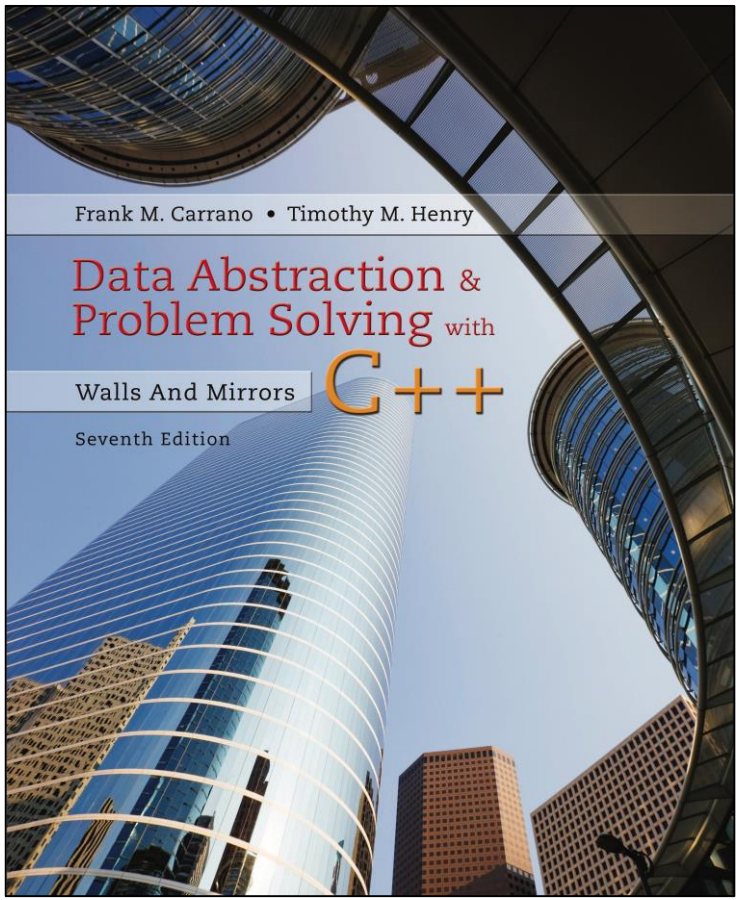
Data Abstraction & Problem Solving with C++: Walls and Mirrors

Seventh Edition



Chapter 8

Lists

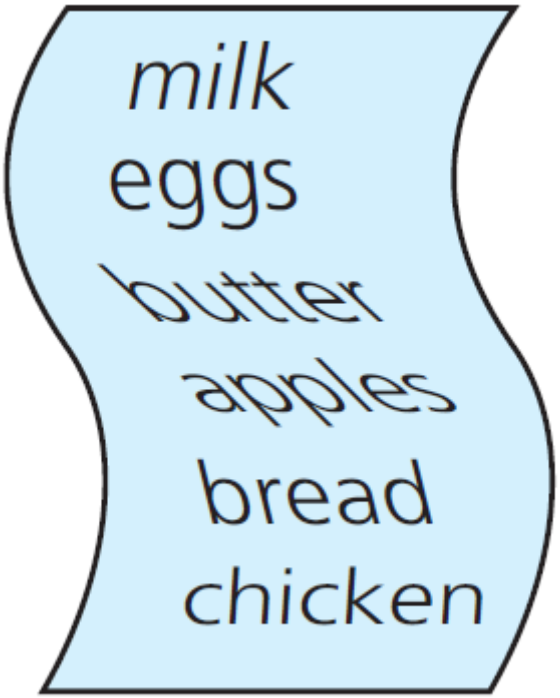
Linked list nodes are random spaces in memory
Not like array blocks

Specifying the ADT List (1 of 4)

- Things you make lists of
  - Chores
  - Addresses
  - Groceries
- Lists contain items of the same type
- Operations
  - Count items
  - Add, remove items
  - Retrieve

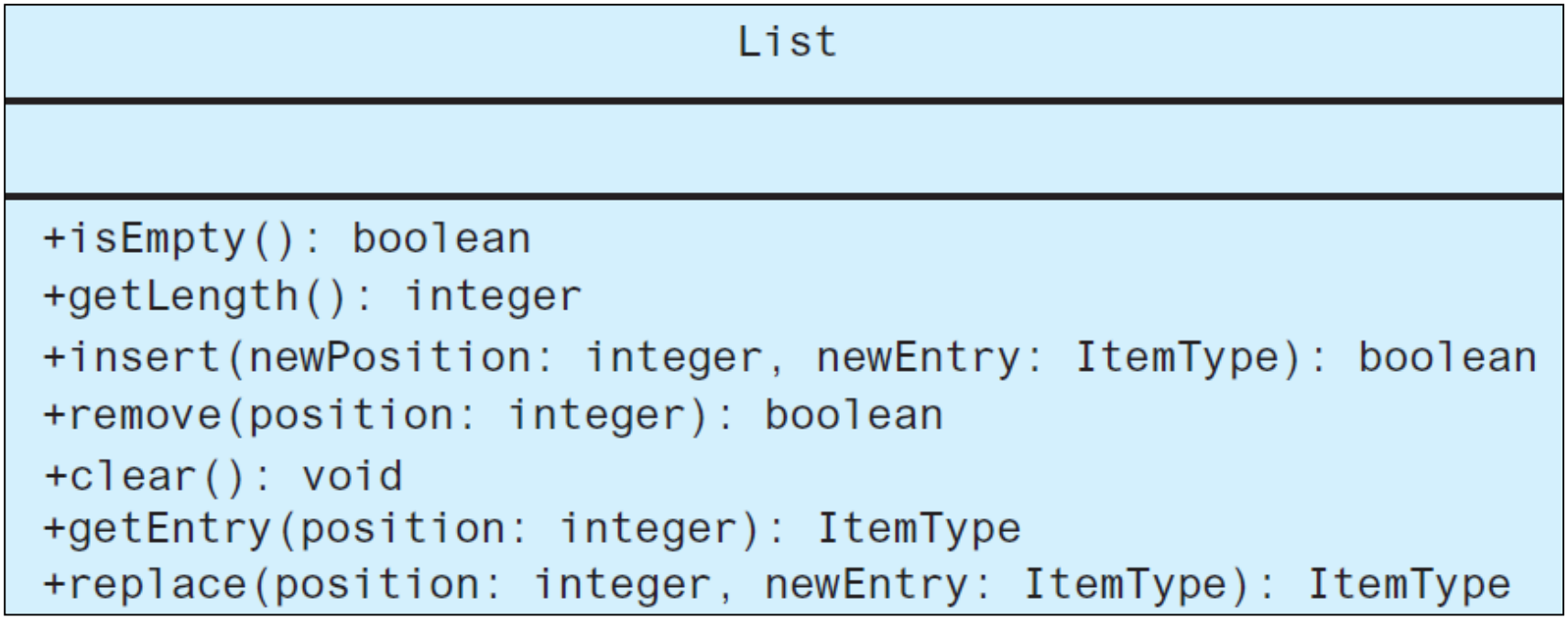
Specifying the ADT List (2 of 4)

Figure 8-1 A grocery list



Specifying the ADT List (3 of 4)

Figure 8-2 UML diagram for the ADT list



Specifying the ADT List (4 of 4)

- Definition: ADT List
  - Finite number of objects
  - Not necessarily distinct
  - Same data type
  - Ordered by position as determined by client
  - Except for the first and last items, each item has a unique predecessor and a unique successor

Axioms for multiplication

$(a \times b) \times c = a \times (b \times c)$
$a \times b = b \times a$
$a \times 1 = a$
$a \times 0 = 0$

Axioms for ADT List

1. (List()).isEmpty() = true
2. (List()).getLength() = 0
3. aList.getLength() = (aList.insert(i, item)).getLength() - 1
4. aList.getLength() = (aList.remove(i)).getLength() + 1
5. (aList.insert(i, item)).isEmpty() = false
6. (List()).remove(i) = false
7. (aList.insert(i, item)).remove(i) = true
8. (aList.insert(i, item)).remove(i) = aList
9. (List()).getEntry(i) => error
10. (aList.insert(i, item)).getEntry(i) = item
11. aList.getEntry(i) = (aList.insert(i, item)).getEntry(i + 1)
12. aList.getEntry(i + 1) = (aList.remove(i)).getEntry(i)
13. (List()).replace(i, item) => error
14. (aList.replace(i, item)).getEntry(i) = item

Using the List Operations (1 of 2)

Displaying the items on a list.

```
// Displays the items on the list aList.
displayList(aList)
{
   for (position = 1 through aList.getLength())
   {
      dataItem = aList.getEntry(position)
      Display dataItem
   }
}
```

Using the List Operations (2 of 2)

Replacing an item.

```
// Replaces the ith entry in the list aList with newEntry.
// Returns true if the replacement was successful; otherwise return false.
replace(aList, i, newEntry)
{
   success = aList.remove(i)
   if (success)
      success = aList.insert(i, newEntry)

   return success
}
```

Interface Template for ADT List (1 of 4)

Listing 8-1 A C++ interface for lists

```
1  /** Interface for the ADT list
2   @file ListInterface.h */
3
4  #ifndef LIST_INTERFACE_
5  #define LIST_INTERFACE_
6
7  template<class ItemType>
8  class ListInterface
9
10 {
11 public:
12    /** Sees whether this list is empty.
13     @return  True if the list is empty; otherwise returns false. */
14    virtual bool isEmpty() const = 0;
15
16    /** Gets the current number of entries in this list.
17     @return  The integer number of entries currently in the list. */
18    virtual int getLength() const = 0;
```

Interface Template for ADT List (2 of 4)

Listing 8-1 [Continued]

```
19
20      /** Inserts an entry into this list at a given position.
21       @pre  None.
22       @post  If 1 <= position <= getLength() + 1 and the insertion is
23          successful, newEntry is at the given position in the list,
24          other entries are renumbered accordingly, and the returned
25          value is true.
26       @param newPosition  The list position at which to insert newEntry.
27       @param newEntry  The entry to insert into the list.
28       @return  True if the insertion is successful, or false if not. */
29      virtual bool insert(int newPosition, const ItemType& newEntry) = 0;
30
31      /** Removes the entry at a given position from this list.
32       @pre  None.
33       @post  If 1 <= position <= getLength() and the removal is successful,
34          the entry at the given position in the list is removed, other
35          items are renumbered accordingly, and the returned value is true.
36       @param position  The list position of the entry to remove.
37       @return  True if the removal is successful, or false if not. */
38      virtual bool remove(int position) = 0;
39
```

Interface Template for ADT List (3 of 4)

Listing 8-1 [Continued]

```
39
40      /** Removes all entries from this list.
41       @post  The list contains no entries and the count of items is 0. */
42      virtual void clear() = 0;
43
44      /** Gets the entry at the given position in this list.
45       @pre  1 <= position <= getLength().
46       @post  The desired entry has been returned.
47       @param position  The list position of the desired entry.
48       @return  The entry at the given position. */
49      virtual ItemType getEntry(int position) const = 0;
50
```

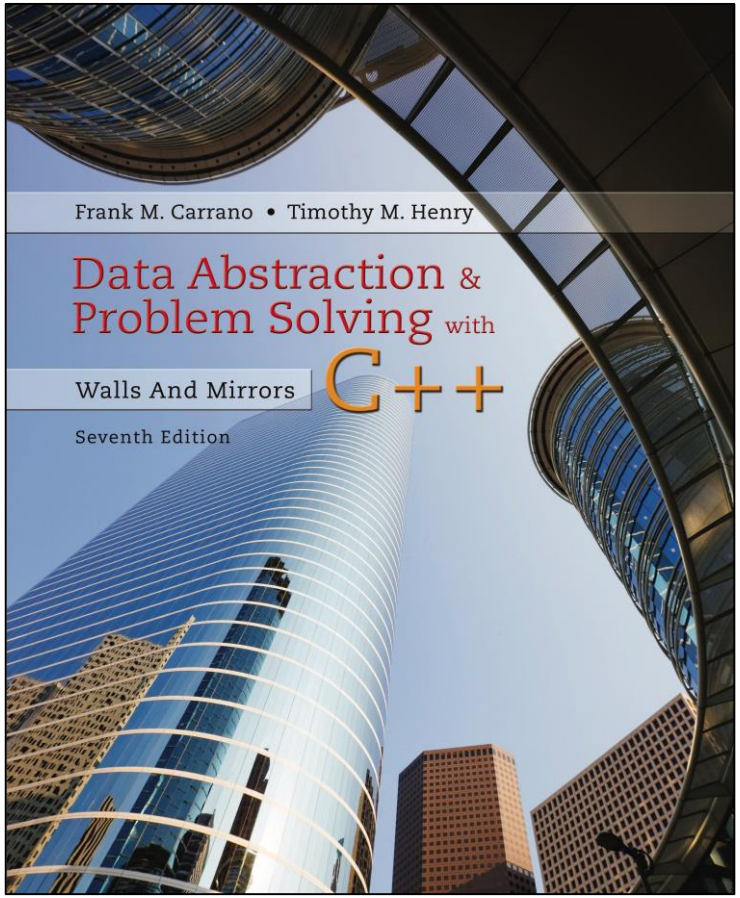
Interface Template for ADT List (4 of 4)

Listing 8-1 [Continued]

```
50
51      /** Replaces the entry at the given position in this list.
52       @pre  1 <= position <= getLength().
53       @post  The entry at the given position is newEntry.
54       @param position  The list position of the entry to replace.
55       @param newEntry  The replacement entry.
56       @return  The replaced entry. */
57      virtual ItemType replace(int position, const ItemType& newEntry) = 0;
58
59      /** Destroys this list and frees its assigned memory. */
60      virtual ~ListInterface() { }
61  }; // end ListInterface
62  #endif
```

Data Abstraction & Problem Solving with C++: Walls and Mirrors

Seventh Edition



Chapter 9

List Implementations

Array-Based Implementation of the ADT List (1 of 2)

List operations in their UML form

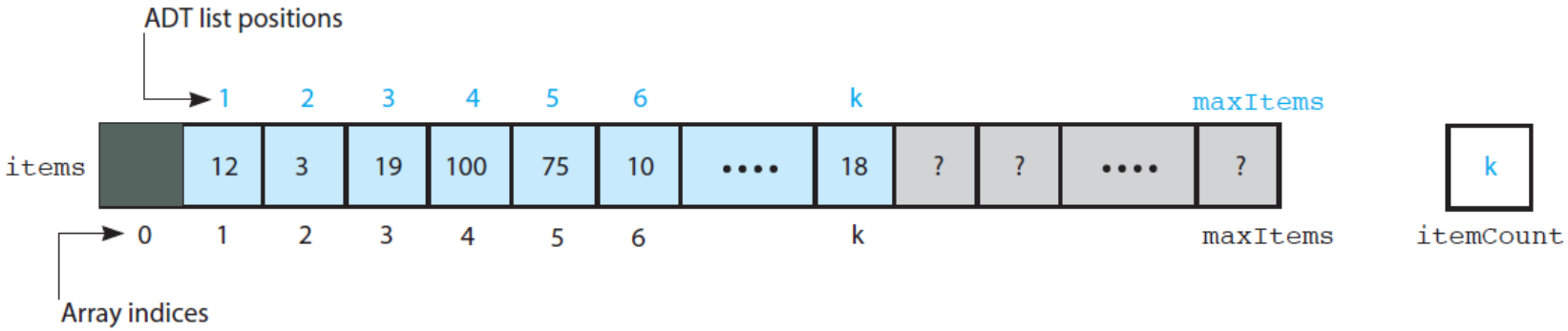
```
+isEmpty(): boolean
+getLength(): integer
+insert(newPosition: integer, newEntry: ItemType): boolean
+remove(position: integer): boolean
+clear(): void
+getEntry(position: integer): ItemType
+replace(position: integer, newEntry: ItemType): ItemType
```

Array-Based Implementation of the ADT List (2 of 2)

- Array-based implementation is a natural choice
  - Both an array and a list identify their items by number
- However
  - ADT list has operations such as **getLength** that an array does not
  - Must keep track of number of entries

The Header File (1 of 7)

Figure 9-1 An array-based implementation of the ADT list



Implementations That Use Exceptions (1 of 2)

Listing 7-5 The header file for the class **PrecondViolatedExcep**

```
1  /** @file PrecondViolatedExcept.h */
2  #ifndef PRECOND_VIOLATED_EXCEPT_
3  #define PRECOND_VIOLATED_EXCEPT_
4
5  #include <stdexcept>
6  #include <string>
7
8  class PrecondViolatedExcept: public std::logic_error
9  {
10 public:
11    PrecondViolatedExcept(const std::string& message = "");
12 }; // end PrecondViolatedExcept
13
14 #endif
```

Implementations That Use Exceptions (2 of 2)

Listing 7-6 Implementation file for the class **PrecondViolatedExcep**

```
1  /** @file PrecondViolatedExcept.cpp */
2  #include "PrecondViolatedExcept.h"
3
4  PrecondViolatedExcept::PrecondViolatedExcept(const std::string& message)
5      : std::logic_error("Precondition Violated Exception: " + message)
6  {
7  }  // end constructor
```

The Header File (2 of 7)

Listing 9-1 The header file for the class **ArrayList**

```
1  /** ADT list: Array-based implementation.
2   @file ArrayList.h */
3
4  #ifndef ARRAY_LIST_
5  #define ARRAY_LIST_
6
7  #include "ListInterface.h"
8  #include "PrecondViolatedExcept.h"
9
10 template<class ItemType>
11 class ArrayList : public ListInterface<ItemType>
12 {
13 private:
14    static const int DEFAULT_CAPACITY = 100; // Default capacity of the list
15    ItemType items[DEFAULT_CAPACITY + 1];     // Array of list items (ignore items[0])
16    int itemCount;                           // Current count of list items
17    int maxItems;                            // Maximum capacity of the list
18
```

The Header File (3 of 7)

Listing 9-1 [Continued]

```
18
19  public:
20     ArrayList();
21     // Copy constructor and destructor are supplied by compiler
22
23     bool isEmpty() const;
24     int getLength() const;
25     bool insert(int newPosition, const ItemType& newEntry);
26     bool remove(int position);
27     void clear();
28
```

The Header File (4 of 7)

Listing 9-1 [Continued]

```
29      /** @throw  PrecondViolatedExcept if position < 1 or position > getLength(). */
30      ItemType getEntry(int position) const throw(PrecondViolatedExcept);
31
32      /** @throw  PrecondViolatedExcept if position < 1 or position > getLength(). */
33      ItemType replace(int position, const ItemType& newEntry)
34                                     throw(PrecondViolatedExcept);
35  }; // end ArrayList
36
37  #include "ArrayList.cpp"
38  #endif
```

The Implementation File (1 of 24)

Constructor, methods **isEmpty** and **getLength**

```
template<class ItemType>
ArrayList<ItemType>::ArrayList() : itemCount(0), maxItems(DEFAULT_CAPACITY)
{
}  // end default constructor
```

```
template<class ItemType>
bool ArrayList<ItemType>::isEmpty() const
{
   return itemCount == 0;
}  // end isEmpty

template<class ItemType>
int ArrayList<ItemType>::getLength() const
{
   return itemCount;
}  // end getLength
```

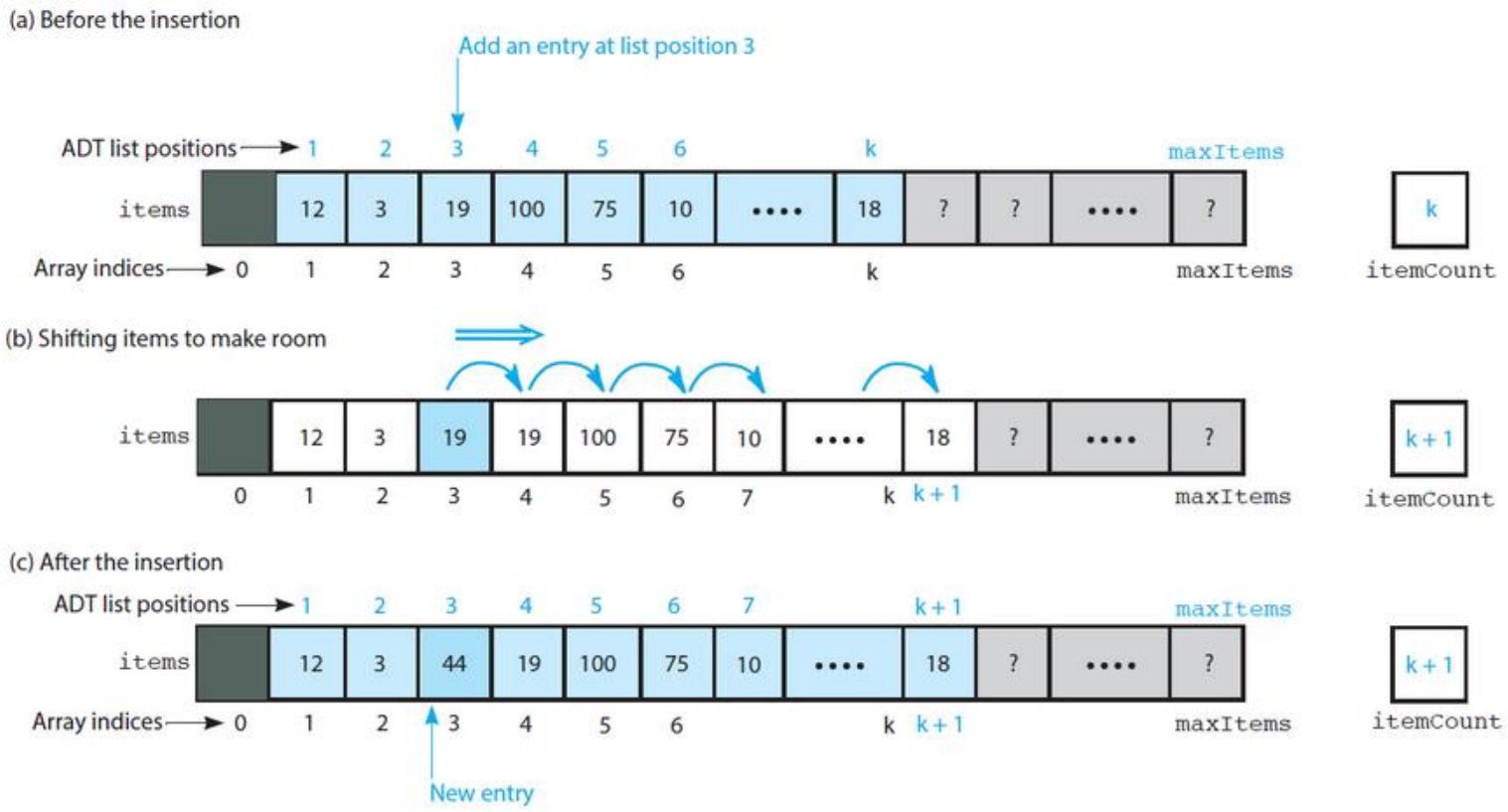
The Implementation File (2 of 24)

Method **getEntry**

```
template<class ItemType>
ItemType ArrayList<ItemType>::getEntry(int position) const
                                throw(PrecondViolatedExcept)
{
   //  Enforce precondition
   bool ableToGet = (position >= 1) && (position <= itemCount);
   if (ableToGet)
      return items[position];
   else
   {
      std::string message = "getEntry() called with an empty list or ";
      message = message + "invalid position.";
      throw(PrecondViolatedExcept(message));
   }  // end if
}  // end getEntry
```

The Implementation File (4 of 24)

Figure 9-2 Shifting items for insertion



The Implementation File (3 of 24)

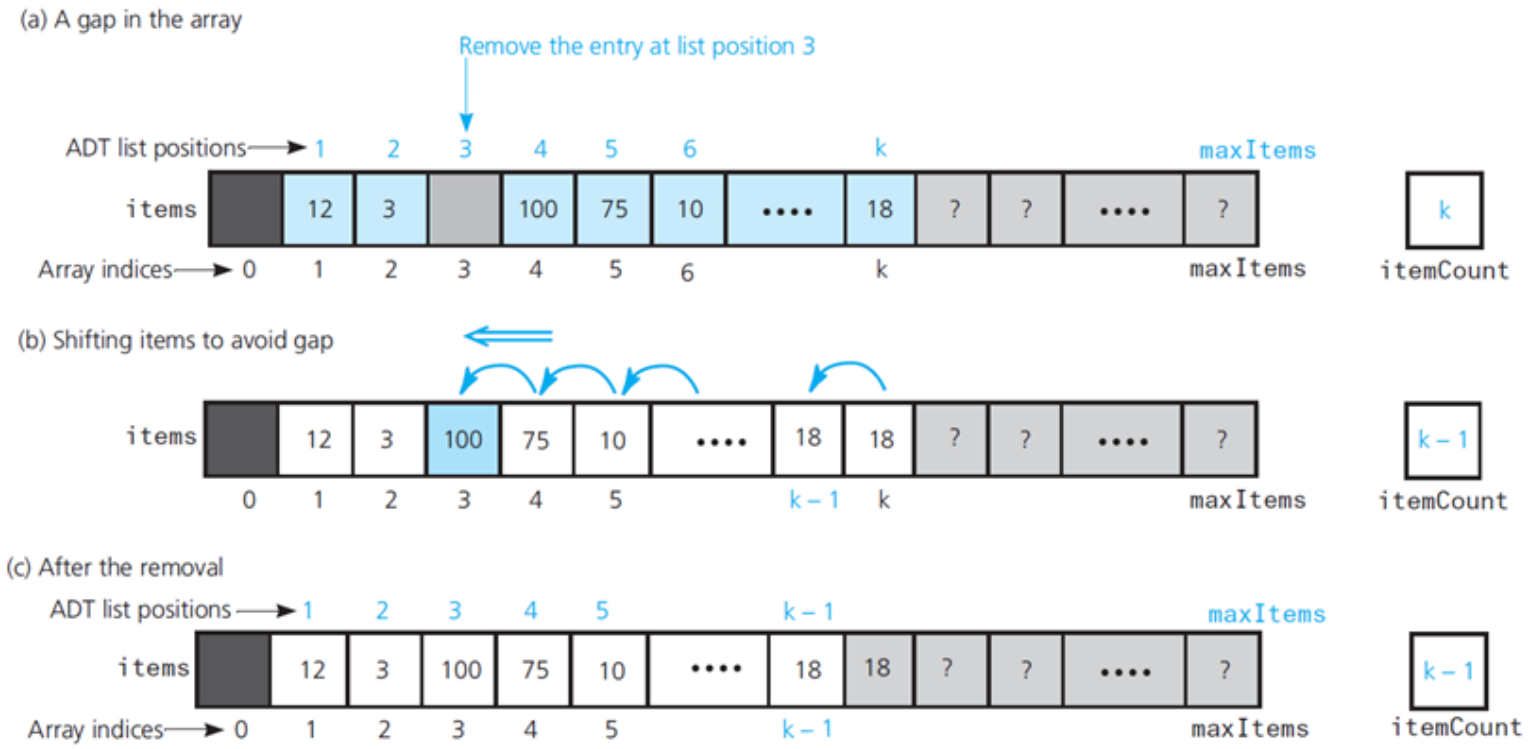
Method **insert**

```
template<class ItemType>
bool ArrayList<ItemType>::insert(int newPosition, const ItemType& newEntry)
{
   bool ableToInsert = (newPosition >= 1) && (newPosition <= itemCount + 1)
                       && (itemCount < maxItems);
   if (ableToInsert)
   {
      // Make room for new entry by shifting all entries at
      // positions from itemCount down to newPosition
      // (no shift if newPosition == itemCount + 1)
      for (int pos = itemCount; pos >= newPosition; pos--)
         items[pos + 1] = items[pos];

      // Insert new entry
      items[newPosition] = newEntry;
      itemCount++; // Increase count of entries
   }  // end if

   return ableToInsert;
}  // end insert
```

items[pos+1] = items[pos] // items shifted right by 1

The Implementation File (8 of 24)

Figure 9-3 Shifting items to remove an entry



The Implementation File (7 of 24)

Method **remove**

```
template<class ItemType>
bool ArrayList<ItemType>::remove(int position)
{
   bool ableToRemove = (position >= 1) && (position <= itemCount);
   if (ableToRemove)
   {
      // Remove entry by shifting all entries after the one at
      // position toward the beginning of the array
      // (no shift if position == itemCount)
      for (int pos = position; pos < itemCount; pos++)
         items[pos] = items[pos + 1];

      itemCount--; // Decrease count of entries
   }  // end if

   return ableToRemove;
}  // end remove
```

item at pos + 1 is now in pos

The Implementation File (6 of 24)

Method **replace**

```
template<class ItemType>
ItemType ArrayList<ItemType>::replace(int position, const ItemType& newEntry)
                              throw(PrecondViolatedExcept)
{
   // Enforce precondition
   bool ableToSet = (position >= 1) && (position <= itemCount);
   if (ableToSet)
   {
      ItemType oldEntry = items[position];
      items[position] = newEntry;
      return oldEntry;
   }
   else
   {
      std::string message = "replace() called with an empty list or ";
      message = message + "invalid position.";
      throw(PrecondViolatedExcept(message));
   }  // end if
}  // end replace
```

The Implementation File (9 of 24)

Method **clear**

```
template<class ItemType>
void ArrayList<ItemType>::clear()
{
   itemCount = 0;
}  // end clear
```

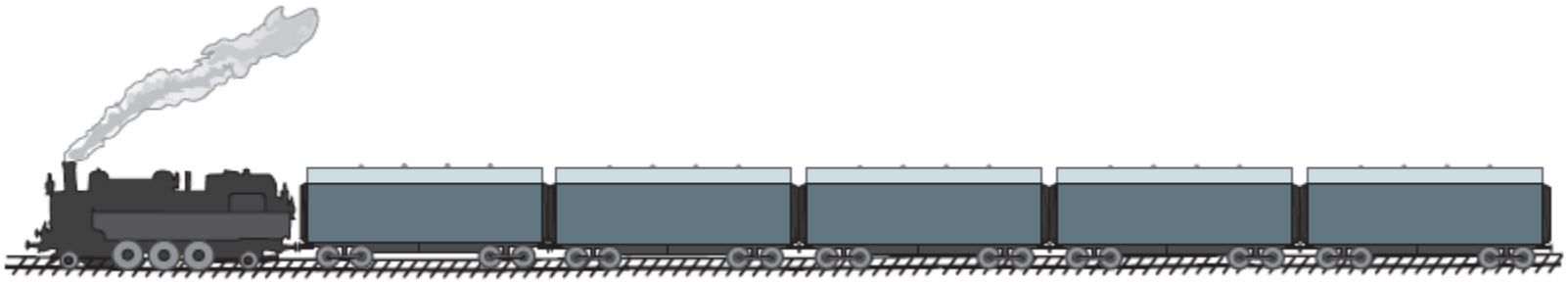
Link-Based Implementation of the ADT List (1 of 2)

- We can use C++ pointers instead of an array to implement ADT list
  - Link-based implementation does not shift items during insertion and removal operations
  - We need to represent items in the list and its length

Preliminaries (1 of 4)

- Another way to organize data items
  - Place them within objects—usually called nodes
  - Linked together into a "chain," one after the other

Figure 4-1 A freight train



Preliminaries (2 of 4)

Figure 4-2 A node

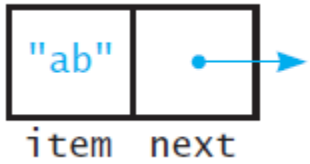
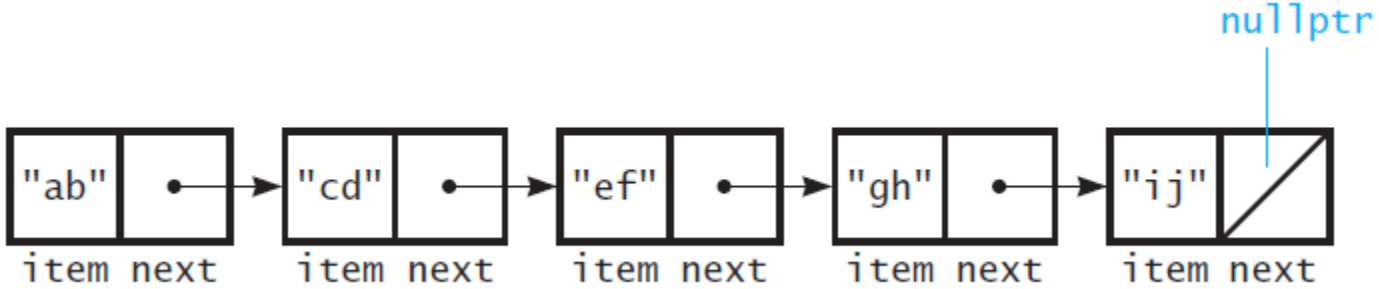
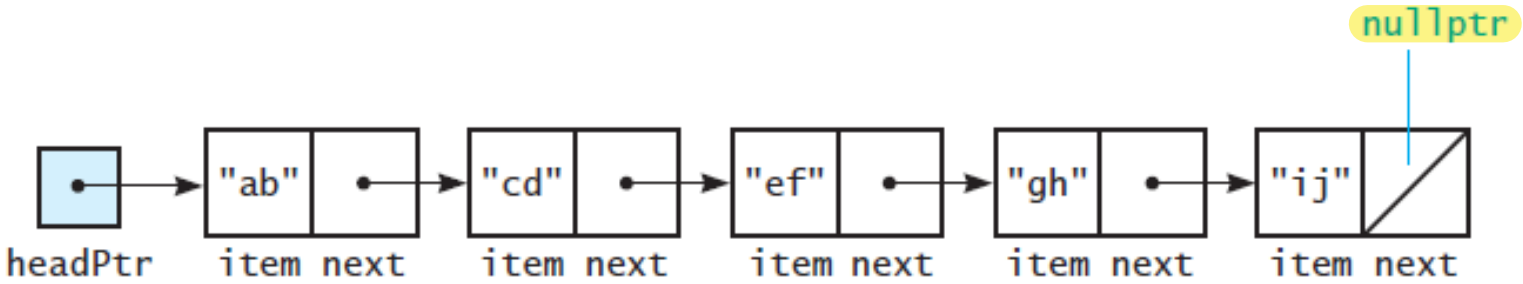


Figure 4-3 Several nodes linked together



Preliminaries (3 of 4)

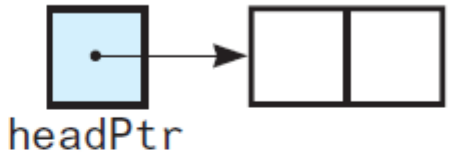
Figure 4-4 A head pointer to the first of several linked nodes



Preliminaries (4 of 4)

Figure 4-5 A lost node

```
headPtr = new Node<std::string>();
```




```
headPtr = nullptr;
```

The Class Node (1 of 3)

Listing 4-1 The header file for the template class **Node**

```
1  /** @file Node.h */
2
3  #ifndef NODE_
4  #define NODE_
5
6  template<class ItemType>
7  class Node
8  {
9  private:
10    ItemType        item; // A data item
11    Node<ItemType>* next; // Pointer to next node
12 public:
13    Node();
14    Node(const ItemType& anItem);
15    Node(const ItemType& anItem, Node<ItemType>* nextNodePtr);
16    void setItem(const ItemType& anItem);
17    void setNext(Node<ItemType>* nextNodePtr);
18    ItemType getItem() const;
19    Node<ItemType>* getNext() const;
20 }; // end Node
21 #include "Node.cpp"
22 #endif
```

The Class Node (2 of 3)

Listing 4-2 The implementation file for the class **Node**

```
/** @file Node.cpp */
#include "Node.h"
#include <cstddef>
template<class ItemType>
Node<ItemType>::Node() : next(nullptr)
{
}  // end default constructor

template<class ItemType>
Node<ItemType>::Node(const ItemType& anItem) : item(anItem), next(nullptr)
{
}  // end constructor

template<class ItemType>
Node<ItemType>::Node(const ItemType& anItem, Node<ItemType>* nextNodePtr) :
                     item(anItem), next(nextNodePtr)
{
}  // end constructor

template<class ItemType>
void Node<ItemType>::setItem(const ItemType& anItem)
{
```

The Class Node (3 of 3)

Listing 4-2 (continued)

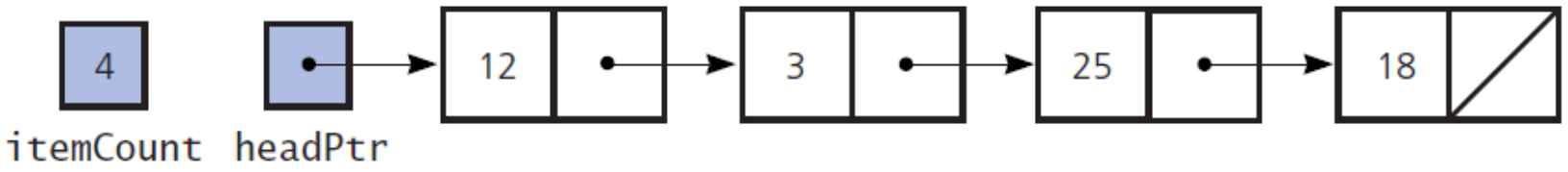
```
template<class ItemType>
void Node<ItemType>::setItem(const ItemType& anItem)
{
   item = anItem;
}  // end setItem

template<class ItemType>
void Node<ItemType>::setNext(Node<ItemType>* nextNodePtr)
{
   next = nextNodePtr;
}  // end setNext

template<class ItemType>
ItemType Node<ItemType>::getItem() const
{
   return item;
}  // end getItem

template<class ItemType>
Node<ItemType>* Node<ItemType>::getNext() const
{
   return next;
}  // end getNext
```

C-style Node Definition

```
typedef desired-type-of-list-item ItemType;

struct Node {          // a node on the list
    ItemType item;     // a data item on the list
    Node* next;        // pointer to next node
};
```

Link-Based Implementation of the ADT List (2 of 2)

Figure 9-4 A link-based implementation of the ADT list



The Header File (5 of 7)

Listing 9-2 The header file for the class **LinkedList**

```
1  /** ADT list: Link-based implementation.
2   @file LinkedList.h */
3
4  #ifndef LINKED_LIST_
5  #define LINKED_LIST_
6
7  #include "ListInterface.h"
8  #include "Node.h"
9  #include "PrecondViolatedExcept.h"
10
11 template<class ItemType>
12 class LinkedList : public ListInterface<ItemType>
13 {
14 private:
15    Node<ItemType>* headPtr; // Pointer to first node in the chain
16                             // (contains the first entry in the list)
17    int itemCount;           // Current count of list items
18    // Locates a specified node in a linked list.
```

The Header File (6 of 7)

Listing 9-2 [Continued]

```
17      int itemCount;                  // Current count of list items
18      // Locates a specified node in a linked list.
19      // @pre  position is the number of the desired node;
20      //       position >= 1 and position <= itemCount.
21      // @post  The node is found and a pointer to it is returned.
22      // @param position  The number of the node to locate.
23      // @return  A pointer to the node at the given position.
24      Node<ItemType>* getNodeAt(int position) const;
25
26    public:
27      LinkedList();
28      LinkedList(const LinkedList<ItemType>& aList);
29      virtual ~LinkedList();
30
31      bool isEmpty() const;
32      int getLength() const;
33      bool insert(int newPosition, const ItemType& newEntry);
34      bool remove(int position);
35      void clear();
```

The Header File (7 of 7)

Listing 9-2 [Continued]

```
34      bool remove(int position);
35      void clear();
36
37      /** @throw  PrecondViolatedExcept if position < 1 or
38                                          position > getLength(). */
39      ItemType getEntry(int position) const throw(PrecondViolatedExcept);
40
41      /** @throw  PrecondViolatedExcept if position < 1 or
42                                          position > getLength(). */
43      ItemType replace(int position, const ItemType& newEntry)
44                                     throw(PrecondViolatedExcept);
45  }; // end LinkedList
46
47  #include "LinkedList.cpp"
48  #endif
```

The Implementation File (10 of 24)

Constructor

```
template<class ItemType>
LinkedList<ItemType>::LinkedList() : headPtr(nullptr), itemCount(0)
{
} // end default constructor
```

The Implementation File (11 of 24)

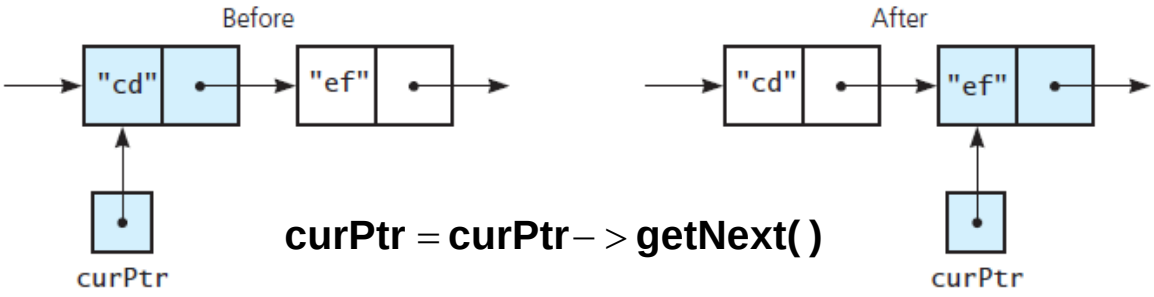
Method **getEntry**

```
template<class ItemType>
ItemType LinkedList<ItemType>::getEntry(int position) const
                                throw(PrecondViolatedExcept)
{
   // Enforce precondition
   bool ableToGet = (position >= 1) && (position <= itemCount);
   if (ableToGet)
   {
      Node<ItemType>* nodePtr = getNodeAt(position);
      return nodePtr->getItem();
   }
   else
   {
      std::string message = "getEntry() called with an empty list or ";
      message = message + "invalid position.";
      throw(PrecondViolatedExcept(message));
   }  // end if
}  // end getEntry
```

The Implementation File (12 of 24)

Method **getNodeAt**

```
template<class ItemType>
Node<ItemType>* LinkedList<ItemType>::getNodeAt(int position) const
{
   // Debugging check of precondition
   assert( (position >= 1) && (position <= itemCount) );

   // Count from the beginning of the chain
   Node<ItemType>* curPtr = headPtr;
   for (int skip = 1; skip < position; skip++)
      curPtr = curPtr->getNext();

   return curPtr ;
}  // end getNodeAt
```

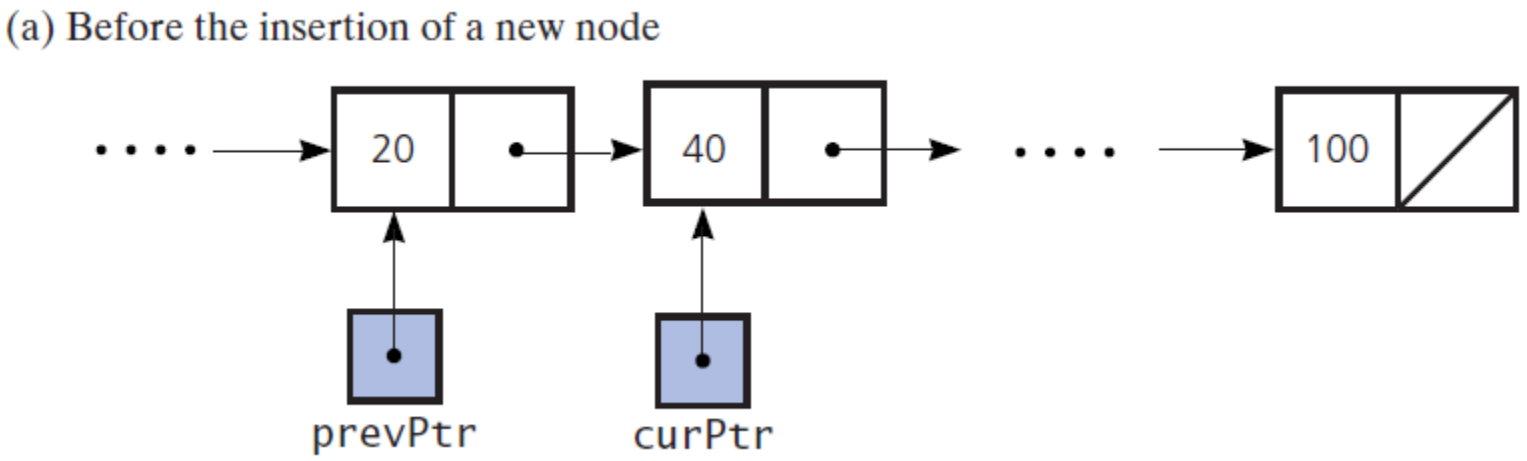


The Implementation File (13 of 24)

- Insertion process requires three high-level steps:
  1. Create a new node and store the new data in it.
  2. Determine the point of insertion.
  3. Connect the new node to the linked chain by changing pointers.

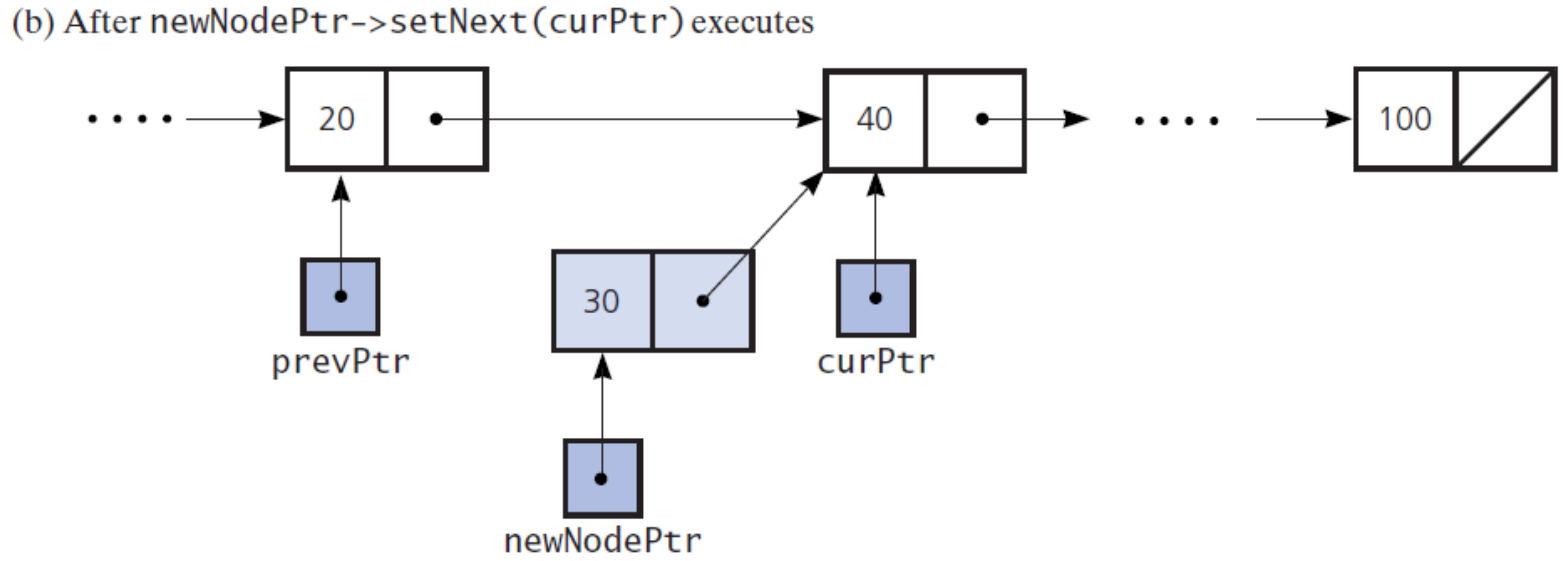
The Implementation File (16 of 24)

Figure 9-5 Inserting a new node between existing nodes of a linked chain



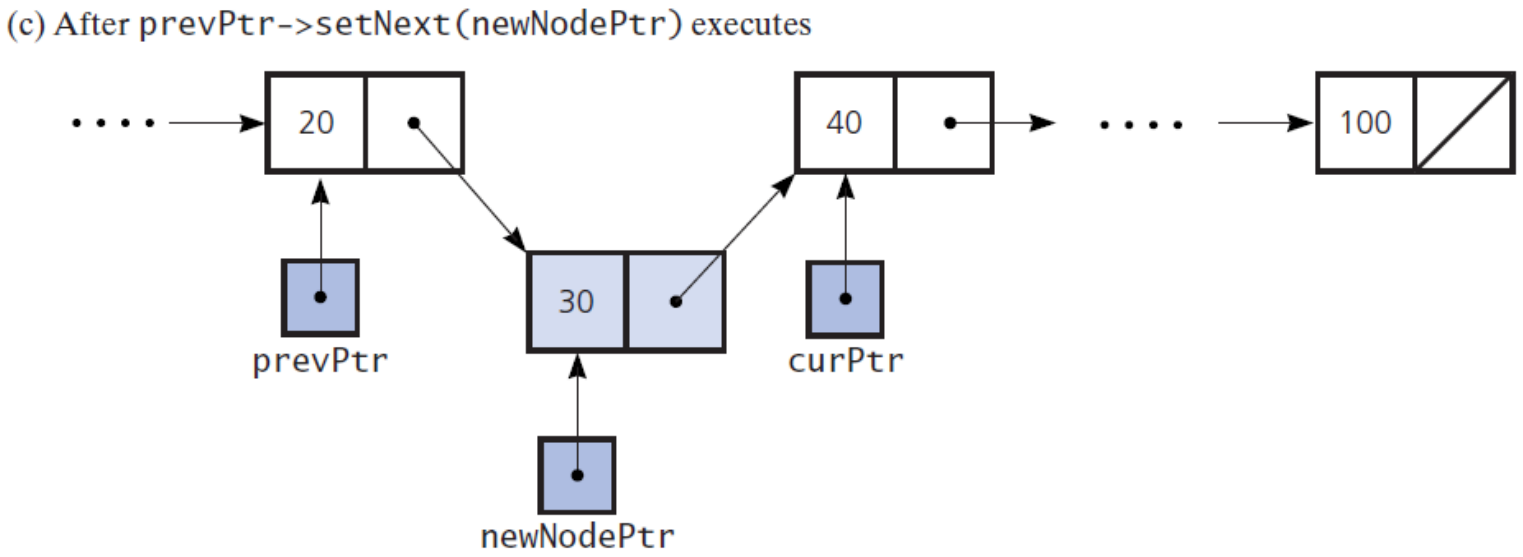
The Implementation File (17 of 24)

Figure 9-5 [Continued]



The Implementation File (18 of 24)

Figure 9-5 [Continued]



The Implementation File (19 of 24)

Figure 9-6 Inserting a new node at the end of a chain of linked nodes

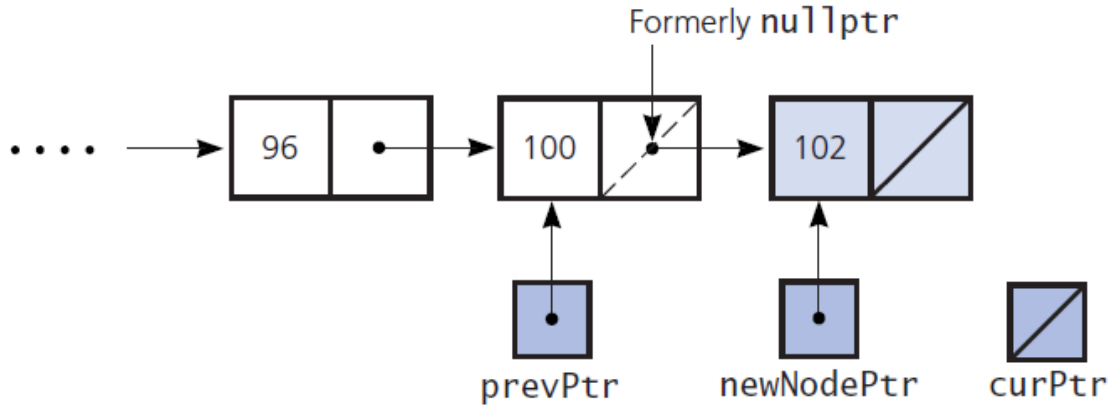
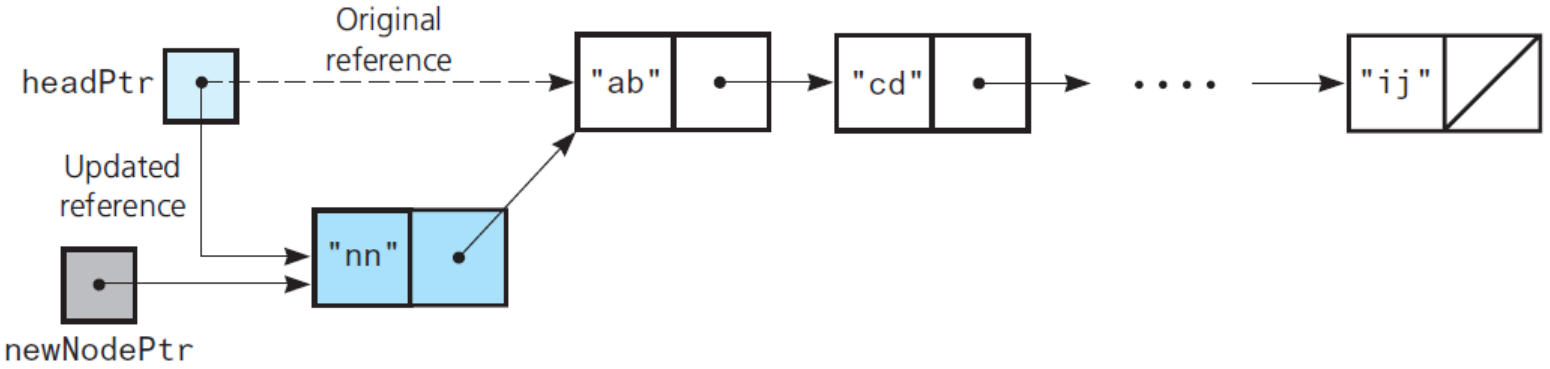


Figure 4-7 Inserting at the beginning of a linked chain



The Implementation File (14 of 24)

Method **insert**

```
template<class ItemType>
bool LinkedList<ItemType>::insert(int newPosition, const ItemType& newEntry)
{
   bool ableToInsert = (newPosition >= 1) && (newPosition <= itemCount + 1);
   if (ableToInsert)
   {
      // Create a new node containing the new entry
      Node<ItemType>* newNodePtr = new Node<ItemType>(newEntry);

      // Attach new node to chain
      if (newPosition == 1)
      {
         // Insert new node at beginning of chain
         newNodePtr->setNext(headPtr);
         headPtr = newNodePtr;
      }
      else
      {
```

The Implementation File (15 of 24)

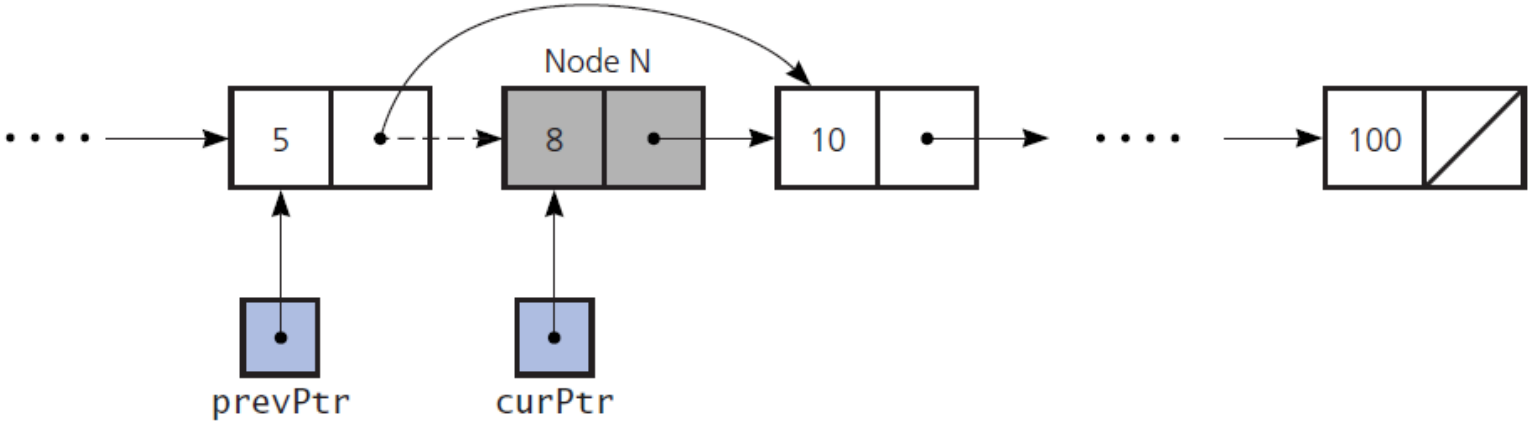
Method **insert**

```
        }
        else
        {
           // Find node that will be before new node
           Node<ItemType>* prevPtr = getNodeAt(newPosition - 1);

           // Insert new node after node to which prevPtr points
           newNodePtr->setNext(prevPtr->getNext());
           prevPtr->setNext(newNodePtr);
        }  // end if

        itemCount++; // Increase count of entries
     }  // end if

     return ableToInsert;
  }  // end insert
```

if end nullptr is next of curptr
so no special case needed

The Implementation File (20 of 24)

Figure 9-7 Removing a node from a chain



The Implementation File (21 of 24)

Figure 9-8 Removing the last node

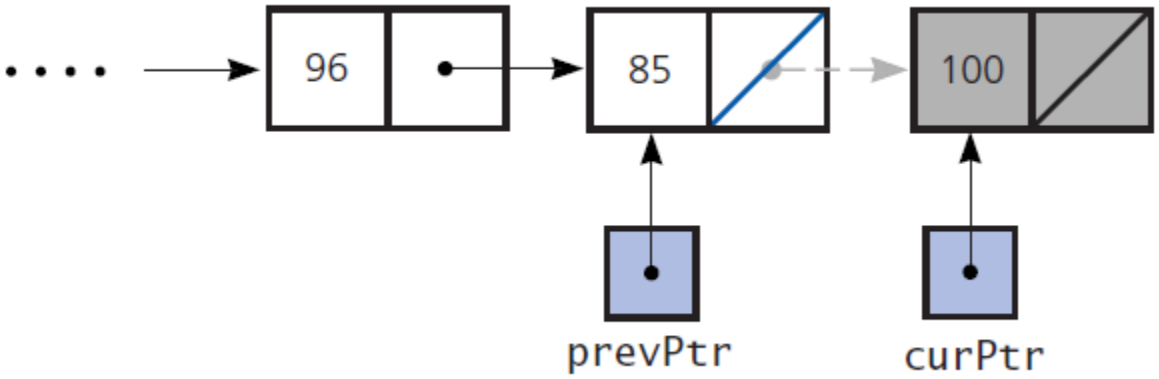
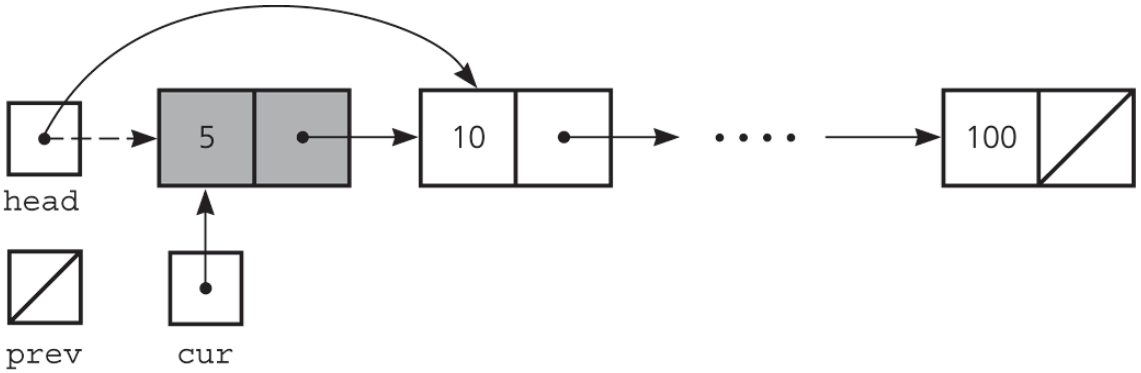


Figure Removing from the beginning of a linked chain



The Implementation File (22 of 24)

Method **remove**

```
template<class ItemType>
bool LinkedList<ItemType>::remove(int position)
{
   bool ableToRemove = (position >= 1) && (position <= itemCount);
   if (ableToRemove)
   {
      Node<ItemType>* curPtr = nullptr;
      if (position == 1)
      {
         // Remove the first node in the chain
         curPtr = headPtr; // Save pointer to node
         headPtr = headPtr->getNext();
      }
      else
      {
         // Find node that is before the one to remove
         Node<ItemType>* prevPtr = getNodeAt(position - 1);
```

used for delete

The Implementation File (23 of 24)

Method **remove**

```
        // Find node that is before the one to remove
        Node<ItemType>* prevPtr = getNodeAt(position - 1);

        // Point to node to remove
        curPtr = prevPtr->getNext();

        // Disconnect indicated node from chain by connecting the
        // prior node with the one after
        prevPtr->setNext(curPtr->getNext());
      }  // end if

      // Return node to system
      curPtr->setNext(nullptr);
      delete curPtr;
      curPtr = nullptr;

      itemCount--; // Decrease count of entries
   }  // end if

   return ableToRemove;
}  // end remove
```

for deleting save the node with curptr

The Implementation File (24 of 24)

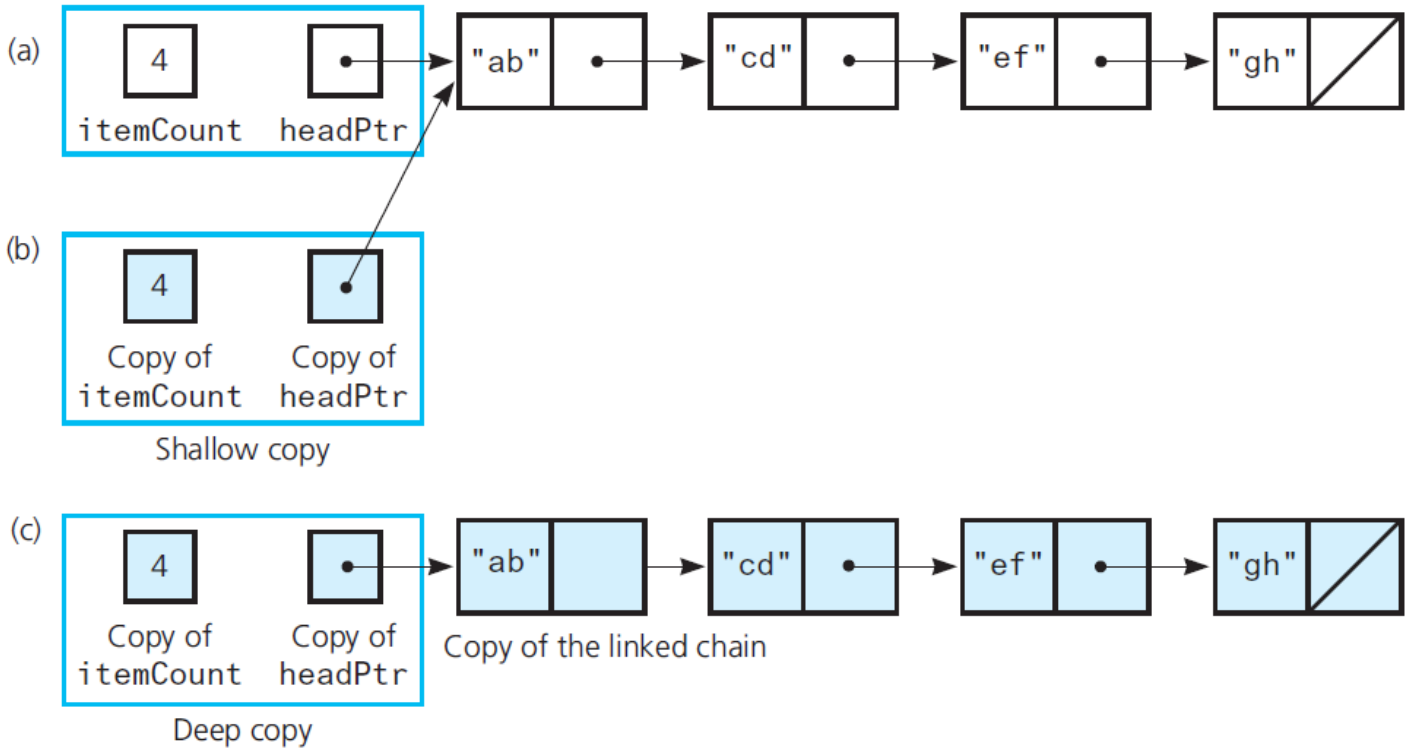
Method **clear** and the destructor

```
template<class ItemType>
void LinkedList<ItemType>::clear()
{
   while (!isEmpty())
      remove(1);
}  // end clear
```

```
template<class ItemType>
LinkedList<ItemType>::~LinkedList()
{
   clear();
}  // end destructor
```

Implementing More Methods (7 of 9)

Figure 4-9 (a) A linked chain and (b) its shallow copy; (c) deep copy



Implementing More Methods (8 of 9)

Copy constructor to accomplish deep copy.

```
template<class ItemType>
LinkedList <ItemType>:: LinkedList (const  LinkedList <ItemType>& aList )
{
   itemCount = aList.itemCount;
   Node<ItemType>* origChainPtr = aList.headPtr;

    if (origChainPtr == nullptr)
       headPtr = nullptr; // Original  list is empty; so is copy
    else
    {
       // Copy first node
       headPtr = new Node<ItemType>();
       headPtr->setItem(origChainPtr->getItem());

       // Copy remaining nodes
       Node<ItemType>* newChainPtr = headPtr;       // Last-node pointer
           origChainPtr = origChainPtr->getNext(); // Advance pointer
       while (origChainPtr != nullptr)
       {
```

Implementing More Methods (9 of 9)

Copy constructor to accomplish deep copy.

```
            origChainPtr = origChainPtr->getNext(); // Advance pointer
        while (origChainPtr != nullptr)
        {
            // Get next item from original chain
            ItemType nextItem = origChainPtr->getItem();

            // Create a new node containing the next item
            Node<ItemType>* newNodePtr = new Node<ItemType>(nextItem);

            // Link new node to end of new chain
            newChainPtr->setNext(newNodePtr);

            // Advance pointers
            newChainPtr = newChainPtr->getNext();
            origChainPtr = origChainPtr->getNext();
        }  // end while

        newChainPtr->setNext(nullptr); // Flag end of new chain
    }  // end if
}  // end copy constructor
```

Using Recursion in LinkedList Methods (1 of 5)

- Possible to process a linked chain by
  - Processing its first node and
  - Then the rest of the chain recursively
- Logic used to add a node

```
if (the insertion position is 1)
    Add the new node to the beginning of the chain
else
    Ignore the first node and add the new node to the rest of the chain
```

- Adding to the beginning of a chain—or subchain—is the base case of this recursion. The beginning of a chain is the easiest place to make an addition.

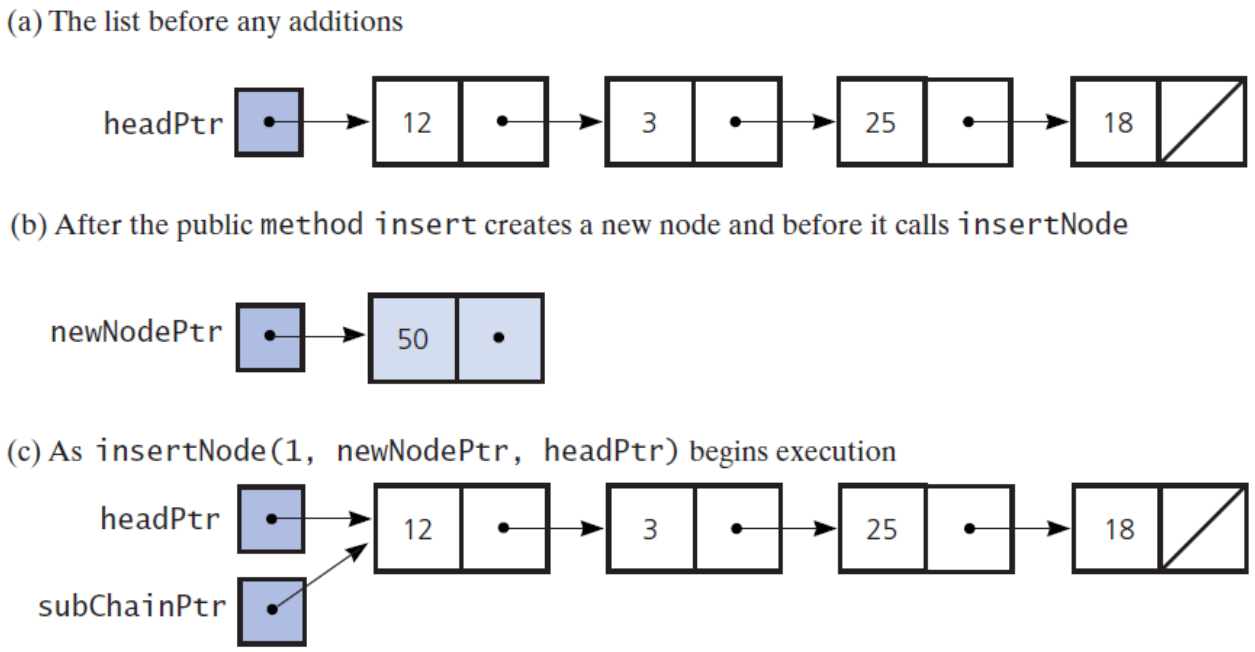
Using Recursion in LinkedList Methods

- If `position` is the desired position of the new node, `newNodePtr` points to the new node, and `subChainPtr` initially points to the chain and later points to the rest of the chain, we can add some detail to the previous logic, as follows:

```
if (position == 1)
{
   newNodePtr->setNext(subChainPtr)
   subChainPtr = newNodePtr
   Increment itemCount
}
else
   Using recursion, add the new node at position position - 1 of the subchain pointed
     to by subChainPtr->getNext()
```

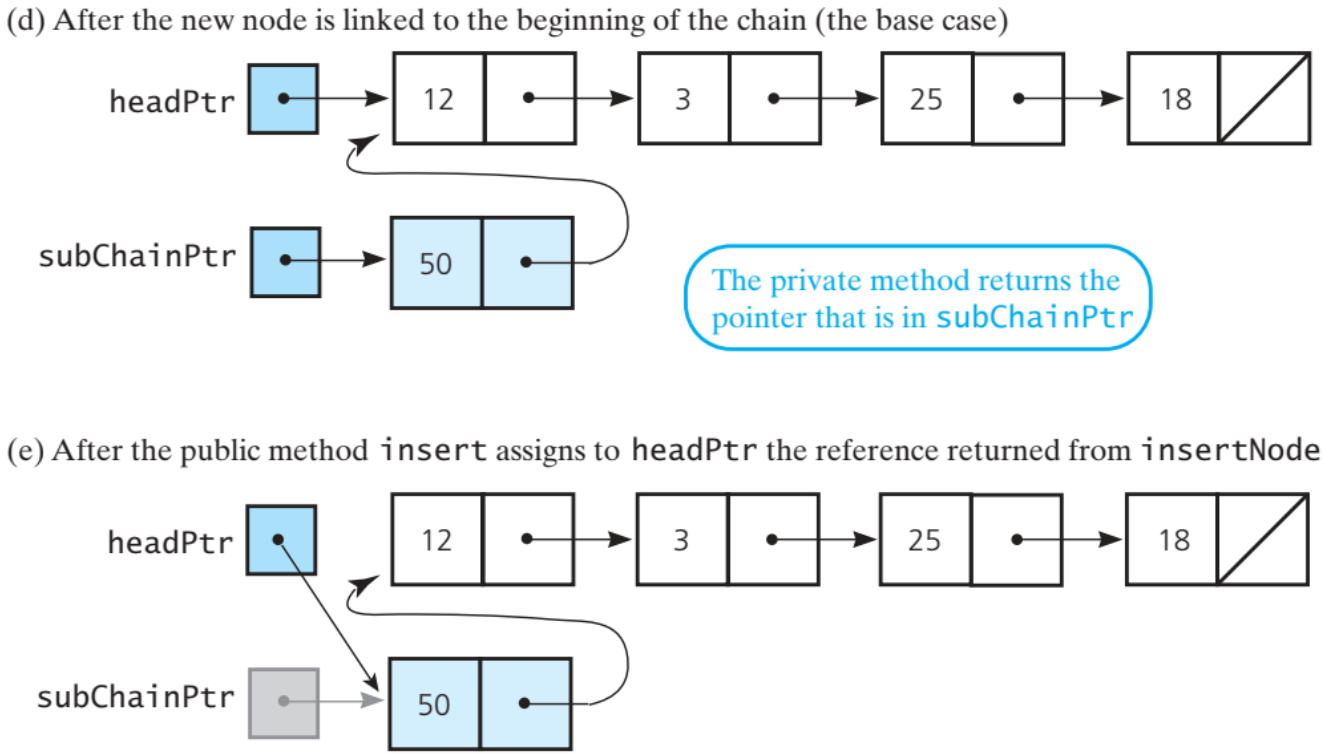
Using Recursion in LinkedList Methods (2 of 5)

Figure 9-9 Recursively adding a node at the beginning of a chain



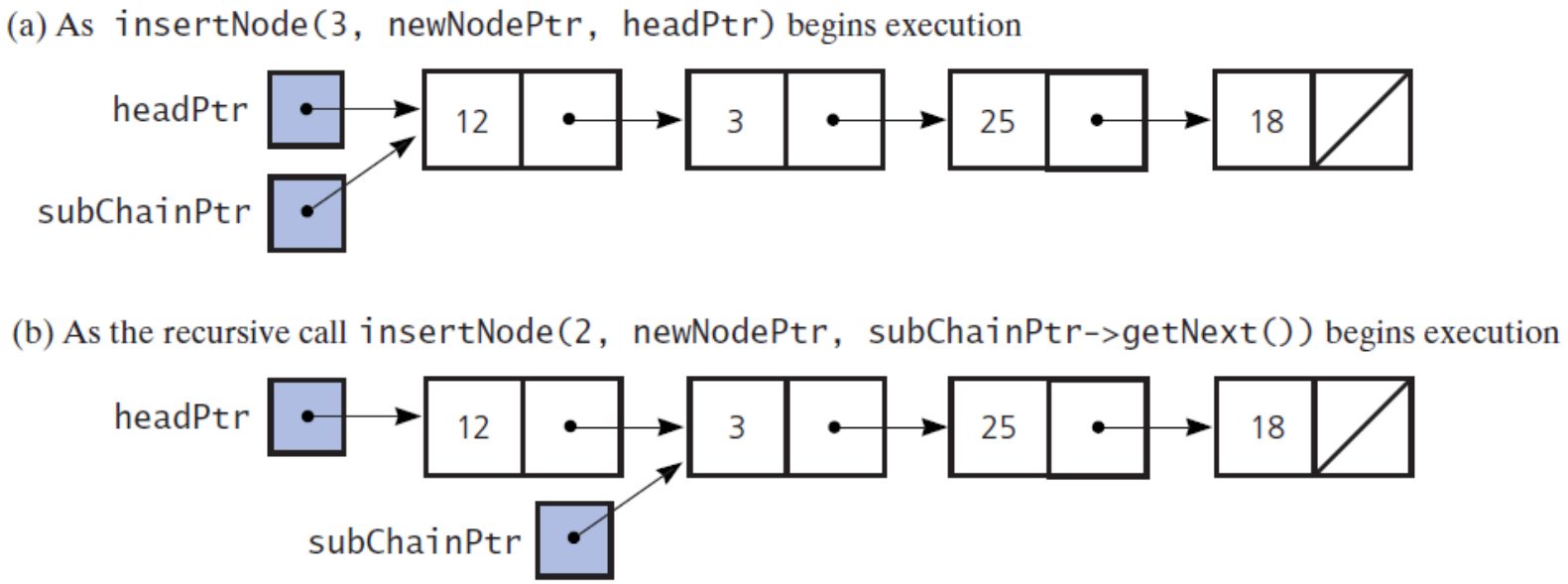
Using Recursion in LinkedList Methods (2 of 5)

Figure 9-9 [Continued]



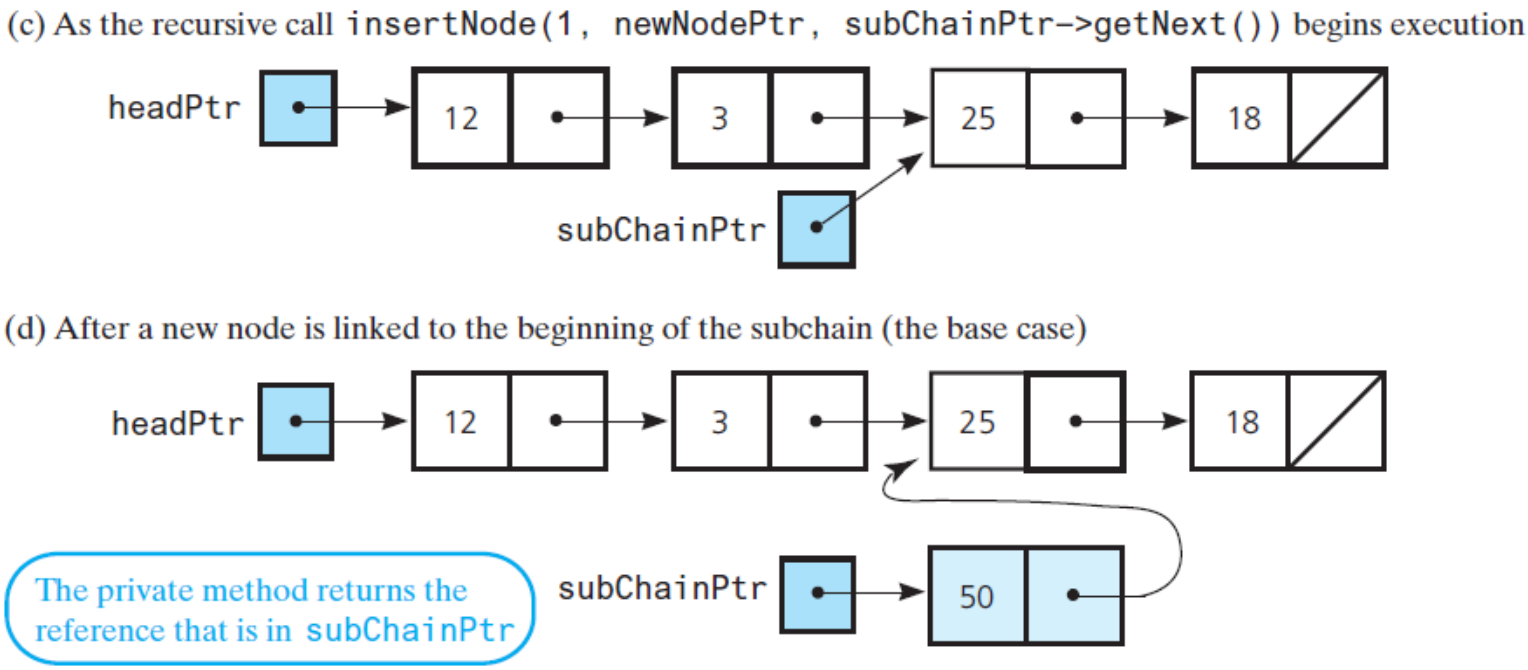
Using Recursion in LinkedList Methods (3 of 5)

Figure 9-10 Recursively adding a node between existing nodes in a chain



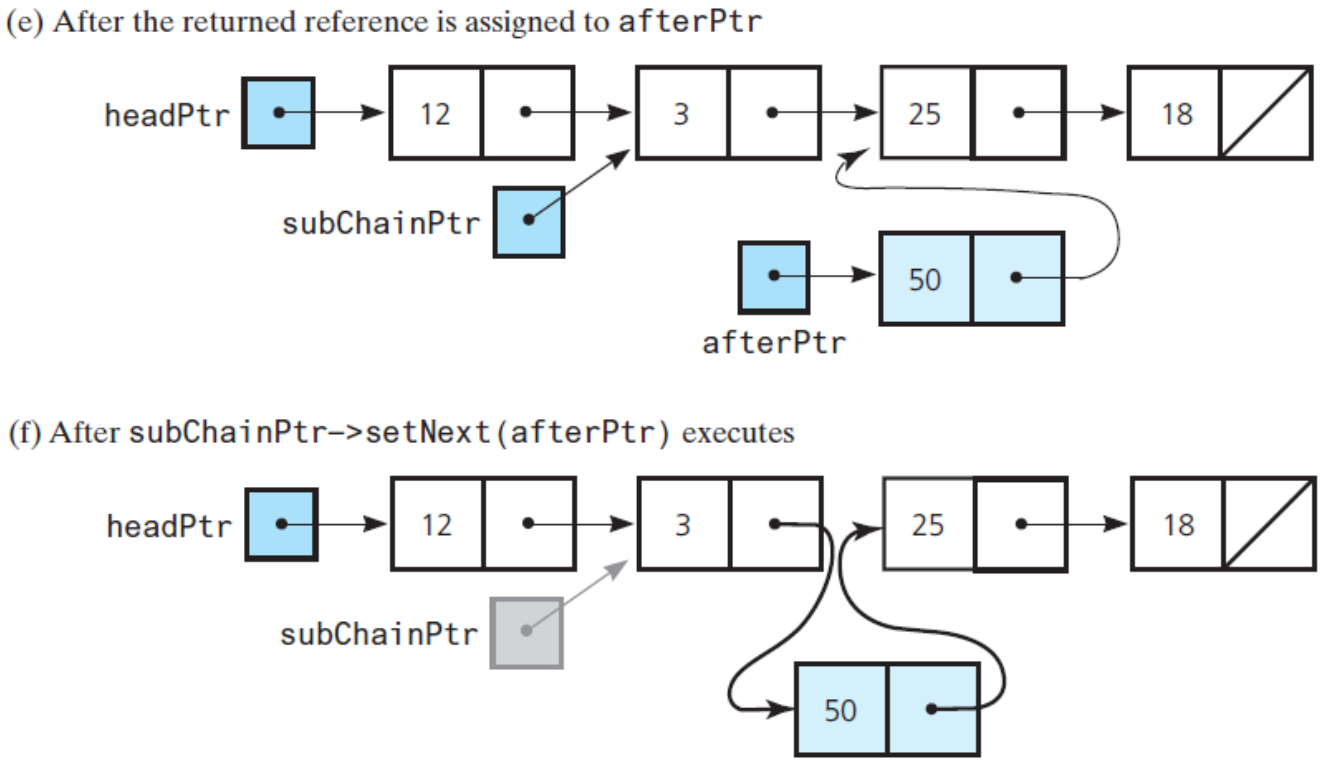
Using Recursion in LinkedList Methods (4 of 5)

Figure 9-10 [Continued]



Using Recursion in LinkedList Methods (5 of 5)

Figure 9-10 [Continued]



Using Recursion in LinkedList Methods

```
// The public method insert:
template<class ItemType>
bool LinkedList<ItemType>::insert(int newPosition, const ItemType& newEntry)
{
   bool ableToInsert = (newPosition >= 1) &&
                       (newPosition <= itemCount + 1);
   if (ableToInsert)
   {
      // Create a new node containing the new entry
      Node<ItemType>* newNodePtr = new Node<ItemType>(newEntry);
      headPtr = insertNode(newPosition, newNodePtr, headPtr);
   }  // end if

   return ableToInsert;
}  // end insert
```

Using Recursion in LinkedList Methods

```
// The private method insertNode:

// Adds a given node to the subchain pointed to by subChainPtr
// at a given position. Returns a pointer to the augmented subchain.
template<class ItemType>
Node<ItemType>* LinkedList<ItemType>::insertNode(int position,
                 Node<ItemType>* newNodePtr, Node<ItemType>* subChainPtr)
{
   if (position == 1)
   {
      // Insert new node at beginning of subchain
      newNodePtr->setNext(subChainPtr);
      subChainPtr = newNodePtr;
      itemCount++;  // Increase count of entries
   }
   else
   {
      Node<ItemType>* afterPtr =
          insertNode(position - 1, newNodePtr, subChainPtr->getNext());
      subChainPtr->setNext(afterPtr);
   }  // end if

   return subChainPtr;
}  // end insertNode
```

Example

- The ADT Sorted List
  - Maintains items in sorted order
  - Inserts and deletes items by their values, not their positions
- Determining the point of insertion or deletion for a sorted linked list

```
for(prevPtr = nullptr, curPtr = headPtr;
    (curPtr != nullptr) && (newValue > curPtr->getItem());
     prevPtr = curPtr, curPtr = curPtr->getNext());
```

Example

- Saving and Restoring a Linked List by Using a File
  - Use an external file to preserve the list between runs
  - Do not write pointers to a file, only data
  - Recreate the list from the file by placing each item at the end of the list
    - Use a tail pointer to facilitate adding nodes to the end of the list
    - Treat the first insertion as a special case by setting the tail to head

Example

- Determine whether or not a linked list is sorted
  - The linked list to which `headPtr` points is sorted if
    - `headPtr` is *`nullptr`* *or* //empty list
    - `headPtr->getNext()` is *`nullptr`* *or* 1 item list
    - `headPtr->getItem() < headPtr->getNext()->getItem()`, and

      `headPtr->getNext()` points to a sorted linked list

Example

- Insert an item into a sorted linked list

```
void linkedListInsert(Node*& headPtr, ItemType newItem) {

    if ((headPtr == nullptr) || (newItem < headPtr->item)){

        Node* newPtr = new Node;
        newPtr->item = newItem;
        newPtr->next = headPtr;
        headPtr = newPtr;
    }
    else
        linkedListInsert(headPtr->next, newItem);
}
```

Comparing Array-Based and Link-Based Implementations (1 of 2)

- Arrays easy to use, but have fixed size
  - Not always easy to predict number of items in ADT
  - Array could waste space
  - Increasing size of dynamically allocated array can waste storage **and** time
  - Can access array items directly with equal access time
  - An array-based implementation is a good choice for a small list

Comparing Array-Based and Link-Based Implementations (2 of 2)

- Linked chains do not have fixed size
  - In a chain of linked nodes, an item points explicitly to the next item
  - Link-based implementation requires more memory
  - Must traverse a linked chain to access its i^{th} node
  - Time to access i^{th} node in a linked chain depends on i
- Insertions and removals with link-based implementation
  - Do not require shifting data
  - Do require a traversal

Circular Linked Lists

- Last node references the first node
- Every node has a successor
- No node in a circular linked list contains `nullptr`

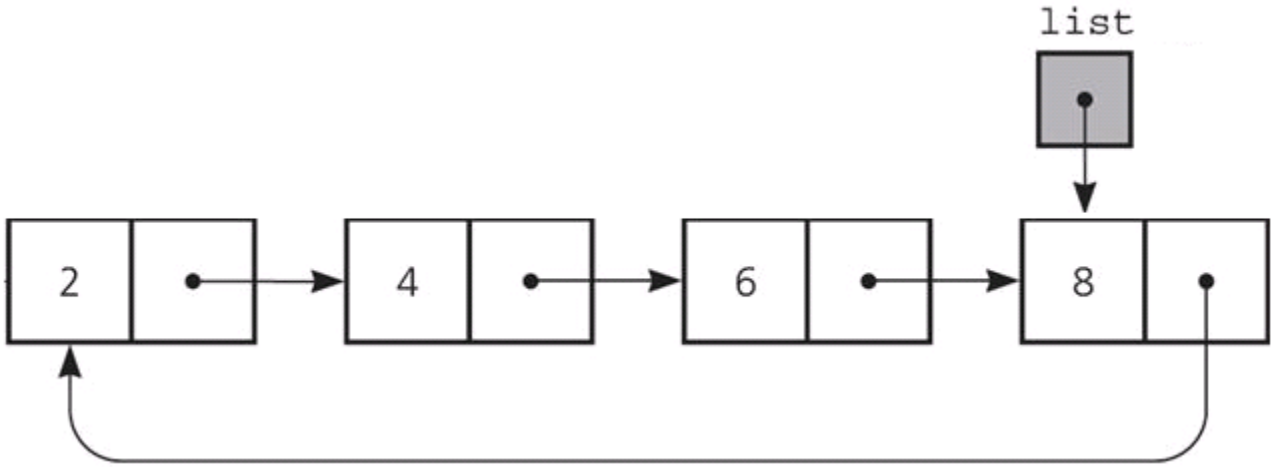


Figure A circular linked list with an external pointer to the last node

Circular Linked Lists

```
// display the data in a circular linked list;
// list points to its last node

if (list != nullptr){
   // list is not empty
   Node* first = list->getNext();// point to first node

   Node* cur = first;                // start at first node

   // Loop invariant: cur points to next node to
   // display
   do{
      display(cur->getItem());    // write data portion
      cur = cur->getNext();       // point to next node
   } while (cur != first);        // list traversed?

}
```

Dummy Head Nodes

- Dummy head node
  - Always present, even when the linked list is empty
  - Insertion and deletion algorithms initialize `prevPtr` to reference the dummy head node, rather than *`nullptr`*

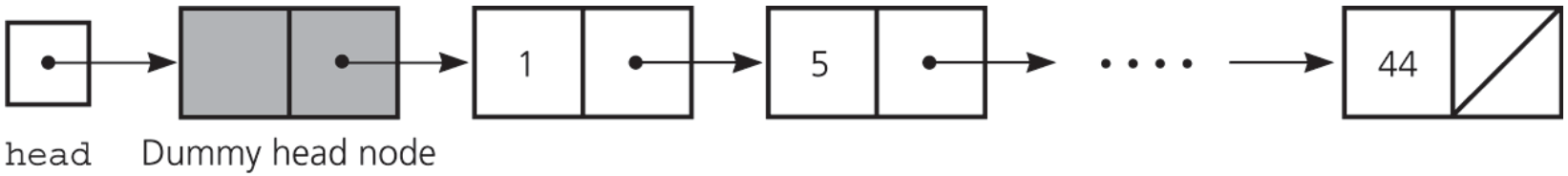


Figure A dummy head node

Doubly Linked Lists

- Each node points to both its predecessor and its successor

```
class DoublyNode {
   private:
      ItemType item;
      DoublyNode<ItemType>* next;
      DoublyNode<ItemType>* prev;
};
```

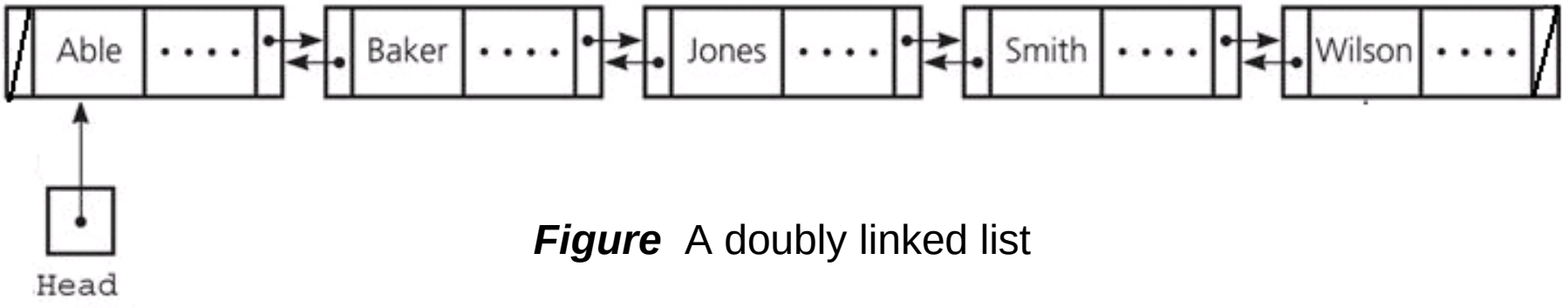


Figure A doubly linked list

Doubly Linked Lists

- To delete the node to which `cur` points

```
cur->getPrev()->setNext(cur->getNext());
cur->getNext()->setPrev(cur->getPrev());
```

- To insert a new node pointed to by `newPtr` before the node pointed to by `cur`

```
newPtr->setNext(cur);
newPtr->setPrev(cur->getPrev());
cur->setPrev(newPtr);
newPtr->getPrev()->setNext(newPtr);
```

A Circular Doubly Linked List with a Dummy Head Node

- `prev` pointer of the dummy head node points to the last node
- `next` pointer of the last node points to the dummy head node
- No special cases for insertions and deletions

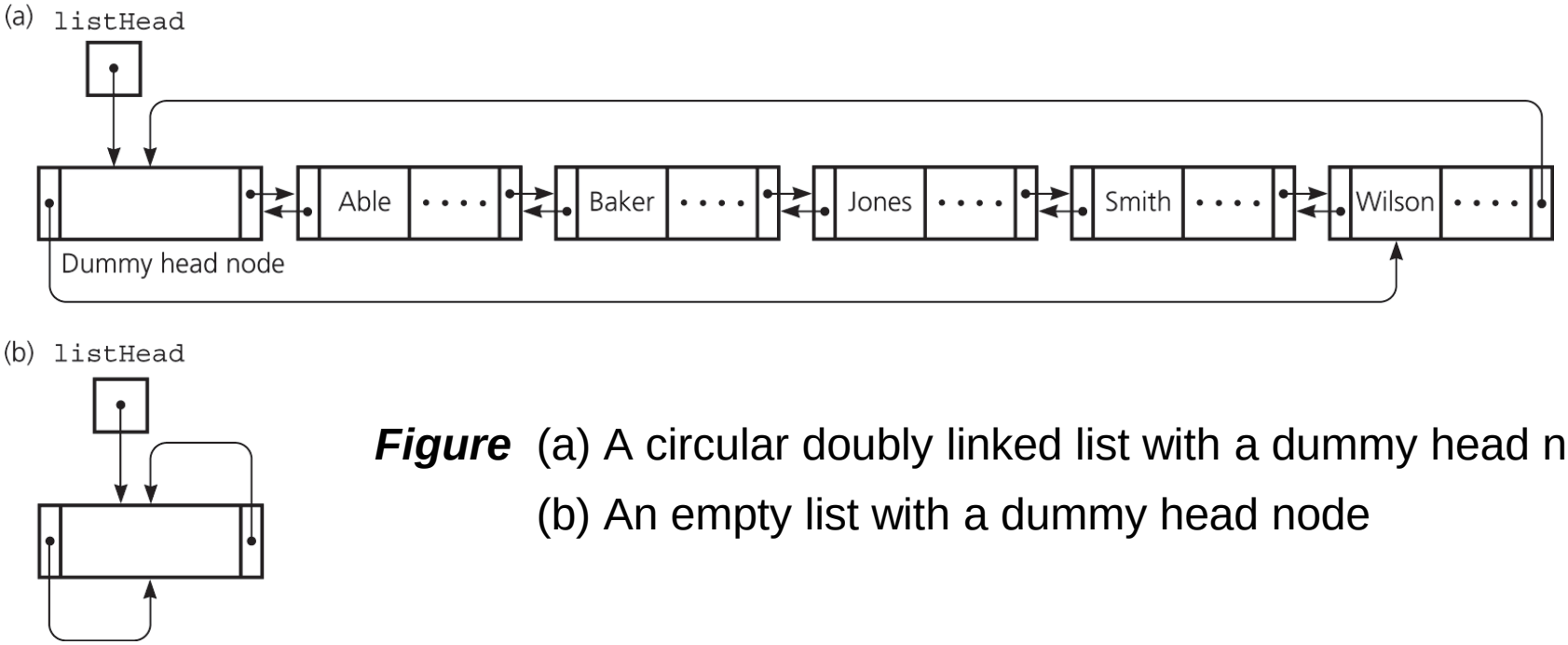


Figure (a) A circular doubly linked list with a dummy head node
(b) An empty list with a dummy head node

Copyright