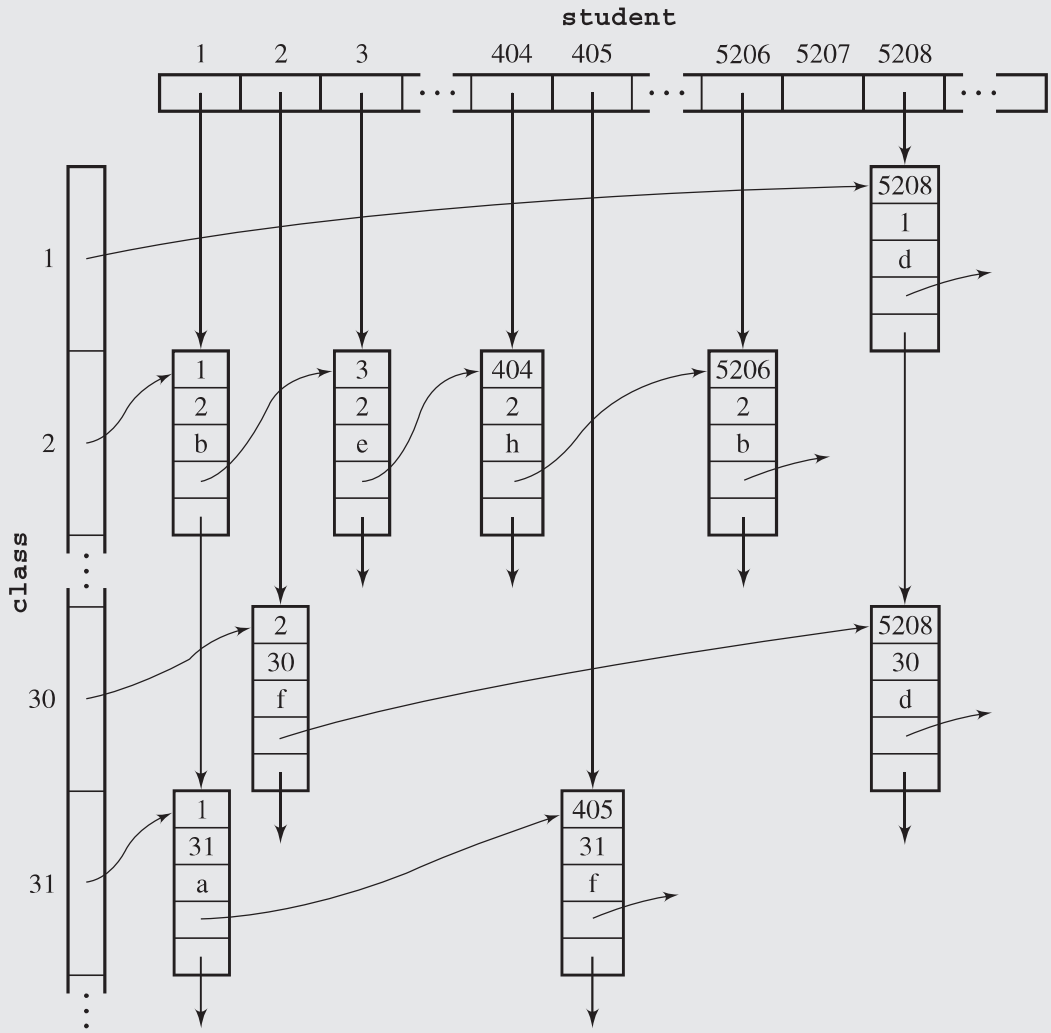
FIGURE 3.23 Student grades implemented using linked lists.



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,