Algorithms and Data Structures 1
CS 0445

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines:
  - Homework 3: this Friday @ 11:59 pm
  - Lab 2: next Monday @ 11:59 pm
  - Programming Assignment 1: Friday Oct. 7th
- Please include all instructors when sending private messages on Piazza, if possible
- **Student Support Hours** of the teaching team are posted on the Syllabus page

Previous Lecture …

- Code efficiency
  - How to determine running time of an algorithm without running it?
  - Count the number of executed basic operations
    - **as a function of the input size**
  - Determine the order of growth of the runtime function
    - Ignore lower order terms
    - Ignore constant factors
    - Big-Oh approximation

Muddiest Points

- **Q: in what case would you specifically want to use a linked list?**
- **Q: Is a linked list usually more or less efficient than an unlinked one?**
- Linked chains grow and shrink in size based on the actual number of data items.
- Arrays are more rigid in the sense that they need to be allocated contiguously.
- If the actual number of used data items is static, that is, doesn't change widely throughout the runtime of the application, an array would be better (more space efficient)
- Otherwise, use a linked chain
-

Muddiest Points

- **Q: is all the memory needed for every reference variable in an array allocated when the array is created, or when each index is filled? i.e. Does a newly formed (empty) array take up the same space in memory as a filled array.**
- Yes! The reference variables inside an array are allocated when the array is created.

Muddiest Points

- **Q: I am still confused by how memory is allocated with a partially filled array. Do the objects within the array determine this or the reference variable type of the array.**
- String[10] uses the same memory as Integer[10], ArrayBag[10], Square[10], …
- Each has 10 reference variables, and all reference variables have the same size (e.g. 4 bytes)

Muddiest Points

- **Q: Clarification on why a linked bag takes exactly double the space than an arraybag.**
- **Q: Why is it that linked chains will always take up 100% more memory than arrays?**
- A linked chain takes exactly double the space of a **full** array. Each node in the chain has one extra reference variable, which is the next field
- **Q: What if the data fields contained in each node are different than those contained in an array?**
- A: The size of the data objects doesn’t affect the size of the array not the chain node.
- Both contain reference variables and all reference variables are the same size.

Muddiest Points

- **Q: Big-Oh runtime is very confusing to me. Are there easy ways to practice and master this material?**
- A: I will prepare a list of examples on determining the Big-Oh approximations of various functions.
- **What are some examples for the different growth rate functions?**
- 1, log log n, $\log^2 n$, n, n log n, n^2, 2^n, n!

Muddiest Points

- **Q: How does one "lose the chain" when incorrectly removing nodes in a chain?**
- If we change what firstNode points to before saving that in another variable.
- Incorrect way to remove first node:

```
firstNode = newNode;
newNode.next = firstNode;
```

- Correct way:

```
newNode.next = firstNode;
firstNode = newNode;
```

Muddiest Points

- **Q: The big oh notation, how does it actual works**
- A: Big-Oh is an approximation tool.
- $5n^2 + 30\ n + 100 = O(n^2)$
- It breaks a function down to its **order of growth**, how fast is the rate of function value increase when input increases
- **Q: I still don't quite get the big O notion. Like why isn't 2^cn not O(2^n)?**
- $2^{cn} = 2^{(c-1)n}\ 2^n$
- cannot be expressed as a *constant x* 2^n

Muddiest Points

- **Q: How would you update the pointer in the linked list to something in the center of the list.**
- We make an outside pointer point to a node in the center by traversing the list starting from its first node.

Muddiest Points

- **Q: Why does big o matter?**
- Because it extracts the order of growth of a function. In algorithm analysis we care more about the order of growth of runtime than about the exact runtime values.

Muddiest Points

- **Q: Calculating the number of executed steps of an algorithm**
- Watch for loops and determine the number of loop iterations

Muddiest Points

- **Q: I still don't fully understand the remove implementation for a linked bag. Would it not be the same, if not more efficient, by traversing the linked list and removing there as replacing the middle element with the first element and removing the first element? Both require a traversal, which would be O(n) but replacing the middle Node's data with the first Node will be an extra operation.**
- You are right! Removing a node from middle of a LinkedBag can be done by:
  - cutting it out of the chain and
  - by replacing its data with firstNode.data
  - both are O(n) because they both require chain traversal
  - cutting the node out is a bit more complicated than simply removing the first node.

Today's Agenda

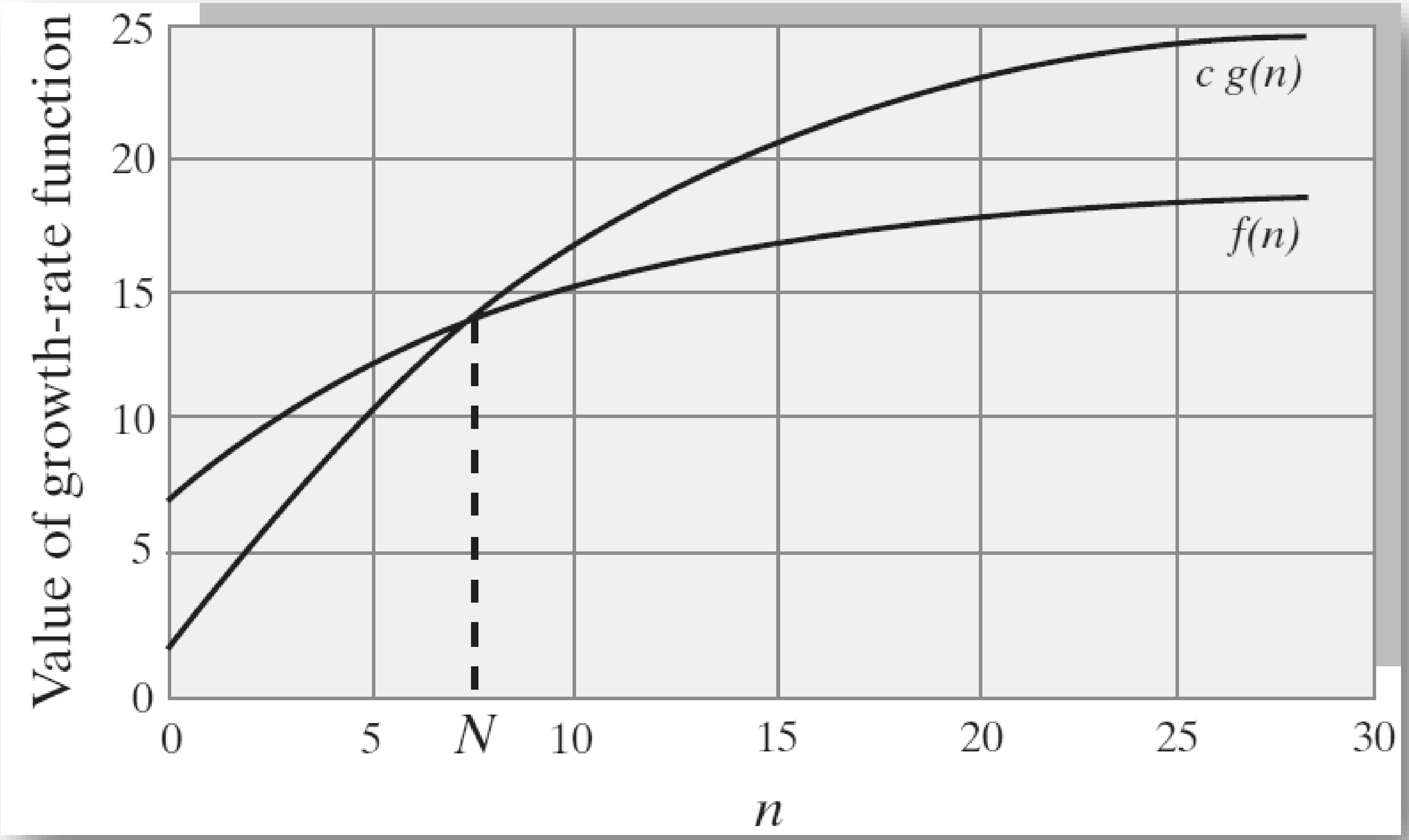
- Big-Oh Approximation
- ADT List
  - Fixed-size array implementation: ArrayList

Big Oh Notation

- A function f(n) is of order at most g(n)
- That is, f(n) is O(g(n))—if
  - A positive real number c and positive integer N exist …
  - Such that $f(n) \leq c * g(n)$ for all $n \geq N$
  - That is, $c * g(n)$ is an upper bound on $f(n)$ when n is sufficiently large

Big Oh Notation

- An illustration of the definition of Big Oh



Big Oh Notation

- Identities for Big Oh Notation

The following identities hold for Big Oh notation:

$$O(k\,g(n)) = O(g(n)) \text{ for a constant } k$$
$$O(g_1(n)) + O(g_2(n)) = O(g_1(n) + g_2(n))$$
$$O(g_1(n)) \times O(g_2(n)) = O(g_1(n) \times g_2(n))$$
$$O(g_1(n) + g_2(n) + \ldots + g_m(n)) = O(\max(g_1(n), g_2(n), \ldots, g_m(n)))$$
$$O(\max(g_1(n), g_2(n), \ldots, g_m(n))) = \max(O(g_1(n)), O(g_2(n)), \ldots, O(g_m(n)))$$

By using these identities and ignoring smaller terms in a growth-rate function, you can usually find the order of an algorithm's time requirement with little effort. For example, if the growth-rate function is $4n^2 + 50n - 10$,

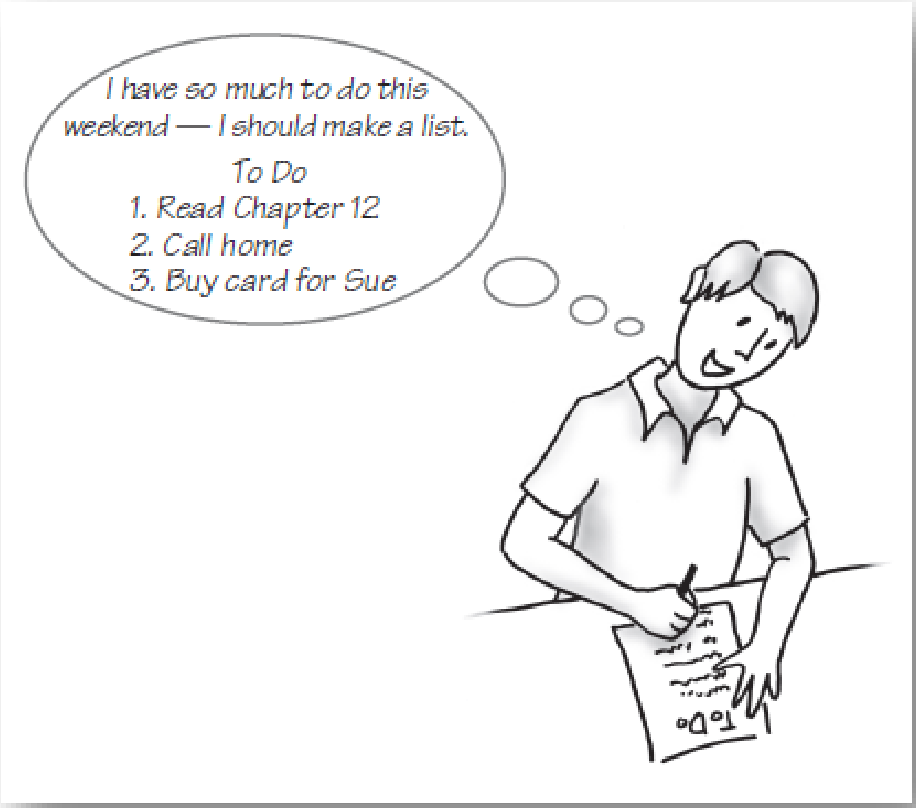
$$\begin{aligned} O(4n^2 + 50n - 10) &= O(4n^2) && \text{by ignoring the smaller terms} \\ &= O(n^2) && \text{by ignoring the constant multiplier} \end{aligned}$$

Complexities of Program Constructs

Construct	Time Complexity
Consecutive program segments $S_1, S_2, \ldots, S_k$ whose growth-rate functions are $g_1, \ldots, g_k$, respectively	$\max(O(g_1), O(g_2), \ldots, O(g_k))$
An `if` statement that chooses between program segments S_1 and S_2 whose growth-rate functions are g_1 and g_2, respectively	$O(\mathit{condition}) + \max(O(g_1), O(g_2))$
A loop that iterates m times and has a body whose growth-rate function is g	$m \times O(g(n))$

Lists

A to-do list



Specifications for the ADT List

- `add (newEntry)`
- `add (newPosition, newEntry)`
- `remove(givenPosition)`
- `clear()`
- `replace( givenPosition, newEntry)`

`getEntry( givenPosition)`

`toArray()`

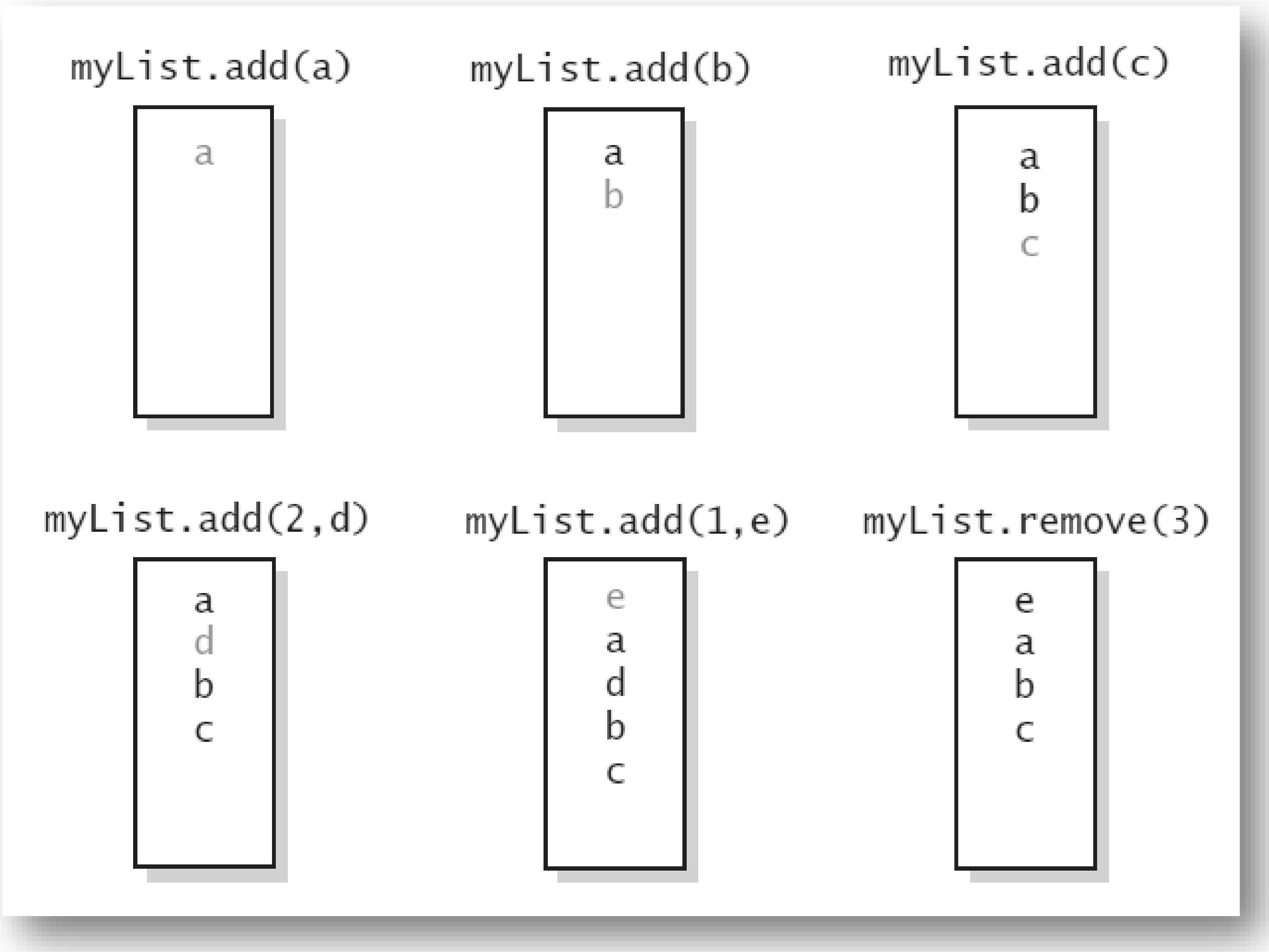
`contains(anEntry)`

`getLength()`

`isEmpty()`

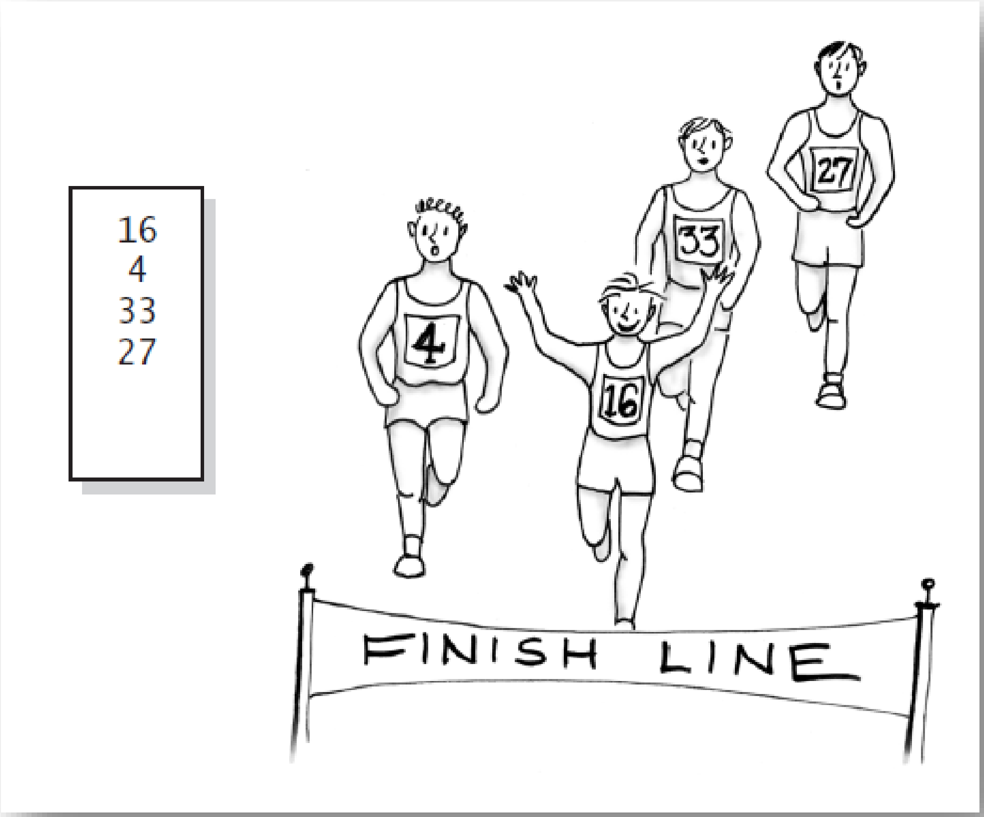
Specifications for the ADT List

The effect of ADT list operations
on an initially empty list



Using the ADT List

A list of numbers that identify runners in the order in which they finished a race



Using the ADT List

A client of a class
that implements **ListInterface**

```java
public class ListClient
{
   public static void main(String[] args)
   {
      testList();
   } // end main

   public static void testList()
   {
      ListInterface<String> runnerList = new AList<>();
//  runnerList has only methods in ListInterface

      runnerList.add("16"); // Winner
      runnerList.add(" 4"); // Second place
      runnerList.add("33"); // Third place
      runnerList.add("27"); // Fourth place
      displayList(runnerList);
   } // end testList

   public static void displayList(ListInterface<String> list)
```

Using the ADT List

A client of a class
that implements **`ListInterface`**

```java
   public static void displayList(ListInterface<String> list)
   {
      int numberOfEntries = list.getLength();
      System.out.println("The list contains " + numberOfEntries +
                         " entries, as follows:");

      for (int position = 1; position <= numberOfEntries; position++)
         System.out.println(list.getEntry(position) +
                            " is entry " + position);

      System.out.println();
   } // end displayList
} // end ListClient
```

Output

```
The list contains 4 entries, as follows:
16 is entry 1
 4 is entry 2
33 is entry 3
27 is entry 4
```

Java Class Library:The Interface `List`

Method headers from the interface `List`

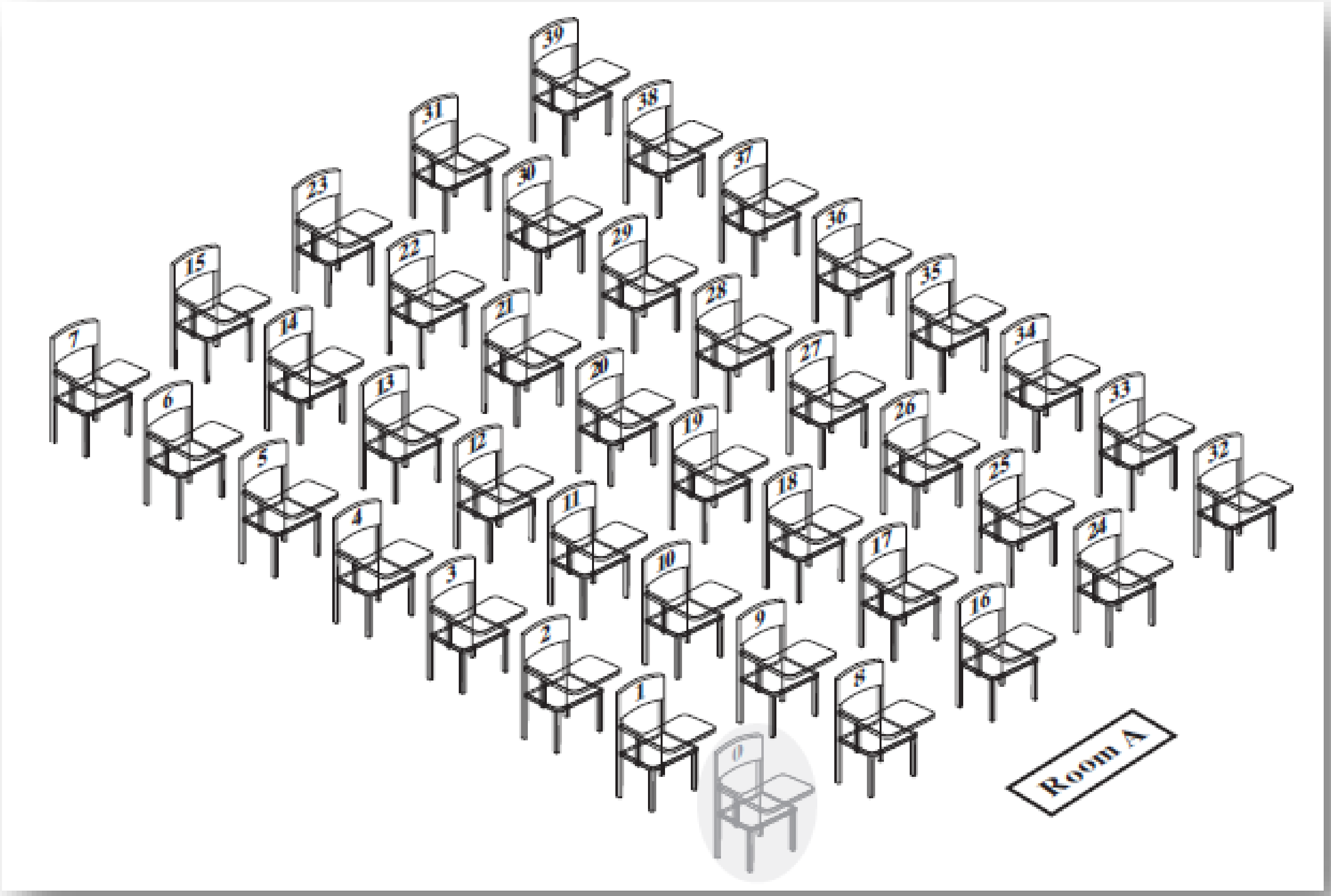
```
public boolean add(T newEntry)
public void add(int index, T newEntry)
public T remove(int index)
public void clear()
public T set(int index, T anEntry) // Like replace
public T get(int index)            // Like getEntry
public boolean contains(Object anEntry)
public int size()                  // Like getLength
public boolean isEmpty()
```

Java Class Library: The Class `ArrayList`

- Available constructors
  - `public ArrayList()`
  - `public ArrayList(int initialCapacity)`
- Similar to java.util.vector
  - Can use either `ArrayList` or `Vector` as an implementation of the interface `List`.

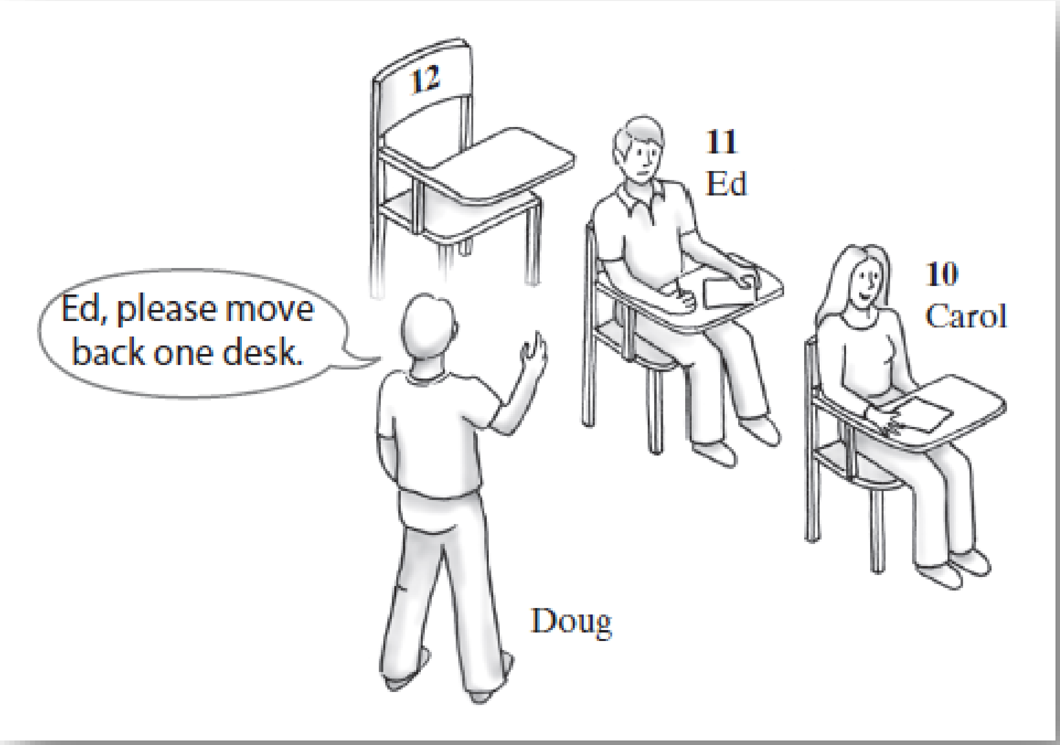
Array to Implement the ADT List

A classroom that contains desks in fixed positions



Array to Implement the ADT List

Seating a new student between two existing students:
At least one other student must move



Array to Implement the ADT List

UML notation for the class **`AList`**

```
AList
-----------------------------------------------
-list: T[]
-numberOfEntries: integer
-DEFAULT_CAPACITY: integer
-MAX_CAPACITY: integer
-initialized: boolean
-----------------------------------------------
+add(newEntry: T): void
+add(newPosition: integer, newEntry: T): void
+remove(givenPosition: integer): T
+clear(): void
+replace(givenPosition: integer, newEntry: T): T
+getEntry(givenPosition: integer): T
+toArray(): T[]
+contains(anEntry: T): boolean
+getLength(): integer
+isEmpty(): boolean
```

Array to Implement the ADT List

The class **AList**

```java
import java.util.Arrays;
/**
    A class that implements a list of objects by using an array.
    Entries in a list have positions that begin with 1.
    Duplicate entries are allowed.
    @author Frank M. Carrano
*/
public class AList<T> implements ListInterface<T>
{
    private T[] list;    // Array of list entries; ignore list[0]
    private int numberOfEntries;
    private boolean initialized = false;
    private static final int DEFAULT_CAPACITY = 25;
    private static final int MAX_CAPACITY = 10000;

    public AList()
    {
        this(DEFAULT_CAPACITY); // Call next constructor
    } // end default constructor

    public AList(int initialCapacity)
```

Array to Implement the ADT List

The class **AList**

```java
    } // end default constructor

    public AList(int initialCapacity)
    {
        // Is initialCapacity too small?
        if (initialCapacity < DEFAULT_CAPACITY)
            initialCapacity = DEFAULT_CAPACITY;
        else // Is initialCapacity too big?
            checkCapacity(initialCapacity);

        // The cast is safe because the new array contains null entries
        @SuppressWarnings("unchecked")
        T[] tempList = (T[])new Object[initialCapacity + 1];
        list = tempList;
        numberOfEntries = 0;
        initialized = true;
    } // end constructor
```

Array to Implement the ADT List

The class **`AList`**

```java
public void add(T newEntry)
{
   checkInitialization();
   list[numberOfEntries + 1] = newEntry;
   numberOfEntries++;
   ensureCapacity();
} // end add

public void add(int newPosition, T newEntry)
{ < Implementation deferred >
} // end add

public T remove(int givenPosition)
{ < Implementation deferred >
} // end remove
```

Array to Implement the ADT List

The class **AList**

```java

   public void clear()
   { < Implementation deferred >
   } // end clear

   public T replace(int givenPosition, T newEntry)
   { < Implementation deferred >
   } // end replace

   public T getEntry(int givenPosition)
   { < Implementation deferred >
   } // end getEntry

   public T[] toArray()
   {
      checkInitialization();

      // The cast is safe because the new array contains null entries
      @SuppressWarnings("unchecked")
      T[] result = (T[])new Object[numberOfEntries];
      for (int index = 0; index < numberOfEntries; index++)
      {
         result[index] = list[index + 1];
      } // end for
```

Array to Implement the ADT List

The class `AList`

```java
         result[index] = list[index + 1];
      } // end for

      return result;
   } // end toArray

   public boolean contains(T anEntry)
   { < Implementation deferred >
   } // end contains

   public int getLength()
   {
      return numberOfEntries;
   } // end getLength

   public boolean isEmpty()
   {
      return numberOfEntries == 0; // Or getLength() == 0
   } // end isEmpty
```

Array to Implement the ADT List

The class **AList**

```
      return numberOfEntries == 0; // or getLength() == 0
   } // end isEmpty

   // Doubles the capacity of the array list if it is full.
   // Precondition: checkInitialization has been called.
   private void ensureCapacity()
   {
      int capacity = list.length - 1;
      if (numberOfEntries >= capacity)
      {
         int newCapacity = 2 * capacity;
         checkCapacity(newCapacity); // Is capacity too big?
         list = Arrays.copyOf(list, newCapacity + 1);
      } // end if
   } // end ensureCapacity
   < This class will define checkCapacity, checkInitialization, and two more private
     methods that will be discussed later. >
} // end AList
```

Array to Implement the ADT List

- Implementation of **`add`** uses a private method **`makeRoom`** to handle the details of moving data within the array

```
// Precondition: The array list has room for another entry.
public void add(int newPosition, T newEntry)
{
   checkInitialization();
   if ((newPosition >= 1) && (newPosition <= numberOfEntries + 1))
   {
      if (newPosition <= numberOfEntries)
         makeRoom(newPosition);
      list[newPosition] = newEntry;
      numberOfEntries++;
      ensureCapacity(); // Ensure enough room for next add
   }
   else
      throw new IndexOutOfBoundsException(
                "Given position of add's new entry is out of bounds.");
} // end add
```

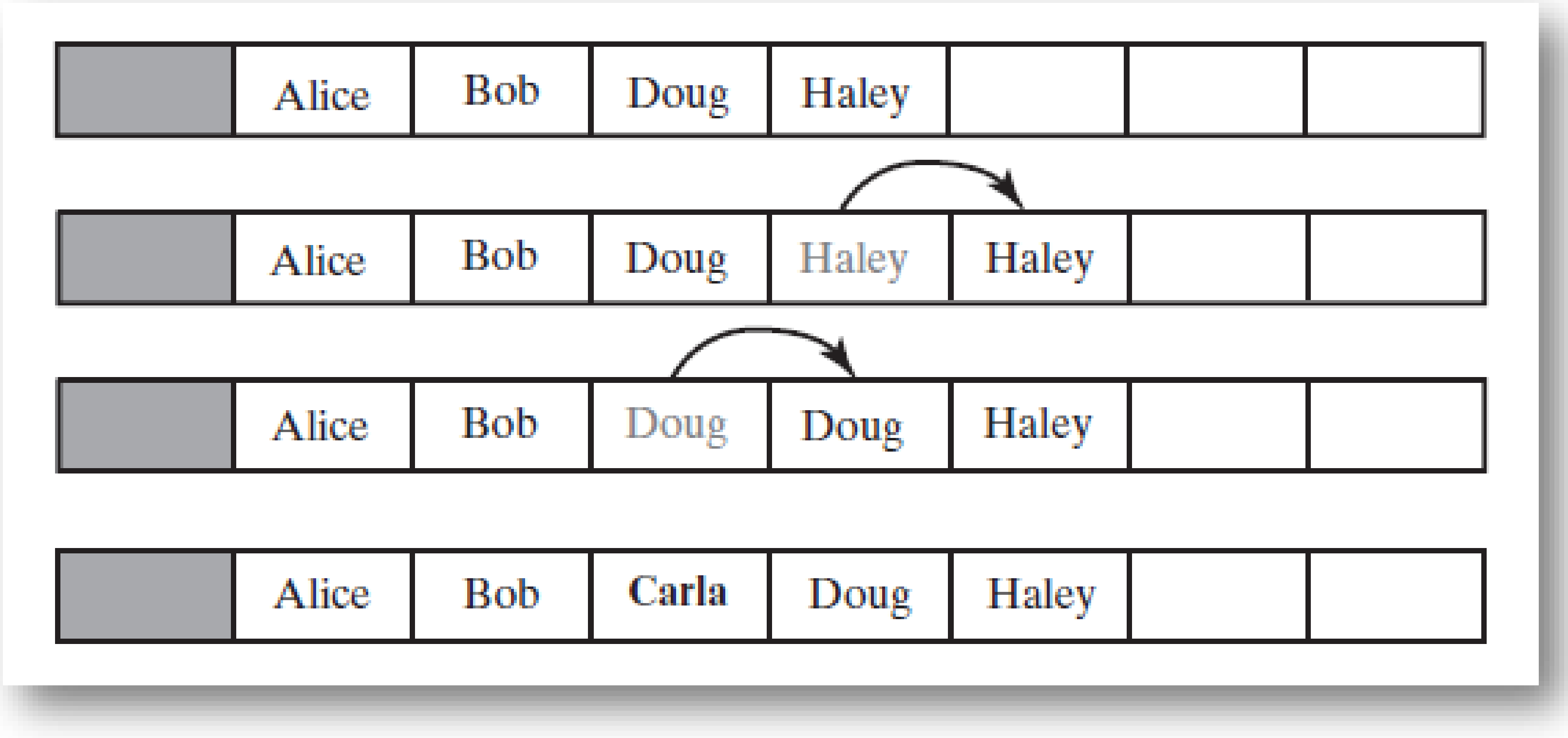
Array to Implement the ADT List

Implement the private method **makeRoom**

```java
// Makes room for a new entry at newPosition.
// Precondition: 1 <= newPosition <= numberOfEntries + 1;
//               numberOfEntries is list's length before addition;
//               checkInitialization has been called.
private void makeRoom(int newPosition)
{
   assert (newPosition >= 1) && (newPosition <= numberOfEntries + 1);

   int newIndex = newPosition;
   int lastIndex = numberOfEntries;

   // Move each entry to next higher index, starting at end of
   // list and continuing until the entry at newIndex is moved
   for (int index = lastIndex; index >= newIndex; index--)
      list[index + 1] = list[index];
} // end makeRoom
```

Array to Implement the ADT List

Making room to insert
Carla as the third entry in an array



Array to Implement the ADT List

Implementation uses a private method **removeGap** to handle the details of moving data within the array.

```
public T remove(int givenPosition)
{
   checkInitialization();
   if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
   {
      assert !isEmpty();
      T result = list[givenPosition]; // Get entry to be removed
      // Move subsequent entries toward entry to be removed,
      // unless it is last in list
      if (givenPosition < numberOfEntries)
         removeGap(givenPosition);
      numberOfEntries--;
      return result; // Return reference to removed entry
   }
   else
      throw new IndexOutOfBoundsException(
                "Illegal position given to remove operation.");
} // end remove
```

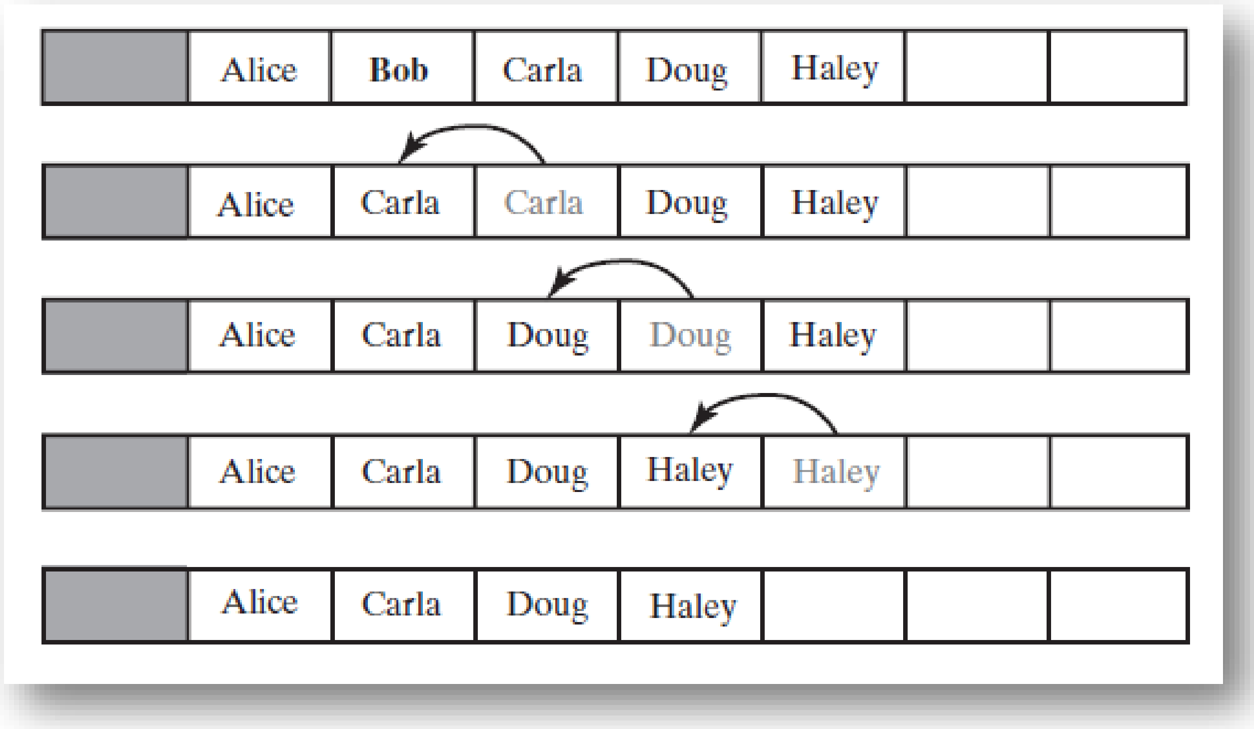
Array to Implement the ADT List

Method `removeGap` shifts list entries within the array

```
// Shifts entries that are beyond the entry to be removed to the
// next lower position.
// Precondition: 1 <= givenPosition < numberOfEntries;
//               numberOfEntries is list's length before removal;
//               checkInitialization has been called.
private void removeGap(int givenPosition)
{
   assert (givenPosition >= 1) && (givenPosition < numberOfEntries);
   int removedIndex = givenPosition;
   int lastIndex = numberOfEntries;
   for (int index = removedIndex; index < lastIndex; index++)
      list[index] = list[index + 1];
} // end removeGap
```

Array to Implement the ADT List

Removing Bob by shifting array entries



Array to Implement the ADT List

Method `replace`

```java
public boolean replace(int givenPosition, T newEntry)
{
   checkInitialization();
   if ((givenPosition >= 1) && (givenPosition <= numberOfEntries))
   {
      assert !isEmpty();
      T originalEntry = list[givenPosition];
      list[givenPosition] = newEntry;
      return originalEntry;
   }
      else
         throw new IndexOutOfBoundsException(
                    "Illegal position given to replace operation.");
} // end replace
```

Array to Implement the ADT List

Method `getEntry`

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

Array to Implement the ADT List

Method **contains** uses a local boolean variable to terminate the loop when we find the desired entry.

```
public boolean contains(T anEntry)
{
   checkInitialization();
   boolean found = false;
   int index = 1;
   while (!found && (index <= numberOfEntries))
   {
      if (anEntry.equals(list[index]))
         found = true;
      index++;
   } // end while
   return found;
} // end contains
```

Array to Implement the ADT List

- Operation that adds a new entry to the end of a list.
- Efficiency O(1) if new if array is not resized.

```
public void add(T newEntry)
{
   checkInitialization();
   list[numberOfEntries] = newEntry;
   numberOfEntries++;
   ensureCapacity();
} // end add
```

Array to Implement the ADT List

Add a new entry to a list at a client-specified position.

```
public void add(int newPosition, T newEntry)
{
   checkInitialization();
   if ((newPosition >= 1) && (newPosition <= numberOfEntries + 1))
   {
      if (newPosition <= numberOfEntries)
         makeRoom(newPosition);
      list[newPosition] = newEntry;
      numberOfEntries++;
      ensureCapacity();
   }
   else
      throw new IndexOutOfBoundsException(
                "Given position of add's new entry is out of bounds.");
   } // end add
```

Array to Implement the ADT List

Method **add** uses method **makeRoom**.

```
private void makeRoom(int newPosition)
{
    int newIndex = newPosition;
    int lastIndex = numberOfEntries;
    for (int index = lastIndex; index >= newIndex; index--)
       list[index + 1] = list[index];
} // end makeRoom
```

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:

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

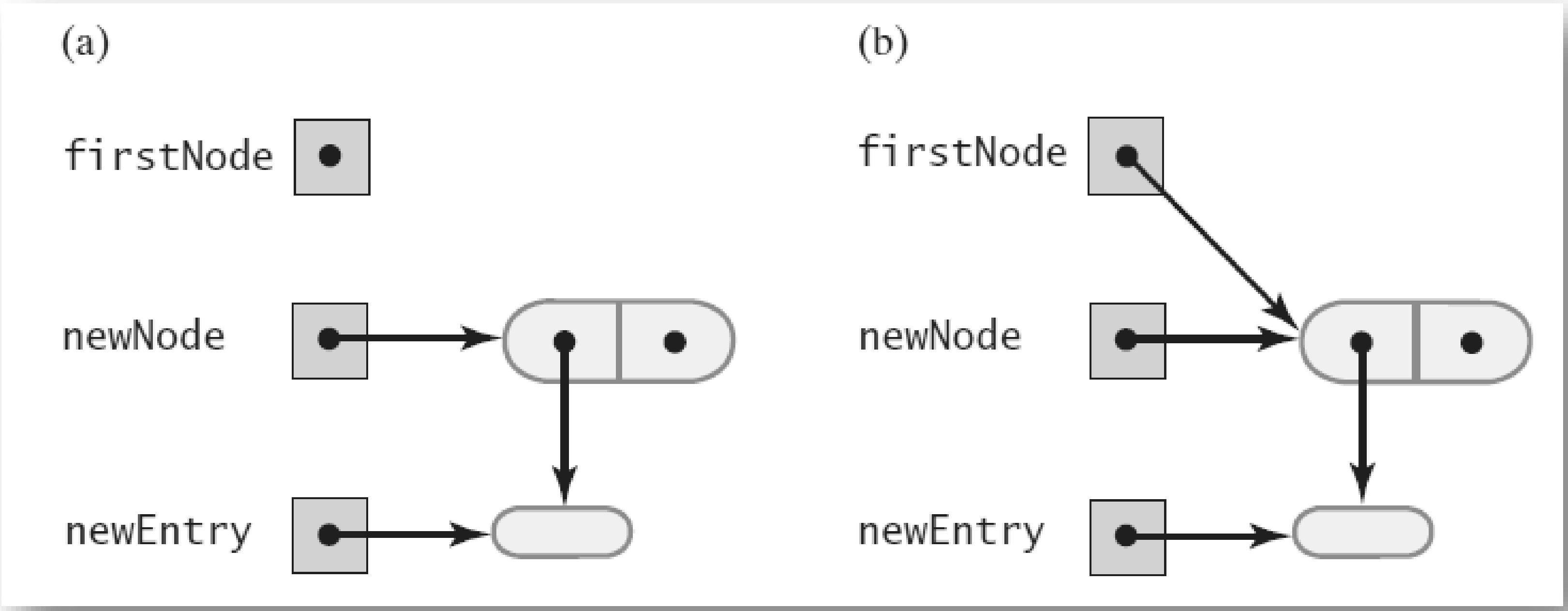
Adding a Node to an empty chain

- 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

Adding a Node

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



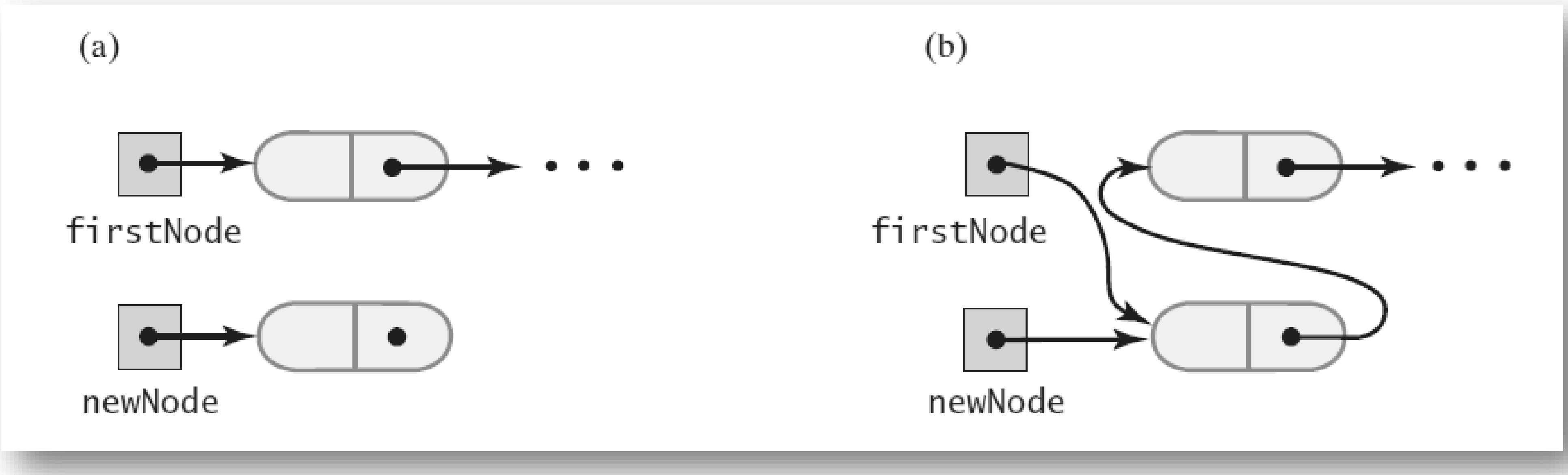
Adding a Node

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
```

Adding a Node

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



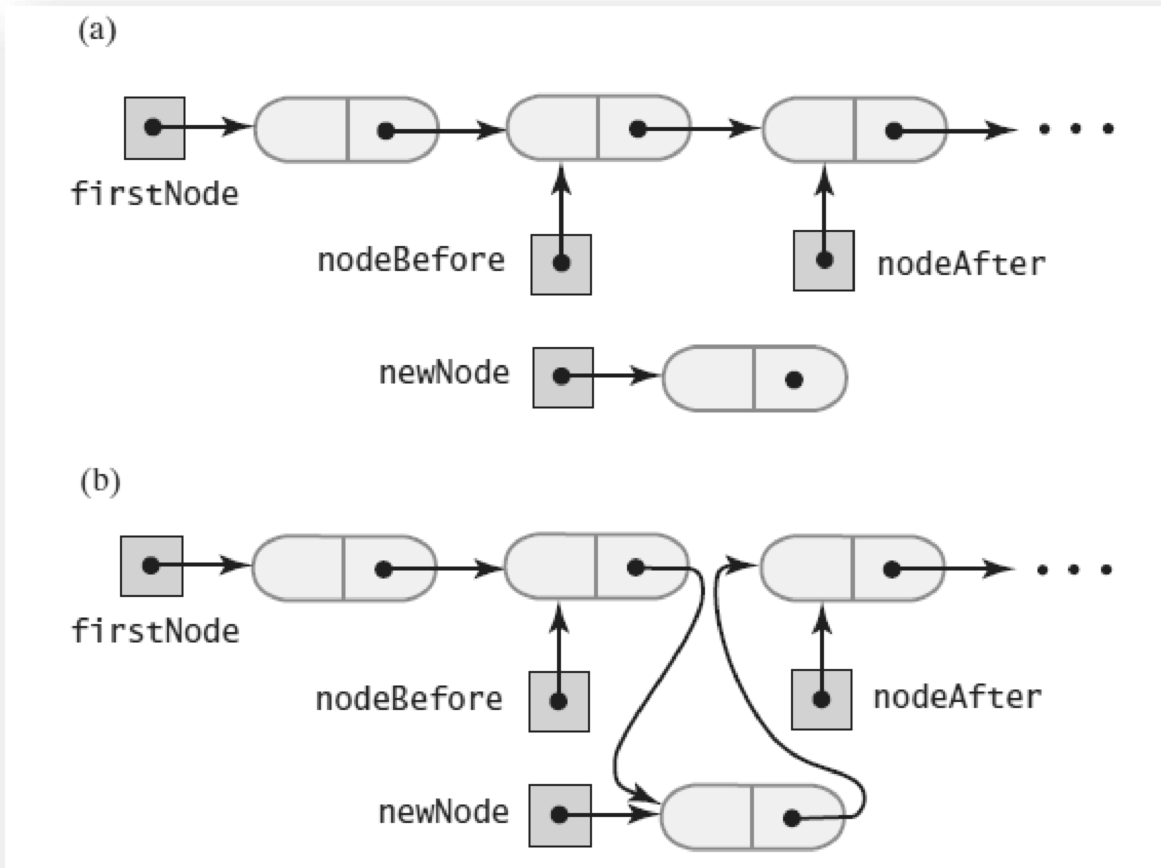
Adding a Node

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 chain of nodes (a) just prior to adding a node between two adjacent nodes; (b) just after adding a node between two adjacent nodes



Adding a Node

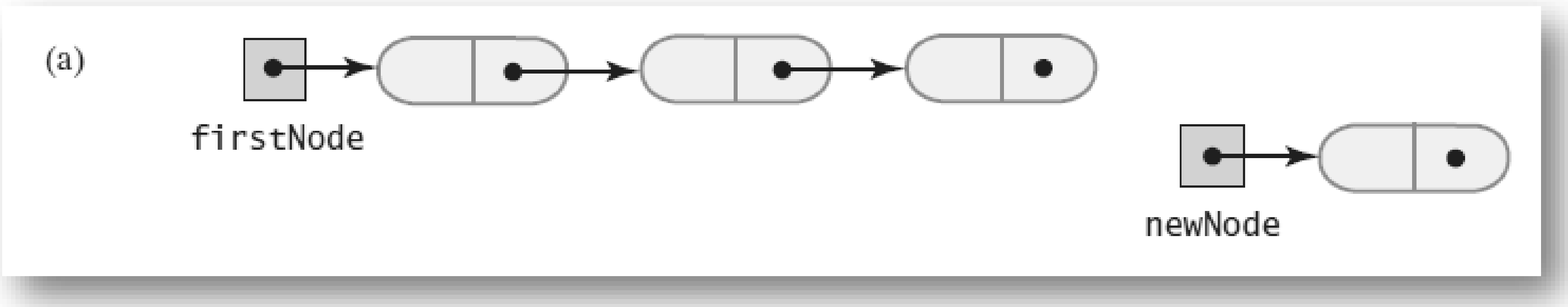
Steps to add a node at the end of a chain.

`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

A chain of nodes

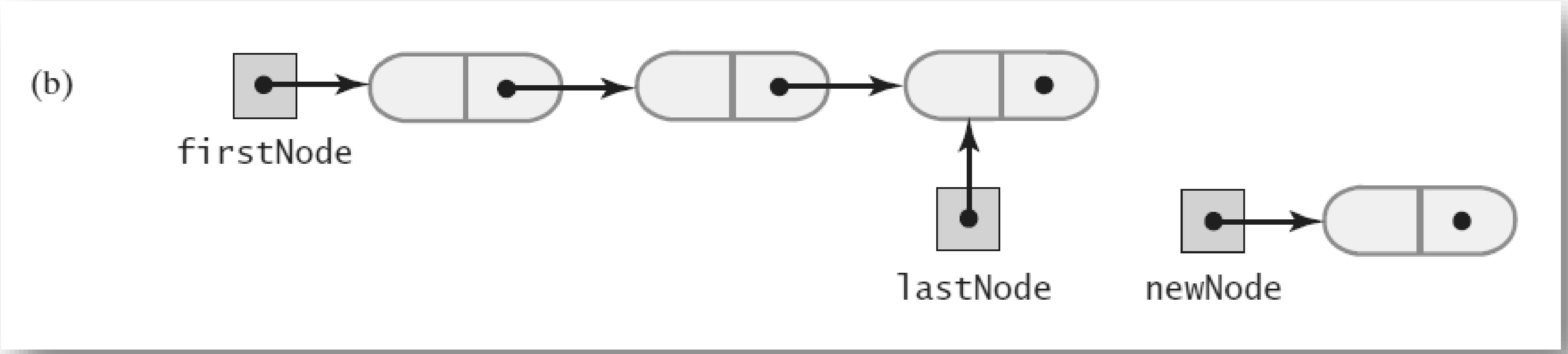
(a) prior to adding a node at the end



Adding a Node

A chain of nodes

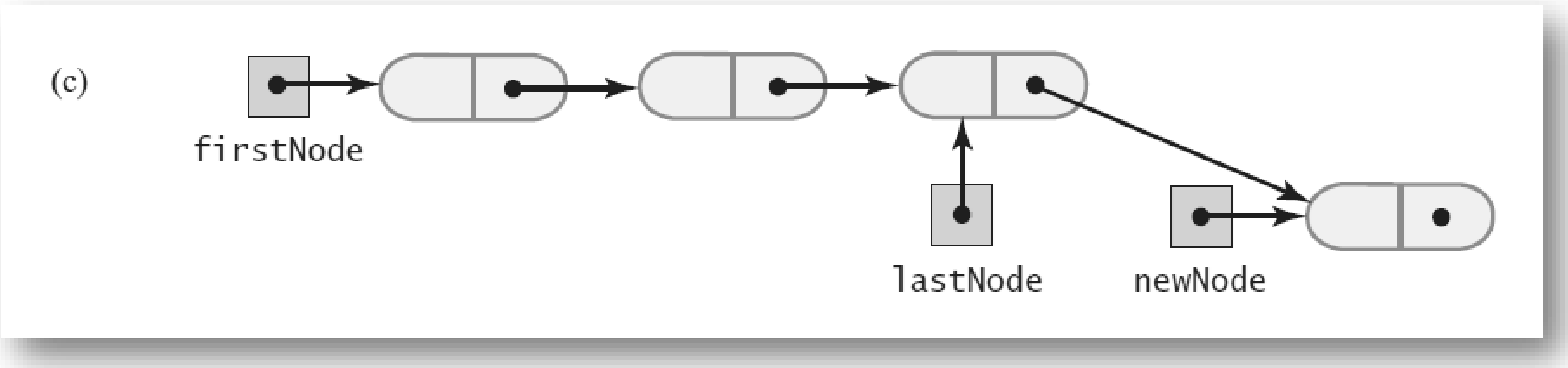
(b) after locating its last node;



Adding a Node

A chain of nodes

(c) after adding a node at the end



Removing a Node from Various Positions

Possible cases

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

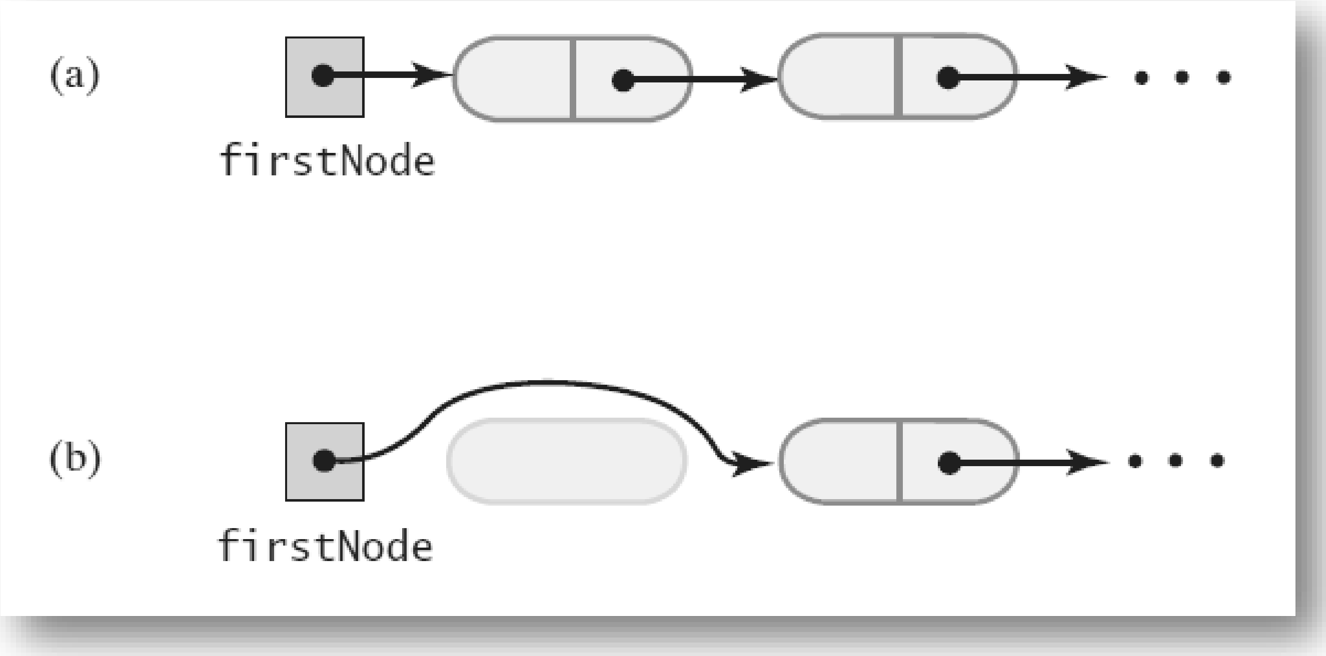
Removing a Node

Steps for removing the first node.

Set `firstNode` *to the link in the first node.*
Since all references to the first node no longer exist, the system automatically recycles the first node's memory.

Removing a Node

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



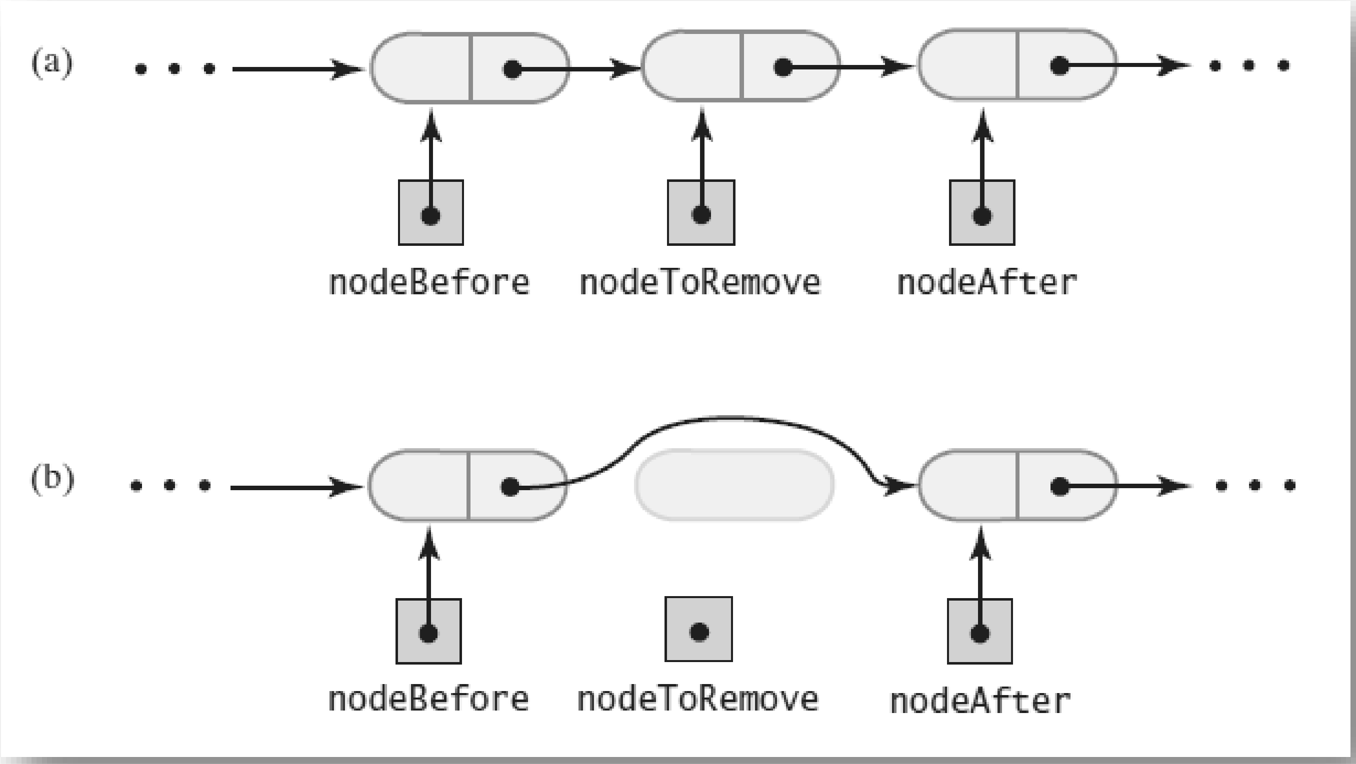
Removing a Node

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 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

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



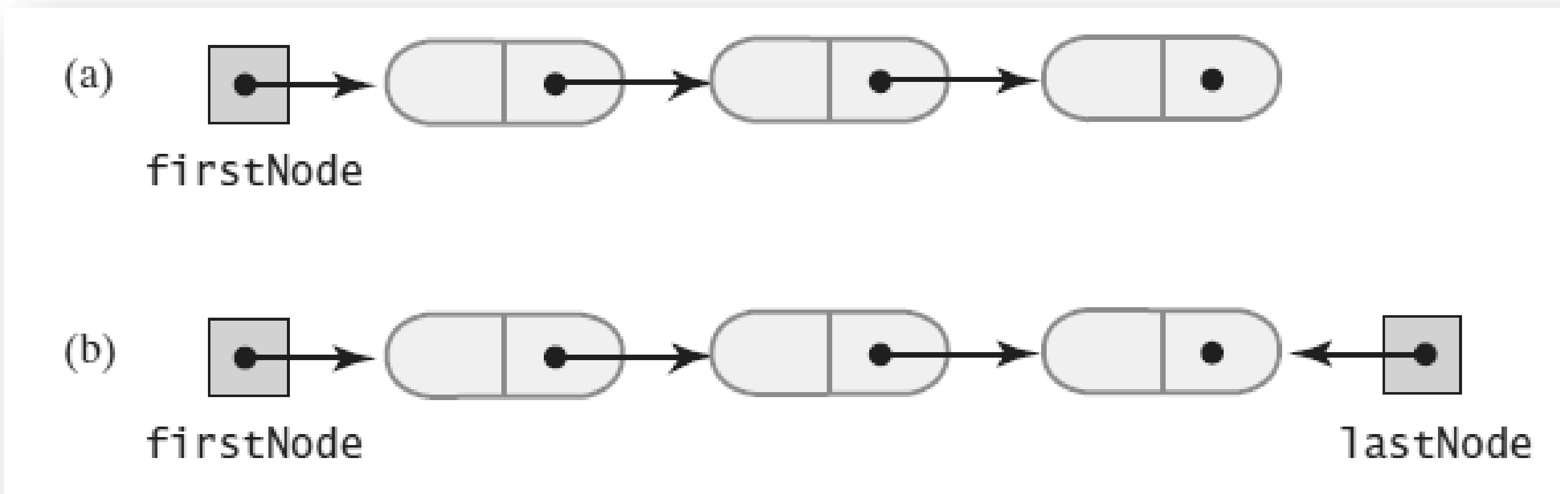
Removing a Node

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

```
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
```

Design Decision: A Link to Last Node

A linked chain with (a) a head reference;
(b) both a head reference and a tail reference



Data Fields and Constructor

An outline of the class **LList**

```
/**
   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. >
  . . .
```

Data Fields and Constructor

An outline of the class **`LList`**

```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
```

Adding to the End of the List

The method **add** assumes method **getNodeAt**

```
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
```

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;
      }
      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`

Note use of assert statement.

```
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
```

Method `toArray`

Traverses chain, loads an array.

```
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
```

Testing Core Methods

A **main** method that tests part of the implementation of the ADT list

```java
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();
```

Testing Core Methods

A **`main`** method that tests part of the implementation of the ADT list

```
   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.
```

Continuing the Implementation

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

```java
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
      {
         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

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

```
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
```

Continuing the Implementation

Retrieving a list entry is straightforward.

```
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
```

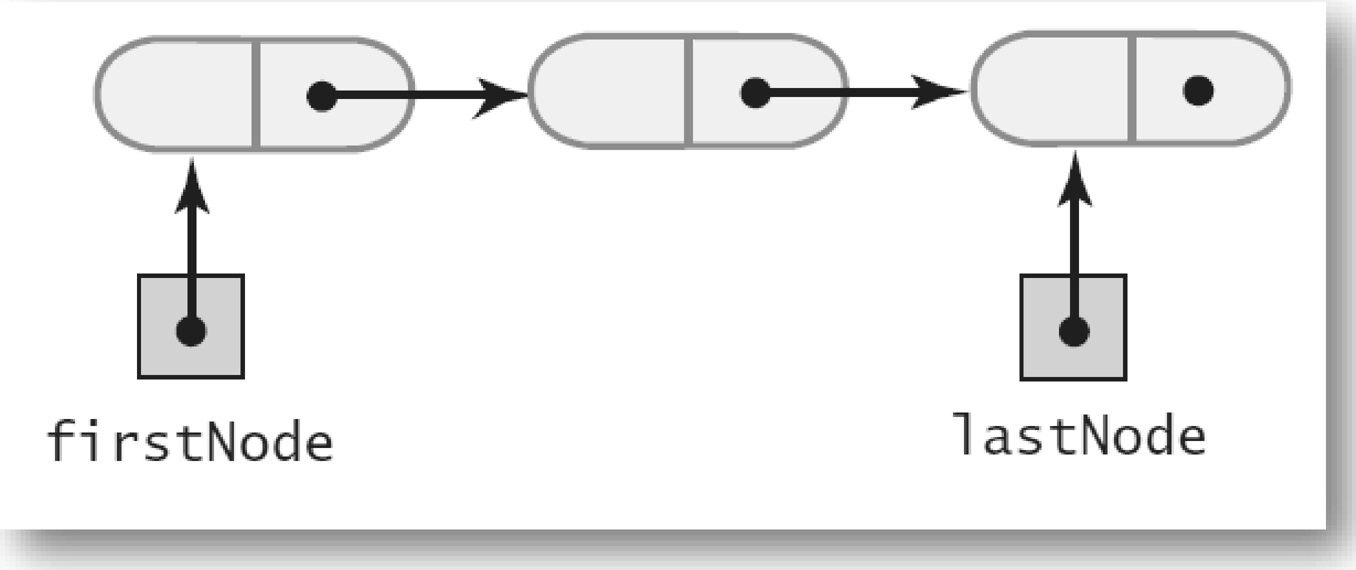
Continuing the Implementation

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

```java
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
```

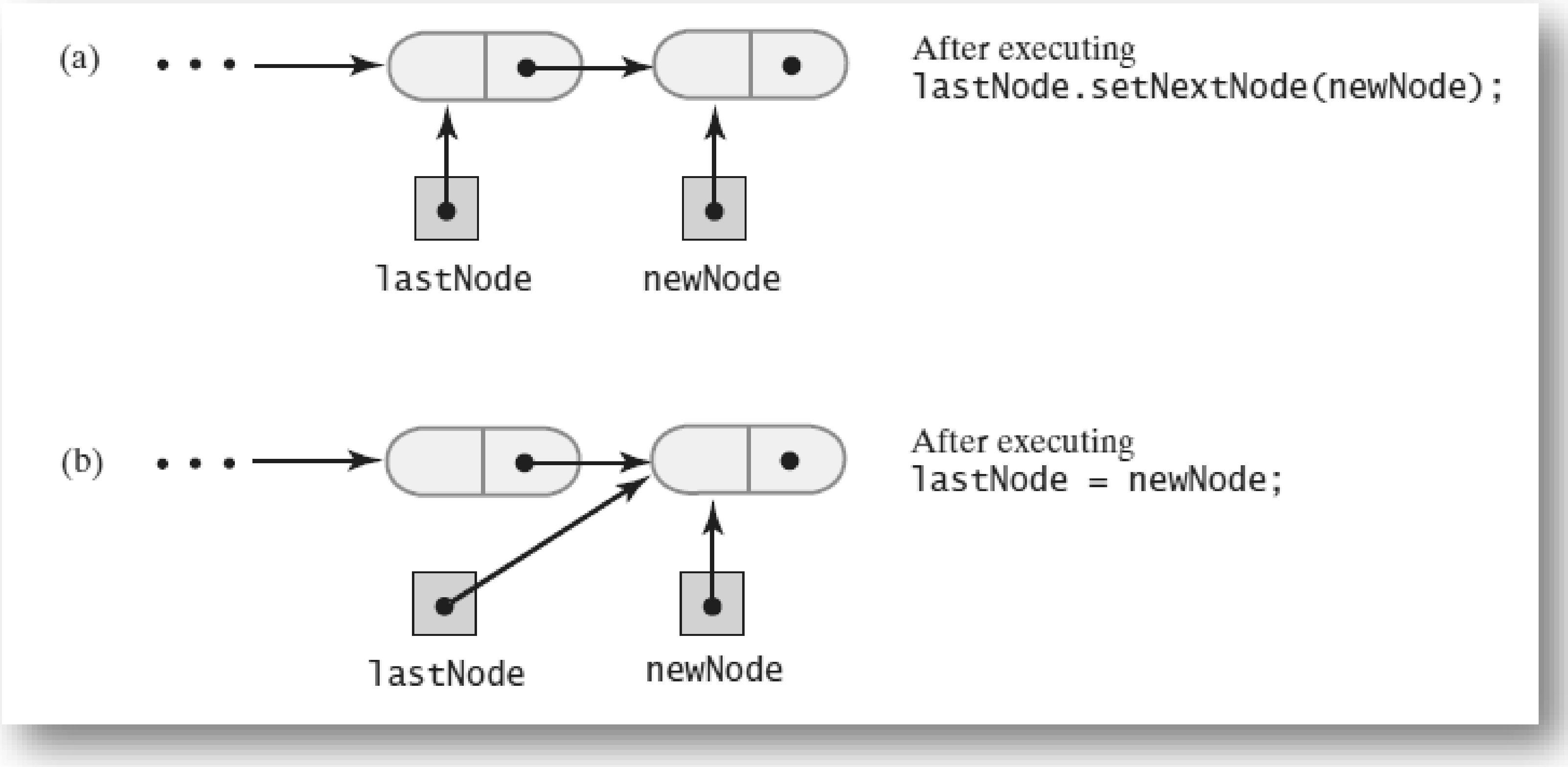
A Refined Implementation

A linked chain with both a head reference and a tail reference



A Refined Implementation

Adding a node to the end of a nonempty chain that has a tail reference



A Refined Implementation

Revision of the first `add` method

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

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

    lastNode = newNode;

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

A Refined Implementation

Implementation of the method that adds by position.

```
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)
```

A Refined Implementation

Implementation of the method that adds by position.

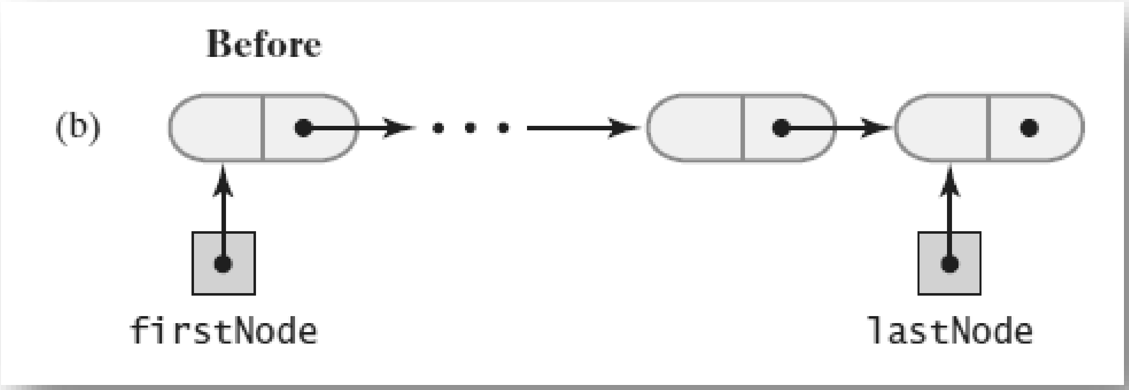
```
        newNode.setNextNode(firstNode);
        firstNode = newNode;
      }
      else if (newPosition == numberOfEntries + 1)
      {
         lastNode.setNextNode(newNode);
         lastNode = newNode;
      }
      else
      {
         Node nodeBefore = getNodeAt(newPosition - 1);
         Node nodeAfter = nodeBefore.getNextNode();
         newNode.setNextNode(nodeAfter);
         nodeBefore.setNextNode(newNode);
      } // end if
      numberOfEntries++;
   }
   else
      throw new IndexOutOfBoundsException(
                 "Illegal position given to add operation.");
} // end add
```

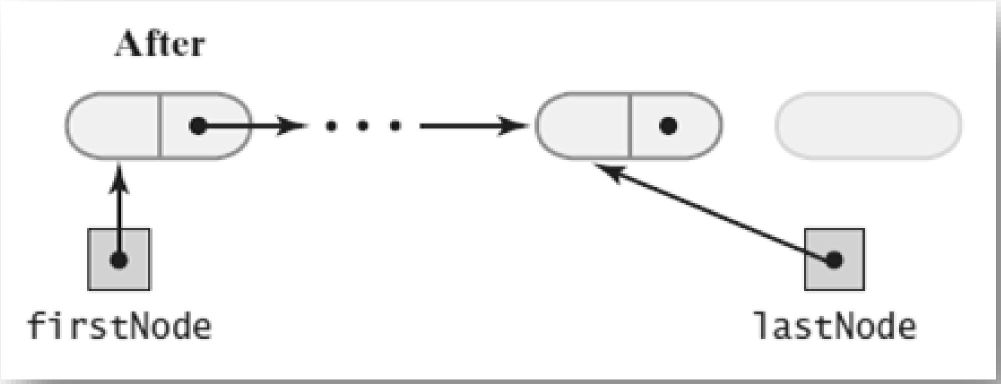
A Refined Implementation

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

A Refined Implementation

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

Implementation of the remove operation:

```
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();
```

A Refined Implementation

Implementation of the remove operation:

```
            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
```

Efficiency of Using a Chain

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

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()`	$O(1)$	$O(1)$	$O(1)$

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.