A List Implementation That Links Data

Chapter 14

Contents

- Operations on a Chain of Linked Nodes
  - Adding a Node at Various Positions
  - Removing a Node from Various Positions
  - The Private Method **`getNodeAt`**

    …

Contents

- Beginning the Implementation
  - The Data Fields and Constructor
  - Adding to the End of the List
  - Adding at a Given Position Within the List
  - The Methods **`isEmpty`** and **`toArray`**
  - Testing the Core Methods

    …

Contents

- Continuing the Implementation
- A Refined Implementation
  - The Tail Reference
- The Efficiency of Using a Chain to Implement the ADT List
- Java Class Library: The Class **`LinkedList`**

Objectives

- Describe linked organization of data
- Implement **`add`** methods of the ADT list by using linked chain of nodes
- Test partially complete implementation of a class

Operations on a Chain of Linked Nodes

- Adding a Node at Various Positions
  - Case 1: Chain is empty
  - Case 2: Adding node at chain's beginning
  - Case 3: Adding node between adjacent nodes
  - Case 4: Adding node to chain's end

Case 1

```
Node newNode = new Node(newEntry);
firstNode = newNode;
```

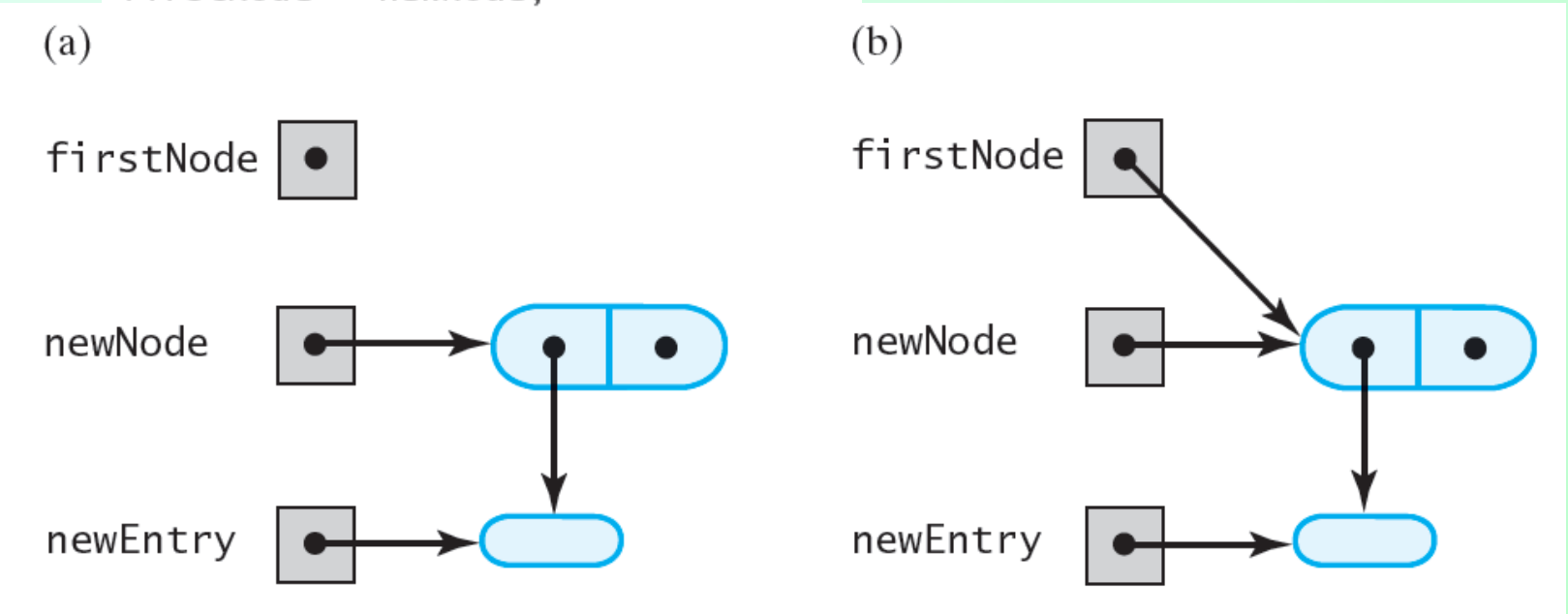


Figure 14-1 (a) An empty chain and a new node;
(b) after adding the new node to a chain that was empty

Case 2

```
Node newNode = new Node(newEntry);
newNode.setNextNode(firstNode);
firstNode = newNode;
```

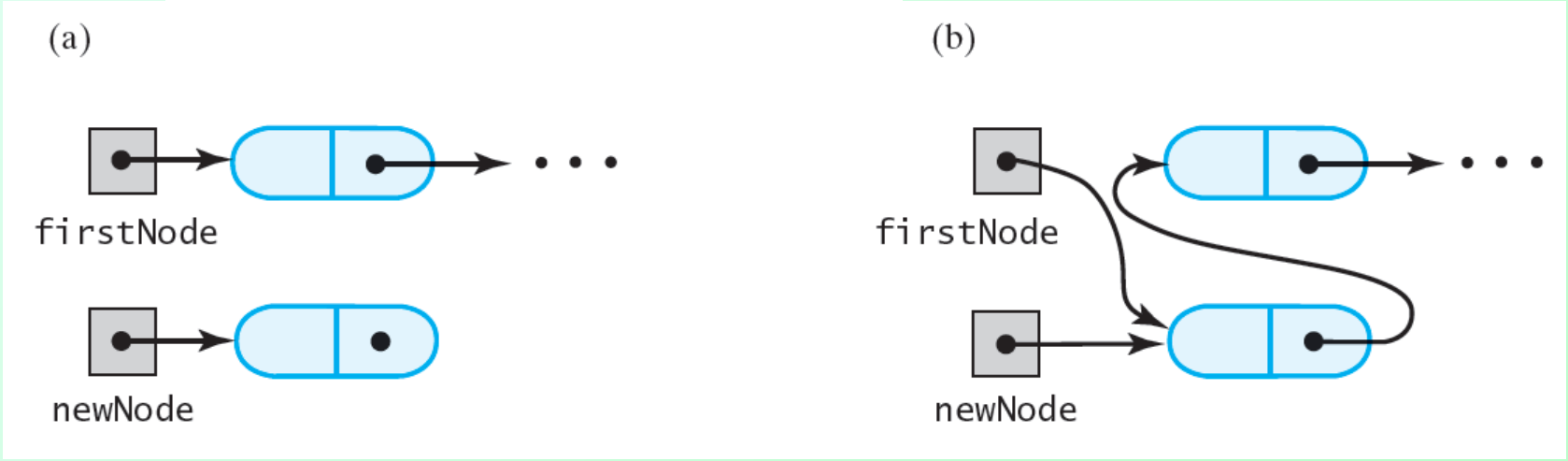
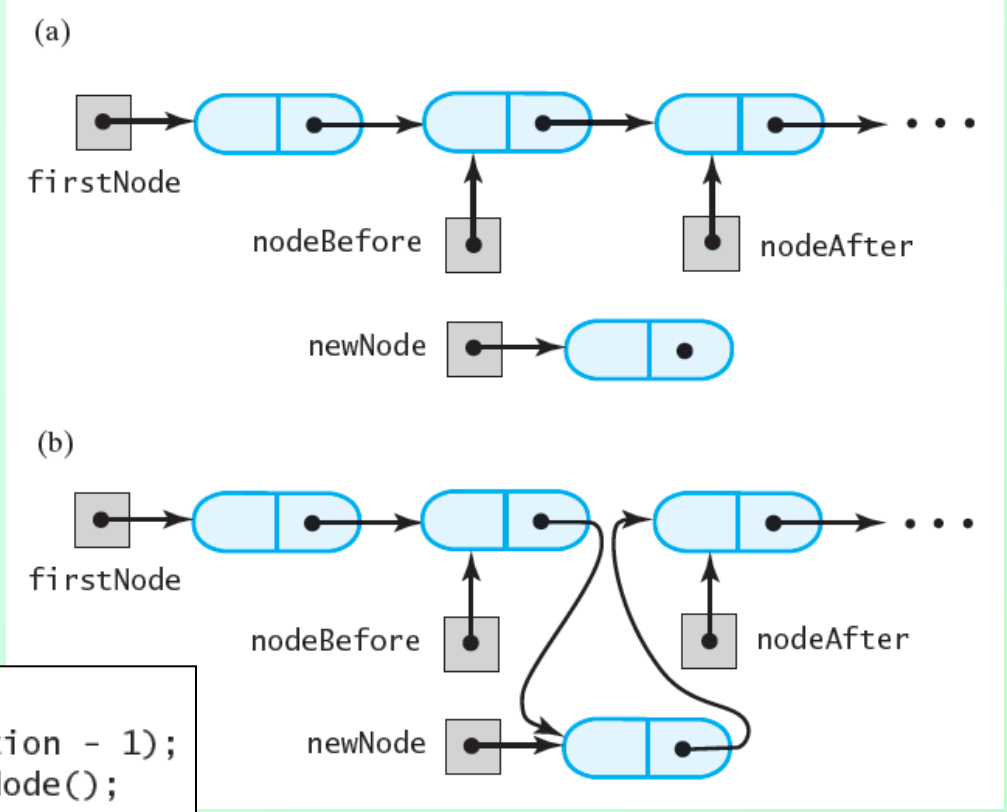


Figure 14-2 A chain of nodes (a) just prior to adding a node at the beginning; (b) just after adding a node at the beginning

Case 3




```
Node newNode = new Node(newEntry);
Node nodeBefore = getNodeAt(newPosition - 1);
Node nodeAfter = nodeBefore.getNextNode();
newNode.setNextNode(nodeAfter);
nodeBefore.setNextNode(newNode);
```

Figure 14-3 A chain of nodes (a) just prior to adding a node between two adjacent nodes; (b) just after adding a node between two adjacent nodes

Case 4

```
Node newNode = new Node(newEntry);
Node lastNode = getNodeAt(numberOfNodes);
lastNode.setNextNode(newNode);
```

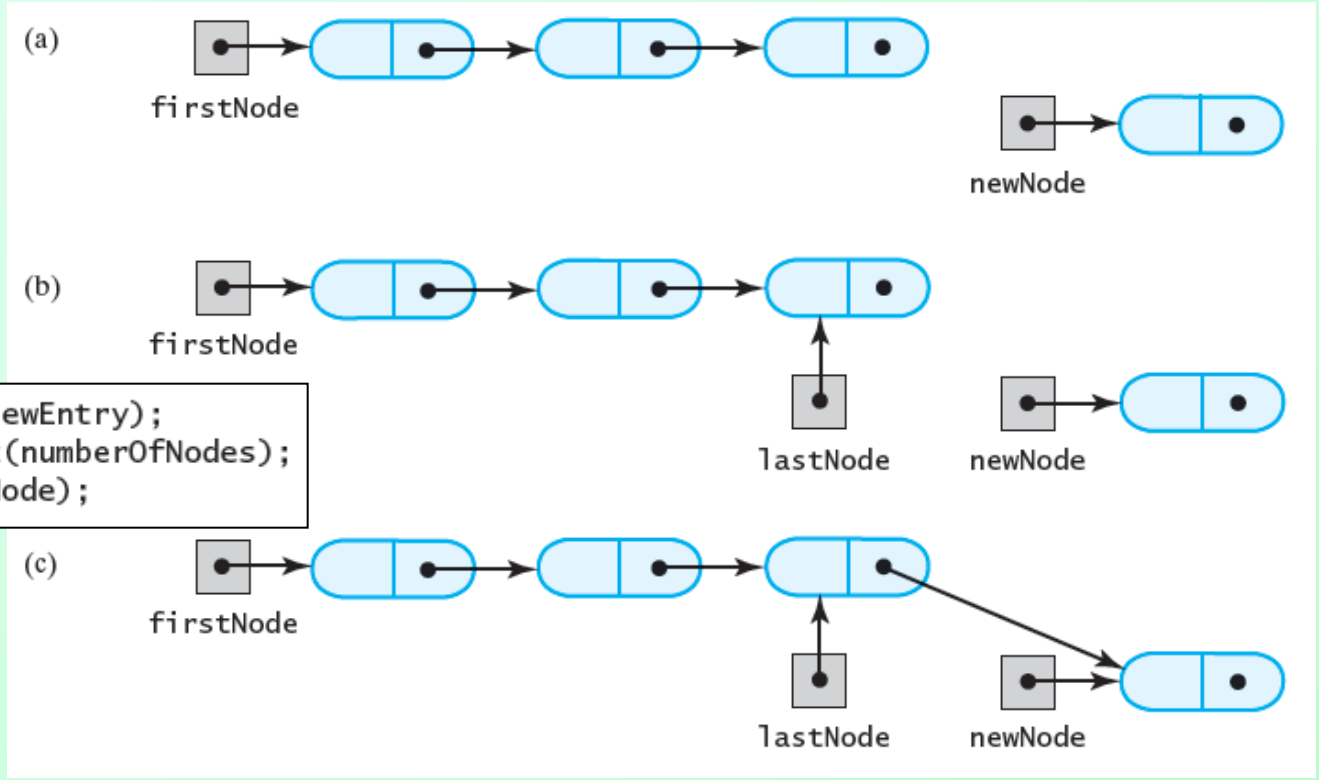


FIGURE 14-4 A chain of nodes
(a) prior to adding a node at the end;
(b) after locating its last node;
(c) after adding a node at the end

Question 1 Describe the steps that the method getNodeAt must take to locate the node at a given position.

Question 1 Describe the steps that the method getNodeAt must take to locate the node at a given position.

To locate the nth node in a chain, getNodeAt starts at the first node and counts nodes as it traverses the chain from node to node, until it reaches the nth one. The following pseudocode describes the steps in more detail:

```
currentNode = firstNode
for (counter = 1  to  n)
        currentNode = currentNode.getNextNode()
```

The desired node is at currentNode

Removing a Node from Various Positions

- Case 1: Removing first node

```
firstNode = firstNode.getNextNode();
```

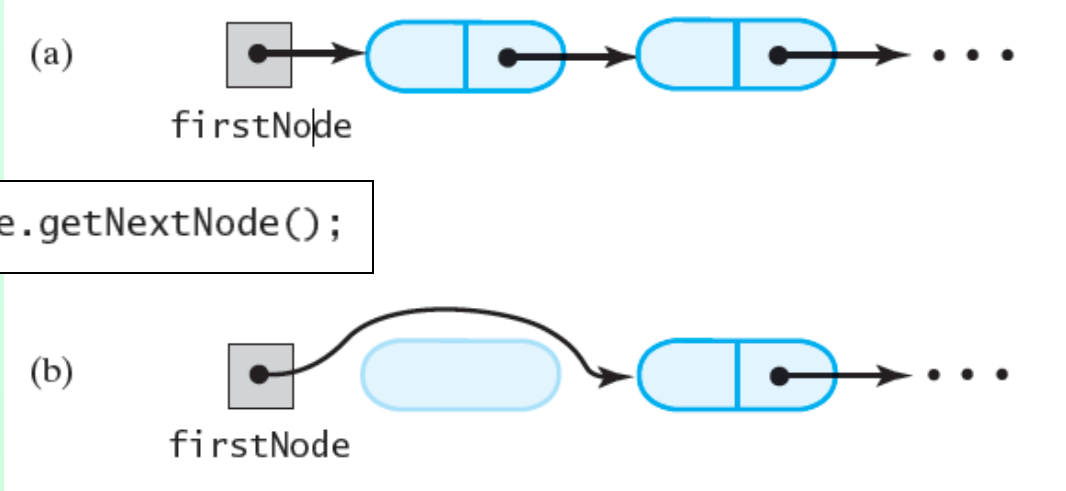


FIGURE 14-5 A chain of nodes (a) just prior to removing the first node; (b) just after removing the first node

Removing a Node from Various Positions

- Case 2: Removing node other than first

```
Node nodeBefore = getNodeAt(givenPosition - 1);
Node nodeToRemove = nodeBefore.getNextNode();
Node nodeAfter = nodeToRemove.getNextNode();
nodeBefore.setNextNode(nodeAfter);
nodeToRemove = null;
```

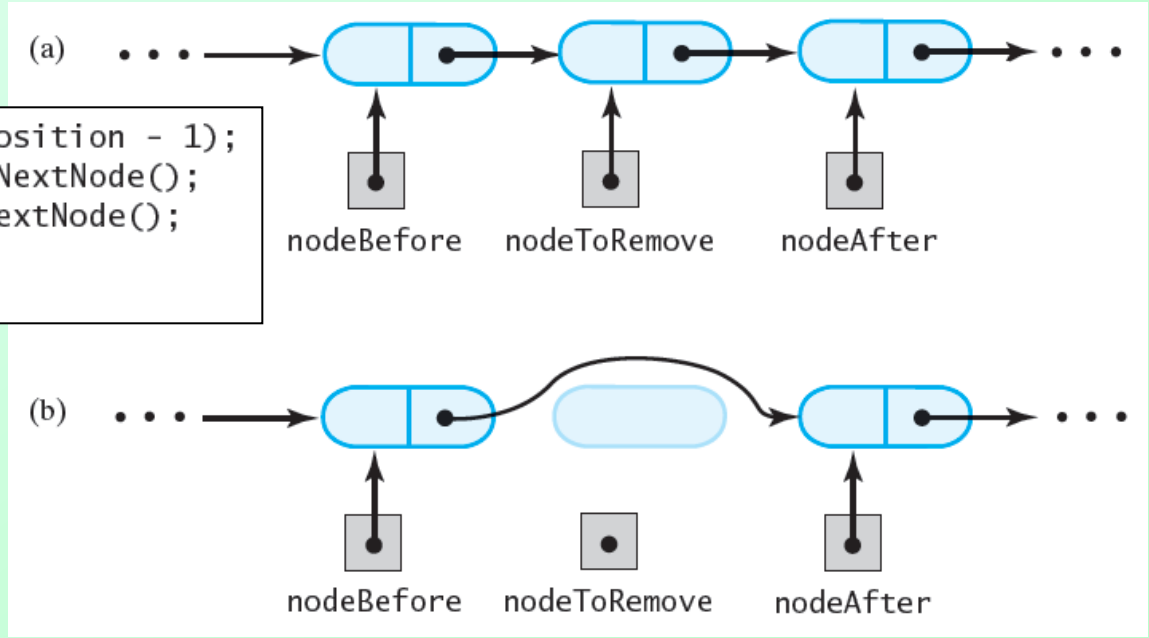


FIGURE 14-6 A chain of nodes (a) just prior to removing an interior node; (b) just after removing an interior node

Question 2 The code that we developed in Segment 14.3 to add a node between two adjacent nodes of a chain is

```
Node newNode = new  Node(newEntry);
Node nodeBefore = getNodeAt(newPosition - 1);
Node nodeAfter = nodeBefore.getNextNode();
newNode.setNextNode(nodeAfter);
nodeBefore.setNextNode(newNode);
```

Is it possible to use this code instead of the following code, which we just developed, to add a node to the end of a chai n? Explain your answer.

```
Node newNode = new  Node(newEntry);
Node lastNode = getNodeAt(numberOfNodes);
lastNode.setNextNode(newNode);
```

Question 2 The code that we developed in Segment 14.3 to add a node between two adja-cent nodes of a chain is

```
Node newNode = new  Node(newEntry);
Node nodeBefore = getNodeAt(newPosition - 1);
Node nodeAfter = nodeBefore.getNextNode();
newNode.setNextNode(nodeAfter);
nodeBefore.setNextNode(newNode);
```

Is it possible to use this code instead of the following code, which we just developed, to add a node to the end of a chai n? Explain your answer.

```
Node newNode = new  Node(newEntry);
Node lastNode = getNodeAt(numberOfNodes);
lastNode.setNextNode(newNode);
```

Yes. With newPosition equal to numberOfNodes + 1, nodeBefore will reference the last n ode in the chain. Moreover, nodeAfter will be null, newNode ’s link field will be set to null, and the last node’s link will reference the new node.

Question 3 Adding a node to an empty chain could be thought of as adding a node to the end of a chain that is empty. Can you use the statements in Segment 14.4 instead of

```
Node newNode = new  Node(newEntry);
firstNode = newNode;
```

which we developed in Segment 14.1 to add a node to an empty chain? Why or why not?

Question 3 Adding a node to an empty chain could be thought of as adding a node to the end of a chain that is empty. Can you use the statements in Segment 14.4 instead of

```
Node newNode = new  Node(newEntry);
firstNode = newNode;
```

which we developed in Segment 14.1 to add a node to an empty chain? Why or why not?

No. The statements given in Segment 14.4 do not assign a new value to firstNode. Also, when the chain is empty, numberOfNodes is zero. The precondition of getNodeAt given in Segment 14.3 requires a positive argument. Even if you redesign getNodeAt, the empty chain will remain a special case.

The Private Method `getNodeAt`

- Returns reference to node at specified position

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

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

   assert currentNode != null;

   return currentNode;
} // end getNodeAt
```

Question 4The statements in Segment 14.4 that add an entry to the end of a chain invoke the method getNodeAt. Suppose that you use these statements repeatedly to create a chain by adding entries to its end.

a. How efficient of time is this approach?

b. Is there a faster way to repeatedly add entries to the end of a chain? Explain.

Question 5 How does getNodeAt's precondition prevent currentNode from becoming null?

Question 4 The statements in Segment 14.4 that add an entry to the end of a chain invoke the method getNodeAt. Suppose that you use these statements repeatedly to create a chain by adding entries to its end.

a. How efficient of time is this approach?

a. This approach is inefficient of time, since each addition causes getNodeAt to traverse the chain from its beginning until it locates the chain's last node. Thus, each addition depends on the length of the chain.

b. Is there a faster way to repeatedly add entries to the end of a chain? Explain.

b. Maintaining a tail reference would allow additions to the end of the chain to occur in O(1) time, that is, independently of the length of the chain.

Question 5 How does getNodeAt's precondition prevent currentNode from becoming null?

Since the chain is not empty, firstNode is not null. Thus, currentNode 's initial value is not null. The loop in getNodeAt can iterate no more than numberOfNodes - 1 times. After the first iteration, currentNode references the second node. After the second iteration, it references the third node. If the loop were to iterate numberOfNodes - 1 times, currentNode would reference the last node. It would not be null.

Beginning the Implementation

- Design Decision:
  - The structure of the chain of linked nodes
- Add statements at beginning
  - Use code from Cases 1 and 2
- Add at end
  - Must invoke **`getNodeAt`**
  - Could maintain both head and tail references (deferred until later)

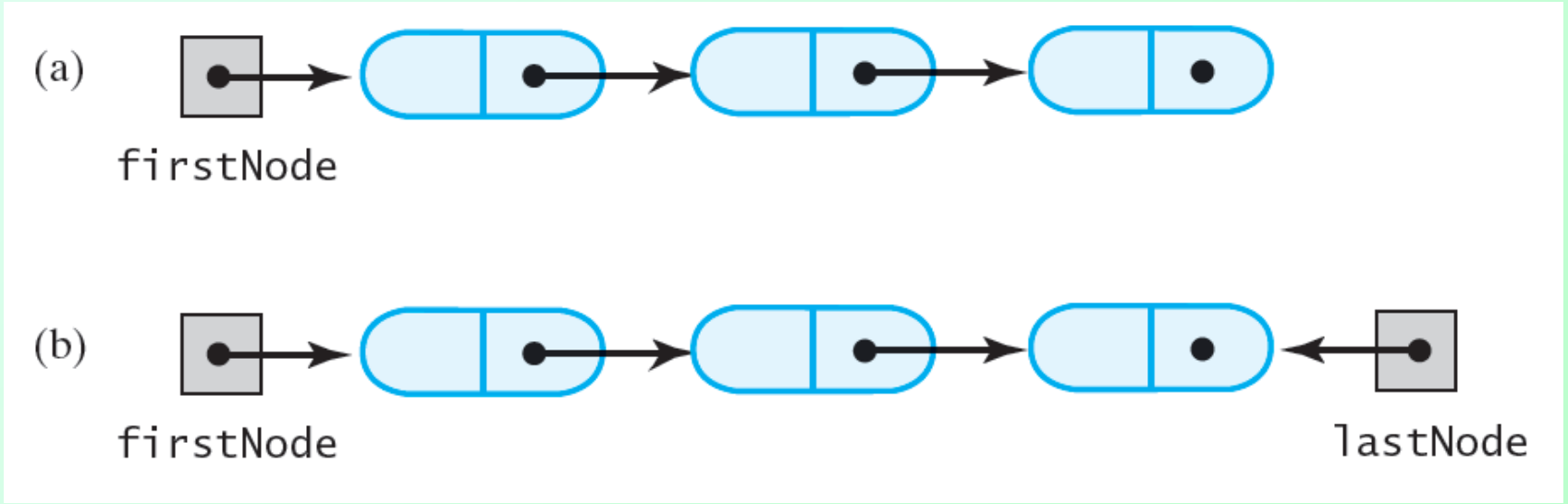


Figure 14-7 A linked chain with (a) a head reference; (b) both a head reference and a tail reference

Implementation

- View whole class, Listing 14-1
- Note
  - Methods for `add`
  - Method `isEmpty`
  - Method `toArray`
- Consider initial test program, Listing 14-2
  - Output of listing 14-2

Note: Code listing files must be in same folder as PowerPoint files for links to work

Implementation

- Finishing the implementation
  - Method `remove`
  - Method `replace`
  - Method `contains`
- Refining the implementation
  - Add tail reference
  - Avoids traversal of entire chain when **`add`** is called

Question 14 Compare the time required to replace an entry in a list using the previous method replace with the time required for the array-based version given in Segment 13.12.

Question 14 Compare the time required to replace an entry in a list using the previous method replace with the time required for the array-based version given in Segment 13.12.

The method replace given in this chapter performs more work than an array-based replace because it must traverse the chain to locate the entry to replace. An array-based replace can locate the desired entry directly, given its array index.

Question 17 What is the Big Oh of the method toArray, as given in Segment 14.14?

Question 18 What is the Big Oh of the method remove, as given in Segment 14.16?

Question 17 What is the Big Oh of the method toArray, as given in Segment 14.14?

O(n).

Question 18 What is the Big Oh of the method remove, as given in Segment 14.16?

O(1) when removing the first entry, or O(n) otherwise.

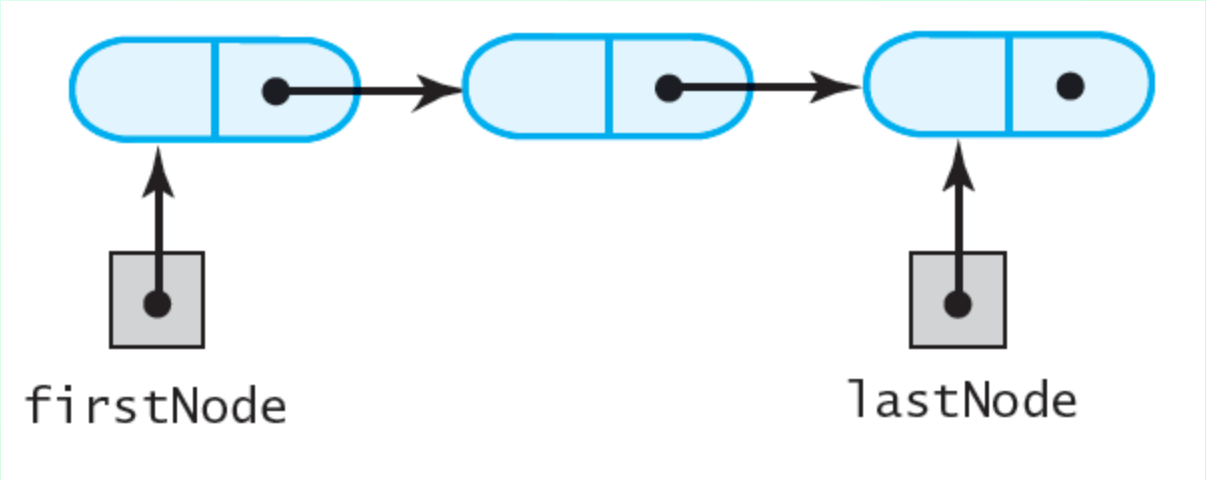


Figure 14-8 A linked chain with both a head reference and a tail reference

Implement with Tail Reference

- Must alter method clear
  - `lastNode = null;`
- Adding to end of list
  - For empty list, head and tail reference new node
  - For non-empty list, use
    `lastNode.setNextNode(newNode);`
    `lastNode = newNode;`
- View changes, Listing 14-A

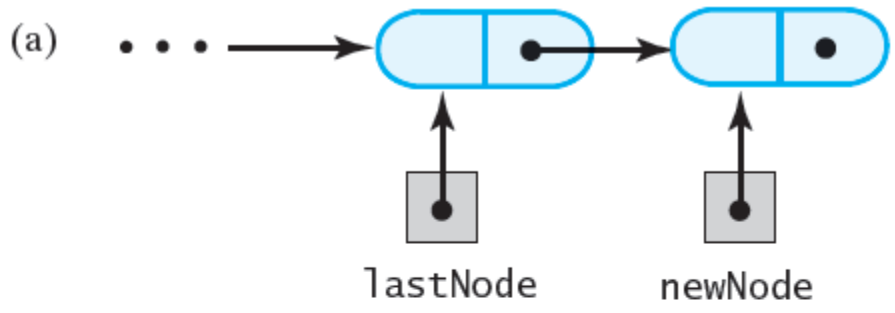


FIGURE 14-9 Adding a node to the end of a nonempty chain that has a tail reference

Operation	AList	LList	LList2
add(newEntry)	O(1)	O(n)	O(1)
add(newPosition, newEntry)	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(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 14-11 The time efficiencies of the ADT list operations for three implementations, expressed in Big Oh notation

Design Decisions

- Efficiency of execution vs. implementation time
- Issues include:
  - Access time
    - Add, remove, search
  - Memory usage
    - Overhead for pointers
    - Wasted memory for arrays

Java Class Library: The Class `LinkedList`

- **`ListInterface`** similar to what we have defined
  - Has more methods
  - May use different name for method
- Class **`LinkedList`**
  - Implements List, Queue, Deque

End

Chapter 14