Linked Lists

3

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.

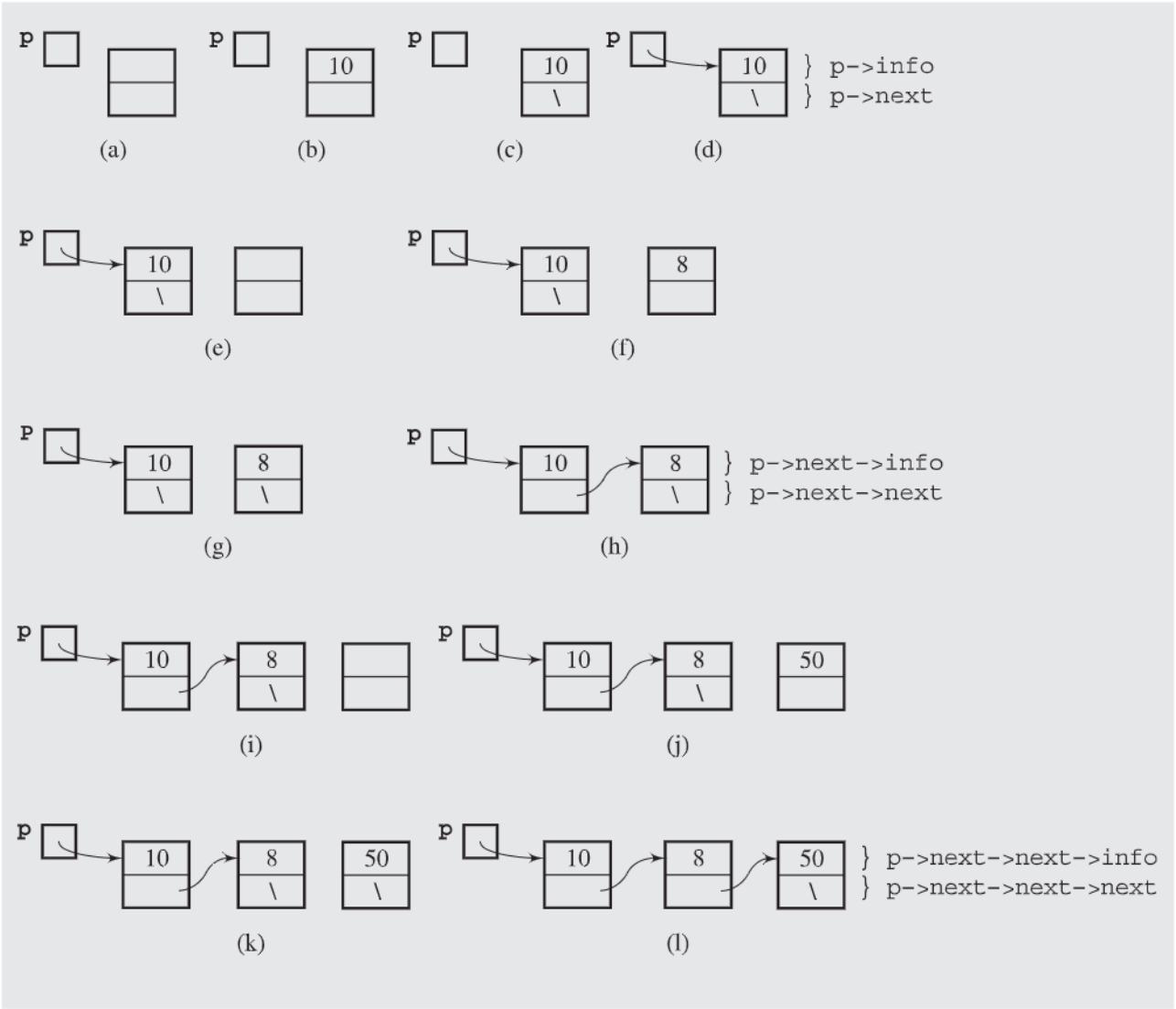
3.1 Singly Linked Lists

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 IntSLLNode {
public:
    IntSLLNode() {
        next = 0;
    }
    IntSLLNode(int i, IntSLLNode *in = 0) {
        info = i; next = in;
    }
```

FIGURE 3.1 A singly linked list.




```
    int info;
    IntSLLNode *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 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 `IntSLLNode` 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. Objects that include such a data member are called self-referential objects.

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 3.1l. One way to create this three-node linked list is to first generate the node for number 10, then the node for 8, and finally the node for 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

```
IntSLLNode *p = new IntSLLNode(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 `IntSLLNode` 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. The second and third steps—initialization of data members of the new `IntSLLNode`—are performed by invoking the constructor `IntSLLNode(10)`, which turns into the constructor `IntSLLNode(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 IntSLLNode(8);
```

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

1. A new node is created (Figure 3.1e).
2. The constructor assigns the number 8 to the `info` member of this node (Figure 3.1f).
3. The constructor assigns null to its `next` member (Figure 3.1g).
4. The new node is included 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 IntSLLNode(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 `next`s 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 1,003rd node on the list? Typing 1,003 `next`s, 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.

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

#ifndef INT_LINKED_LIST
#define INT_LINKED_LIST

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

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

FIGURE 3.2 *(continued)*

```
private:
    IntSLLNode *head, *tail;
};

#endif

//************************  intSLList.cpp ***************************

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

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

Continues

FIGURE 3.2 *(continued)*

```
        delete head;
        head = tail = 0;
    }
    else {              // if more than one node in the list,
        IntSLLNode *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 nonempty 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
            IntSLLNode *tmp = head;
            head = head->next;
            delete tmp;            // and old head is deleted;
        }
        else {                     // if more than one node in the list
            IntSLLNode *pred, *tmp;
            for (pred = head, tmp = head->next; // and a nonhead 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;
            }
        }
}
bool IntSLList::isInList(int el) const {
    IntSLLNode *tmp;
    for (tmp = head; tmp != 0 && !(tmp->info == el); tmp = tmp->next);
    return tmp != 0;
}
```

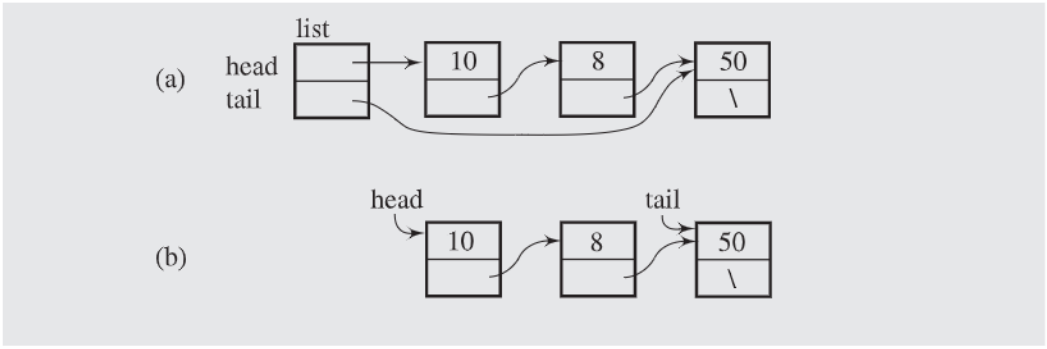
The singly linked list implementation in Figure 3.2 uses two classes: one class, `IntSLLNode`, for nodes of the list, and another, `IntSLList`, for access to the list. The class `IntSLList` defines two data members, `head` and `tail`, which are pointers to the first and the last nodes of a list. This explains why all members of `IntSLLNode` 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 `IntSLLNode` 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.

FIGURE 3.3 A singly linked list of integers.



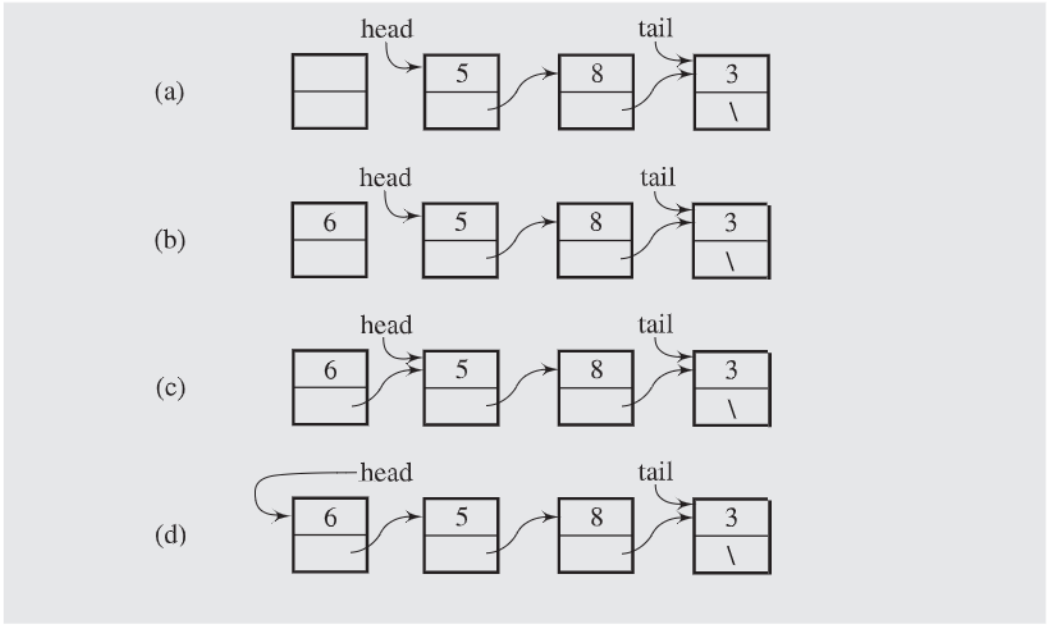
Besides the `head` and `tail` members, the class `IntSLList` also defines member functions that allow us to manipulate the lists. We now look more closely at some basic operations on linked lists presented in Figure 3.2.

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 3.4 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 `IntSLLNode(el,head)`. The last step is executed directly in the function by assigning the address of the newly created node to `head`.

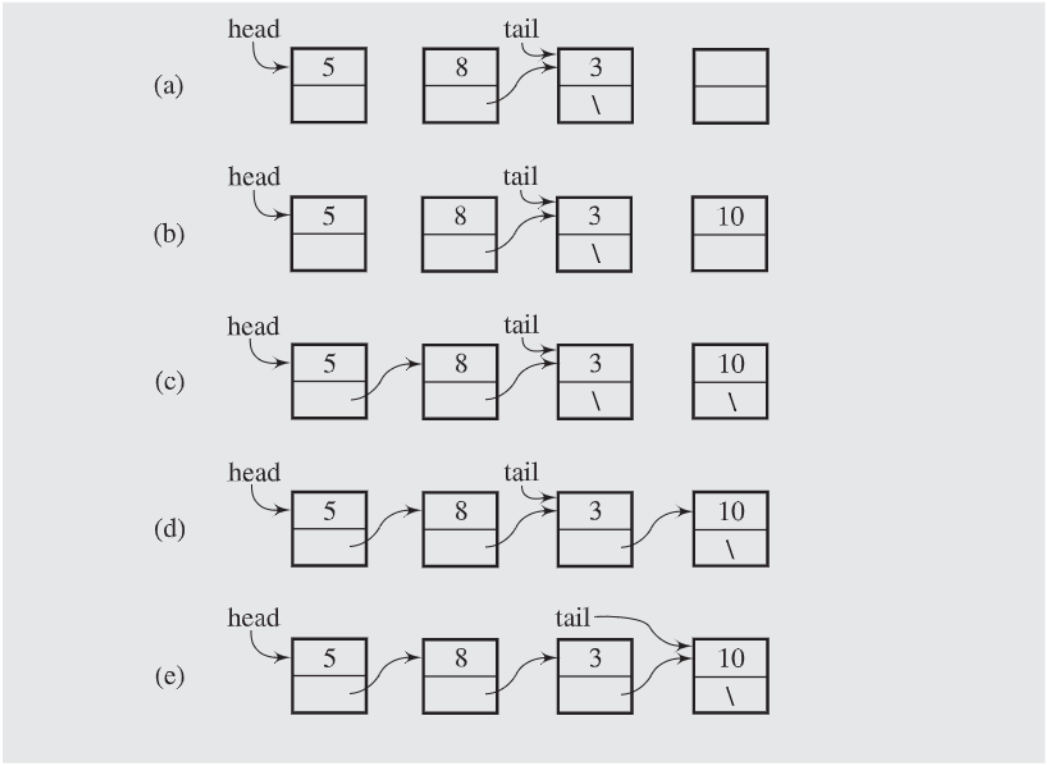
The member function `addToHead()` singles out one special case, namely, inserting a new node in an empty linked list. In an empty linked list, both `head` and `tail` are null; therefore, both become pointers to the only node of the new list. When inserting in a nonempty list, only `head` needs to be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

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

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.

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



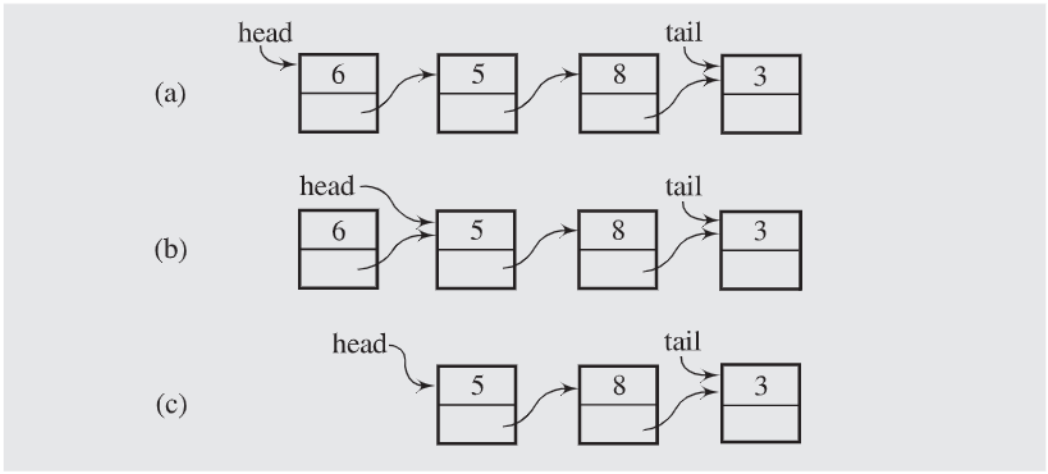
The process of inserting a new node at the beginning of the list is very similar to the process of inserting a node at the end of the list. This is because the implementation of `IntSLList` uses two pointer members: `head` and `tail`. For this reason, both `addToHead()` and `addToTail()` can be executed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

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



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 a 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 the caller may wish to continue 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` statement, with the solution appropriate to the particular case. If the statement 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 delete;
```

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

```
int IntSLList::deleteFromHead() {
    if (!isEmpty()) {          // if nonempty 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 a 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 nonempty 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. In either case, this means that the if statement in deleteFromHead() is redundant.

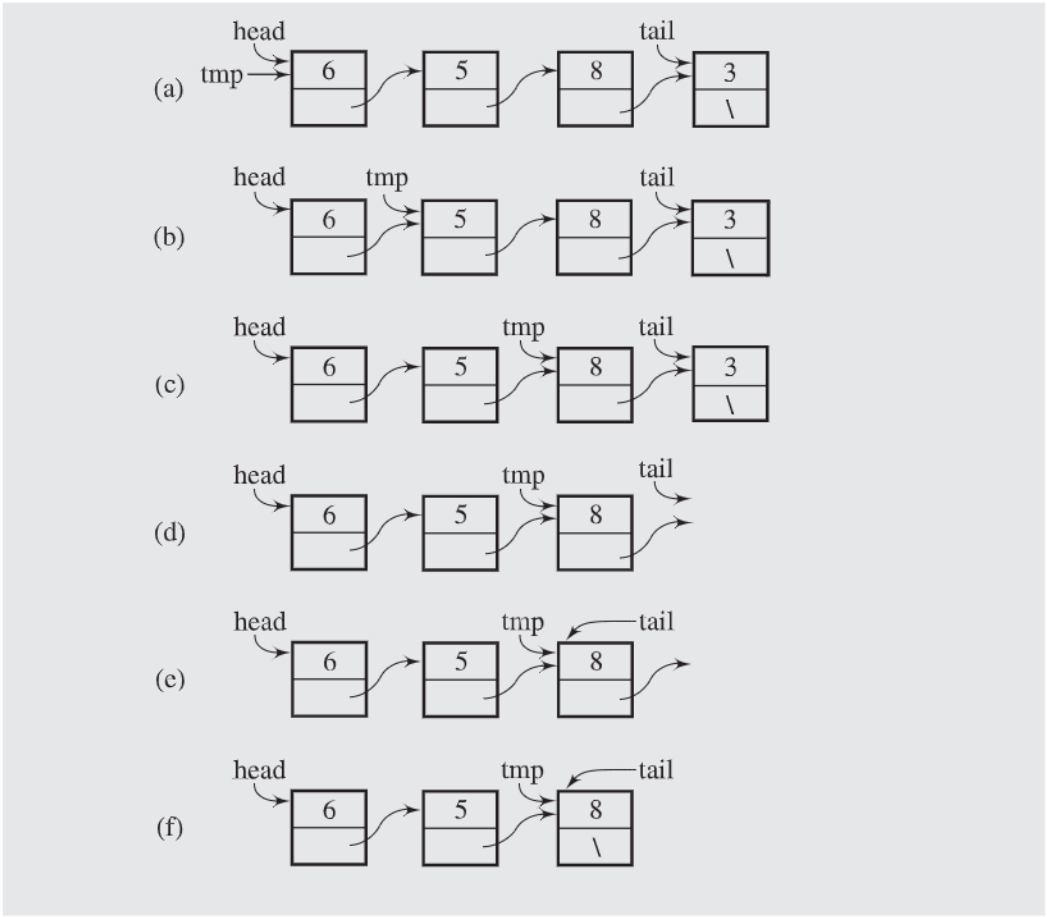
The second special case is when the list has only one node to be removed. In this case, the list becomes empty, which requires setting tail and head to null.

The second deletion operation consists of 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 that scans 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 to 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);
```

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



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.

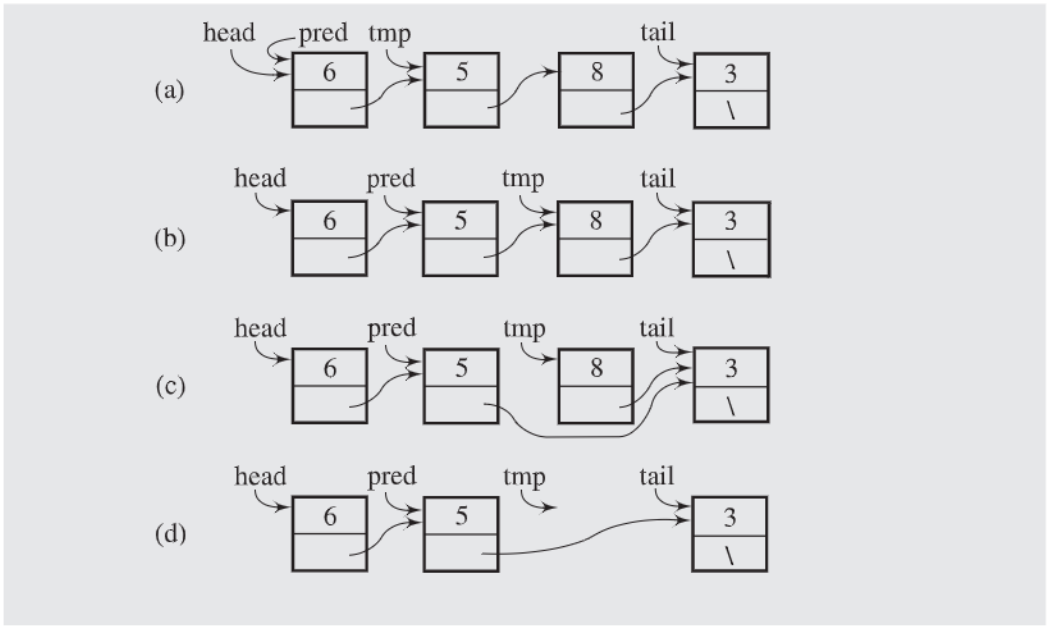
The most time-consuming part of `deleteFromTail()` is finding the next to last node performed by the `for` loop. It is clear that the loop performs $n - 1$ iterations in a list of n nodes, which is the main reason this member function takes $O(n)$ time to delete the last node.

The two discussed deletion operations remove a node from the head or from the tail (that is, always from the same position) and return the 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.

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 by first 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 the 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).

FIGURE 3.8 Deleting a node from a singly linked list.



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

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 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 != 0`: if `tmp` is not null, `el` was found and `true` is returned. If `tmp` is null, the search was unsuccessful and `false` is returned.

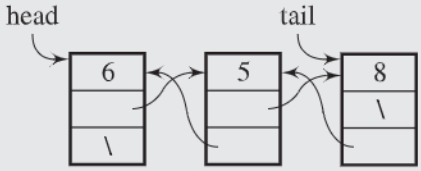
With reasoning similar to that used to determine the efficiency of `deleteNode()`, `isInList()` takes $O(1)$ time in the best case and $O(n)$ in the worst and average cases.

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 store only integers. If we wanted a linked list for float numbers or for 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 that 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.

FIGURE 3.9 A doubly linked list.



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

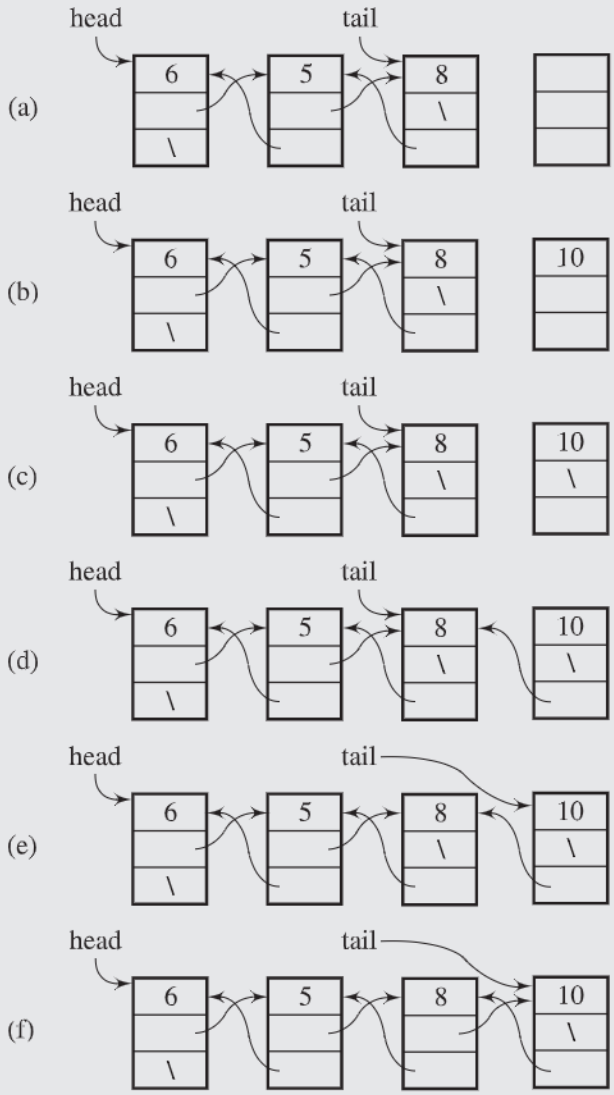
FIGURE 3.10 An implementation of a doubly linked list.

```
//***************************  genDLList.h  ***************************
#ifndef DOUBLY_LINKED_LIST
#define DOUBLY_LINKED_LIST

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

template<class T>
class DoublyLinkedList {
public:
    DoublyLinkedList() {
        head = tail = 0;
    }
    void addToDLLTail(const T&);
    T deleteFromDLLTail();
    . . . . . .
protected:
    DLLNode<T> *head, *tail;
};
template<class T>
void DoublyLinkedList<T>::addToDLLTail(const T& el) {
    if (tail != 0) {
         tail = new DLLNode<T>(el,0,tail);
         tail->prev->next = tail;
    }
    else head = tail = new DLLNode<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;
```

FIGURE 3.10 *(continued)*

```
            delete tail->next;
            tail->next = 0;
        }
        return el;
    }
    . . . . . .
    #endif
```

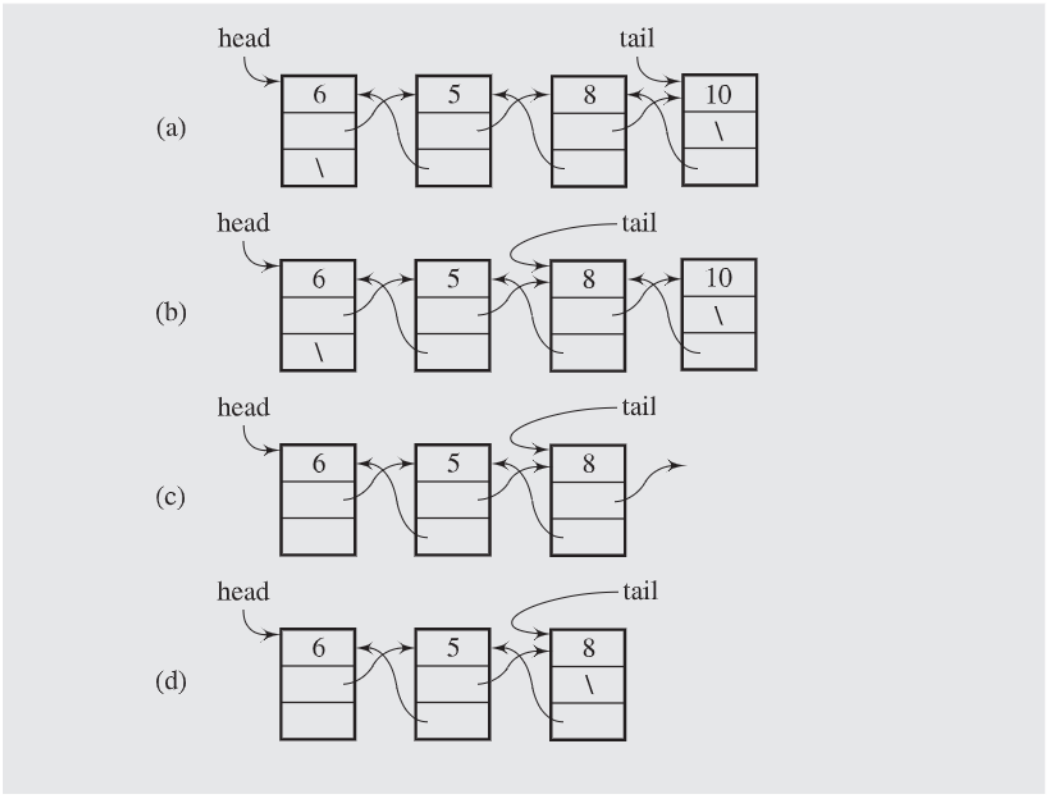
FIGURE 3.11 Adding a new node at the end of a doubly linked list.



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 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 the `prev` member to 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, the 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 the object stored in the removed node.

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



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 attempting to delete the last node. As with the singly linked list's `deleteFromHead()`, the caller should have an `if` statement

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

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.

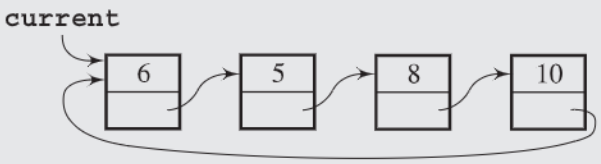
Because of the immediate accessibility of the last node, both `addToDLLTail()` and `deleteFromDLLTail()` execute in constant time $O(1)$.

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 ensure that each process has a fair share of the resource. 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 in 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.

FIGURE 3.13 A circular singly linked list.



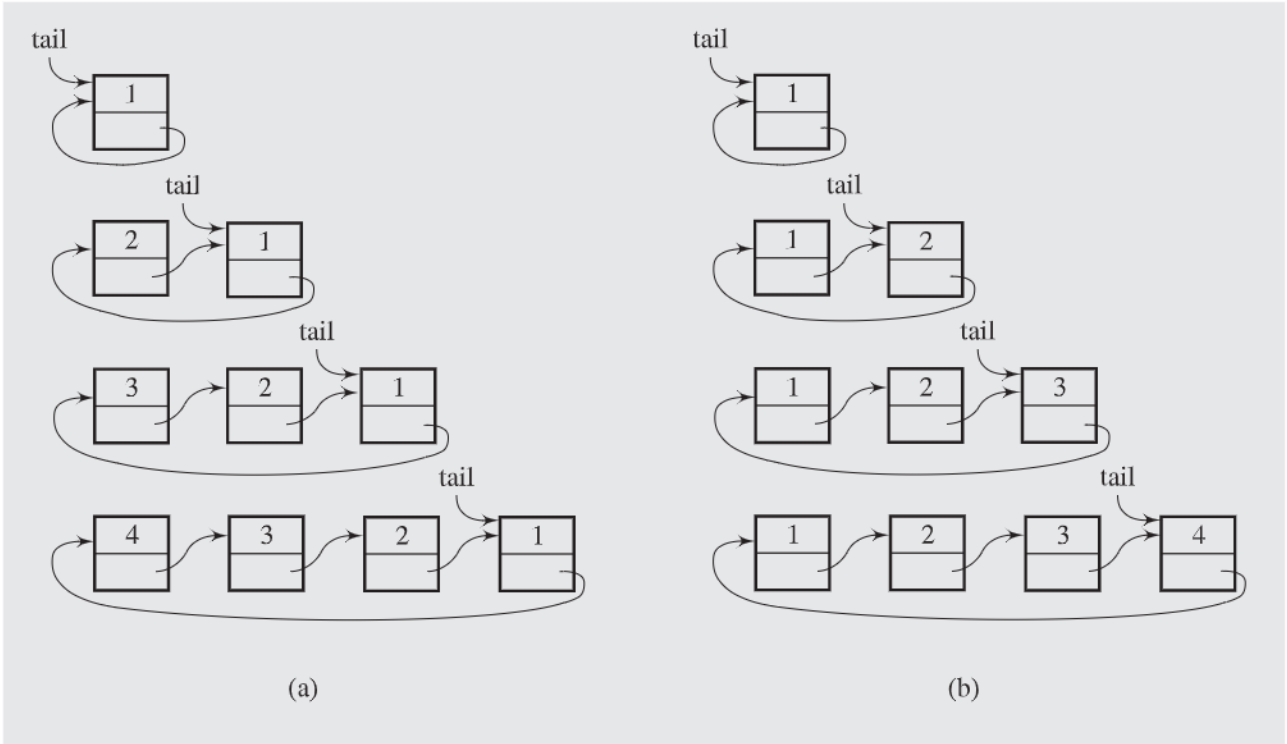
In an implementation of a circular singly linked list, we can use only one permanent pointer, `tail`, to the list even though operations on the list require access to the tail and its successor, the head. To that end, a linear singly linked list as discussed in Section 3.1 uses two permanent pointers, `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 IntSLLNode(el);
        tail->next = tail;
```

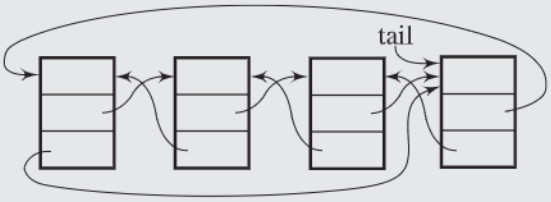
```
        }
        else {
            tail->next = new IntSLLNode(el,tail->next);
            tail = tail->next;
        }
    }
```

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



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 to its predecessor after deleting the node. 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 because there is direct access to the next to last node that needs to be updated in the case of such a deletion. In this list, both insertion and deletion of the tail node can be done in $O(1)$ time.

FIGURE 3.15 A circular doubly linked list.



3.4 Skip Lists

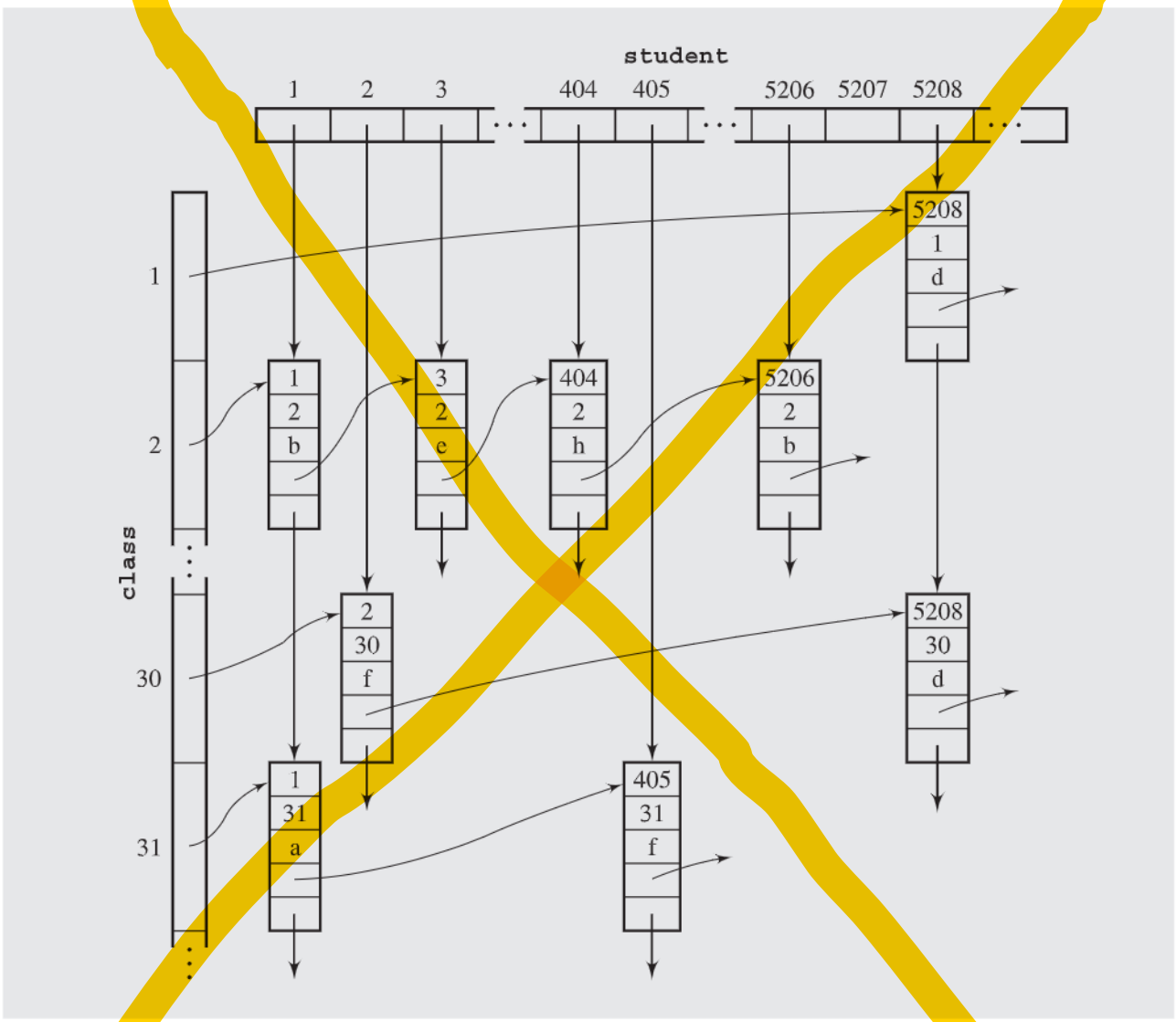
Linked lists have one serious drawback: they require sequential scanning to locate a searched-for element. The search starts from the beginning of the list and stops when either a searched-for element is found or the end of the list is reached without finding this element. Ordering elements on the list can speed up searching, but a sequential search is still required. Therefore, we may think about lists that allow for skipping certain nodes to avoid sequential processing. A *skip list* is an interesting variant of the ordered linked list that makes such a nonsequential search possible (Pugh 1990).

In a skip list of n nodes, for each k and i such that $1 \le k \le \lfloor \lg n \rfloor$ and $1 \le i \le \lfloor n/2^{k-1} \rfloor - 1$, the node in position $2^{k-1} \cdot i$ points to the node in position $2^{k-1} \cdot (i + 1)$. This means that every second node points to the node two positions ahead, every fourth node points to the node four positions ahead, and so on, as shown in Figure 3.16a. This is accomplished by having different numbers of pointers in nodes on the list: half of the nodes have just one pointer, one-fourth of the nodes have two pointers, one-eighth of the nodes have three pointers, and so on. The number of pointers indicates the *level* of each node, and the number of levels is $maxLevel = \lfloor \lg n \rfloor + 1$.

Searching for an element `el` consists of following the pointers on the highest level until an element is found that finishes the search successfully. In the case of reaching the end of the list or encountering an element `key` that is greater than `el`, the search is restarted from the node preceding the one containing `key`, but this time starting from a pointer on a lower level than before. The search continues until `el` is found, or the first-level pointers are followed to reach the end of the list or to find an element greater than `el`. Here is a pseudocode for this algorithm:

```
search(element el)
    p = the nonnull list on the highest level i;
    while el not found and i ≥ 0
        if p->key > el
            p = a sublist that begins in the predecessor of p on level --i;
        else if p->key < el
            if p is the last node on level i
                p = a nonnull sublist that begins in p on the highest level < i;
                i = the number of the new level;
            else p = p->next;
```

FIGURE 3.23 Student grades implemented using linked lists.



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 for 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.

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. A new list is generated with the instruction

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

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

Member Function	Action and Return Value
`void assign(iterator first, iterator last)`	Remove all the nodes in the list and insert into it the elements from the range indicated by iterators `first` and `last`.
`void assign(size_type n, el const T& el = T())`	Remove all the nodes in the list and insert into it `n` copies of (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 last node of the list.
`iterator begin()`	Return an iterator that references the first node of the list.
`const_iterator begin() const`	Return an 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 an iterator that is past the last node of the list.
`iterator erase (iterator i)`	Remove the node referenced by iterator `i` and return an iterator referencing the element after the one removed.
`iterator erase(iterator first, iterator last)`	Remove the nodes in the range indicated by iterators `first` and `last` and return an iterator referencing the element after the last one removed.
`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(iterator i, const T& el = T())`	Insert `el` before the node referenced by iterator `i` and return an iterator referencing the new node.
`void insert(iterator i, size_type n, const T& el)`	Insert `n` copies of `el` before the node referenced by iterator `i`.
`void insert(iterator i, iterator first, iterator 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(size_type n, const T& el = T())`	Construct a list with `n` copies of `el` of type `T`.
`list(iterator first, iterator last)`	Construct a list with the elements from the range indicated by iterators `first` and `last`.

FIGURE 3.24 *(continued)*

`list(const list<T>& lst)`	Copy constructor.
`size_type max_size() const`	Return the maximum number of nodes for the list.
`void merge(list<T>& 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(list<T>& lst, Comp 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()`.
`void pop_back()`	Remove the last node of the list.
`void pop_front()`	Remove the first node of the list.
`void push_back(const T& el)`	Insert `el` at the end of the list.
`void push_front(const T& el)`	Insert `el` at the head of the list.
`void remove(const T& el)`	Remove from the list all the nodes that include `el`.
`void remove_if(Pred f)`	Remove the nodes for which a one-argument Boolean function `f()` returns `true`.
`void resize(size_type n, const T& 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.
`void reverse()`	Reverse the list.
`reverse_iterator rbegin()`	Return an iterator that references the last node of the list.
`const_reverse_iterator rbegin() const`	Return an iterator that references the last node of the list.
`reverse_iterator rend()`	Return an iterator that is before the first node of the list.
`const_reverse_iterator rend() const`	Return an iterator that is before the first node 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(Comp f)`	Sort elements of the list in the order specified by a two-argument Boolean function `f()`.
`void splice(iterator i, list<T>& lst)`	Remove the nodes of list `lst` and insert them into the list before the position referenced by iterator `i`.
`void splice(iterator i, list<T>& lst, iterator 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(iterator i, list<T>& lst, iterator first, iterator 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`.

Continues

FIGURE 3.24 *(continued)*

`void swap(list<T>& 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(Comp f)`	Remove duplicate elements from the sorted list where being a duplicate is specified by a two-argument Boolean function `f()`.

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). The 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>
#include <functional>

using namespace std;

int 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)
```

FIGURE 3.25 *(continued)*

```
    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();
    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)
    return 0;
}
```

3.8 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, because 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 directly to the tenth cell. 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,