A List Implementation that Links Data

Chapter 14

Data Structures and Abstractions with Java, 4e, Global Edition
Frank Carrano

Analogy

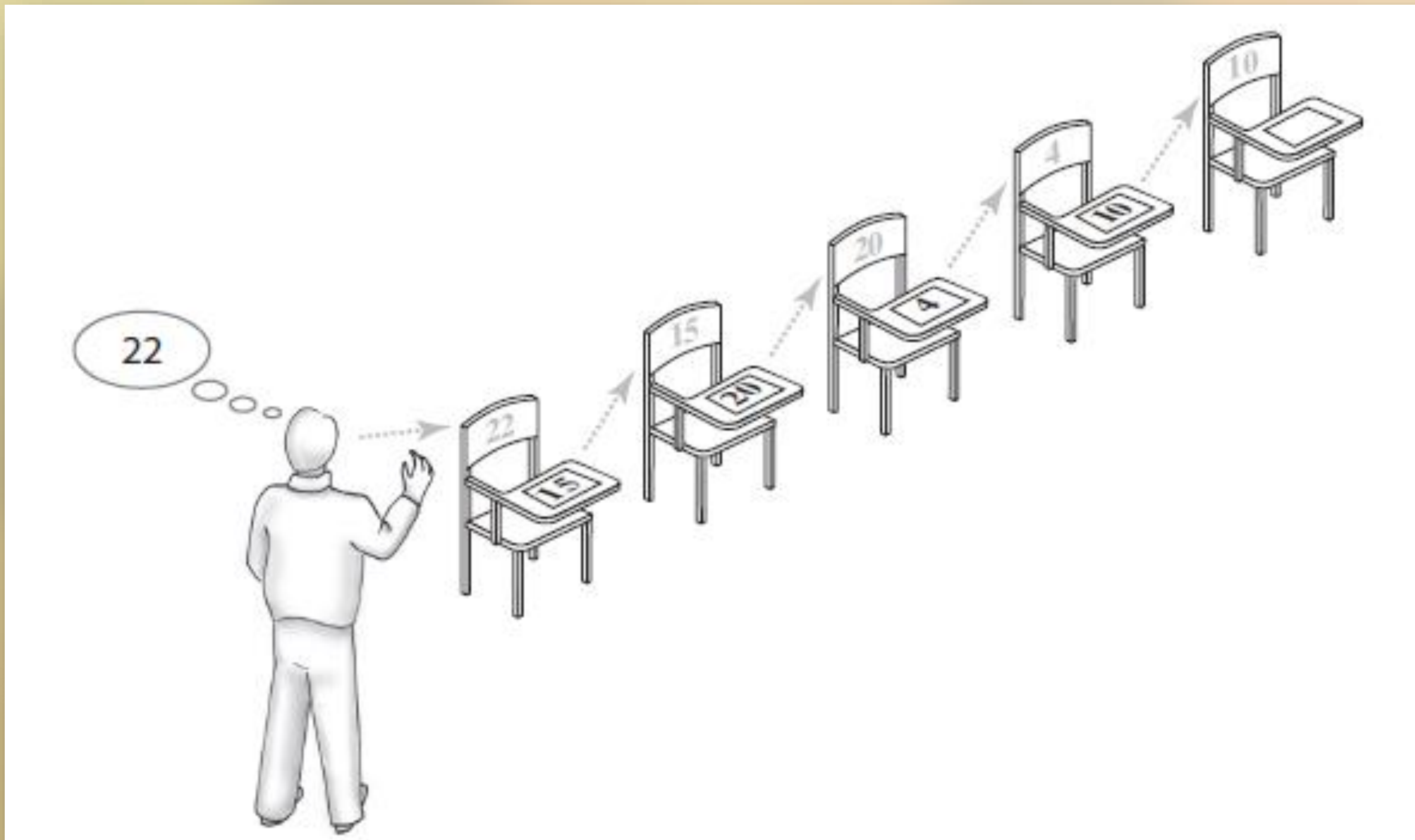


FIGURE 3-1 A chain of five desks

Advantages of Linked Implementation

- Uses memory only as needed
- When entry removed, unneeded memory returned to system
- Avoids moving data when adding or removing entries

Disadvantages of Linked Implementation

- Removing specific entry requires search of array or chain
- Chain requires more memory than array of same length

The Private Class **Node**

```java
private class Node
{
   private T    data; // Entry in bag
   private Node next; // Link to next node

   private Node(T dataPortion)
   {
      this(dataPortion, null);
   } // end constructor

   private Node(T dataPortion, Node nextNode)
   {
      data = dataPortion;
      next = nextNode;
   } // end constructor
} // end Node
```


LISTING 3-1 The private inner class **Node**

Class `Node` That Has `Set` and `Get` Methods

```

    private T getData()
    {
        return data;
    } // end getData

    private void setData(T newData)
    {
        data = newData;
    } // end setData

    private Node getNextNode()
    {
        return next;
    } // end getNextNode

```


LISTING 3-4 The inner class `Node` with `set` and `get` methods

Class **Node** That Has **Set** and **Get** Methods

```java
    private Node getNextNode()
    {
        return next;
    } // end getNextNode

    private void setNextNode(Node nextNode)
    {
        next = nextNode;
    } // end setNextNode
} // end Node
```


LISTING 3-4 The inner class **Node** with **set** and **get** methods

Adding a Node at Various Positions

Possible cases:

1. Chain is empty
2. Adding node at chain’s beginning
3. Adding node between adjacent nodes
4. Adding node to chain’s end

Adding a Node

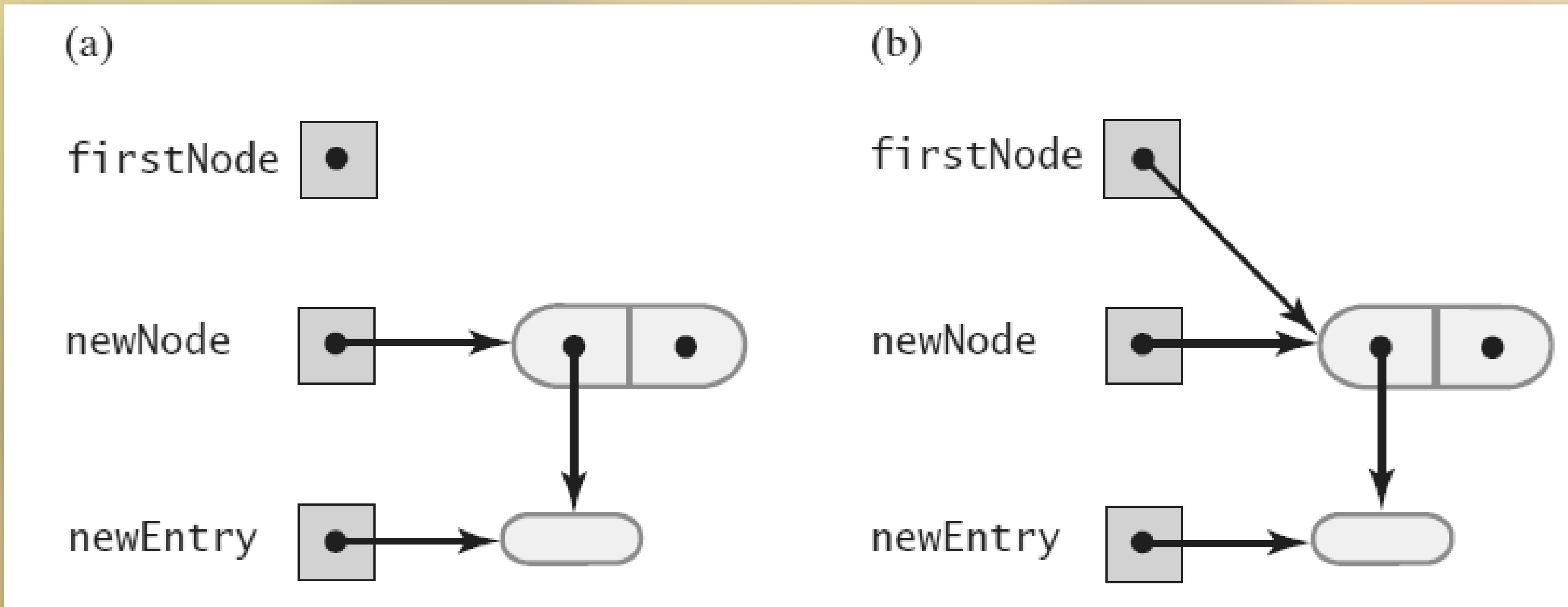


FIGURE 14-1 (a) An empty chain and a new node; (b) after adding the new node to a chain that was empty

Adding a Node

newNode *references a new instance of* Node
Place newEntry *in* newNode
firstNode = *address of* newNode

This pseudocode establishes a new node for the given data

Adding a Node

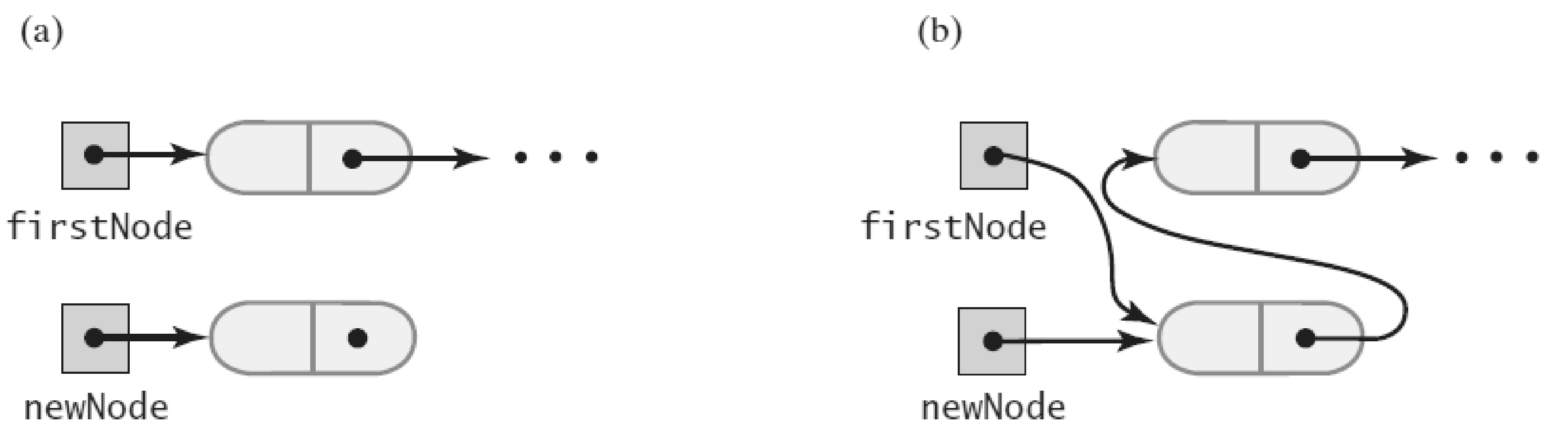


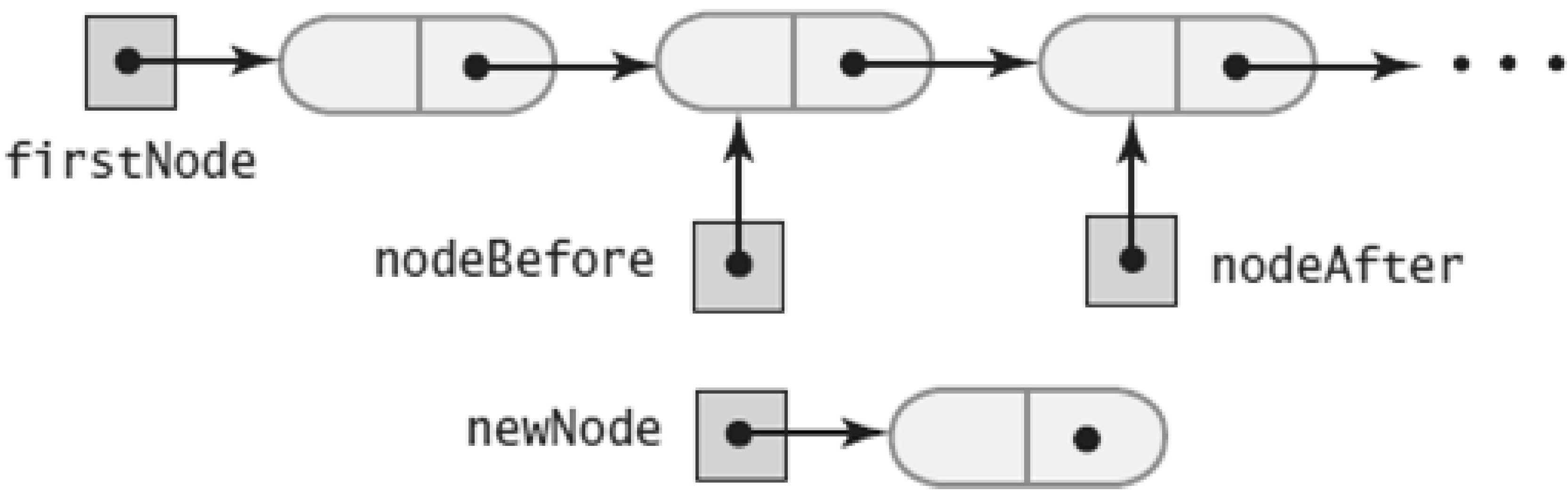
FIGURE 14-2 A chain of nodes (a) just prior to adding a node at the beginning; (b) just after adding a node at the beginning

Adding a Node

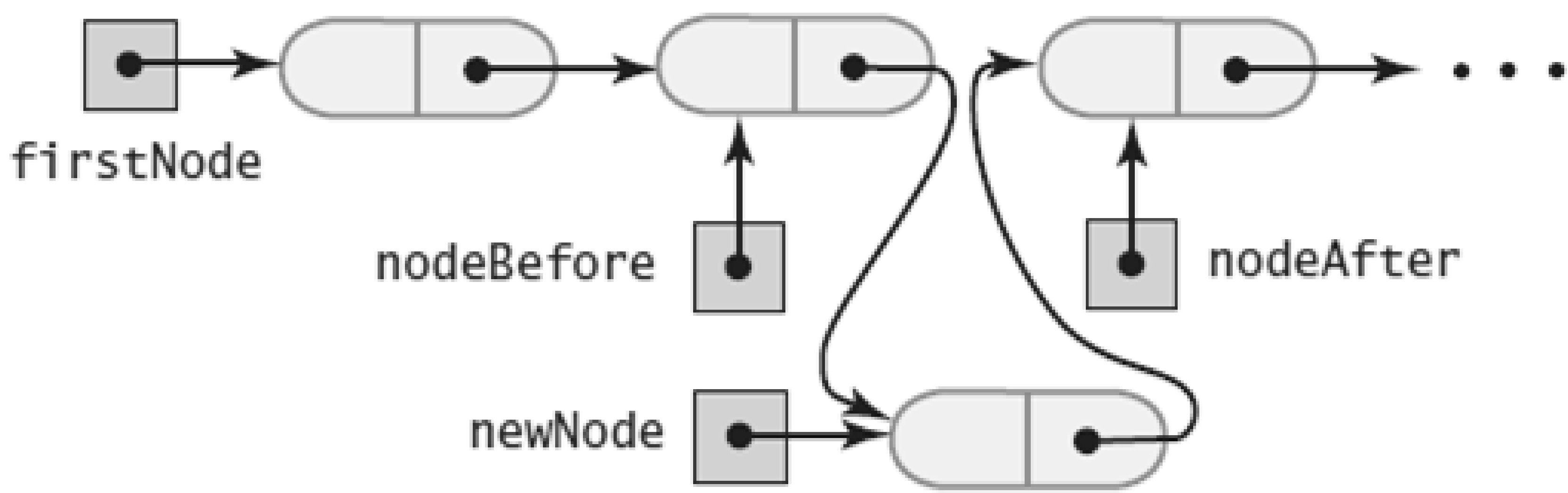
```
newNode references a new instance of Node
Place newEntry in newNode
Set newNode's link to firstNode
Set firstNode to newNode
```

This pseudocode describes the steps needed to add a node to the beginning of a chain.

(a)


firstNode
nodeBefore
nodeAfter
newNode

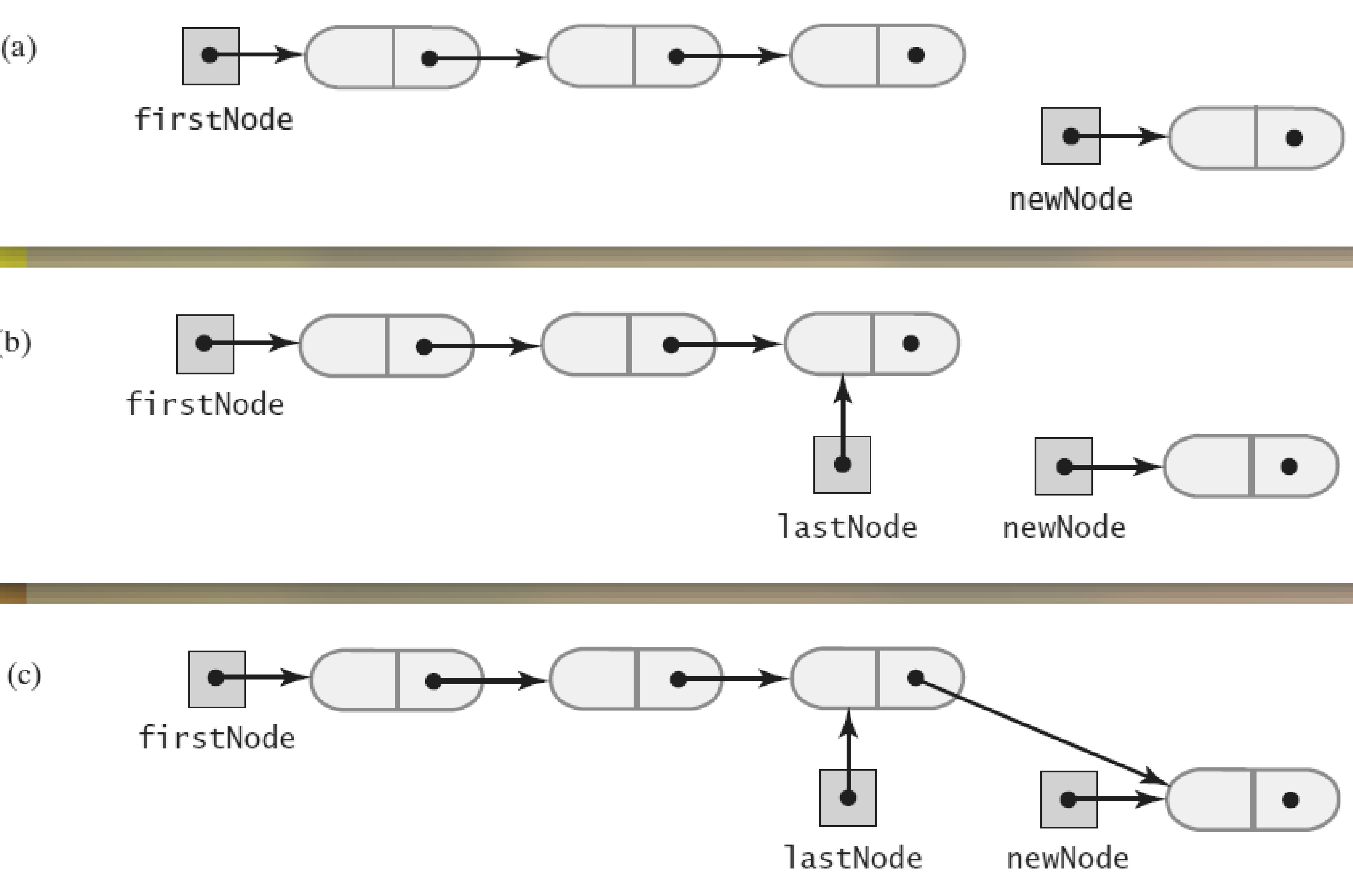
(b)


firstNode
nodeBefore
nodeAfter
newNode

Adding a Node

newNode *references the new node*
Place newEntry *in* newNode
Let nodeBefore *reference the node that will be before the new node*
Set nodeAfter *to* nodeBefore*'s link*
Set newNode*'s link to* nodeAfter
Set nodeBefore*'s link to* newNode

Pseudocode to add a node to a chain between two existing, consecutive nodes



(a)
firstNode
newNode
(b)
firstNode
lastNode
newNode
(c)
firstNode
lastNode
newNode

Adding a Node

`newNode` *references a new instance of* `Node`
Place `newEntry` *in* `newNode`
Locate the last node in the chain
Place the address of `newNode` *in this last node*

Steps to add a node at the end of a chain.

Removing a Node from Various Positions

Possible cases

1. Removing the first node
2. Removing a node other than first one

Removing a Node

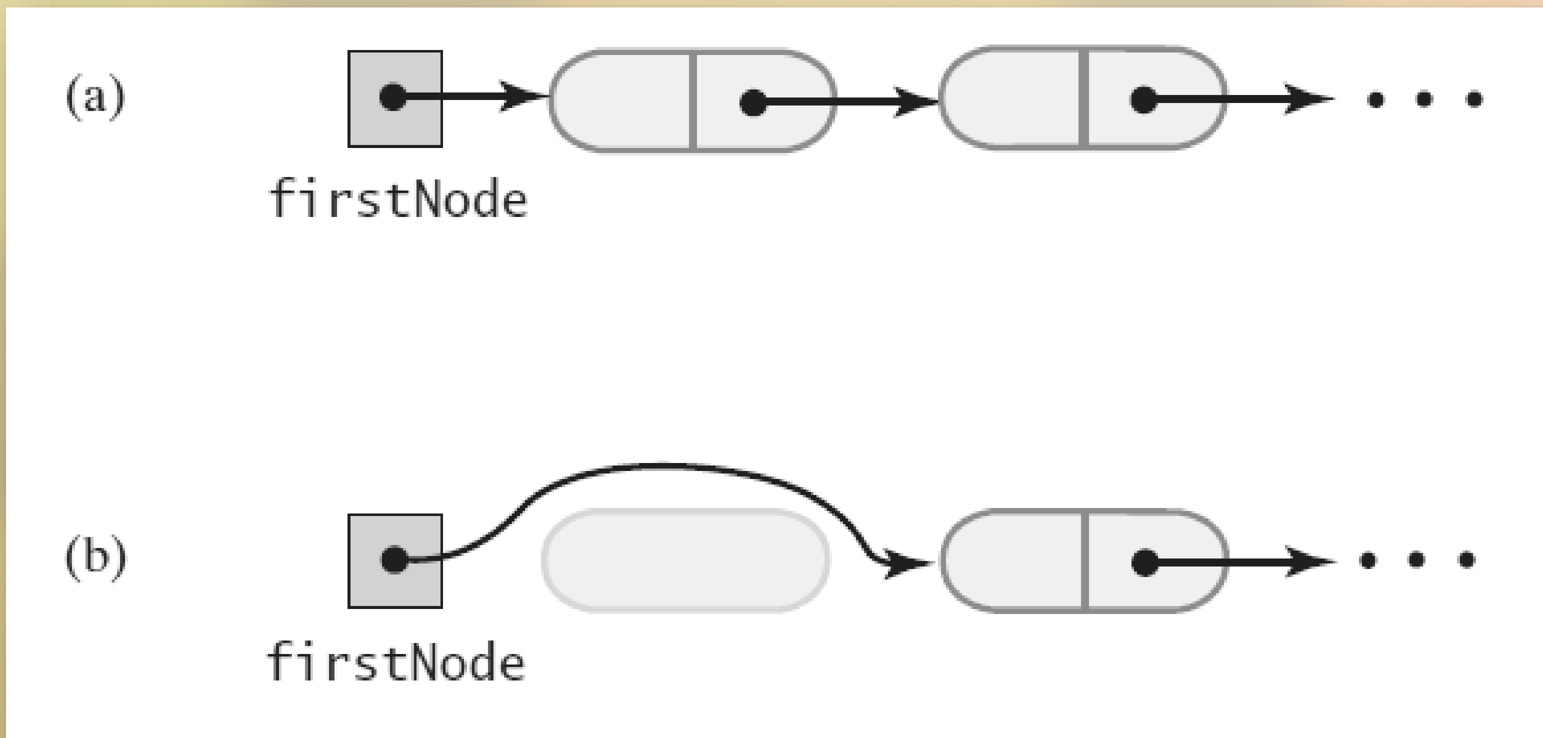


FIGURE 14-5 A chain of nodes (a) just prior to removing the first node; (b) just after removing the first node

Removing a Node

Set `firstNode` *to the link in the first node.*

Since all references to the first node no longer exist, the system automatically recycles the first node's memory.

Steps for removing the first node.

Removing a Node

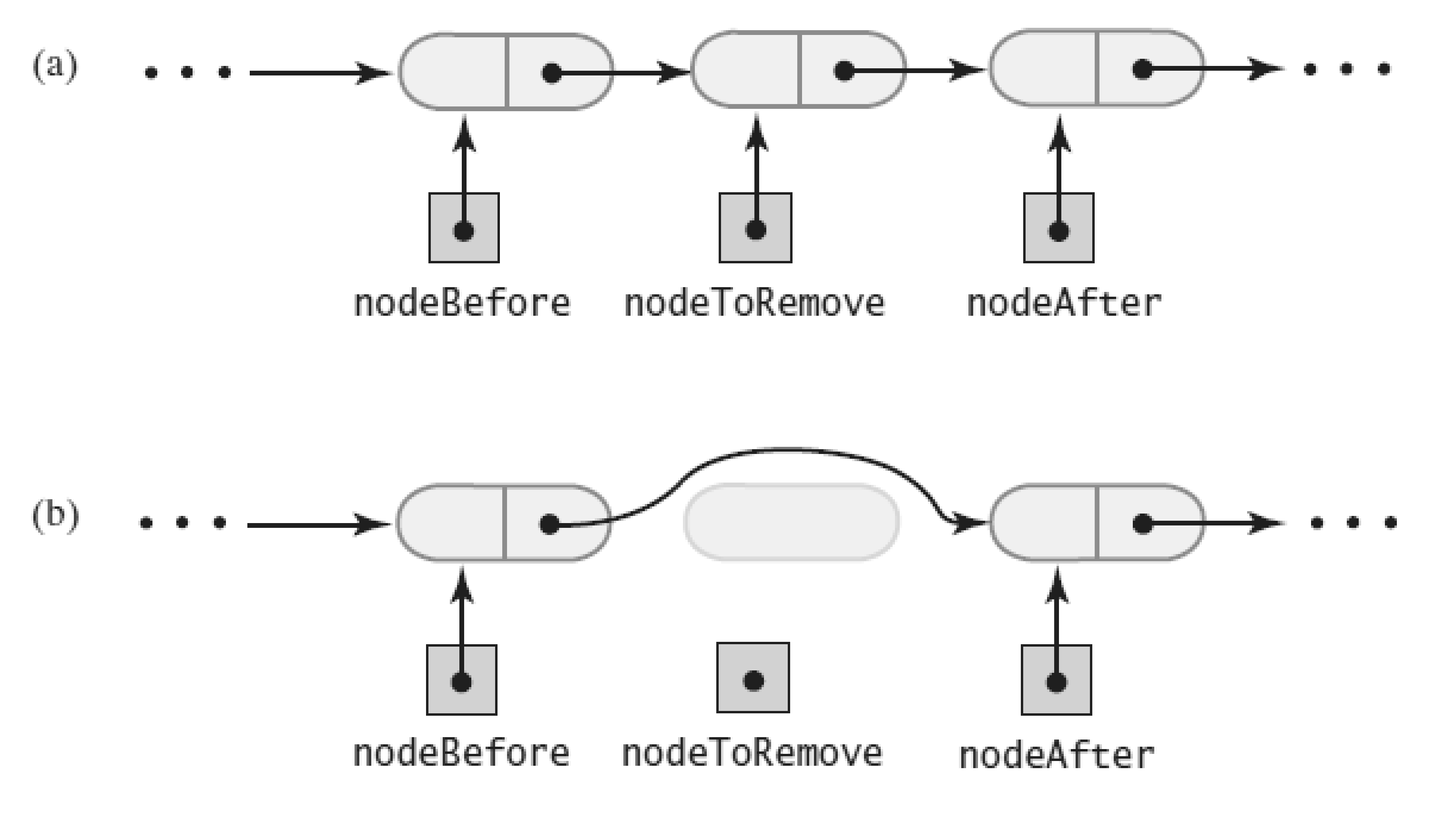


FIGURE 14-6 A chain of nodes (a) just prior to removing an interior node; (b) just after removing an interior node

Removing a Node

Let nodeBefore *reference the node before the one to be removed.*

Set nodeToRemove *to* nodeBefore*'s link;* nodeToRemove *now references the node to be removed.*

Set nodeAfter *to* nodeToRemove*'s link;* nodeAfter *now references the node after the one to be removed.*

Set nodeBefore*'s link to* nodeAfter. *(*nodeToRemove *is now disconnected from the chain.)*

Set nodeToRemove *to* null.

Since all references to the disconnected node no longer exist, the system automatically recycles the node's memory.

Removing a node other than the first one.

Removing a Node

```
private Node getNodeAt(int givenPosition)
{
   assert (firstNode != null) &&
          (1 <= givenPosition) && (givenPosition <= numberOfNodes);
   Node currentNode = firstNode;

   // Traverse the chain to locate the desired node
   // (skipped if givenPosition is 1)
   for (int counter = 1; counter < givenPosition; counter++)
      currentNode = currentNode.getNextNode();

      assert currentNode != null;

      return currentNode;
} // end getNodeAt
```

Operations on a chain depended on the method **getNodeAt**

Continuing the Implementation

```
public T remove(int givenPosition)
{
   T result = null;                           // Return value
   if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
   {
      assert !isEmpty();
      if (givenPosition == 1)                 // Case 1: Remove first entry
      {
         result = firstNode.getData();        // Save entry to be removed
         firstNode = firstNode.getNextNode(); // Remove entry
      }
      else                                    // Case 2: Not first entry
```

The `remove` method returns the entry that it deletes from the list

Continuing the Implementation

```
    else                                        // Case 2: Not first entry
    {
       Node nodeBefore = getNodeAt(givenPosition - 1);
       Node nodeToRemove = nodeBefore.getNextNode();
       result = nodeToRemove.getData();         // Save entry to be removed
       Node nodeAfter = nodeToRemove.getNextNode();
       nodeBefore.setNextNode(nodeAfter);       // Remove entry
    } // end if
    numberOfEntries--;                          // Update count
    return result;                              // Return removed entry
  }
  else
  throw new IndexOutOfBoundsException(
          "Illegal position given to remove operation.");
} // end remove
```

The **`remove`** method returns the entry that it deletes from the list

Continuing the Implementation

```
public T replace(int givenPosition, T newEntry)
{
   if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
   {
      assert !isEmpty();
      Node desiredNode = getNodeAt(givenPosition);
      T originalEntry = desiredNode.getData();
      desiredNode.setData(newEntry);
      return originalEntry;
   }
   else
      throw new IndexOutOfBoundsException(
                "Illegal position given to replace operation.");
} // end replace
```

Replacing a list entry requires us to replace the data portion of a node with other data.

Data Fields and Constructor

```
/**
   A linked implementation of the ADT list.
   @author Frank M. Carrano
*/
public class LList<T> implements ListInterface<T>
{
   private Node firstNode; // Reference to first node of chain
   private int numberOfEntries;

   public LList()
   {
      initializeDataFields();
   } // end default constructor

   public void clear()
   {
      initializeDataFields();
   } // end clear
   < Implementations of the public methods add, remove, replace, getEntry, contains,
     getLength, isEmpty, and toArray go here. >
   . . .
```


LISTING 14-1 An outline of the class `LList`

Data Fields and Constructor

```java

   // Initializes the class's data fields to indicate an empty list.
   private void initializeDataFields()
   {
      firstNode = null;
      numberOfEntries = 0;
   } // end initializeDataFields

   // Returns a reference to the node at a given position.
   // Precondition: List is not empty;
   //               1 <= givenPosition <= numberOfEntries.
   private Node getNodeAt(int givenPosition)
   {
      < See Segment 14.7. >
   } // end getNodeAt

   private class Node // Private inner class
   {
      < See Listing 3-4 in Chapter 3. >
   } // end Node
} // end LList
```


LISTING 14-1 An outline of the class `LList`

Adding to the End of the List

```
public void add(T newEntry)
{
   Node newNode = new Node(newEntry);

   if (isEmpty())
      firstNode = newNode;
   else                               // Add to end of nonempty list
   {
      Node lastNode = getNodeAt(numberOfEntries);
      lastNode.setNextNode(newNode); // Make last node reference new node
   } // end if

   numberOfEntries++;
} // end add
```

The method `add` assumes method `getNodeAt`

Adding at a Given Position

```
public void add(int newPosition, T newEntry)
{
   if ((newPosition >= 1) && (newPosition <= numberOfEntries + 1))
   {
      Node newNode = new Node(newEntry);
      if (newPosition == 1)                    // Case 1
      {
         newNode.setNextNode(firstNode);
         firstNode = newNode;
      }
```

Adding at a Given Position

```
      else                               // Case 2: List is not empty
      {                                  // and newPosition > 1
         Node nodeBefore = getNodeAt(newPosition - 1);
         Node nodeAfter = nodeBefore.getNextNode();
         newNode.setNextNode(nodeAfter);
         nodeBefore.setNextNode(newNode);
      } // end if
      numberOfEntries++;
   }
   else
      throw new IndexOutOfBoundsException(
                "Illegal position given to add operation.");
} // end add
```

Method `isEmpty`

```
public boolean isEmpty()
{
   boolean result;
   if (numberOfEntries == 0) // Or getLength() == 0
   {
      assert firstNode == null;
      result = true;
   }
   else
   {
      assert firstNode != null;
      result = false;
   } // end if

   return result;
} // end isEmpty
```

Note use of assert statement.

Method `toArray`

```
public T[] toArray()
{
   // The cast is safe because the new array contains null entries
   @SuppressWarnings("unchecked")
   T[] result = (T[])new Object[numberOfEntries];

   int index = 0;
   Node currentNode = firstNode;
   while ((index < numberOfEntries) && (currentNode != null))
   {
      result[index] = currentNode.getData();
      currentNode = currentNode.getNextNode();
      index++;
   } // end while

   return result;
} // end toArray
```


Traverses chain, loads an array.

Testing Core Methods

```
public static void main(String[] args)
{
   System.out.println("Create an empty list.");
   ListInterface<String> myList = new LList<>();
   System.out.println("List should be empty; isEmpty returns " +
                      myList.isEmpty() + ".");
   System.out.println("\nTesting add to end:");
   myList.add("15");
   myList.add("25");
   myList.add("35");
   myList.add("45");
   System.out.println("List should contain 15 25 35 45.");
   displayList(myList);
   System.out.println("List should not be empty; isEmpty() returns " +
                      myList.isEmpty() + ".");
   System.out.println("\nTesting clear():");
   myList.clear();
```

LISTING 14-2 A `main` method that tests part of the implementation of the ADT list

Testing Core Methods

```
      System.out.println("List should be empty; isEmpty returns " +
                         myList.isEmpty() + ".");
} // end main
```

Output

```
Create an empty list.
List should be empty; isEmpty returns true.

Testing add to end:
List should contain 15 25 35 45.
List contains 4 entries, as follows:
15 25 35 45
List should not be empty; isEmpty() returns false.

Testing clear():
List should be empty; isEmpty returns true.
```

LISTING 14-2 A `main` method that tests part of the implementation of the ADT list

Continuing the Implementation

```
public T getEntry(int givenPosition)
{
   if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
   {
      assert !isEmpty();
      return getNodeAt(givenPosition).getData();
   }
   else
   throw new IndexOutOfBoundsException(
              "Illegal position given to getEntry operation.");
} // end getEntry
```

Retrieving a list entry is straightforward.

Continuing the Implementation

```
public boolean contains(T anEntry)
{
   boolean found = false;
   Node currentNode = firstNode;

   while (!found && (currentNode != null))
   {
      if (anEntry.equals(currentNode.getData()))
         found = true;
      else
         currentNode = currentNode.getNextNode();
   } // end while

   return found;
} // end contains
```

Checking to see if an entry is in the list, the method `contains`.

Design Decision
A Link to Last Node

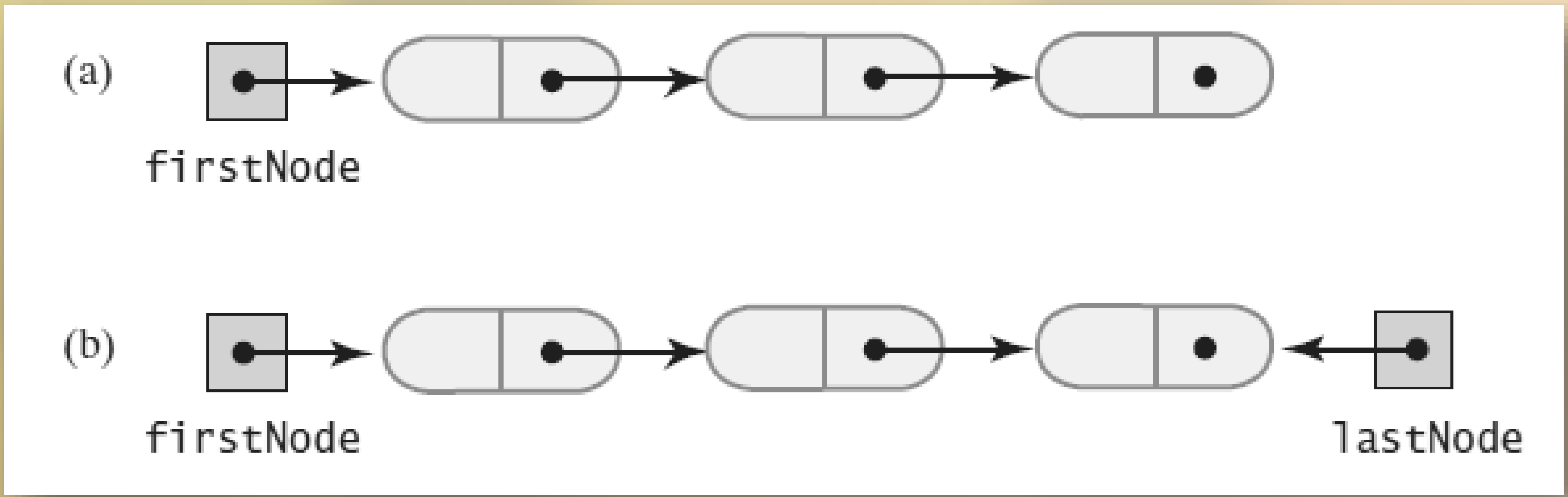


FIGURE 14-7 A linked chain with (a) a head reference; (b) both a head reference and a tail reference

A Refined Implementation

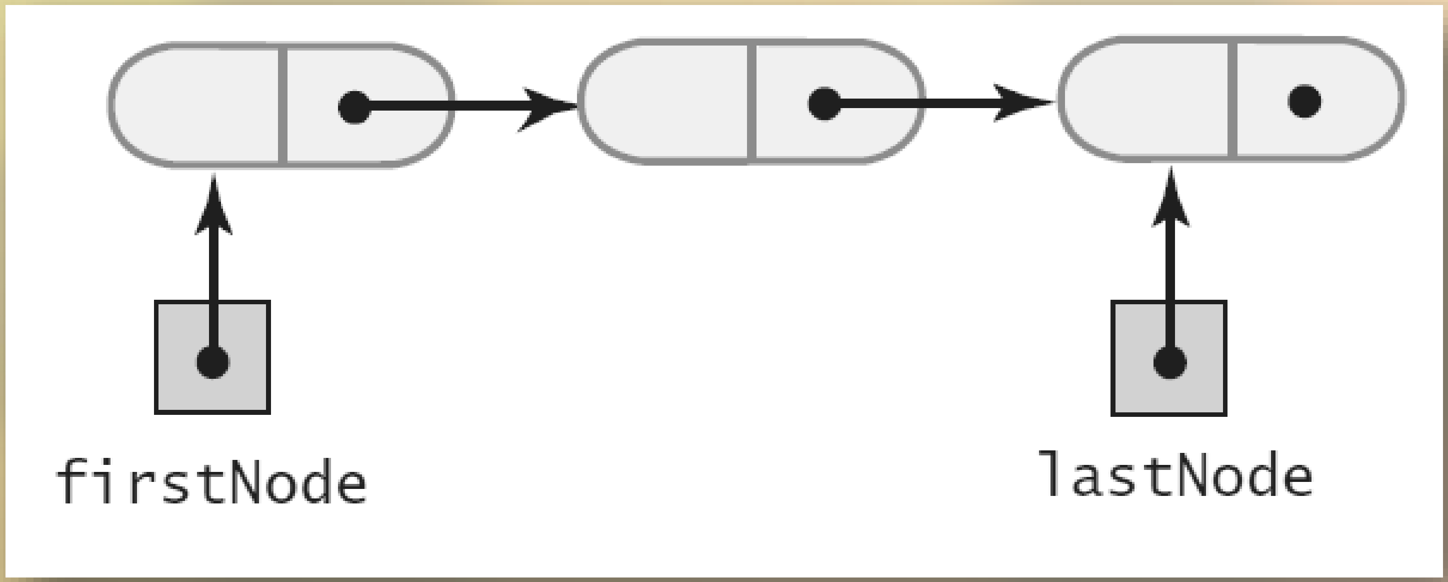


FIGURE 14-8 A linked chain with both a head reference and a tail reference

A Refined Implementation

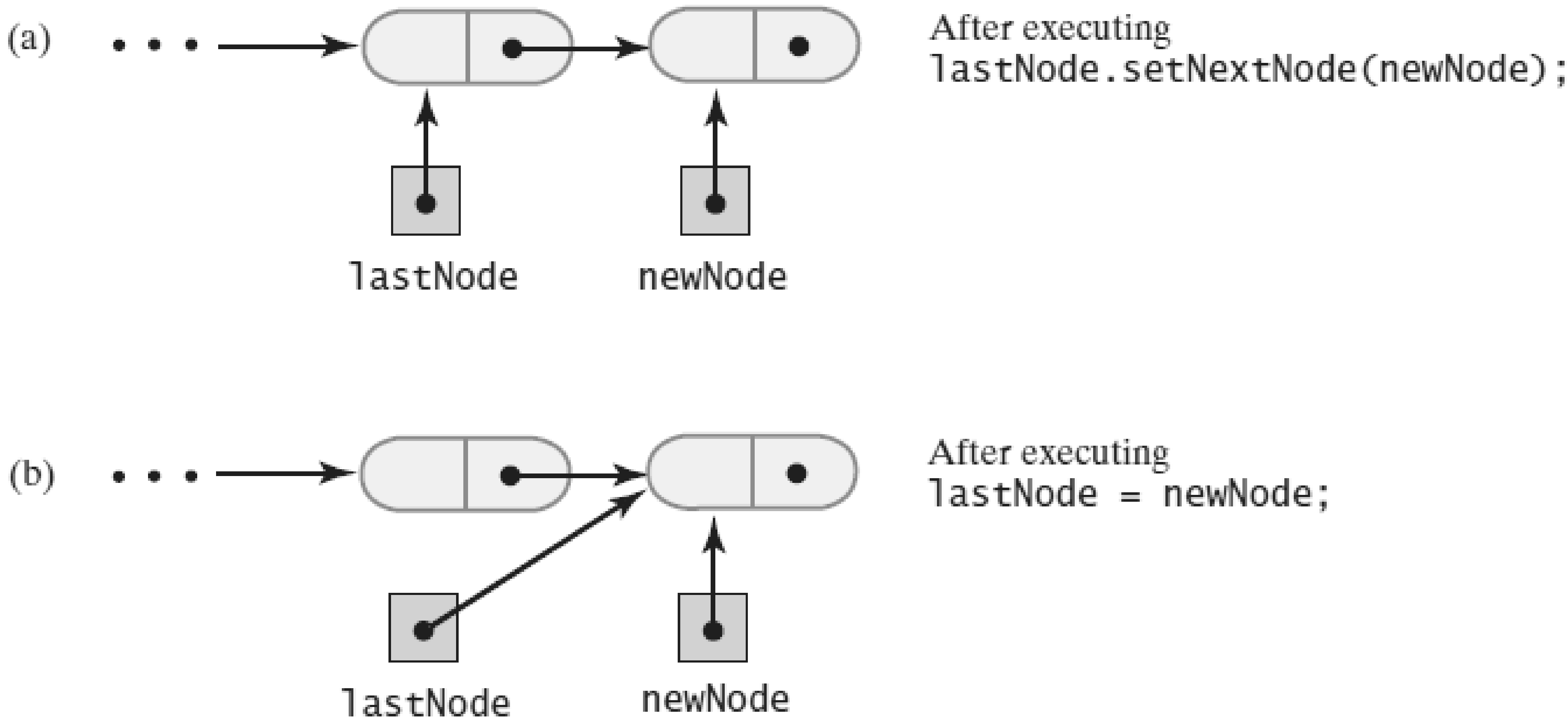


FIGURE 14-9 Adding a node to the end of a nonempty chain that has a tail reference

A Refined Implementation

```
public void add(T newEntry)
{
    Node newNode = new Node(newEntry);

    if (isEmpty())
        firstNode = newNode;
    else
        lastNode.setNextNode(newNode);

    lastNode = newNode;

    numberOfEntries++;
} // end add
```


Revision of the first `add` method

A Refined Implementation

```
public void add(int newPosition, T newEntry)
{
if ((newPosition >= 1) && (newPosition <= numberOfEntries + 1))
{
     Node newNode = new Node(newEntry);
     if (isEmpty())
     {
        firstNode = newNode;
        lastNode = newNode;
     }
     else if (newPosition == 1)
     {
        newNode.setNextNode(firstNode);
        firstNode = newNode;
     }
     else if (newPosition == numberOfEntries + 1)
```


Implementation of the method that adds by position.

```
         newNode.setNextNode(firstNode);
         firstNode = newNode;
      }
      else if (newPosition == numberOfEntries + 1)
      {
         lastNode.setNextNode(newNode);
         lastNode = newNode;
      }
      else
      {
         Node nodeBefore = getNodeAt(newPosition - 1);
         Node nodeAfter = nodeBefore.getNextNode();
         newNode.setNextNode(nodeAfter);
         nodeBefore.setNextNode(newNode);
      } // end if

      numberOfEntries++;
   }
   else
      throw new IndexOutOfBoundsException(
                  "Illegal position given to add operation.");
} // end add
```


Implementation of the method that adds by position.

A Refined Implementation

FIGURE 14-10 Removing the last node from a chain that has both head and tail references when the chain contains (a) one node

A Refined Implementation

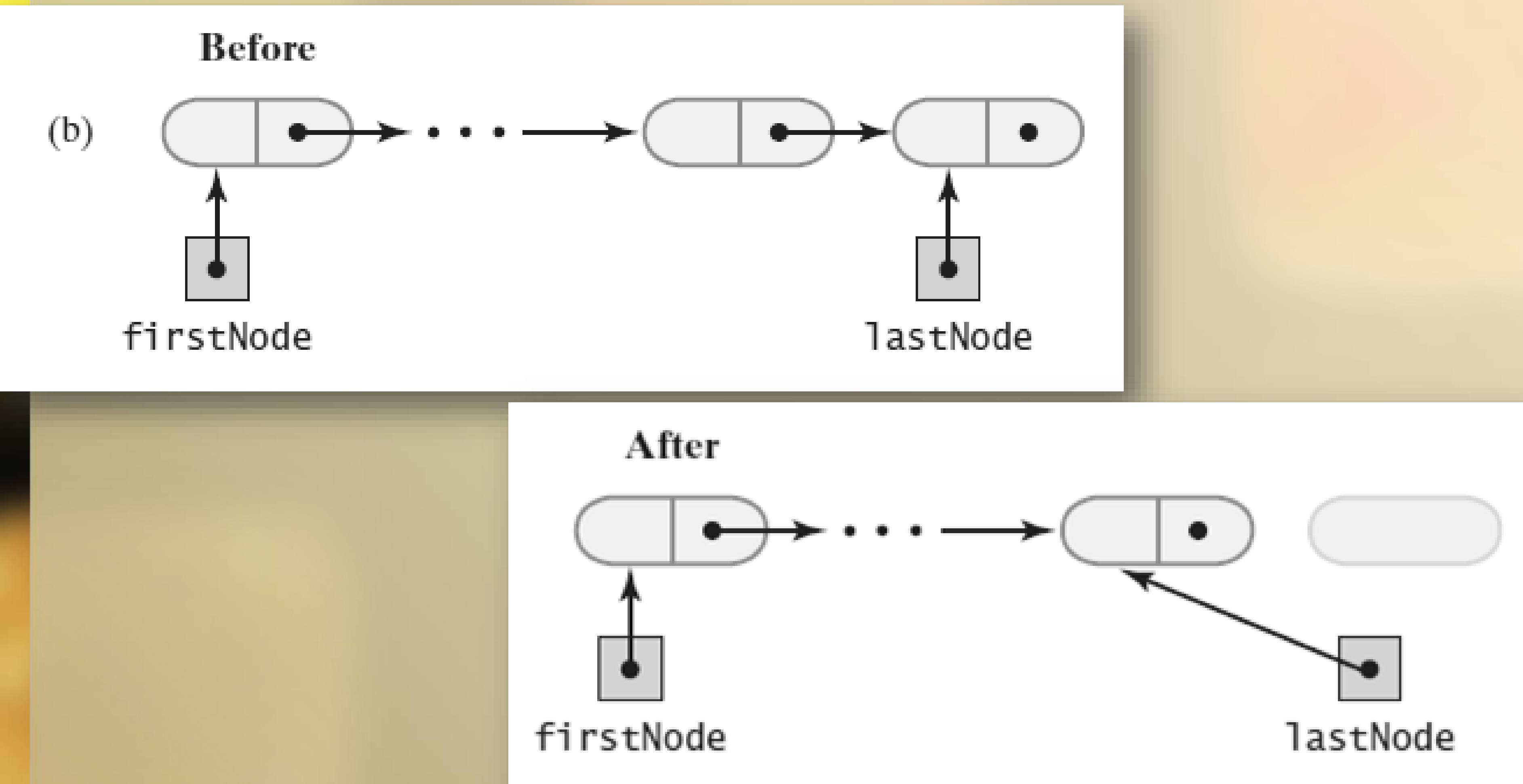


FIGURE 14-10 Removing the last node from a chain that has both head and tail references when the chain contains (b) more than one node

A Refined Implementation

```
public T remove(int givenPosition)
{
   T result = null;                              // Return value

   if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
   {
      assert !isEmpty();
      if (givenPosition == 1)                    // Case 1: Remove first entry
      {
         result = firstNode.getData();           // Save entry to be removed
         firstNode = firstNode.getNextNode();
         if (numberOfEntries == 1)
            lastNode = null;                     // Solitary entry was removed
      }
      else                                       // Case 2: Not first entry
      {
         Node nodeBefore = getNodeAt(givenPosition - 1);
         Node nodeToRemove = nodeBefore.getNextNode();
```

Implementation of the remove operation:

A Refined Implementation

```
         Node nodeBefore = getNodeAt(givenPosition - 1);
         Node nodeToRemove = nodeBefore.getNextNode();
         Node nodeAfter = nodeToRemove.getNextNode();
         nodeBefore.setNextNode(nodeAfter);
         result = nodeToRemove.getData();     // Save entry to be removed
         if (givenPosition == numberOfEntries)
            lastNode = nodeBefore;            // Last node was removed
      } // end if
      numberOfEntries--;
   }
   else
      throw new IndexOutOfBoundsException(
               "Illegal position given to remove operation.");
   return result;                             // Return removed entry
} // end remove
```

Implementation of the remove operation:

Java Class Library: The Class `LinkedList`

- Implements the interface `List`
- `LinkedList` defines more methods than are in the interface `List`
- You can use the class `LinkedList` as implementation of ADT
  - queue
  - deque
  - or list.

End

Chapter 14