Lesson 8
Linked List

Linked Lists

- Linked list: a collection of items (nodes) containing two components:
  - Data
  - Address (link) of the next node in the list

FIGURE 16-1 Structure of a node

Linked Lists (cont'd.)

- Example:
  - Link field in the last node is `nullptr`

FIGURE 16-2 Linked list

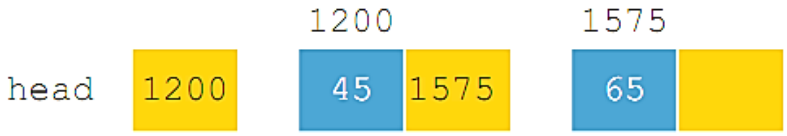


FIGURE 16-3 Linked list and values of the links

Linked Lists (cont'd.)

- A node is declared as a `class` or `struct`
  - Data type of a node depends on the specific application
  - Link component of each node is a pointer

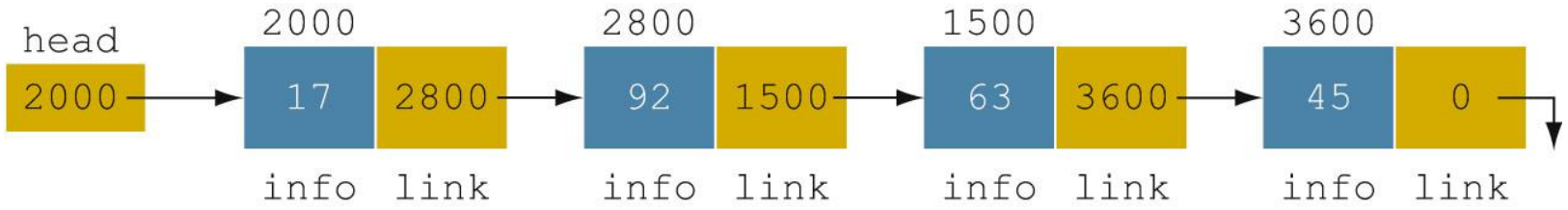
```
struct nodeType
{
    int info;
    nodeType *link;
};
```

- Variable

```
nodeType *head;
```

Linked Lists: Some Properties

- Example: linked list with four nodes (Figure 16-4)



	Value	Explanation
`head`	2000	
`head->info`	17	Because `head` is 2000 and the `info` of the node at location 2000 is 17
`head->link`	2800	
`head->link->info`	92	Because `head->link` is 2800 and the `info` of the node at location 2800 is 92

Linked Lists: Some Properties (cont'd.)

- `current = head;`
  - Copies value of `head` into `current`

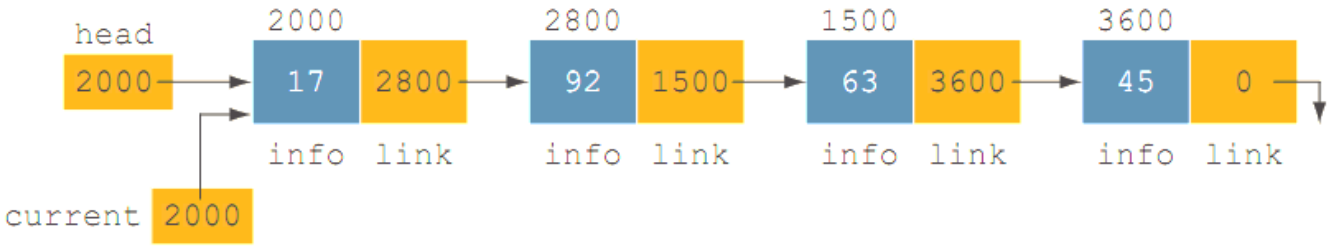


FIGURE 16-5 Linked list after the statement `current = head;` executes

	Value
`current`	2000
`current->info`	17
`current->link`	2800
`current->link->info`	92

Linked Lists: Some Properties (cont'd.)

- `current = current->link;`

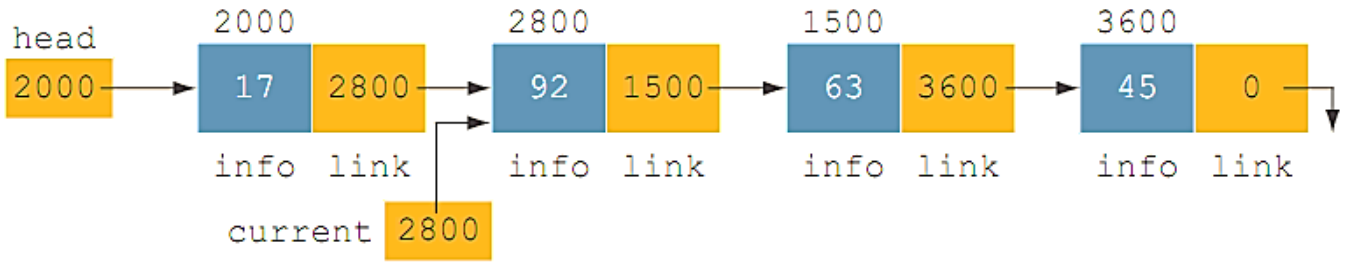


FIGURE 16-6 List after the statement `current = current->link;` executes

	Value
`current`	2800
`current->info`	92
`current->link`	1500
`current->link->info`	63

Traversing a Linked List

- Basic operations of a linked list:
  - Search for an item in the list
  - Insert an item in the list
  - Delete an item from the list
- Traversal: given a pointer to the first node of the list, step through the nodes of the list

Traversing a Linked List (cont'd.)

- To traverse a linked list:

```
current = head;

while (current != NULL)
{
    //Process the current node
    current = current->link;
}
```

- Example:

```
current = head;

while (current != NULL)
{
    cout << current->info << " ";
    current = current->link;
}
```

Item Insertion and Deletion

- Definition of a node:

```
struct nodeType
{
    int info;
    nodeType *link;
};
```

- Variable declaration:

```
nodeType *head, *p, *q, *newNode;
```

Insertion

- To insert a new node with `info 50` after `p` in this list:

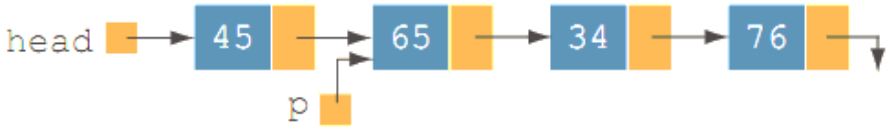
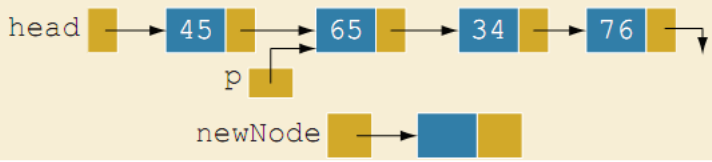
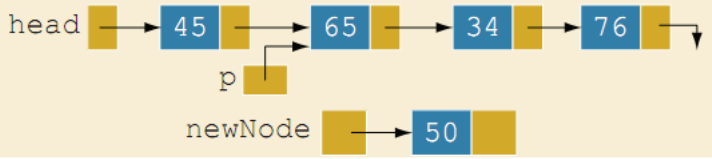
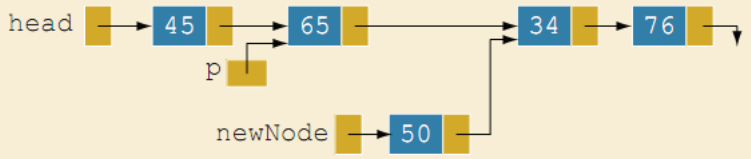
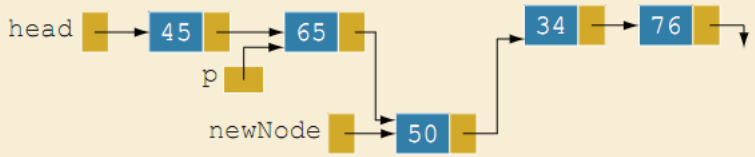


FIGURE 16-7 Linked list before item insertion

```
newNode = new nodeType;   //create newNode
newNode->info = 50;        //store 50 in the new node
newNode->link = p->link;
p->link = newNode;
```

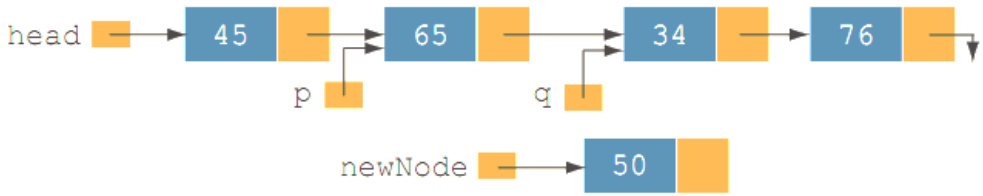
Insertion (cont'd.)

TABLE 16-1 Inserting a Node in a Linked List

Statement	Effect
`newNode = new nodeType;`	
`newNode->info = 50;`	
`newNode->link = p->link;`	
`p->link = newNode;`	

Insertion (cont'd.)

- Can use two pointers to simplify the insertion code somewhat:



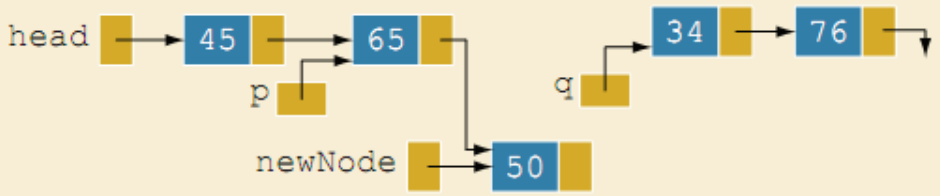
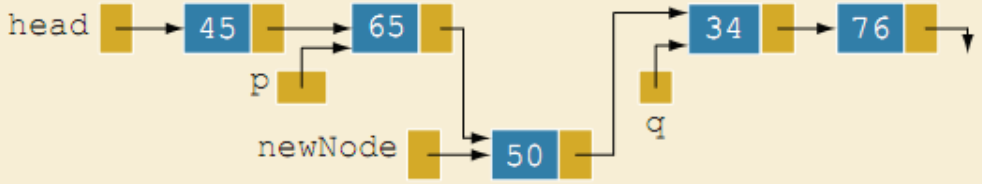
- To i

FIGURE 16-9 List with pointers p and q

```
newNode->link = q;
p->link = newNode;
```

Insertion (cont'd.)

TABLE 16-2 Inserting a Node in a Linked List Using Two Pointers

Statement	Effect
`p->link = newNode;`	
`newNode->link = q;`	

Deletion

- Node with `info 34` is to be deleted:

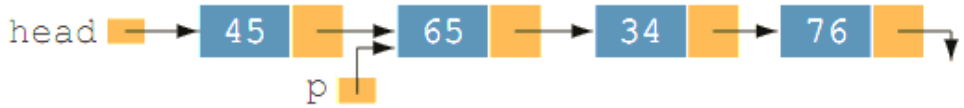


FIGURE 16-10 Node to be deleted is with `info 34`

```
p->link = p->link->link;
```

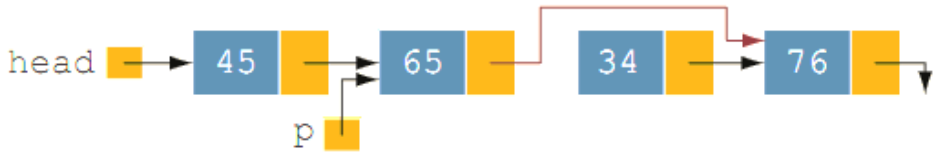


FIGURE 16-11 List after the statement `newNode->link = q;` executes

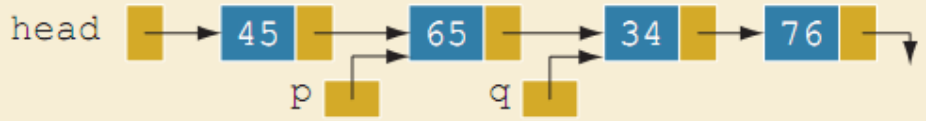
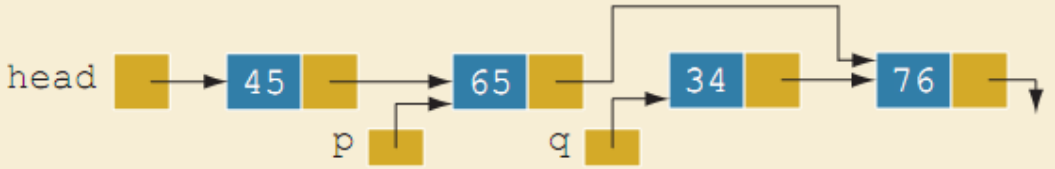
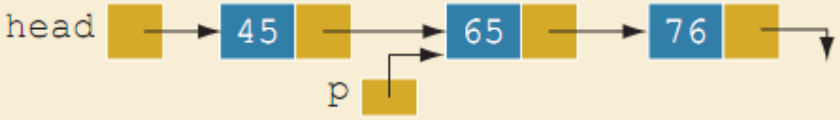
Deletion (cont'd.)

- Node with `info 34` is removed from the list, but memory is still occupied
  - Node is dangling
  - Must keep a pointer to the node to be able to deallocate its memory

```
q = p->link;
p->link = q->link;
delete q;
```

Deletion (cont'd.)

TABLE 16-3 Deleting a Node from a Linked List

Statement	Effect
`q = p->link;`	
`p->link = q->link;`	
`delete q;`	

Building a Linked List

- If data is unsorted, the list will be unsorted
- Can build a linked list forward or backward
  - Forward: a new node is always inserted at the end of the linked list
  - Backward: a new node is always inserted at the beginning of the list

Building a Linked List Forward

- Need three pointers to build the list:
  - One to point to the first node in the list, which cannot be moved
  - One to point to the last node in the list
  - One to create the new node
- Example:
  - Data: `2 15 8 24 34`

Building a Linked List Forward (cont'd.)

FIGURE 16-12 Empty list

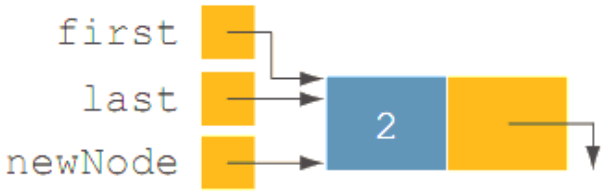


FIGURE 16-14 List after inserting `newNode` in it

Building a Linked List Forward (cont'd.)

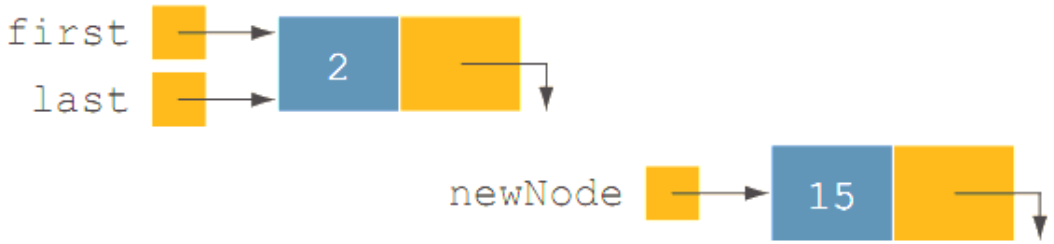


FIGURE 16-15 List and newNode with info 15

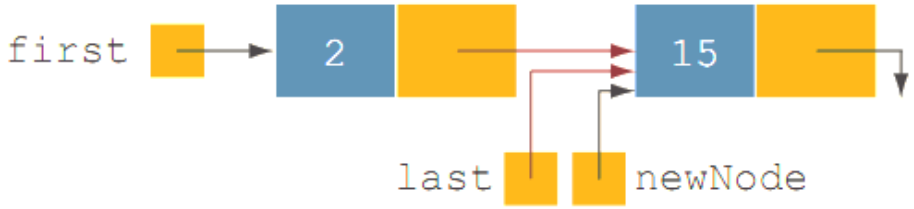


FIGURE 16-16 List after inserting newNode at the end

Building a Linked List Forward (cont'd.)

- Now repeat this process three more times:

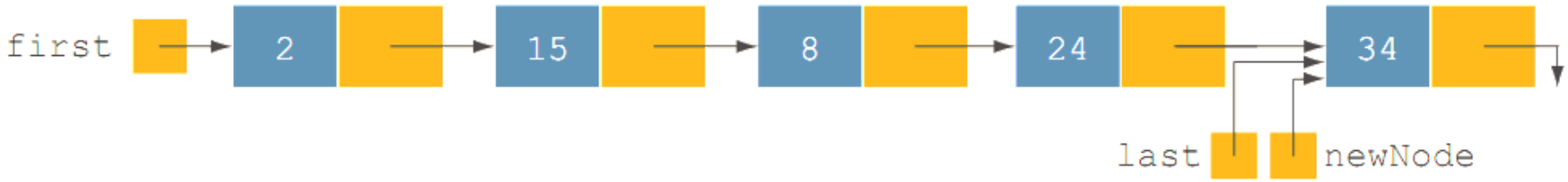


FIGURE 16-17 List after inserting 8, 24, and 34

Building a Linked List Backward

- Algorithm to build a linked list backward:
  - Initialize `first` to `nullptr`
  - For each item in the list
    - Create the new node, `newNode`
    - Store the data in `newNode`
    - Insert `newNode` before `first`
    - Update the value of the pointer `first`

Linked List as an ADT

- Basic operations on linked lists:
  - Initialize the list
  - Determine whether the list is empty
  - Print the list
  - Find the length of the list
  - Destroy the list
  - Retrieve `info` contained in the first or last node
  - Search the list for a given item

Doubly Linked Lists (cont'd.)

- Operations:
  - Initialize or destroy the list
  - Determine whether the list is empty
  - Search the list for a given item
  - Retrieve the first or last element of the list
  - Insert or delete an item
  - Find the length of the list
  - Print the list
  - Make a copy of the list

Insert a Node (cont'd.)

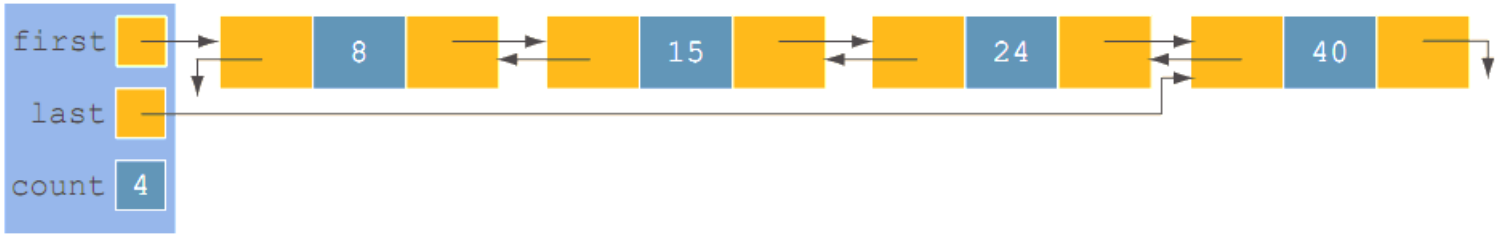


FIGURE 16-40 Doubly linked list before inserting 20

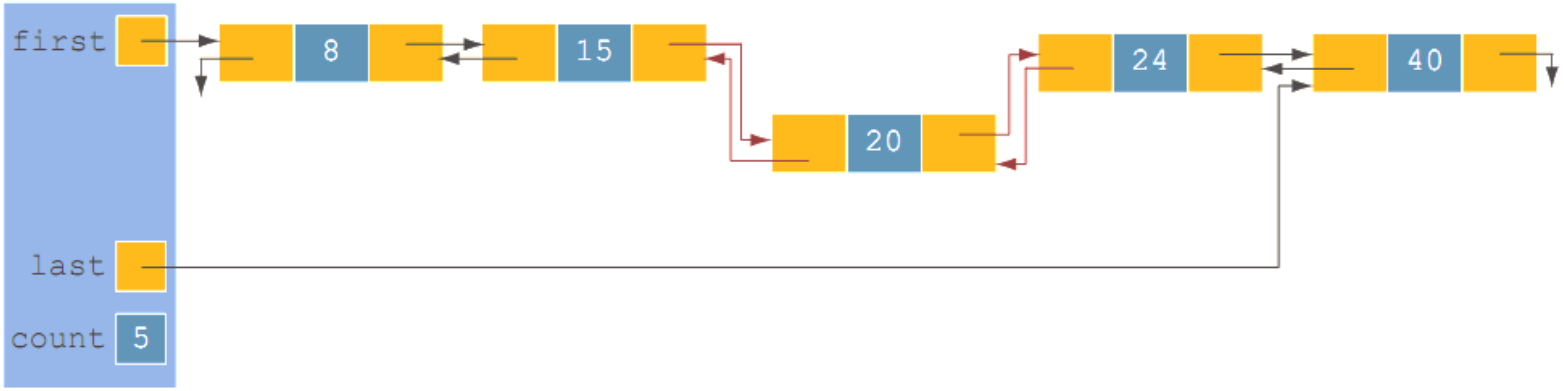


FIGURE 16-41 Doubly linked list after inserting 20

Delete a Node

- Case 1: The list is empty
- Case 2: The item to be deleted is first node in list
  - Must update the pointer `first`
- Case 3: Item to be deleted is somewhere in the list
- Case 4: Item to be deleted is not in the list
- After deleting a node, `count` is decremented by 1

Delete a Node (cont'd.)

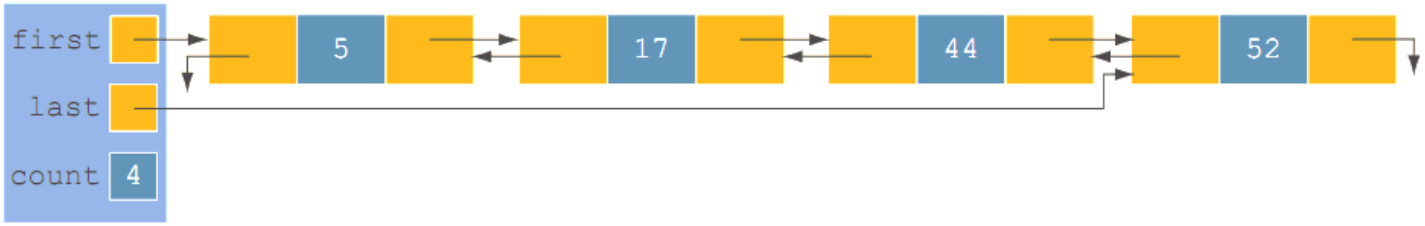


FIGURE 16-42 Doubly linked list before deleting 17

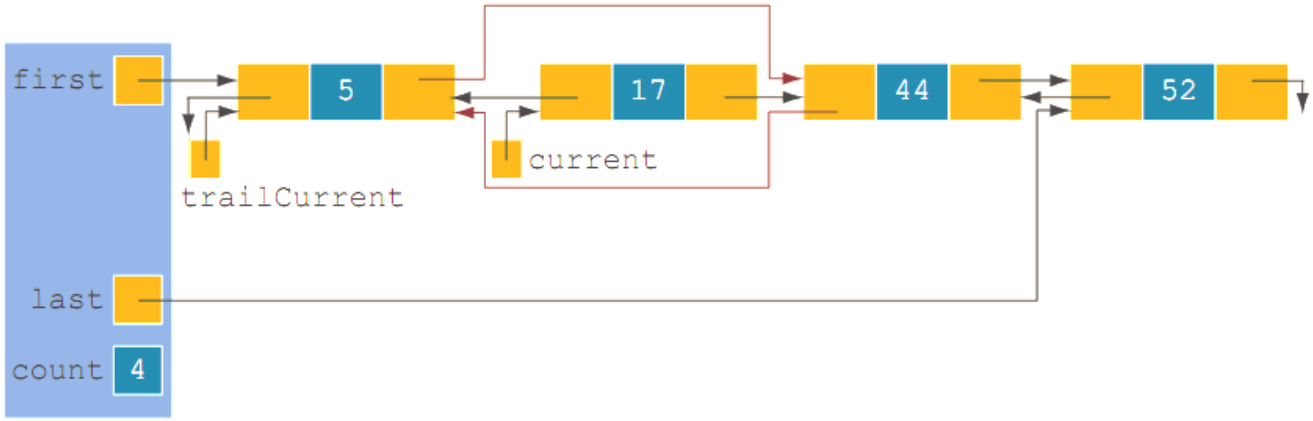


FIGURE 16-43 List after adjusting the links of the nodes before and after the node with info 17

Delete a Node (cont'd.)

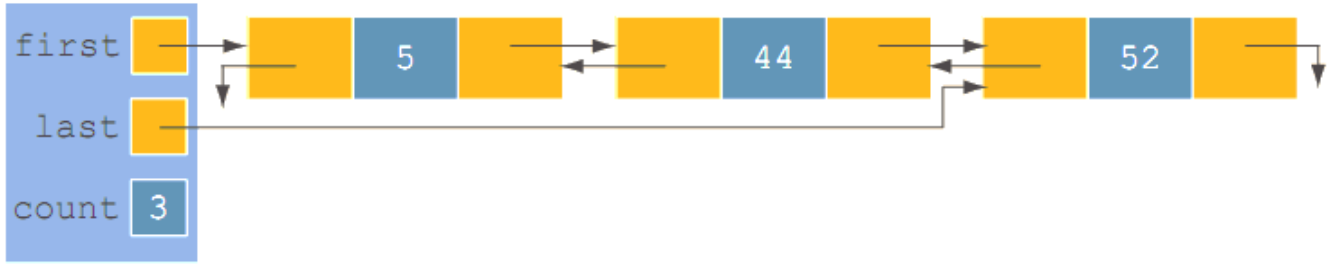


FIGURE 16-44 List after deleting the node with `info` 17

Circular Linked Lists

- Circular linked list: a linked list in which the last node points to the first node

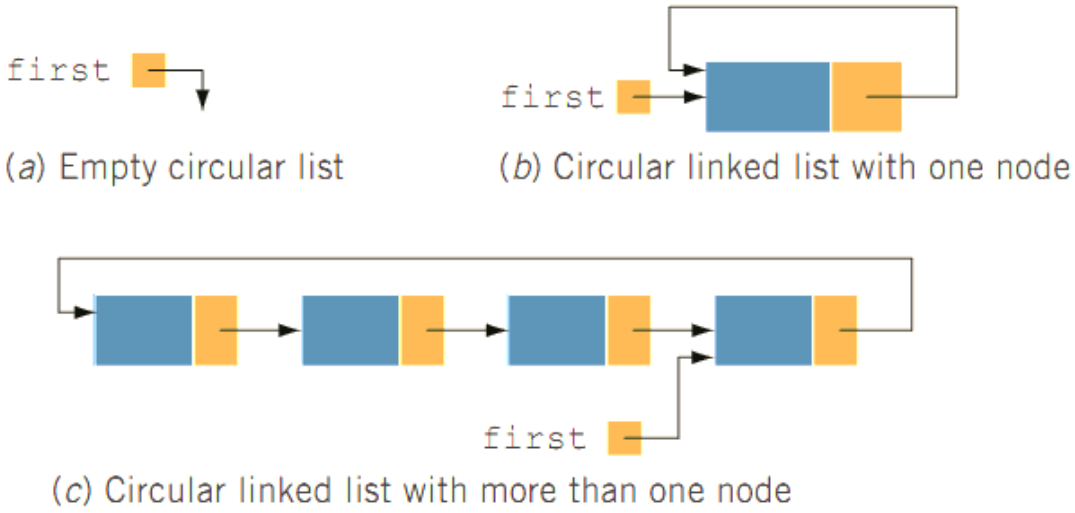


FIGURE 16-45 Circular linked lists

Circular Linked Lists (cont'd.)

- Operations on a circular list:
  - Initialize the list (to an empty state)
  - Determine if the list is empty
  - Destroy the list
  - Print the list
  - Find the length of the list
  - Search the list for a given item
  - Insert or delete an item
  - Copy the list