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 4: this Friday @ 11:59 pm
  - Lab 3: 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 …

- Big-Oh Approximation
- ADT List
  - resizable array implementation: ArrayList
    - Constructors, add, makeRoom, ensureCapacity

Muddiest Points

- **Q: In the Live Code List implementation, it is defined as "private final T[] list;" with a max array size of 1000. To clarify, we cannot set our list in the ArrayList class as final if we need to double the size of the array to create more space, right?**
- Right!

Muddiest Points

- **Q: Answers to the BigO TopHat question. Specifically, why we "Ignore rarely-occurring corner cases"? (First, what exactly does 'corner' mean in this case.) I thought the purpose of the BigO was to analyze large values of n, i.e worse case scenarios**
- **Q: When to care about best case, worst case, and average case for big O efficiency**
- The running time of some algorithms depends not only on the input size (*n*) but also on the *values* of the input
- Certain values cause the algorithm to run longer than other cases. These input values are sometime called corner cases
- Example: Searching for an item that doesn't exist in an array vs. searching for an item that happens to be at the first entry: We must keep searching until the end of the array for the former case
- Large values of n ≠ worst-case scenarios

Muddiest Points

- **Q: The only thing I (still) don't really get is the I think it's called type casting in the toArray() method, to create/initialize the array as a generic.**
- toArray() method returns T[]
- After type checking, Java Compiler will change T to its upper-bound type, Object in this case
- Java Compiler will also insert type casting parentheses into the following statement
  - String[] items = bagOfStrings.toArray();
- It becomes:
  - String[] items = (String[]) bagOfStrings.toArray();
- This type casting will throw ClassCastException at run-time
  - because Object[] is a supertype of String[]

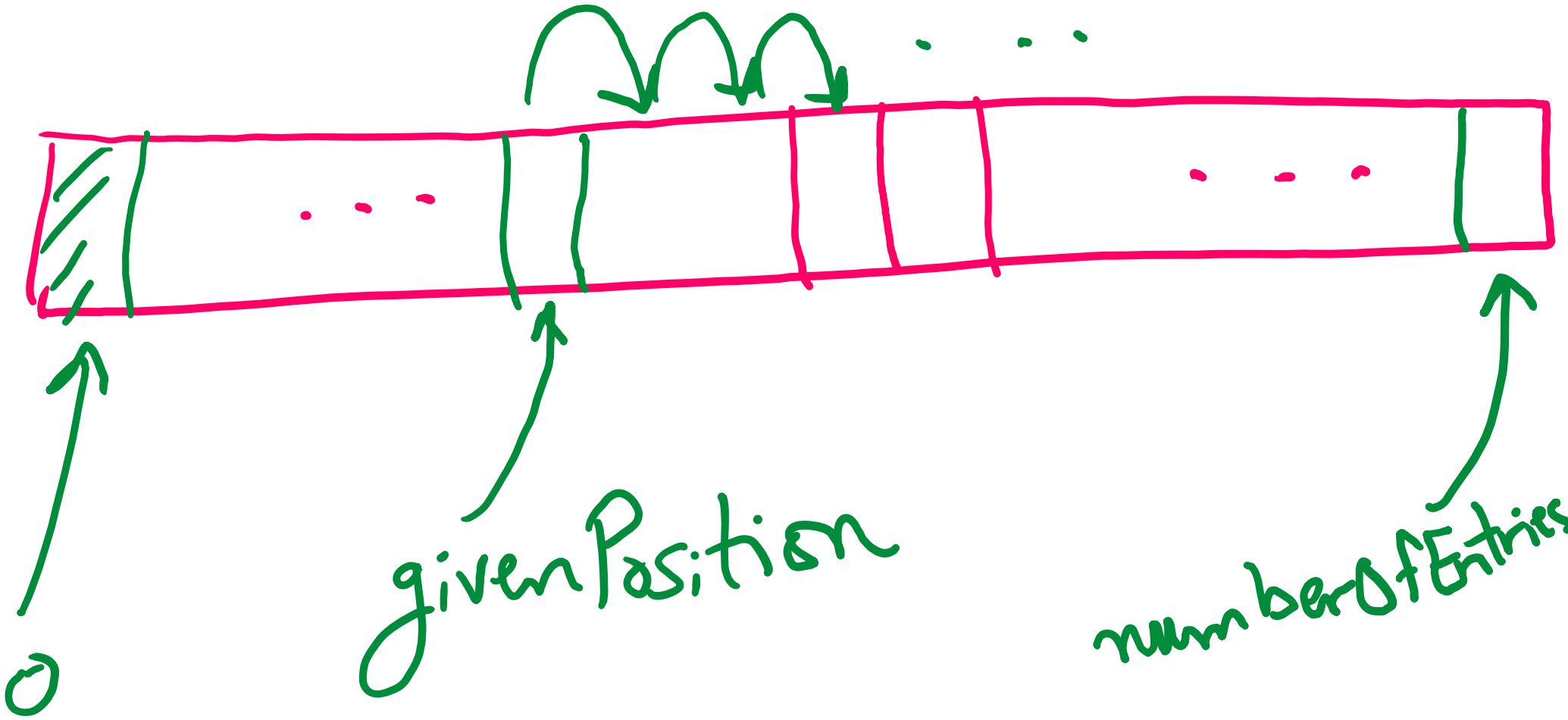
Muddiest Points

- **Q: Why skip 0 in the list?**
- **Q: Why are we doing the +1 after capacity?**
- In the ArrayList implementation, we made a *design decision* to leave entry 0 empty
- The implication is that
  - an array's capacity is (.length – 1) instead of .length
- This is done to match the client perception of the List numbering as starting from 1
  - Positions sent by the client start from 1

Muddiest Points

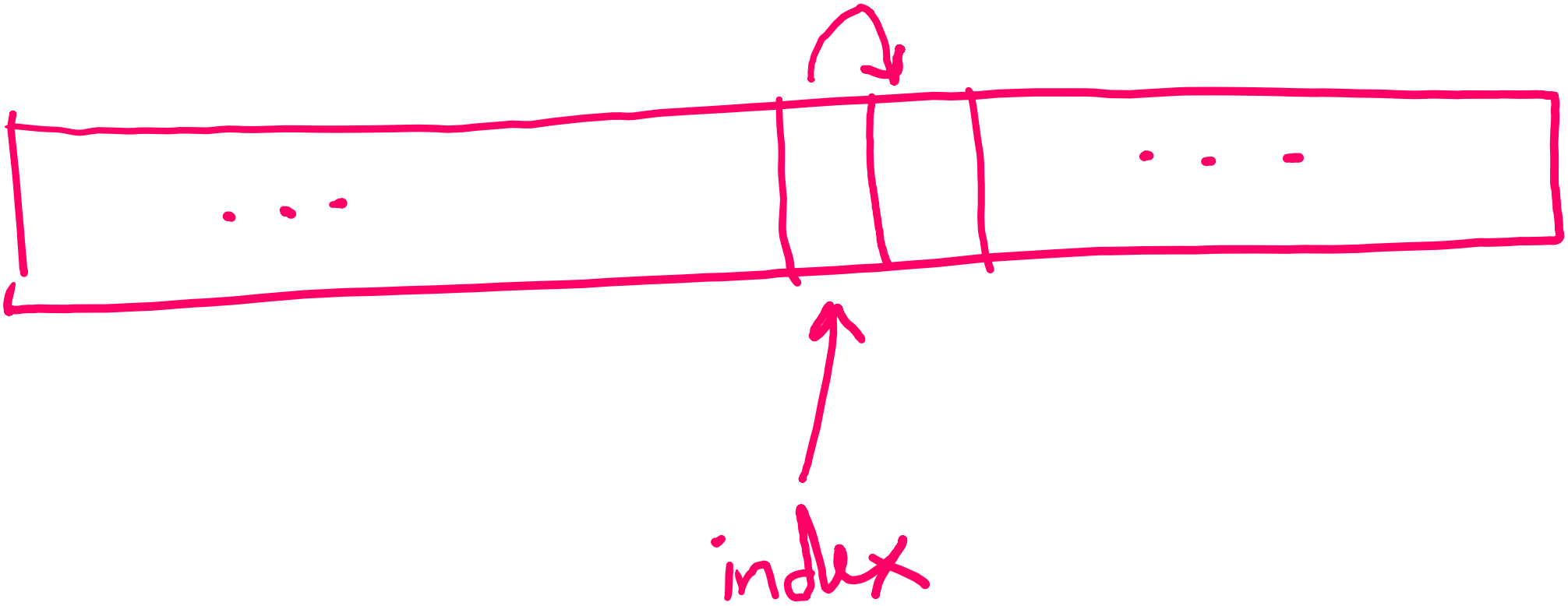
- **Q: how does the adding and shifting work (for an ArrayList)?**
- Since ADT List contains ordered items, if an item is added into any position, we need to shift entries up (i.e., towards end of the array) to make room for the new item
  - Appending is an exception to that. Why?

How to make room for an entry at givenPosition



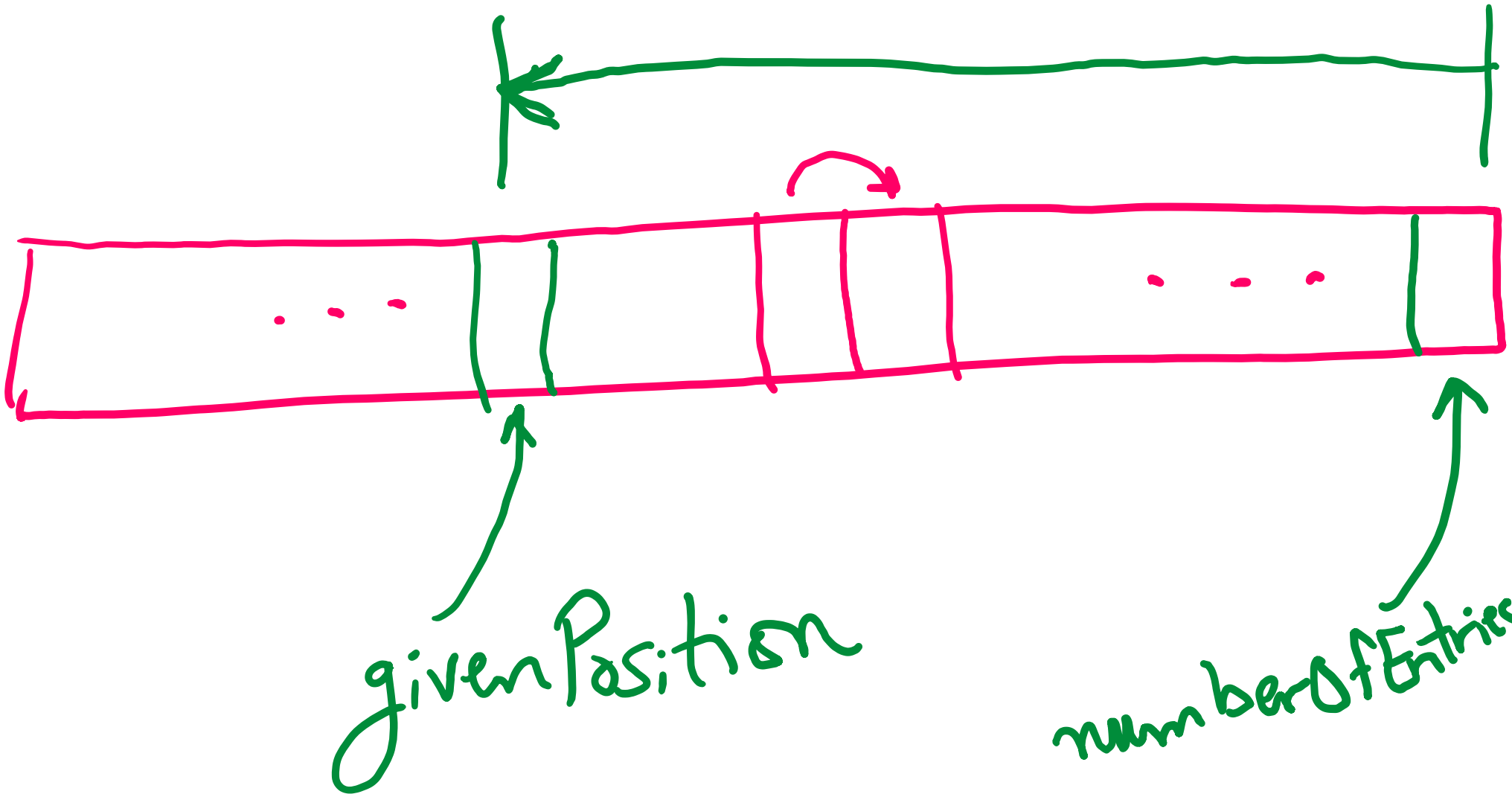
How to make room for an entry at givenPosition

```
list[index + 1] = list[index];
```



What are the bounds of index?

```
list[index + 1] = list[index];
```



Muddiest Points

- **Q: when you throw an exception like IllegalStateException, do you have to define the exception (in a new file) or is it already defined?**
- Some exception types are already defined in the Java libraries (IllegalStateException is one of them)
- You can also define and throw your own exception types in separate files or in the same file (as nested classes)

Today's Agenda

- Big-Oh Approximation
- ADT List
  - resizable array implementation: ArrayList
    - Rest of the methods
  - Linked implementation: LinkedList

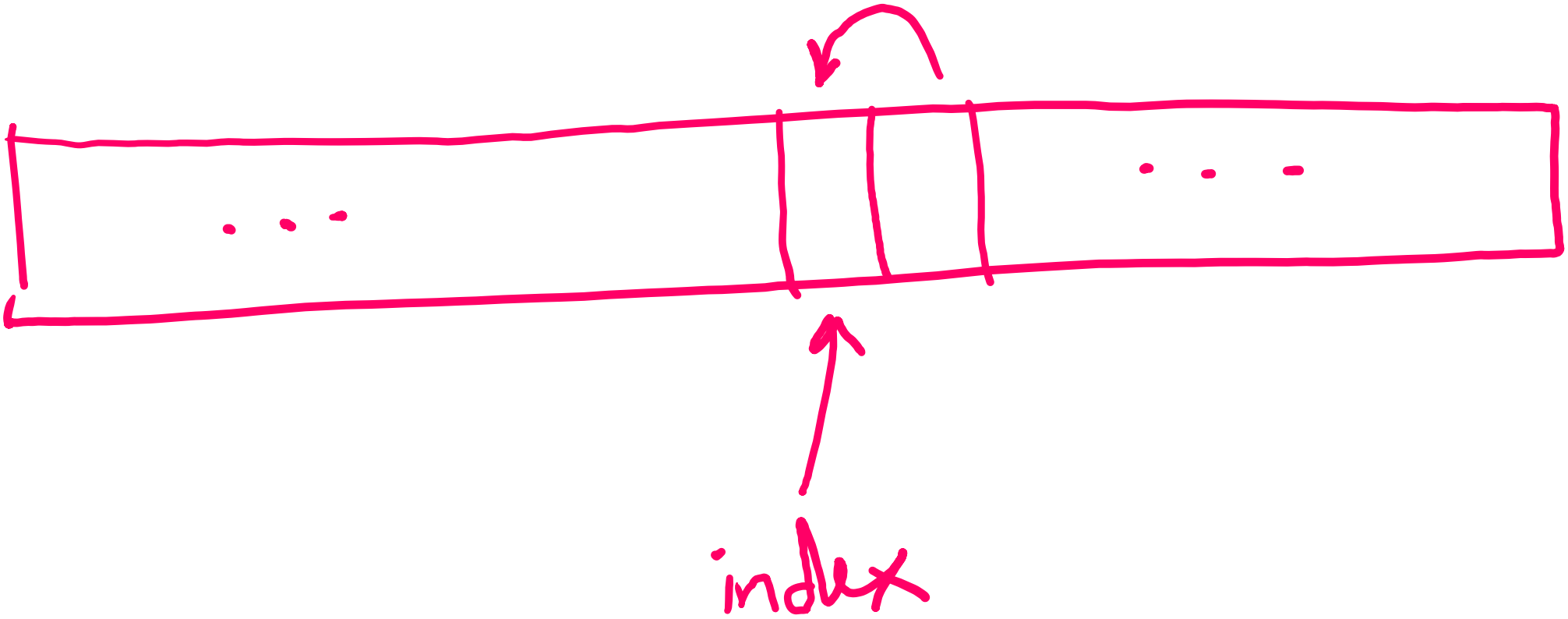
remove

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
```

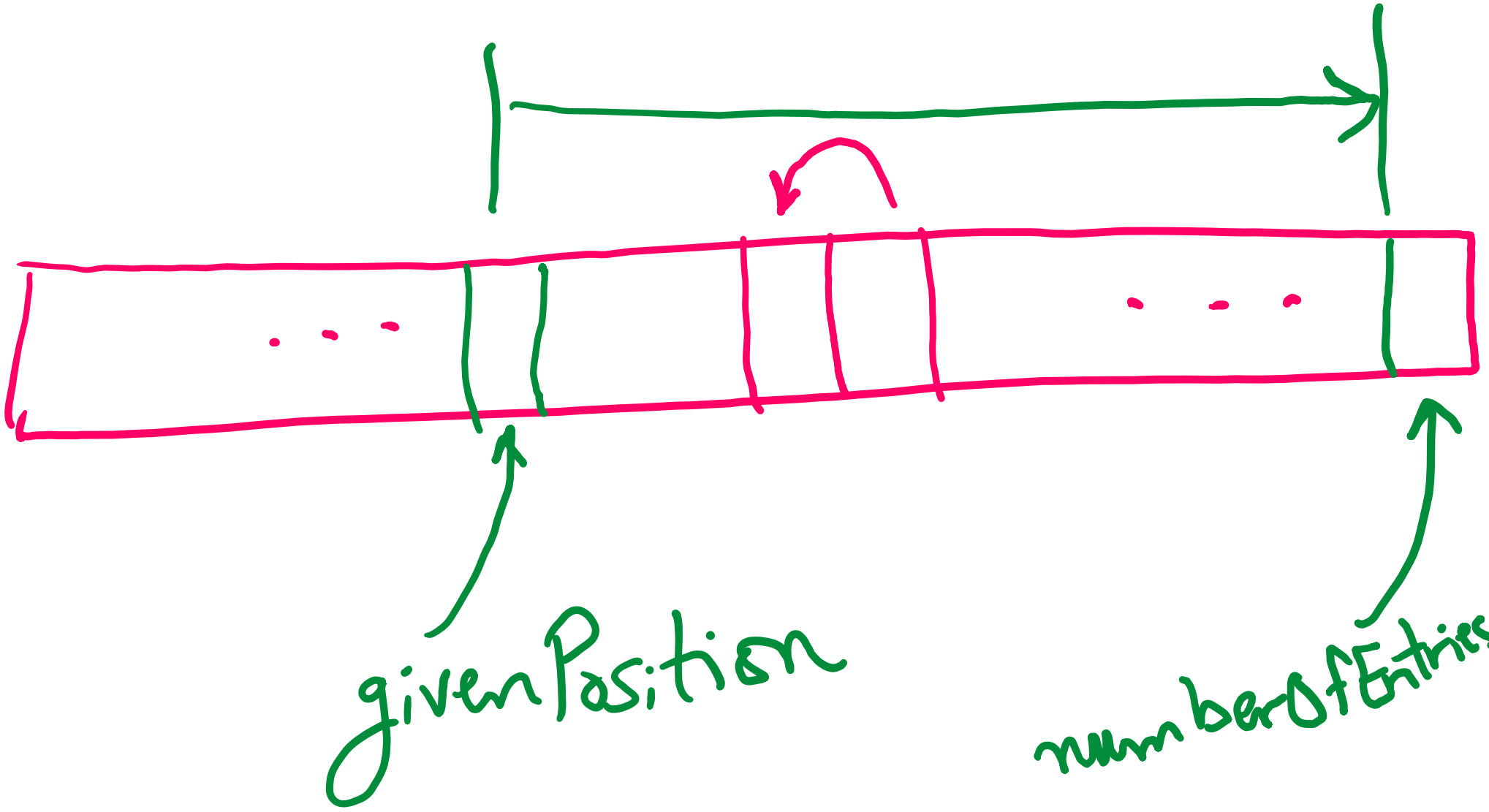
How to remove a gap inside the array

```
list[index] = list[index+1];
```



What are the bounds of index?

```
list[index] = list[index+1];
```



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
```

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
```

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
```

Method contains

```
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
```

Running time of append

- 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
```

Running time of makeRoom

- Running time depends on input size and on client-specified position
- What is the best-case for the specified position?
  - numberOfEntries+1 ➔ same as append → O(1)
- What is the worst-case?
  - 1 ➔ for loop goes from n down to 1 → n iterations → O(n)

```
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
```

Running time of general add

- add calls makeRoom
- Worst-case runtime of add is O(n)

```
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
```

Linked Implementation (LinkedList)

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

Data Fields and Constructor

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

```
1  /**
2     A linked implementation of the ADT list.
3     @author Frank M. Carrano
4  */
5  public class LList<T> implements ListInterface<T>
6  {
7     private Node firstNode; // Reference to first node of chain
8     private int numberOfEntries;
9
10    public LList()
11    {
12       initializeDataFields();
13    } // end default constructor
14
15    public void clear()
16    {
17       initializeDataFields();
18    } // end clear
19    < Implementations of the public methods add, remove, replace, getEntry, contains,
        getLength, isEmpty, and toArray go here. >
20    . . .
21
```

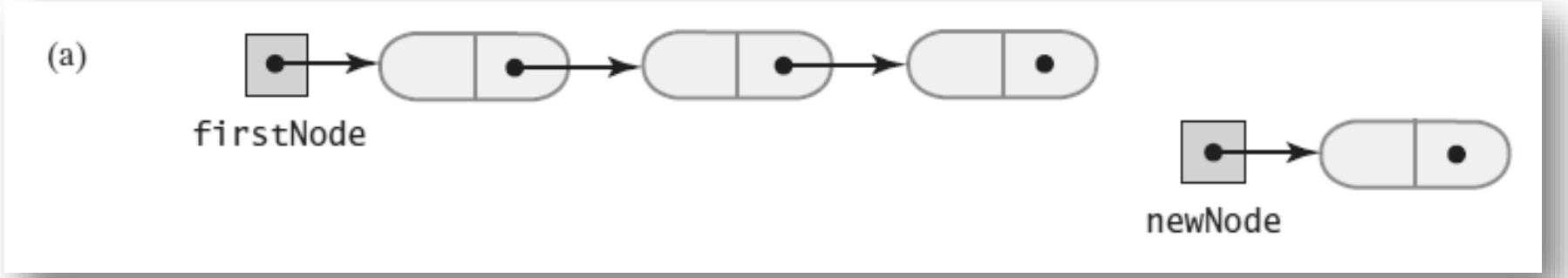
Data Fields and Constructor

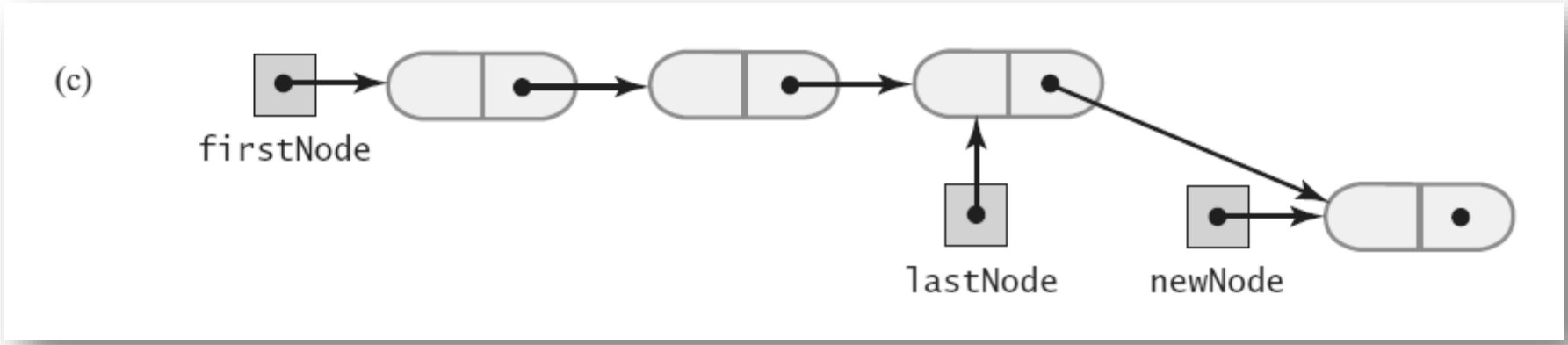
An outline of the class `LList`

```
21
22      // Initializes the class's data fields to indicate an empty list.
23      private void initializeDataFields()
24      {
25         firstNode = null;
26         numberOfEntries = 0;
27      } // end initializeDataFields
28
29      // Returns a reference to the node at a given position.
30      // Precondition: List is not empty;
31      //                1 <= givenPosition <= numberOfEntries.
32      private Node getNodeAt(int givenPosition)
33      {
           < See Segment 14.7. >
34      } // end getNodeAt
35
36      private class Node // Private inner class
37      {
           < See Listing 3-4 in Chapter 3. >
38      } // end Node
39   } // end LList
```

Adding a Node at the end of the chain

A chain of nodes prior to and after adding a node at the end





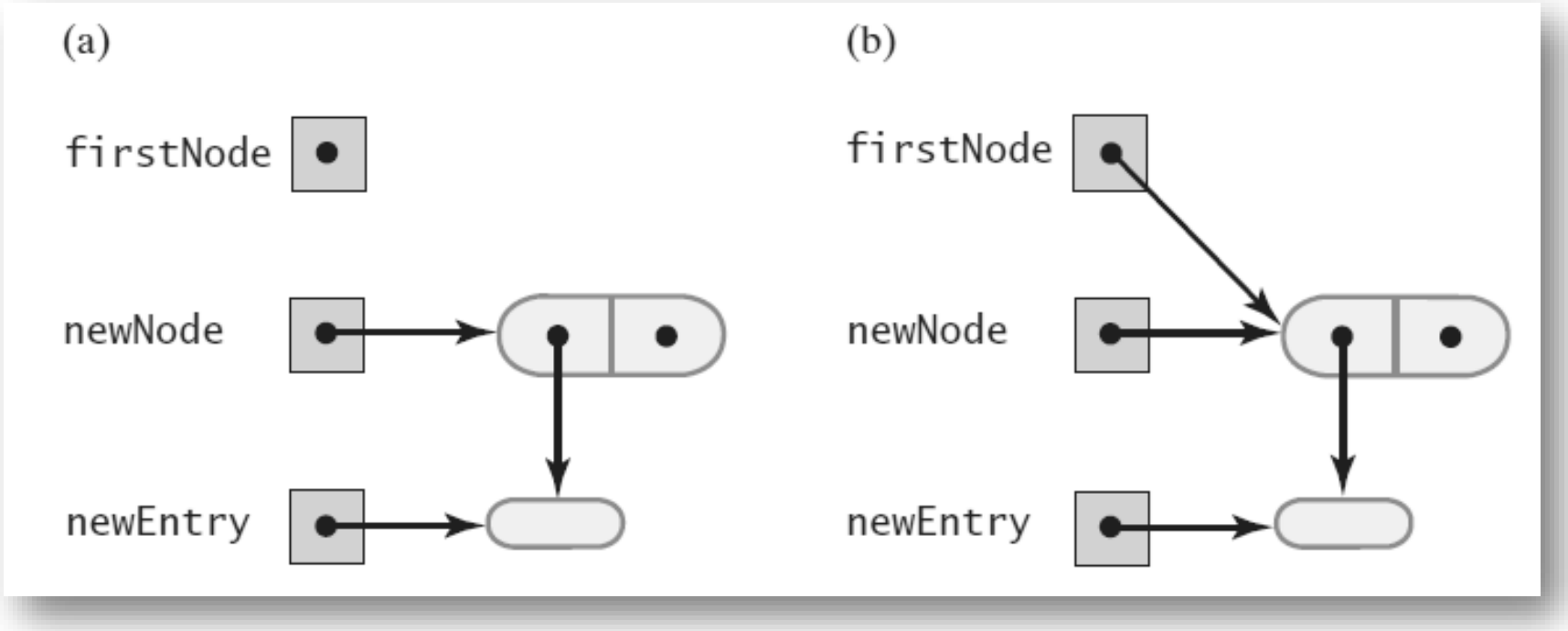
Operations on a chain depended on the method `getNodeAt`

```
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
```

Adding a Node to an empty chain

(a) An empty chain and a new node;

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



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
```

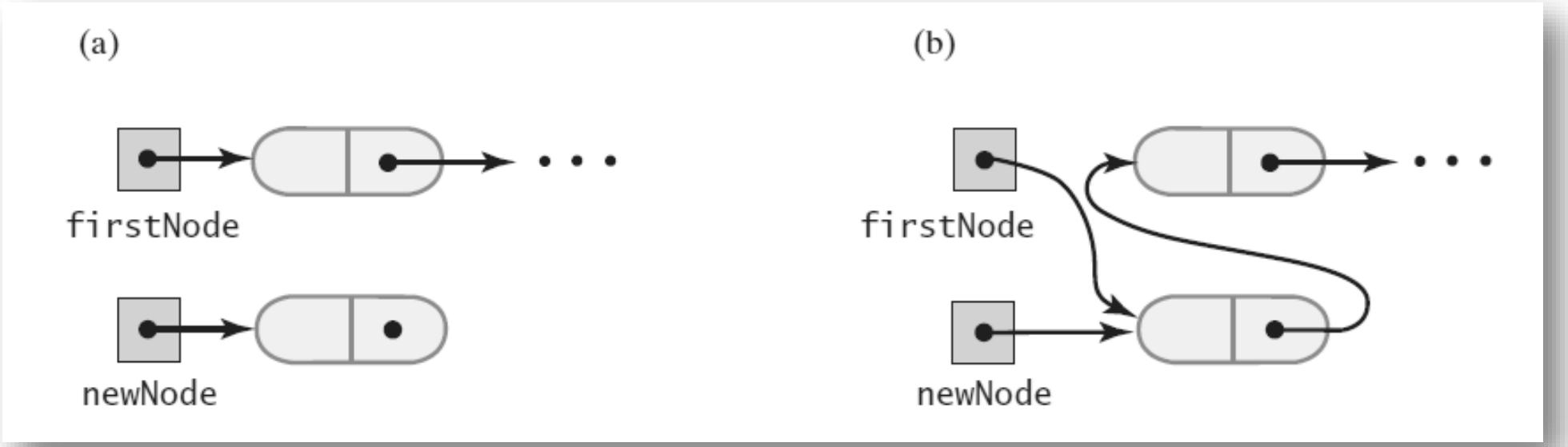
Adding a Node at client-specified position

Possible cases:

- Adding node at chain's beginning
- Adding node between adjacent nodes

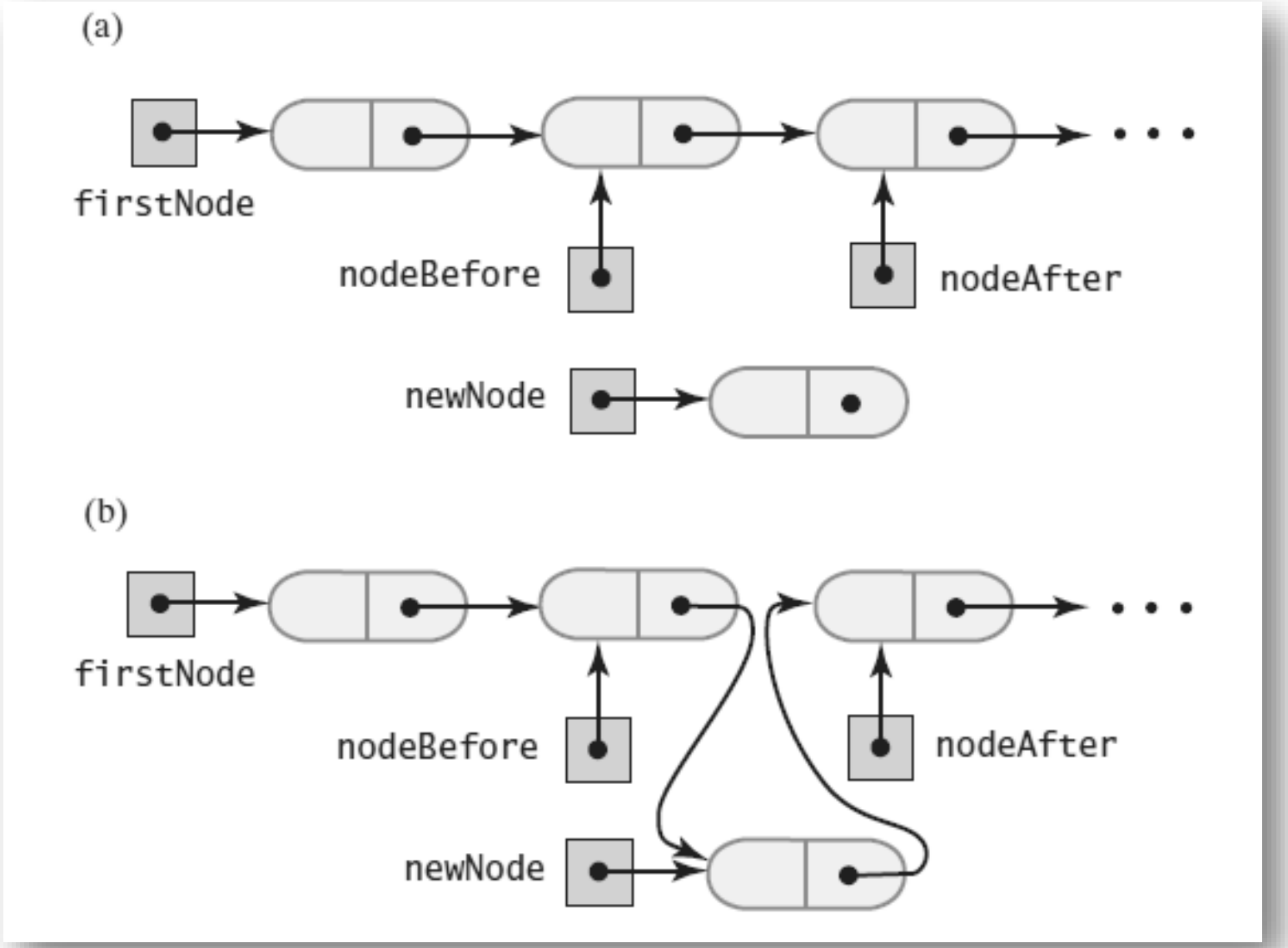
Adding a Node at the beginning of the chain

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 in the middle of the chain

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 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
```

Removing a Node at Various Positions

Possible cases:

1. Removing from an empty chain

   IndexOutOfBounds exception

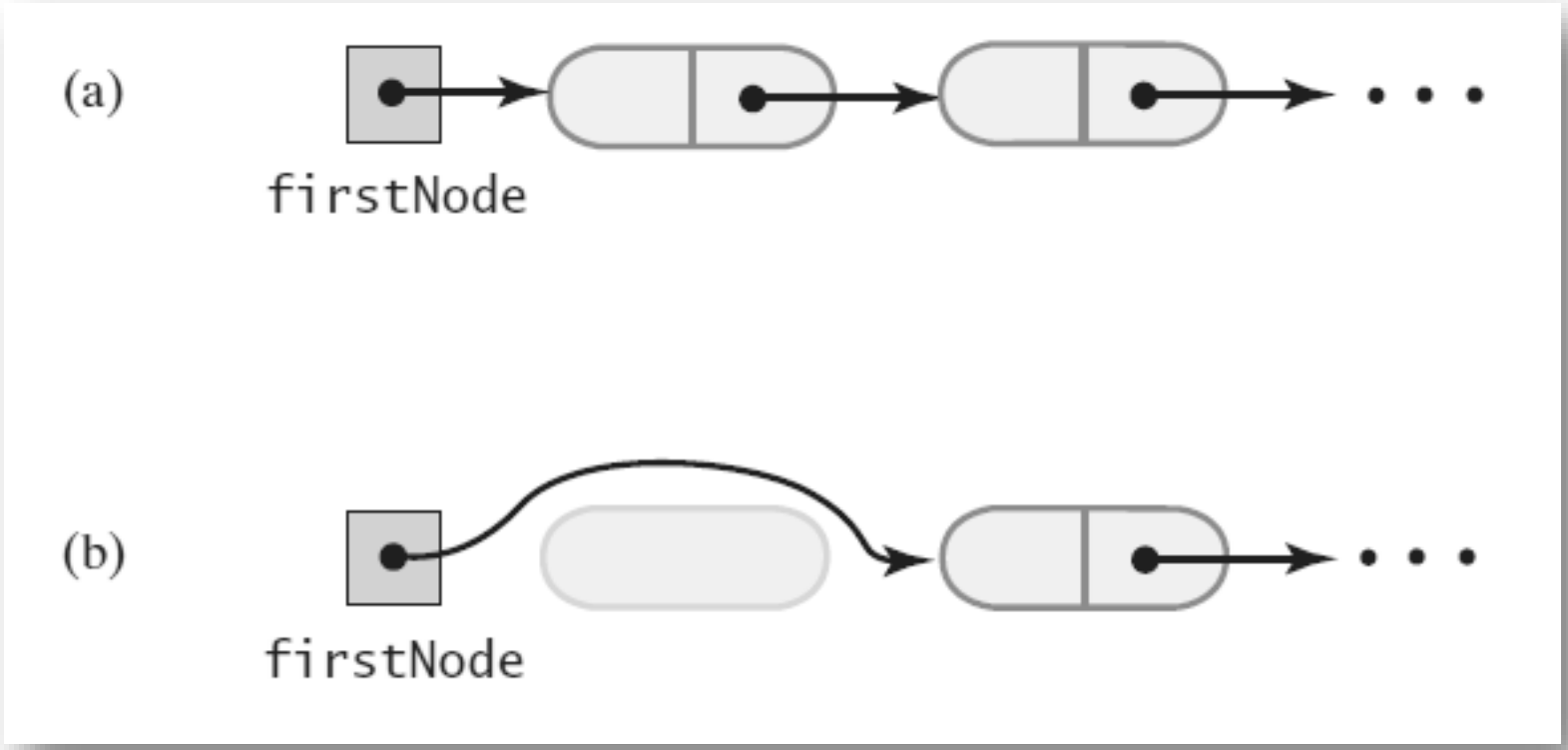
2. Removing the first node

3. Removing the last node

4. Removing a node other than first and last

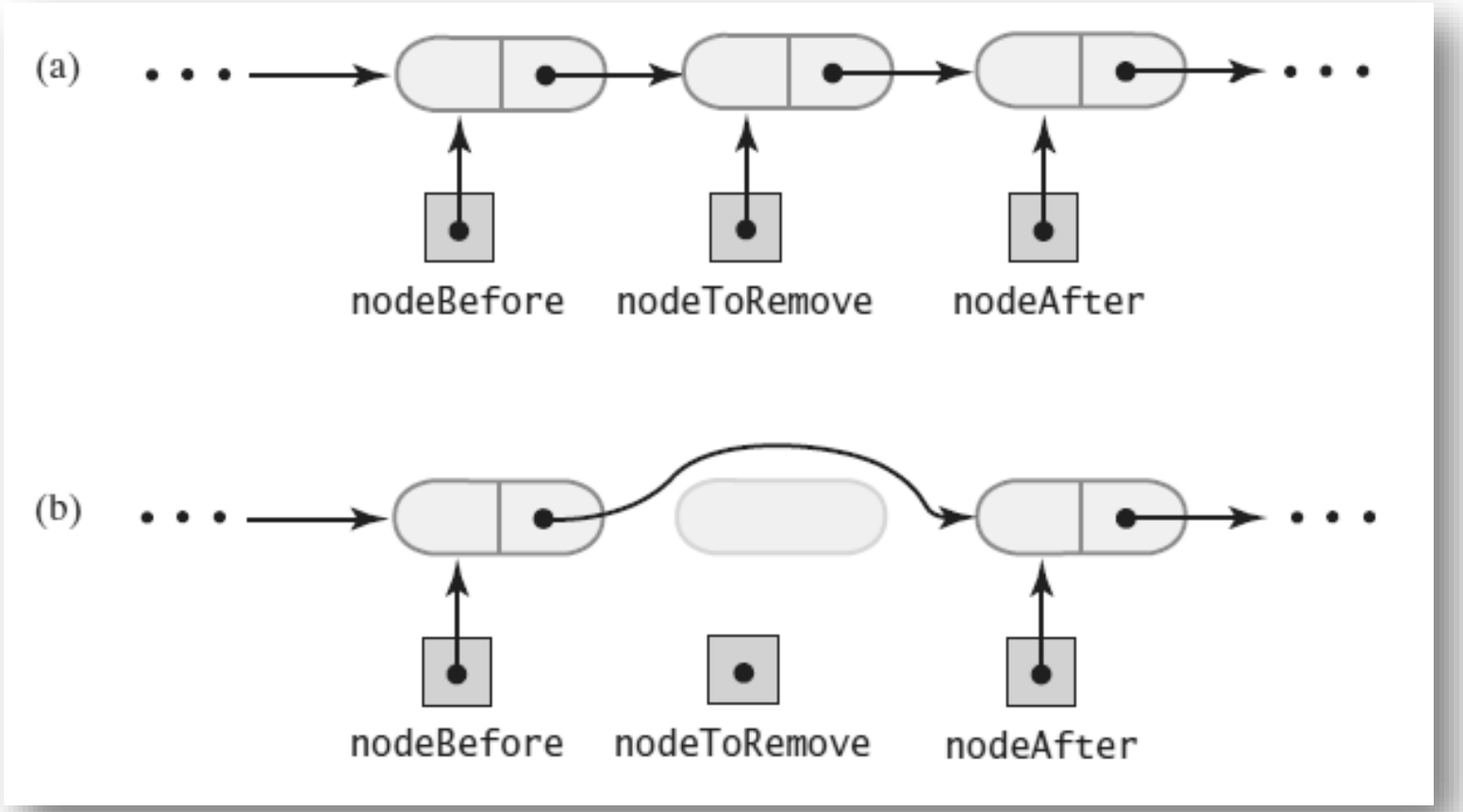
Removing the first node

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



Removing a Node other than first node

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



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
```

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 with items in the list

```
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
```

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 given getNodeAt

```
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
```