of new nodes at the front of a list and another at the end. There is a general tendency for all methods to improve their efficiency with the size of the file. Also, insertion at the tail gave consistently better results than insertion at the head. 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 in 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 and that of 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* because the 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.

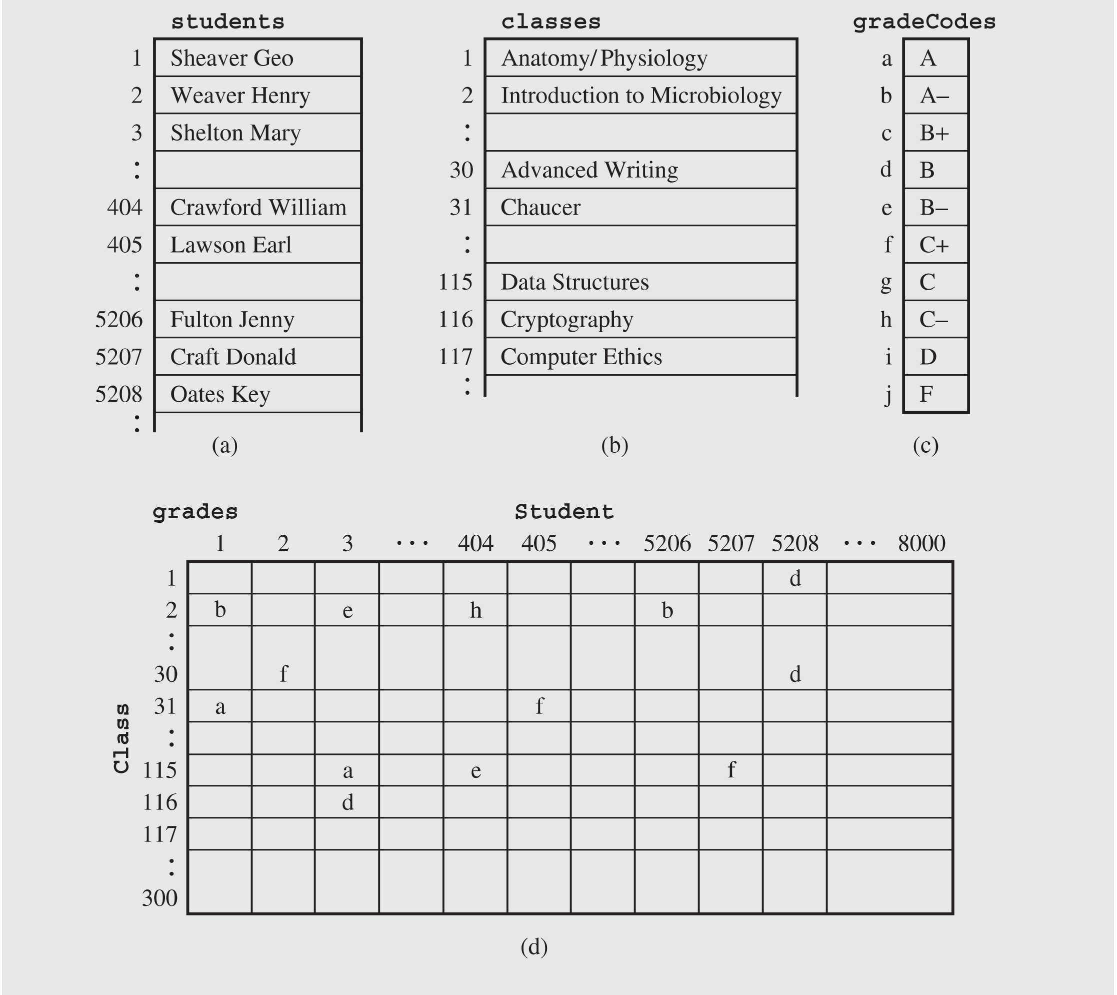
As an example, consider the problem of storing grades for all students in a university for a certain semester. Assume that there are 8,000 students and 300 classes. A natural implementation is a two-dimensional array `grades` where student numbers are indexes of the columns and class numbers are the 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 2 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 that requires only one byte of storage.

The entire table (Figure 3.21d) occupies 8,000 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.

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

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



students

1	Sheaver Geo
2	Weaver Henry
3	Shelton Mary
:	
404	Crawford William
405	Lawson Earl
:	
5206	Fulton Jenny
5207	Craft Donald
5208	Oates Key
:	

(a)

classes

1	Anatomy/Physiology
2	Introduction to Microbiology
:	
30	Advanced Writing
31	Chaucer
:	
115	Data Structures
116	Cryptography
117	Computer Ethics
:	

(b)

gradeCodes

a	A
b	A–
c	B+
d	B
e	B–
f	C+
g	C
h	C–
i	D
j	F

(c)

grades — Student (columns), Class (rows)

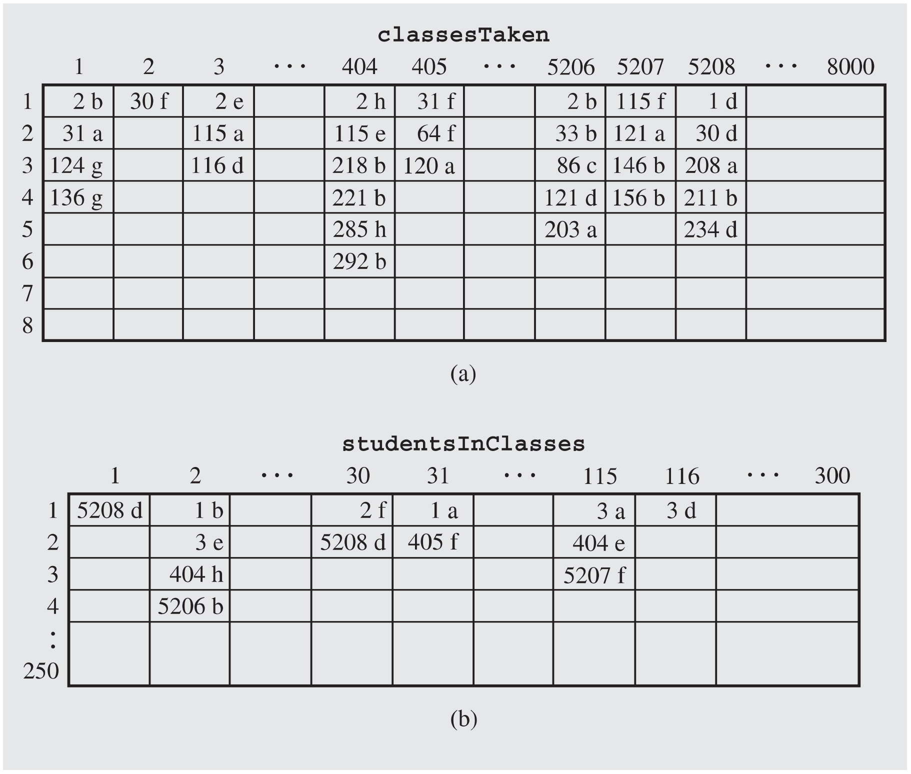
	1	2	3	⋯	404	405	⋯	5206	5207	5208	⋯ 8000
1										d	
2	b		e		h			b			
: 30		f								d	
31	a					f					
:											
115			a		e				f		
116			d								
117											
: 300											

(d)

A better solution is to use two two-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 members: a student or class number and a grade. We assume that a student can take at most 8 classes and that there can be at most 250 students signed up for a class. We need two arrays, because 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 2 bytes to store an integer. With this new structure, 3 bytes are needed for

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									

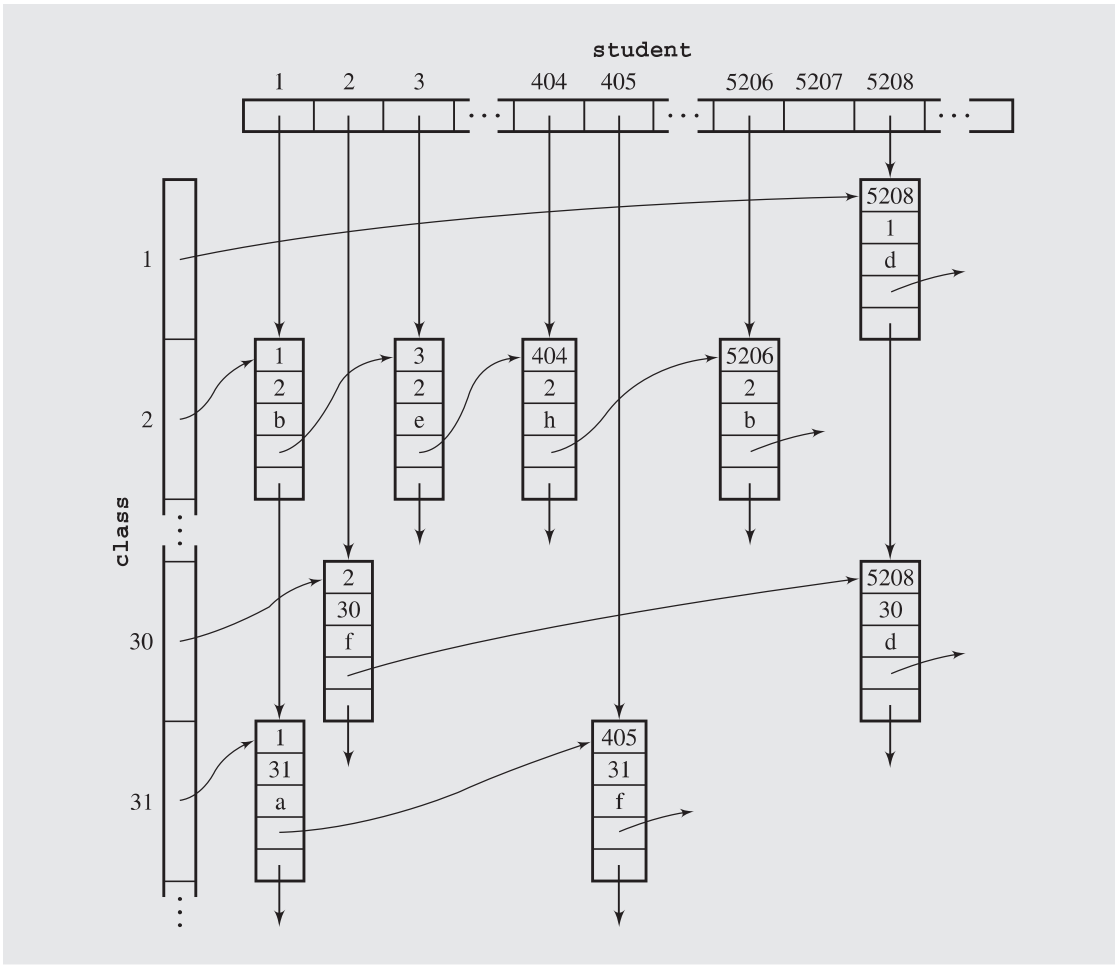
(b)

each cell. Therefore, `classesTaken` occupies 8,000 students · 8 classes · 3 bytes = 192,000 bytes, `studentsInClasses` occupies 300 classes · 250 students · 3 bytes = 225,000 bytes; 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.

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 because most classes have fewer than 250 students, and most students take fewer than 8 classes. This structure is also inflexible: if a class can be taken by more than 250 students, a problem occurs that has to be circumvented in an artificial way. One way is to create a nonexistent class that 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 2 bytes and one node occupies 9 bytes, the entire structure can be stored in 8,000 students · 4

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.