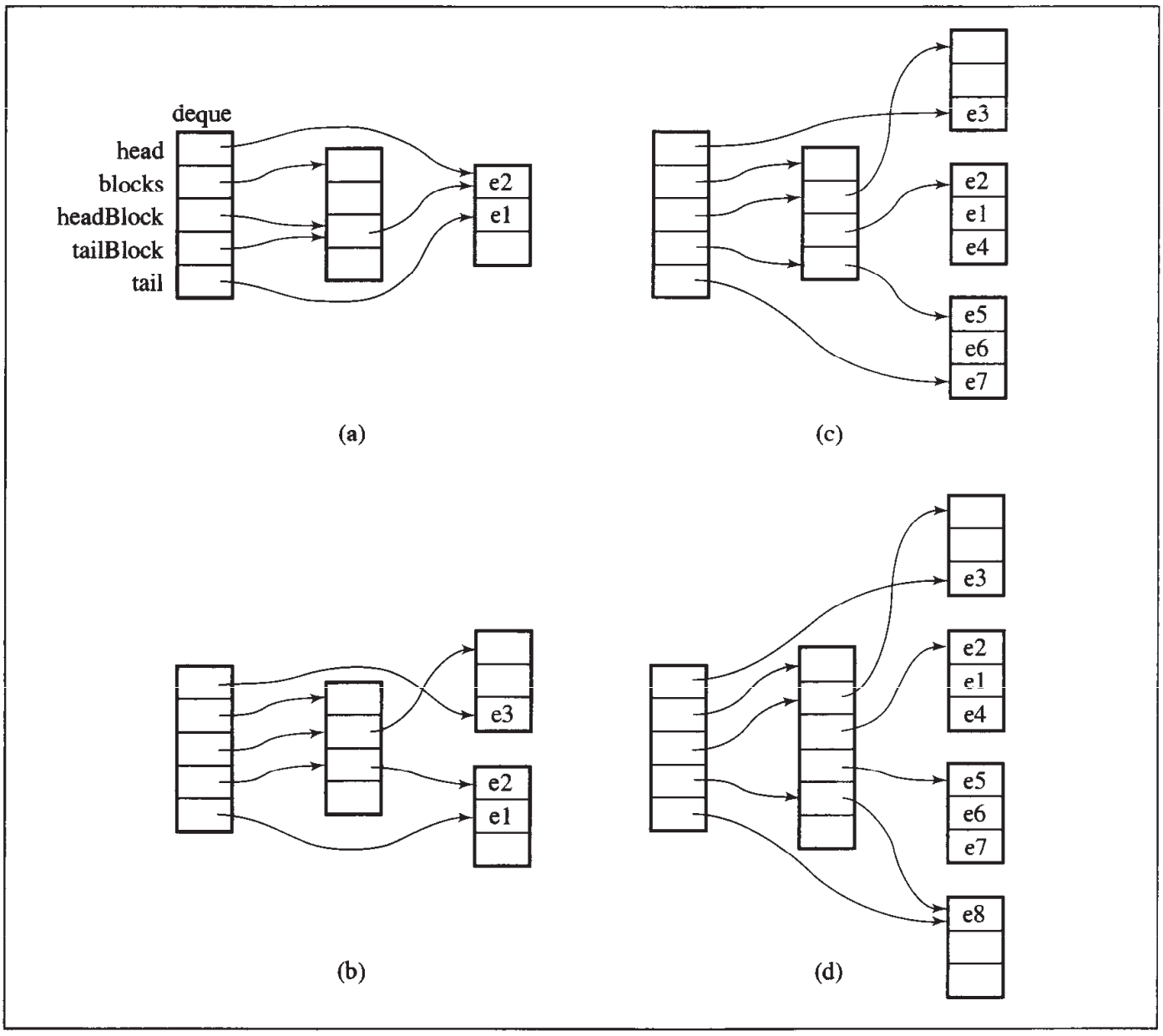
```
#include <iostream.h>
#include <algorithm>
#include <deque>

using namespace std;

void main() {
    deque<int> dq1;
    dq1.push_front(1);                              // dq1 = (1)
    dq1.push_front(2);                              // dq1 = (2 1)
    dq1.push_back(3);                               // dq1 = (2 1 3)
    dq1.push_back(4);                               // dq1 = (2 1 3 4)
    deque<int> dq2(dq1.begin()+1,dq1.end()-1); // dq2 = (1 3)
    dq1[1] = 5;                                     // dq1 = (2 5 3 4)
    dq1.erase(dq1.begin());                         // dq1 = (5 3 4)
    dq1.insert(dq1.end()-1,2,6);                    // dq1 = (5 3 6 6 4)
    sort(dq1.begin(),dq1.end());                    // dq1 = (3 4 5 6 6)
    deque<int> dq3;
    dq3.resize(dq1.size()+dq2.size());              // dq3 = (0 0 0 0 0 0 0)
    merge(dq1.begin(),dq1.end(),dq2.begin(),dq2.end(),dq3.begin());
    // dq1 = (3 4 5 6 6) and dq2 = (1 3) ==> dq3 = (1 3 3 4 5 6 6)
}
```

To discuss one possible implementation, assume that the array of pointers has four cells and an array of data has three cells, that is, `blockSize = 3`. An object deque includes fields `head`, `tail`, `headBlock`, `tailBlock`, and `blocks`. After execution of `push_front(e1)` and `push_front(e2)` with an initially empty deque, the situation is as in Figure 3.28a. First, the array `blocks` is created and then one data block accessible from a middle cell of `blocks`. Next, e1 is inserted in the middle of the data block. The subsequent calls place elements consecutively in the first half of the data array. The third call to `push_front()` cannot successfully place e3 in the current data array; therefore, a new data array is created and e3 is located in the last cell (Figure 3.28b). Now we execute `push_back()` four times. Element e4 is placed in an existing data array accessible from deque through `tailBlock`. Elements e5, e6, and e7 are placed in a new data block, which also becomes accessible through `tailBlock` (Figure 3.28c). The next call to `push_back()` affects the pointer array `blocks` because the last data block is full and the block is accessible for the last cell of `blocks`. In this case, a new pointer array is created which contains (in this implementation) twice as many cells as the number of data blocks. Next, the pointers from old array `blocks` are copied to the new array, and then a new data block can be created to accommodate

FIGURE 3.28 Changed on the deque in the process of pushing new elements.



element e8 being inserted (Figure 3.28d). This is an example of the worst case for which between n/`blockSize` and n/`blockSize` + 2 cells have to be copied from the old array to the new one; therefore, in the worst case, the pushing operation takes $O(n)$ time to perform. But assuming that `blockSize` is a large number, the worst case can be expected to occur very infrequently. Most of the time, the pushing operation requires constant time.

Inserting an element into a deque is very simple conceptually. To insert an element in the first half of the deque, the front element is pushed onto the deque, and all elements that should precede the new element are copied to the preceding cell. Then

the new element can be placed in the desired position. To insert an element into the second half of the deque, the last element is pushed onto the deque, and elements that should follow the new element in the deque are copied to the next cell.

With the discussed implementation, a random access can be performed in constant time. For the situation illustrated in Figure 3.28, that is, with declarations

```
T **blocks;
T **headBlock;
T *head;
```

the subscript operator can be overloaded as follows:

```
T& operator[] (int n) {
    if (n < blockSize - (head - *headBlock))    // if n is
        return * (head + n);                    // in the first
    else {                                      // block;
        n = n - (blockSize - (head - *headBlock));
        int q = n / blockSize + 1;
        int r = n % blockSize;
        return *(*(headBlock + q) + r);
    }
}
```

Although access to a particular position requires several arithmetic, dereferencing, and assignment operations, the number of operations is constant for any size of the deque.

3.9 Concluding Remarks

Linked lists have been introduced to overcome limitations of arrays by allowing dynamic allocation of necessary amounts of memory. Also, linked lists allow easy insertion and deletion of information, since such operations have a local impact on the list. To insert a new element at the beginning of an array, all elements in the array have to be shifted to make room for the new item; hence, insertion has a global impact on the array. Deletion is the same. So should we always use linked lists instead of arrays?

Arrays have some advantages over linked lists, namely that they allow random accessing. To access the tenth node in a linked list, all nine preceding nodes have to be passed. In the array, we can go to the tenth cell immediately. Therefore, if an immediate access of any element is necessary, then an array is a better choice. This was the case with binary search and it will be the case with most sorting algorithms (see Chapter 9). But if we are constantly accessing only some elements—the first, the second, the last, and the like—and if changing the structure is the core of an algorithm, then using a linked list is a better option. A good example is a queue, which is discussed in the next chapter.

Another advantage in the use of arrays is space. To hold items in arrays, the cells have to be of the size of the items. In linked lists, we store one item per node and the

node also includes at least one pointer; in doubly linked lists, the node contains two pointers. For large linked lists, a significant amount of memory is needed to store the pointers. Therefore, if a problem does not require many shifts of data, then having an oversized array may not be wasteful at all if its size is compared to the amount of space needed for the linked structure storing the same data as the array.

3.10 Case Study: A Library

This case study is a program that can be used in a small library to include new books in the library, to check out books to people, and to return them.

As this program is a practice in the use of linked lists, almost everything is implemented in terms of such lists. But to make the program more interesting, it uses linked lists of linked lists that also contain cross-references (see Figure 3.29).

First, there could be a list including all authors of all books in the library. However, searching through such a list can be time-consuming, so the search can be sped up by choosing at least one of the two following strategies:

- The list can be ordered alphabetically, and the search can be interrupted if we find the name, if we encounter an author's name greater than the one we are searching for, or if we reach the end of list.
- We can use an array of pointers to the author structures and indexed with letters; each slot of the array points to the linked list of authors whose names start with the same letter.

The best strategy is to combine both approaches. However, in this case study, only the second approach is used, and the reader is urged to amplify the program by adding the first approach. Note the articles a, an, and the at the beginning of the titles should be disregarded during the sorting operation.

The program uses an array `catalog` of all the authors of the books included in the library and an array `people` of all the people who have used the library at least once. Both arrays are indexed with letters so that, for instance, position `catalog['F']` refers to a linked list of all the authors whose name starts with F.

Because we can have several books by the same author, one of the data members of the author node refers to the list of books of this author that can be found in the library. Similarly, because each person can check out several books, the node corresponding to this person contains a reference to the list of books currently checked out by this person. This fact is also indicated by setting the `person` member of the checked-out book to the node pertaining to the person who is taking the book out.

Books can be returned, and that fact should be reflected by removing the appropriate nodes from the list of the checked-out books of the person who returns them. The `person` member in the node related to the book that is being returned has to be reset to null.

The program defines four classes: `Author`, `Book`, `Person`, and `CheckedOutBook`. To define different types of linked lists, the STL resources are used, in particular, the library `<list>`. But because of the considerable cross-referencing as shown in

FIGURE 3.29 Linked lists indicating library status.

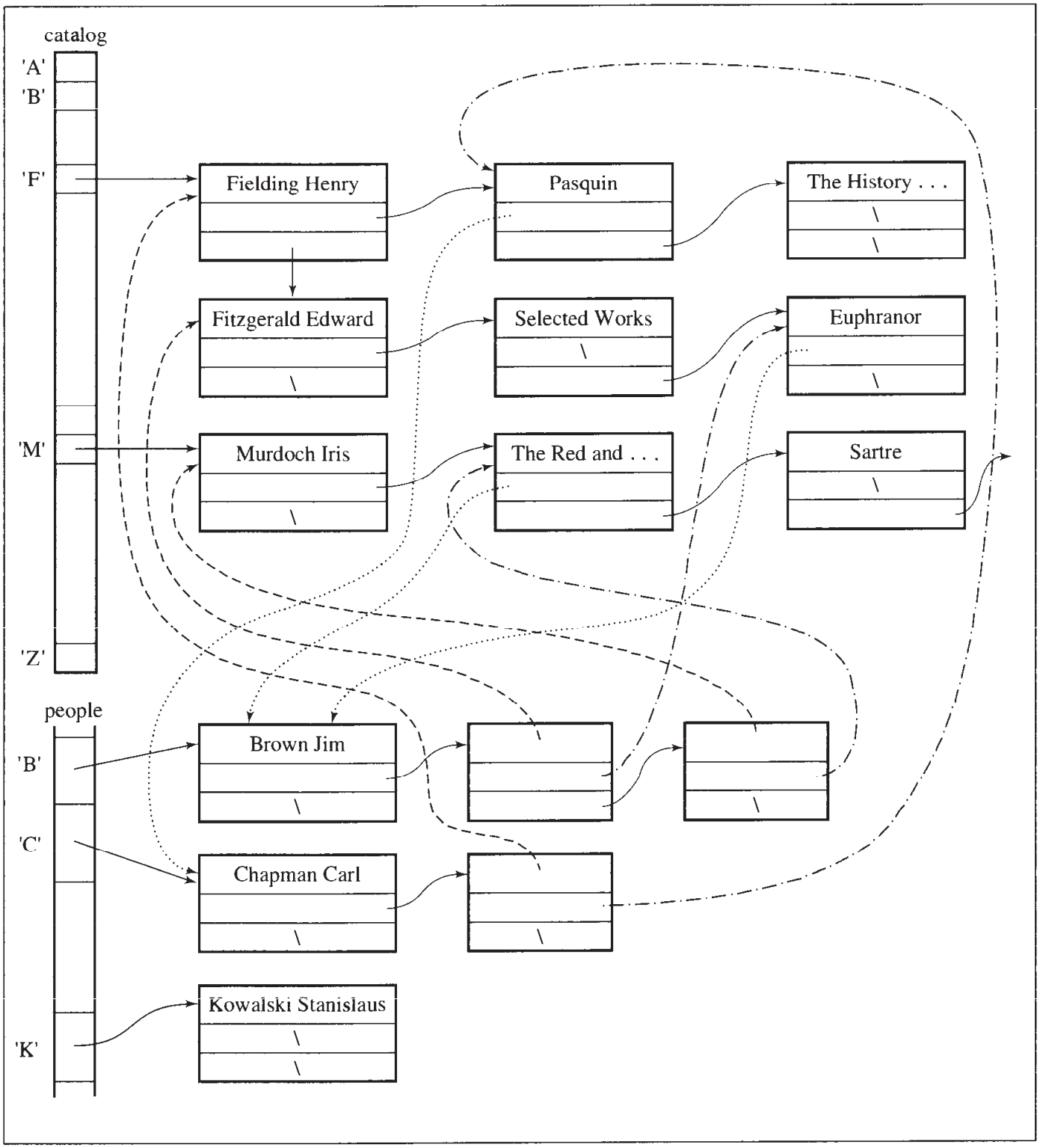


Figure 3.29, in most cases, we do not create lists of objects of a particular class type but lists of pointers to the objects. For example, `catalog` is not defined as an array of linked lists of `Author` objects,

```
list<Author> catalog['Z'+1];
```

but as an array of pointers to `Author` objects

```
list<Author*> catalog['Z'+1];
```

The only exception is the declaration

```
list<CheckedOutBook> books;
```

in class `Person`, but this is because the two data members of class `CheckedOutBook` are pointers already.

Creating lists of pointers to nodes poses certain challenges. In particular, the generic algorithm `find()` cannot be used because it compares node addresses rather than nodes themselves; hence, finding a particular piece of information is always unsuccessful. To solve the problem, the program uses a generic function `findIt()` that takes a list `lst` of pointers to nodes as the first argument as a node `el` to be searched for as the second argument. Locally, it defines an iterator `ref` for the list, and for each node of the list, a comparison `**ref == el` is performed. In this comparison, iterator `ref` is doubly dereferenced: The left asterisk in `**ref` dereferences iterator `ref` because `*ref` is a node in `lst` currently pointed to by `ref`, and the right asterisk dereferences the pointer in the node `*ref` so that `**ref` is an information node pointed to by a pointer in the node pointed to by `ref`. This information node is compared to `el` using an overloaded operator ==.

The same double dereferencing is used in overloading the output operator << for lists,

```
template<class T>
ostream& operator<< (ostream& out, const list<T>& lst) {
    for (list<T>::iterator ref = lst.begin(); ref != lst.end(); ref++)
       out << **ref; // overloaded <<
    return out;
}
```

which requires overloading the same operator for three classes used in the program.

The program allows the user to choose one of the five operations: including a book, checking a book out, returning it, showing the current status of the library, and exiting the program. The operation is chosen after a menu is displayed and a proper number is entered. The cycle of displaying the menu and executing an elected operation ends with choosing the exit option. Here is an example of the status for a situation shown in Figure 3.29.

```
Library has the following books:

Fielding Henry
    * Pasquin - checked out to Chapman Carl
    * The History of Tom Jones
```

```
Fitzgerald Edward
    * Selected Works
    * Euphranor - checked out to Brown Jim
Murdoch Iris
    * The Red and the Green - checked out to Brown Jim
    * Sartre
    * The Bell

The following people are using the library:

Brown Jim has the following books
    * Fitzgerald Edward, Euphranor
    * Murdoch Iris, The Red and the Green
Chapman Carl has the following books
    * Fielding Henry, Pasquin
Kowalski Stanislaus has no books
```

Note that the diagram in Figure 3.29 is somewhat simplified, since strings are not stored directly in structures, only pointers to strings. Hence, technically, each name and title should be shown outside structures with links leading to them. A fragment of Figure 3.29 is shown in Figure 3.30 with implementation details shown more explicitly. Figure 3.31 contains the code for the library program.

3.11 Exercises

1. Assume that a circular doubly linked list has been created, as in Figure 3.32. After each of the following assignments, indicate changes made in the list by showing which links have been modified. The second assignment should make changes in the list modified by the first assignment and so on.

```
list->next->next->next = list->prev;

list->prev->prev->prev = list->next->next->next->prev;

list->next->next->next->prev = list->prev->prev->prev;

list->next = list->next->next;

list->next->prev->next = list->next->next->next;
```

2. How many nodes does the shortest linked list have? The longest linked list?
3. The linked list in Figure 3.11 was created in Section 3.2 with three assignments. Create this list with only one assignment.
4. Merge two ordered singly linked lists of integers into one ordered list.
5. Delete an *i*th node on a linked list. Be sure that such a node exists.

FIGURE 3.30 Fragment of structure from Figure 3.29 with all the objects used in the implementation.

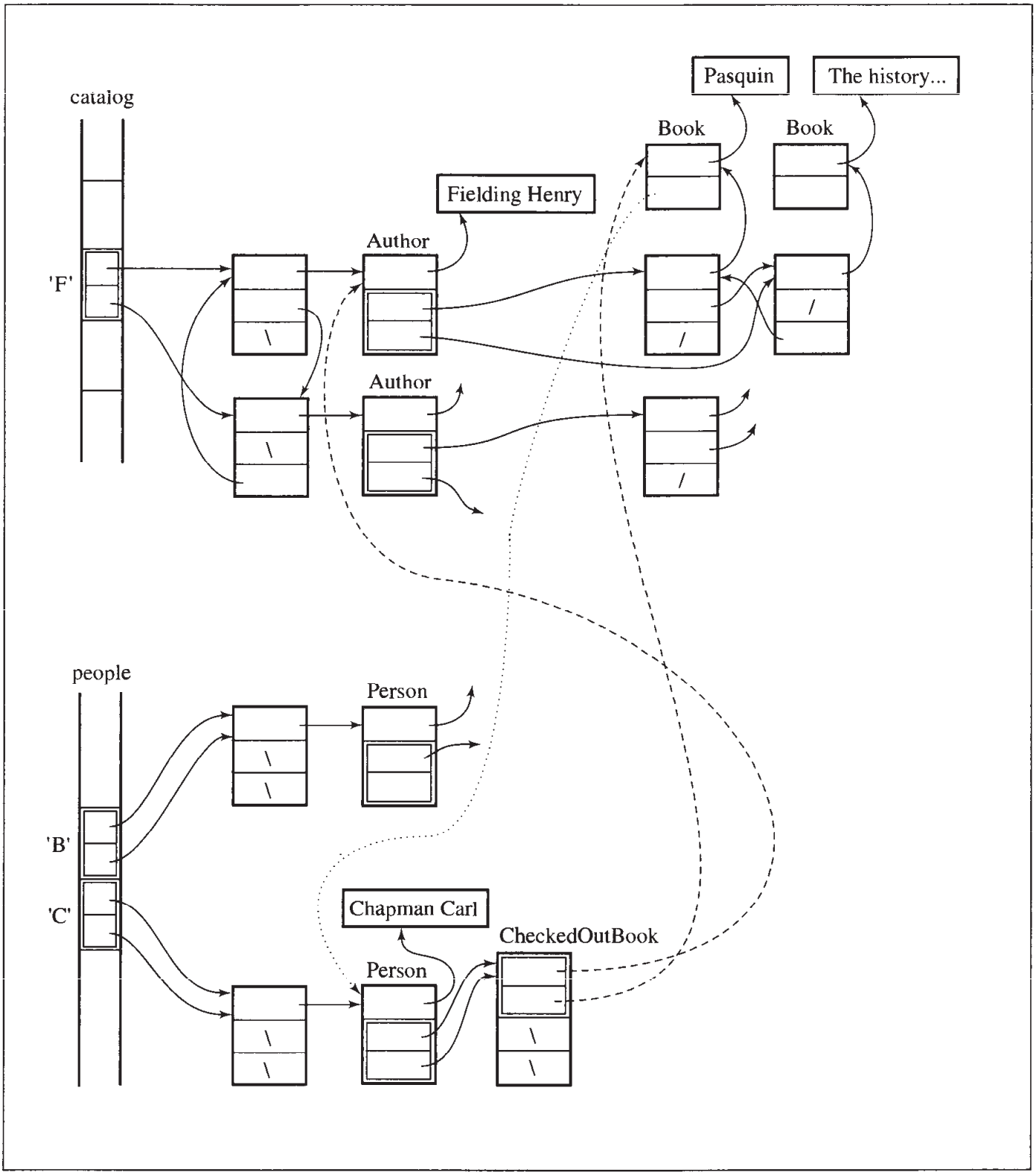


FIGURE 3.31 The library program.

```
#include <iostream>
#include <string>
#include <list>

using namespace std;

class Person;        // forward declaration;

class Book {
public:
    Book() {
        person = 0;
    }
    bool operator== (const Book& bk) const {
        return strcmp(title,bk.title) == 0;
    }
private:
    char *title;
    Person *person;
    friend ostream& operator<< (ostream&,const Book&);
    friend ostream& operator<< (ostream&,const Person&);
    friend class CheckedOutBook;
    friend void includeBook();
    friend void checkOutBook();
    friend void returnBook();
};

class Author {
public:
    Author() {
    }
    bool operator== (const Author& ar) const {
        return strcmp(name,ar.name) == 0;
    }
private:
    char *name;
    list<Book*>books;
    friend ostream& operator<< (ostream&,const Author&);
    friend ostream& operator<< (ostream&,const Person&);
    friend class CheckedOutBook;
    friend void includeBook();
```

Continues

FIGURE 3.31 *(continued)*

```
    friend void checkOutBook();
    friend void returnBook();
};

class CheckedOutBook {
public:
    CheckedOutBook(Author *ar = 0, Book *bk = 0) {
        author = ar; book = bk;
    }
    bool operator== (const CheckedOutBook& bk) const {
        return strcmp(author->name,bk.author->name) == 0 &&
               strcmp(book->title,bk.book->title) == 0;
    }
    bool operator!= (const CheckedOutBook& bk) const {
         return !(*this == bk);
    }
    bool operator< (const CheckedOutBook& bk) const {
        if (strcmp(author->name,bk.author->name) == 0)
            return strcmp(book->title,bk.book->title) < 0;
        return strcmp(author->name,bk.author->name) < 0;
    }
    bool operator> (const CheckedOutBook& bk) const {
         return !(*this == bk) && !(*this < bk);
    }
private:
    Author* author;
    Book* book;
    friend ostream& operator<< (ostream&,const Person&);
    friend void checkOutBook();
    friend void returnBook();
};

class Person {
public:
    Person() {
    }
    bool operator== (const Person& pn) const {
        return strcmp(name,pn.name) == 0;
    }
private:
    char *name;
```

FIGURE 3.31 *(continued)*

```
    list<CheckedOutBook> books;
    friend ostream& operator<< (ostream&,const Book&);
    friend ostream& operator<< (ostream&,const Person&);
    friend void checkOutBook();
    friend void returnBook();
};

list<Author*> catalog['Z'+1];
list<Person*> people['Z'+1];

template<class T>
ostream& operator<< (ostream& out, const list<T>& lst) {
    for (list<T>::iterator ref = lst.begin(); ref != lst.end(); ref++)
        out << **ref; // overloaded <<
    return out;
}

ostream& operator<< (ostream& out, const Book& bk) {
    out << "    * " << bk.title;
    if (bk.person != 0)
        out << " - checked out to " << bk.person->name; // overloaded <<
    out << endl;
    return out;
}

ostream& operator<< (ostream& out, const Author& ar) {
    out << ar.name << endl << ar.books;  // overloaded <<
    return out;
}

ostream& operator<< (ostream& out, const Person& pr) {
    out << pr.name;
    if (!pr.books.empty()) {
        out << " has the following books:\n";
        list<CheckedOutBook>::iterator bk = pr.books.begin();
        for ( ; bk != pr.books.end(); bk++)
            out << "    * " << bk->author->name << ", "
                << bk->book->title << endl;
    }
    else out << " has no books\n";
    return out;
```

Continues

FIGURE 3.31 *(continued)*

```
}

template<class T1, class T2>
list<T2>::iterator findIt(const list<T2>& lst, const T1& el) {
    for (list<T2>::iterator ref = lst.begin(); ref != lst.end(); ref++)
        if (**ref == el) // overloaded ==
            break;
    return ref;
}

char* getString(char *msg) {
    char s[82], i, *destin;
    cout << msg;
    cin.get(s,80);
    while (cin.get(s[81]) && s[81] != '\n');  // discard overflowing
    destin = new char[strlen(s)+1];           // characters;
    for (i = 0; destin[i] = toupper(s[i]); i++);
    return destin;
}

void status() {
    register int i;
    cout << "Library has the following books:\n\n";
    for (i = 'A'; i <= 'Z'; i++)
        if (!catalog[i].empty())
            cout << catalog[i];
    cout << "\nThe following people are using the library:\n\n";
    for (i = 'A'; i <= 'Z'; i++)
        if (!people[i].empty())
            cout << people[i];
}

void includeBook() {
    Author *newAuthor = new Author;
    Book *newBook = new Book;
    newAuthor->name = getString("Enter author's name: ");
    newBook->title  = getString("Enter the title of the book: ");
    list<Author*>::iterator oldAuthor =
         findIt(catalog[newAuthor->name[0]],*newAuthor);
    if (oldAuthor == catalog[newAuthor->name[0]].end()) {
         newAuthor->books.push_front(newBook);
```

FIGURE 3.31 *(continued)*

```
        catalog[newAuthor->name[0]].push_front(newAuthor);
    }
    else (*oldAuthor)->books.push_front(newBook);
}

void checkOutBook() {
    Person *person = new Person;
    Author author;
    Book book;
    list<Author*>::iterator authorRef;
    list<Book*>::iterator bookRef;
    person->name = getString("Enter person's name: ");
    while (true) {
        author.name = getString("Enter author's name: ");
        authorRef = findIt(catalog[author.name[0]],author);
        if (authorRef == catalog[author.name[0]].end())
            cout << "Misspelled author's name\n";
        else break;
    }
    while (true) {
        book.title = getString("Enter the title of the book: ");
        bookRef = findIt((*authorRef)->books,book);
        if (bookRef == (*authorRef)->books.end())
            cout << "Misspelled title\n";
        else break;
    }
    list<Person*>::iterator personRef;
    personRef = findIt(people[person->name[0]],*person);
    CheckedOutBook checkedOutBook(*authorRef,*bookRef);
    if (personRef == people[person->name[0]].end()) { // a new person
        person->books.push_front(checkedOutBook);    // in the library;
        people[person->name[0]].push_front(person);
        (*bookRef)->person = *people[person->name[0]].begin();
    }
    else {
        (*personRef)->books.push_front(checkedOutBook);
        (*bookRef)->person = *personRef;
    }
}

void returnBook() {
```

Continues

FIGURE 3.31 *(continued)*

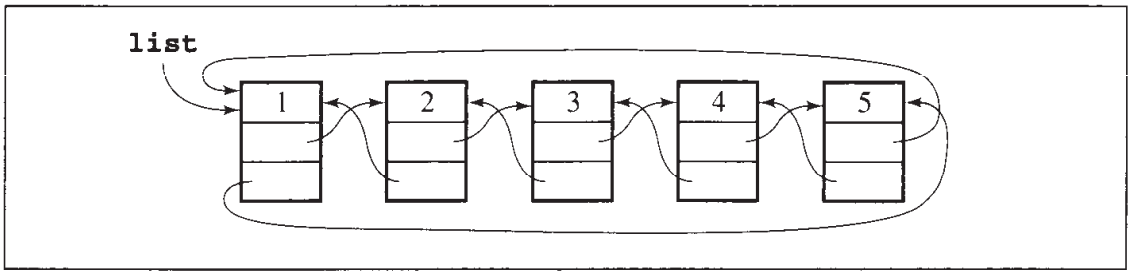
```
    Person person;
    Book book;
    Author author;
    list<Person*>::iterator personRef;
    list<Book*>::iterator bookRef;
    list<Author*>::iterator authorRef;
    while (true) {
        person.name = getString("Enter person's name: ");
        personRef = findIt(people[person.name[0]],person);
        if (personRef == people[person.name[0]].end())
             cout << "Misspelled person's name\n";
        else break;
    }
    while (true) {
        author.name = getString("Enter author's name: ");
        authorRef = findIt(catalog[author.name[0]],author);
        if (authorRef == catalog[author.name[0]].end())
             cout << "Misspelled author's name\n";
        else break;
    }
    while (true) {
        book.title = getString("Enter the title of the book: ");
        bookRef = findIt((*authorRef)->books,book);
        if (bookRef == (*authorRef)->books.end())
             cout << "Misspelled title\n";
        else break;
    }
    CheckedOutBook checkedOutBook(*authorRef,*bookRef);
    (*bookRef)->person = 0;
    (*personRef)->books.remove(checkedOutBook);
}

int menu() {
    int option;
    cout << "\nEnter one of the following options:\n"
         << "1. Include a book in the catalog\n2. Check out a book\n"
         << "3. Return a book\n4. Status\n5. Exit\n"
         << "Your option? ";
    cin >> option;
    cin.get();          // discard '\n';
    return option;
}
```

FIGURE 3.31 *(continued)*

```
void main() {
    while (true)
        switch (menu()) {
            case 1: includeBook();  break;
            case 2: checkOutBook(); break;
            case 3: returnBook();   break;
            case 4: status();       break;
            case 5: return;
            default: cout << "Wrong option, try again: ";
        }
}
```

FIGURE 3.32 A circular doubly linked list.



6. Delete from list L_1 nodes whose positions are to be found in an ordered list L_2. For instance, if L_1 = (A B C D E) and L_2 = (2 4 8), then the second and the fourth nodes are to be deleted from list L_1 (the eighth node does not exist), and after deletion, L_1 = (A C E).

7. Delete from list L_1 nodes occupying positions indicated in ordered lists L_2 and L_3. For instance, if L_1 = (A B C D E), L_2 = (2 4 8), and L_3 = (2 5), then after deletion, L_1 = (A C).

8. Delete from an ordered list L nodes occupying positions indicated in list L itself. For instance, if L = (1 3 5 7 8), then after deletion, L = (1 7).

9. A linked list does not have to be implemented with pointers. Suggest other implementations of linked lists.

10. Write a member function to check whether two singly linked lists have the same contents.

11. Write a member function to reverse a singly linked list using only one pass through the list.

12. Insert a new node into a singly linked list (a) before and (b) after a node pointed by `p` in this list (possibly the first or the last). Do not use a loop in either operation.

13. Attach a singly linked list to the end of another singly linked list.

14. Put numbers in a singly linked list in ascending order. Use this operation to find the median in the list of numbers.

15. How can a singly linked list be implemented so that insertion requires no test for whether `head` is null?

16. Insert a node in the middle of a doubly linked list.

17. Write code for class `IntCircularSLList` for a circular singly linked list that includes equivalents of the member functions listed in Figure 3.2.

18. Write code for class `IntCircularDLList` for a circular doubly linked list that includes equivalents of the member functions listed in Figure 3.2.

19. How likely is the worst case for searching a skip list to occur?

20. Consider the move-to-front, transpose, count, and ordering methods.

(a) In what case is a list maintained by these methods not changed?

(b) In what case do these methods require an exhaustive search of lists for each search, assuming that only elements in the list are searched for?

21. In the discussion of self-organizing lists, only the number of comparisons was considered as the measure of different methods' efficiency. This measure can, however, be greatly affected by a particular implementation of the list. Discuss how the efficiency of the move-to-front, transpose, count, and ordering methods are affected in the case when the list is implemented as

(a) an array

(b) a singly linked list

(c) a doubly linked list

22. For doubly linked lists, there are two variants of the move-to-front and transpose methods (Valiveti and Oommen 1993). A *move-to-end* method moves a node being accessed to the end opposite from which the search started. For instance, if the doubly linked list is a list of items *A B C D* and the search starts from the left end to access node *C*, then the reorganized list is *A B D C*. If the search for *C* started from the right end, the resulting list is *C A B D*.

The *swapping* technique transposes a node with this predecessor also with respect to the end from which the search started. Assuming that only elements of the list are in the data, what is the worst case for a move-to-end doubly linked list when the search is made alternately from the left and from the right? For a swapping list?

23. What is the maximum number of comparisons for optimal search for the 14 letters shown in Figure 3.19?

24. Adapt the binary search to linked lists. How efficient can this search be?

3.12 PROGRAMMING ASSIGNMENTS

1. Farey fractions of level one are defined as sequence $\left(\frac{0}{1}, \frac{1}{1}\right)$. This sequence is extended in level two to form a sequence $\left(\frac{0}{1}, \frac{1}{2}, \frac{1}{1}\right)$, sequence $\left(\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}\right)$ at level three, sequence $\left(\frac{0}{1}, \frac{1}{4}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{1}{1}\right)$ at level four, so that at each level n, a new fraction $\frac{a+b}{c+d}$ is inserted between two neighbor fractions $\frac{a}{c}$ and $\frac{b}{d}$ only if $c + d \leq n$. Write a program which for a number n entered by the user creates—by constantly extending it—a linked list of fractions at level n and then displays them.

2. Write a simple airline ticket reservation program. The program should display a menu with the following options: reserve a ticket, cancel a reservation, check whether a ticket is reserved for a particular person, and display the passengers. The information is maintained on an alphabetized linked list of names. In a simpler version of the program, assume that tickets are reserved for only one flight. In a fuller version, place no limit on the number of flights. Create a linked list of flights with each node including a pointer to a linked list of passengers.

3. Read Section 12.1 about sequential-fit methods. Implement the discussed methods with linked lists and compare their efficiency.

4. Write a program to simulate managing files on disk. Define the disk as a one-dimensional array `disk` of size `numOfSectors*sizeOfSector`, where `sizeOfSector` indicates the number of characters stored in one sector. (For the sake of debugging, make it a very small number.) A pool of available sectors is kept in a linked list `sectors` of three-field structures: two fields to indicate ranges of available sectors and one `next` field. Files are kept in a linked list `files` of four-field structures: file name, the number of characters in the file, a pointer to a linked list of sectors where the contents of the file can be found, and the `next` field.

   a. In the first part, implement functions to save and delete files. Saving files requires claiming a sufficient number of sectors from `pool`, if available. The sectors may not be contiguous, so the linked list assigned to the file may contain several nodes. Then the contents of the file have to be written to the sectors assigned to the file. Deletion of a file only requires removing the nodes corresponding with this file (one from `files` and the rest from its own linked list of sectors) and transferring the sectors assigned to this file back to `pool`. No changes are made in `disk`.

   b. File fragmentation slows down file retrieval. In the ideal situation, one cluster of sectors is assigned to one file. However, after many operations with files, it may not be possible. Extend the program to include a function `together()` to transfer files to contiguous sectors, that is, to create a situation illustrated in Figure 3.33. Fragmented files `file1` and `file2` occupy only one cluster of sectors after `together()` is finished. However, particular care should be taken not to overwrite sectors occupied by other files. For example, `file1` requires eight sectors; five sectors are free at the beginning of `pool`, but sectors 5 and 6 are occupied by `file2`. Therefore, a file *f* occupying such sectors has to be located first by scanning `files`. The contents of these sectors must be

transferred to unoccupied positions, which requires updating the sectors belonging to *f* in the linked list; only then can the released sectors be utilized. One way of accomplishing this is by copying from the area into which one file is copied chunks of sectors of another file into an area of the disk large enough to accommodate these chunks. In the example in Figure 3.33, contents of `file1` are first copied to sectors 0 through 4, and then copying is temporarily suspended because sector 5 is occupied. Thus, contents of sectors 5 and 6 are moved to sector 12 and 14, and the copying of `file1` is resumed.

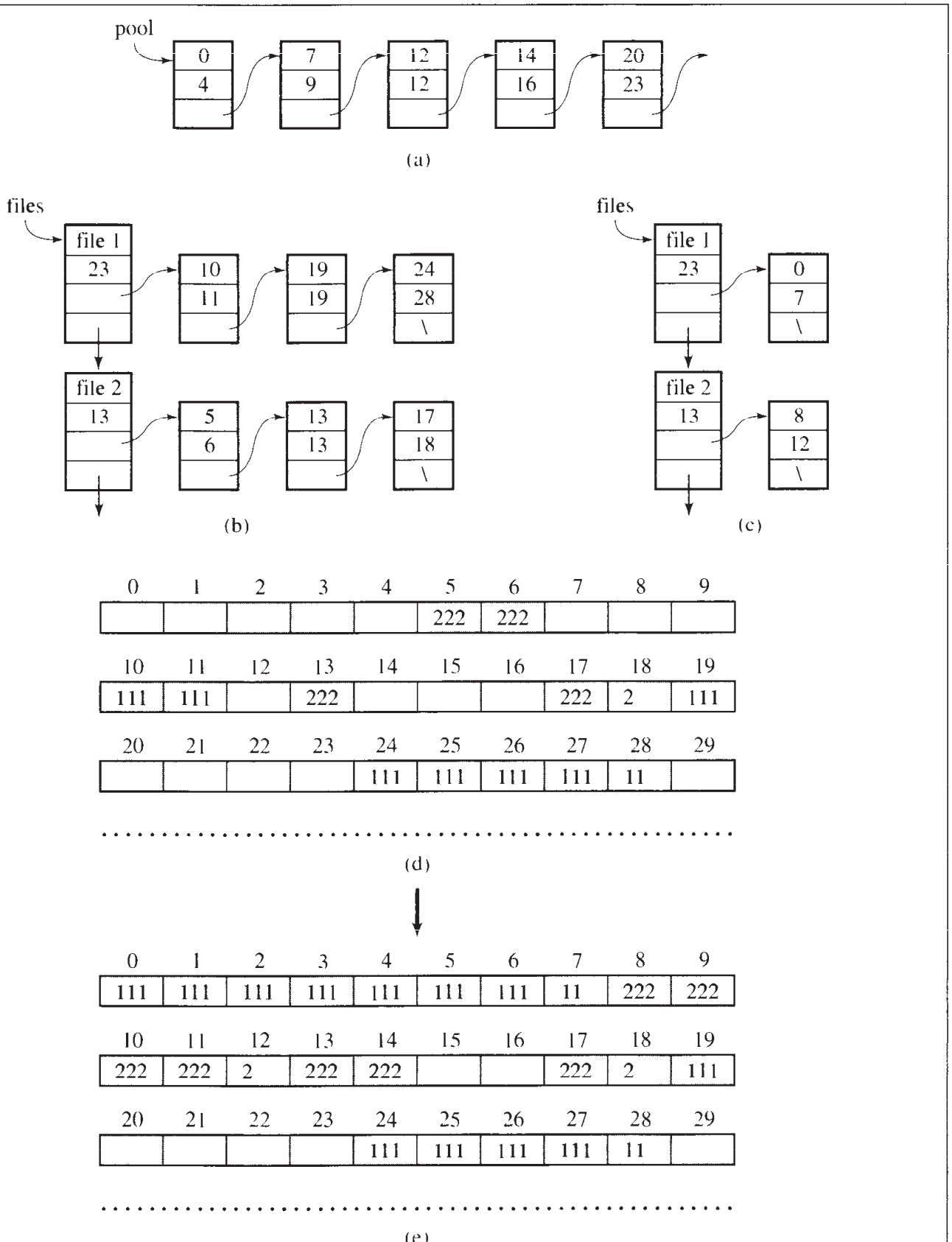
5. Write a simple line editor. Keep the entire text on a linked list, one line in a separate node. Start the program with entering `EDIT file`, after which a prompt appears along with the line number. If the letter `I` is entered with a number *n* following it, then insert the text to be followed before line *n*. If `I` is not followed by a number, then insert the text before the current line. If `D` is entered with two numbers *n* and *m*, one *n*, or no number following it, then delete lines *n* through *m*, line *n*, or the current line. Do the same with the command, `L`, which stands for listing lines. If `A` is entered, then append the text to the existing lines. Entry `E` signifies exit and saving the text in a file. Here is an example:

```
 EDIT testfile
1> The first line
2>
3> And another line
4> I 3
3> The second line
4> One more line
5> L
1> The first line
2>
3> The second line
4> One more line
5> And another line     // This is now line 5, not 3;
5> D 2                  // line 5, since L was issued from line 5;
4> L                    // line 4, since one line was deleted;
1> The first line
2> The second line      // this and the following lines
3> One more line        // now have new numbers;
4> And another line
4> E
```

6. Extend the case study program in this chapter to have it store all the information in the file `Library` at exit and initialize all the linked lists using this information at the invocation of the program. Also, extend it by adding more error checking, such as not allowing the same book to be checked out at the same time to more than one person or not including the same person more than once in the library.

7. Test the efficiency of skip lists. In addition to the functions given in this chapter, implement `skipListDelete()` and then compare the number of node accesses in

FIGURE 3.33 Linked lists used to allocate disk sectors for files: (a) a pool of available sectors; two files (b) before and (c) after putting them in contiguous sectors; the situation in sectors of the disk (d) before and (e) after this operation.



searching, deleting, and inserting for large numbers of elements. Compare this efficiency with the efficiency of linked lists and ordered linked lists. Test your program on a randomly generated order of operations to be executed on the elements. These elements should be processed in random order. Then try your program on nonrandom samples.

8. Write a rudimentary lint program to check whether all variables have been initialized and whether local variables have the same names as global variables. Create a linked list of global variables and, for each function, create a linked list of local variables. In both lists, store information on the first initialization of each variable and check if any initialization has been made before a variable is used for the first time. Also, compare both lists to detect possible matches and issue a warning if a match is found. The list of local variables is removed after the processing of one function is finished and created anew when a new function is encountered. Consider the possibility of maintaining alphabetical order on both lists.

Bibliography

Bentley, Jon L. and McGeoch, Catharine C., "Amortized Analyses of Self-Organizing Sequential Search Heuristics," *Communications of the ACM* 28 (1985), 404–411.

Foster, John M., *List Processing*, London: McDonald, 1967.

Hansen, Wilfred J., "A Predecessor Algorithm for Ordered Lists," *Information Processing Letters* 7 (1978), 137–138.

Hester, James H. and Hirschberg, Daniel S., "Self-Organizing Linear Search," *Computing Surveys* 17 (1985), 295–311.

Pugh, William, "Skip Lists: a Probabilistic Alternative to Balanced Trees," *Communications of the ACM* 33 (1990), 668–676.

Rivest, Ronald, "On Self-Organizing Sequential Search Heuristics," *Communications of the ACM* 19 (1976), No. 2, 63–67.

Sleator, Daniel D. and Tarjan, Robert E., "Amortized Efficiency of List Update and Paging Rules," *Communications of the ACM* 28 (1985), 202–208.

Valiveti, R. S. and Oommen, B. J., "Self-Organizing Doubly Linked Lists," *Journal of Algorithms* 14 (1993), 88–114.

Wilkes, Maurice V., "Lists and Why They Are Useful," *Computer Journal* 7 (1965), 278–281.