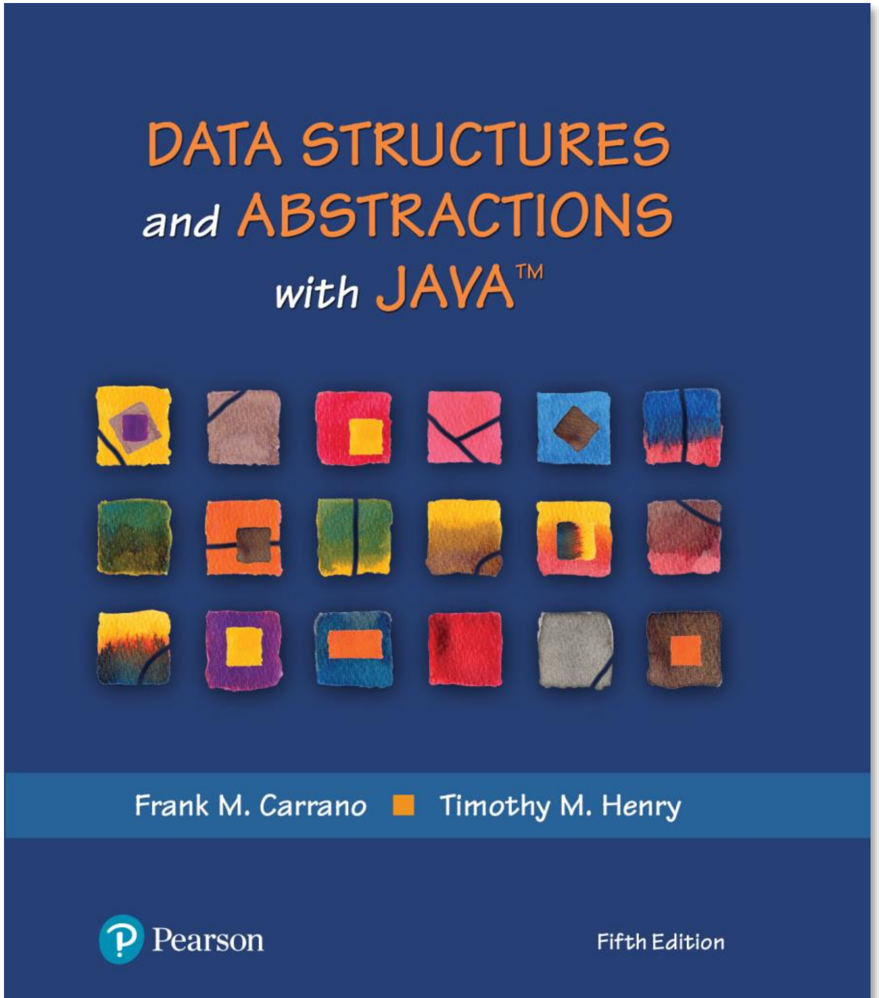
Data Structures and Abstractions with Java™

5th Edition



Chapter 12

A List Implementation That Links Data

Pearson

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

Adding a Node at Various Positions

- Possible cases:
  - Chain is empty
  - Adding node at chain’s beginning
  - Adding node between adjacent nodes
  - Adding node to chain’s end

Adding a Node

This pseudocode establishes a new node for the given data

newNode *references a new instance of* Node

Place newEntry *in* newNode

firstNode = *address of* newNode

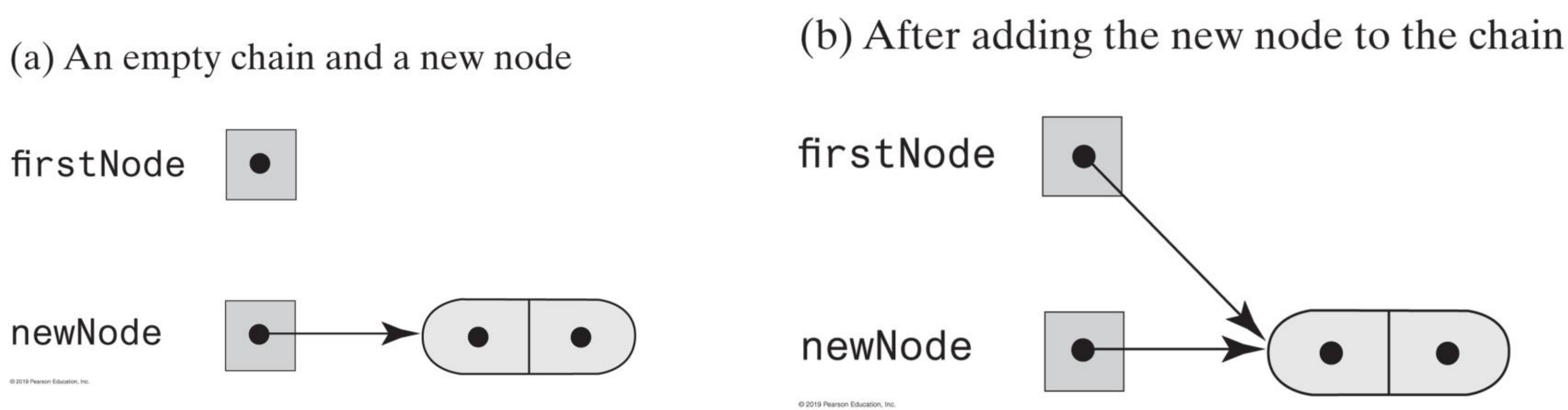


FIGURE 12-1 Adding a node to an empty chain

Adding a Node

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

newNode *references a new instance of* Node

Place newEntry *in* newNode

Set newNode*'s link to* firstNode

Set firstNode *to* newNode

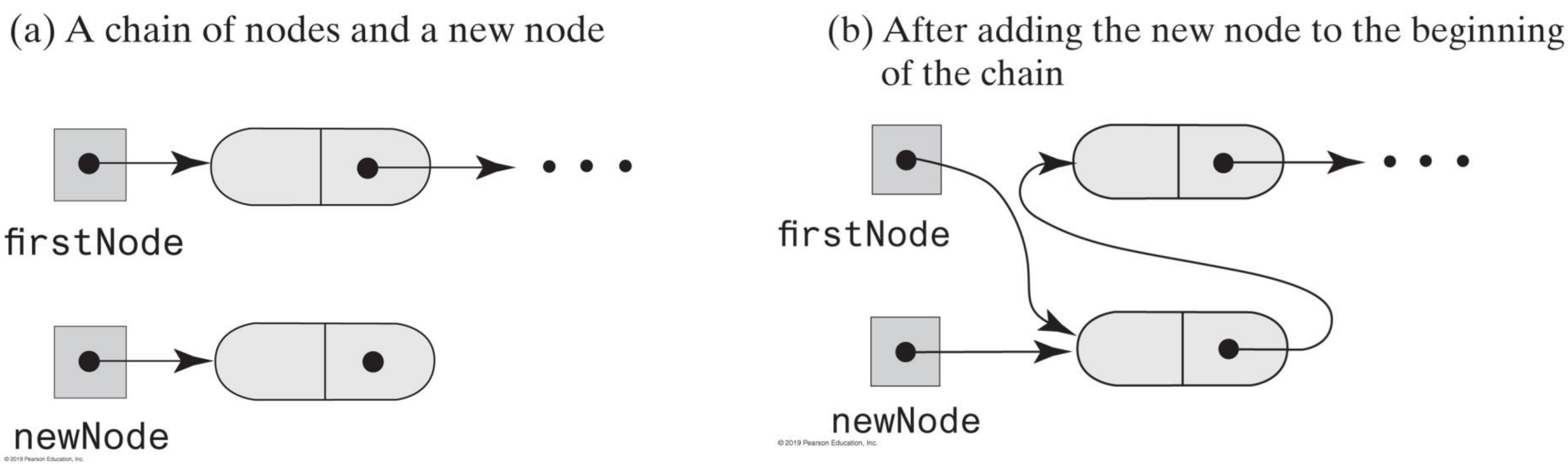


FIGURE 12-2 Adding a node to the beginning of a chain

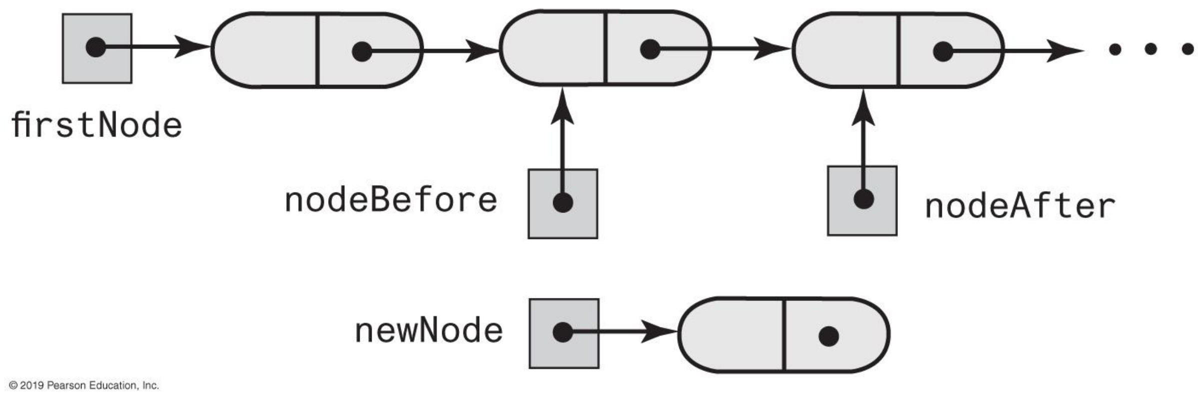
Adding a Node

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

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

Adding a Node

(a) A chain of nodes and a new node



(b) After adding the new node between adjacent nodes

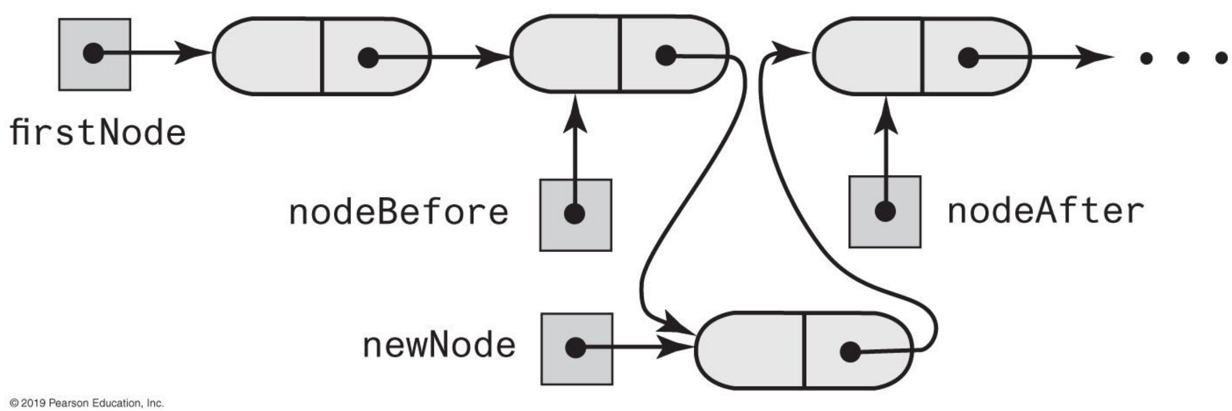


FIGURE 12-3 Adding a node between two adjacent nodes

Adding a Node

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

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*

Adding a Node

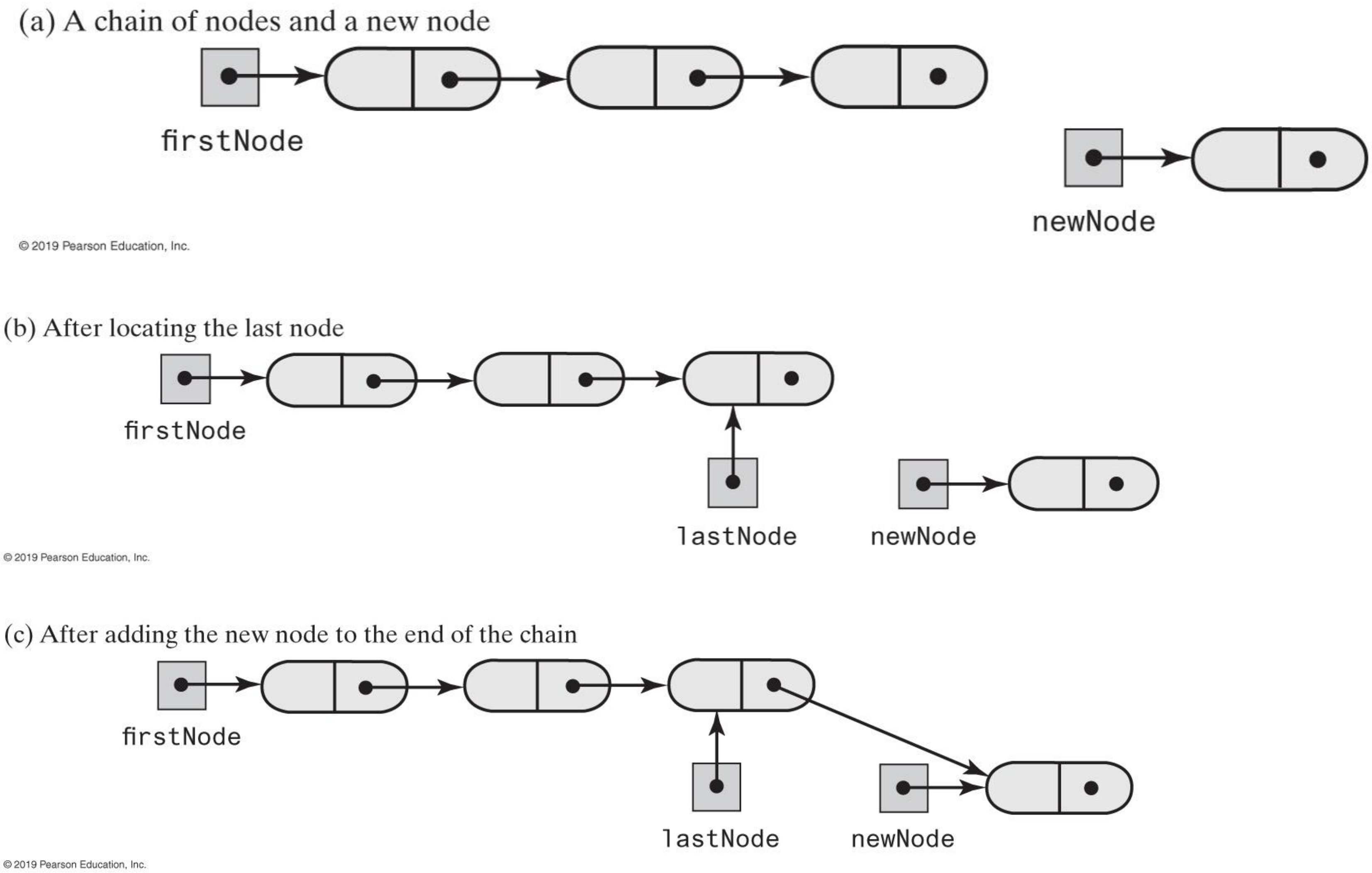


FIGURE 12-4 Adding a node to the end of a chain

Removing a Node

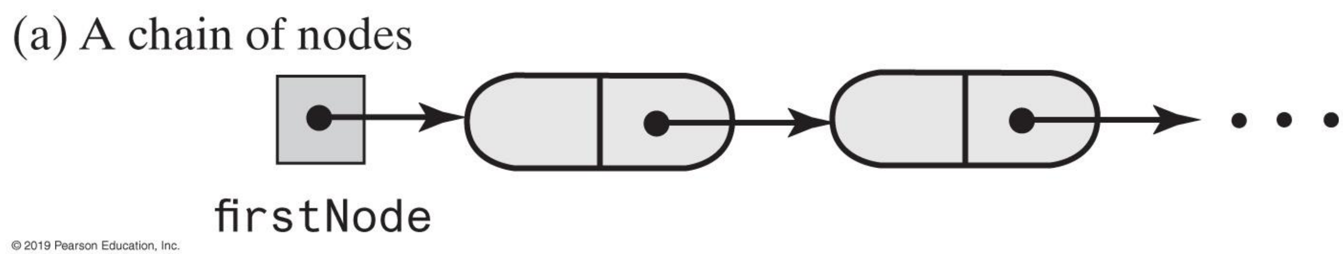
- Possible cases
  - Removing the first node
  - Removing a node other than first one

Removing a Node

Steps for removing the first node.

Set firstNode *to the link in the first node;* firstNode *now either references the second node or is* null *if the chain had only one node.*

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



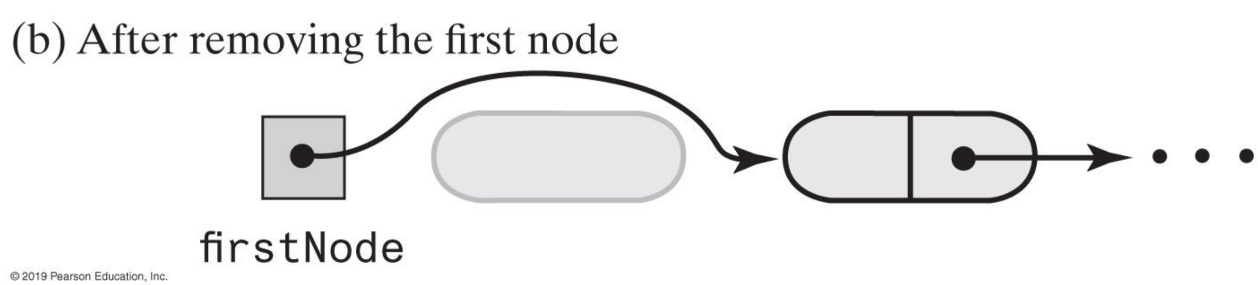


FIGURE 12-5 Removing the first node from a chain

Removing a Node

Removing a node other than the first one.

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 either references the node after the one to be removed or is* null.

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

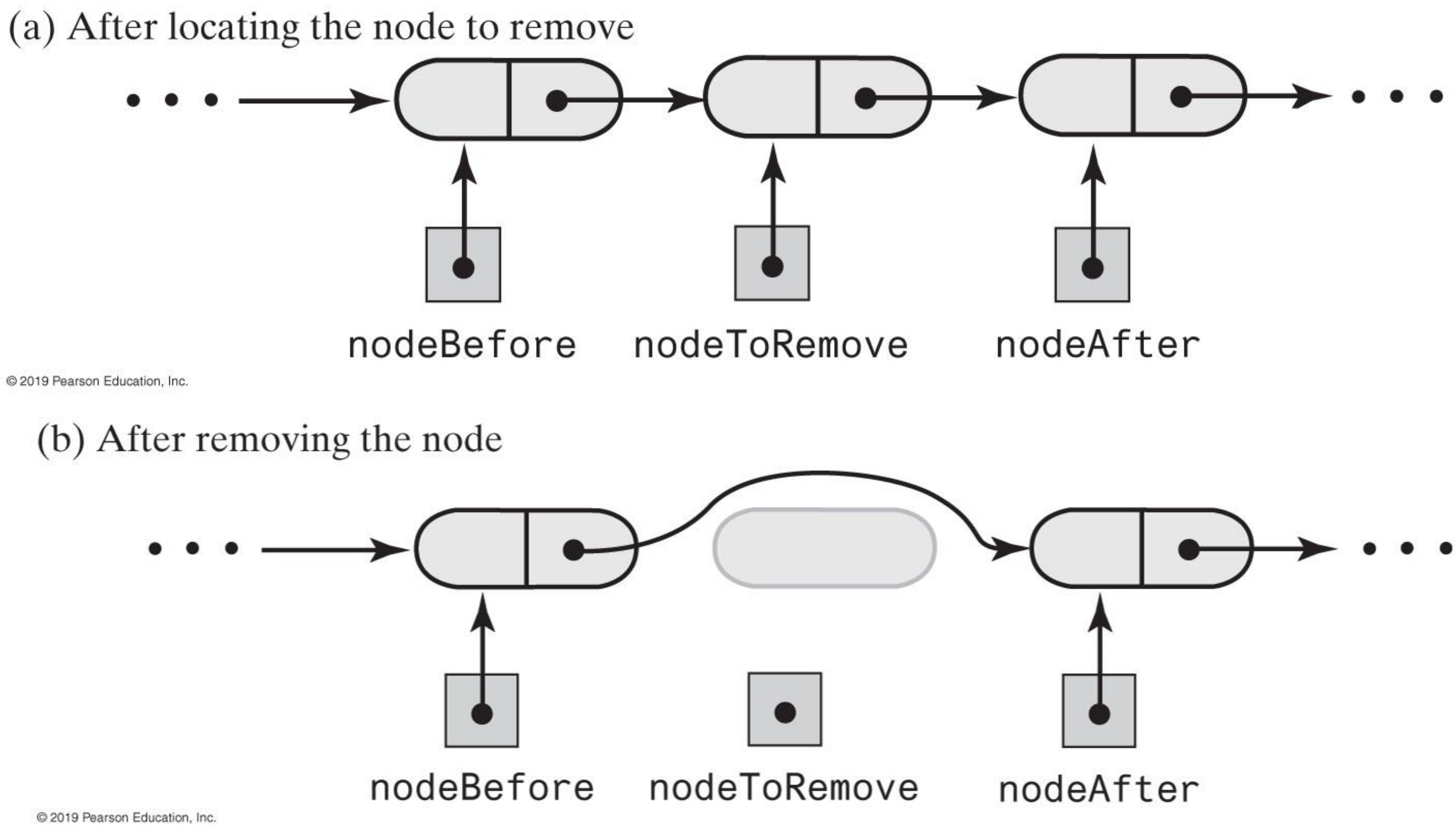


FIGURE 12-6 Removing an interior node from a chain

Removing a Node

```
// Returns a reference to the node at a given position.
// Precondition: The chain is not empty;
//             1 <= givenPosition <= numberOfEntries.
private Node getNodeAt(int givenPosition)
{
   // Assertion: (firstNode != null) &&
   //          (1 <= givenPosition) && (givenPosition <= numberOfEntries)
   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();
   // Assertion: currentNode != null
   return currentNode;
} // end getNodeAt
```

forloop alternatives

while(currentNode != null){

if(givent position equals counter)

break

currentNode = currentNode.getNextNode()

counter++;

}

Operations on a chain depended on the method `getNodeAt`

Using a Tail Reference

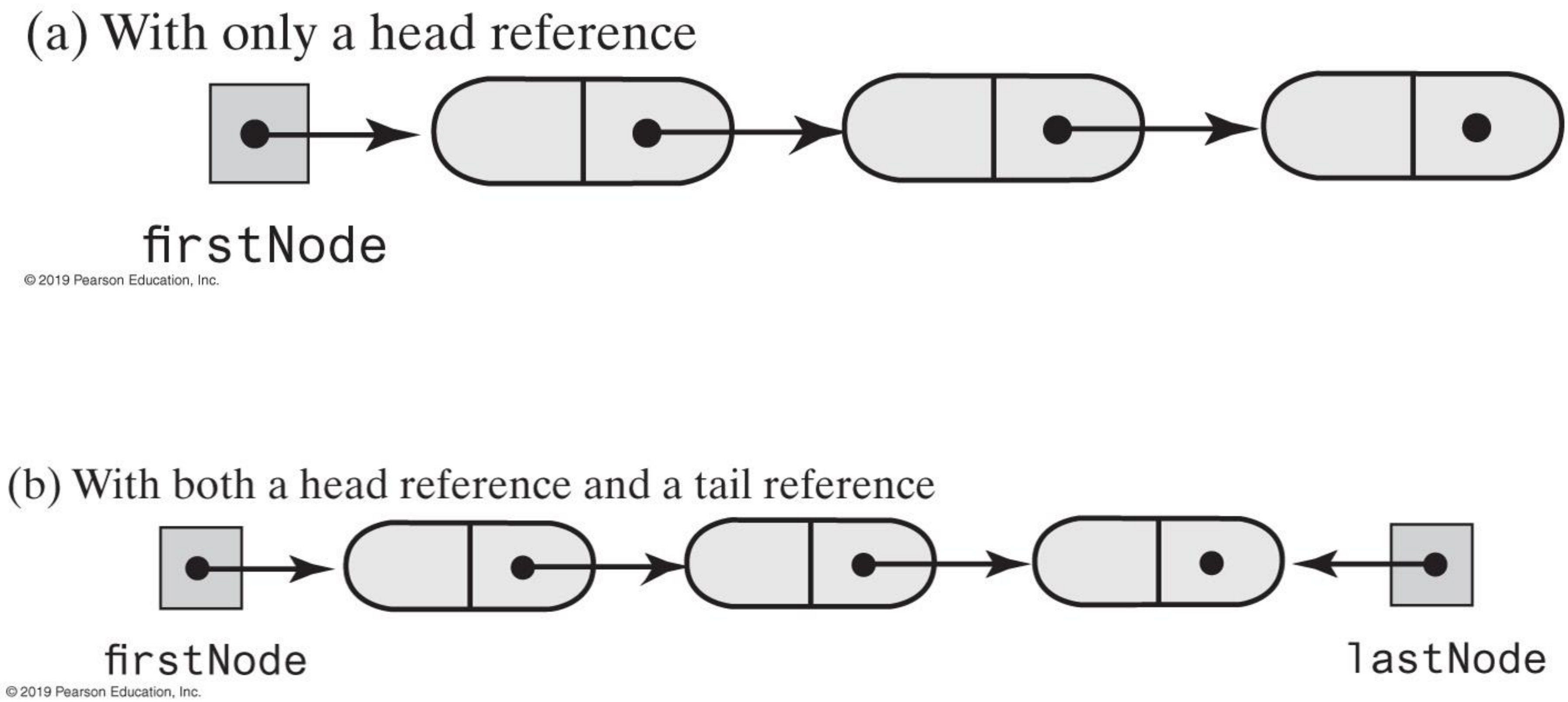


FIGURE 12-7 Two linked chains

Data Fields and Constructor (Part 1)

```
/** A linked implemention of the ADT list. */
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. >
  . . . */

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

LISTING 12-1 An outline of the class `LList`

Data Fields and Constructor (Part 2)

```
// Returns a reference to the node at a given position.
// Precondition: The chain is not empty;
//            1 <= givenPosition <= numberOfEntries.
private Node getNodeAt(int givenPosition)
{
  // Assertion: (firstNode != null) &&
  //          (1 <= givenPosition) && (givenPosition <= numberOfEntries)
  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();
  // Assertion: currentNode != null
  return currentNode;
} // end getNodeAt

    private class Node
    {
          // < Implementation of private inner class Node >
    } // end Node

} // end LList
```

LISTING 12-1 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 givenPosition, T newEntry)
{
  if ((givenPosition >= 1) && (givenPosition <= numberOfEntries + 1))
  {
    Node newNode = new Node(newEntry);
    if (givenPosition == 1)              // Case 1
    {
      newNode.setNextNode(firstNode);
      firstNode = newNode;
    }
    else                                                   // Case 2: list is not empty
    {                               // and givenPosition > 1
      Node nodeBefore = getNodeAt(givenPosition - 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
```

add method.

Method `isEmpty`

```
public boolean isEmpty()
{
  boolean result;

  if (numberOfEntries == 0) // Or getLength() == 0
  {
    // Assertion: firstNode == null
    result = true;
  }
  else
  {
    // Assertion: firstNode != null
    result = false;
  } // end if

  return result;
} // end isEmpty
```

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

LISTING 12-2 A main method that tests part of the implementation of the ADT list

Testing Core Methods

Program Output

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

Testing add to end:
List should contain 15 25 35 45.
The 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 12-2 A main method that tests part of the implementation of the ADT list

`remove` method returns entry it deletes from list

```
public T remove(int givenPosition)
{
  T result = null;                          // Return value
  if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
  {
    // Assertion: !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
    {
      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
```

Continuing the Implementation

```
public T replace(int givenPosition, T newEntry)
{
  if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
  {
    // Assertion: !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.

Continuing the Implementation

```
public T getEntry(int givenPosition)
{
   if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
   {
      // Assertion: !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`.

A Refined Linked Implementation

```
private Node firstNode;      // Head reference to first node
private Node lastNode;       // Tail reference to last node
private int  numberOfEntries; // Number of entries in list
```

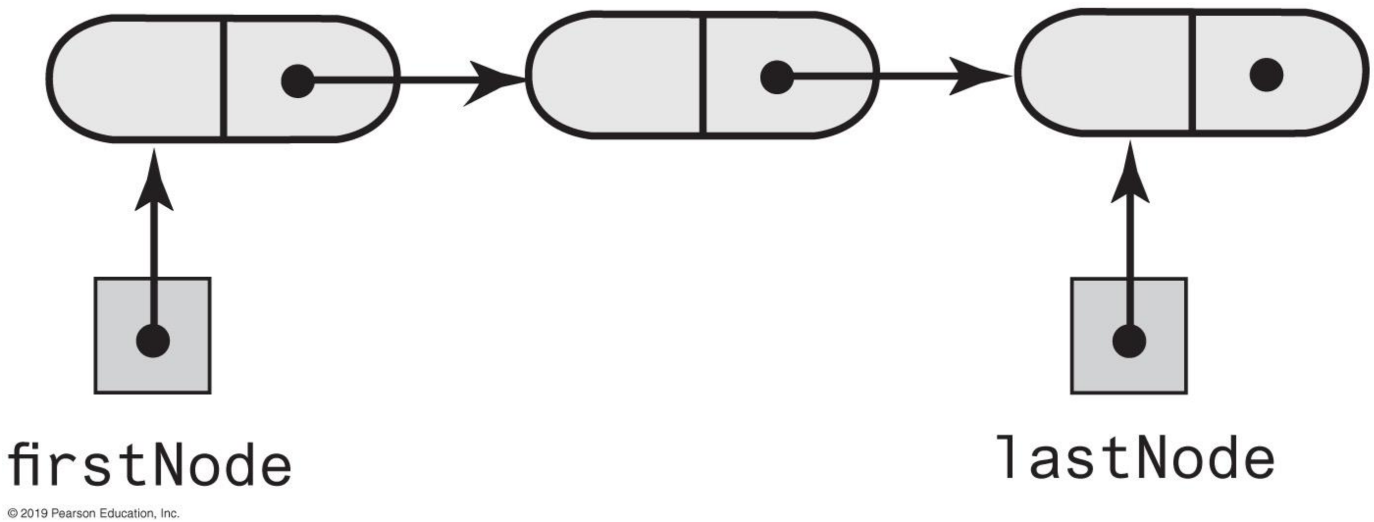
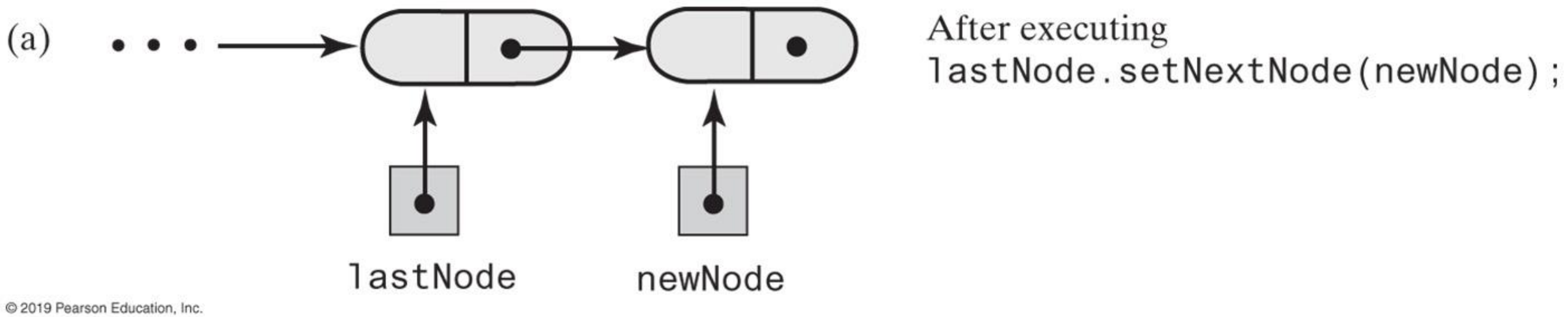


FIGURE 12-8 A linked chain with both a head reference and a tail reference

A Refined Linked Implementation



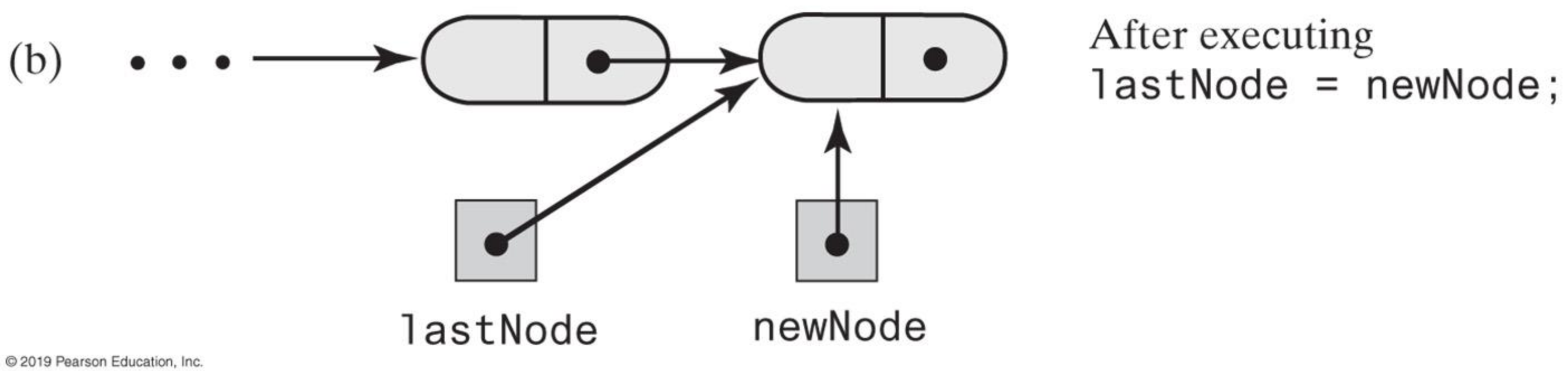


FIGURE 12-9 Adding a node to the end of a nonempty chain that has a tail reference

A Refined Linked 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 Linked Implementation - refined add by position

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

A Refined Linked Implementation

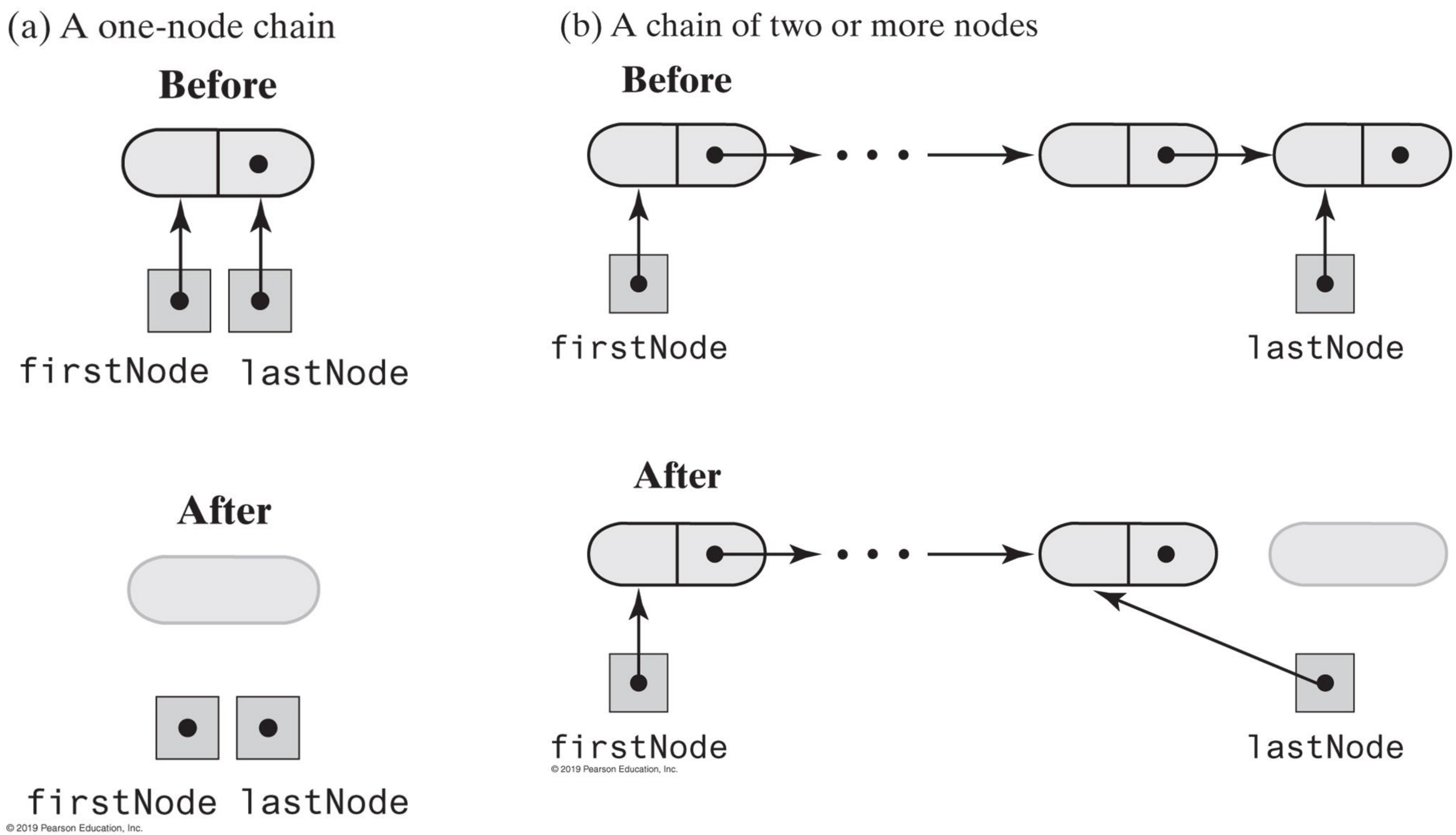


FIGURE 12-10 Before and after removing the last node from a chain that has both head and tail references and contains one or more nodes

A Refined Linked Implementation — refined `remove`

```
public T remove(int givenPosition) {
  T result = null;                  // Return value
  if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
  {
  // Assertion: !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();
      Node nodeAfter = nodeToRemove.getNextNode();
      nodeBefore.setNextNode(nodeAfter);
      result = nodeToRemove.getData();
      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
```

Efficiency of Using a Chain

best; average; worst

Operation	Alist	LList	LListWithTail
`add(newEntry)`	O(1)	O(*n*)	O(1)
`add(givenPosition, newEntry)`	O(*n*); O(n); O(1)	O(1); O(*n*)	O(1); O(*n*); O(1)
`toArray()`	O(*n*)	O(*n*)	O(*n*)
`remove(givenPosition)`	O(*n*); O(*n*); O(1)	O(1); O(*n*)	O(1); O(*n*)
`replace(givenPosition, newEntry)`	O(1)	O(1); O(*n*)	O(1); O(*n*); O(1)
`getEntry(givenPosition)`	O(1)	O(1); O(*n*)	O(1); O(*n*); O(1)
`contains(anEntry)`	O(*n*)	O(*n*)	O(*n*)
`clear(), getLength(), isEmpty()`	O(1)	O(1)	O(1)

FIGURE 12-11 The time efficiencies of the ADT list operations for three implementations, expressed in Big Oh notation

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 12