University of Pittsburgh

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
- Draft slides and handouts available on Canvas
- Lecture recordings are available under Panopto Video on Canvas
- Please use "Regrade Request" feature in GradeScope with any issues with grades
- 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 …

- ADT Bag Implementations
  - Fixed-size array: ArrayBag
    - copy constructor
  - Resizable array: ResizableArrayBag
    - add
  - Linked implementation: LinkedBag

Muddiest Points

- **Q: What is the difference between ArrayBag and Array?**
- A: ArrayBag is an implementation that *uses* a fixed-size array to implement the ADT Bag
- **Q: I am still very confused with all the terminology. For example, “reference object”, reference variables vs. data objects**
- A: An object is an instance of a class. A reference variable points to an object.
  - For example, String x = new String();
  - *x* is a reference variable that points to the String object created by the **new** keyword
  - There is no such thing as “reference object”. Perhaps confused with reference**d** object.

Muddiest Points

- **Q: what is a null pointer exception**
- A: A NullPointerException is a run-time exception raised by the Java run-time every time a reference variable with null value is derefeferenced
    - e.g., ArrayBag<Integer> x;

      x.add(10);
    - The code above will raise a NullPointerException because the reference variable x is dereferenced (using the dot operator) while still being null.
    - How would you fix that?

Muddiest Points

- **Q: does the equal method just check that the two types of objects are the same?**
- A: Practically speaking, no.
- .equals typically checks for types and for values of instance variables.
- Theoretically, one may define equals to do whatever check one wants.

Muddiest Points

- **Q: must we type " "unchecked" " in the parentheses after a SuppressWarning**
- A: Yes, if we want to suppress the unchecked type cast warning. There are other vendor-specific warnings that can be suppressed.

Muddiest Points

- **Q: Traversing the nodes is a bit confusing**
- A: When traversing the nodes of a linked chain, we follow the following steps
    - initialize a *scout* variable to point to the first node
        - Node scout = firstNode;
    - keep moving the scout variable over the nodes until it traverses over the last node
    - How do we know that the scout traversed over the last node
        - while(scout != null)
    - How do we move the scout variable to the next node
        - scout = scout.next

```
Node scout = firstNode
while(scout != null){
  //do something with the node pointed to by scout
   scout = scout.next;
}
```

Muddiest Points

- **Q: Can you go over specific item node removal again?**
- **Q: Maybe if you could go over once more the difference between removing a specified and an unspecified item from a list.**
- A: To remove an unspecific item, we follow the following steps
  - Save the data object pointed to by the first node by making a reference variable point to it
  - Remove the first node
- A: To remove a specific item, we follow the following steps
  - traverse the nodes to find a node that points to an equal object
  - Save the object by making a reference variable point to it
  - Make the found node point to the data object of the first node
  - Remove the first node
- **Q: When removing an item from a linked list how do you make the first item = to the first node.**
- A: The first node points to the first data item.

Muddiest Points

- **Q: Why don't we add new nodes to the end instead of the beginning?**
- A: When we add to the beginning of the chain, we don’t need to traverse the chain to reach the last node. This makes adding at the beginning *faster*.
- **Q: Where is the information stored on other nodes when using .data**
- A: Objects are stored on a memory region called “the heap”

Muddiest Points

- **Q: What would make the difference between a LinkedList and a LinkedBag? the order?**
- A: Yes, and the set of operations that are make sense once you have the items ordered.
- **Q: What is the function of the referenceTo method?**
- A: To traverse the nodes until a node with an equal object found. If so, return a reference to the node; otherwise, return null.

Muddiest Points

- **Q: How can you have a node as a private variable within the Node class? Wouldn't it not be defined yet?**
- A: Java compiler parses class declarations before name resolution
- What would make the difference between a LinkedList and a LinkedBag? the order?
- What is the function of the referenceTo method?

Muddiest Points

Q: If we were provided for at least a minute or two at the end of class there may be more muddiest points provided. Ending lecture exactly (or after!) scheduled class time causes many of us to run to our next classes and forgo muddiest points.

- A: I am sorry about that. Noted!

Today’s Agenda

- A final thought on LinkedBag
- Code efficiency
- ADT List
    - Fixed-size array implementation: ArrayList

Cons of Using a Chain

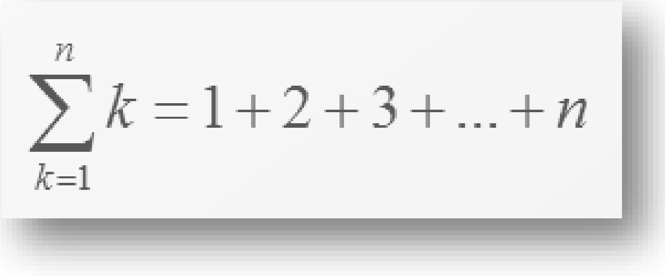
- Removing specific entry requires search of array or chain
- Chain requires more memory than array of same logical size
  - why?

Why do we care about efficient code?

- Computers are faster, have larger memories
  - So why worry about efficient code?
- And … how do we measure efficiency?

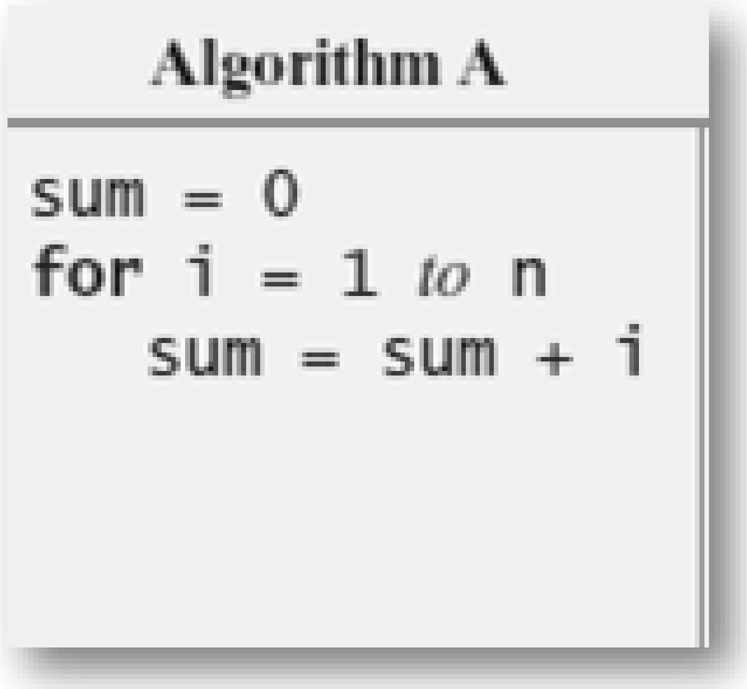
Example

- Consider the problem of summing: computing the sum
  1 + 2 + . . . + n for an integer n > 0



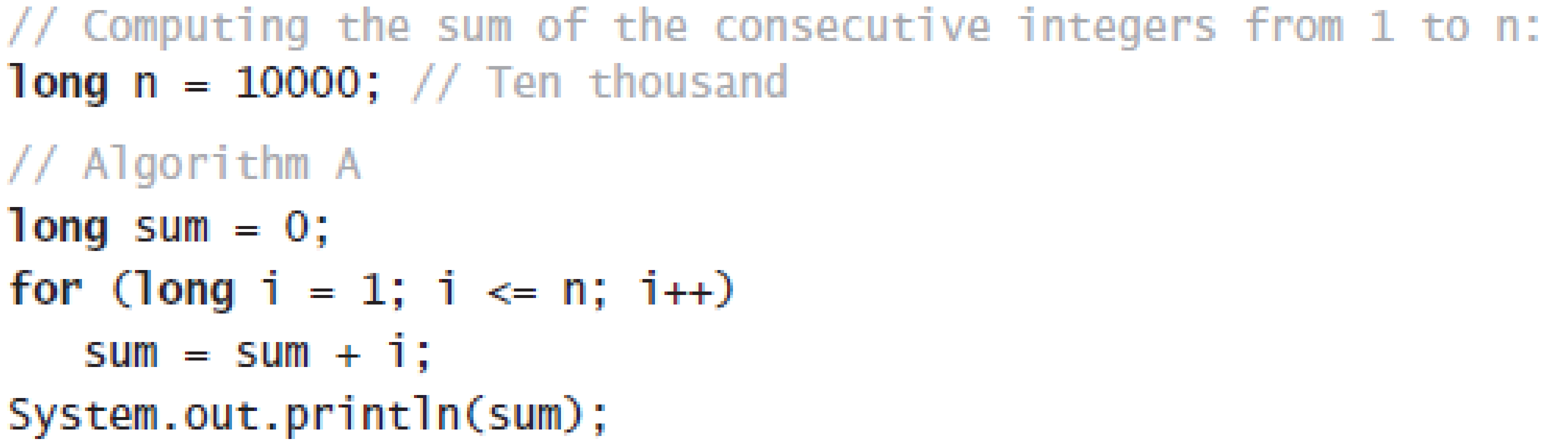
$$\sum_{k=1}^{n} k = 1 + 2 + 3 + ... + n$$

One solution



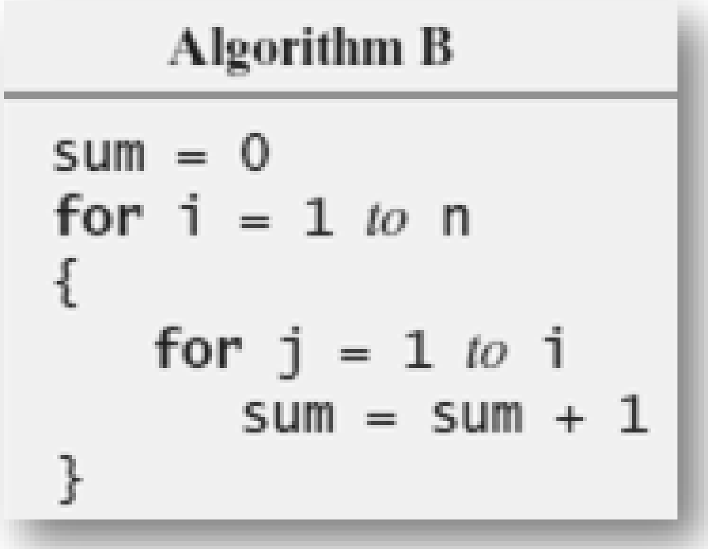
Algorithm A

```
sum = 0
for i = 1 to n
    sum = sum + i
```



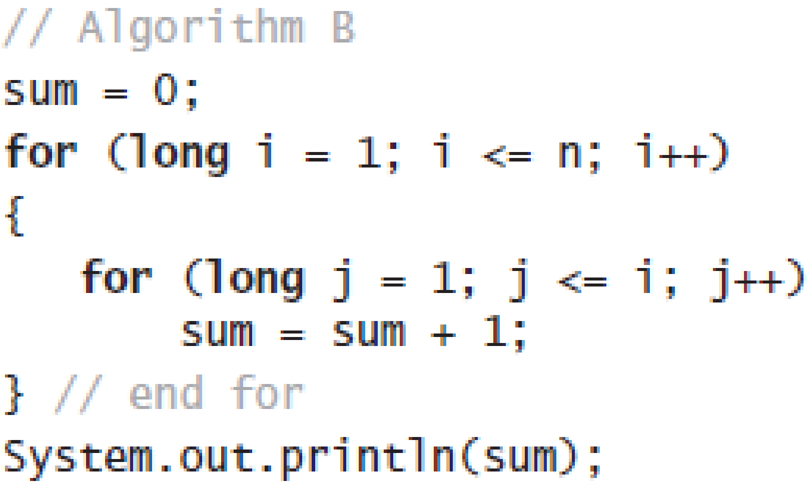
```
// Computing the sum of the consecutive integers from 1 to n:
long n = 10000; // Ten thousand

// Algorithm A
long sum = 0;
for (long i = 1; i <= n; i++)
   sum = sum + i;
System.out.println(sum);
```

Another solution



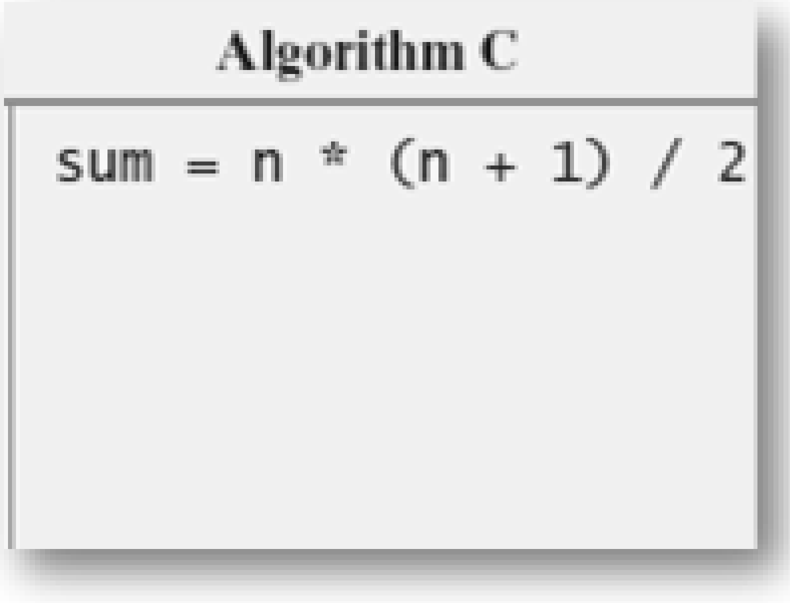
Algorithm B

```
sum = 0
for i = 1 to n
{
    for j = 1 to i
        sum = sum + 1
}
```



```
// Algorithm B
sum = 0;
for (long i = 1; i <= n; i++)
{
   for (long j = 1; j <= i; j++)
      sum = sum + 1;
} // end for
System.out.println(sum);
```

And a third solution



Algorithm C

```
sum = n * (n + 1) / 2
```

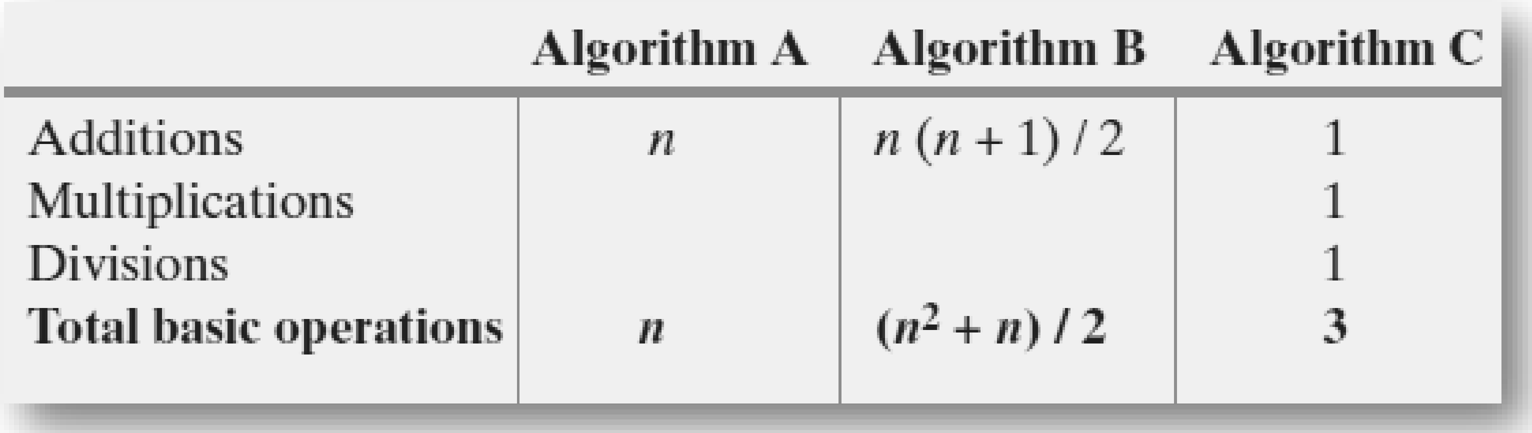
```
// Algorithm C
sum = n * (n + 1) / 2;
System.out.println(sum);
```

Which is “best”?

- An algorithm has both time and space constraints – that is complexity
  - Time complexity
  - Space complexity
- The study of time and space complexities of algorithms is called analysis of algorithms

Counting Basic Operations

- A basic operation of an algorithm
  - The most significant contributor to its total time requirement

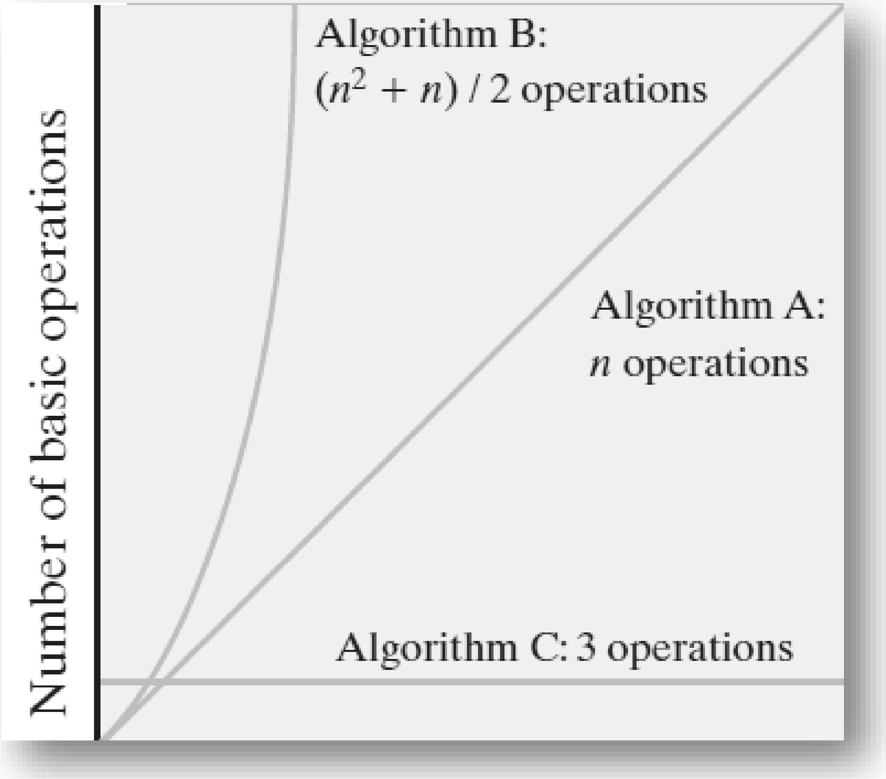


	Algorithm A	Algorithm B	Algorithm C
Additions	n	$n(n+1)/2$	1
Multiplications			1
Divisions			1
Total basic operations	$\boldsymbol{n}$	$\boldsymbol{(n^2+n)/2}$	**3**

- The number of basic operations required by the sum algorithms

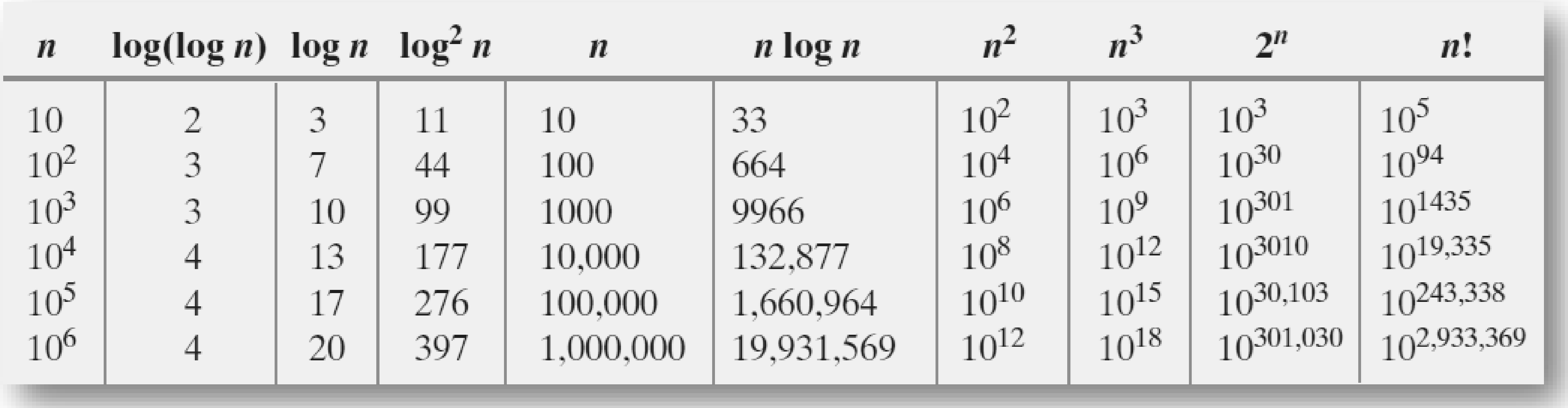
Counting Basic Operations

- The number of basic operations required by the sum algorithms as a function of *n*



Counting Basic Operations

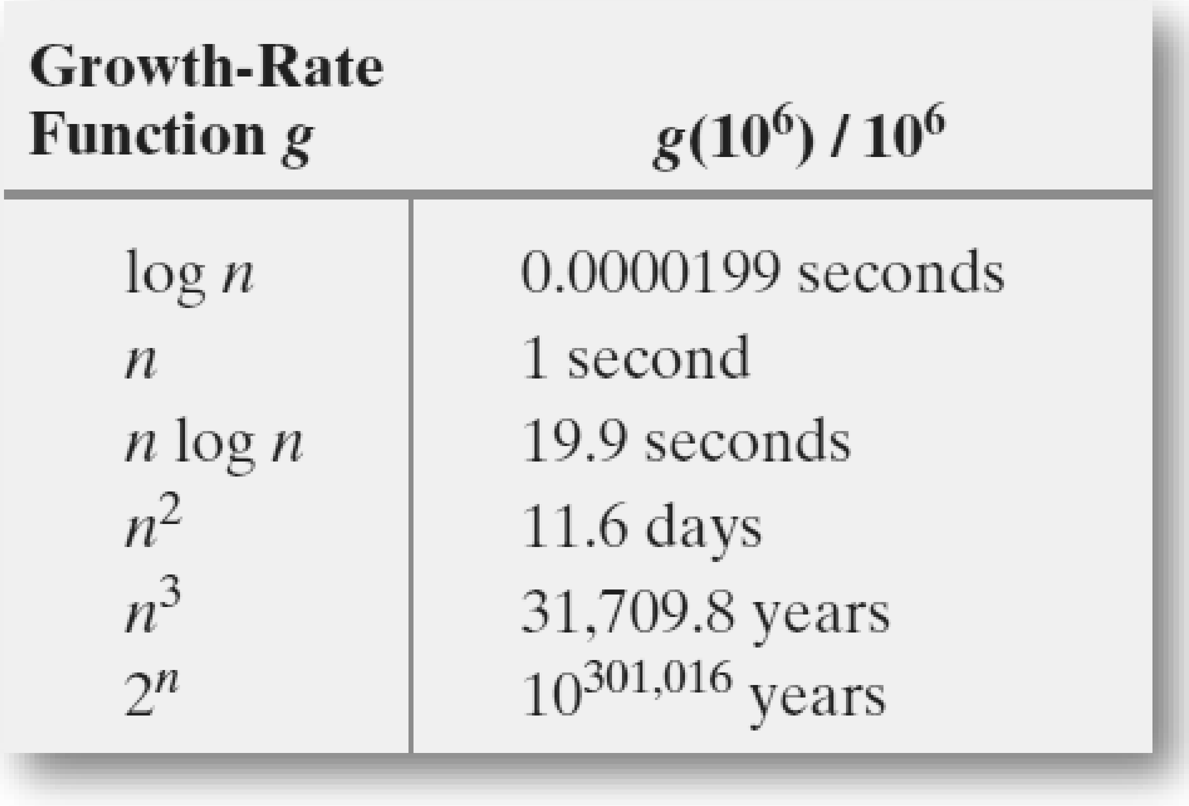
- Typical growth-rate functions evaluated at increasing values of *n*



n	$\log(\log n)$	$\log n$	$\log^2 n$	n	$n \log n$	n^2	n^3	2^n	$n!$
10	2	3	11	10	33	10^2	10^3	10^3	10^5
10^2	3	7	44	100	664	10^4	10^6	10^{30}	10^{94}
10^3	3	10	99	1000	9966	10^6	10^9	10^{301}	10^{1435}
10^4	4	13	177	10,000	132,877	10^8	10^{12}	10^{3010}	$10^{19,335}$
10^5	4	17	276	100,000	1,660,964	10^{10}	10^{15}	$10^{30,103}$	$10^{243,338}$
10^6	4	20	397	1,000,000	19,931,569	10^{12}	10^{18}	$10^{301,030}$	$10^{2,933,369}$

Picturing Efficiency

- The time required to process one million items by algorithms of various orders at the rate of one million operations per second



Growth-Rate Function g	$g(10^6) / 10^6$
$\log n$	0.0000199 seconds
n	1 second
$n \log n$	19.9 seconds
n^2	11.6 days
n^3	31,709.8 years
2^n	$10^{301,016}$ years

Best, Worst, and Average Cases

- For some algorithms, execution time depends only on size of data set
- Other algorithms depend on the nature of the data itself
  - Here we seek to know best case, worst case, average case

Time complexity of an algorithm

- Count the number of **executed** steps (basic operations or just lines of code)

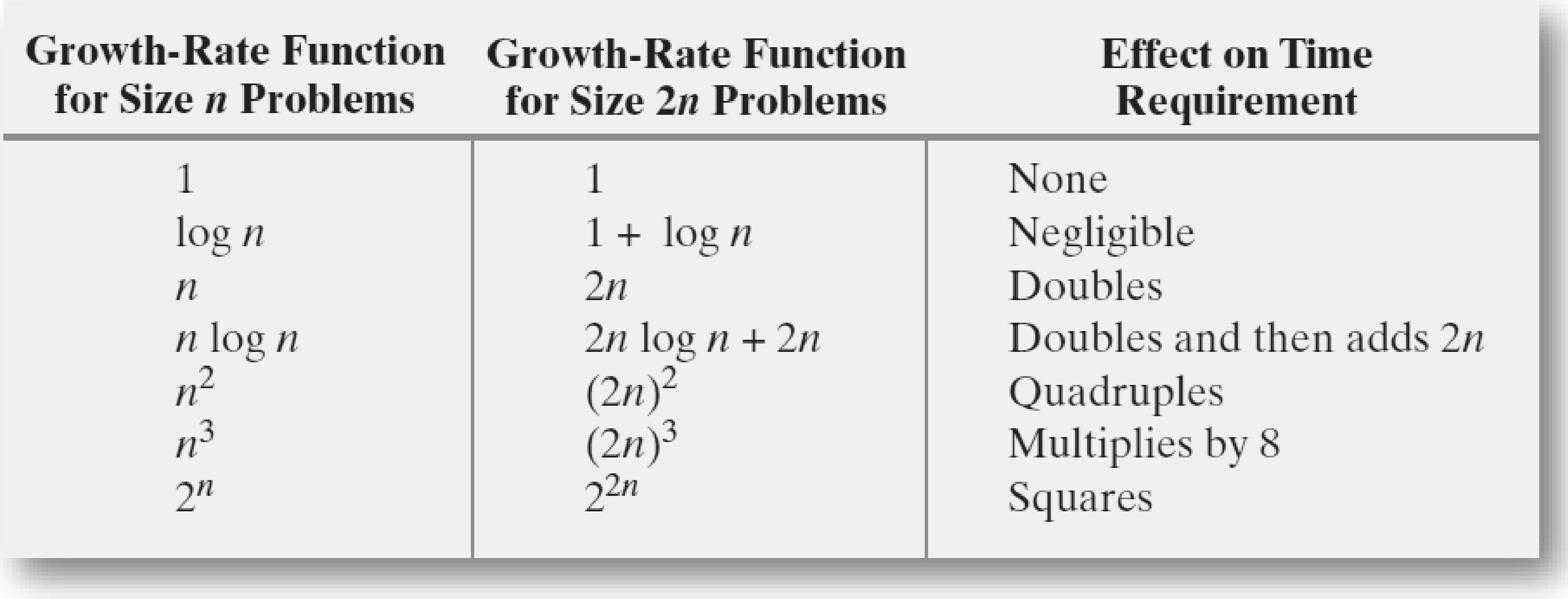
  -
    ```
    sum = 0
    for i = 1 to n
      sum = sum + i
    ```
  - Number of executed lines is 2n + 2
- Let *f(n)* = the number of executed steps
  - *n* is the input size
    - very roughly, the number of keyboard presses needed to enter the input
  - *f(n)* may depend only on *n* or on the actual values of the input
    - In the latter, need to find *f(n)* for best, average, worst cases

Time complexity of an algorithm

- Convert the function f into the ***Big-Oh notation***
  - Ignore lower order terms
    - e.g., constant < log log n < log n < $\log^2 n$ < n < n log n < n^2 < n^3 < 2^n < n!
    - e.g., n^2 + log n = $O(n^2)$
  - Ignore constant factors
    - *c*n = O(n)*, where c is a constant (doesn't depend on *n*)
    - 2^{cn} is **not** $O(2^n)$
  - *f(n) = 2n + 2 = O(2n) = O(n)*

Picturing Efficiency

- The effect of doubling the problem size on an algorithm's time requirement

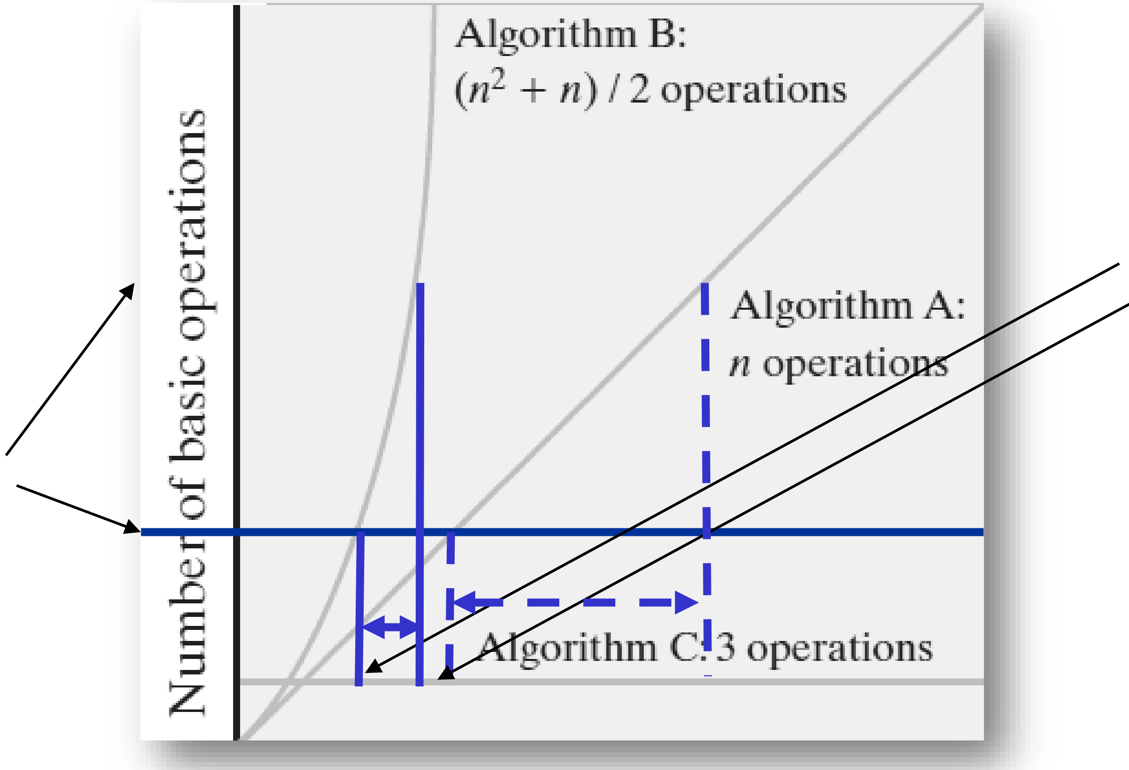


Growth-Rate Function for Size *n* Problems	Growth-Rate Function for Size 2*n* Problems	Effect on Time Requirement
1	1	None
$\log n$	$1 + \log n$	Negligible
n	$2n$	Doubles
$n \log n$	$2n \log n + 2n$	Doubles and then adds $2n$
n^2	$(2n)^2$	Quadruples
n^3	$(2n)^3$	Multiplies by 8
2^n	2^{2n}	Squares

Riding Moore's law

- Writing an efficient algorithm (with less time complexity) is important
  - Such algorithm rides the exponentially-growing curve of hardware-speed ``better"

Number of operations that machine M can do in time T seconds

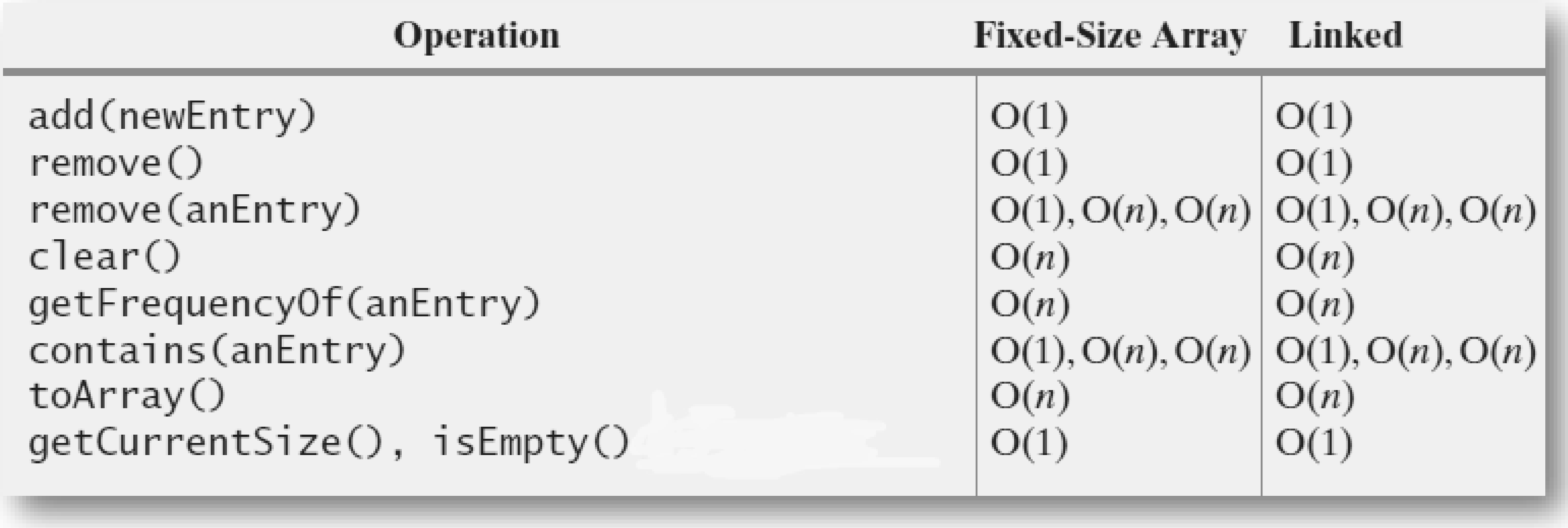


Problem size solvable by machine M using Algorithm B in T second

Problem size solvable by machine M using Algorithm A in T second

Efficiency of Implementations of ADT Bag

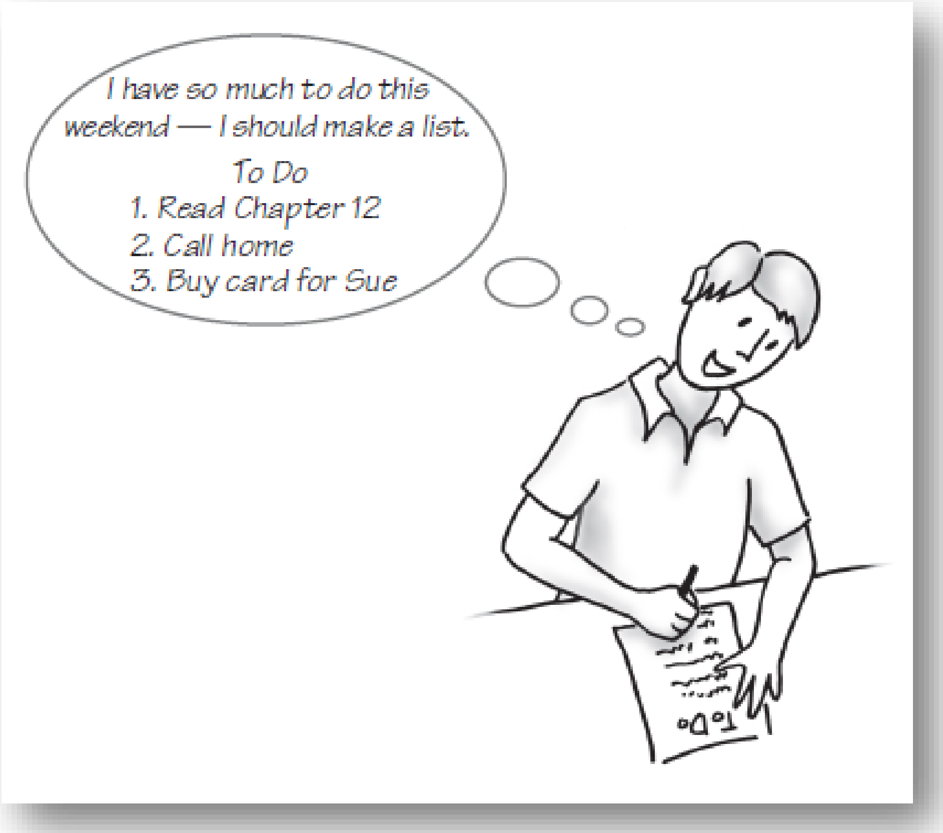
- The time efficiencies of the ADT bag operations for two implementations, expressed in Big Oh notation



Operation	Fixed-Size Array	Linked
`add(newEntry)`	$O(1)$	$O(1)$
`remove()`	$O(1)$	$O(1)$
`remove(anEntry)`	$O(1), O(n), O(n)$	$O(1), O(n), O(n)$
`clear()`	$O(n)$	$O(n)$
`getFrequencyOf(anEntry)`	$O(n)$	$O(n)$
`contains(anEntry)`	$O(1), O(n), O(n)$	$O(1), O(n), O(n)$
`toArray()`	$O(n)$	$O(n)$
`getCurrentSize(), isEmpty()`	$O(1)$	$O(1)$

Lists

A to-do list



Specifications for the ADT List

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

`getEntry( givenPosition)`

`toArray()`

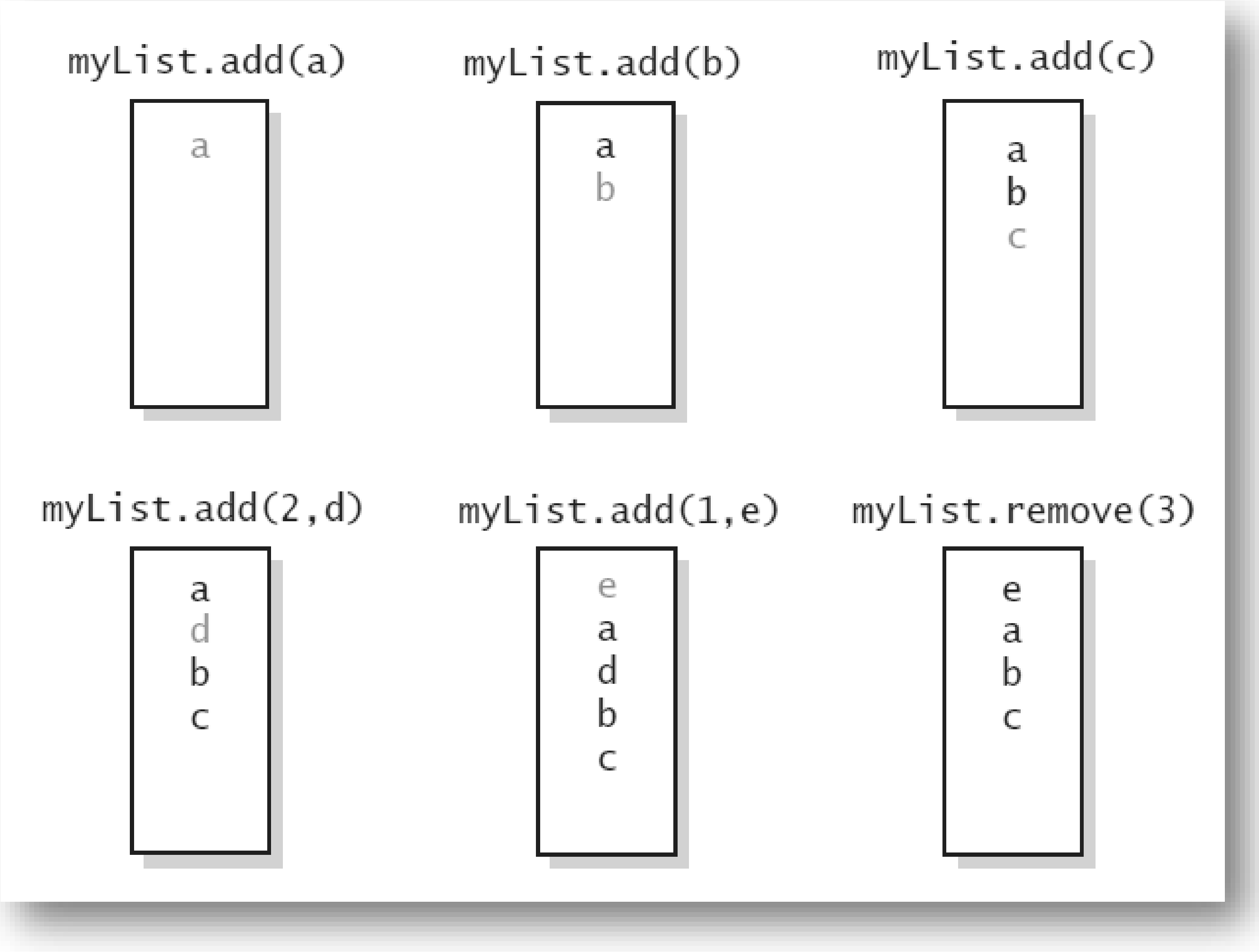
`contains(anEntry)`

`getLength()`

`isEmpty()`

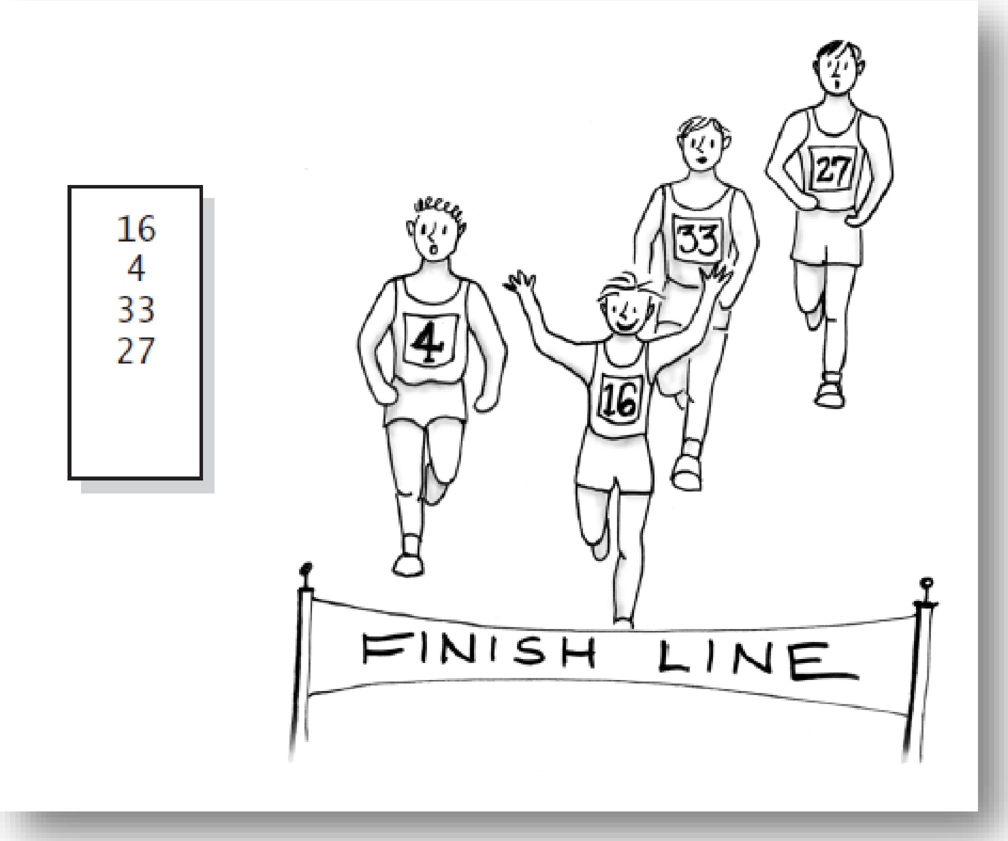
Specifications for the ADT List

The effect of ADT list operations
on an initially empty list



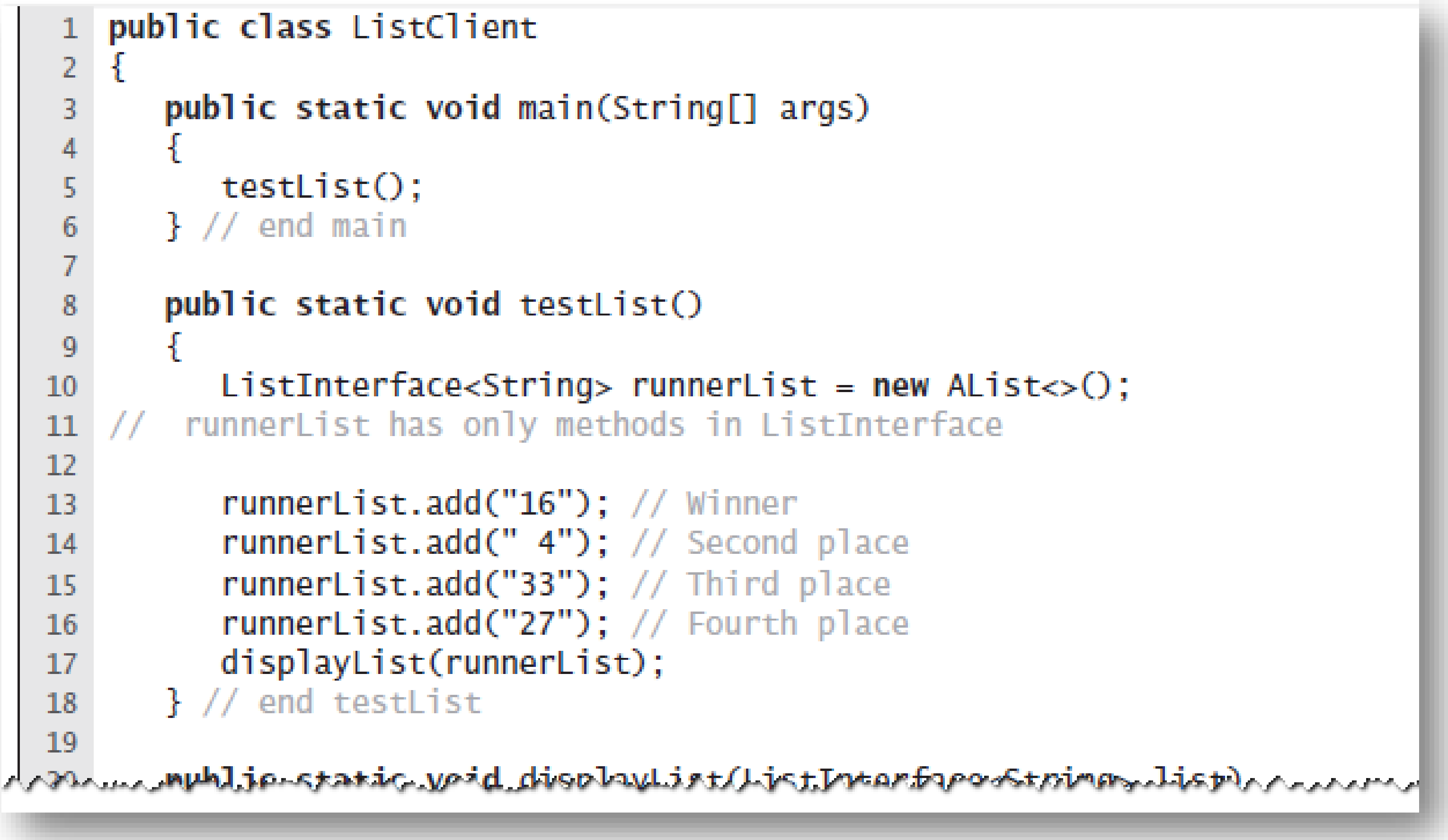
Using the ADT List

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



Using the ADT List

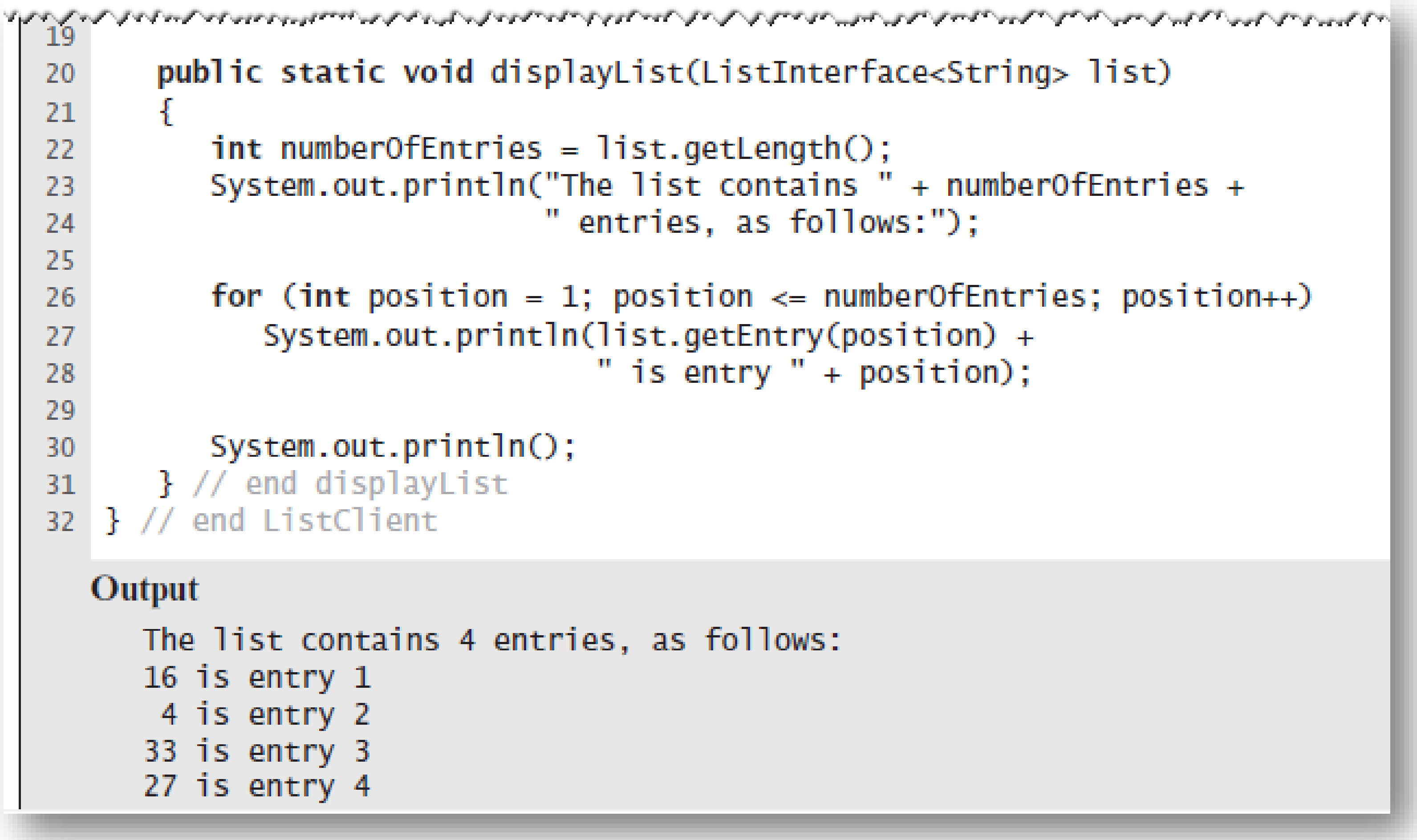
A client of a class
that implements **ListInterface**



```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`**



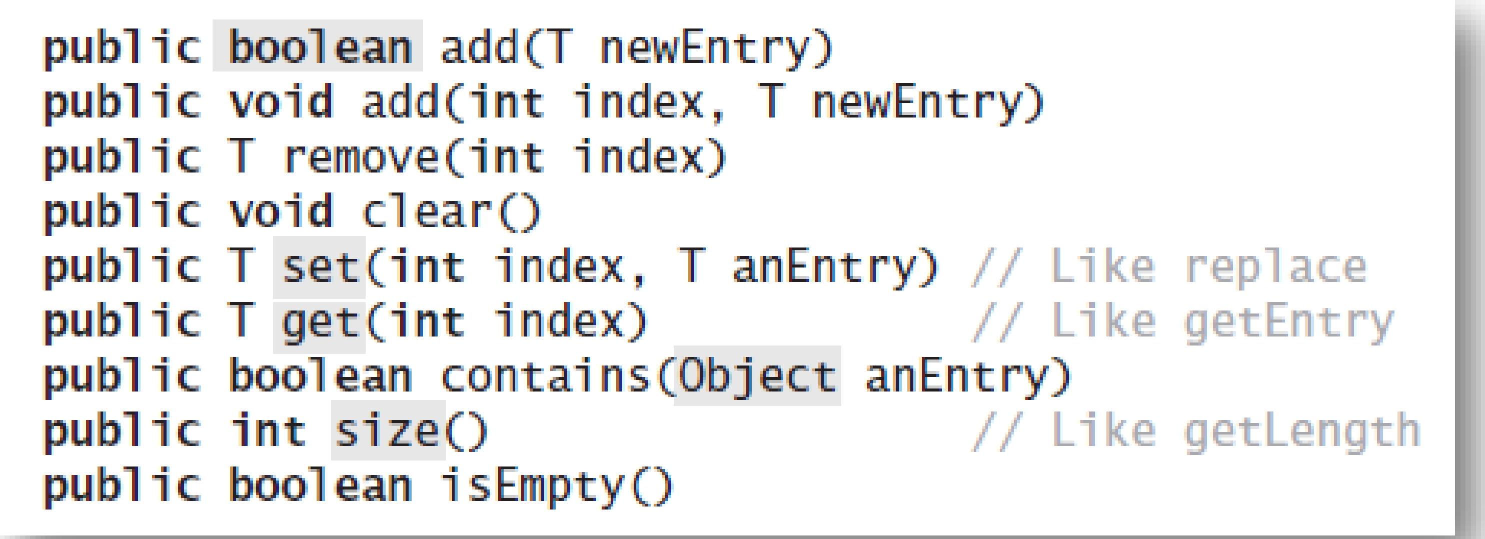
```
   public static void displayList(ListInterface<String> list)
   {
      int numberOfEntries = list.getLength();
      System.out.println("The list contains " + numberOfEntries +
                         " entries, as follows:");

      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



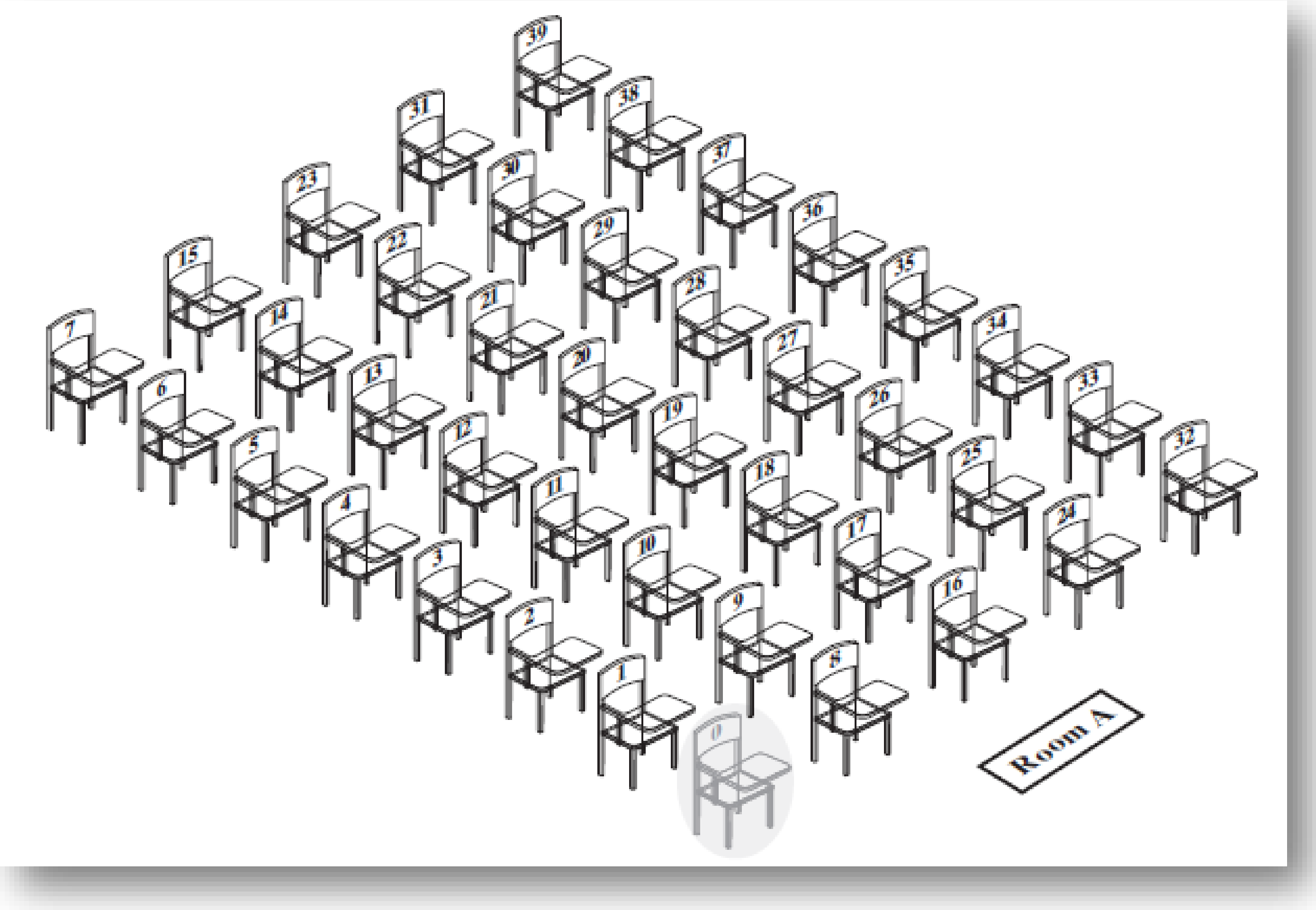
```
public boolean add(T newEntry)
public void add(int index, T newEntry)
public T remove(int index)
public void clear()
public T set(int index, T anEntry) // Like replace
public T get(int index)            // Like getEntry
public boolean contains(Object anEntry)
public int size()                  //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

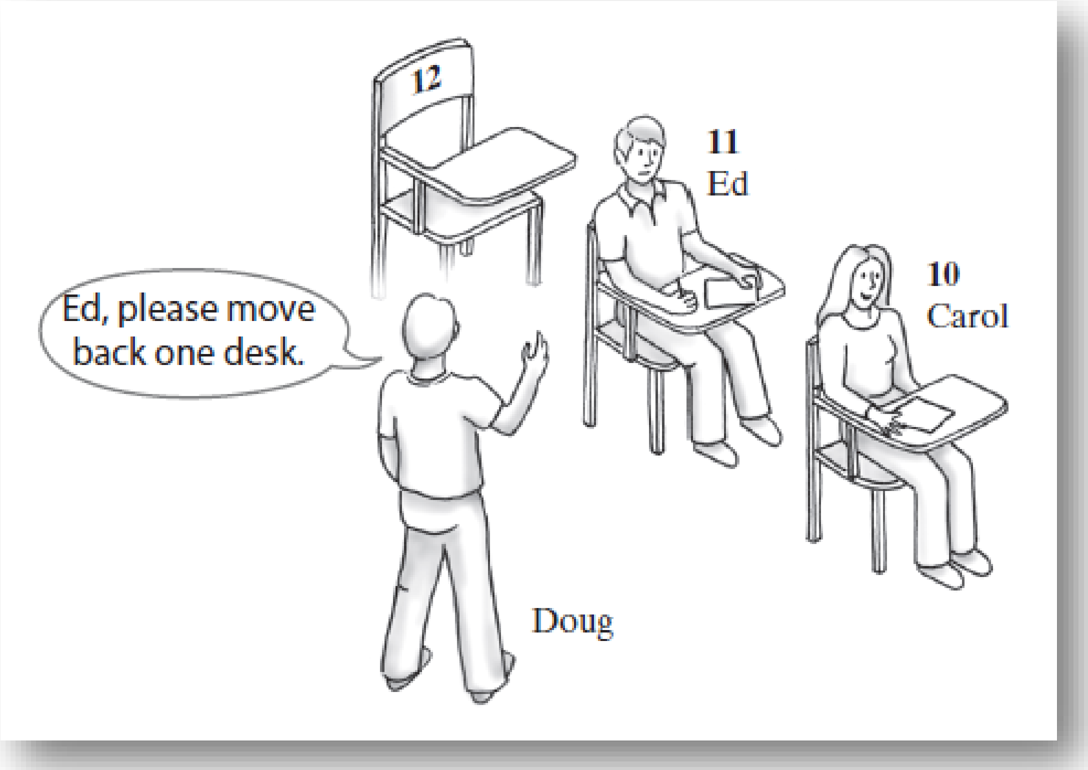
Array to Implement the ADT List

A classroom that contains desks in fixed positions



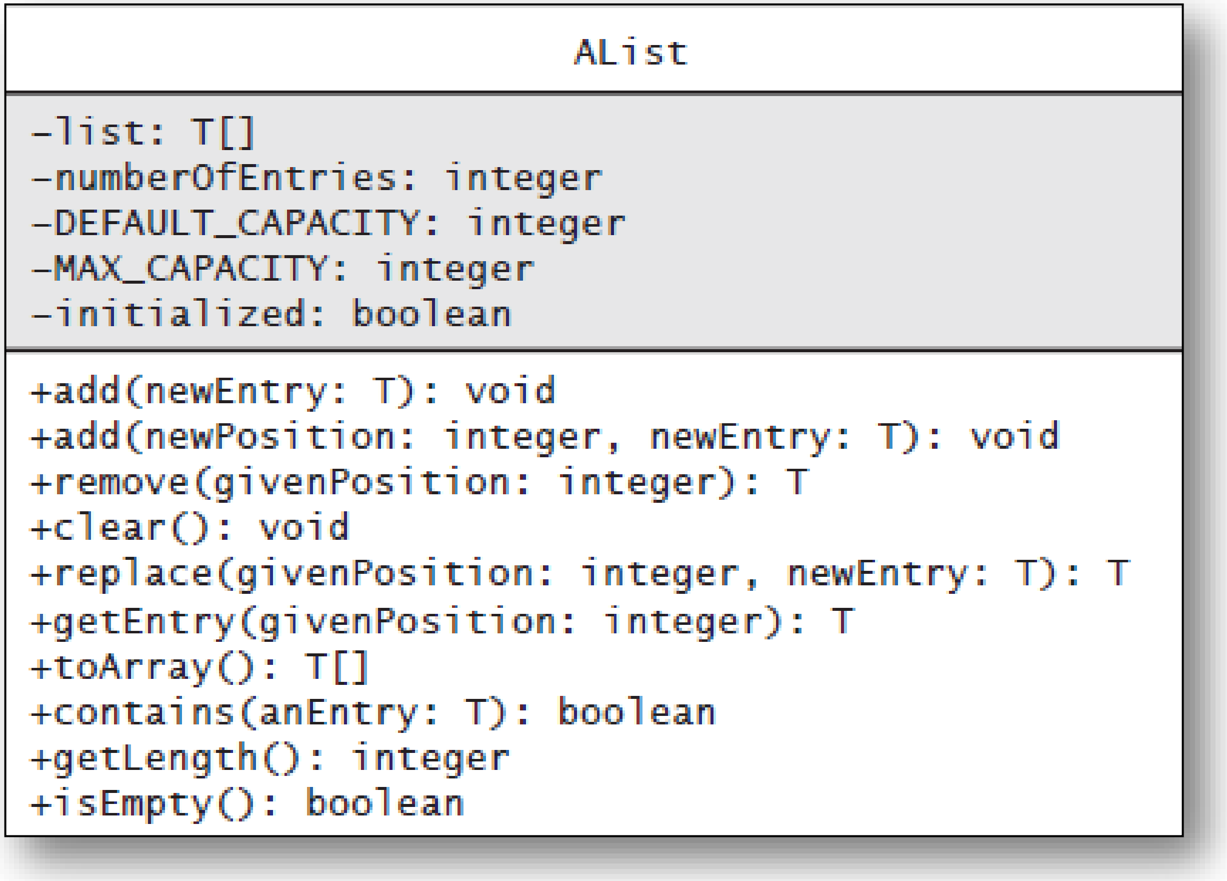
Array to Implement the ADT List

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



Array to Implement the ADT List

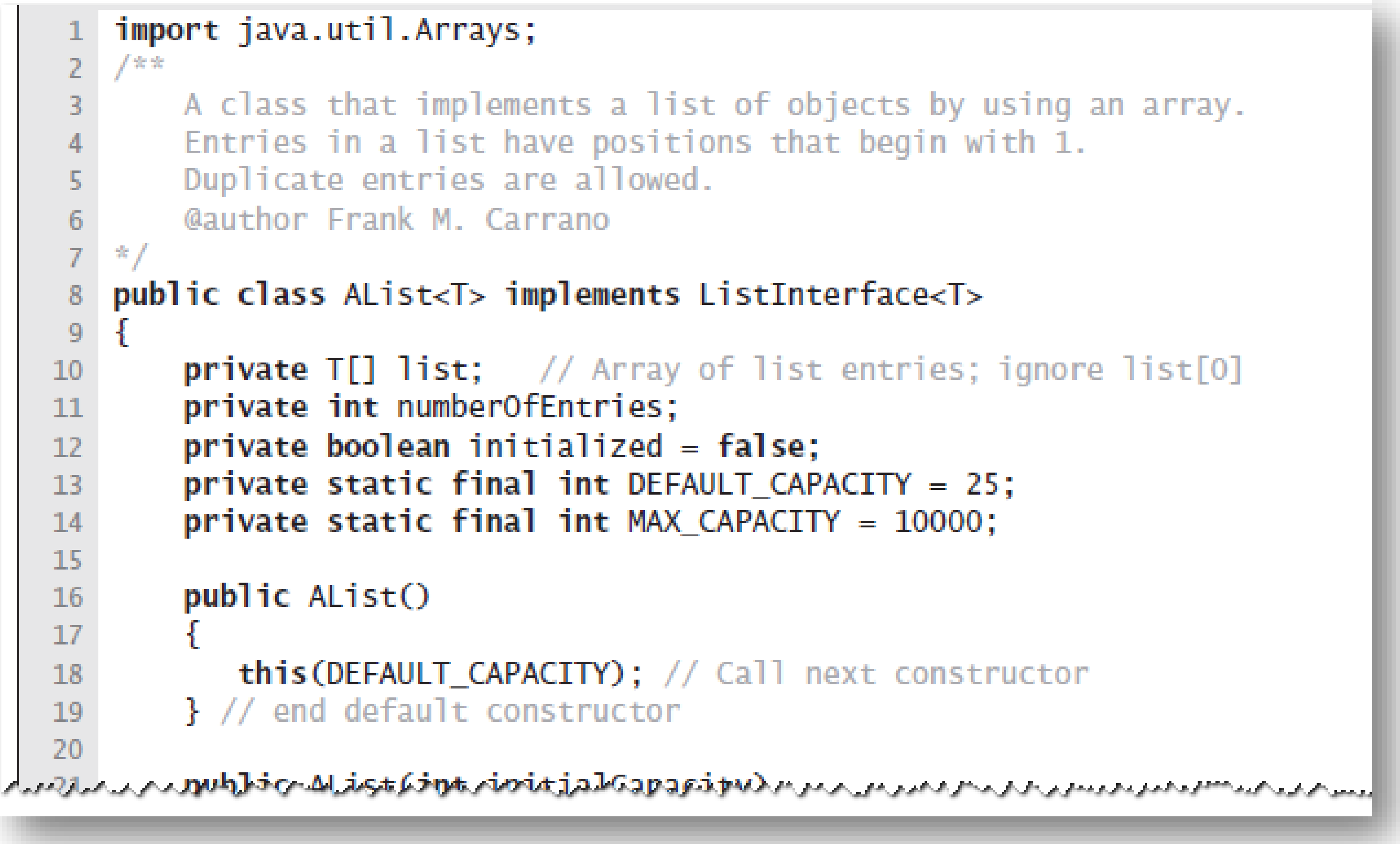
UML notation for the class **AList**



```
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

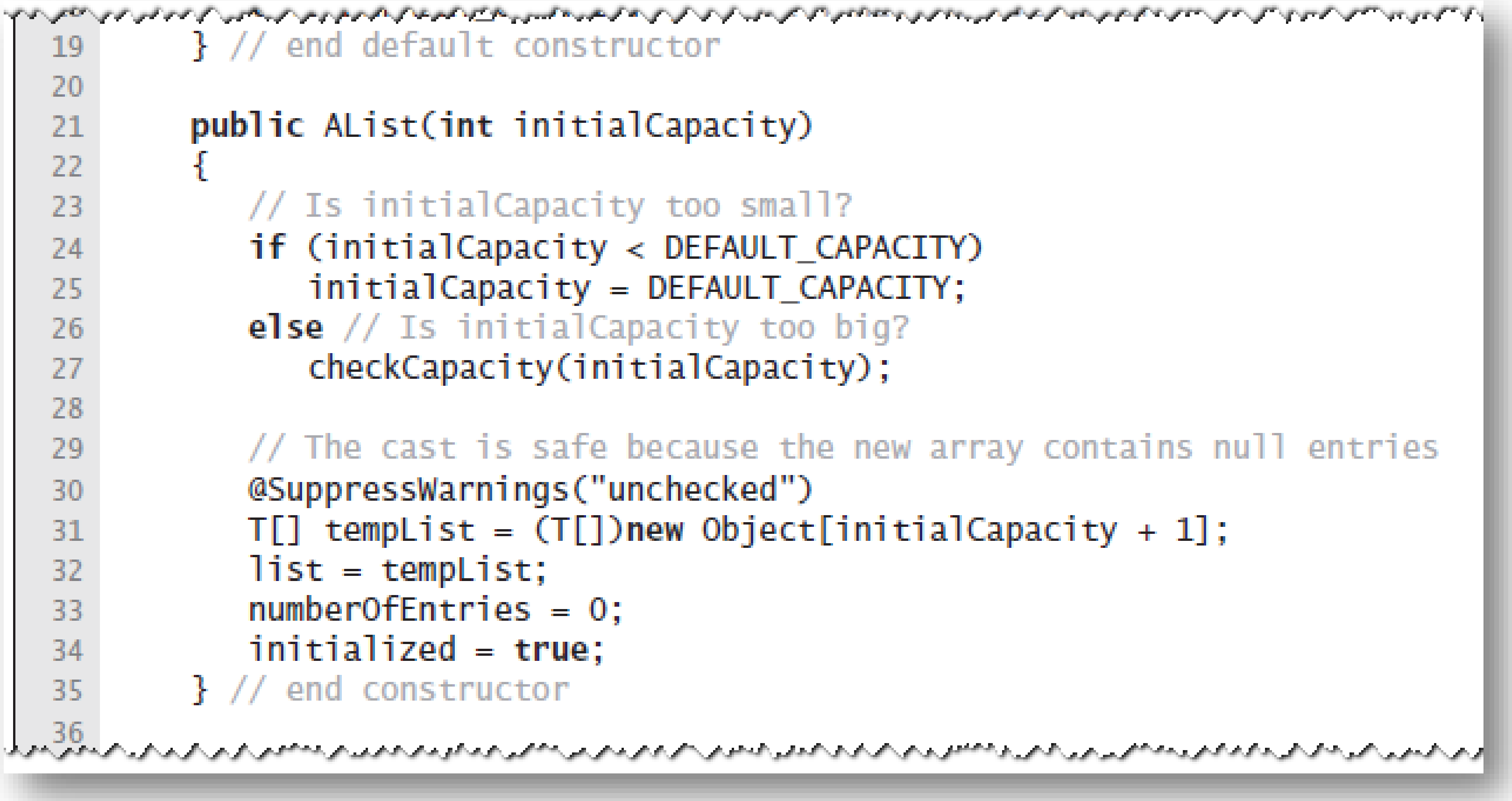
The class **`AList`**



```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

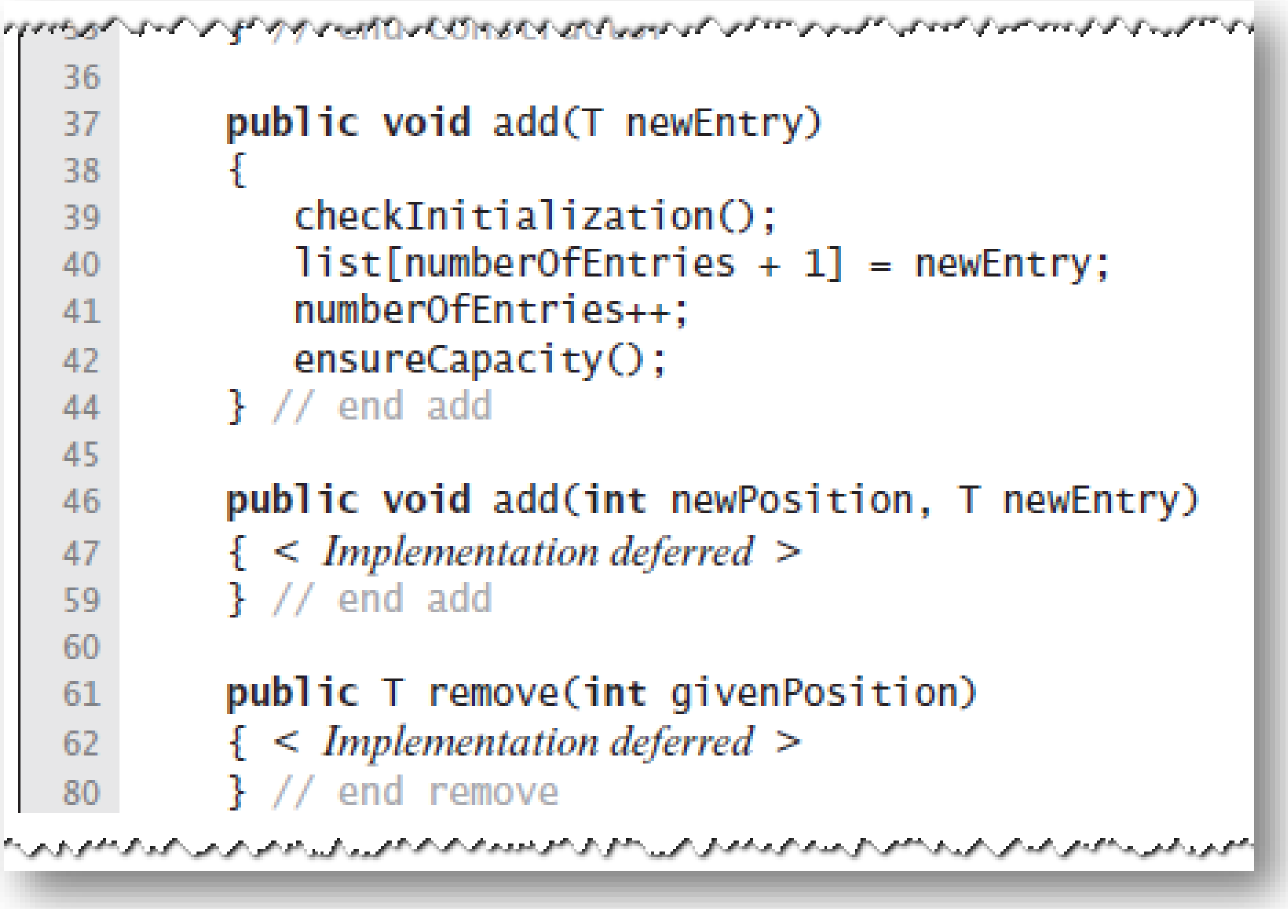
The class **AList**



```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

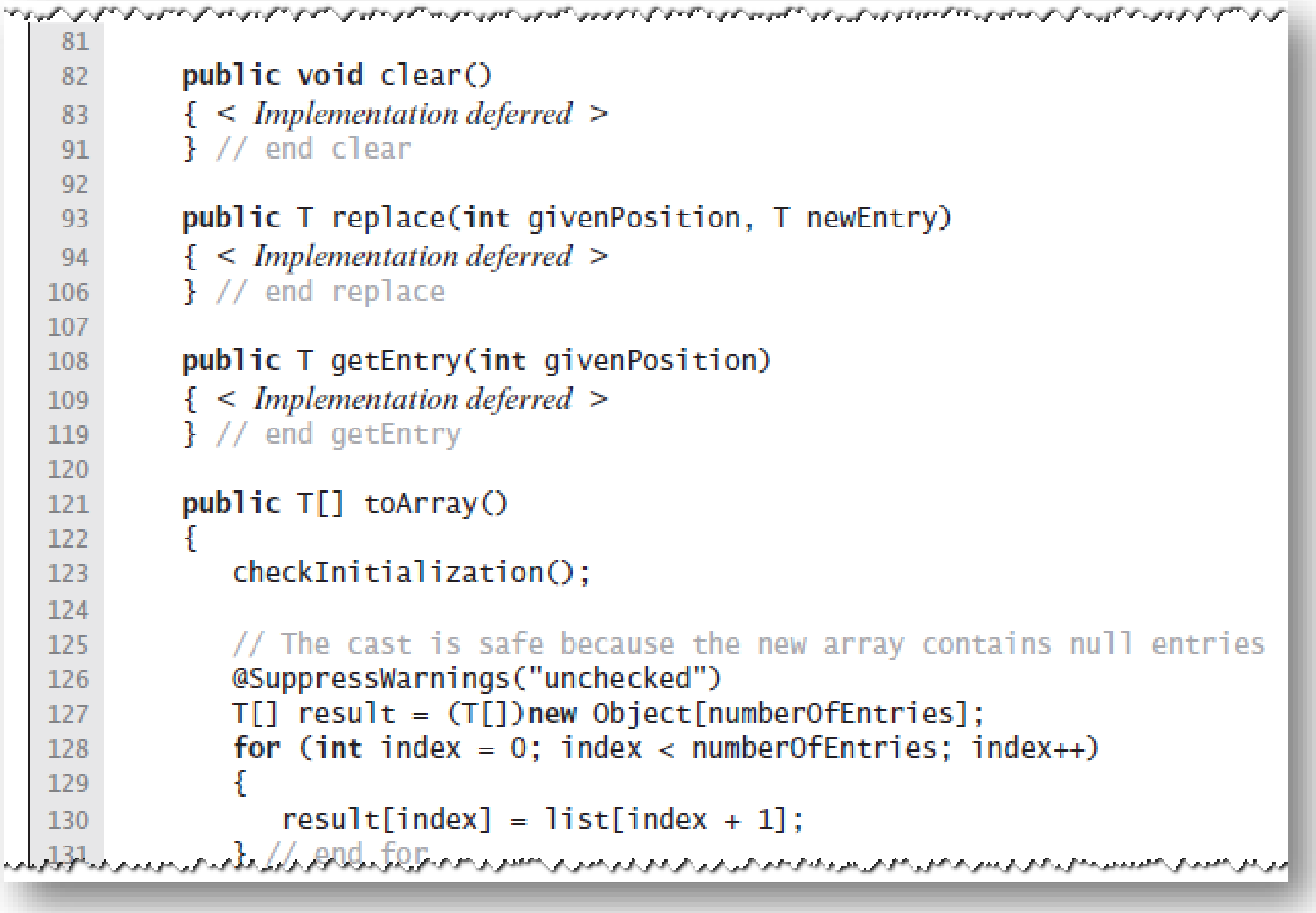
The class **`AList`**



```
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

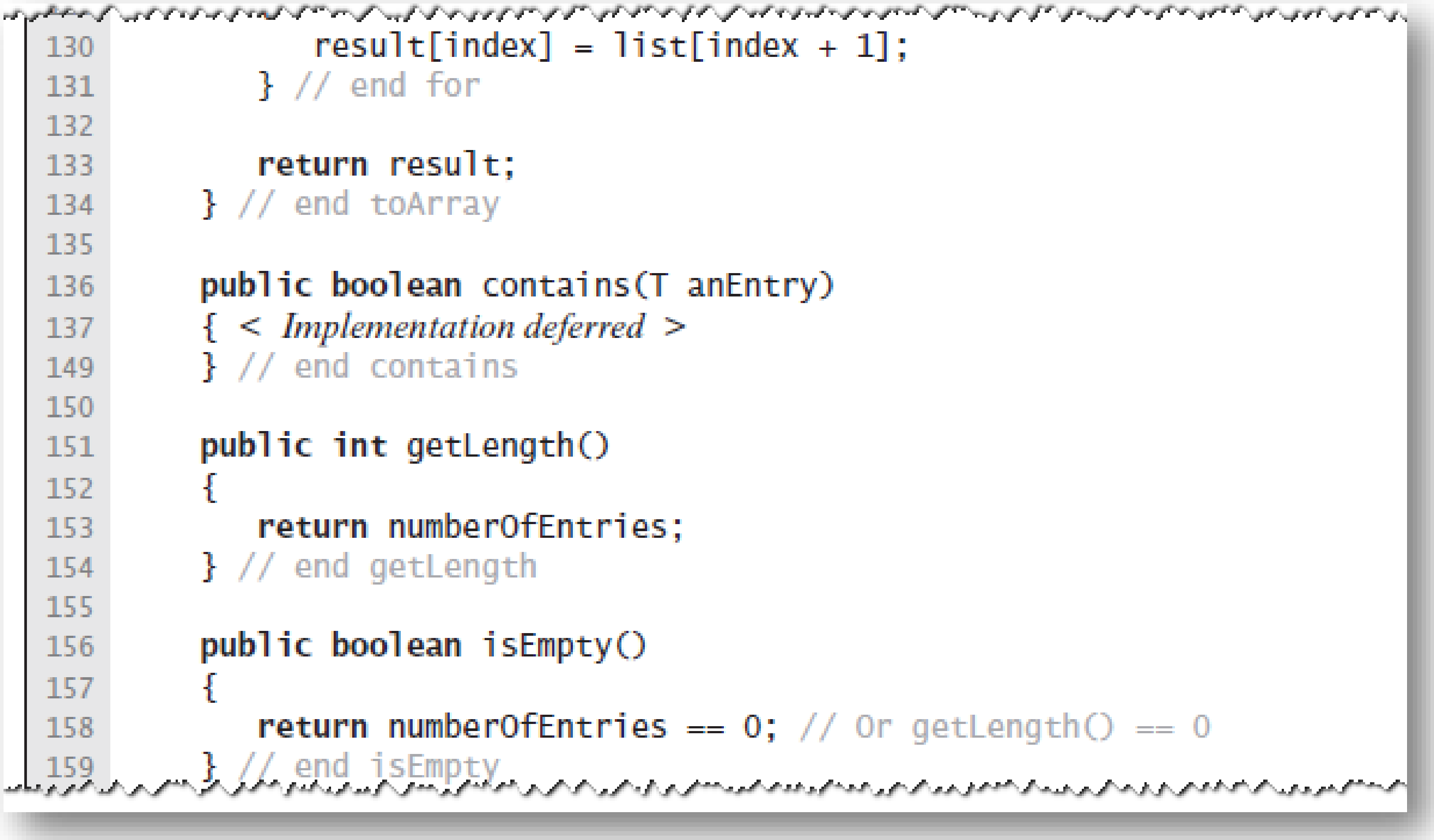
The class **AList**



```
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

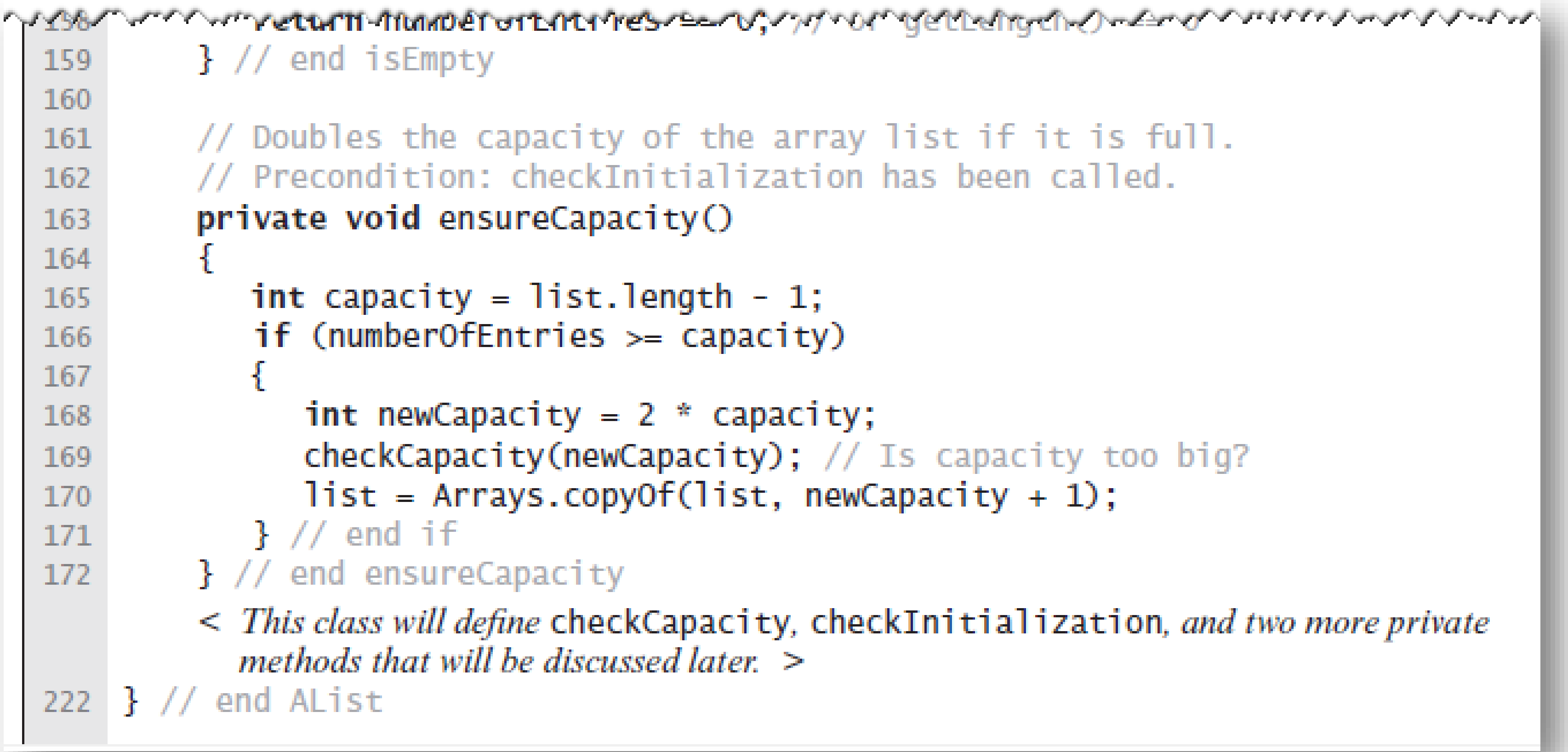
The class **AList**



```
         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

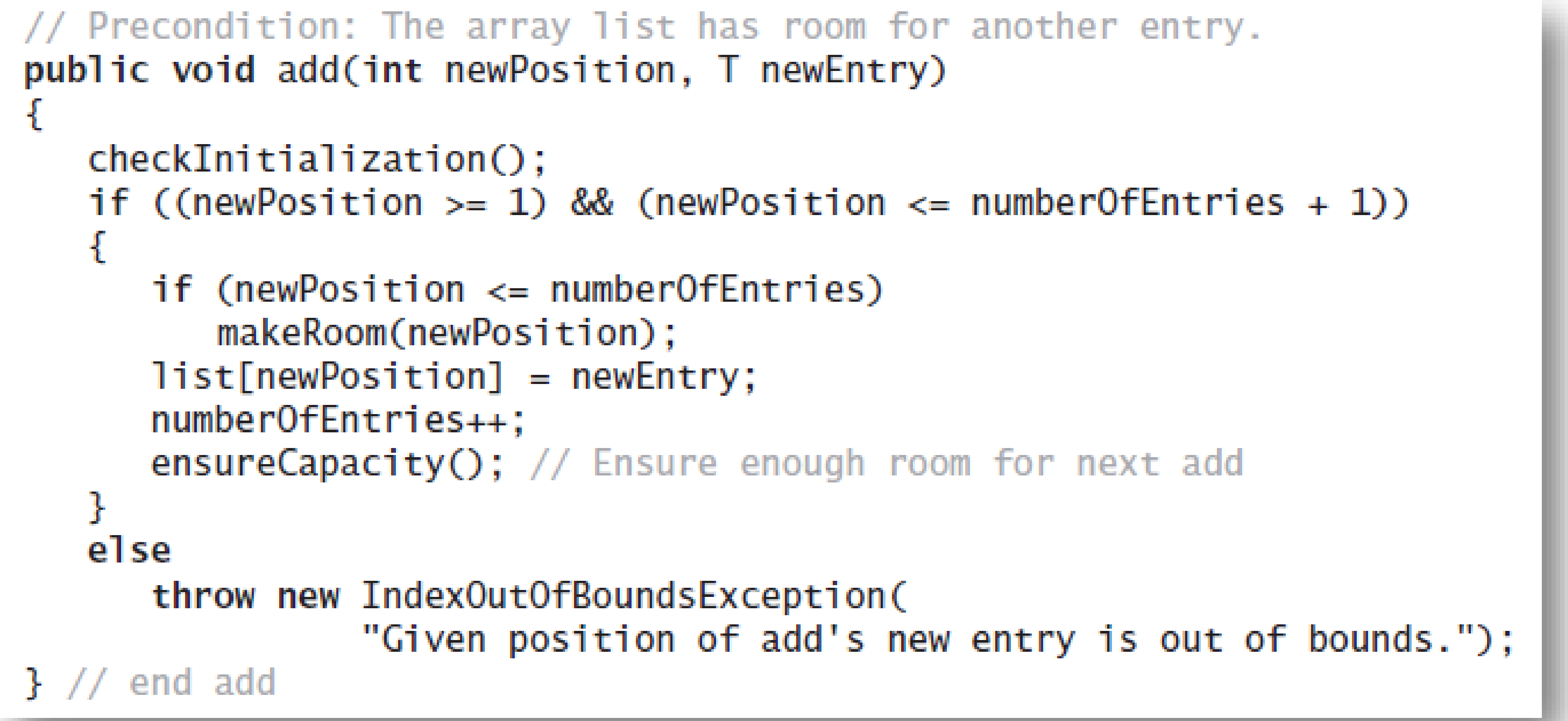
The class `AList`



```
      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

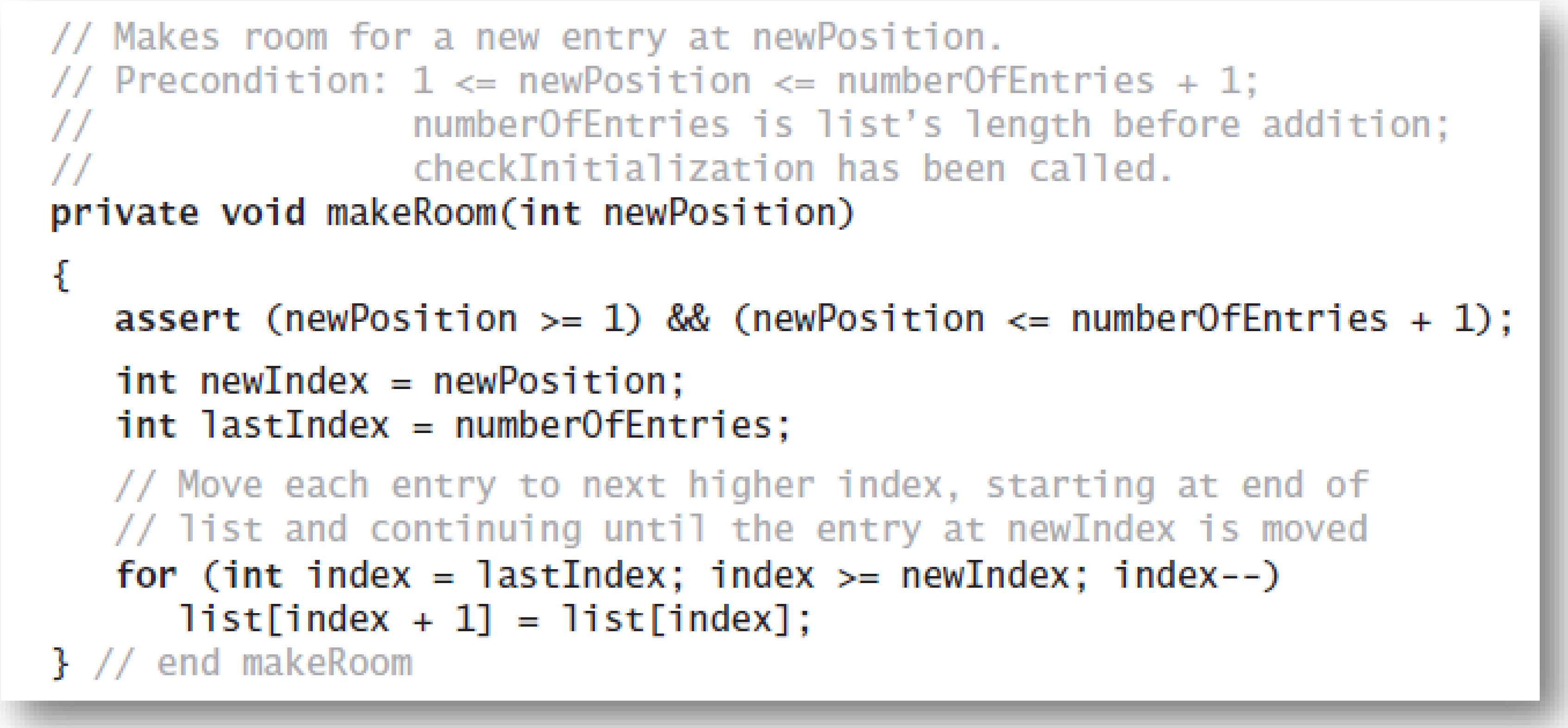
- Implementation of **add** uses a private method **makeRoom** to handle the details of moving data within the array



```
// 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

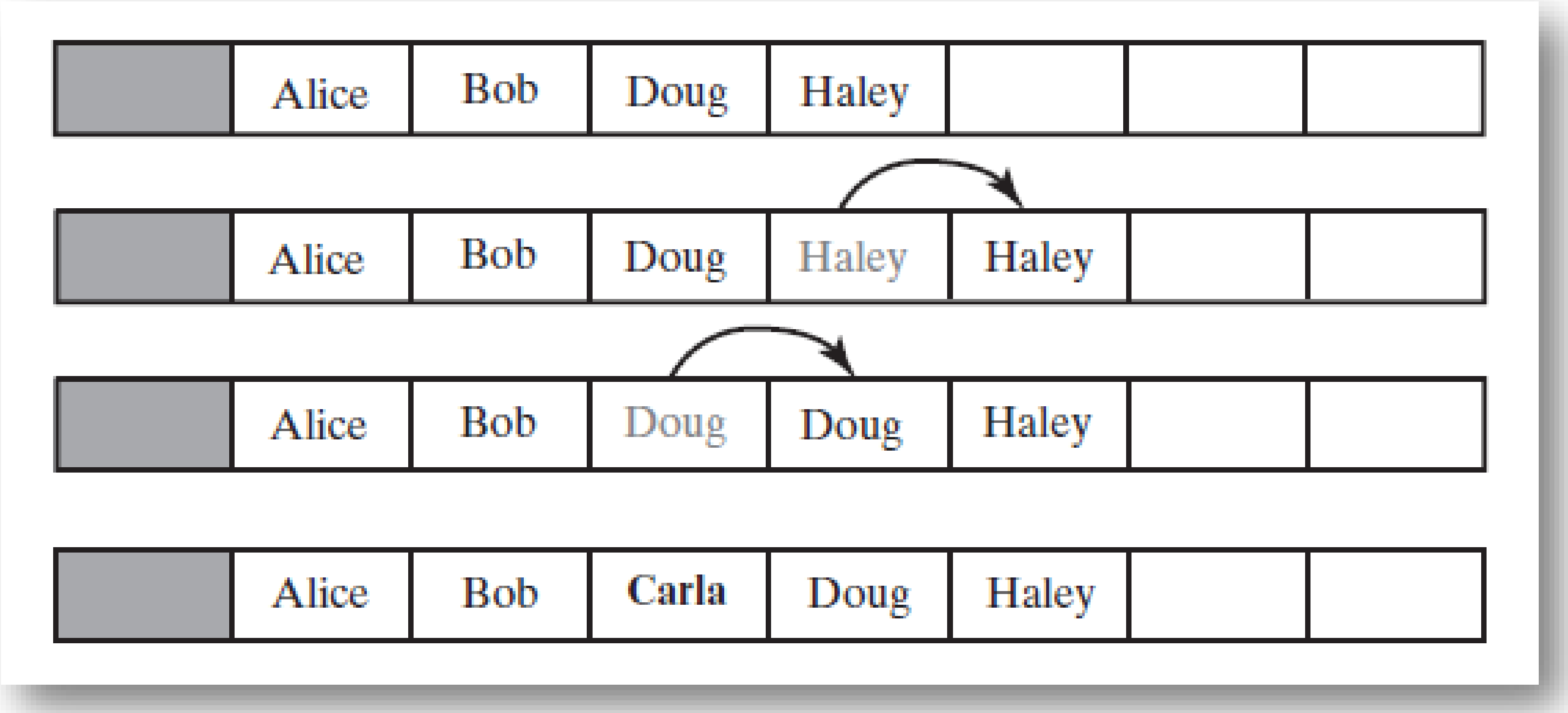
Implement the private method **`makeRoom`**



```
// 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

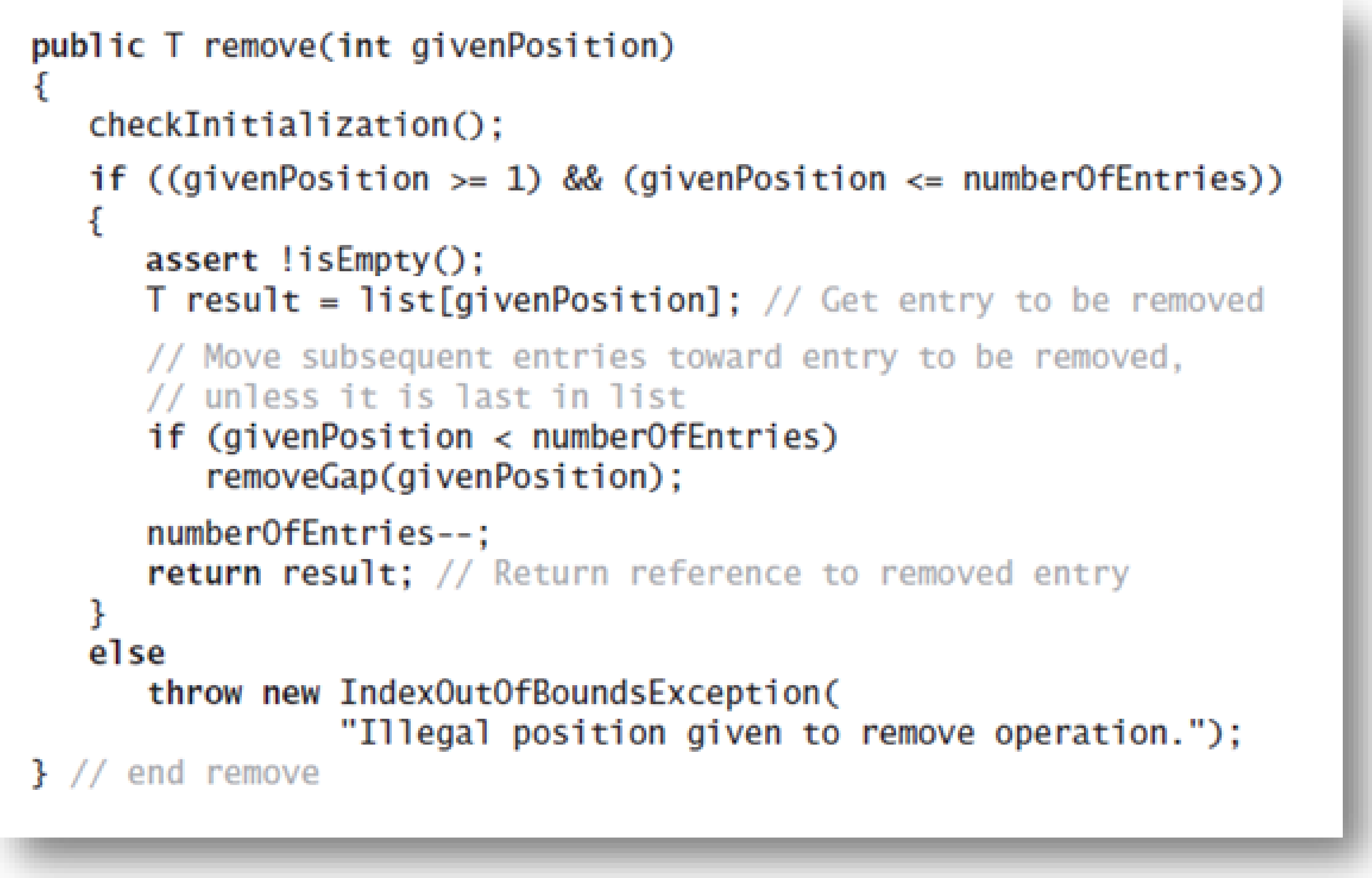
Making room to insert
Carla as the third entry in an array



	Alice	Bob	Doug	Haley			
	Alice	Bob	Doug	Haley	Haley		
	Alice	Bob	Doug	Doug	Haley		
	Alice	Bob	Carla	Doug	Haley		

Array to Implement the ADT List

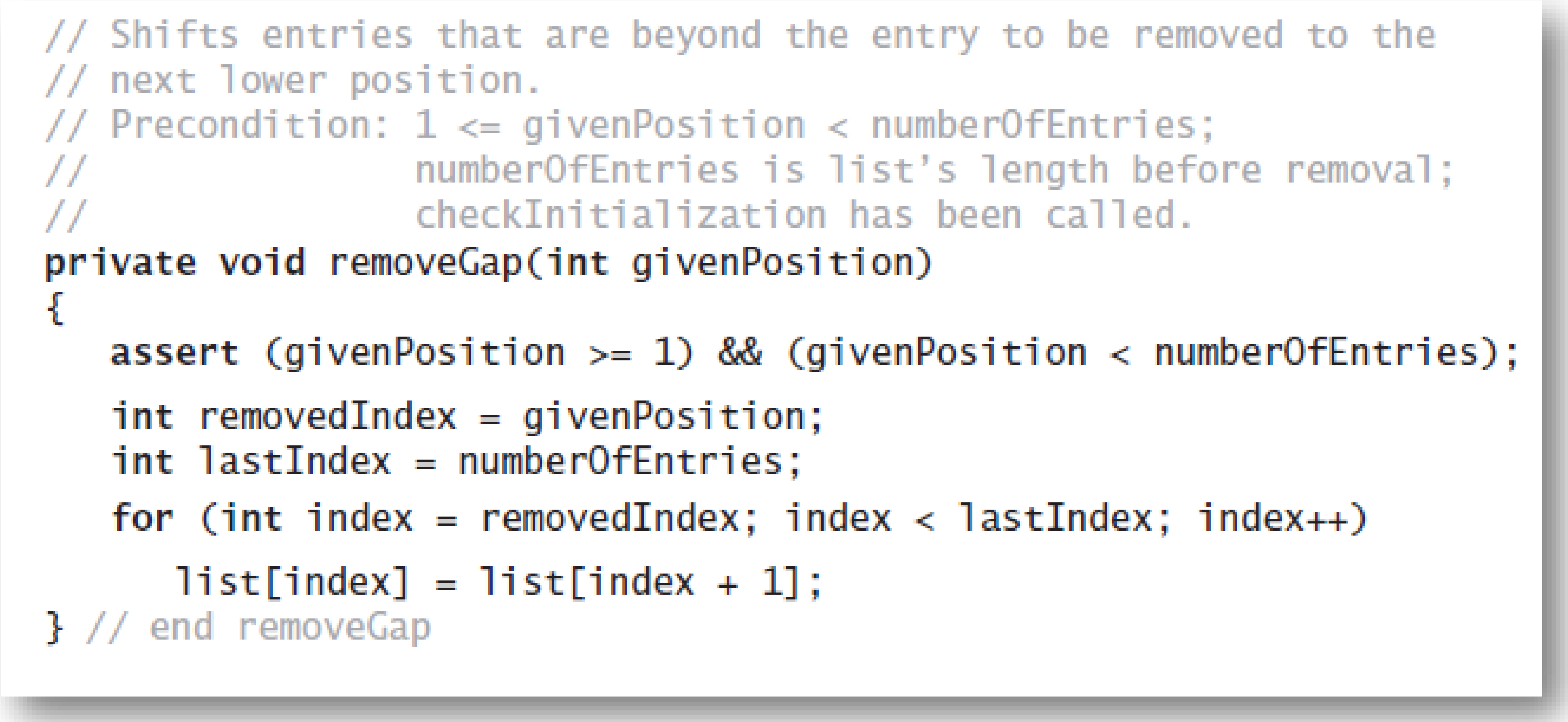
Implementation uses a private method **removeGap** to handle the details of moving data within the array.



```
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

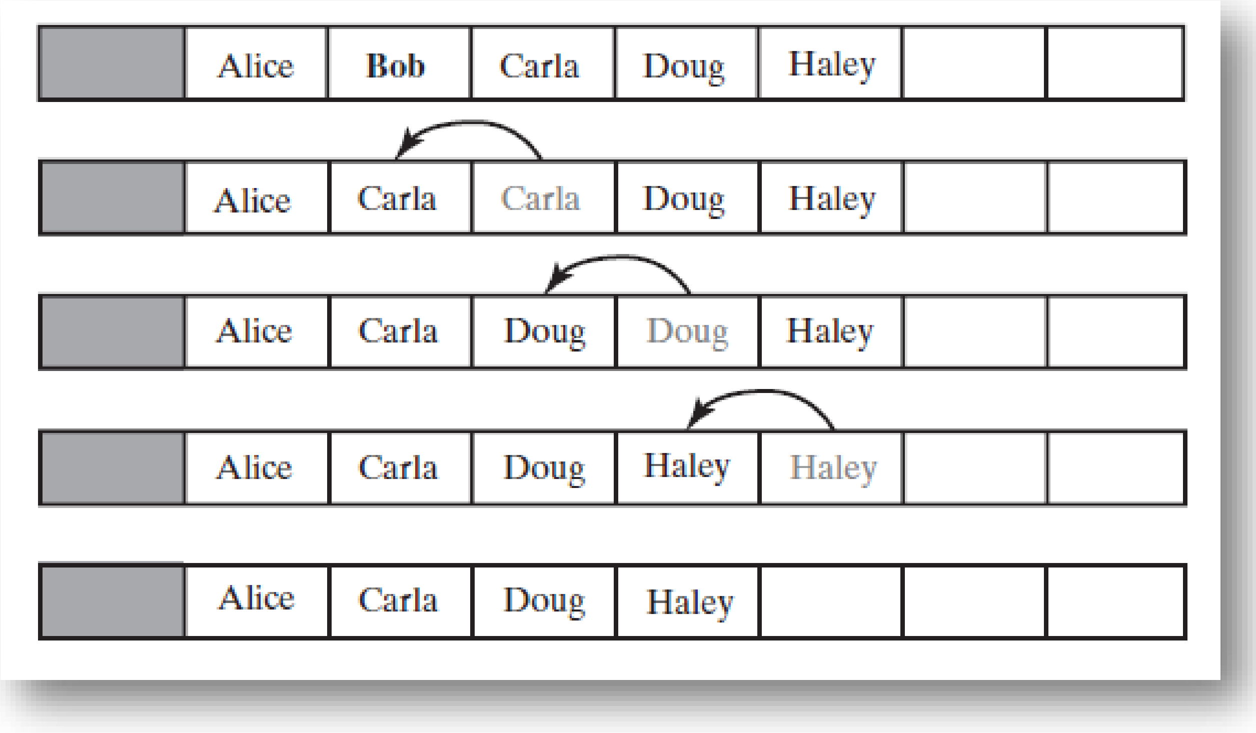
Method **`removeGap`** shifts
list entries within the array



```java
// 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

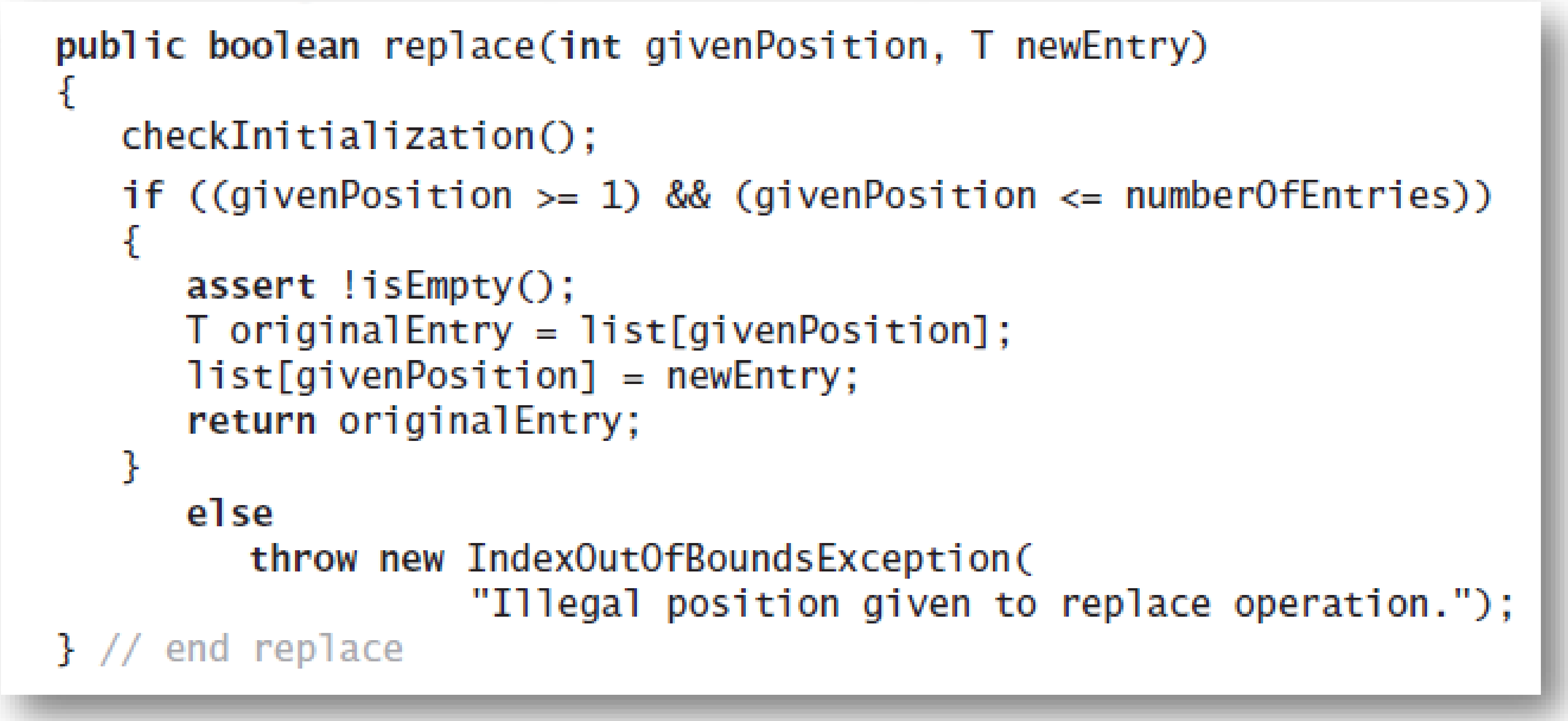
Removing Bob by shifting array entries



	Alice	Bob	Carla	Doug	Haley		
	Alice	Carla	Carla	Doug	Haley		
	Alice	Carla	Doug	Doug	Haley		
	Alice	Carla	Doug	Haley	Haley		
	Alice	Carla	Doug	Haley			

Array to Implement the ADT List

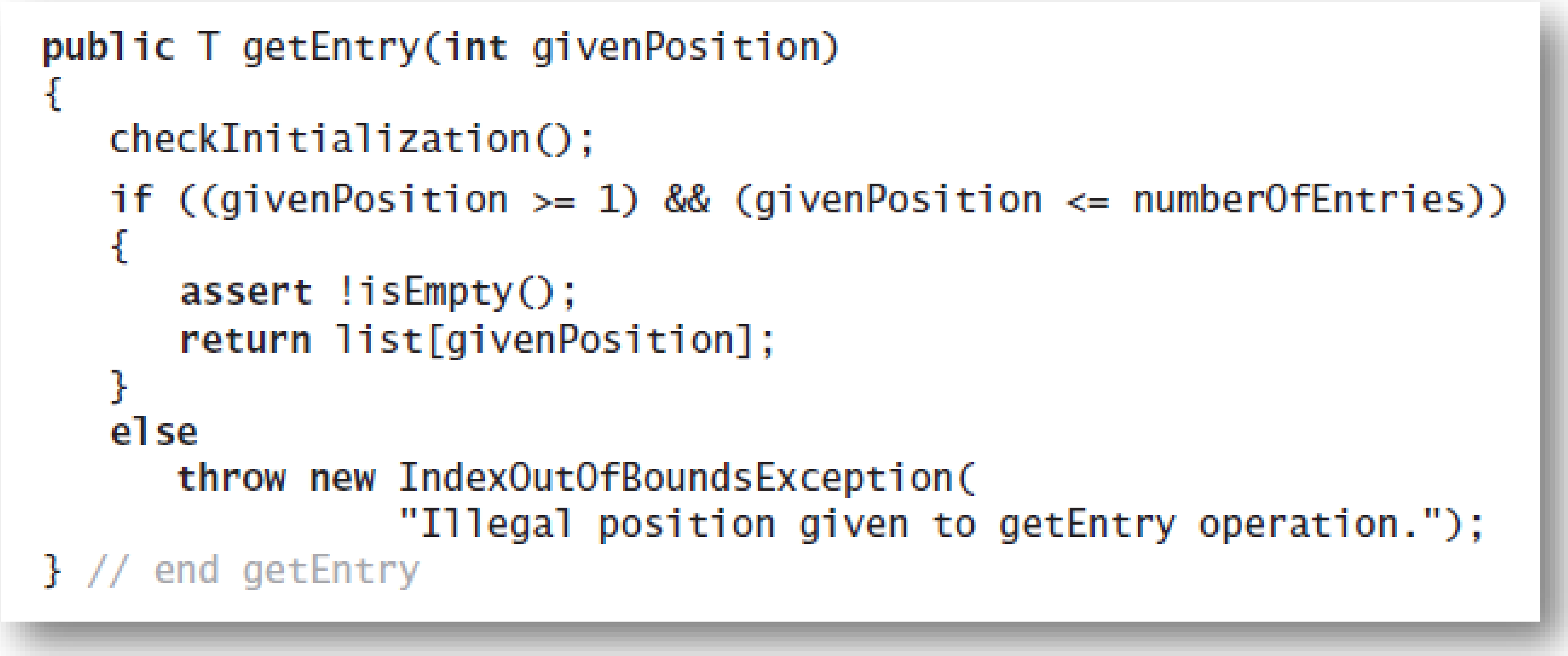
Method replace



```
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

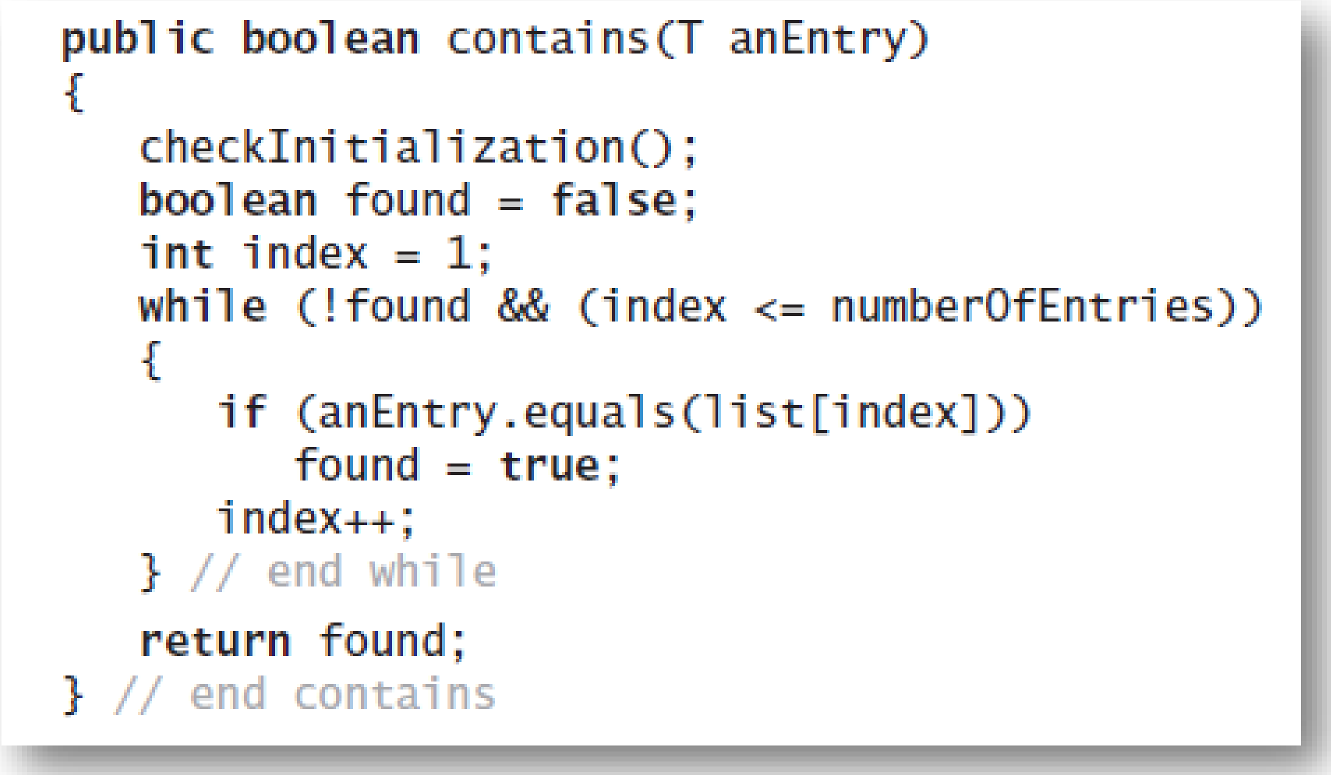
Method getEntry



```
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

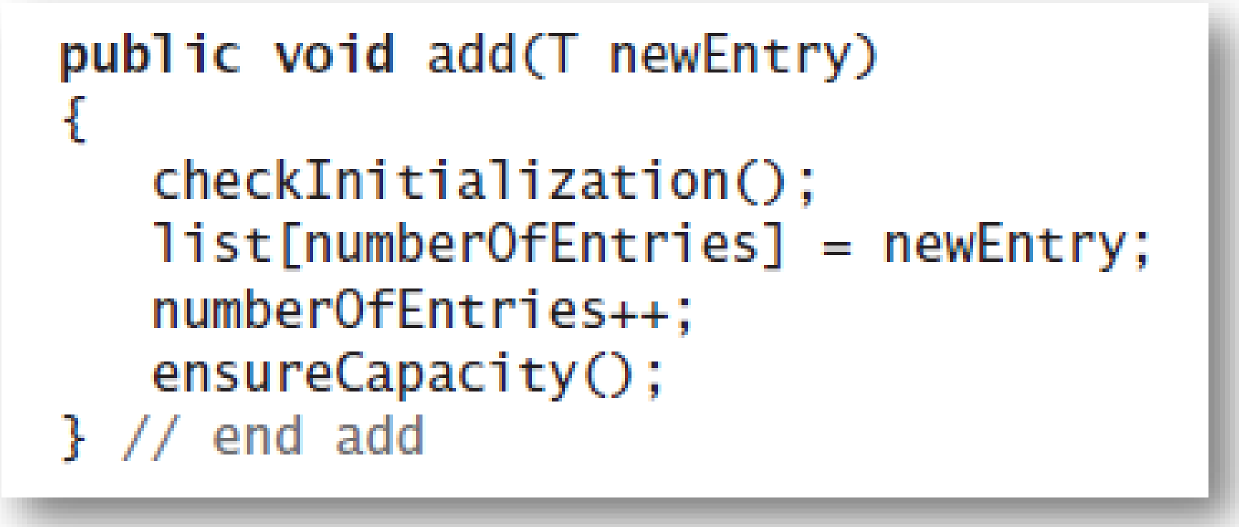
Method **contains** uses a local boolean variable to terminate the loop when we find the desired entry.



```
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

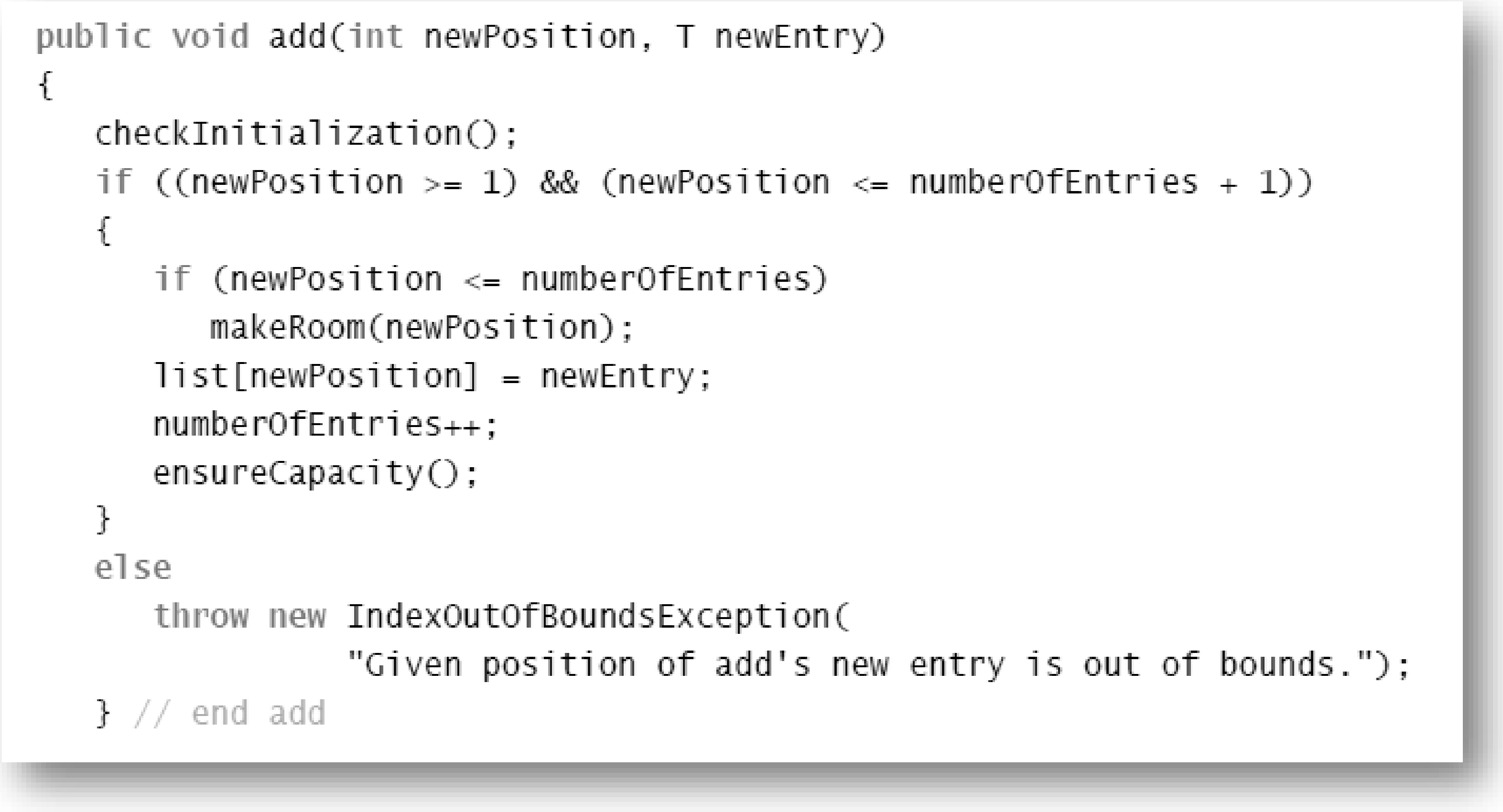
- Operation that adds a new entry to the end of a list.
- Efficiency O(1) if new if array is not resized.



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

Array to Implement the ADT List

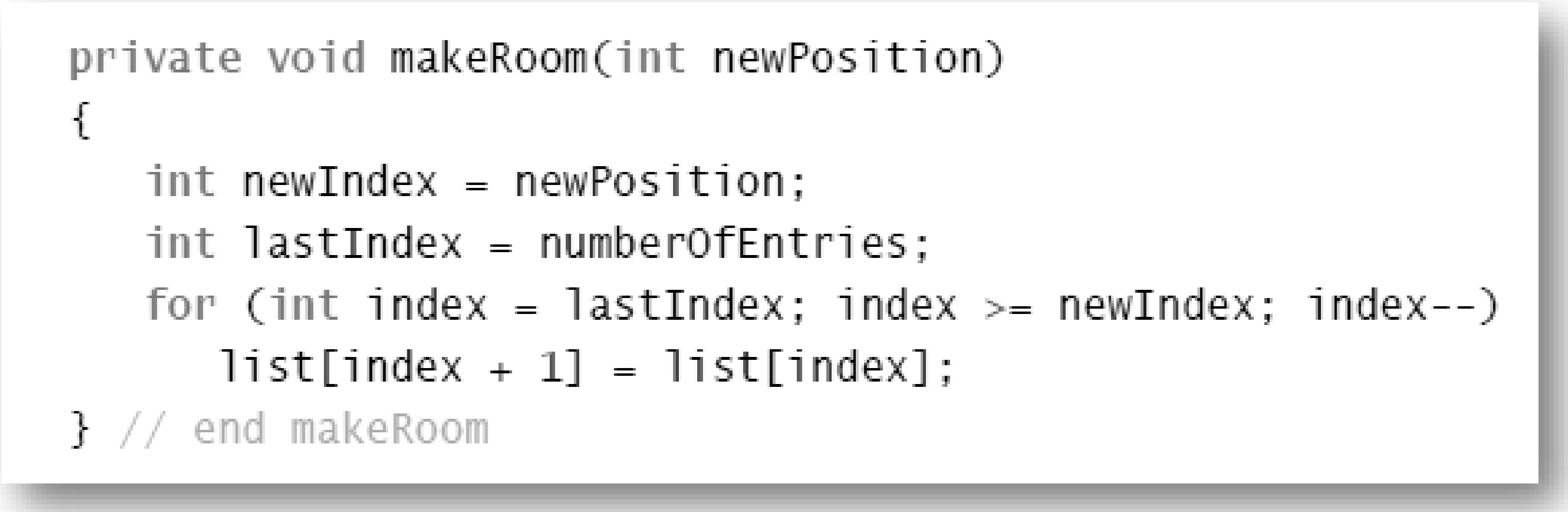
Add a new entry to a list at a client-specified position.



```
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`**.



```java
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

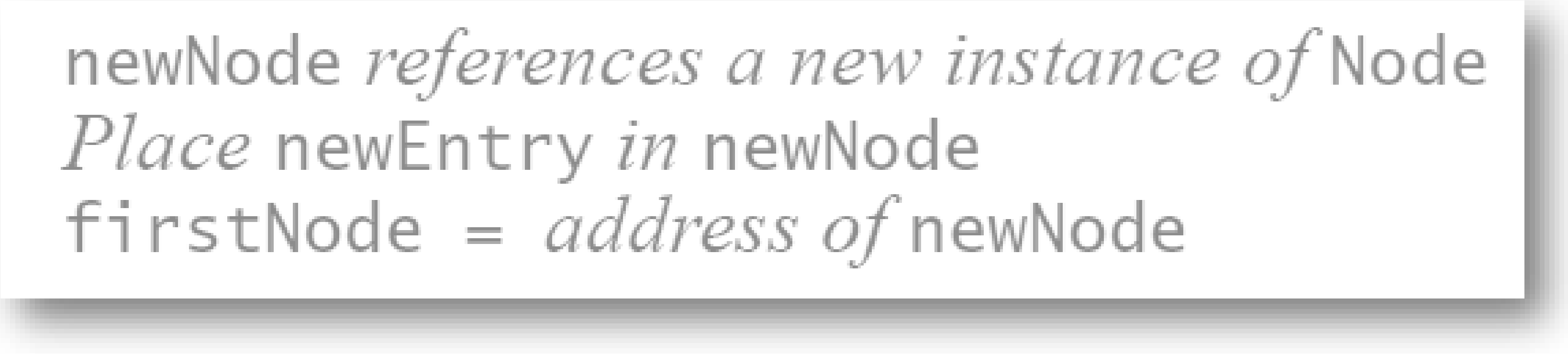
Adding a Node at Various Positions

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

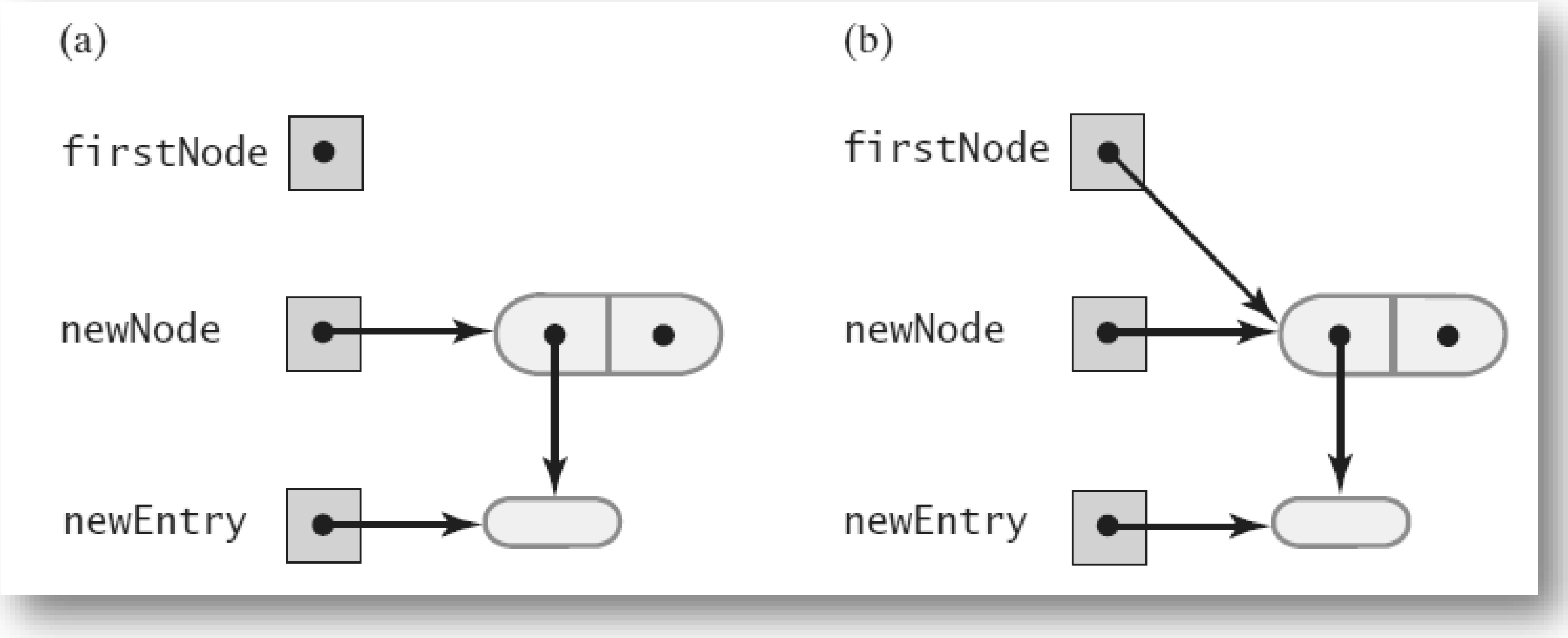
- This pseudocode establishes a new node for the given data



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

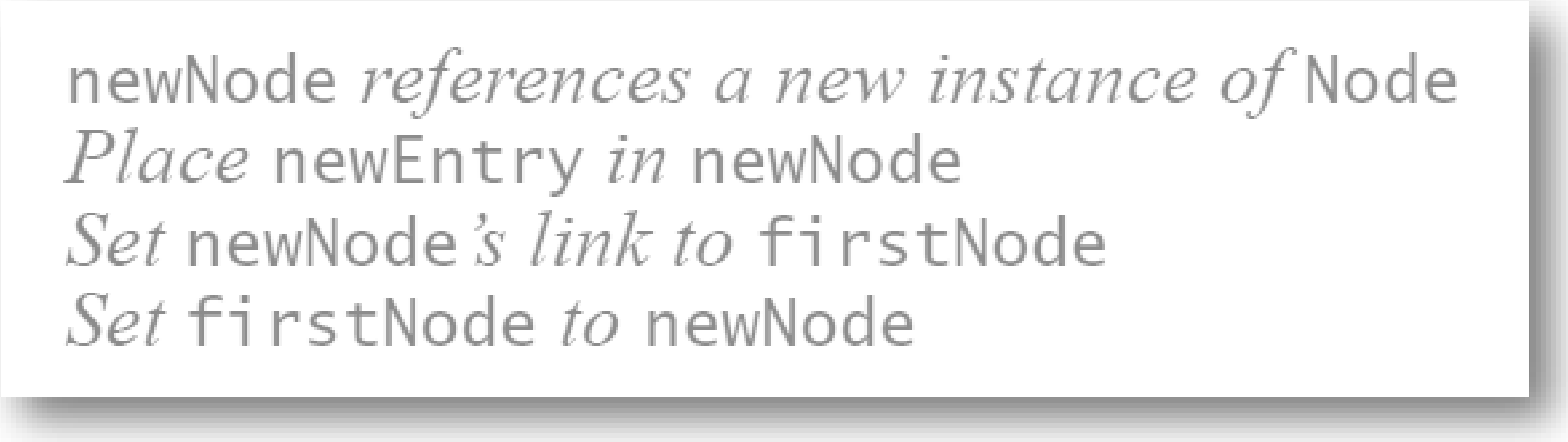
Adding a Node

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



Adding a Node

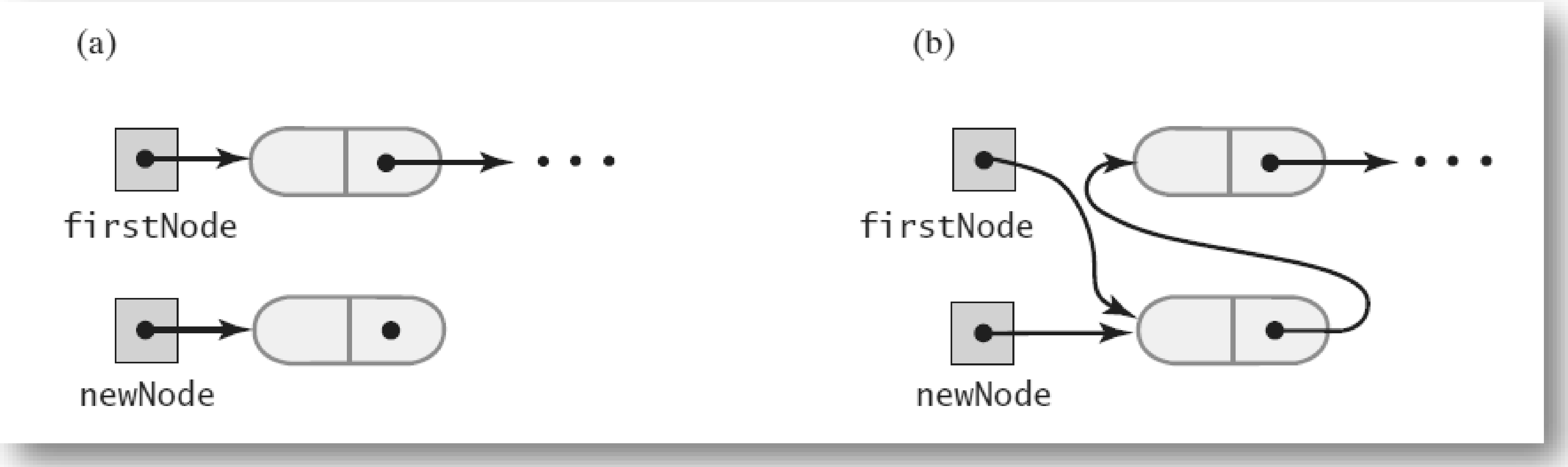
This pseudocode describes the steps needed to add a node to the beginning of a chain.



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

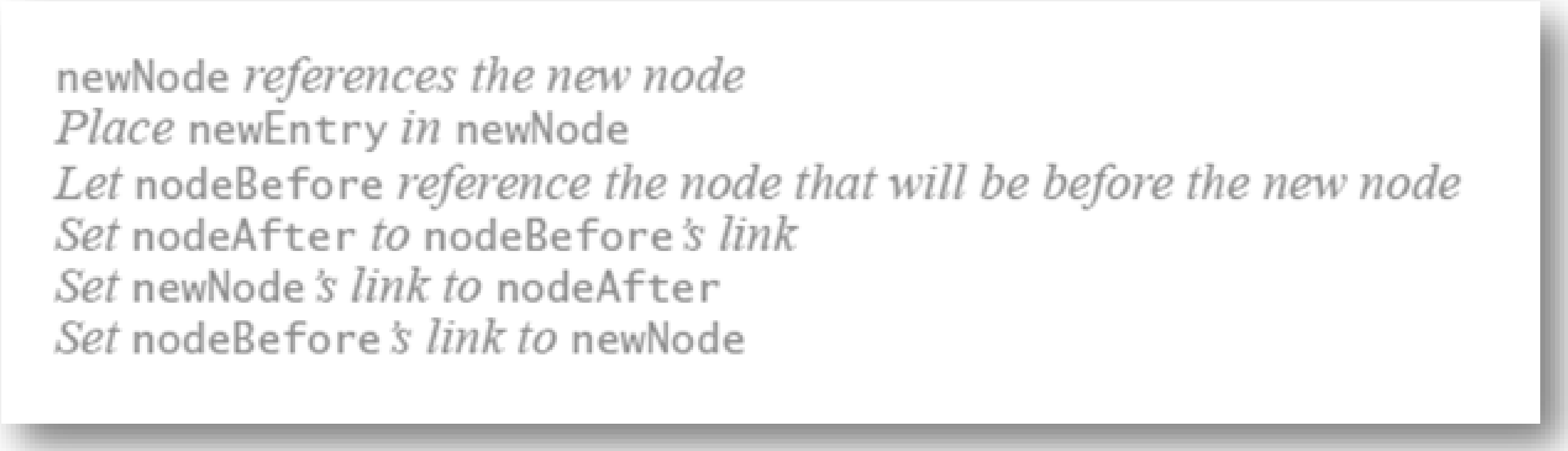
Adding a Node

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

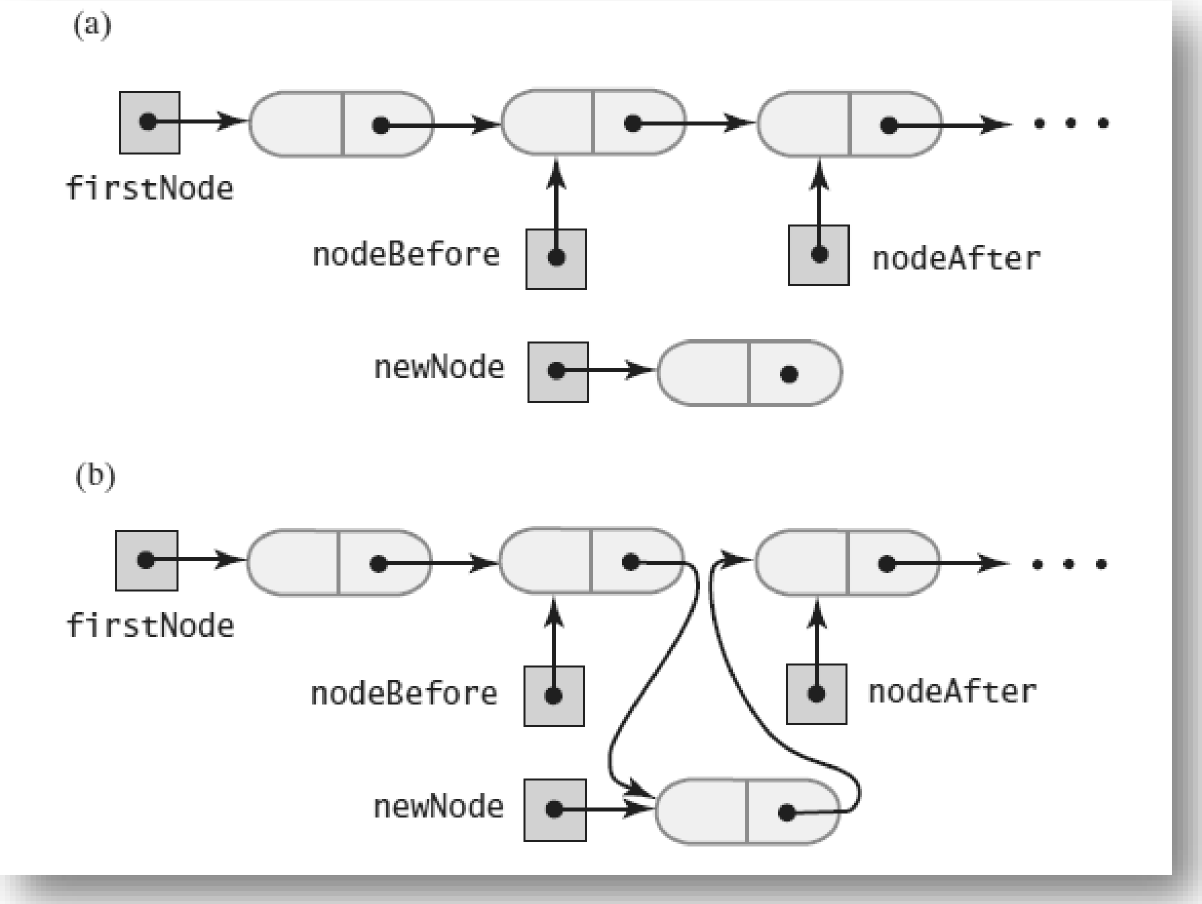
Pseudocode to add a node to a chain between two existing, consecutive nodes



```
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
```

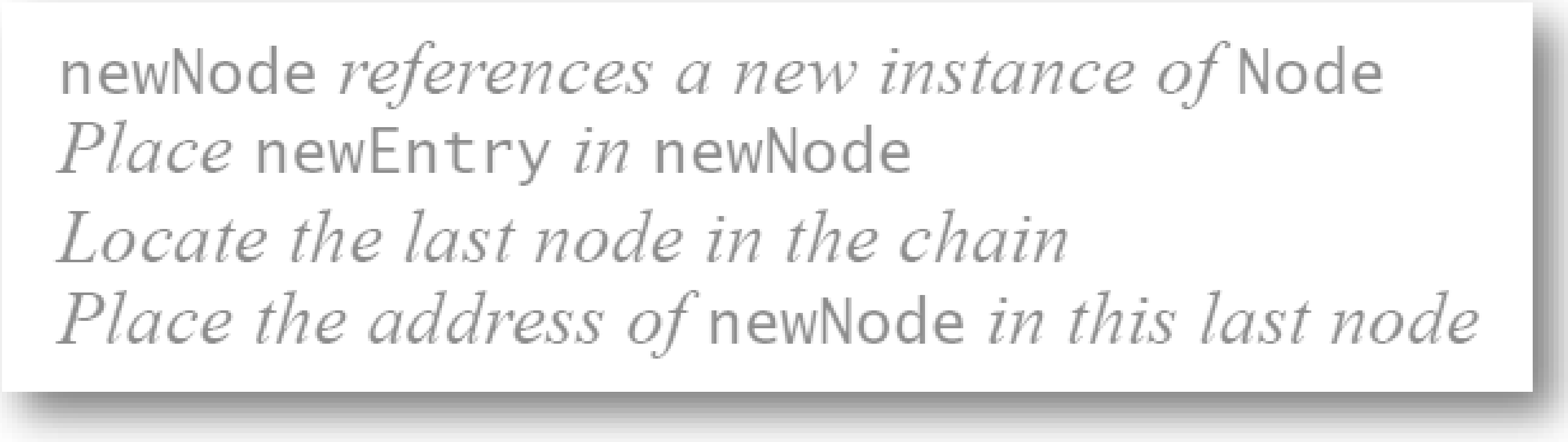
Adding a Node

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.

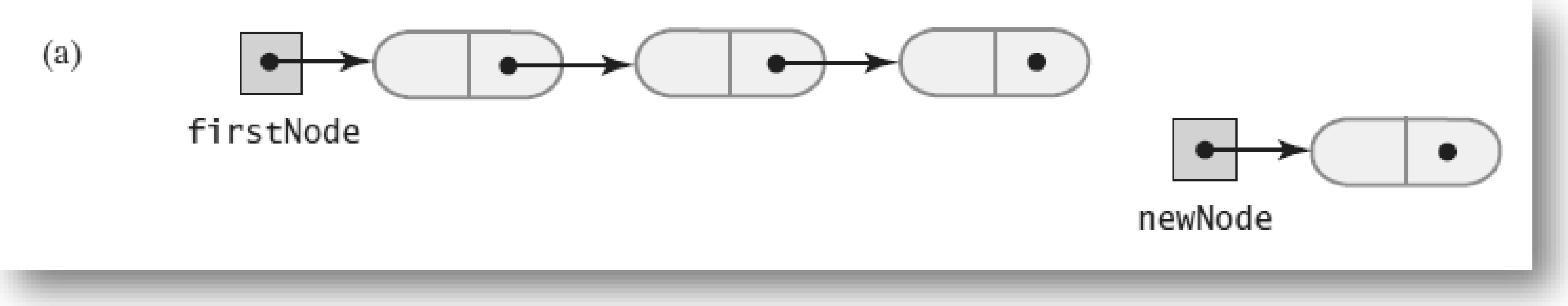


```
newNode references a new instance of Node
Place newEntry in newNode
Locate the last node in the chain
Place the address of newNode in this last node
```

Adding a Node

A chain of nodes

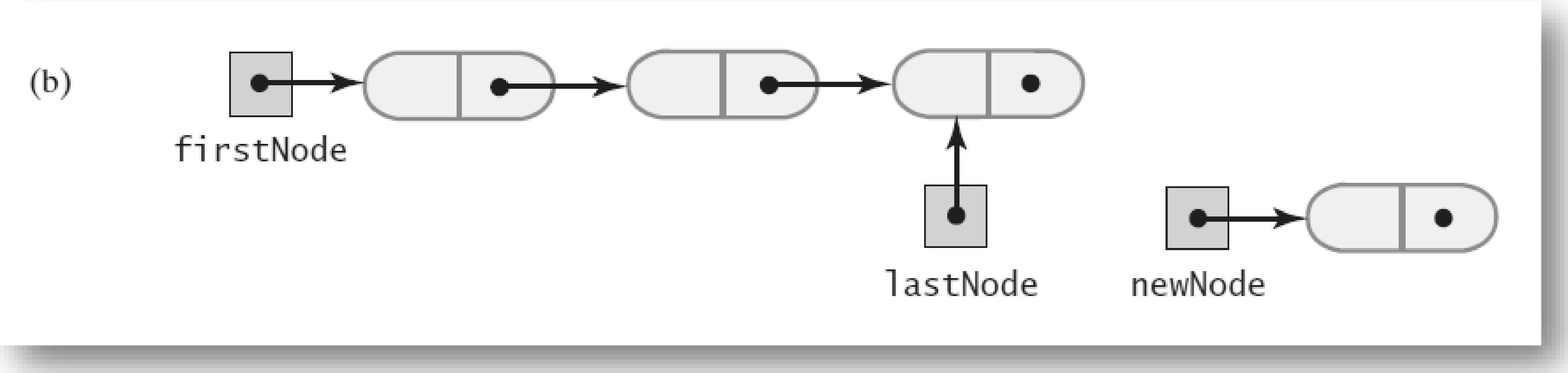
(a) prior to adding a node at the end



Adding a Node

A chain of nodes

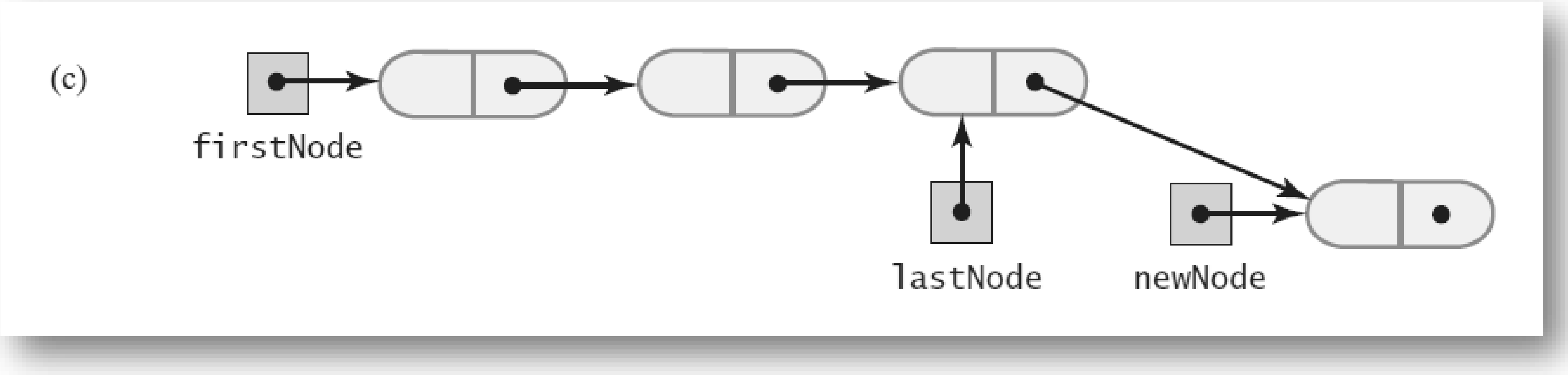
(b) after locating its last node;



Adding a Node

A chain of nodes

(c) after adding a node at the end



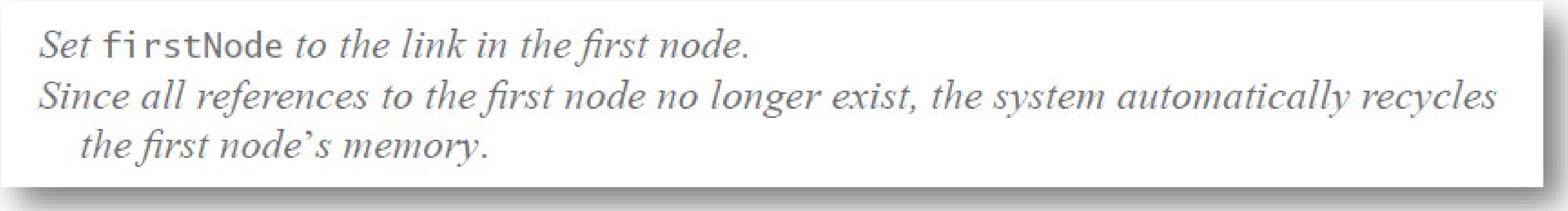
Removing a Node from Various Positions

Possible cases

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

Removing a Node

Steps for removing the first node.

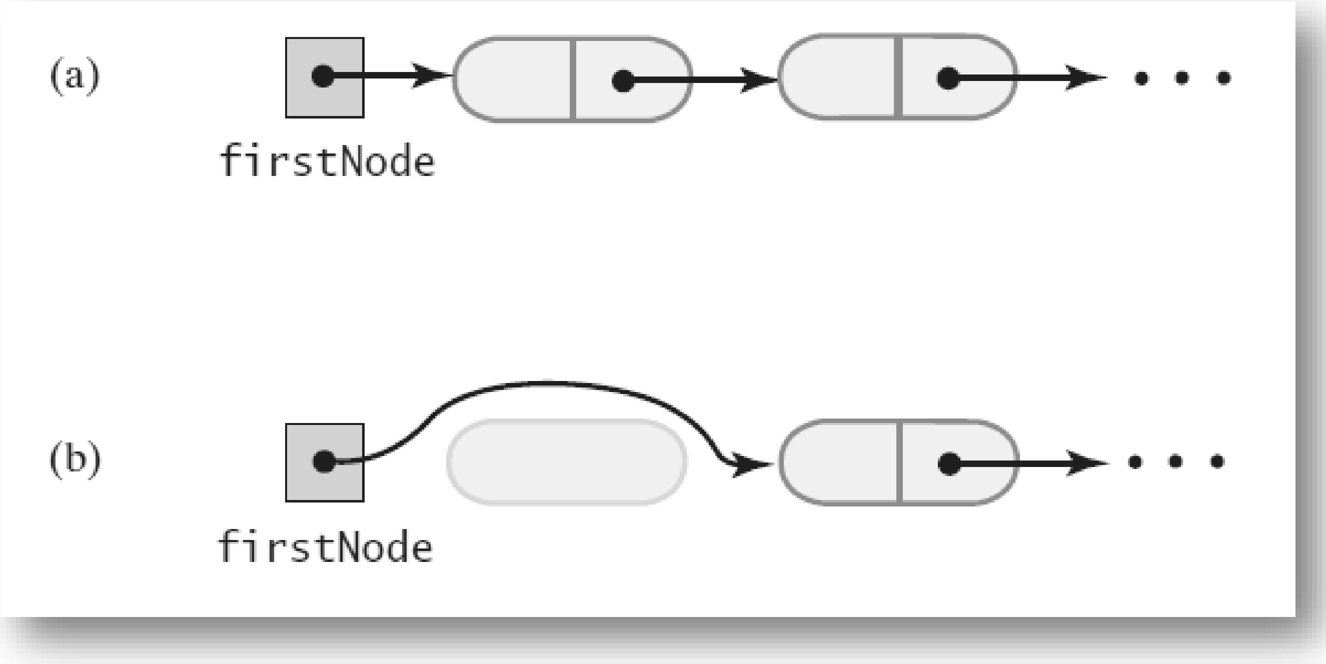


Set `firstNode` *to the link in the first node.*

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

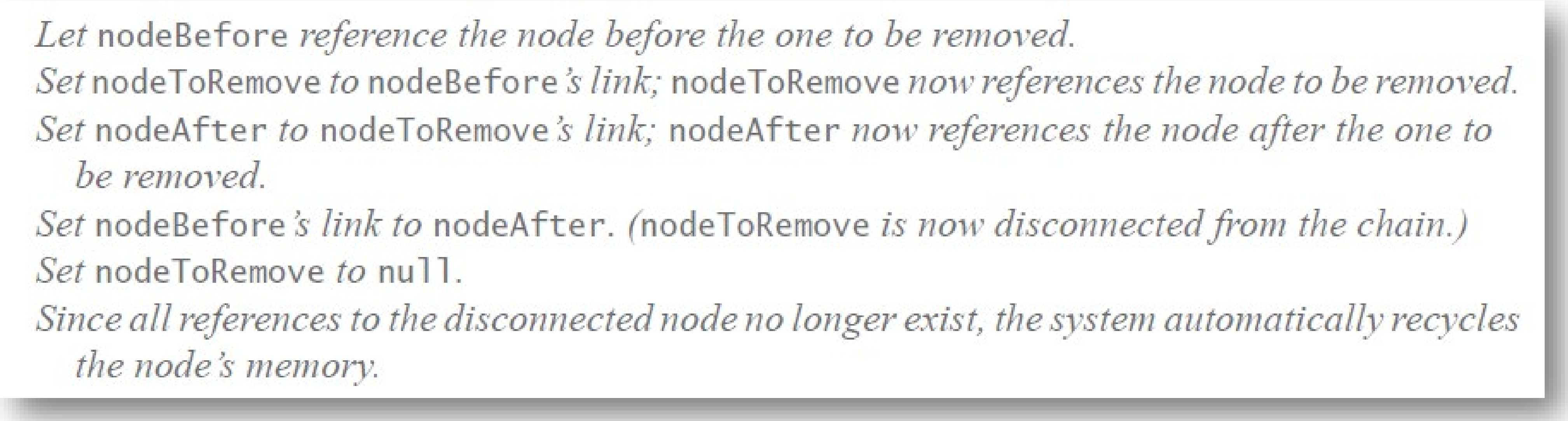
Removing a Node

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.*

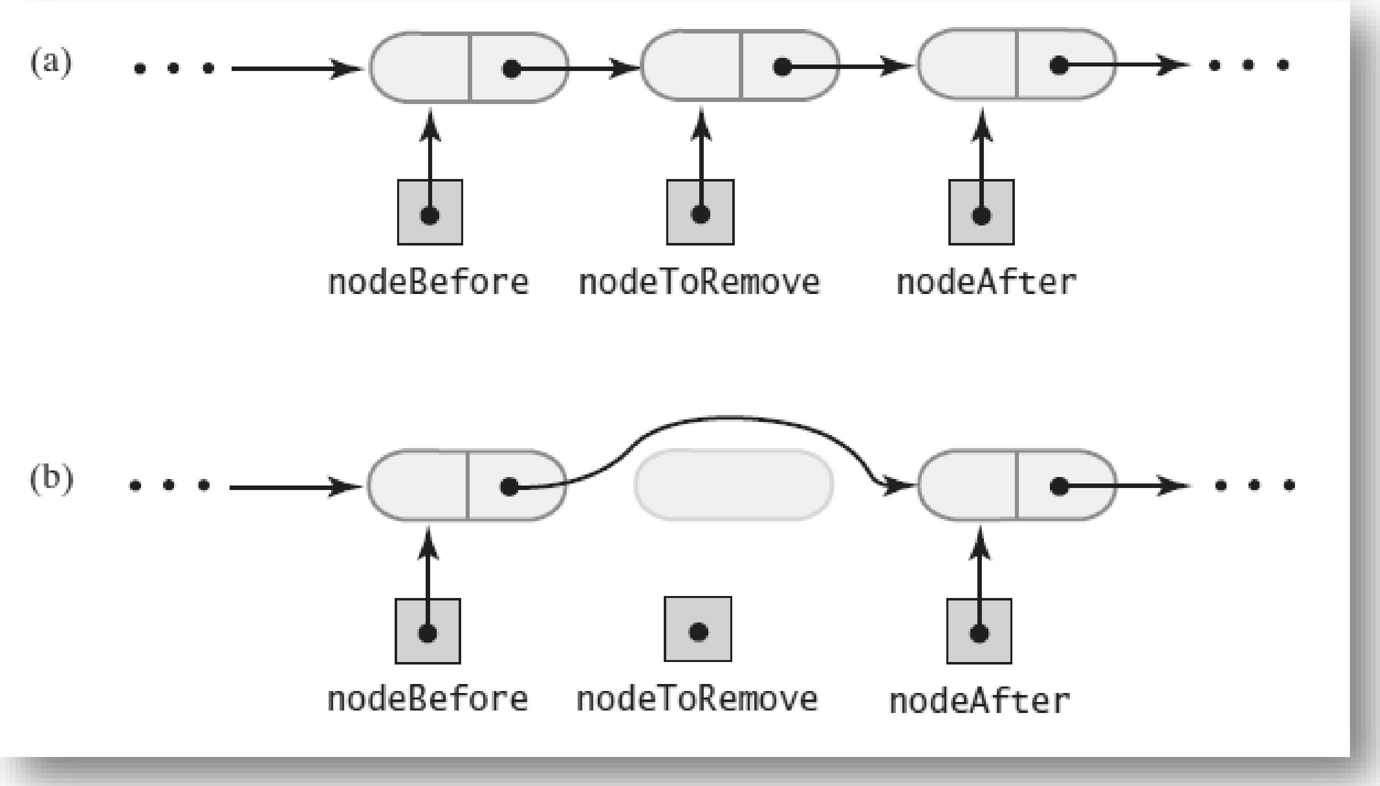
Set `nodeBefore`*'s link to* `nodeAfter`. *(*`nodeToRemove` *is now disconnected from the chain.)*

Set `nodeToRemove` *to* `null`.

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

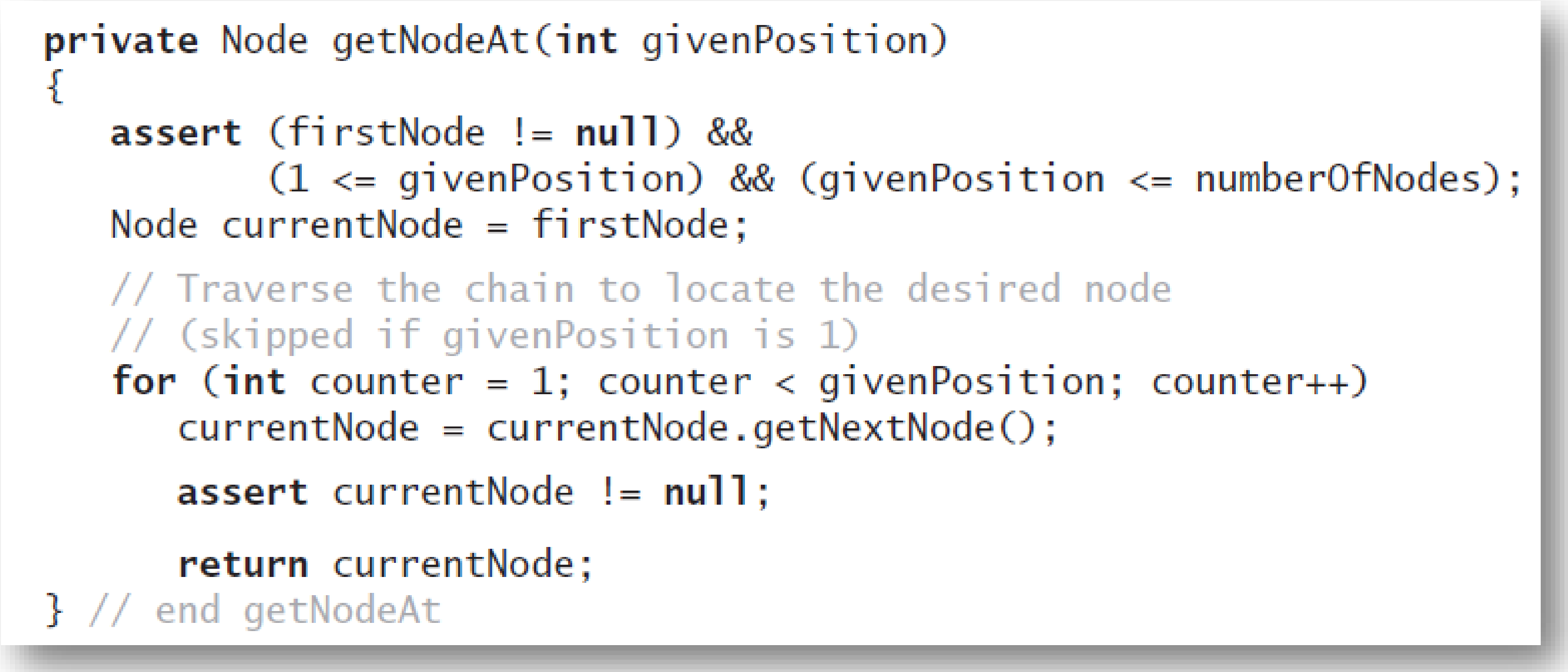
Removing a Node

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



Removing a Node

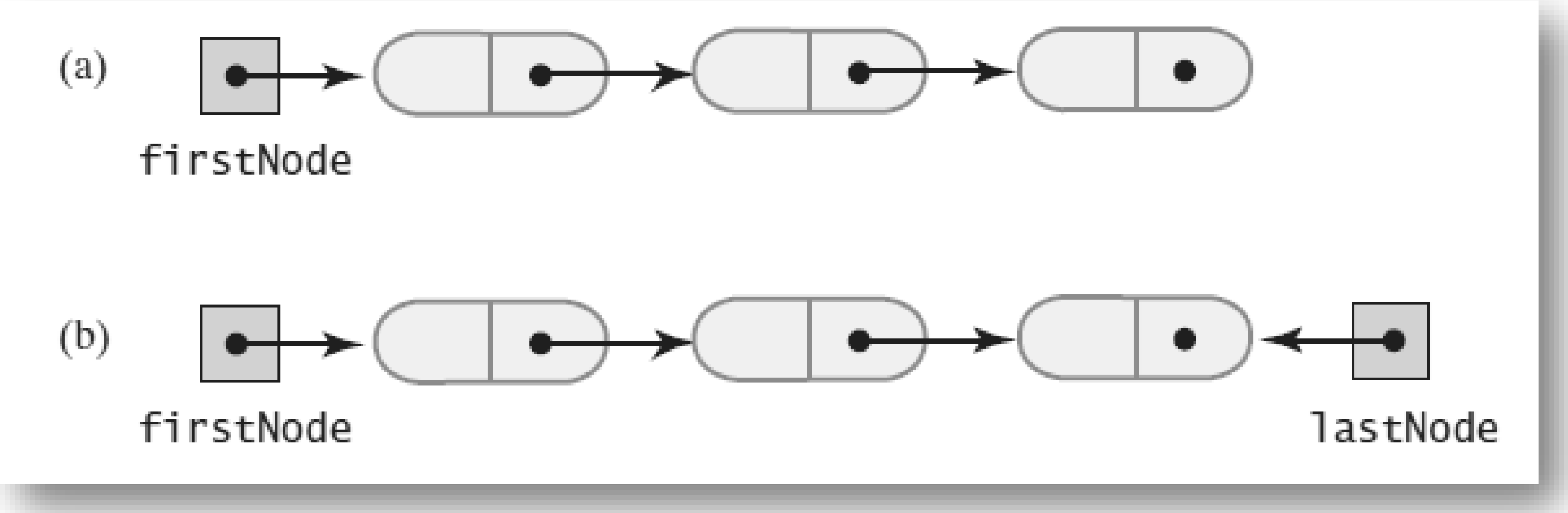
Operations on a chain depended on the method `getNodeAt`



```
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
```

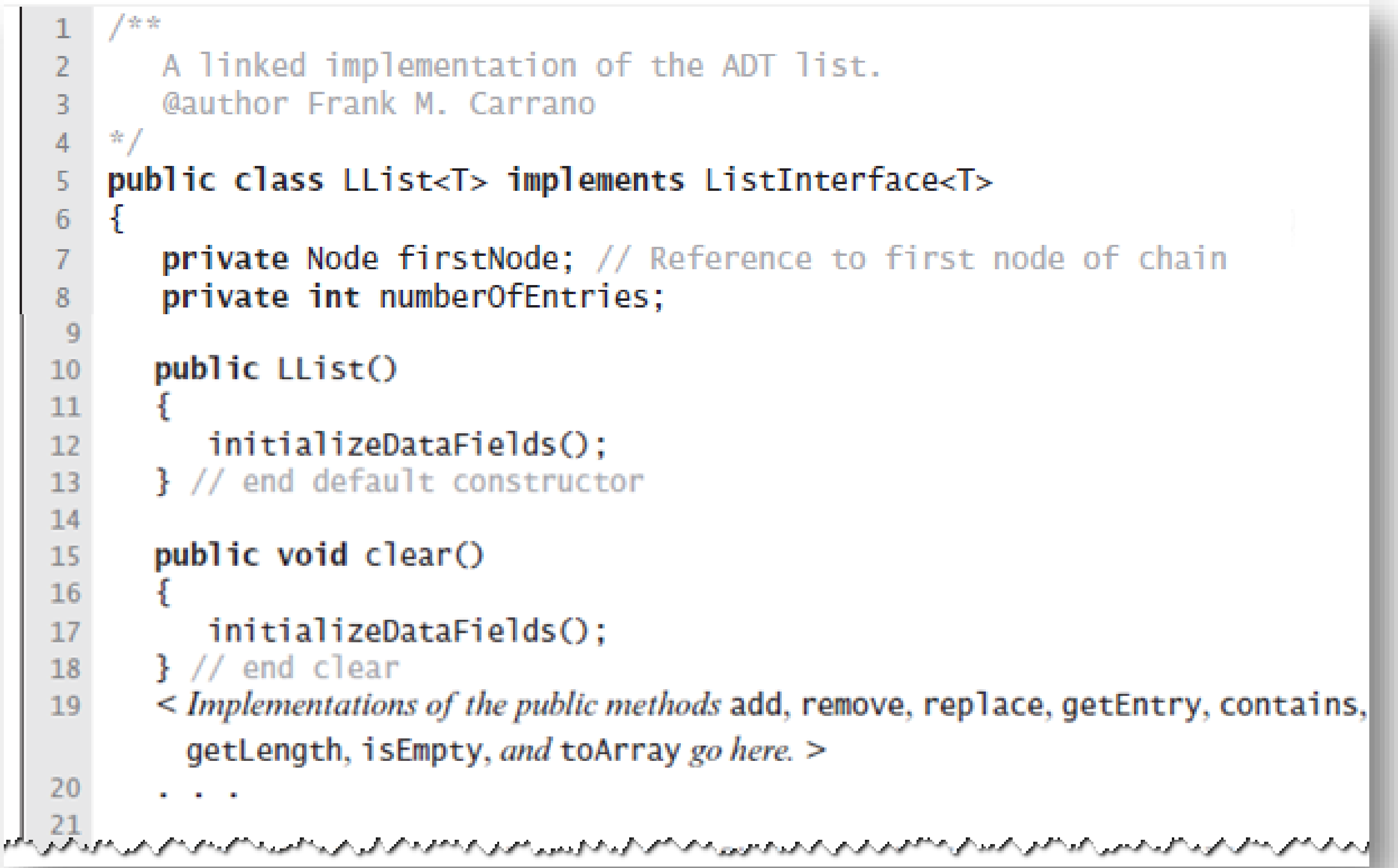
Design Decision: A Link to Last Node

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



Data Fields and Constructor

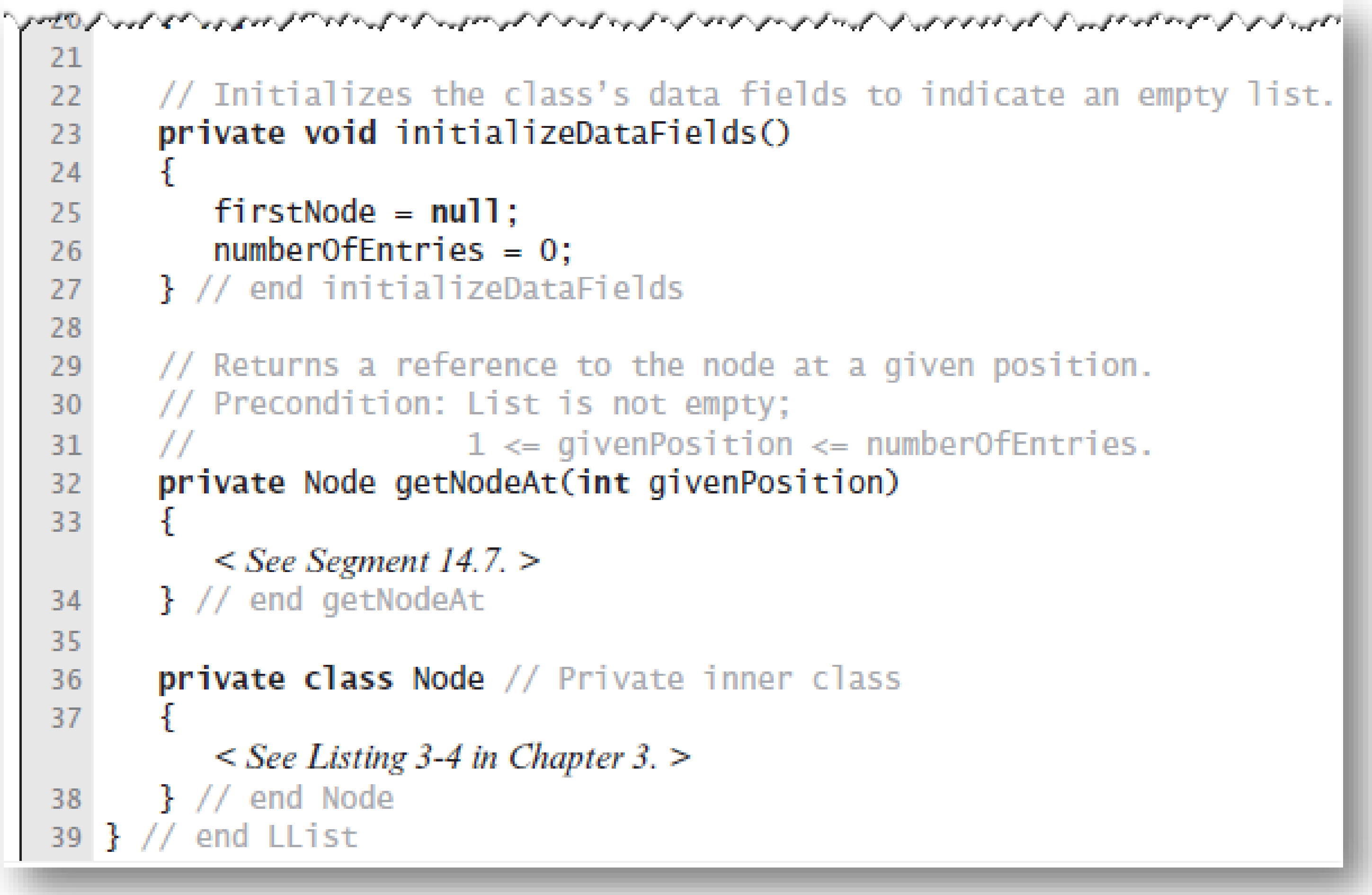
An outline of the class `LList`



```
/**
   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

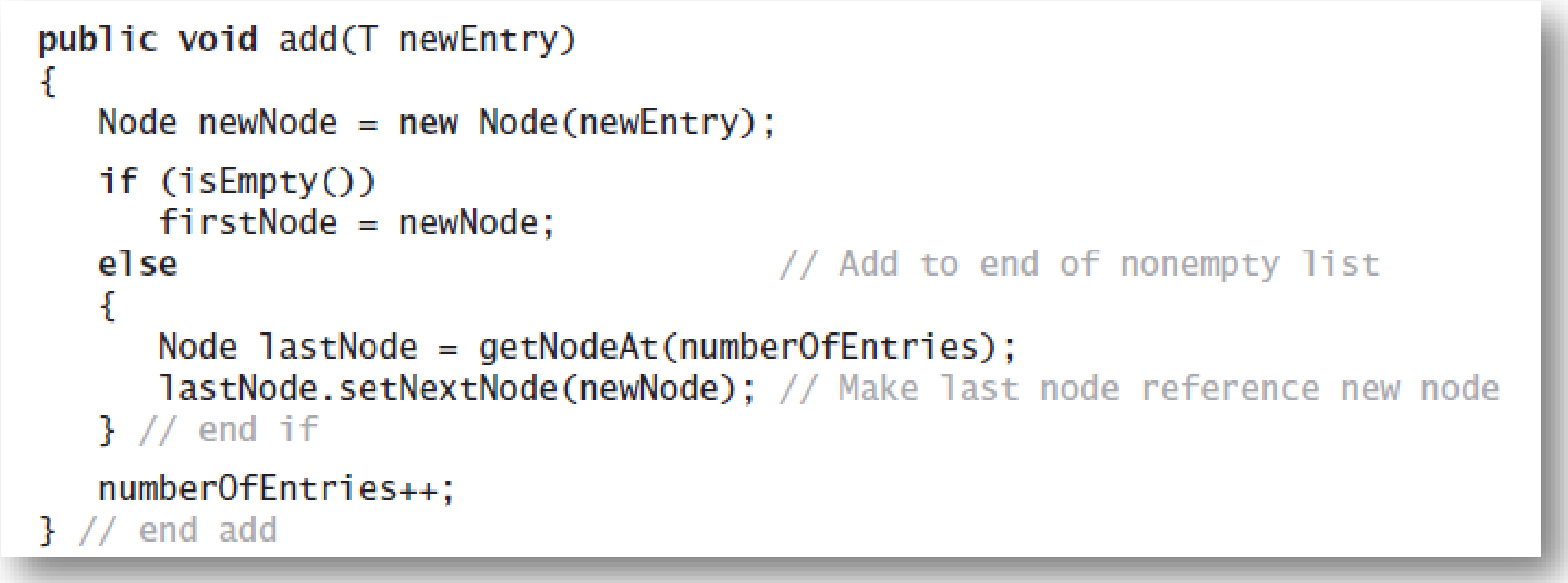
An outline of the class **`LList`**



```
   // 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
```

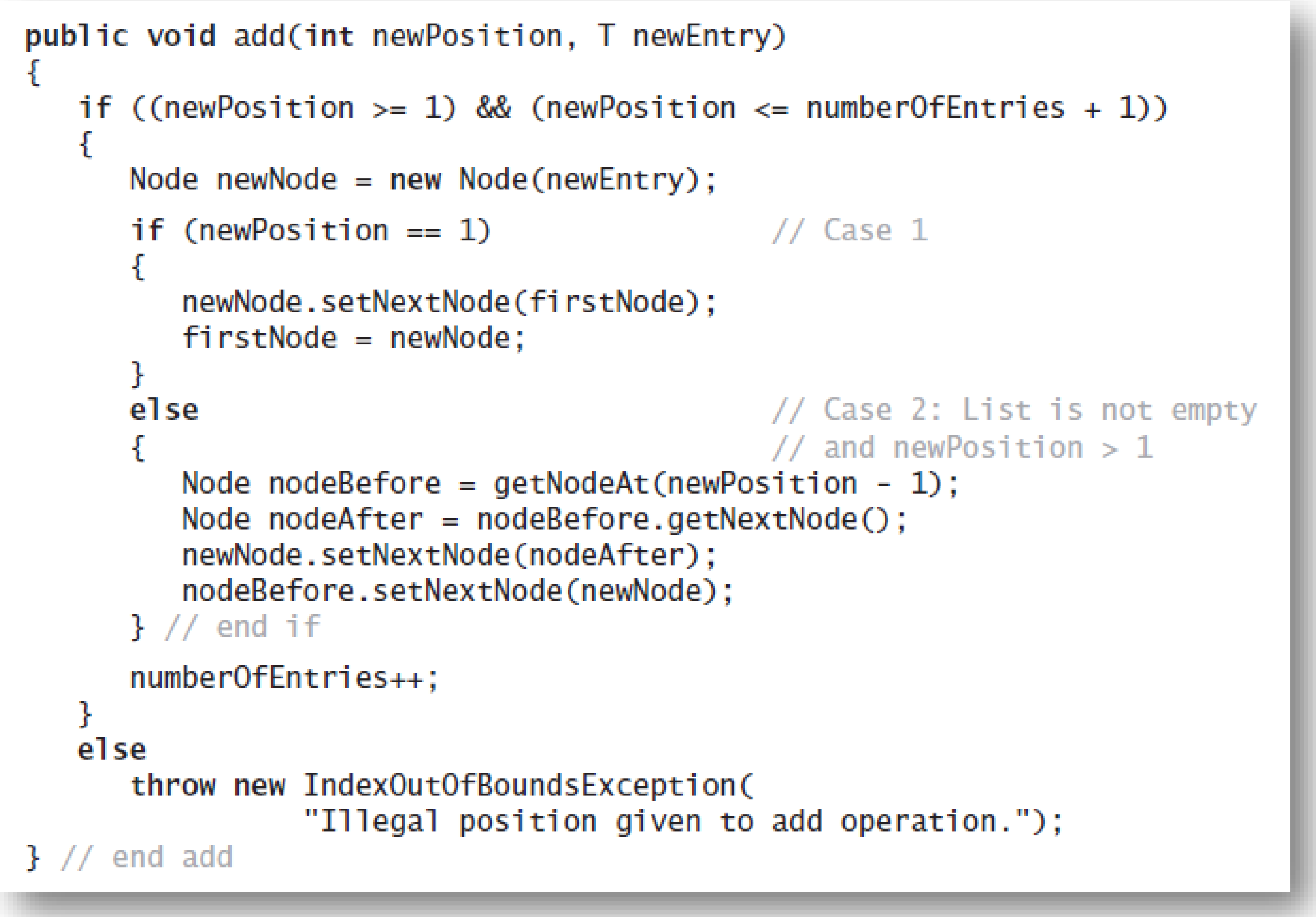
Adding to the End of the List

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
```

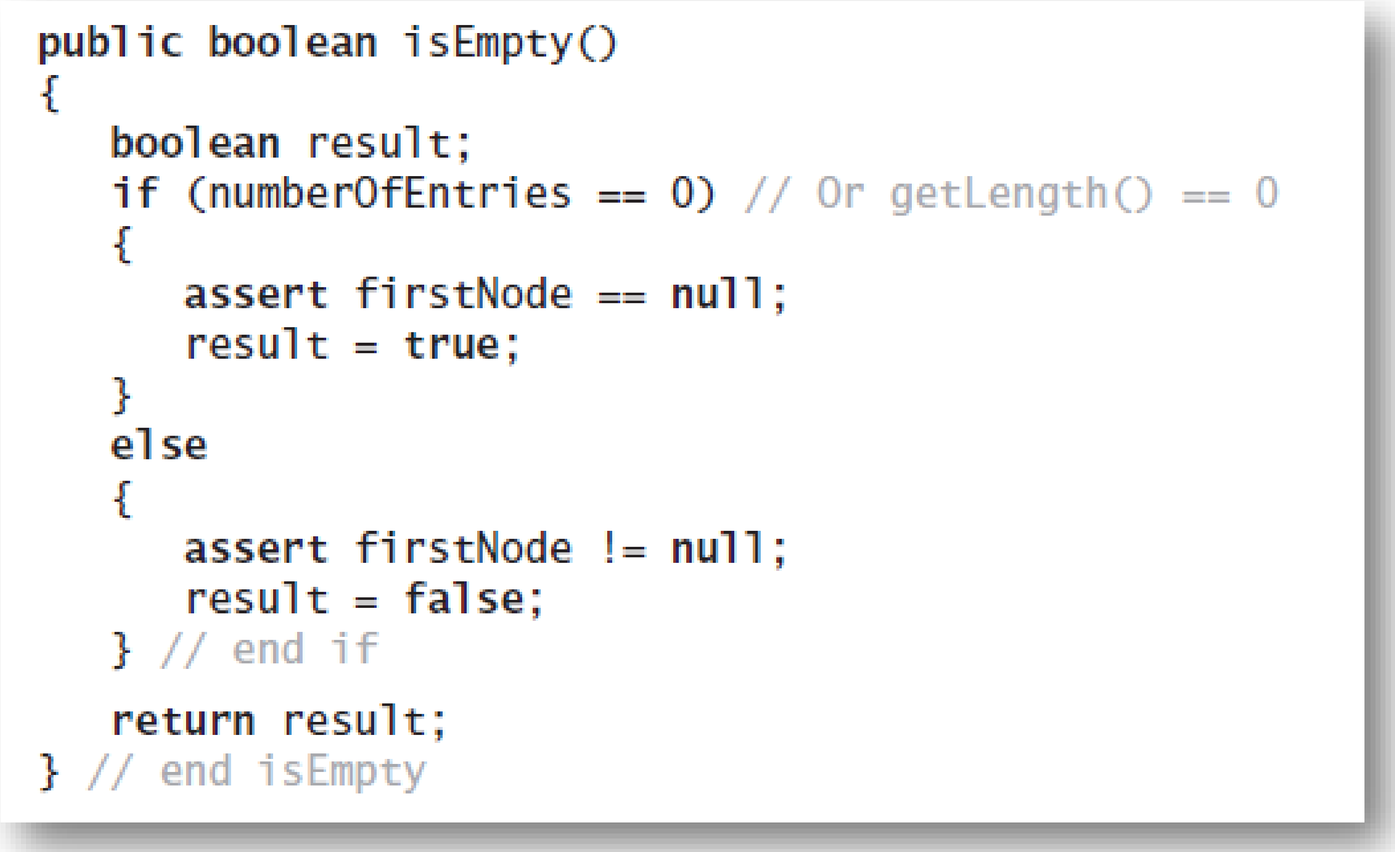
Adding at a Given Position



```
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

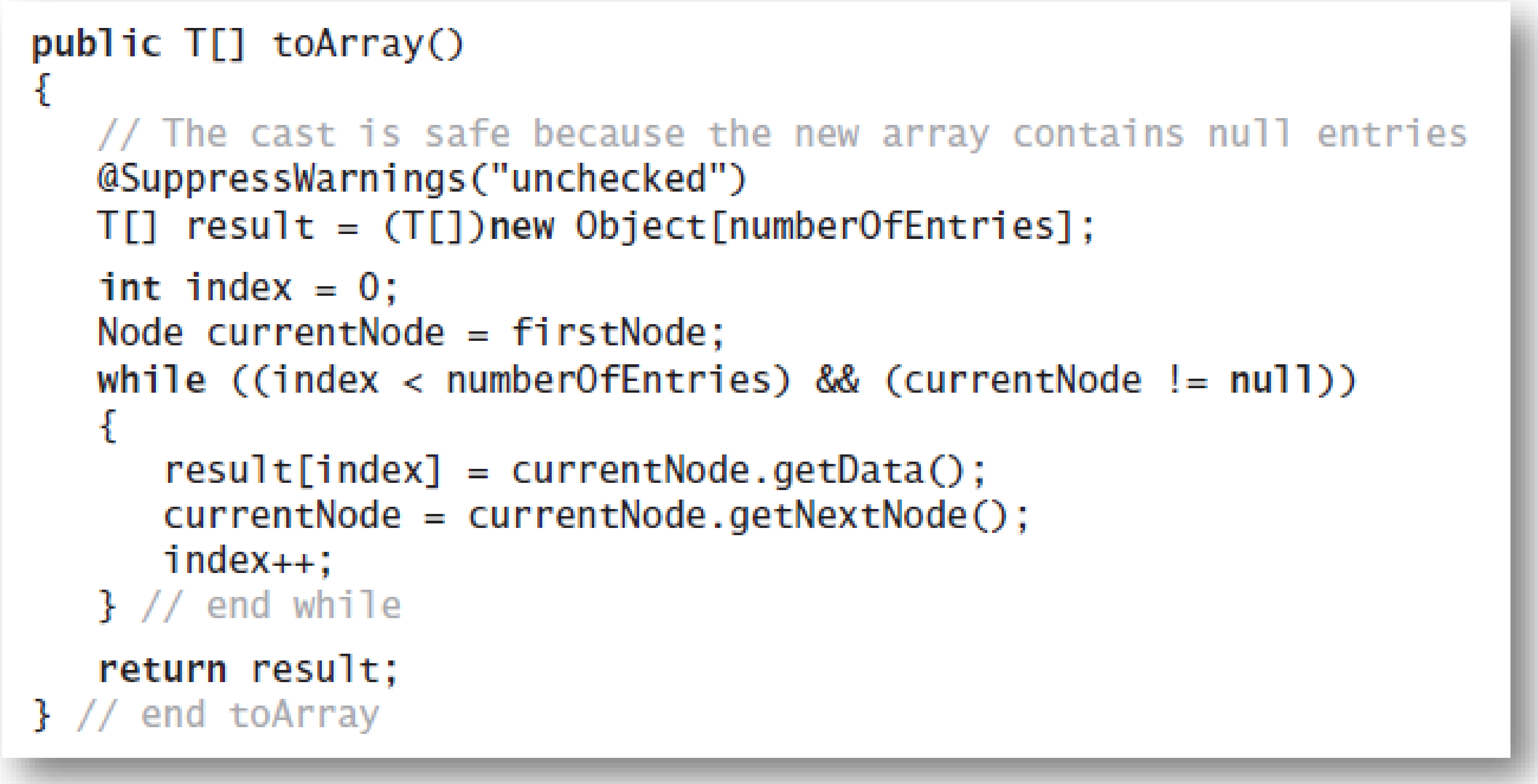
Note use of assert statement.



```java
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

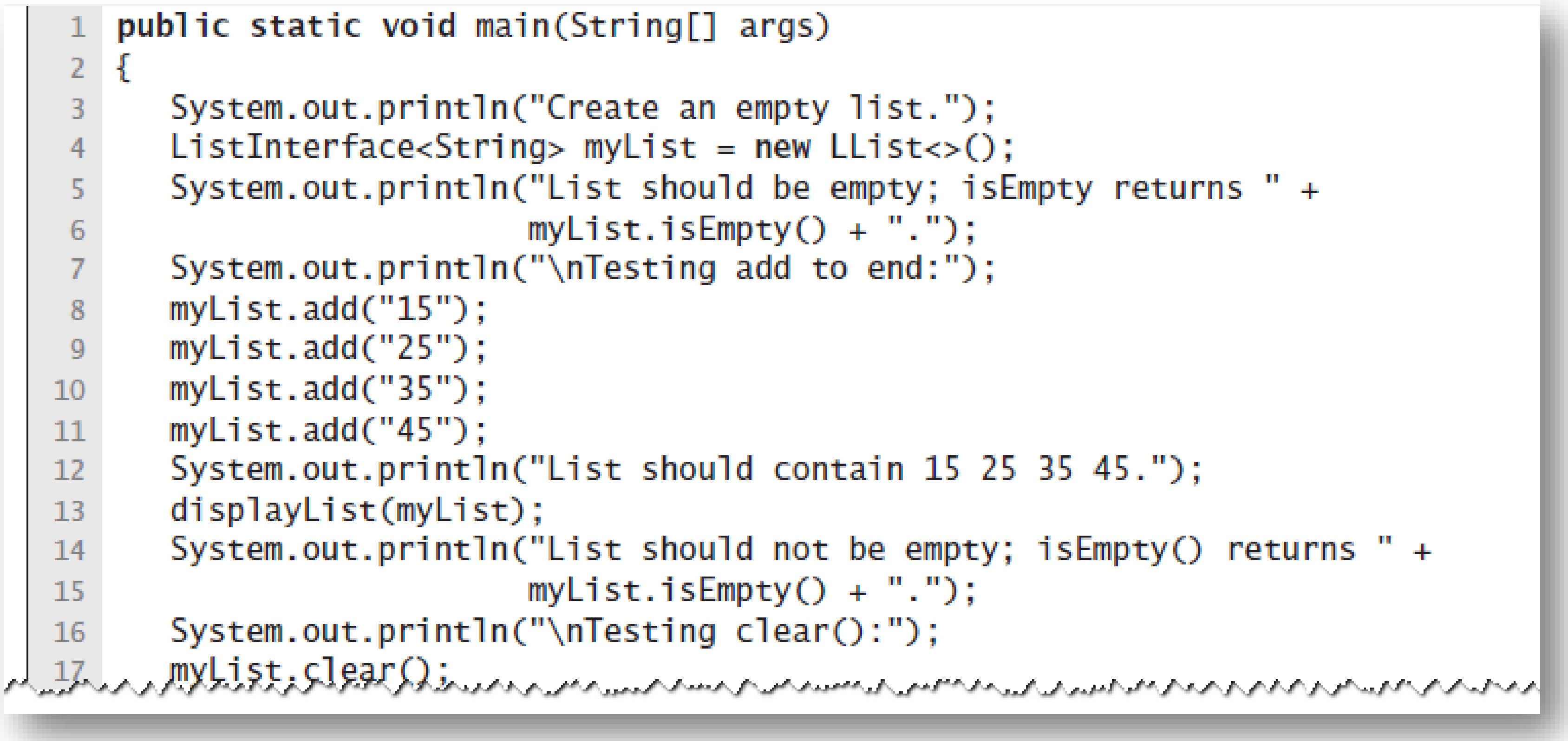
Traverses chain, loads an array.



```java
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

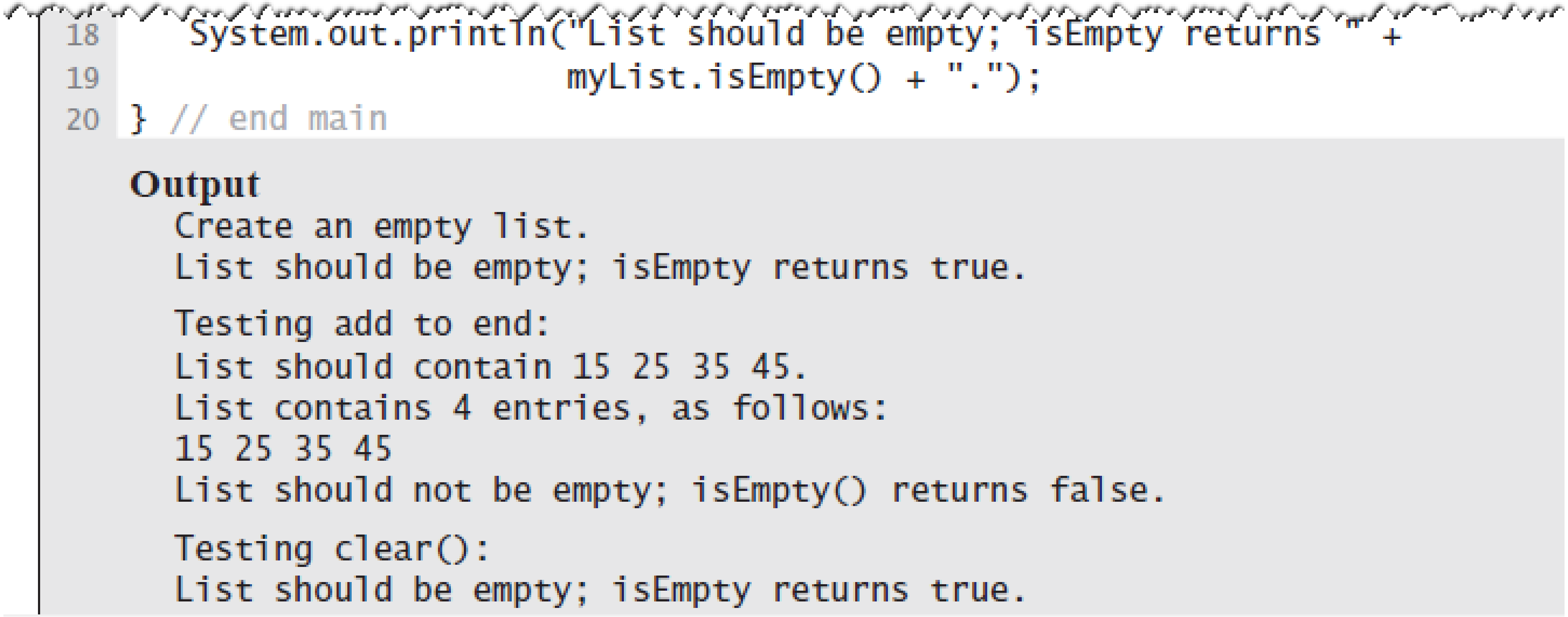
A `main` method that tests part of the implementation of the ADT list



```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



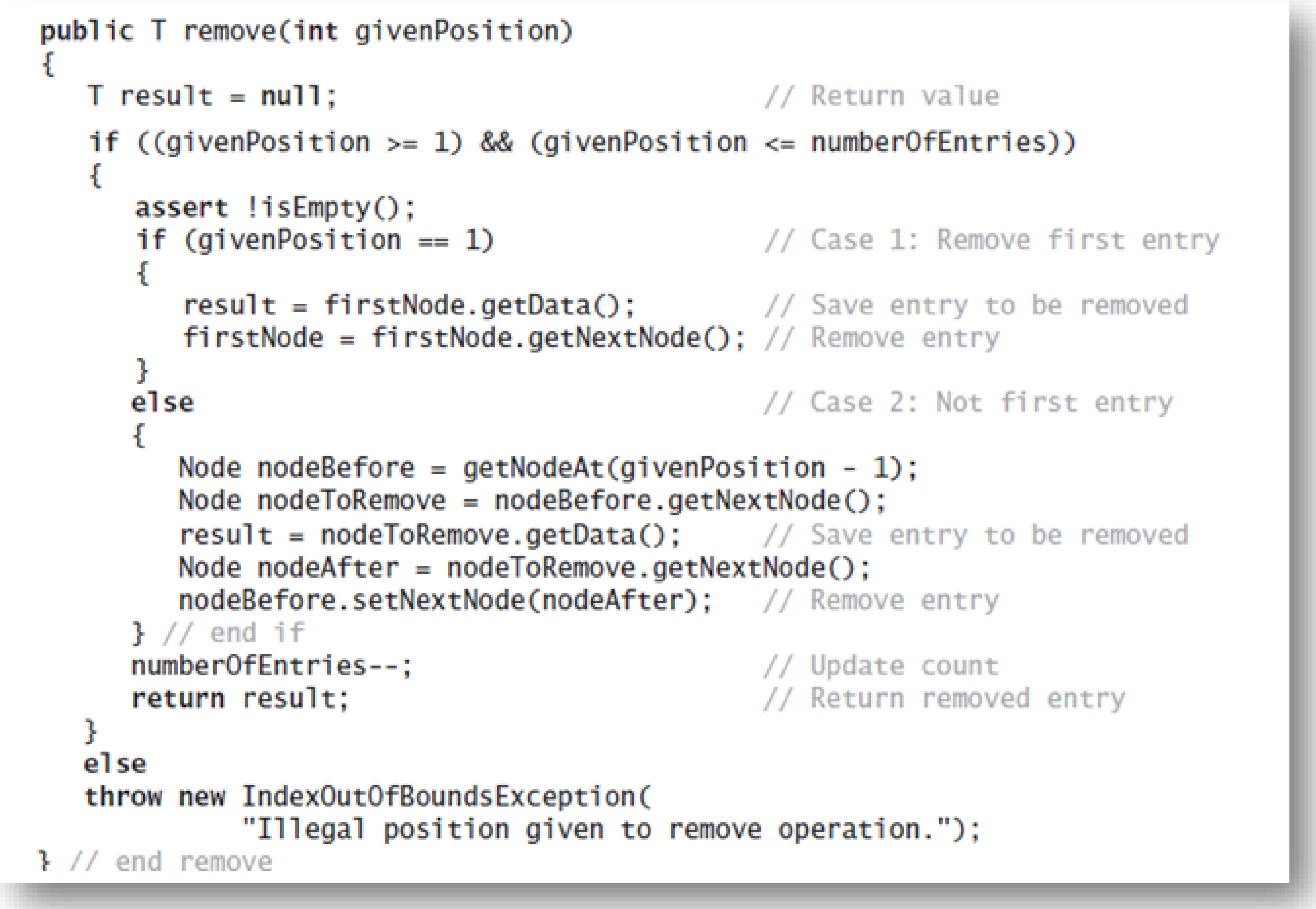
```
   System.out.println("List should be empty; isEmpty returns " +
                      myList.isEmpty() + ".");
} // end main
```

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

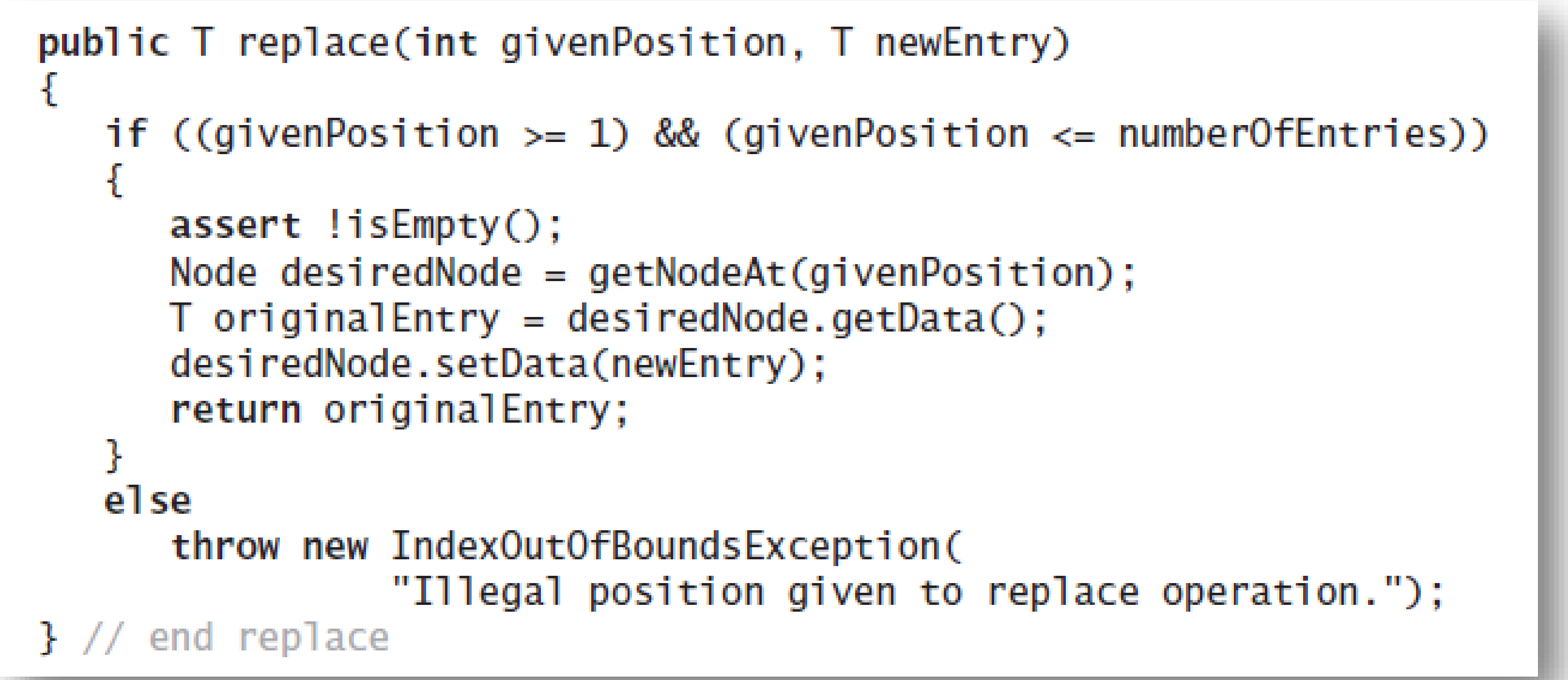
The **remove** method returns the entry that it deletes from the list



```
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

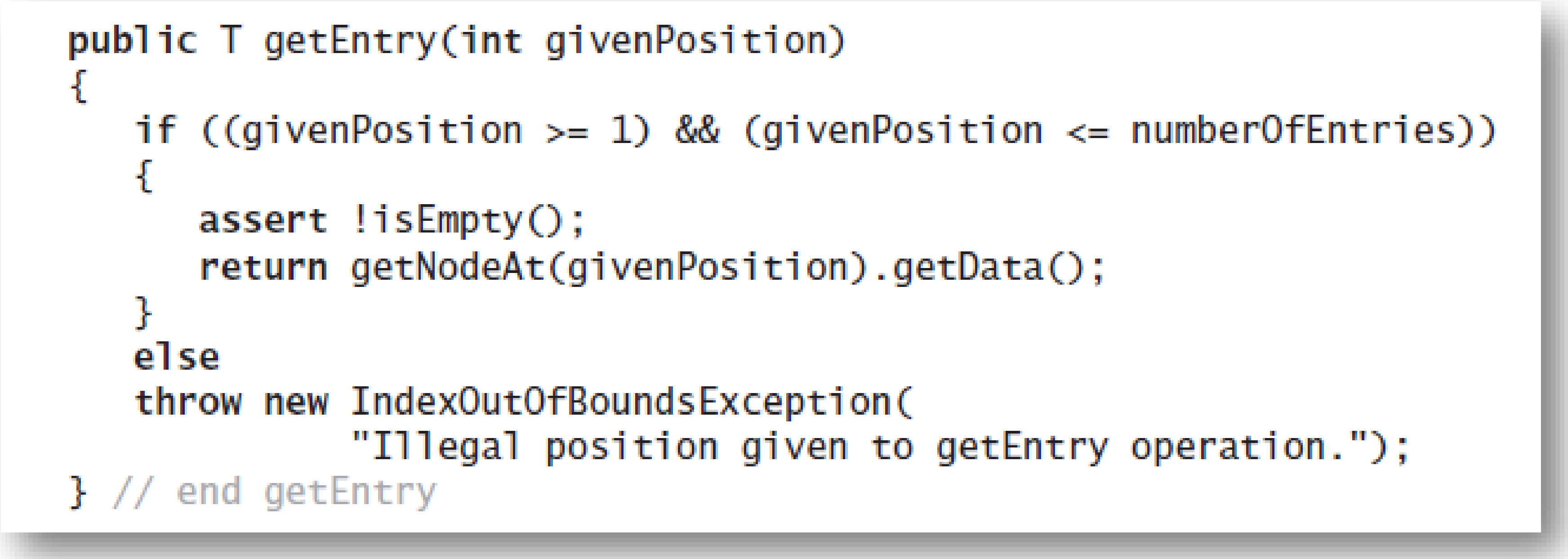
Replacing a list entry requires us to replace the data portion of a node with other data.



```java
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

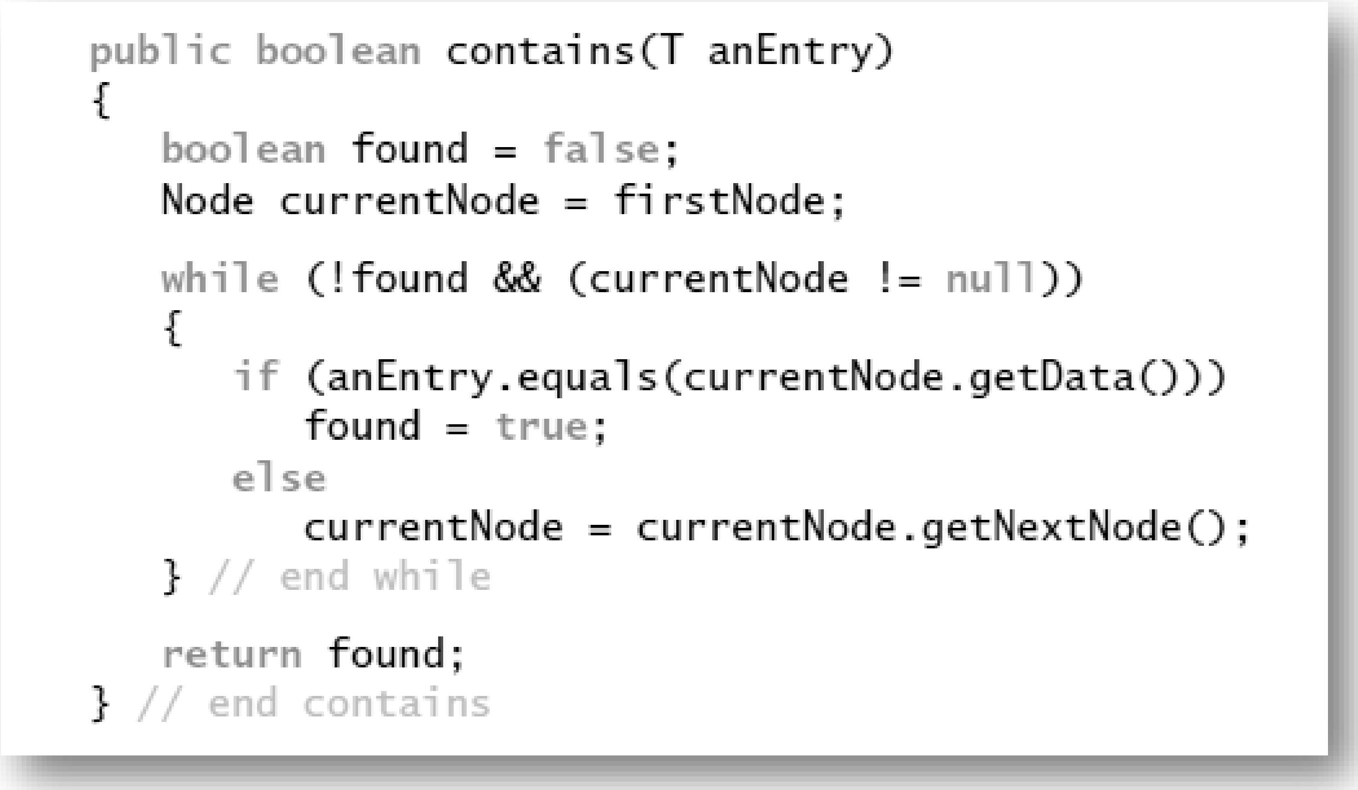
Retrieving a list entry is straightforward.



```
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

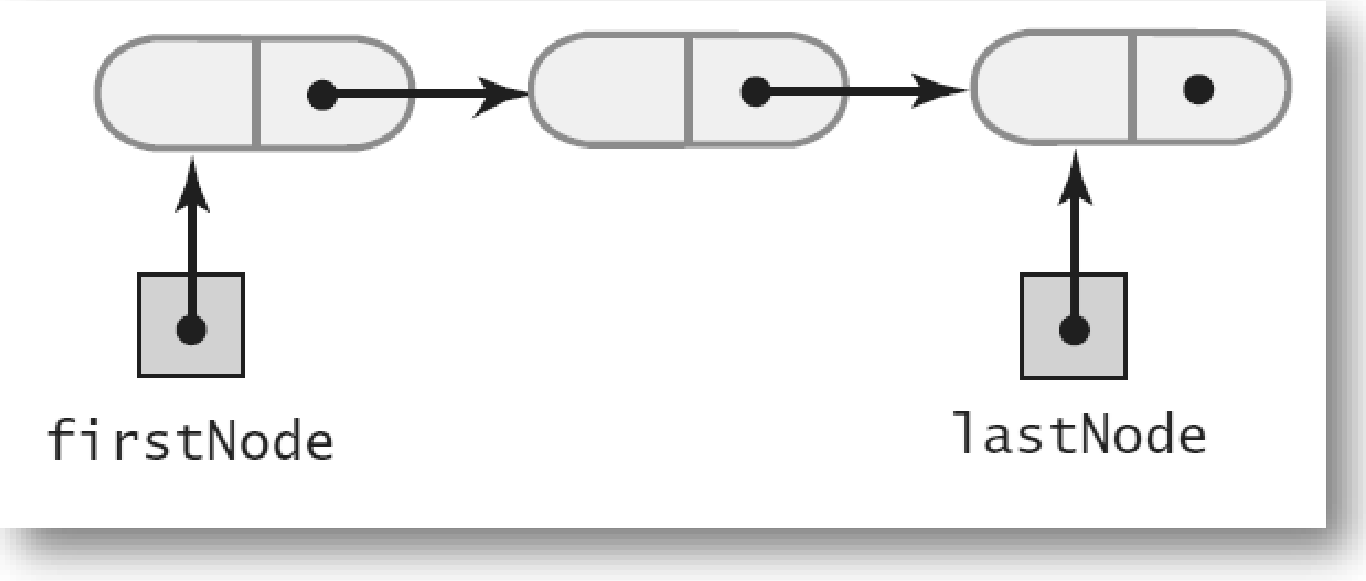
Checking to see if an entry is in the list, the method **contains**.



```
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
```

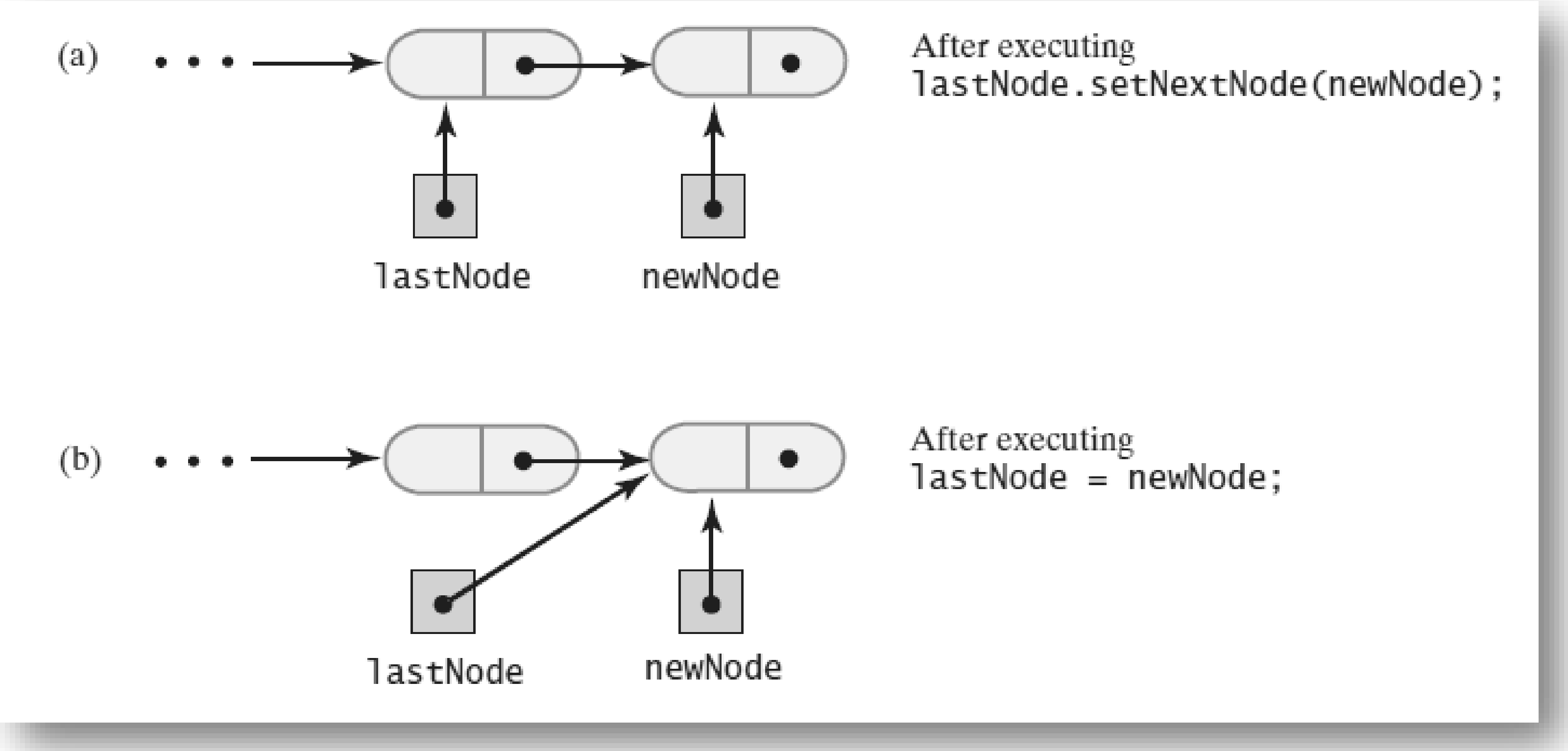
A Refined Implementation

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

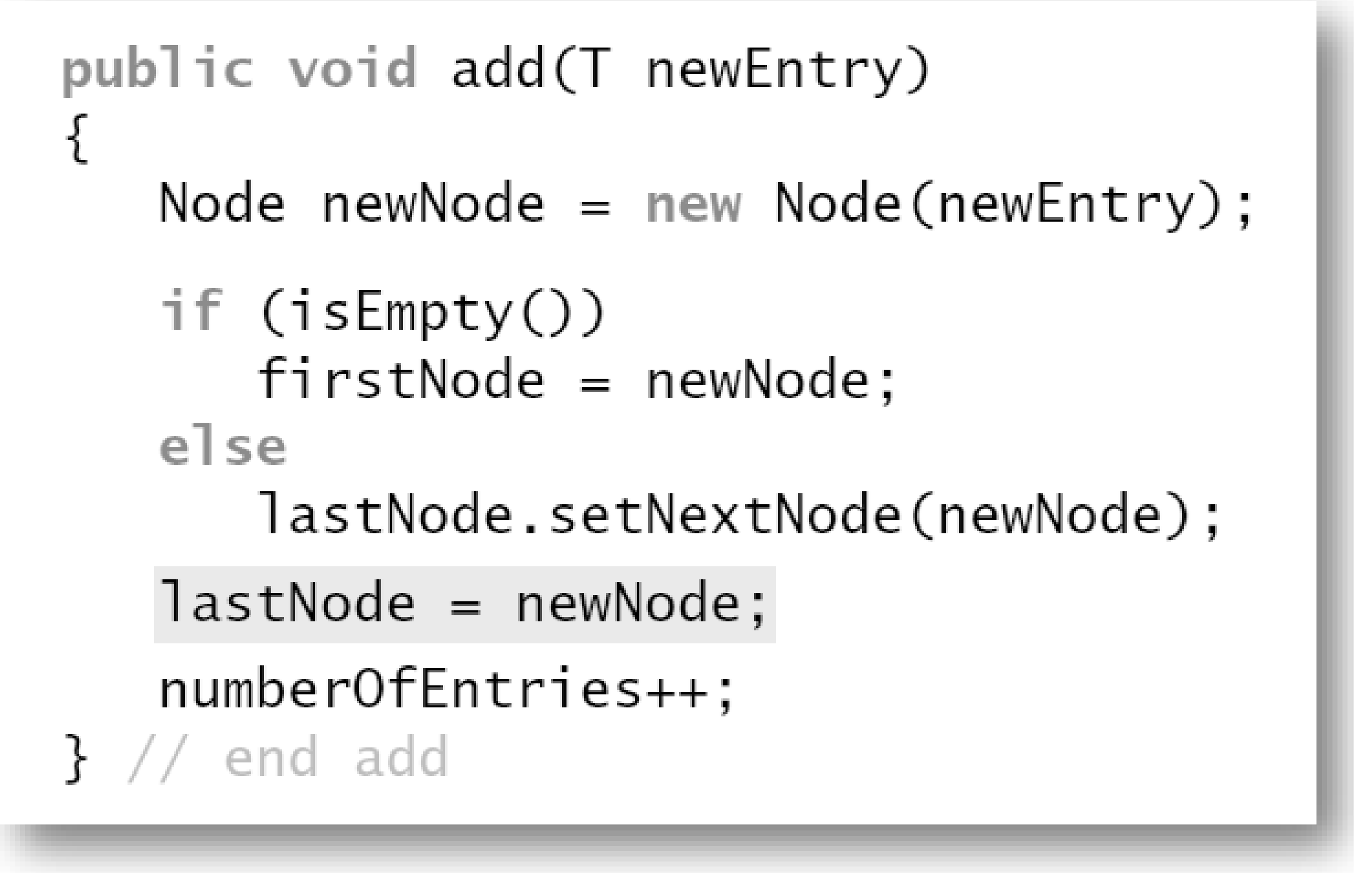


After executing `lastNode.setNextNode(newNode);`

After executing `lastNode = newNode;`

A Refined Implementation

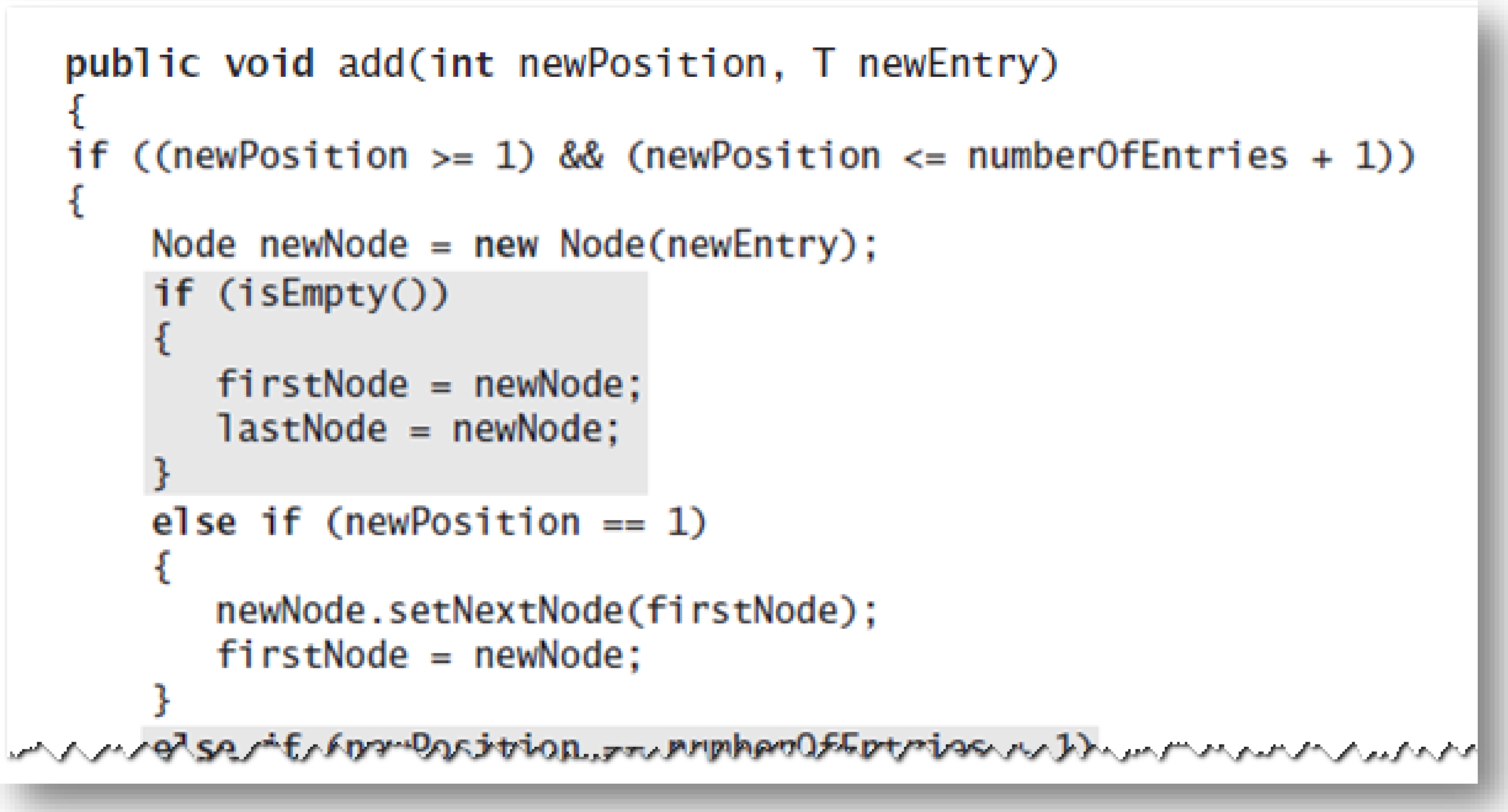
Revision of the first `add` method



```
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

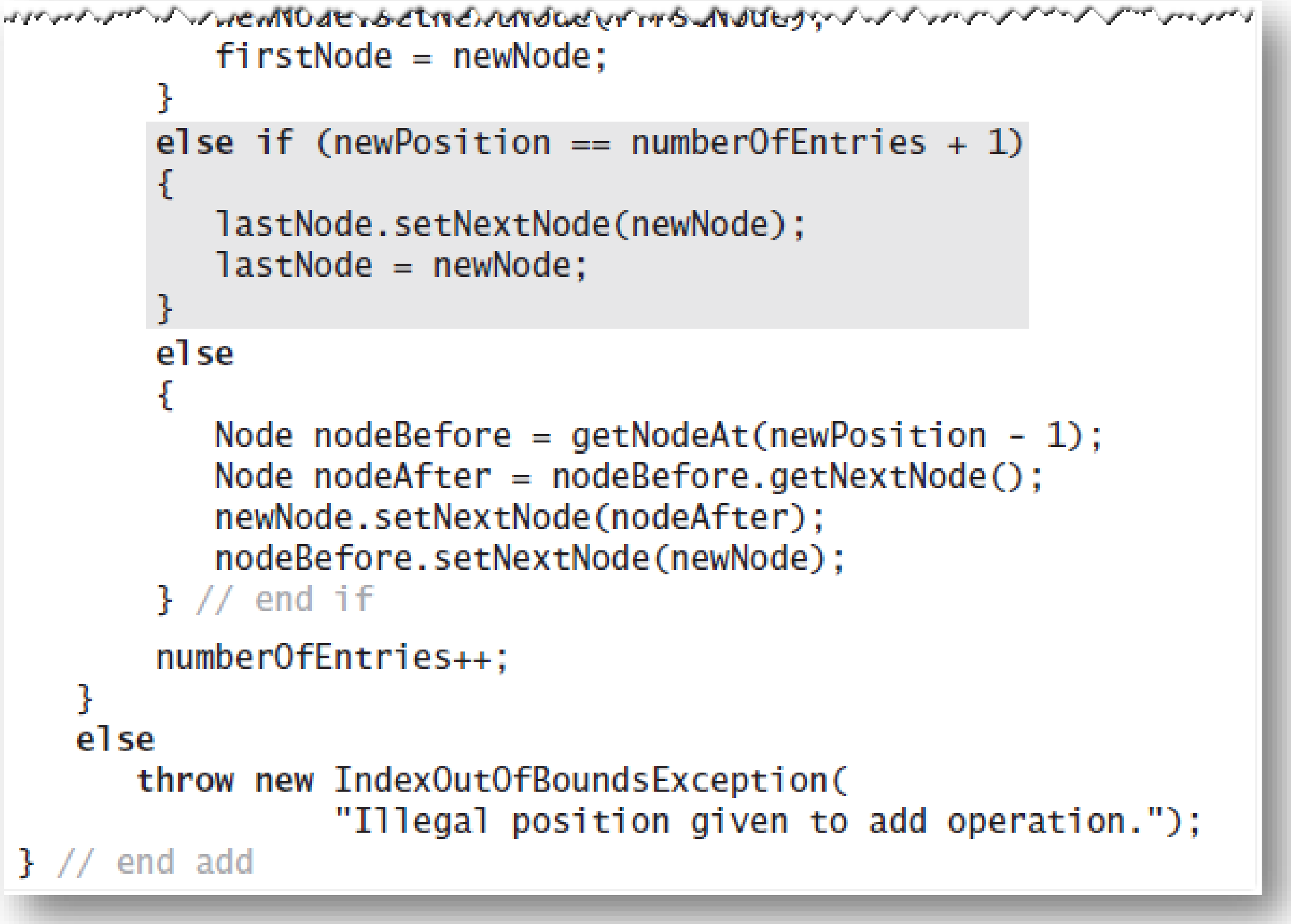
Implementation of the method that adds by position.



```
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

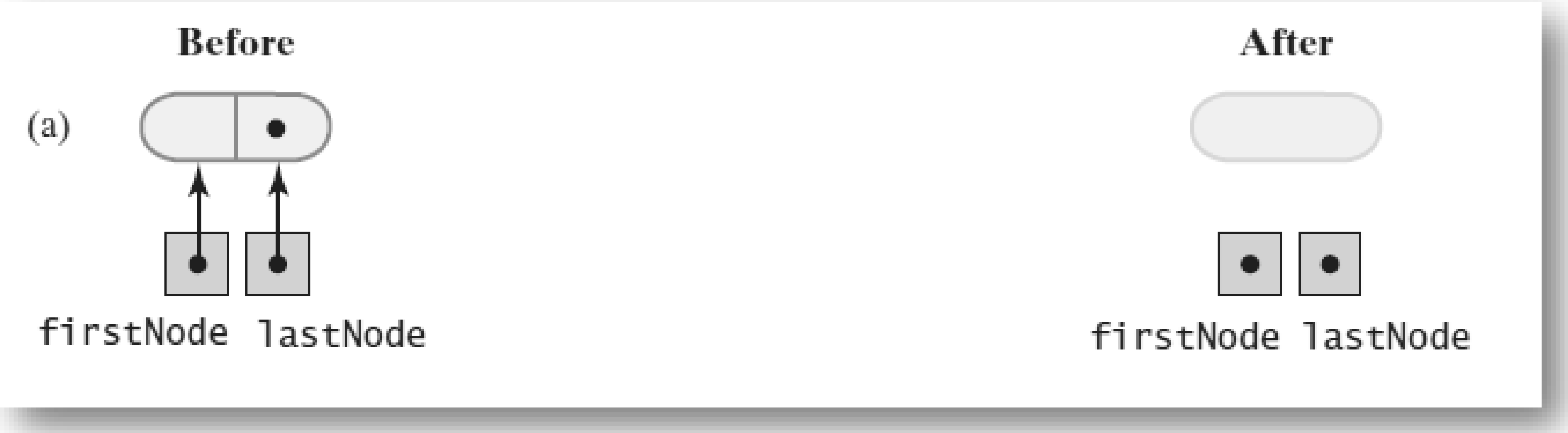
Implementation of the method that adds by position.



```
          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
```

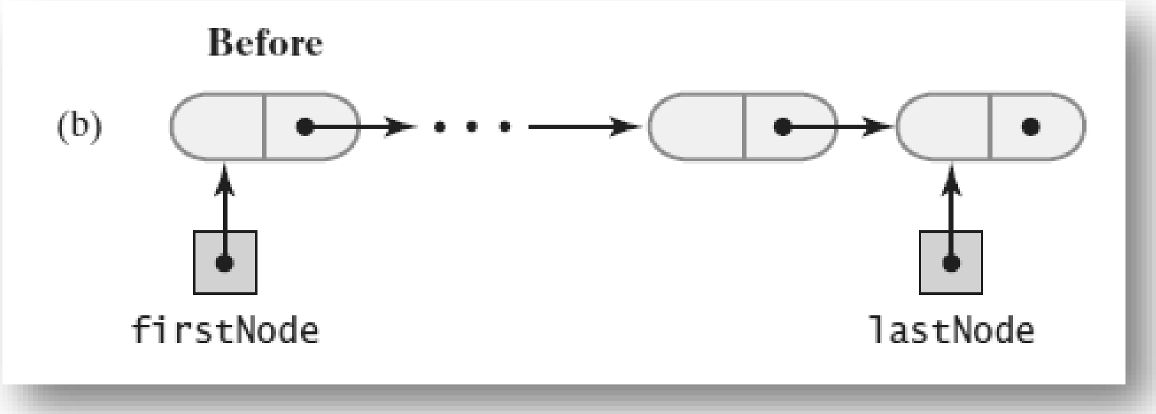
A Refined Implementation

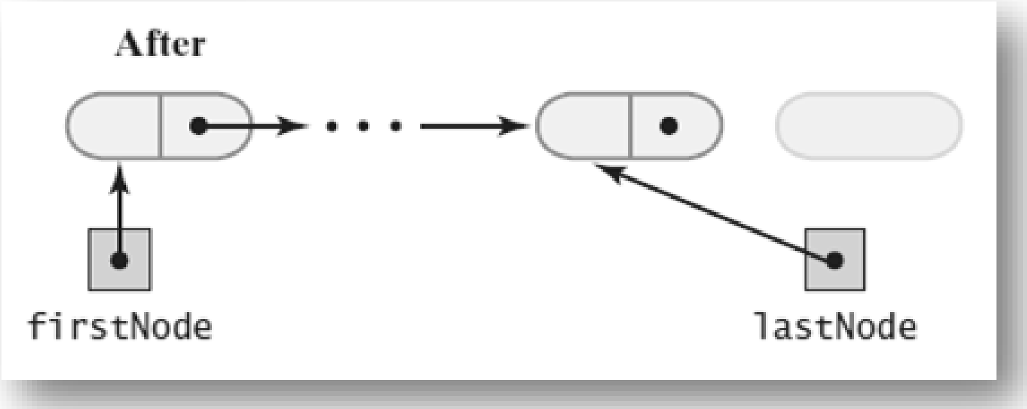
Removing the last node from a chain that has both head and tail references when
the chain contains (a) one node



A Refined Implementation

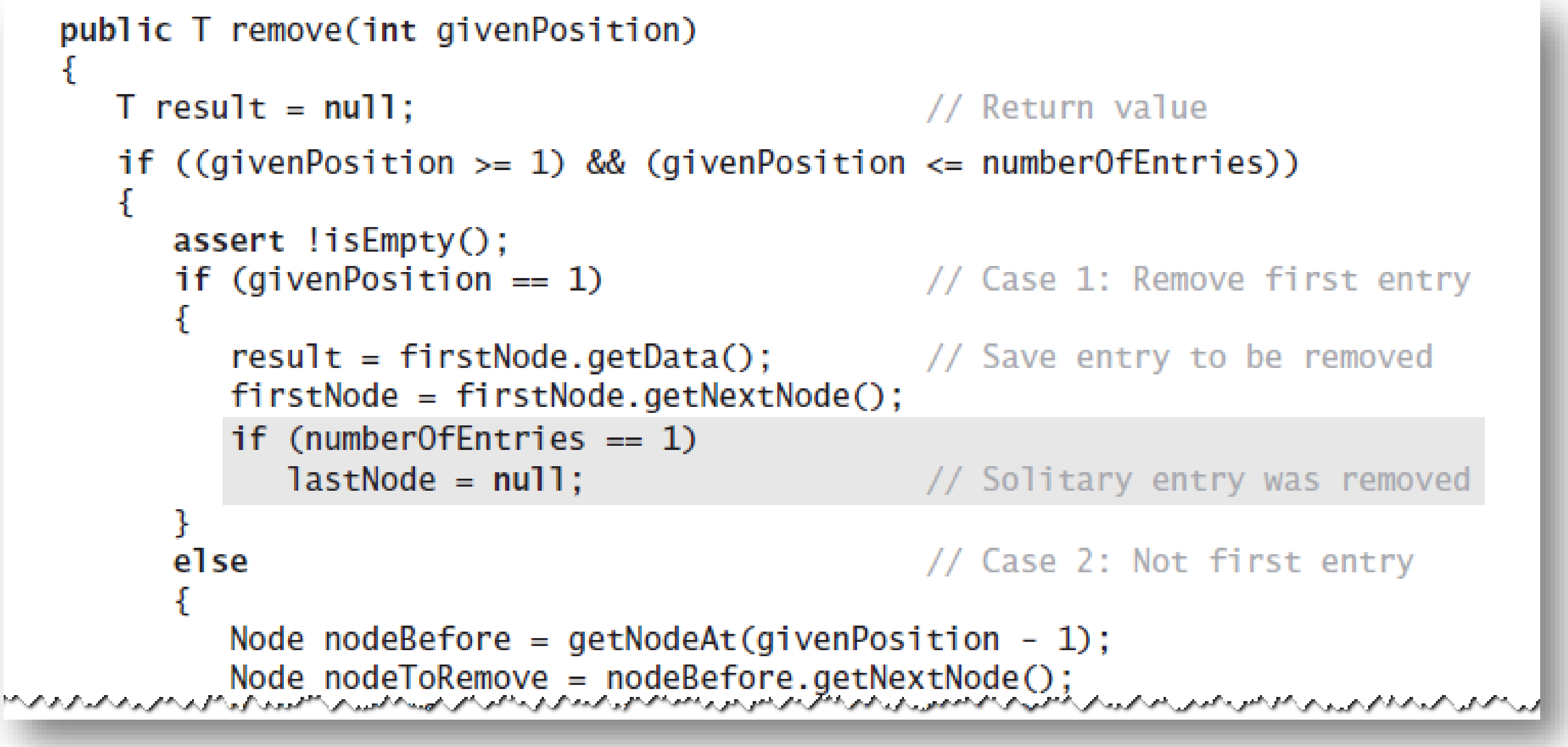
Removing the last node from a chain that has both head and tail references when the chain contains (b) more than one node





A Refined Implementation

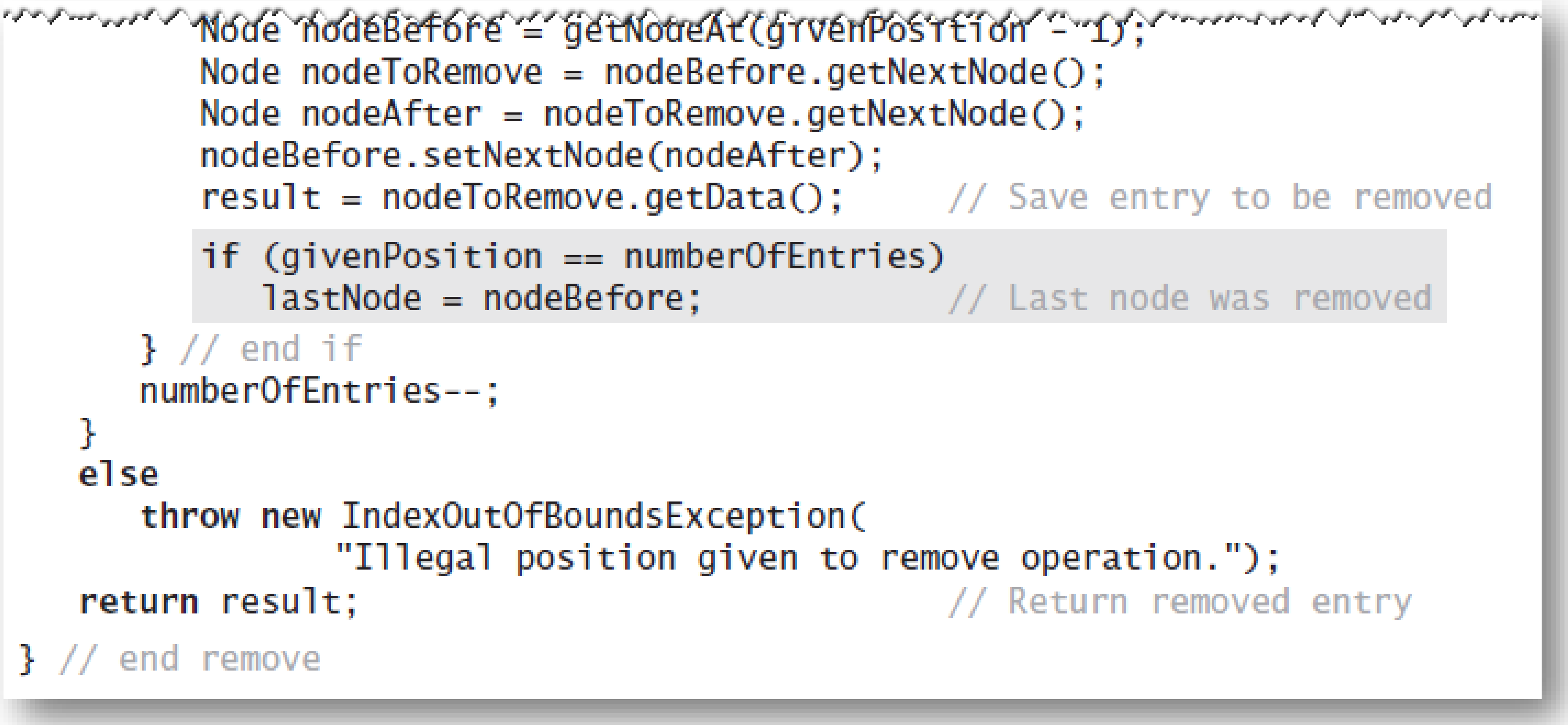
Implementation of the remove operation:



```
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

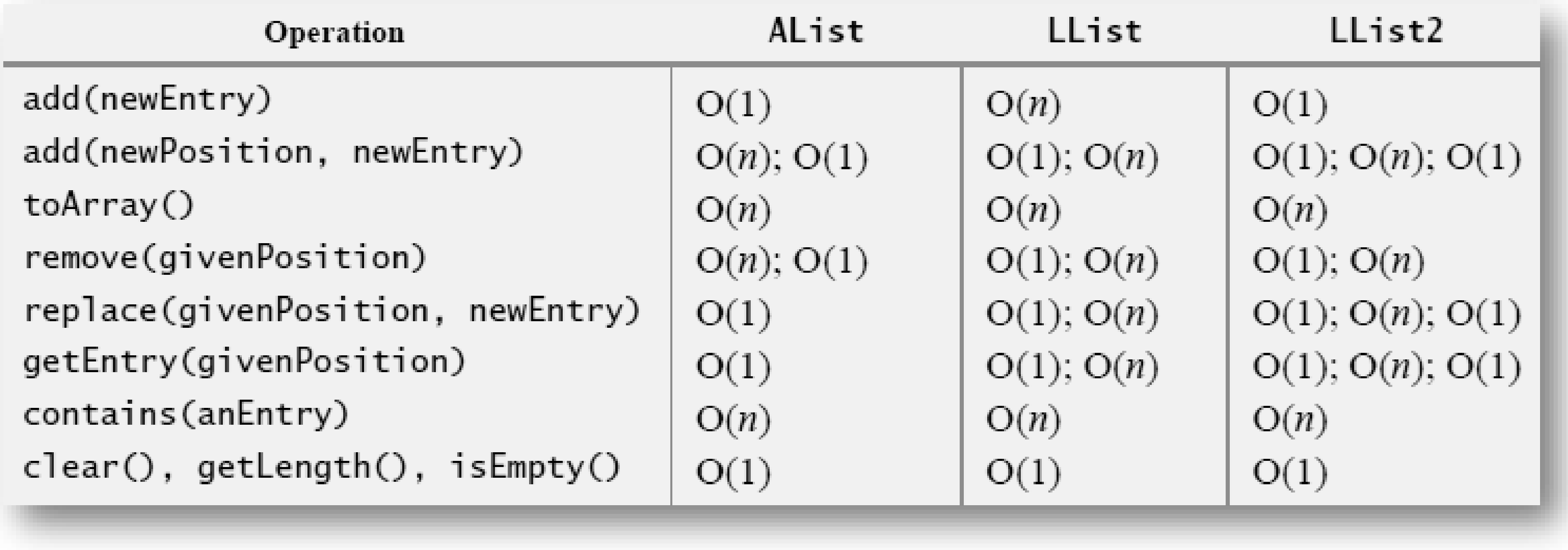
Implementation of the remove operation:



```
         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.