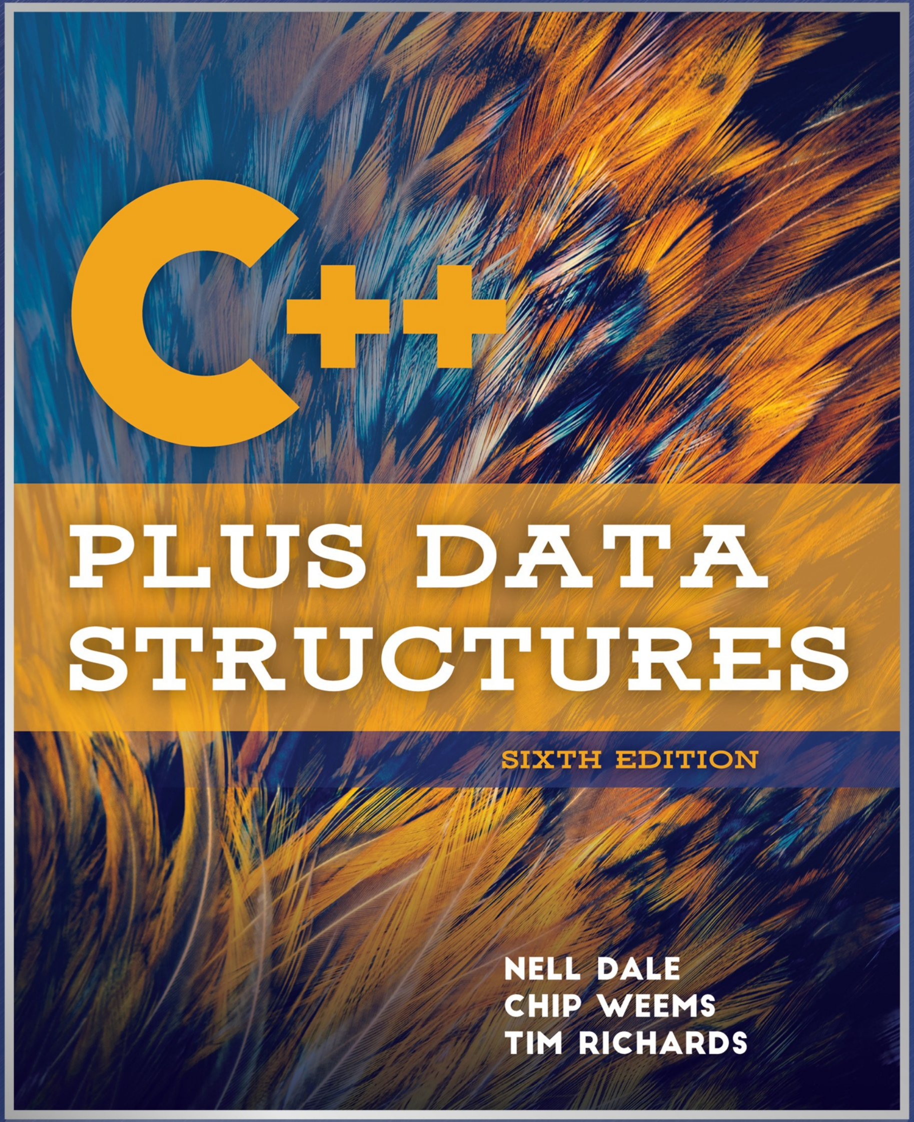
Chapter 6
Lists Plus



C++ Templates

- **Generic Data Type:** A type for which only the operators are defined; the ItemType used by our ADTs is generic
- **Template:** A C++ feature by which the compiler generates multiple versions of a class by using parameterized types
- Essentially allows “blanks” in the class definition that clients can fill in to customize the class
- **`template<class ItemType>`**

Example: Stack ADT Template

```
template<class ItemType>
class StackType
{
public:
    StackType();
    bool IsEmpty() const;
    bool IsFull() const;
    void Push(ItemType item);
    void Pop();
    ItemType Top() const;
private:
    int top;
    ItemType items[MAX_ITEMS];
};
```

Example: Stack ADT Template (cont.)

Clients use the template like so:

```
StackType<int> intStack;
StackType<float> floatStack;
StackType<char> charStack;
```

The compiler will generate a specialized version of StackType for each variable.

Function Templates

- Functions are turned into templates using the template keyword, just like classes
- A template class's methods must also be function templates if they use the type variable in the template
- For example, Push has an ItemType parameter and must be made into a function template

Example: Push Template

```
template<class ItemType>
void StackType<ItemType>::Push(ItemType
newItem)
{
    if (IsFull())
        throw FullStack();
    top++;
    items<top> = newItem;
}
```

Source Files and Templates

- Usually, the class header (StackType.h) and member function definitions (StackType.cpp) are two separate files
- This way, the class’s object code can be compiled independently of the client code
- But with templates, the compiler must know the actual type parameter for the template, which appears in the client code

Circular Linked Lists

- A linked list in which every node has a successor
- The “last” element is succeeded by the “first” element
- The class definition doesn’t change, but traversing the list is a little more complex

Circular Linked List (cont.)

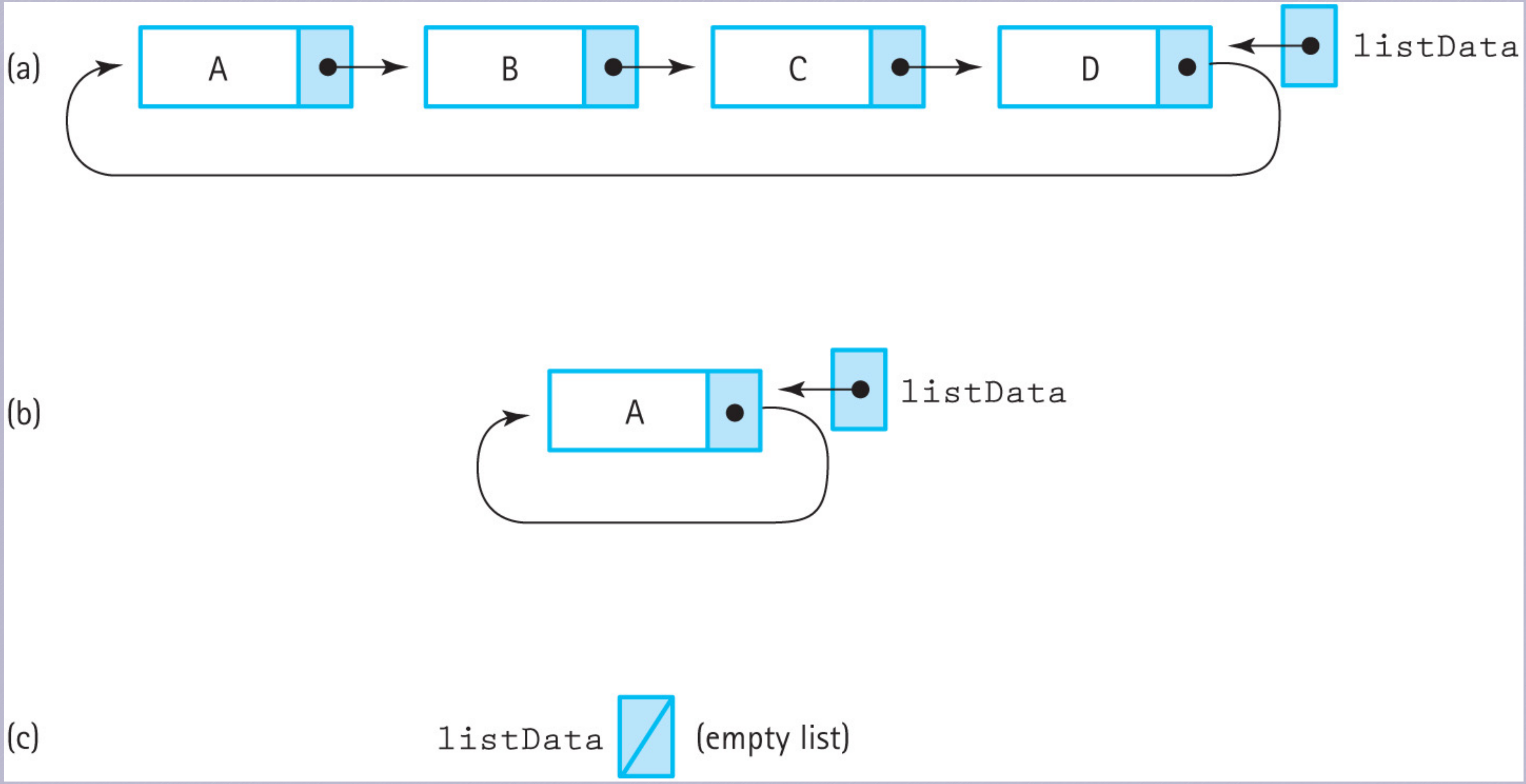
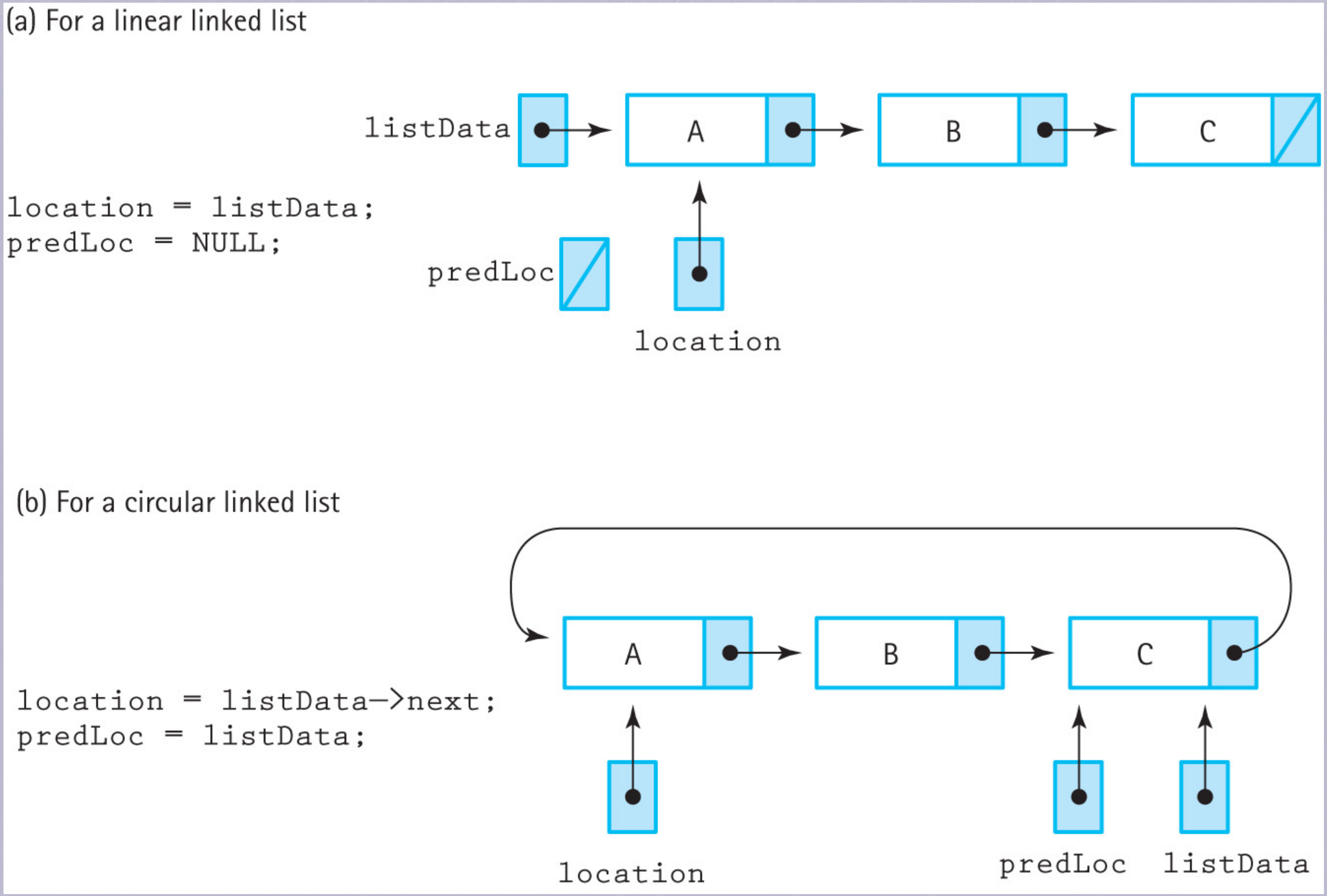


Figure 6.2 Circular linked lists with the external pointer pointing to the rear element

Circular Linked List: Finding Items

- GetItem, PutItem, and DeleteItem all search the list, create a helper function called **FindItem**
- Search stops when:
  - A key greater than or equal to the target item’s key is found
  - It encounters the first item in the list again (this is the “end” of the list)
- Returns location, previous location, and a flag
  - If the flag is true, location points to the found item; if false, it points to the item’s successor

Circular Linked List: Finding Items



(a) For a linear linked list

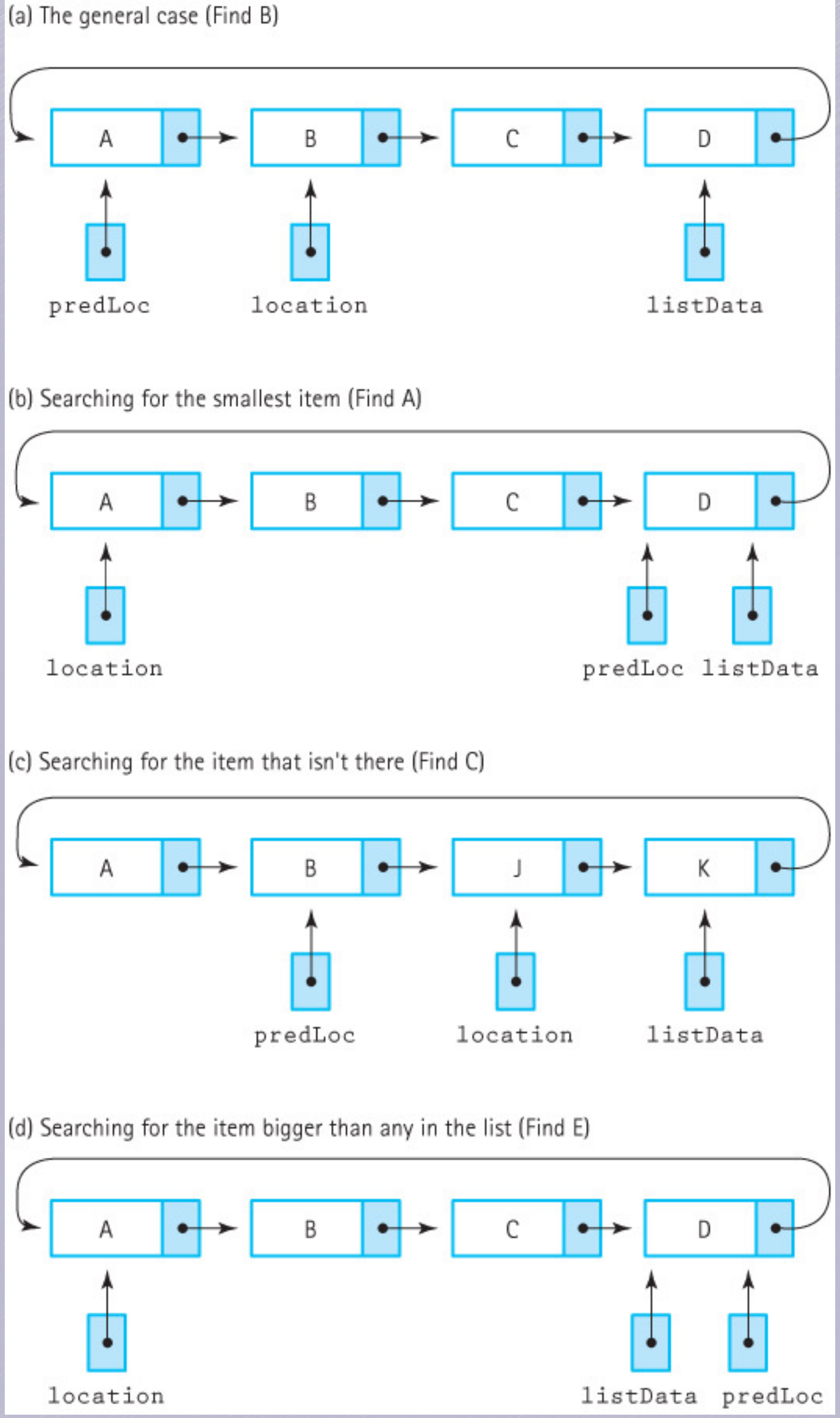
```
location = listData;
predLoc = NULL;
```

(b) For a circular linked list

```
location = listData->next;
predLoc = listData;
```

Figure 6.3 Circular linked lists with the external pointer pointing to the rear element

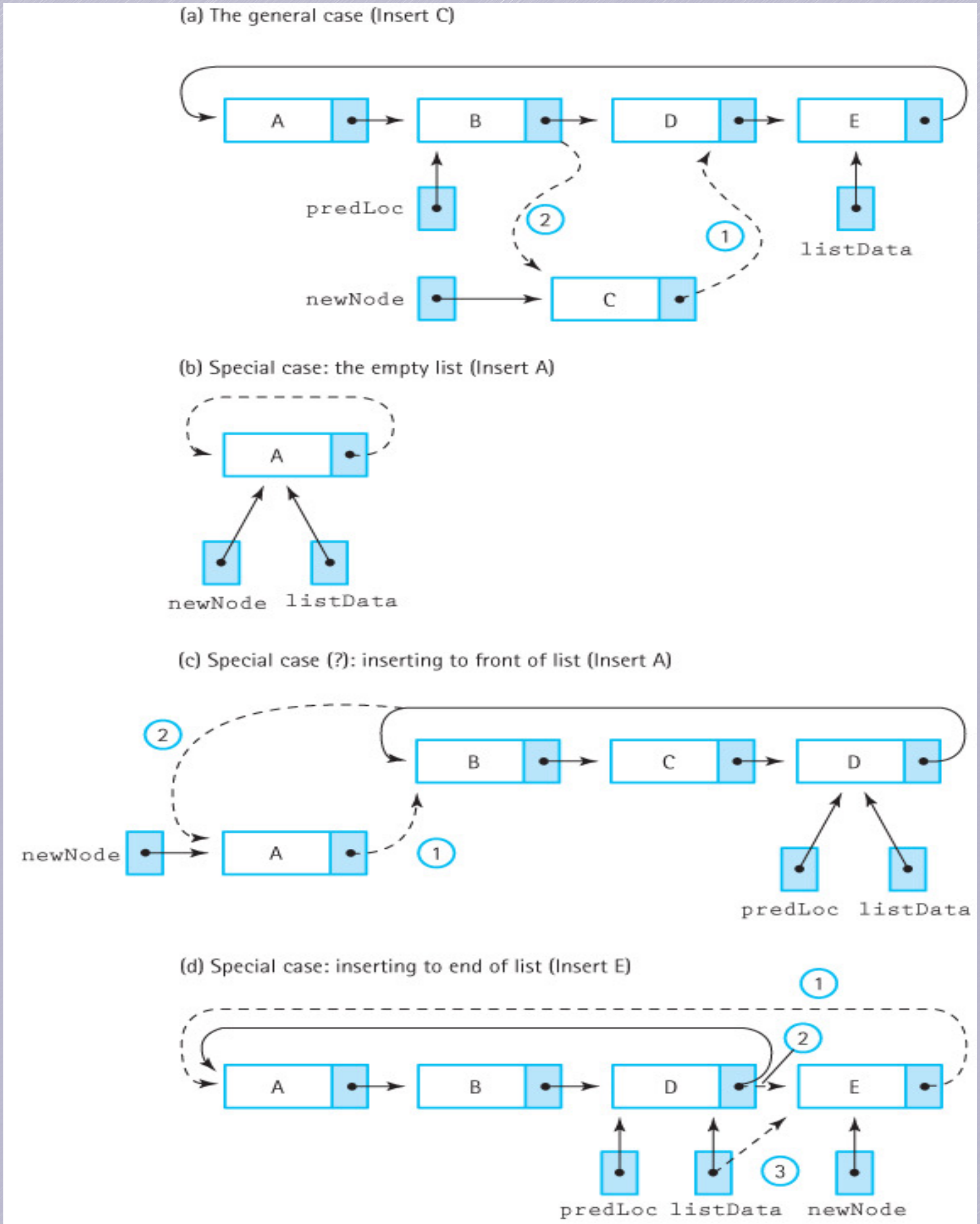
Figure 6.4 The FindItem operation for a circular list (a) The general case (Find B); (b) Searching for the smallest item (Find A); (c) Searching for the item that isn't there (Find C); (d) Searching for the item bigger than any in the list (Find E)


(a) The general case (Find B)
A
B
C
D
predLoc
location
listData
(b) Searching for the smallest item (Find A)
A
B
C
D
location
predLoc
listData
(c) Searching for the item that isn't there (Find C)
A
B
J
K
predLoc
location
listData
(d) Searching for the item bigger than any in the list (Find E)
A
B
C
D
location
listData
predLoc

Circular Linked List: Inserting Items

- General case: Link predecessor to new node and new node to successor
- Inserting into an empty list: The new node points to itself
- Inserting into the front of a list: Only special in regular linked lists
- Inserting at the end of a list: Update the external pointer

Figure 6.5 Inserting into a circular linked list (a) The general case (Insert C); (b) Special case: the empty list (Insert A); (c) Special case (?): inserting to front of list (Insert A); (d) Special case: inserting to end of list (Insert E)



(a) The general case (Insert C)
A
B
D
E
predLoc
listData
2
1
newNode
C
(b) Special case: the empty list (Insert A)
A
newNode
listData
(c) Special case (?): inserting to front of list (Insert A)
2
B
C
D
newNode
A
1
predLoc
listData
(d) Special case: inserting to end of list (Insert E)
1
2
A
B
D
E
3
predLoc
listData
newNode

Circular Linked List: Deleting Items

- General case: Update predecessor to point to deleted item’s successor, then delete item
- Deleting the only item in list: Set external pointer to NULL
- Deleting item at the end of a list: Update external pointer to point to deleted node’s predecessor

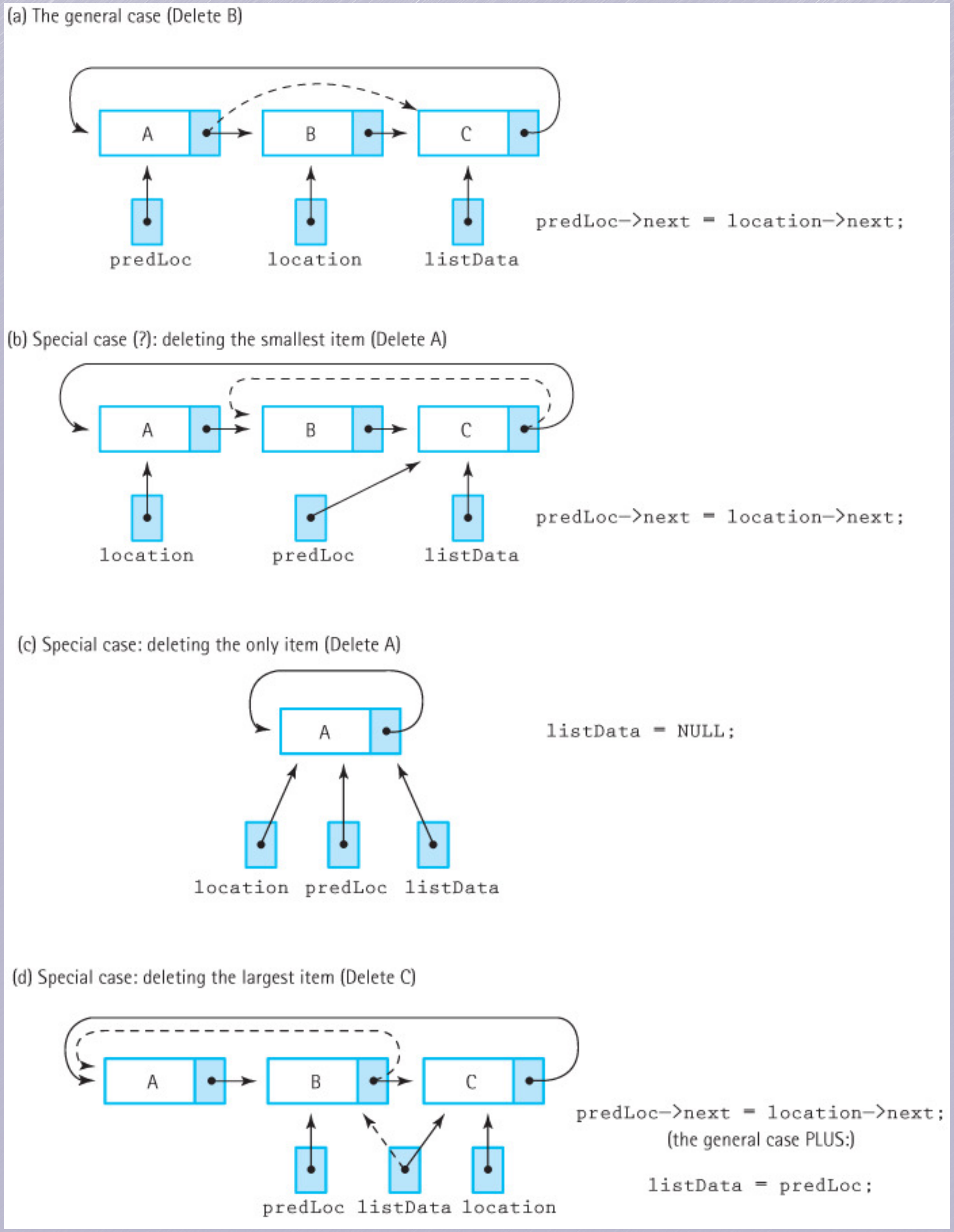

(a) The general case (Delete B)
A
B
C
predLoc
location
listData
predLoc->next = location->next;
(b) Special case (?): deleting the smallest item (Delete A)
A
B
C
location
predLoc
listData
predLoc->next = location->next;
(c) Special case: deleting the only item (Delete A)
A
listData = NULL;
location
predLoc
listData
(d) Special case: deleting the largest item (Delete C)
A
B
C
predLoc->next = location->next;
(the general case PLUS:)
listData = predLoc;
predLoc
listData
location

Figure 6.6 Deleting from a circular linked list (a) The general case (Delete B) (b) Special case (?): deleting the smallest item (Delete A) (c) Special case: deleting the only item (Delete A) (d) Special case: deleting the largest

Doubly Linked List

- A linked list in which every node has 2 pointers, linking it to its successor and predecessor
  - First node has NULL predecessor pointer
  - Last node has NULL successor pointer
- Can walk forward or backward through the list

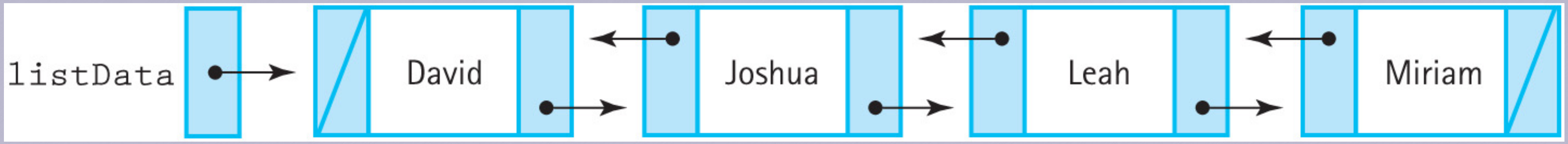


Figure 6.7 A linear doubly linked list

Doubly Linked List: Finding Items

- “Inchworm” search no longer needed, since the previous element can be accessed directly
- FindItem only needs to return the pointer to the item or the item’s successor

Doubly Linked List Operations

- Insertion and deletion are slightly more complex due to the additional pointers
- Both the predecessor and the successor of the target node must have their pointers updated
- Operating on items at either end of the list is similar to singly linked lists

Doubly Linked List Operations (cont.)

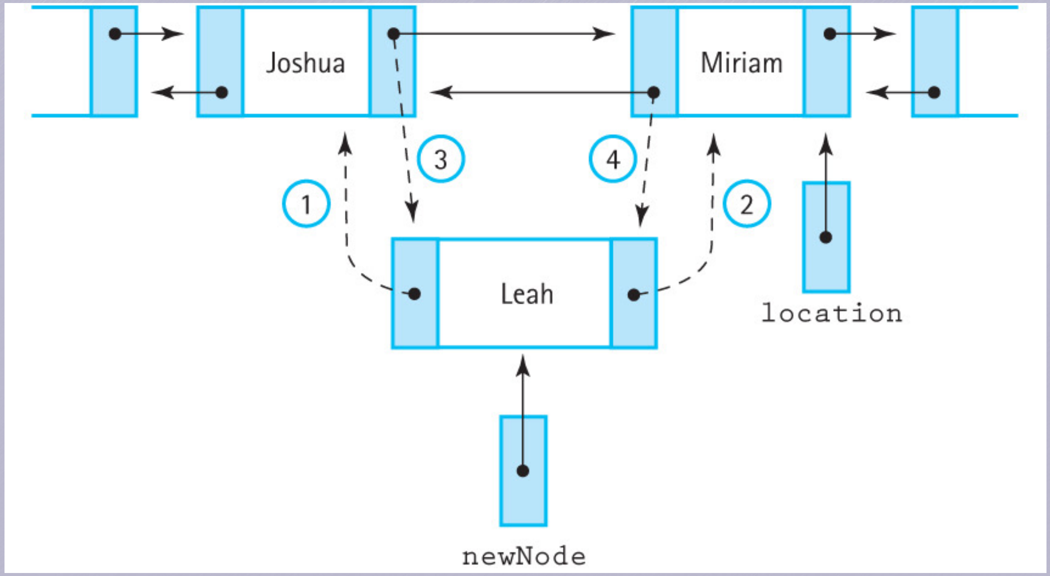


Figure 6.9 Linking the new node into the list

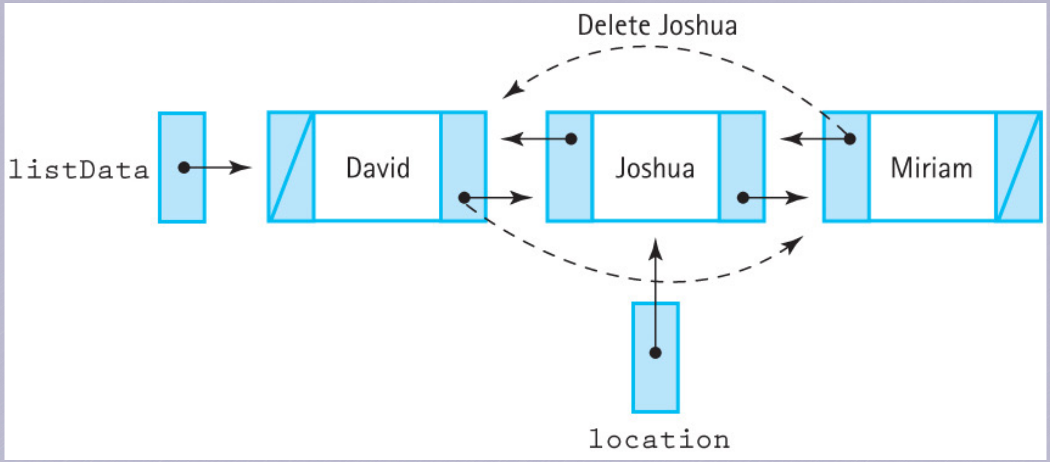


Figure 6.10 Deleting from a doubly linked list