Chapter 16:
Linked Lists

Introduction

- Data can be organized and processed sequentially using an array, called a sequential list
- Problems with an array
  - Array size is fixed
  - <u>Unsorted array</u>: searching for an item is slow
  - <u>Sorted array</u>: insertion and deletion is slow because it requires data movement

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`

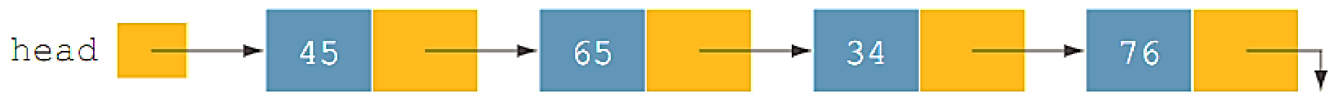


FIGURE 16-2 Linked list

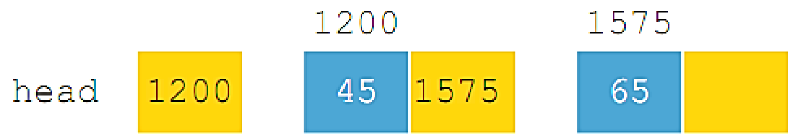


FIGURE 16-3 Linked list and values of the links

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

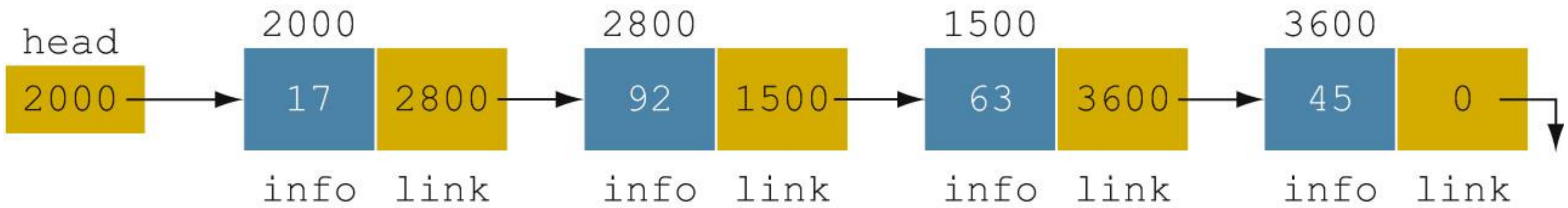
```
struct nodeType
{
    int info;
    nodeType *link;
};
```

- Variable declaration:

```
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`

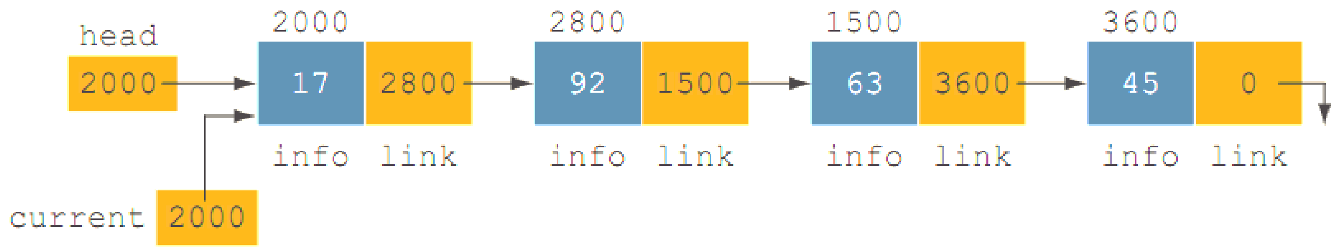


FIGURE 16-5 Linked list after the statement `current = head;` executes

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

Linked Lists: Some Properties (cont'd.)

- `current = current->link;`

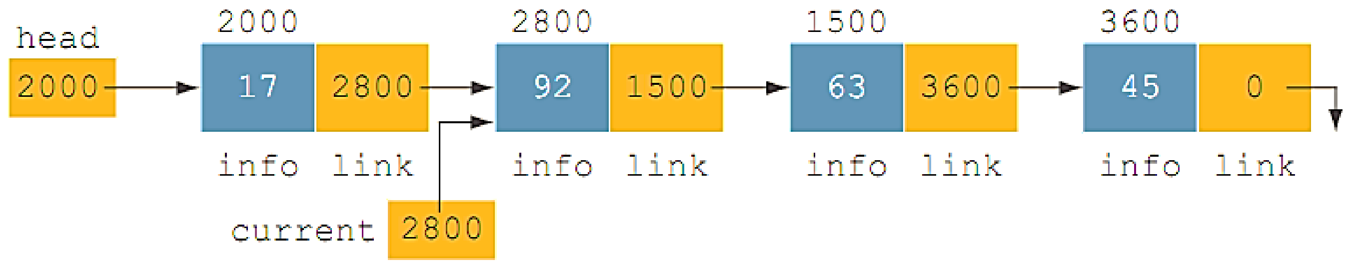


FIGURE 16-6 List after the statement `current = current->link;` executes

	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:

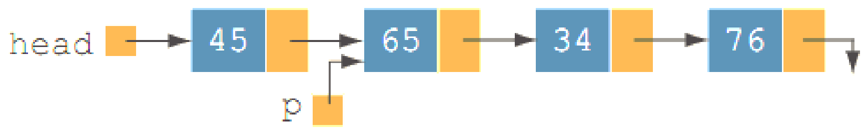
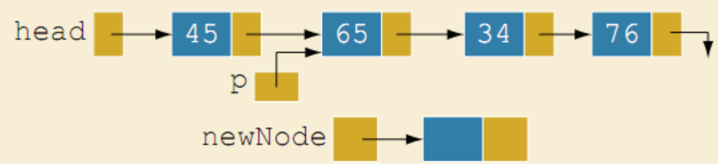
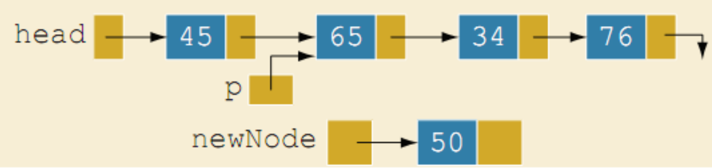
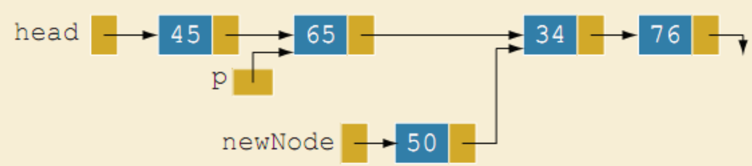
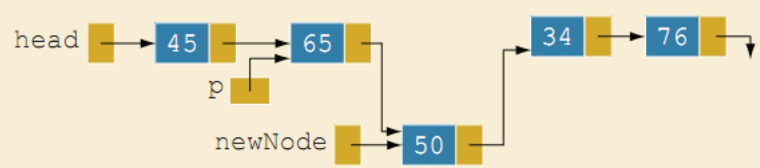


FIGURE 16-7 Linked list before item insertion

```
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:

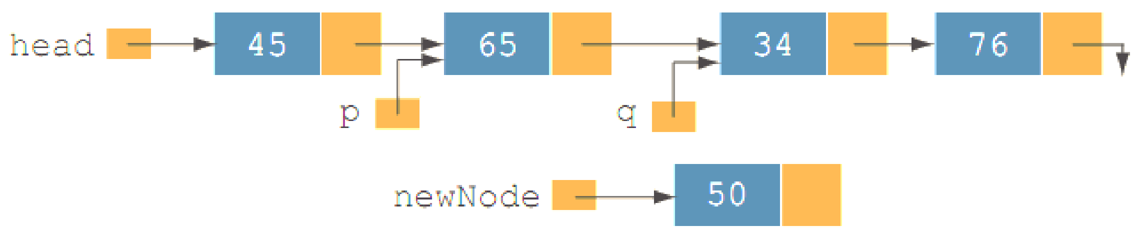


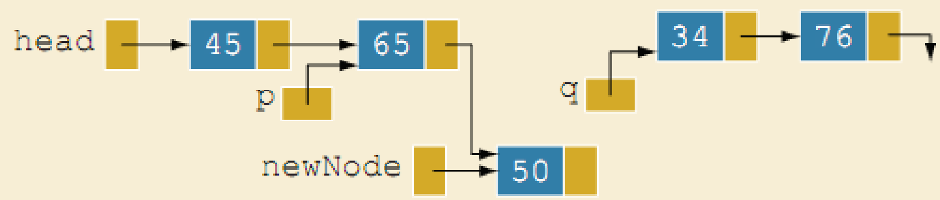
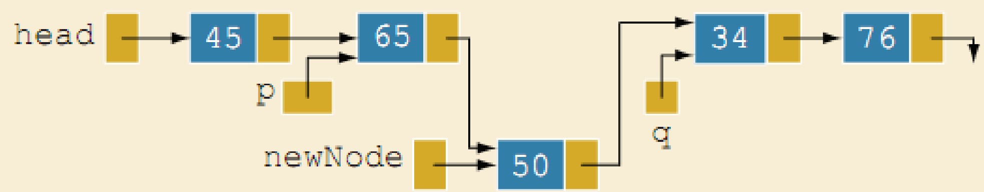
FIGURE 16-9 List with pointers p and q

- To insert `newNode` between `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:

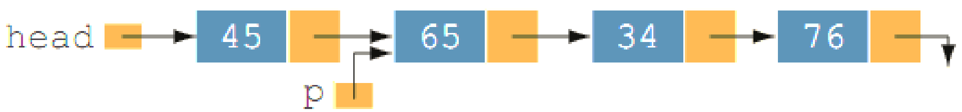


FIGURE 16-10 Node to be deleted is with `info` 34

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

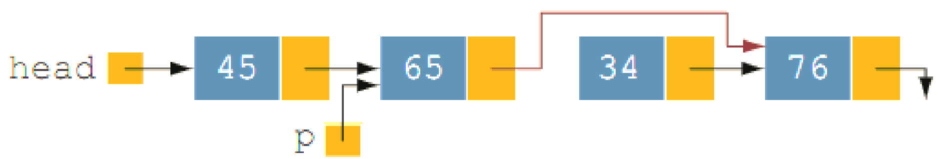


FIGURE 16-11 List after the statement `newNode->link = q;` executes

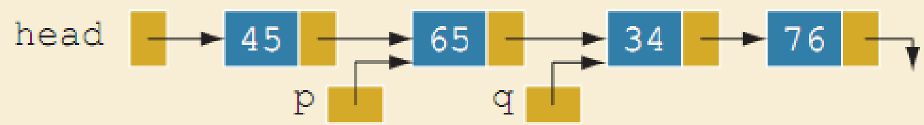
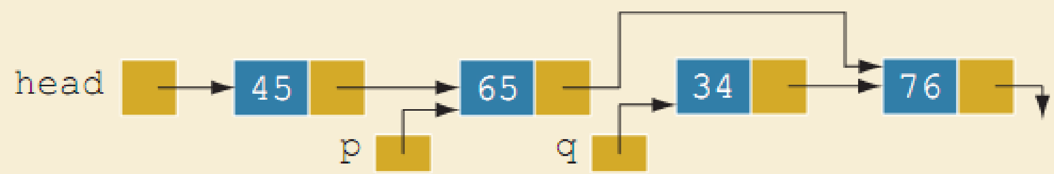
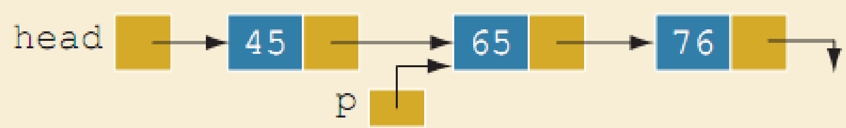
Deletion (cont'd.)

- 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.)

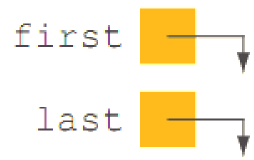


FIGURE 16-12 Empty list

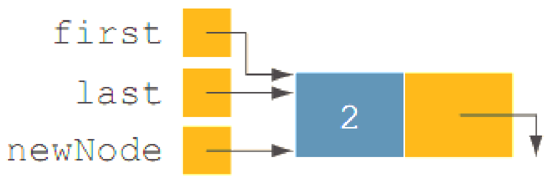


FIGURE 16-14 List after inserting newNode in it

Building a Linked List Forward (cont'd.)

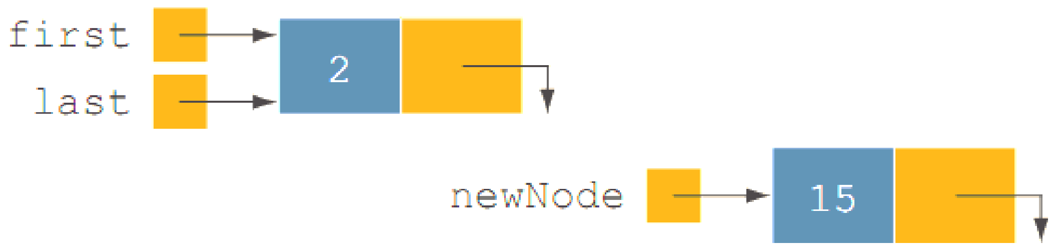


FIGURE 16-15 List and `newNode` with `info 15`

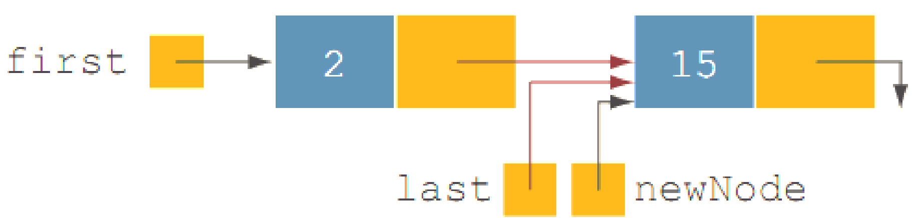


FIGURE 16-16 List after inserting `newNode` at the end

Building a Linked List Forward (cont'd.)

- Now repeat this process three more times:

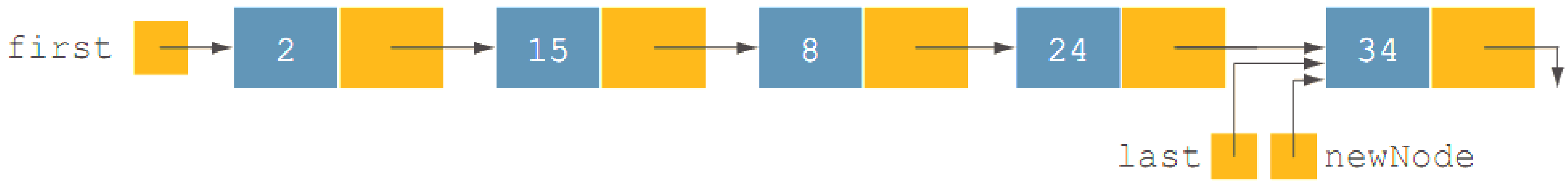


FIGURE 16-17 List after inserting 8, 24, and 34

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

Linked List as an ADT (cont'd.)

- Basic operations on linked lists (cont'd.):
  - Insert an item in the list
  - Delete an item from the list
  - Make a copy of the linked list

Structure of Linked List Nodes

- Each node has two member variables
- We implement the node of a linked list as a `struct`
- Definition of the `struct nodeType`:

```
//Definition of the node

template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type> *link;
};
```

Member Variables of the `class linkedListType`

- `linkedListType` has three member variables:
  - Two pointers: `first` and `last`
  - `count`: the number of nodes in the list

```
protected:
    int count;      //variable to store the number of
                    //elements in the list
    nodeType<Type> *first; //pointer to the first node
                           //of the list
    nodeType<Type> *last;  //pointer to the last node
                           //of the list
```

Linked List Iterators

- To process each node of the list
  - List must be traversed, starting at first node
- Iterator: object that produces each element of a container, one element at a time
  - The two most common iterator operations:

    ++ (the increment operator)

    * (the dereferencing operator)

Linked List Iterators (cont'd.)

- An iterator is an object
  - Need to define a class (`linkedListIterator`) to create iterators to objects of the `class linkedListType`
  - Will have one member variable to refer to the current node

Linked List Iterators (cont'd.)

```
linkedListType<Type>
-----------------------------------------------
#count: int
#*first: nodeType<Type>
#*last: nodeType<Type>
-----------------------------------------------
+operator=(const linkedListType<Type>&):
                    const linkedListType<Type>&
+initializeList(): void
+isEmptyList() const: bool
+print() const: void
+length() const: int
+destroyList(): void
+front() const: Type
+back() const: Type
+search(const Type&) const = 0: bool
+insertFirst(const Type&) = 0: void
+insertLast(const Type&) = 0: void
+deleteNode(const Type&) = 0: void
+begin(): linkedListIterator<Type>
+end(): linkedListIterator<Type>
+linkedListType()
+linkedListType(const linkedListType<Type>&)
+~linkedListType()
-copyList(const linkedListType<Type>&): void
```

FIGURE 16-20 UML class diagram of the **class** `linkedListType`

Default Constructor

- Default constructor:
  - Initializes the list to an empty state

```
template <class Type>
linkedListType<Type>::linkedListType() //default constructor
{
    first = NULL;
    last = NULL;
    count = 0;
}
```

Print the List

- Function `print`:
  - Prints data contained in each node
  - Traverses the list using another pointer

Length of a List

- Function `length`:
  - Returns the count of nodes in the list
  - Uses the `count` variable

Retrieve the Data of the First or Last Node

- Function `front`:
  - Returns the info contained in the first node
  - If list is empty, program will be terminated
- Function `back`:
  - Returns the info contained in the last node
  - If list is empty, program will be terminated

Begin and End

- Function `begin`:
  - Returns an iterator to the first node in the list
- Function `end`:
  - Returns an iterator to one past the last node in the list

Copy the List

- Function `copyList`:
  - Makes an identical copy of a linked list
- Steps:
  - Create a node called `newNode`
  - Copy the `info` of the original node into `newNode`
  - Insert `newNode` at the end of the list being created

Destructor & Copy Constructor

- Destructor:
  - Deallocates memory occupied by nodes when the class object goes out of scope
  - Calls `destroyList` to traverse the list and delete each node
- Copy constructor:
  - Makes an identical copy of the linked list
  - Calls function `copyList`

Search the List

- Function `search`:
  - Searches the list for a given item
- Steps:
  - Compare search item with current node in the list
    - If `info` of current node is the same as search item, stop the search
    - Otherwise, make the next node the current node
  - Repeat Step 1 until item is found or until no more data is left in the list

Insert the First Node

- Function `insertFirst`:
  - Inserts a new item at the beginning of the list
- Steps:
  - Create a new node
  - Store the new item in the new node
  - Insert the node before `first`
  - Increment `count` by 1

Insert the Last Node

- Function `insertLast`:
  - Inserts a new node after `last`
  - Similar to `insertFirst` function

Delete a Node

- Function `deleteNode`:
  - Deletes a node with given `info` from the list
  - Several possible cases to manage
- Case 1: List is empty
  - If the list is empty, output an error message
- Case 2: Node to be deleted is the first node
  - Adjust the pointer `first` and `count`
  - If no other nodes, set `first` and `last` to `nullptr`

Delete a Node (cont'd.)

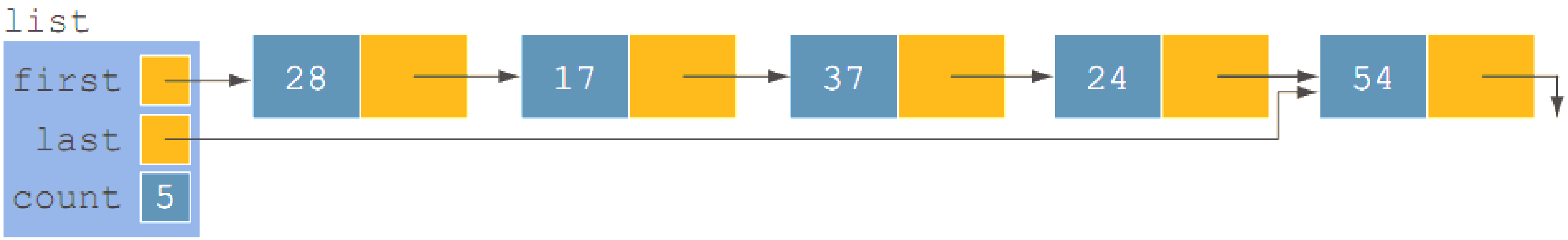


FIGURE 16-23 list with more than one node

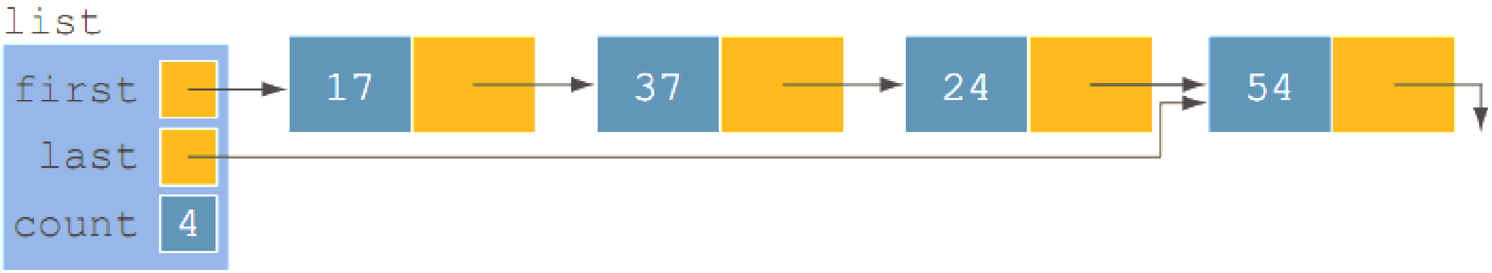


FIGURE 16-24 list after deleting node with info 28

Delete a Node (cont'd.)

- Case 3: Node to be deleted is not the first one
  - Case 3a: Node to be deleted is not last one
    - Update link field of the previous node
  - Case 3b: Node to be deleted is the last node
    - Update link field of the previous node to `nullptr`
    - Update `last` pointer to point to previous node

Delete a Node (cont'd.)

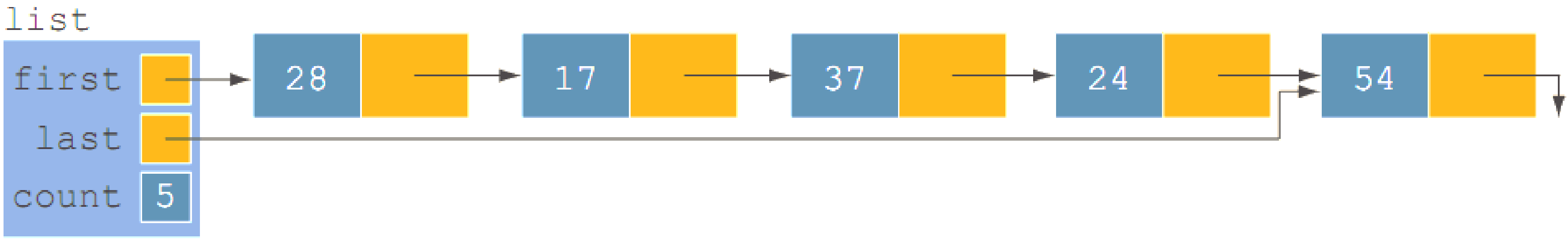


FIGURE 16-25 list before deleting 37

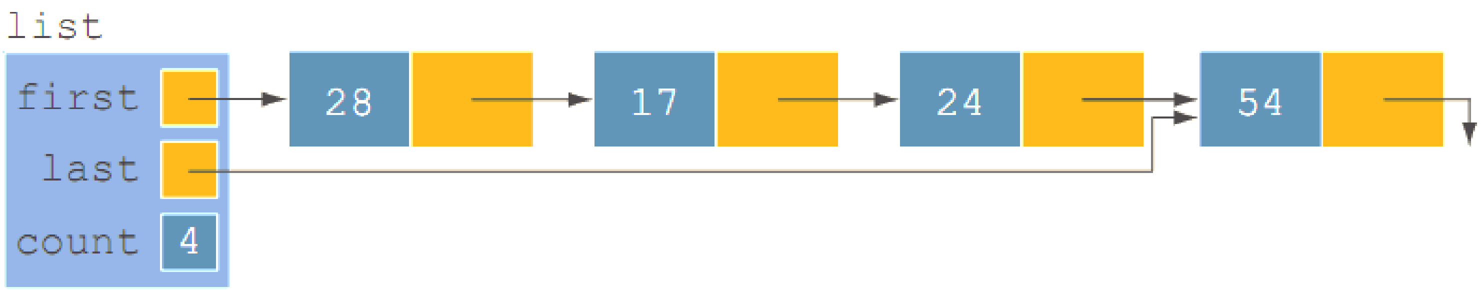


FIGURE 16-26 list after deleting 37

Delete a Node (cont'd.)

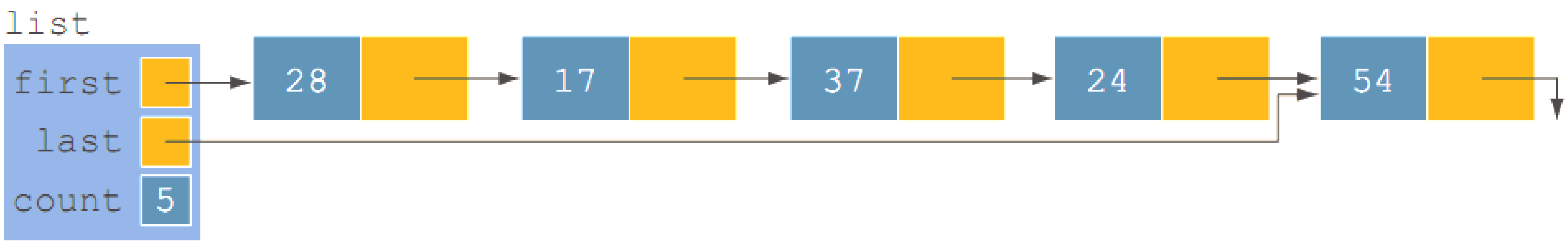


FIGURE 16-27 list before deleting 54

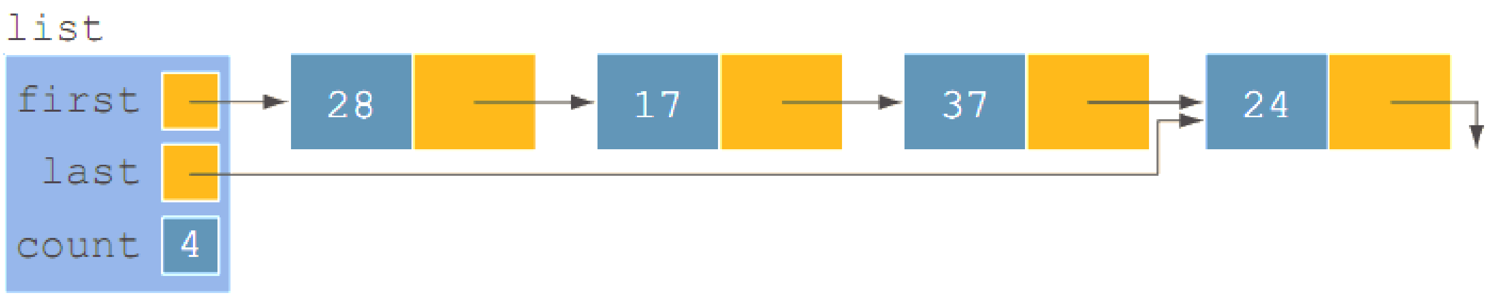


FIGURE 16-28 list after deleting 54

Search the List

- Steps:
  - Compare the search item with the current node in the list
    - If `info` of current node is >= to search item, stop search
    - Otherwise, make the next node the current node
  - Repeat Step 1 until an item in the list >= to search item is found, or no more data is left in the list

Insert a Node (cont'd.)

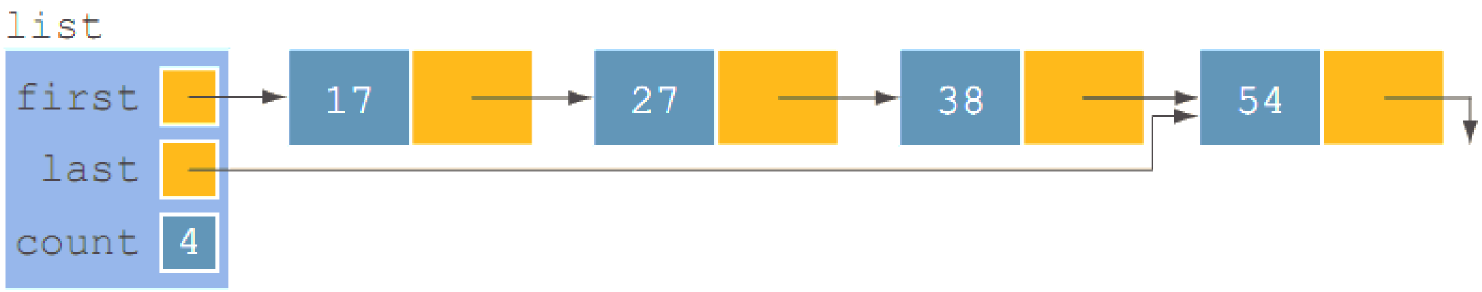


FIGURE 16-33 list before inserting 65

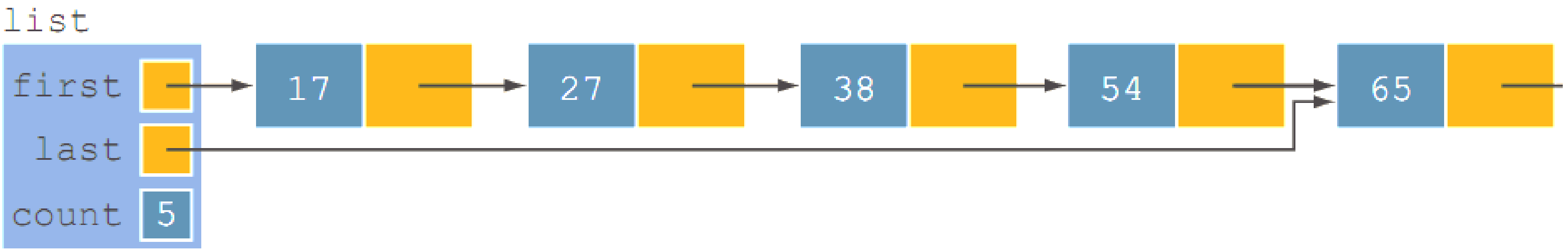


FIGURE 16-34 list after inserting 65

Insert a Node (cont'd.)

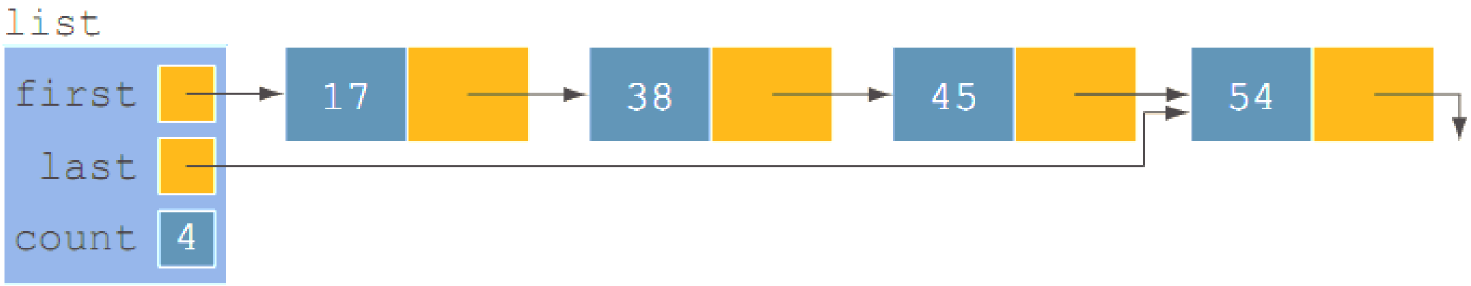


FIGURE 16-35 list before inserting 27

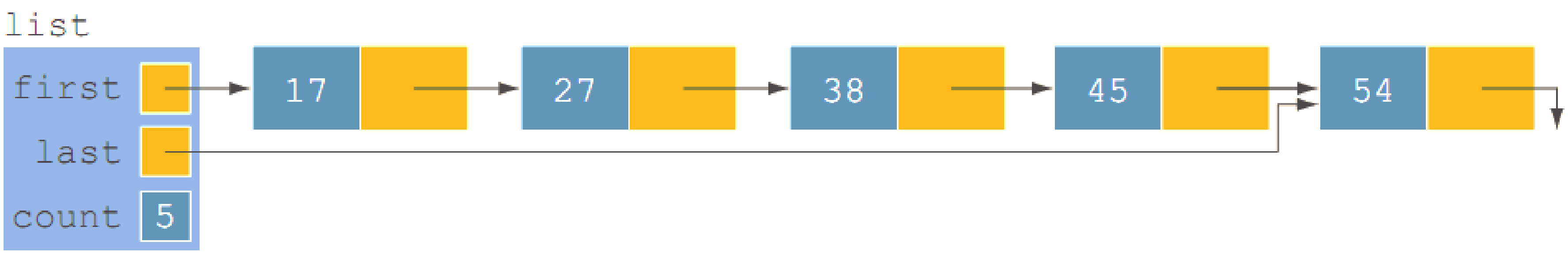


FIGURE 16-36 list after inserting 27

Doubly Linked Lists

- Doubly linked list: every node has next and back pointers
  - Can be traversed in either direction

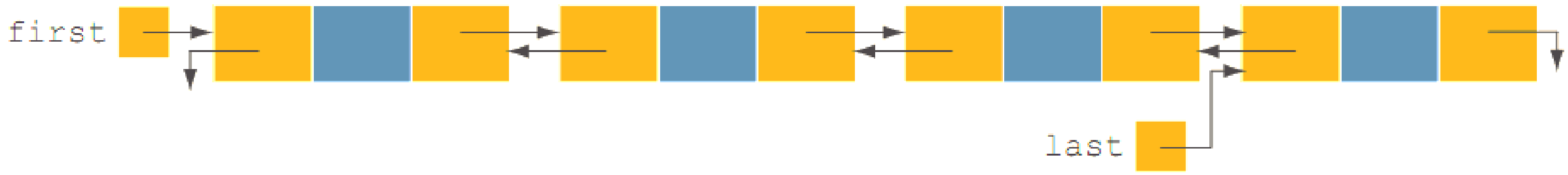


FIGURE 16-39 Doubly linked list

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

Search the List

- Function `search`:
  - Returns true if search item is found, otherwise false
  - Algorithm is same as that for an ordered linked list

First and Last Elements

- Function `front`
  - Returns first element of the list
- Function `back`
  - Returns last element of the list
- If list is empty, both functions will terminate the program

Insert a Node

- Four insertion cases:
  - Case 1: Insertion in an empty list
  - Case 2: Insertion at beginning of a nonempty list
  - Case 3: Insertion at end of a nonempty list
  - Case 4: Insertion somewhere in nonempty list
- Cases 1 & 2 require update to pointer `first`
- Cases 3 & 4 are similar:
  - After inserting item, increment `count` by 1

Insert a Node (cont'd.)

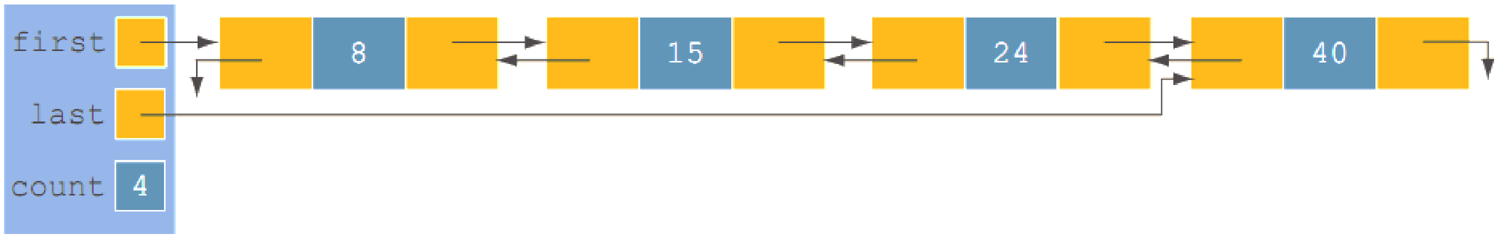


FIGURE 16-40 Doubly linked list before inserting 20

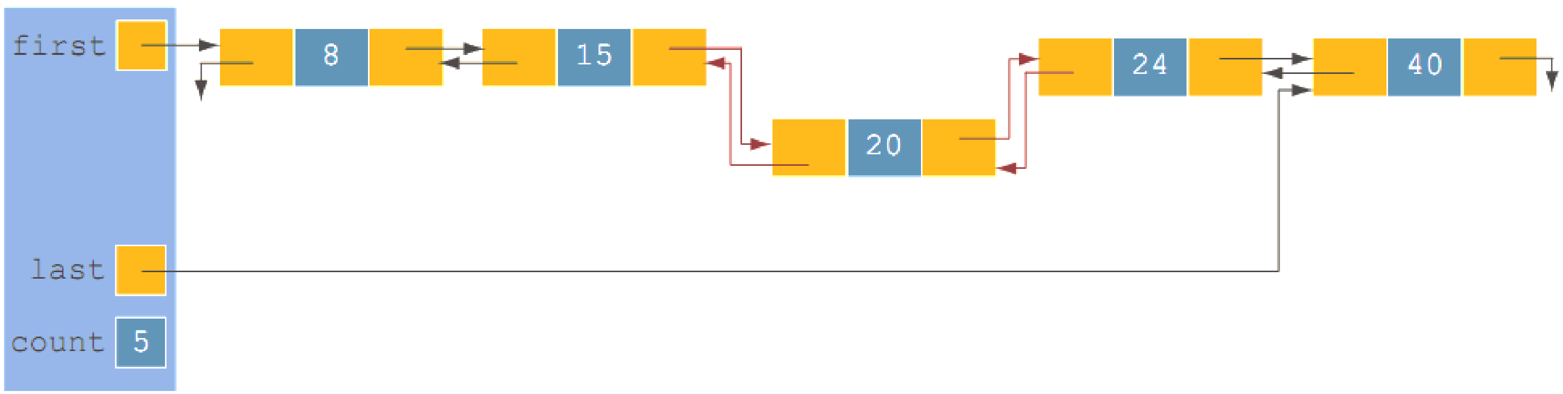


FIGURE 16-41 Doubly linked list after inserting 20

Delete a Node

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

Delete a Node (cont'd.)

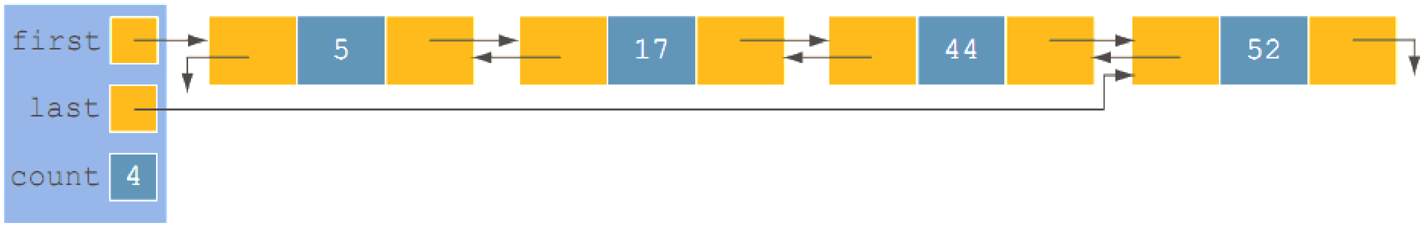


FIGURE 16-42 Doubly linked list before deleting 17

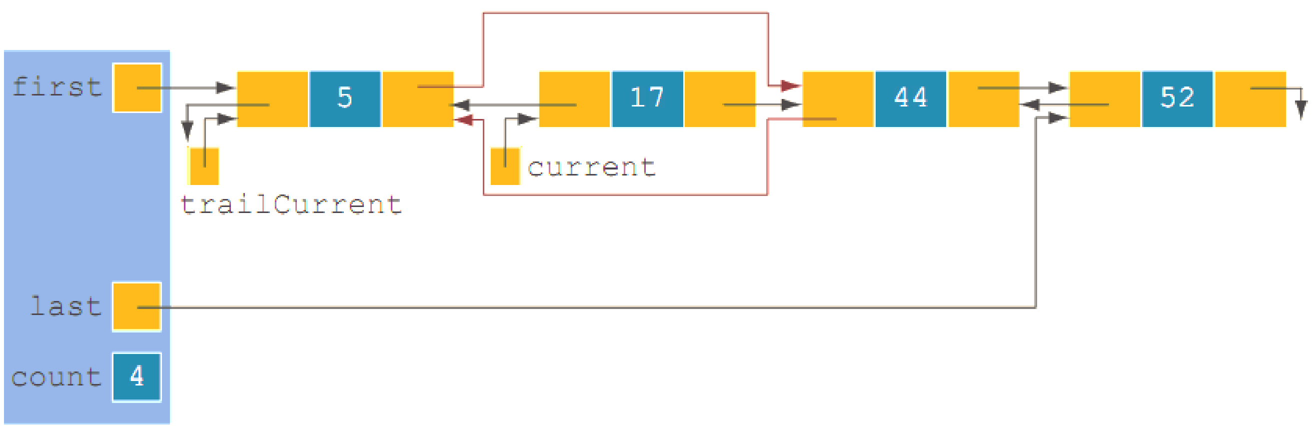


FIGURE 16-43 List after adjusting the links of the nodes before and after the node with `info` 17

Delete a Node (cont'd.)

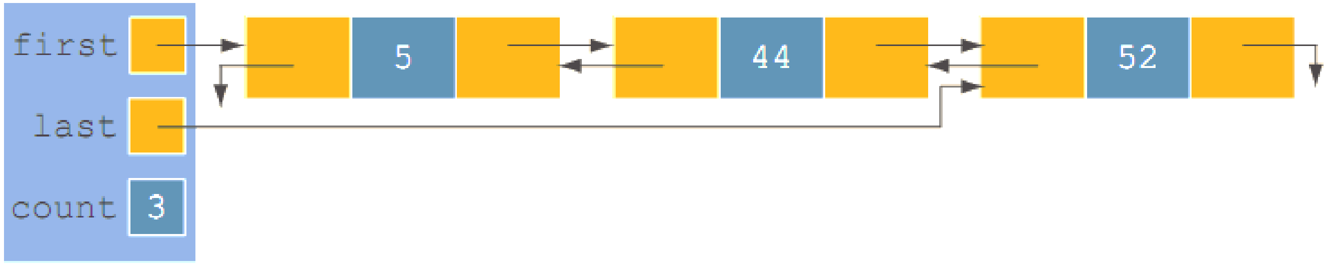


FIGURE 16-44 List after deleting the node with info 17

Circular Linked Lists

- <u>Circular linked list</u>: a linked list in which the last node points to the first node

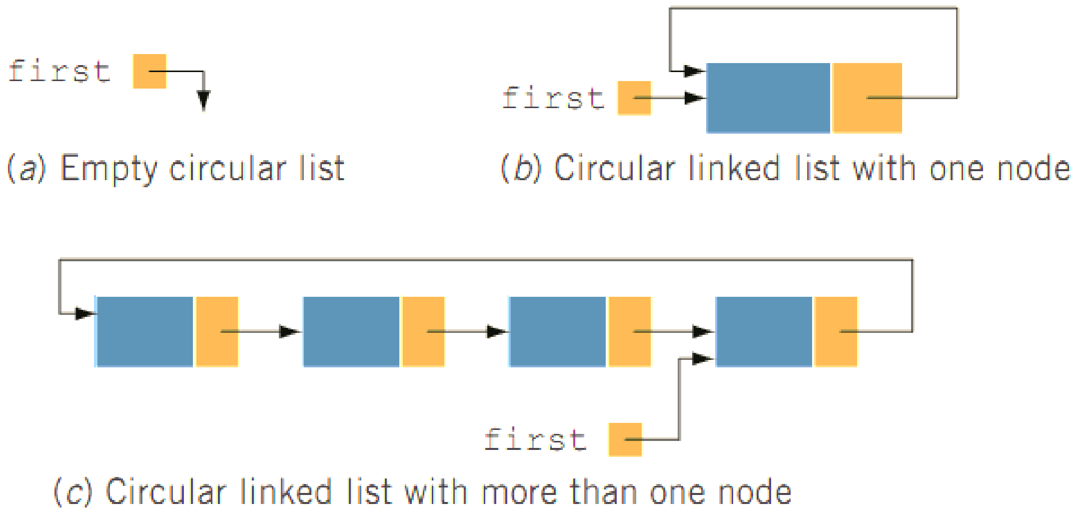


FIGURE 16-45 Circular linked lists

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