DATA STRUCTURES and ALGORITHMS in C++

FOURTH EDITION

Chapter 3: Linked Lists

Objectives

Looking ahead – in this chapter, we'll consider:

- Singly Linked Lists
- Doubly Linked Lists
- Circular Lists
- Self-Organizing Lists
- Lists in the Standard Template Library

Introduction

- Arrays are useful in many applications but suffer from two significant limitations
  - The size of the array must be known at the time the code is compiled
  - The elements of the array are the same distance apart in memory, requiring potentially extensive shifting when inserting a new element
- This can be overcome by using ***linked lists***, collections of independent memory locations (***nodes***) that store data and links to other nodes
- Moving between the nodes is accomplished by following the links, which are the addresses of the nodes
- There are numerous ways to implement linked lists, but the most common utilizes pointers, providing great flexibility

Singly Linked Lists

- If a node contains a pointer to another node, we can string together any number of nodes, and need only a single variable to access the sequence
- In its simplest form, each node is composed of a datum and a link (the address) to the next node in the sequence
- This is called a ***singly linked list***, illustrated in figure 3.1
- Notice the single variable *p* used to access the entire list
- Also note the last node in the list has a null pointer (\ )

Singly Linked Lists (continued)

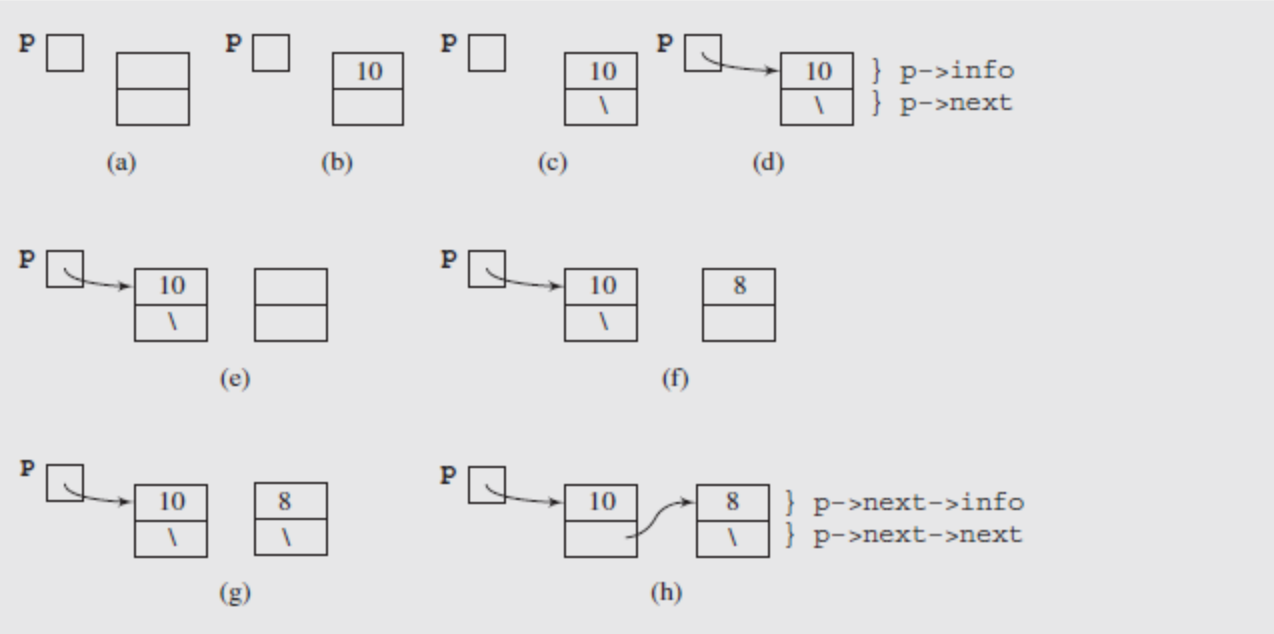


Fig. 3.1 A singly linked list

- The nodes in this list are objects created from the following class definition

Singly Linked Lists (continued)

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

- As can be seen here and in the previous figure, a node consists of two data members, `info` and `next`

Singly Linked Lists (continued)

- The `info` member stores the node's information content; the `next` member points to the next node in the list
- Notice that the definition refers to the class itself, because the `next` pointer points to a node of the same type being defined
- Objects that contain this type of reference are called ***self-referential objects***
- The definition also contains two constructors
  - One sets the next pointer to 0 and leaves `info` member undefined
  - The other initializes both members
- The remainder of figure 3.1 shows how a simple linked list can be created using this definition

Singly Linked Lists (continued)

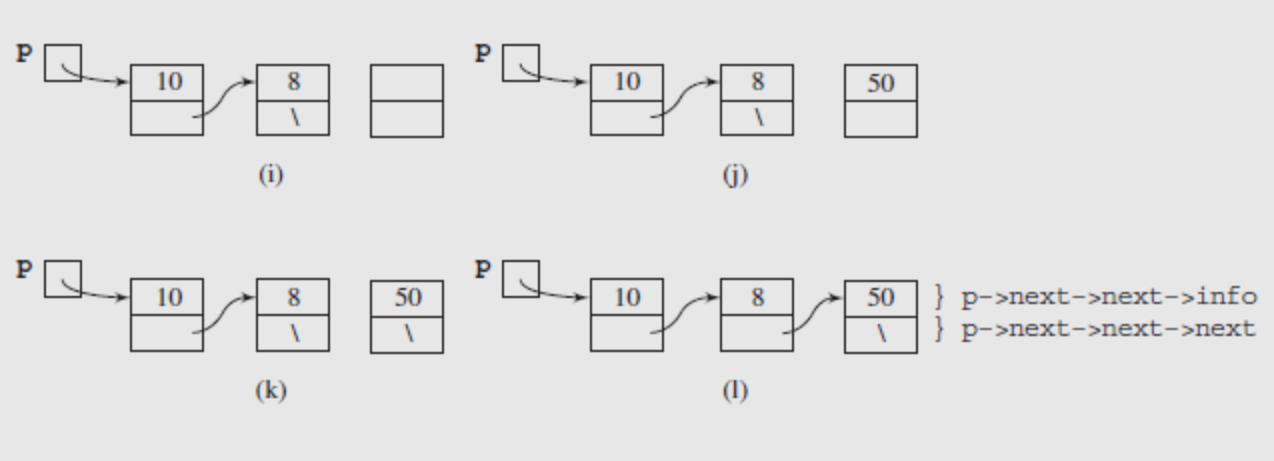


Fig. 3.1 (continued) A singly linked list

- This example illustrates a disadvantage of single-linked lists: the longer the list, the longer the chain of `next` pointers that need to be followed to a given node
- This reduces flexibility, and is prone to errors
- An alternative is to use an additional pointer to the end of the list, as seen in figure 3-2 (pages 78-80) and figure 3-3(a)

Singly Linked Lists (continued)

- The code uses two classes
  - `IntSLLNode`, which defines the nodes of the list
  - `IntSLList`, which defines two pointers, `head` and `tail`, as well as various member functions to manipulate the list
- An example of this list in shown in figure 3-3

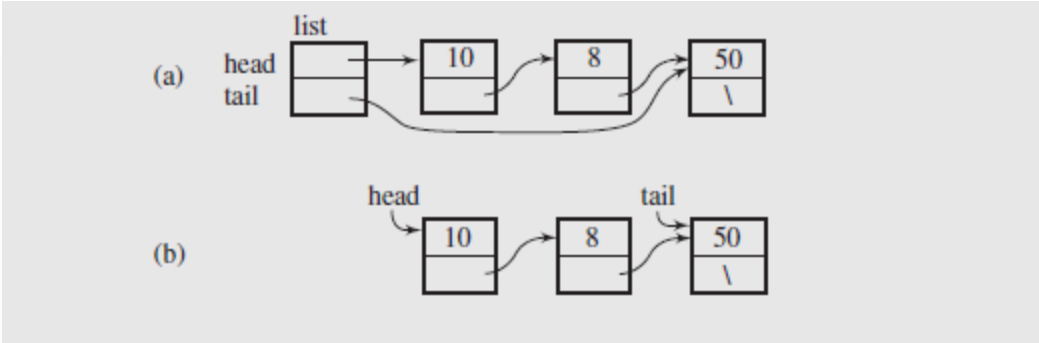


Fig. 3-3 A singly linked list of integers

Singly Linked Lists (continued)

- While figure 3-3a shows the structure of the list, a more common representation is shown in figure 3-3b
- Let's consider some common operations on this type of list
- Insertion
  - Inserting a node at the beginning of a list is straightforward
  - First, a new node is created (figure 3-4a)
  - The `info` member of the node is initialized (figure 3-4b)
  - The `next` member is initialized to point to the first node in the list, which is the current value of `head` (figure 3-4c)
  - `head` is then updated to point to the new node (figure 3-4d)

Singly Linked Lists (continued)

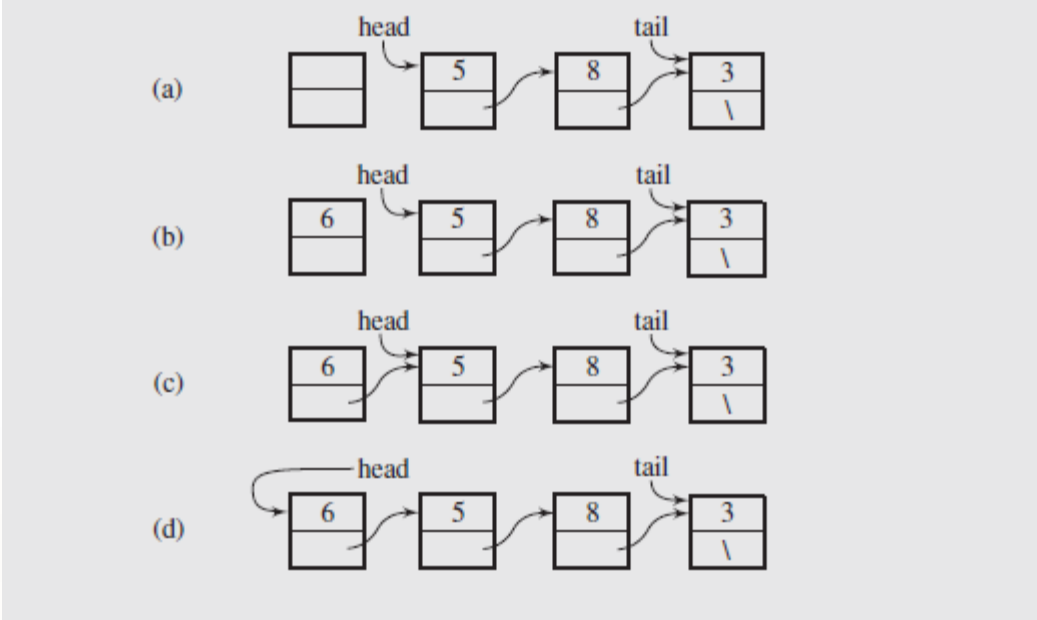


Fig. 3-4 Inserting a new node at the beginning of a singly linked list

– Note that if the list is initially empty, both `head` and `tail` would be set to point to the new node

Singly Linked Lists (continued)

- Insertion (continued)
  - Inserting a node at the end of a list is likewise easy to accomplish as illustrated in the next slide
  - The new node is created and the `info` member of the node is initialized (figures 3-5a and 3-5b)
  - The `next` member is initialized to null, since the node is at the end of the list (figure 3-5c)
  - The `next` member of the current last node is set to point to the new node (figure 3-5d)
  - Since the new node is now the end of the list, the `tail` pointer has to be updated to point to it (figure 3-5e)
  - As before, if the list is initially empty, both `head` and `tail` would be set to point to the new node

Singly Linked Lists (continued)

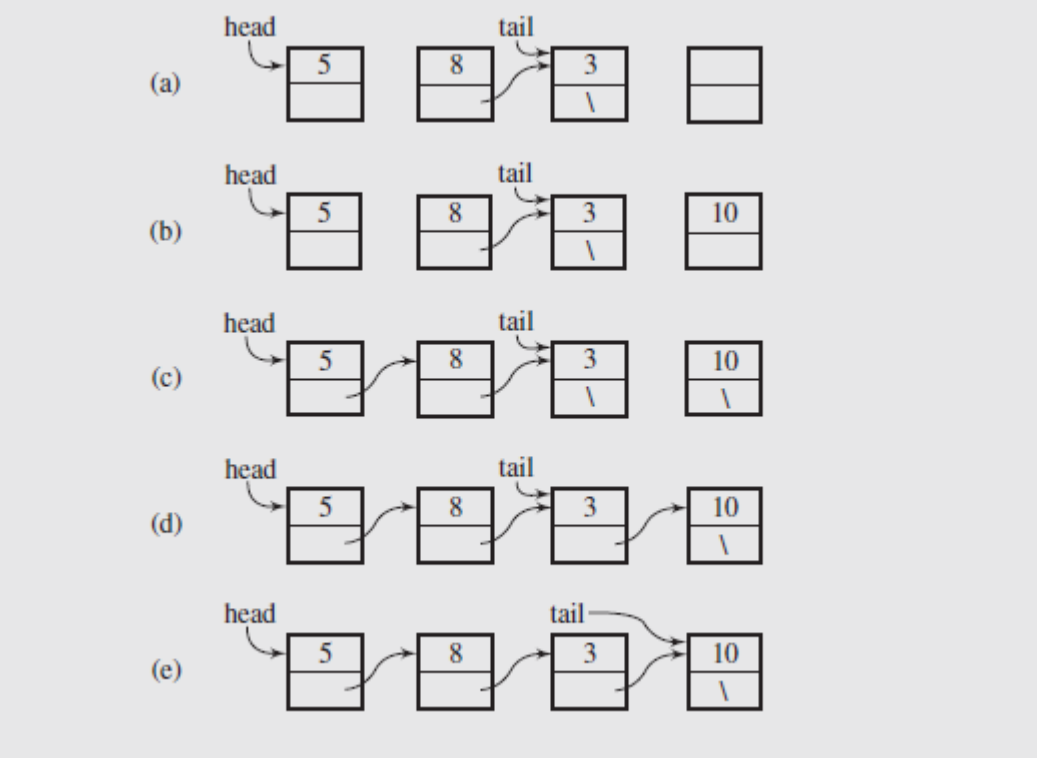


Fig. 3-5 Inserting a new node at the end of a singly linked list

Singly Linked Lists (continued)

- Deletion
  - The operation of deleting a node consists of returning the value stored in the node and releasing the memory occupied by the node
  - Again, we can consider operations at the beginning and end of the list
  - To delete at the beginning of the list, we first retrieve the value stored in the first node (`head → info`)
  - Then we can use a temporary pointer to point to the node, and set `head` to point to `head → next`
  - Finally, the former first node can be deleted, releasing its memory
  - These operations are illustrated in figure 3.6(a) – (c)

Singly Linked Lists (continued)

- Deletion (continued)

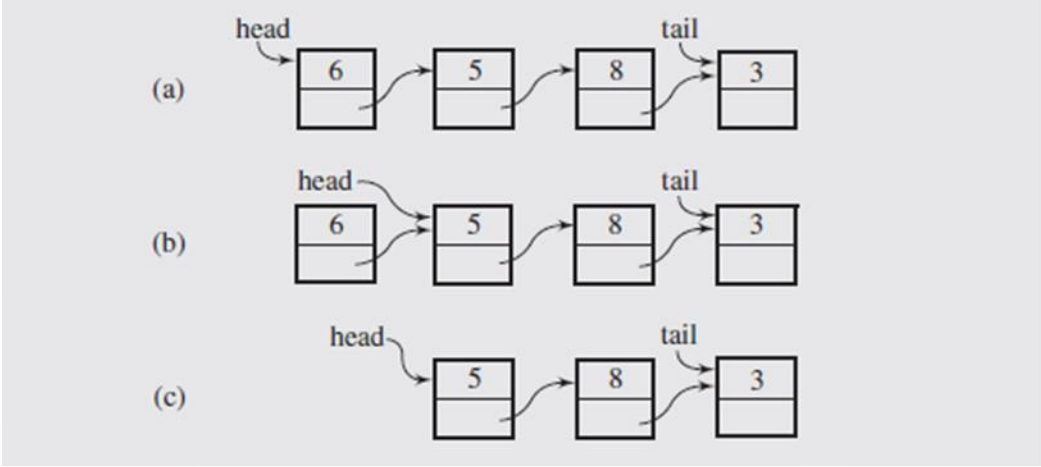


Fig. 3-6 Deleting a node at the beginning of a singly linked list

  - Two special cases exist when carrying out this deletion
  - The first arises when the list is empty, in which case the caller must be notified of the action to take
  - The second occurs when a single node is in the list, requiring that `head` and `tail` be set to null to indicate the list is now empty

Singly Linked Lists (continued)

- Deletion (continued)
  - Deleting at the end of a list requires additional processing
  - This is because the `tail` pointer must be backed up to the previous node in the list
  - Since this can't be done directly, we need a temporary pointer to traverse the list until `tmp → next = tail`
  - This is illustrated in figures 3-7 (a) – (c) in the next slide
  - Once we have located that node, we can retrieve the value contained in `tail → info`, delete that node, and set `tail = tmp`
  - This is illustrated in figures 3-7 (d) – (f) in the next slide

Singly Linked Lists (continued)

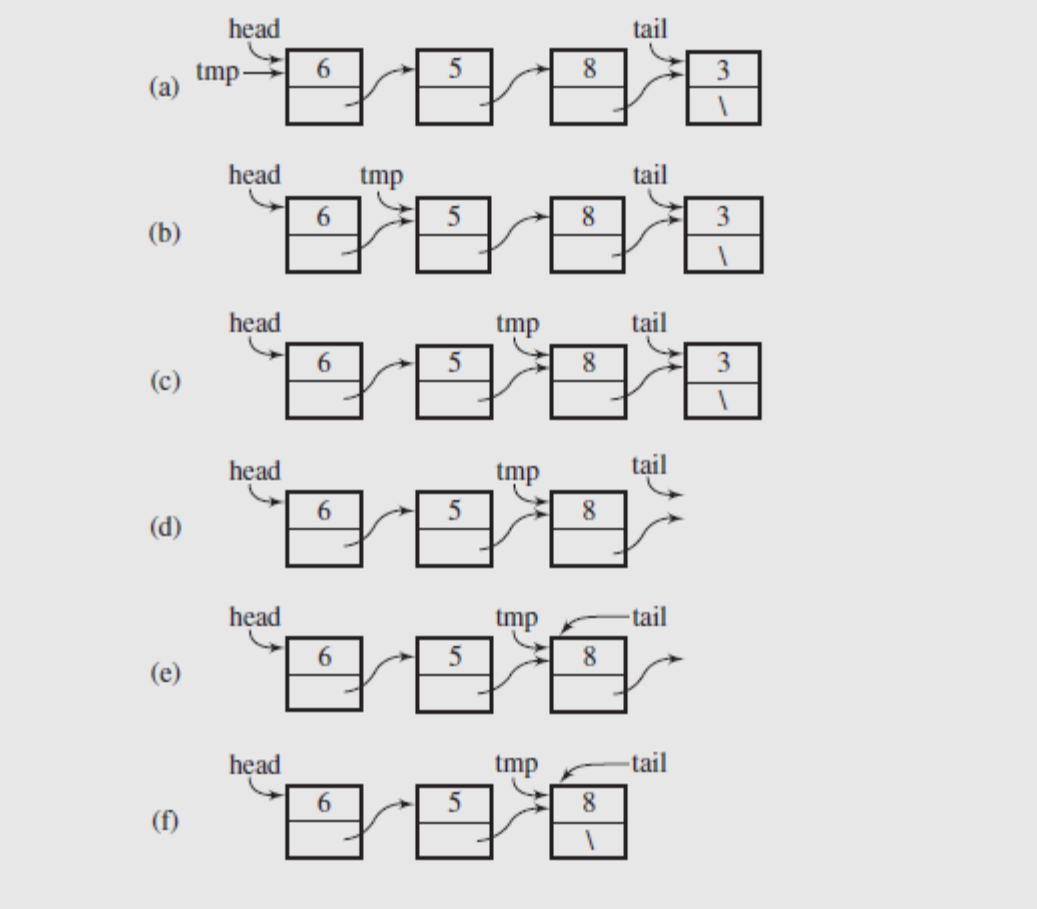


Fig. 3-7 Deleting a node from the end of a singly linked list

Singly Linked Lists (continued)

- Deletion (continued)
  - The same special cases apply to deleting a node from the end of a list as they did to deleting a node from the beginning of a list
  - Now these deletion operations simply delete the physically first or last node in the list
  - What if we want to delete a specific node based on its `info` member?
  - In that case we have to locate the specific node, then link around it by linking the previous node to the following node
  - But again, to do this we need to keep track of the previous node, and we need to keep track of the node containing the target value
  - This will require two pointers, as shown in figure 3-8 (a) – (d)

Singly Linked Lists (continued)

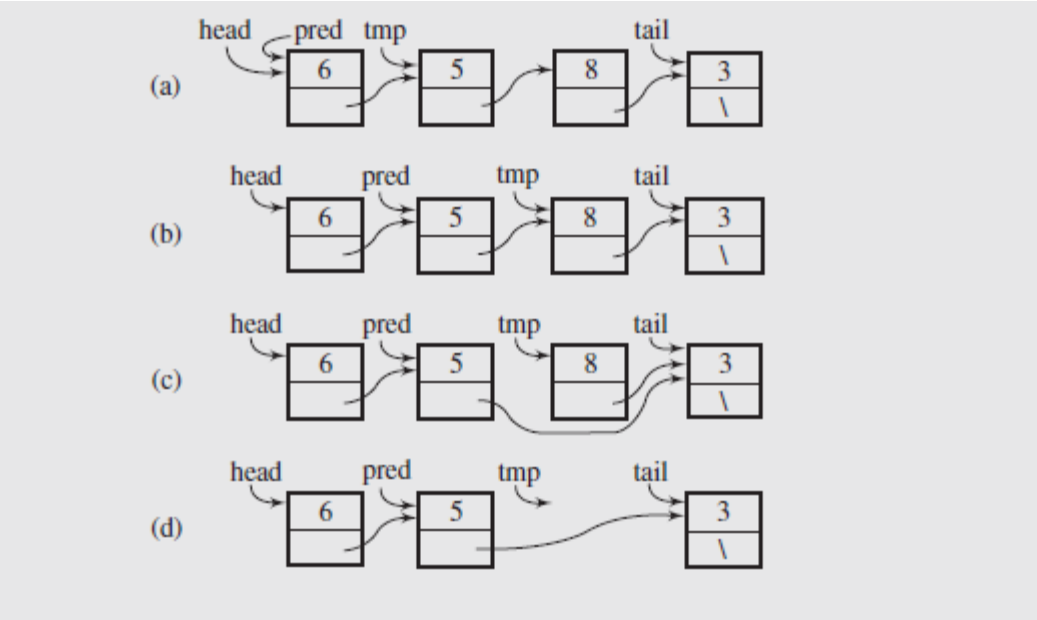


Fig. 3-8 Deleting a node from a singly linked list

- As can be seen, the two extra pointers, `pred` and `tmp`, are initialized to the first and second nodes in the list
- They traverse the list until `tmp → info` matches the target value

Singly Linked Lists (continued)

- Deletion (continued)
  - At that point, we can set `pred → next = tmp → next` which "bypasses" the target node, allowing it to be deleted
  - There are several cases to consider when this type of deletion is carried out
    - Removing a node from an empty list or trying to delete a value that isn't in the list
    - Deleting the only node in the list
    - Removing the first or last node from a list with at least two nodes

Singly Linked Lists (continued)

- Searching
  - The purpose of a search is to scan a linked list to find a particular data member
  - No modification is made to the list, so this can be done easily using a single temporary pointer
  - We simply traverse the list until the `info` member of the node `tmp` points to matches the target, or `tmp → next` is null
  - If the latter case occurs, we have reached the end of the list and the search fails

Reverse the list

We want to reverse the list using the existing nodes of the list, i.e., without creating new nodes.

Reverse the list

We want to reverse the list using the existing node of the list, i.e., without creating new nodes.

```
void reverse() {
  if (head == NULL || head -> next == NULL) return;
   Node* p = head;
   Node* q = p-> next; p->next = NULL;
  while (q != NULL) {
    Node* temp = q -> next;
    q->next = p;
    p = q;
    q = temp;
   }
  head = p;
 }
```

Remove negative items from the list

Example:

List: -3, 4, 5, -2, 11 becomes 4, 5, 11

We will write this one recursively.

Doubly Linked Lists

- The difficulty in deleting a node from the end of a singly linked list points out one major limitation of that structure
- We continually have to scan to the node just before the end in order to delete correctly
- If the nature of processing requires frequent deletions of that type, this significantly slows down operations
- To address this problem, we can redefine the node structure and add a second pointer that points to the previous node
- Lists constructed from these nodes are called ***doubly linked lists***, one of which is shown in figure 3-9

Doubly Linked Lists (continued)

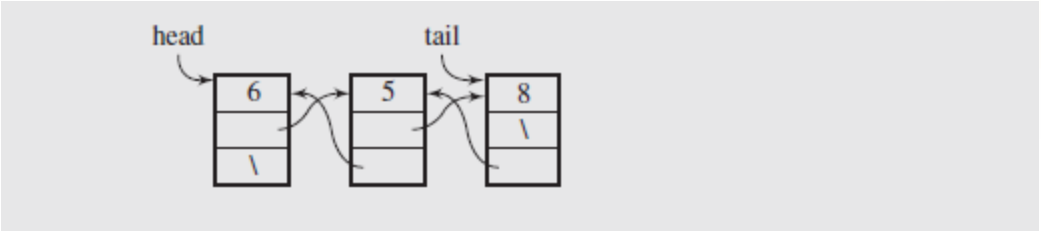


Fig. 3-9 A doubly linked list

- The code that implements and manipulates this is shown in part in figure 3-10 (pages 91 and 92)
- The methods that manipulate these types of lists are slightly more complicated than their singly linked counterparts
- However, the process is still straightforward as long as one keeps track of the pointers and their relationships
- We'll look at two: inserting and removing a node at the end of the list

Doubly Linked Lists (continued)

- Insertion of a new node requires the following steps:
  - First, the new node is created and the data member is initialized
  - Since the node is being inserted at the end of the list, its `next` member is set to null
  - The `prev` member is set to `tail` to link it back to the former end of the list
  - The `tail` pointer is now set to point to this new node
  - To complete the link, the `next` member of the previous node is set to point to the new node
  - These steps are illustrated in figure 3.11(a) – (f)

Doubly Linked Lists (continued)

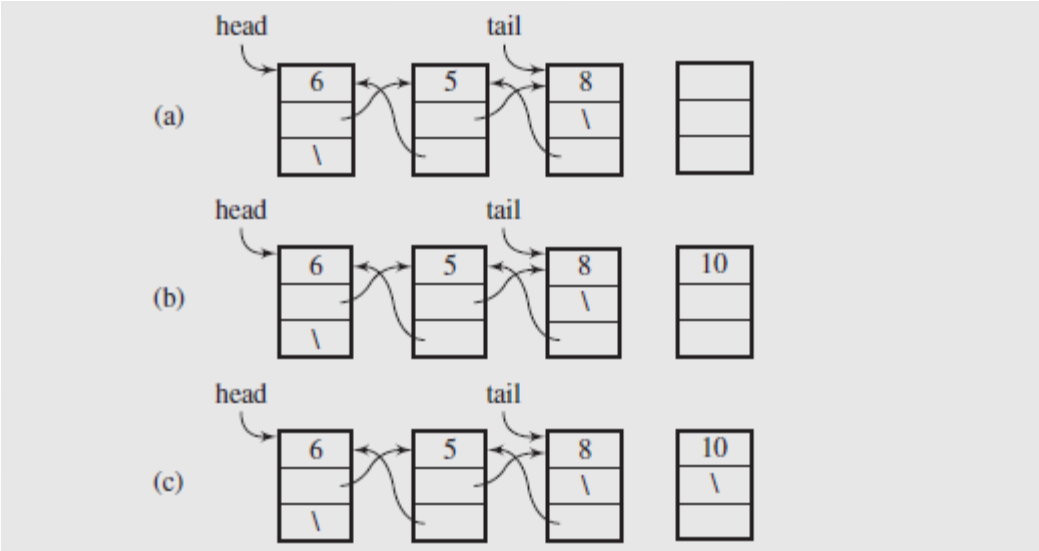


Fig. 3-11 Adding a new node at the end of a doubly linked list

Doubly Linked Lists (continued)

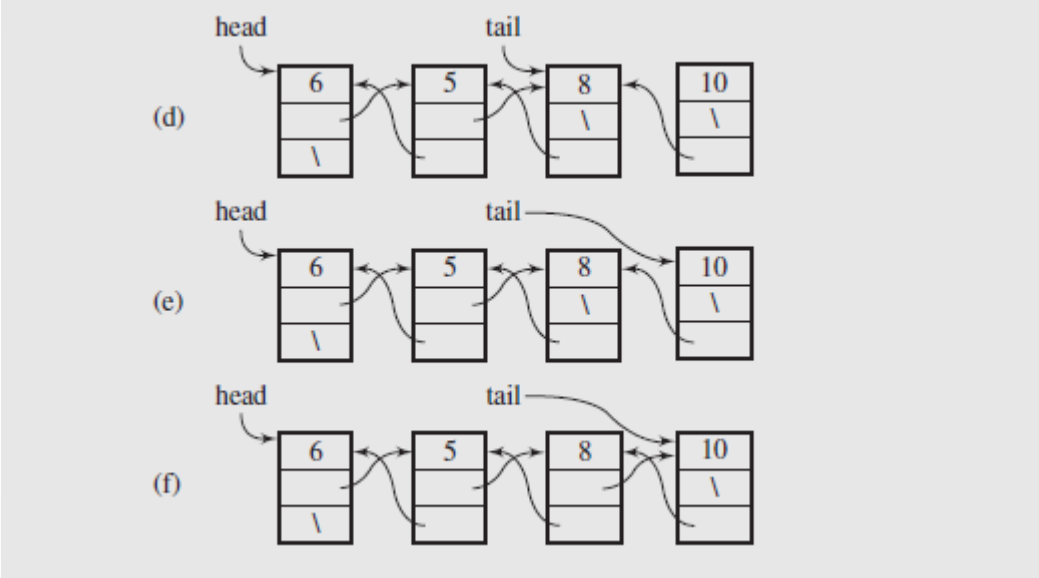


Fig. 3-11 (continued) Adding a new node at the end of a doubly linked list

Doubly Linked Lists (continued)

- Insertion (continued)
  - A special case exists if the node being inserted is the only node in the list
  - In this case there is no previous node, so both `head` and `tail` point to the new node and in the last step, `head` would be set to point to the new node
- Deletion
  - Deleting a node from the end of a doubly linked list is also easy, because there is a direct link to the previous node in the list
  - This eliminates the need to traverse the list to find the previous node
  - To do this, we retrieve the data member from the node, then set `tail` to the node's predecessor

Doubly Linked Lists (continued)

- Deletion
  - The node can then be deleted, and the `next` pointer of the new last node set to null
  - This process is illustrated in figure 3-12(a) – (d)
  - A couple of special cases need to be dealt with
    - If the node being deleted is the only node in the list, `head` and `tail` need to be set to null
    - Also, if the list is empty, an attempt to delete a node should be handled and reported to the user

Doubly Linked Lists (continued)

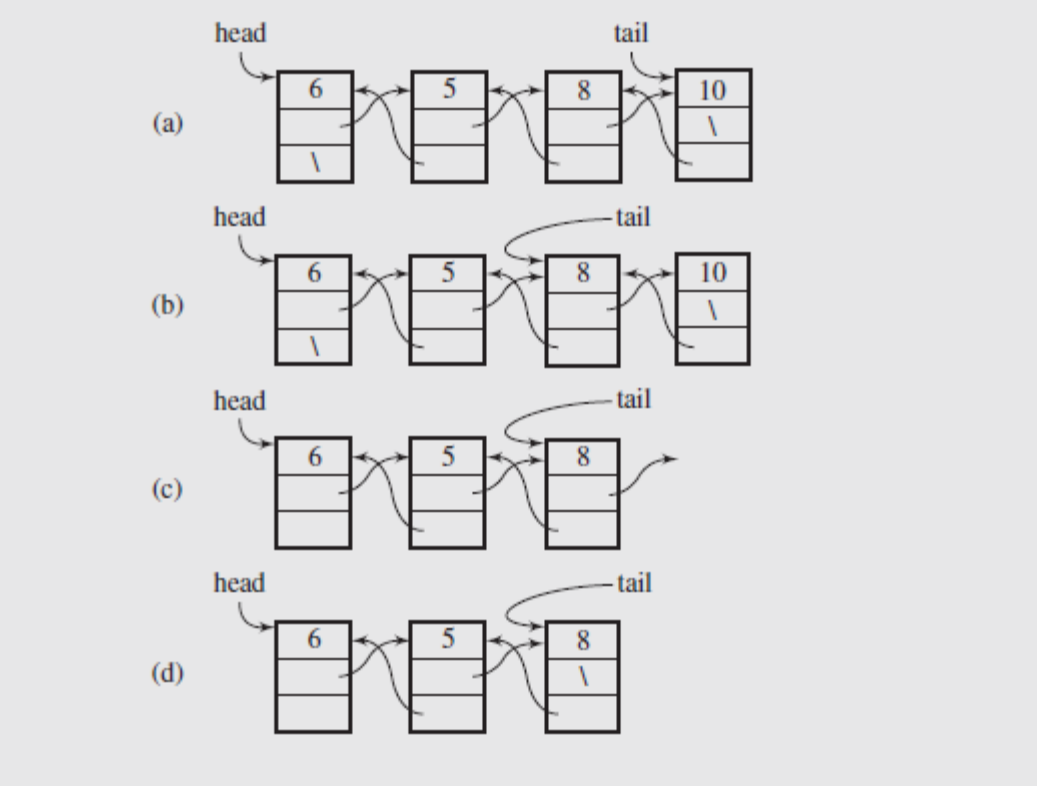


Fig. 3-12 Deleting a node from the end of a doubly linked list

Circular Lists

- Another useful arrangement of nodes is the ***circular list***; in this structure the nodes form a ring

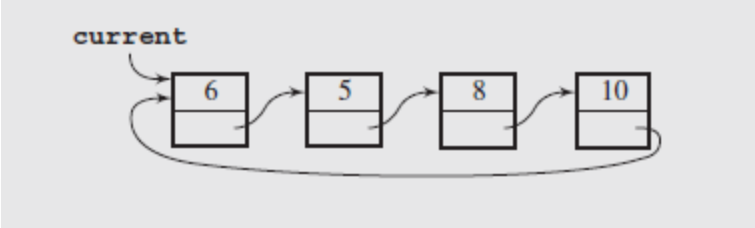


Fig. 3-13 A circular singly linked list

- The implementation requires only one permanent pointer (usually referred to as `tail`)
- Insertions at the front and the end of this type of list are shown in figure 3-14(a) and (b)

Circular Lists (continued)

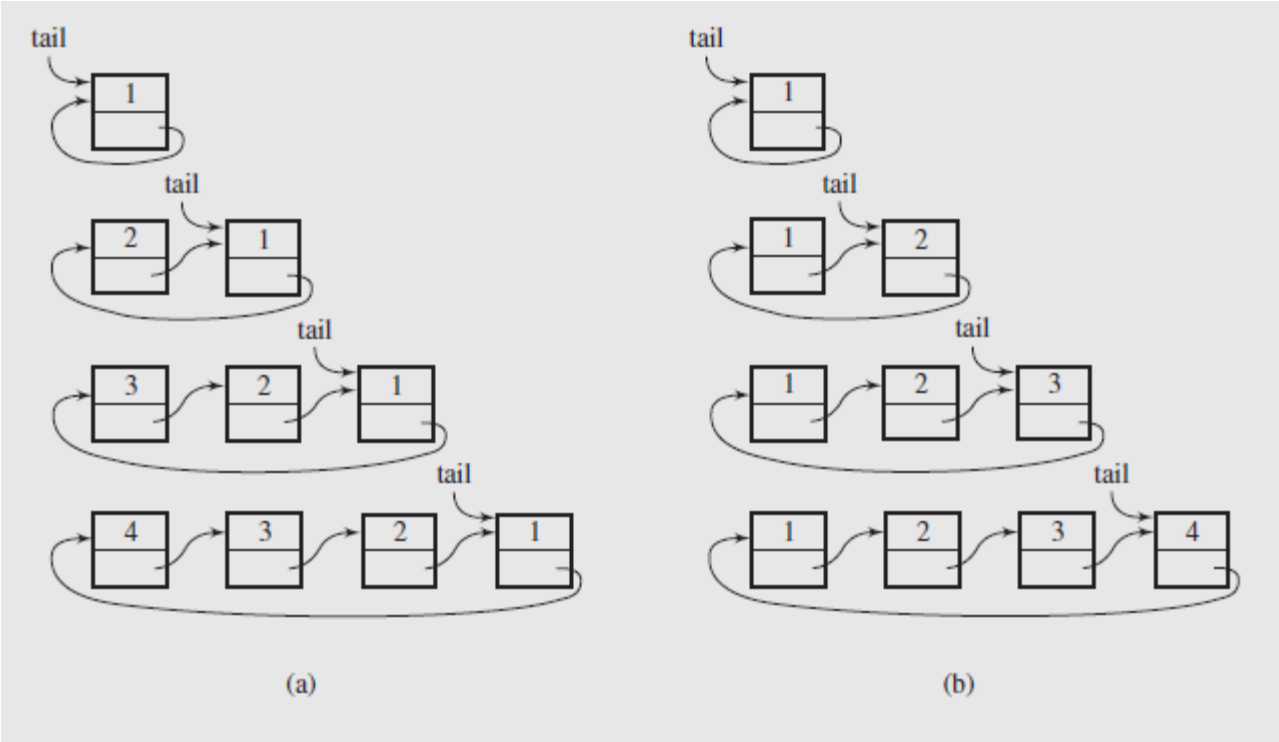


Fig. 3-14 Inserting nodes (a) at the front of a circular singly linked list and (b) at its end

Circular Lists (continued)

- Code to insert a node at the end of the list follows:

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

Circular Lists (continued)

- The simplicity of this list does present a few problems
  - Deleting nodes requires a loop to locate the predecessor of the `tail` node, much as we saw with singly linked lists
  - Operations that require processing the list in reverse are going to be inefficient
- To deal with this, the list can be made doubly linked
- This forms two rings, one going forward through the `next` pointers, and the other backwards through the `prev` pointers
- This is illustrated in figure 3-15

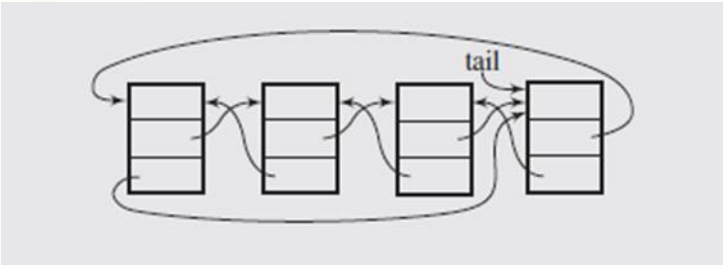


Fig. 3-15 A circular doubly linked list

Self-Organizing Lists

- Skip lists were introduced as a technique to speed up the searching process in lists
- Another approach is based on dynamically re-organizing the lists as they are used
- There are several ways to accomplish this, and this section focuses on four approaches, illustrated in figure 3-18(a) - (d):
  - ***Move-to-front***: when found, the target is moved to the front of the list
  - ***Transpose***: when the element is found, it is swapped with its predecessor in the list
  - ***Count***: the list is ordered by frequency of access
  - ***Ordering***: the list is ordered based on the natural nature of the data

Self-Organizing Lists (continued)

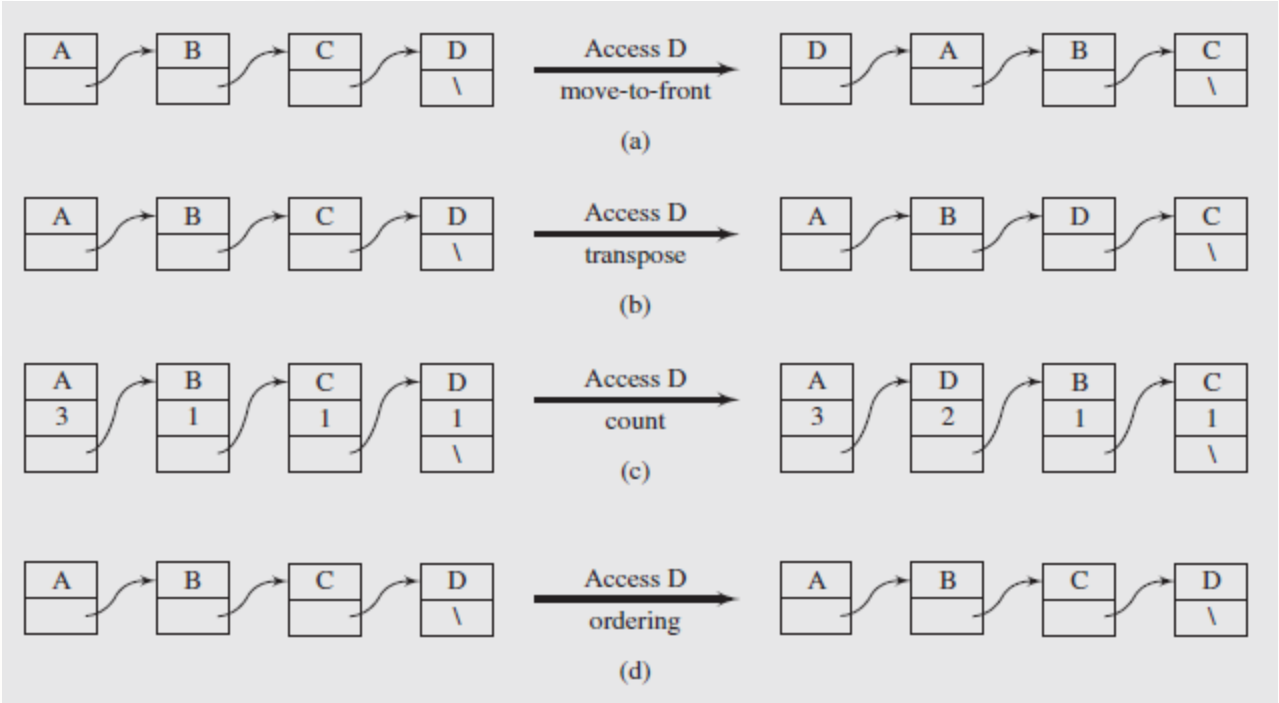


Fig. 3-18 Accessing an element on a linked list and changes on the list depending on the self-organization technique applied: (a) move-to-front method, (b) transpose method, (c) count method, and (d) ordering method, in particular, alphabetical ordering, which leads to no change

Self-Organizing Lists (continued)

- For the first three techniques, new information is in a node placed at the end of the list
- In the ordering technique, placement is based on maintaining the order of the elements
- These operations are illustrated in figure 3.18 (e) and (f)

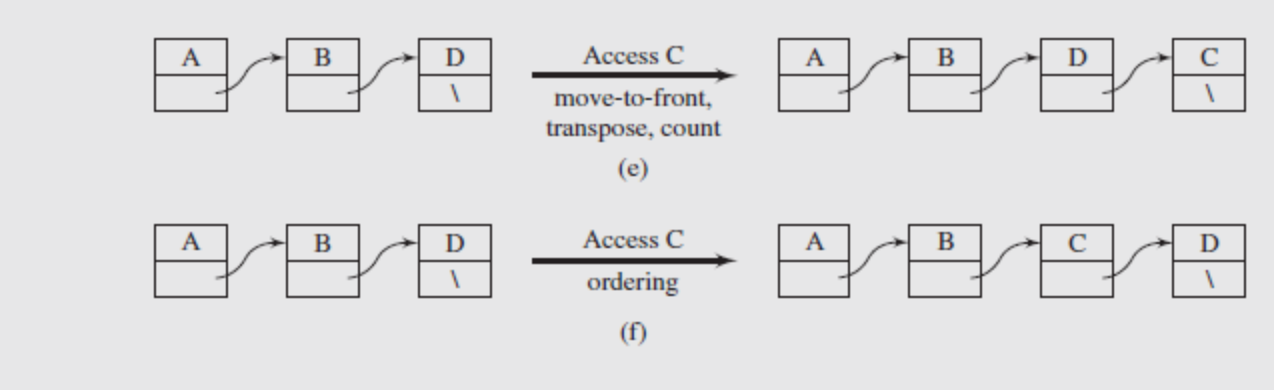


Fig. 3-18 (continued) In the case when the desired element is not in the list, (e) the first three methods add a new node with this element at the end of the list and (f) the ordering method maintains an order on the list

Self-Organizing Lists (continued)

- An example of searching lists organized in this manner is shown in figure 3-19 (page 103)
- As the example indicates, the goal of the first three techniques is to place the most likely looked-for items near the beginning of the list
- Ordering, on the other hand, uses properties of the data itself to organize the list
- Counting can be looked at as ordering, but often it is simply frequency of access and stored as separate information
- Sample runs of these techniques are shown in figure 3-20

Self-Organizing Lists (continued)

- Three of the techniques use both insertion at the head and the tail of the list for comparison
- Notice there is a general improvement as the size of the file increases
- Earlier on, the focus is on inserting new items, while later processing is concerned with organization of existing items
- From the figures, move-to-front and count perform better than the others
- A skip list is included for comparison; while its numbers are quite different, the data is analyzed only in terms of comparisons, not the effort of linking and relinking nodes

Self-Organizing Lists (continued)

different words/all words	189/362	1448/7349	3049/12948	6835/93087
optimal	29.7	15.3	15.5	7.6
plain (head)	79.5	76.3	81.0	86.6
plain (tail)	66.1	48.8	43.4	19.3
move to front (head)	61.8	29.2	35.8	15.0
move to front (tail)	58.2	29.7	35.7	14.3
transpose (head)	65.1	72.4	77.8	75.4
transpose (tail)	78.6	39.6	43.0	18.2
count	57.1	30.2	35.9	13.4
alphabetical order	55.3	50.2	53.8	54.9
skip list	11.1	4.8	4.3	3.6

Fig. 3-20 Measuring the efficiency of different methods using formula (number of data comparison)/(combined length) expressed in percentages

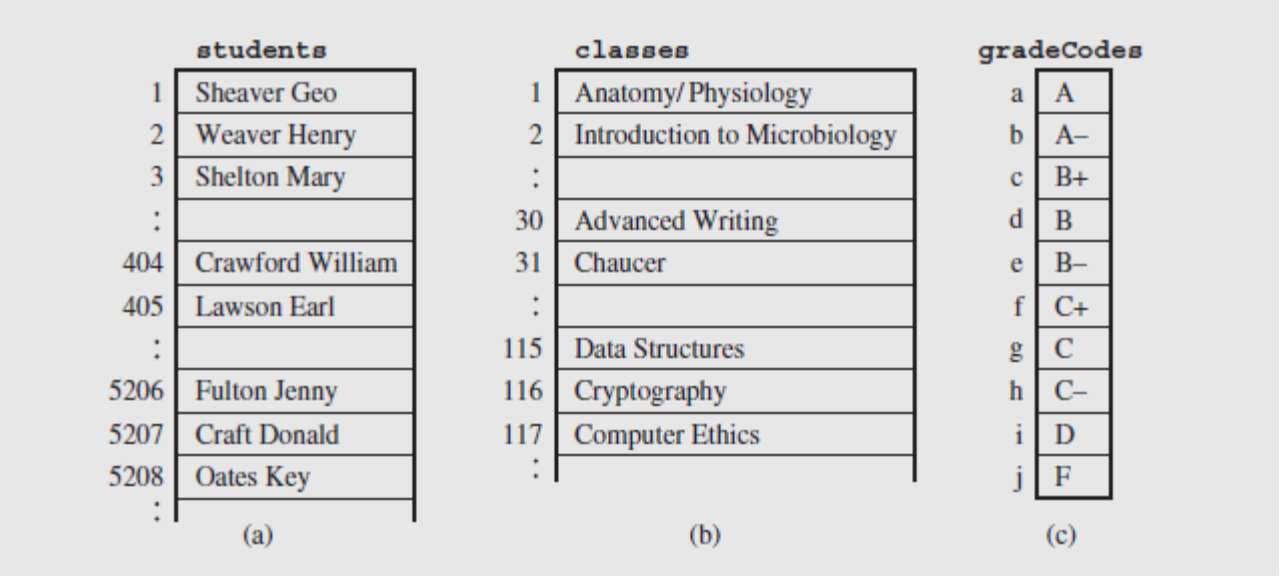
- The conclusion to be drawn is that for smaller lists, simple linking suffices
- As the amount of data and frequency of access increases, other structures should be employed

Sparse Tables

- Tables are a data structure of choice in many applications due to their ease of implementation, use, and access
- However, in many cases the size of the table can lead to difficulties
- This is especially true if the table is mostly unoccupied, known as a ***sparse table***
- Alternative representations using linked lists can be more useful in cases like this
- Consider the example presented in the text, where we want to store the grades for 8000 students in 300 classes

Sparse Tables (continued)

- Student numbers can represent rows, and course numbers columns
- To save space, grades can be encoded using single characters
- The three tables that support this representation are shown in figure 3-21 (a) – (c)



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)

Fig. 3-21 Arrays and sparse table used for storing student grades

Sparse Tables (continued)

- The actual grade table is in figure 3-21(d)

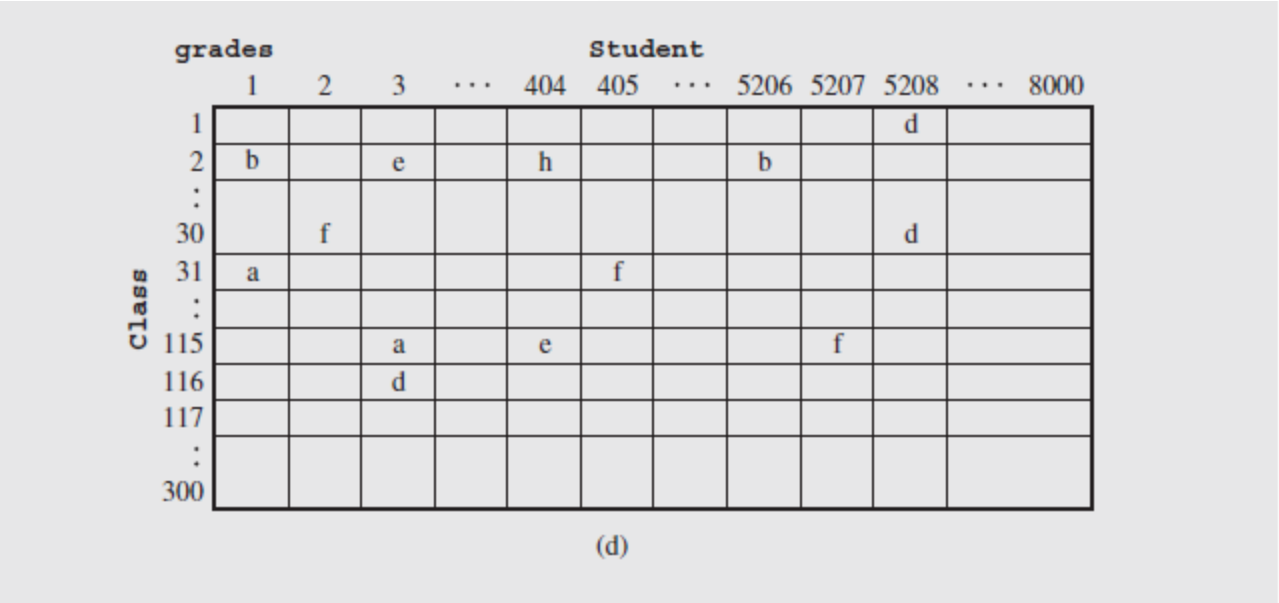


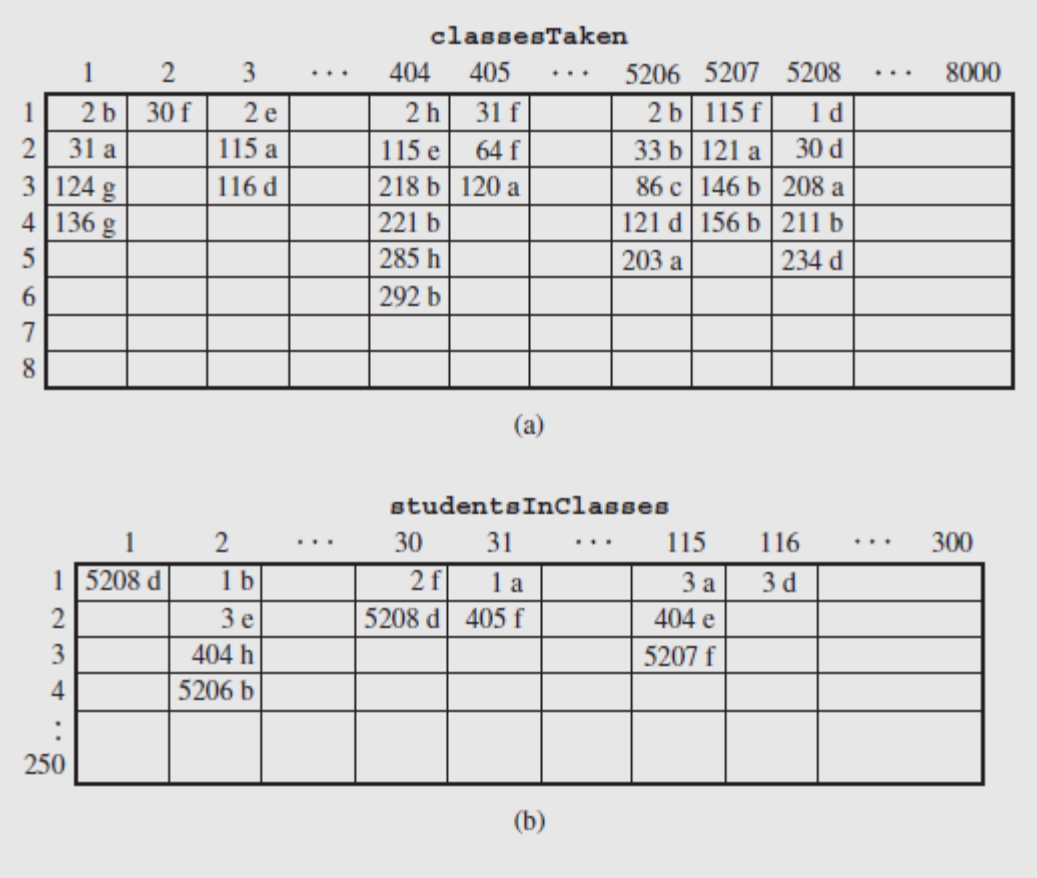
Fig. 3.21 (continued) Arrays and sparse table used for storing student grades

- As can be seen, this table has many open locations

Sparse Tables (continued)

- The table itself is 8000 (students) by 300 (classes), with one byte per grade, totaling 2.4 million bytes
- However, if every student takes only four classes each semester, there would only be four entries in each column, wasting almost 99% of the total space of the table
- An alternative approach is suggested by figure 3.22, where the data is organized into two tables
  - The first, `classesTaken`, records each class a student takes (to a maximum of eight), along with the student's grade
  - The second, `studentsInClasses`, records the students in each class (to a maximum of 250), along with their grade

Sparse Tables (continued)



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)

Fig. 3.22 Two-dimensional arrays for storing student grades

Sparse Tables (continued)

- The use of two tables speeds processing for various lists
- With these tables, and the assumption that an integer requires 2 bytes of storage, 417,000 bytes would be needed
- While considerably less than the original single table, it is still wasteful, and inflexible if conditions change
- A more useful and open-ended solution utilizes two arrays of linked lists, as shown in figure 3.23
- One array has pointers to a list of students taking a given class, the other pointers to a list of classes taken by a student
- Each node stores the student number, class number, grade, a pointer to the next class for the student, and a pointer to the next student for the class

Sparse Tables (continued)

- If a pointer occupies two bytes, each integer two bytes, and a character one byte, then each node is nine bytes in size
- For 8000 students and an average of four classes, this list occupies 288,000 bytes, which is roughly 10% of the original table and 70% of the two table variation
- Moreover, there is no wasted space, and the lists can be easily extended

Lists in the Standard Template Library

- The STL implements lists as generic doubly linked lists with head and tail pointers, as shown in figure 3.9
- The class is available in a program through the directive `#include <list>`, and a list of the methods in the list container is shown in figure 3.24 (pages 110 – 112)
- An example of their use is shown in the program in figure 3.25 (pages 112 and 113)

Concluding Remarks

- The motivation for introducing linked lists was to allow dynamic allocation of memory, using only what was needed
- Insertion and deletion of data was also easily effected
- This does not eliminate the need for arrays; random access to elements cannot be achieved with lists
- However for specific elements, or for algorithms that focus on changing the structure of the data, lists are preferable
- Arrays also have less overhead in space utilization; all that is needed is the space for data
- Lists need to allocate pointers, which can add substantial space overhead depending on the application