CS 570: Data Structures

Collections Framework: Linked Lists

Instructor: Iraklis Tsekourakis

Email: itsekour@stevens.edu

CHAPTER 2 (PART 2)

Lists and the
`Collections` Framework

Week 5

- Reading Assignment: Koffman and Wolfgang, Sections 2.5-2.10

Single-Linked Lists

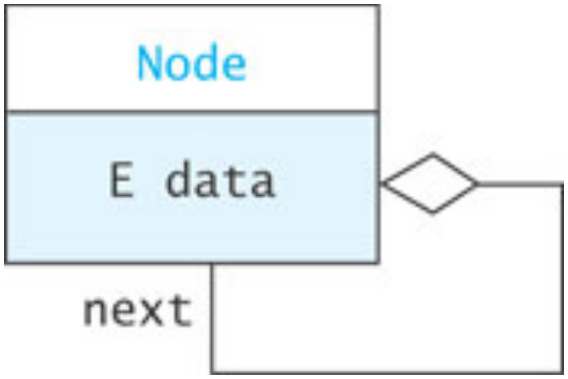
Section 2.5

Single-Linked Lists

- A linked list is useful for inserting and removing at arbitrary locations
- The `ArrayList` is limited because its `add` and `remove` methods operate in linear (O(*n*)) time—requiring a loop to shift elements
- A linked list can add and remove elements at a known location in O(1) time
- In a linked list, instead of an index, each element is linked to the following element

A List Node

- A node can contain:
  - a data item
  - one or more links
- A link is a reference to a list node
- In our structure, the node contains a data field named `data` of type `E`
- and a reference to the next node, named `next`



List Nodes for Single-Linked Lists


```
private static class Node<E> {
  private E data;
  private Node<E> next;

  /** Creates a new node with a null next field
      @param dataItem  The data stored
  */
  private Node(E dataItem) {
    data = dataItem;
    next = null;
  }
}
```

List Nodes for Single-Linked Lists


```
 /** Creates a new node that references
another node
      @param dataItem  The data stored
      @param nodeRef  The node referenced by
          new node
  */
 private Node(E dataItem, Node<E> nodeRef) {
    data = dataItem;
    next = nodeRef;
  }
}
```

List Nodes for Single-Linked Lists (cont.)


```
private static class Node<E> {
  private E data;
  private Node<E> next;

  /** Creates a new node with a
      @param dataItem  The data
  */
  private Node(E data) {
    data = dataItem;
    next = null;
  }
```

The keyword `static` indicates that the `Node<E>` class will not reference its outer class

Static inner classes are also called *nested classes*

List Nodes for Single-Linked Lists (cont.)


```
private static class Node<E> {
  private E data;
  private Node<E> next;

  /** Creates a new node with a nul
      @param dataItem  The data sto
  */
  private Node(E dataItem) {
    data = dataItem;
    next = null;
  }
}
```

Generally, all details of the `Node` class should be private. This applies also to the data fields and constructors.

Connecting Nodes (cont.)

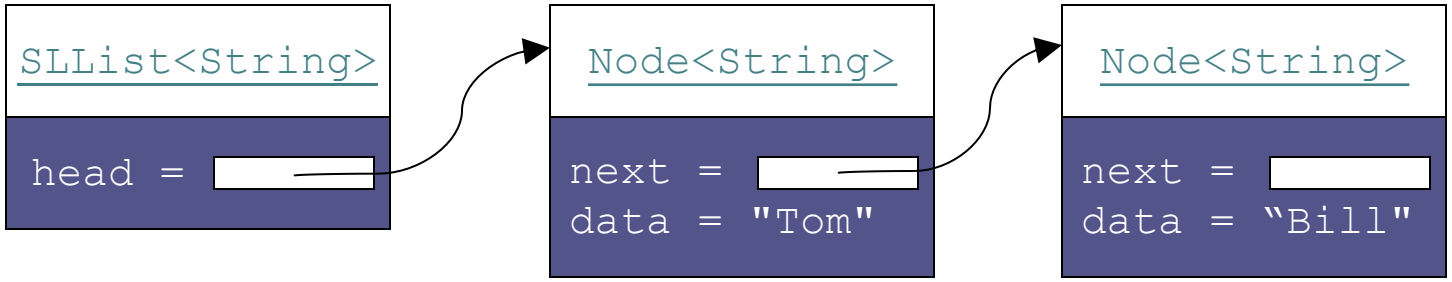

```
Node<String> tom = new Node<String>("Tom");
Node<String> bill = new Node<String>(“Bill");
Node<String> harry = new
                  Node<String>("Harry");
Node<String> sam = new Node<String>("Sam");


tom.next = bill;
bill.next = harry;
harry.next = sam;
```

A Single-Linked List Class

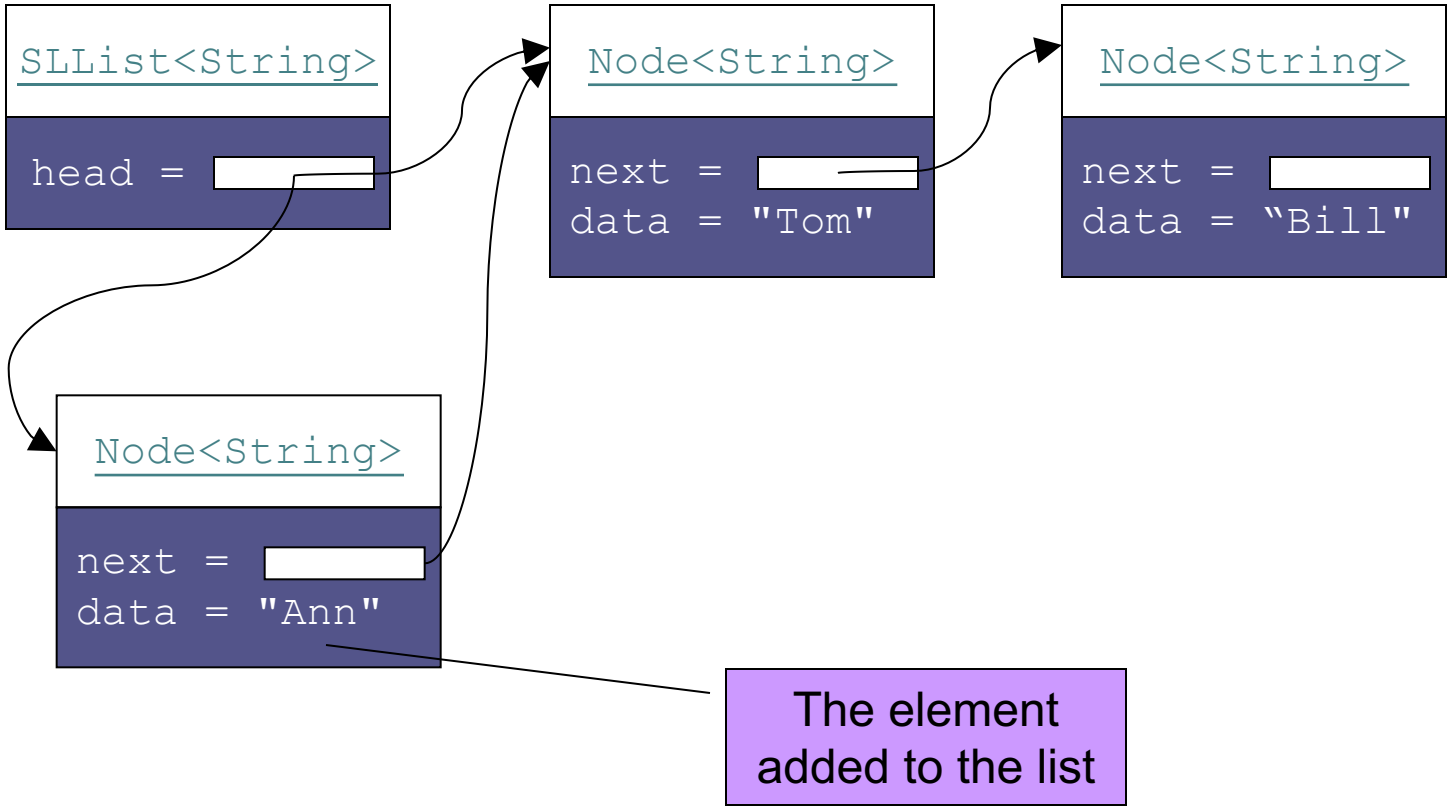
- Generally, we do not have individual references to each node
- A `SingleLinkedList` object has a data field `head`, the *list head*, which references the first list node

```
public class SingleLinkedList<E> {
  private Node<E> head = null;
  private int size = 0;
  ...
}
```

SLList: An Example List



Implementing `SLList.addFirst(E item)`



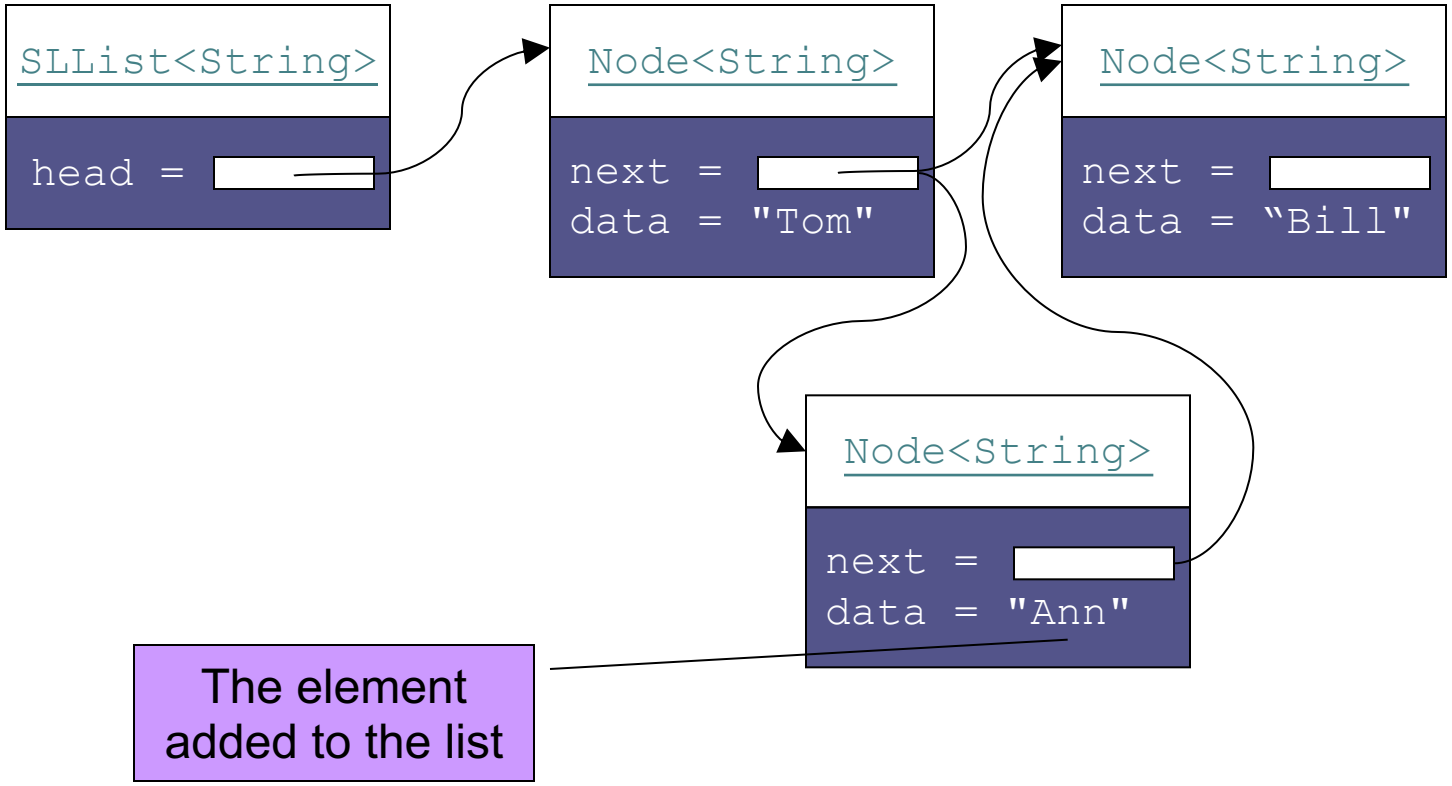
Implementing `SLList.addFirst(E item)` (cont.)


```
private void addFirst (E item) {
  Node<E> temp = new Node<E>(item, head);
  head = temp;
  size++;
}
```

or, more simply

```
private void addFirst (E item) {
  head = new Node<E>(item, head);
  size++;
}
```

Implementing `addAfter(Node<E> node, E item)`



Implementing `addAfter(Node<E> node, E item)` (cont.)

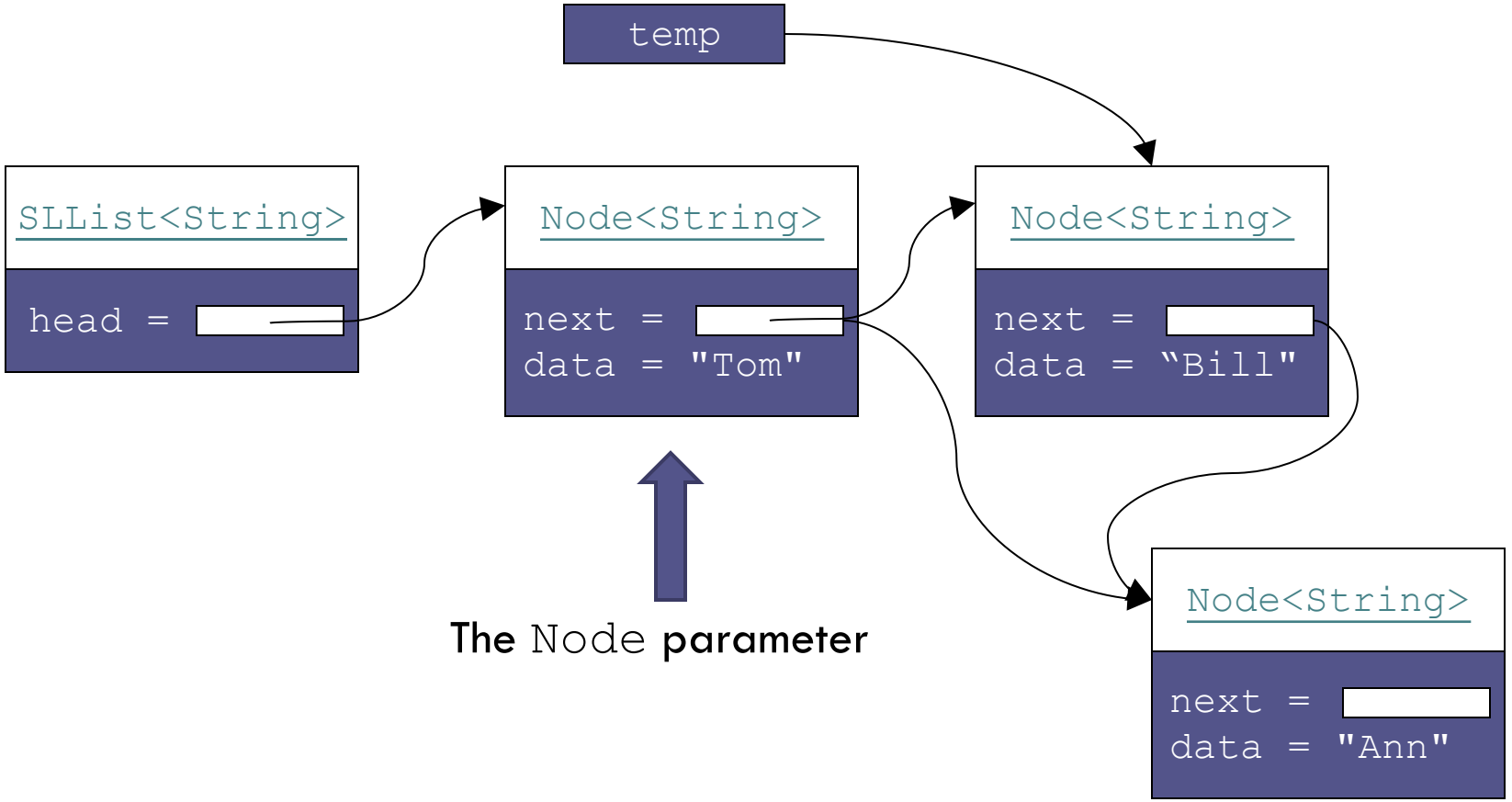

```
private void addAfter (Node<E> node, E item) {
  Node<E> temp = new Node<E>(item, node.next);
  node.next = temp;
  size++;
}
```

or, more simply ...

```
private void addAfter (Node<E> node, E item) {
  node.next = new Node<E>(item, node.next);
  size++;
}
```

We declare this method `private` since it should not be called from outside the class. Later we will see how this method is used to implement the public add methods.

Implementing `removeAfter(Node<E> node)`

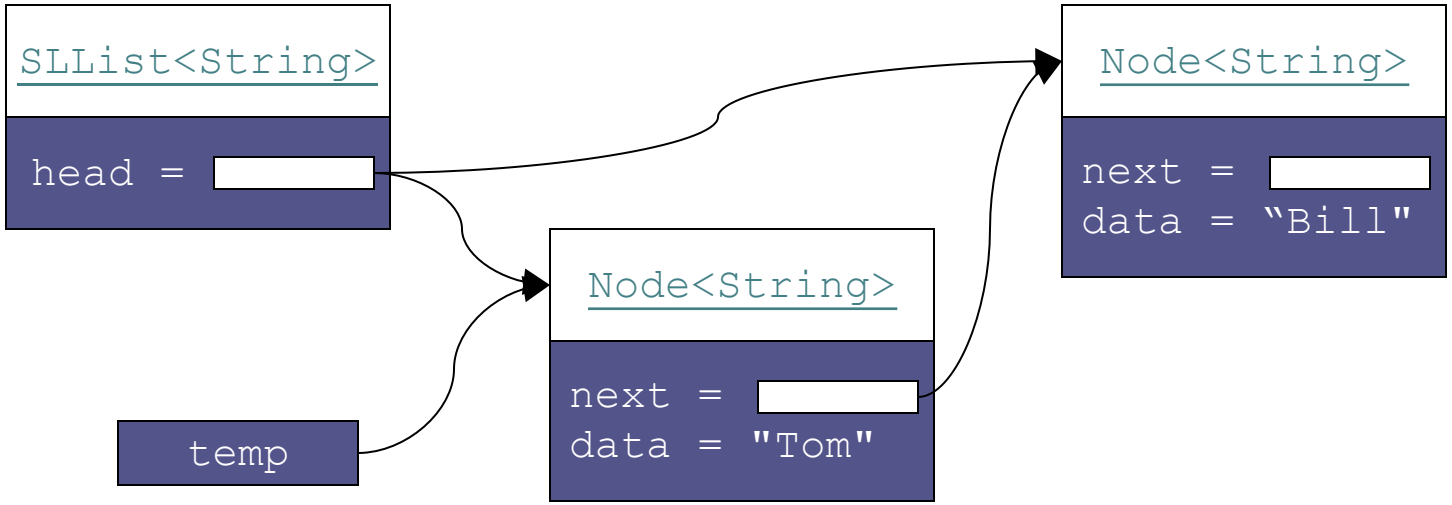


Implementing `removeAfter(Node<E> node)` (cont.)


```
private E removeAfter (Node<E> node)
{
  Node<E> temp = node.next;
  if (temp != null) {
    node.next = temp.next;
    size--;
    return temp.data;
  } else {
    return null;
  }
}
```

Implementing

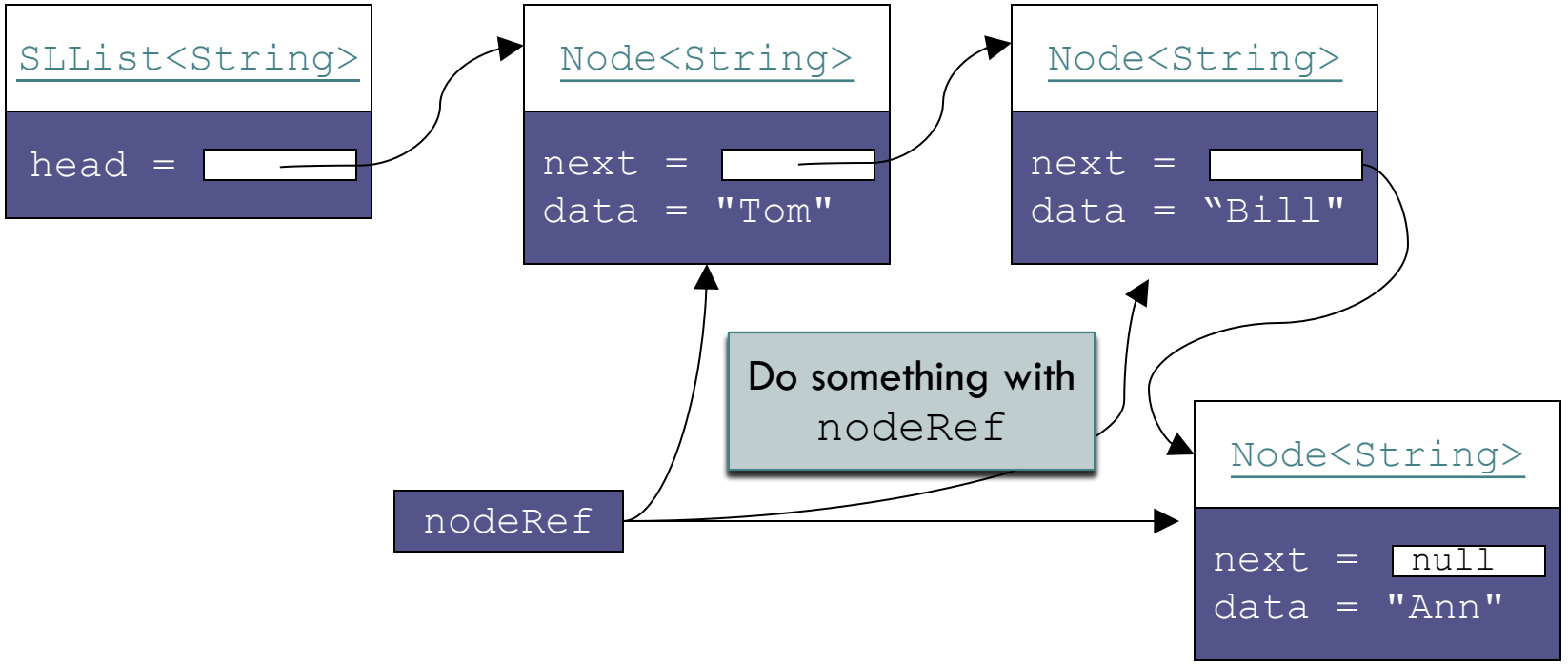
SLList.removeFirst()



Implementing SLList.removeFirst() (cont.)


```
private E removeFirst () {
   Node<E> temp = head;
   if (head != null) {
      head = head.next;
      size--;
      return temp.data
   } else {
      return null;
   }
}
```

Traversing a Single-Linked List



Traversing a Single-Linked List (cont.)


```
public String toString() {
  Node<String> nodeRef = head;
  StringBuilder result = new StringBuilder();
  while (nodeRef != null) {
    result.append(nodeRef.data);
    if (nodeRef.next != null) {
      result.append(" ==> ");
    }
    nodeRef = nodeRef.next;
  }
  return result.toString();
}
```

SLList.getNode(int)

- In order to implement methods required by the List interface, we need an additional helper method:

```
private Node<E> getNode(int index) {
   Node<E> node = head;
   for (int i=0; i<index && node != null;
       i++) {
     node = node.next;
   }
   return node;
}
```

Completing the `SingleLinkedList` Class

Method	Behavior
`public E get(int index)`	Returns a reference to the element at position `index`.
`public E set(int index, E anEntry)`	Sets the element at position `index` to reference `anEntry`. Returns the previous value.
`public int size()`	Gets the current size of the `List`.
`public boolean add(E anEntry)`	Adds a reference to `anEntry` at the end of the `List`. Always returns `true`.
`public void add(int index, E anEntry)`	Adds a reference to `anEntry`, inserting it before the item at position `index`.
`int indexOf(E target)`	Searches for `target` and returns the position of the first occurrence, or –1 if it is not in the `List`.

public E get(int index)


```
public E get (int index) {
 if (index < 0 || index >= size) {
    throw new
       IndexOutOfBoundsException(Integer.toString(index));
  }
  Node<E> node = getNode(index);
  return node.data;
}
```

public E set(int index, E newValue)


```
public E set (int index, E anEntry) {
  if (index < 0 || index >= size) {
    throw new
     IndexOutOfBoundsException(Integer.toString
          (index));
  }
  Node<E> node = getNode(index);
  E result = node.data;
  node.data = newValue;
  return result;
}
```

public void add(int index, E item)


```
public void add (int index, E item) {
 if (index < 0 || index > size) {
   throw new
     IndexOutOfBoundsException(Integer.toString
           (index));
 }
 if (index == 0) {
   addFirst(item);
 } else {
   Node<E> node = getNode(index-1);
   addAfter(node, item);
 }
}
```

public boolean add(E item)

- To add an item to the end of the list

```
public boolean add (E item) {
   add(size, item);
   return true;
  }
```

Performance of `SinglyLinkedList`

- The `set` and `get` methods:
- Inserting or removing general elements:
- Adding at the beginning:
- Adding at the end:

Double-Linked Lists and Circular Lists

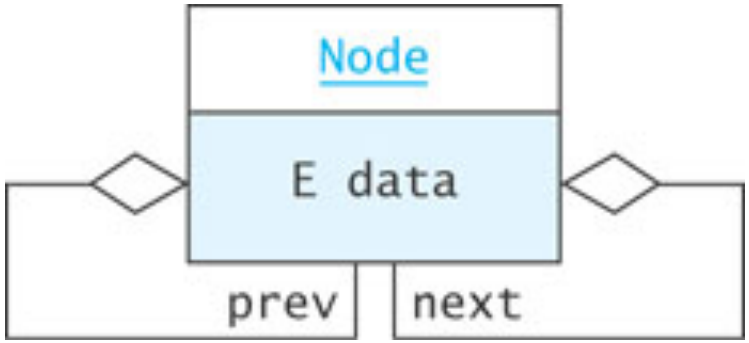
Section 2.6

Double-Linked Lists

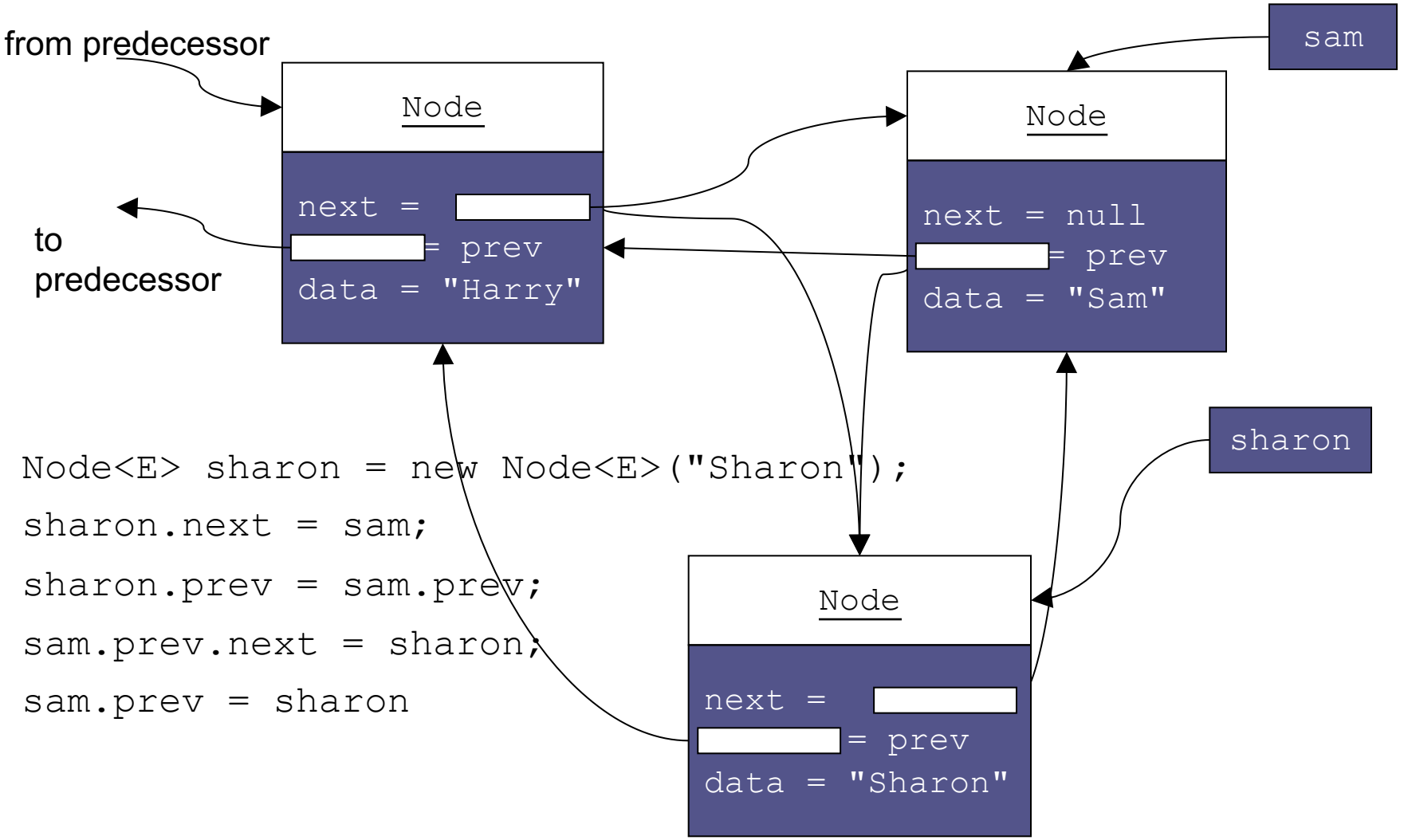
- Limitations of a singly-linked list include:
  - Insertion at the front is O(1); insertion at other positions is O(*n*)
  - Insertion is convenient only after a referenced node
  - Removing a node requires a reference to the previous node
  - We can traverse the list only in the forward direction
- We can overcome some of these limitations:
  - Add a reference in each node to the previous node, creating a *double-linked list*

Node **Class**

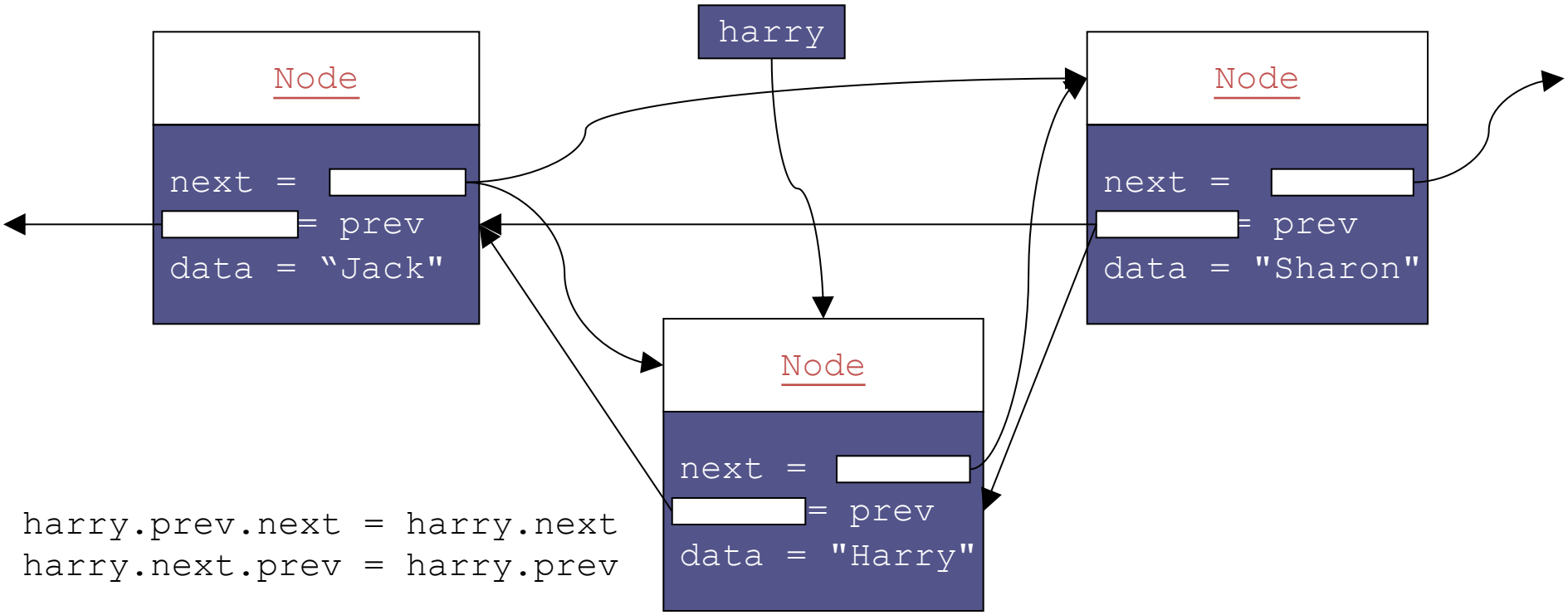

```
private static class Node<E> {
   private E data;
   private Node<E> next = null;
   private Node<E> prev = null;

   private Node(E dataItem) {
      data = dataItem;
   }
}
```



Inserting into a Double-Linked List



Removing from a Double-Linked List



A Double-Linked List Class

- So far we have worked only with internal nodes
- As with the single-linked class, it is best to access the internal nodes with a double-linked list object

- A double-linked list object has data fields:
  - `head` (a reference to the first list `Node`)
  - `tail` (a reference to the last list `Node`)
  - `size`
- Insertion at either end is O(1); insertion elsewhere is still O(*n*)

Circular Lists

- Circular double-linked list:
  - Link last node to the first node, and
  - Link first node to the last node
- We can also build singly-linked circular lists:
  - Traverse in forward direction only
- **Advantages:**
  - Continue to traverse even after passing the first or last node
  - Visit all elements from any starting point
  - Never fall off the end of a list
- **Disadvantage:** Code must avoid an infinite loop!

The `LinkedList` Class and the `Iterator`, `ListIterator`, and `Iterable` Interfaces

Section 2.7

The `LinkedList` Class

Method	Behavior
`public void add(int index, E obj)`	Inserts object `obj` into the list at position `index`.
`public void addFirst(E obj)`	Inserts object `obj` as the first element of the list.
`public void addLast(E obj)`	Adds object `obj` to the end of the list.
`public E get(int index)`	Returns the item at position `index`.
`public E getFirst()`	Gets the first element in the list. Throws `NoSuchElementException` if the list is empty.
`public E getLast()`	Gets the last element in the list. Throws `NoSuchElementException` if the list is empty.
`public boolean remove(E obj)`	Removes the first occurrence of object `obj` from the list. Returns `true` if the list contained object `obj`; otherwise, returns `false`.
`public int size()`	Returns the number of objects contained in the list.

The Iterator

- An iterator can be viewed as a moving place marker that keeps track of the current position in a particular linked list
- An `Iterator` object for a list starts at the list head
- The programmer can move the `Iterator` by calling its `next` method.
- The `Iterator` stays on its current list item until it is needed
- An `Iterator` traverses in $O(n)$ while a list traversal using `get()` calls in a linked list is $O(n^2)$

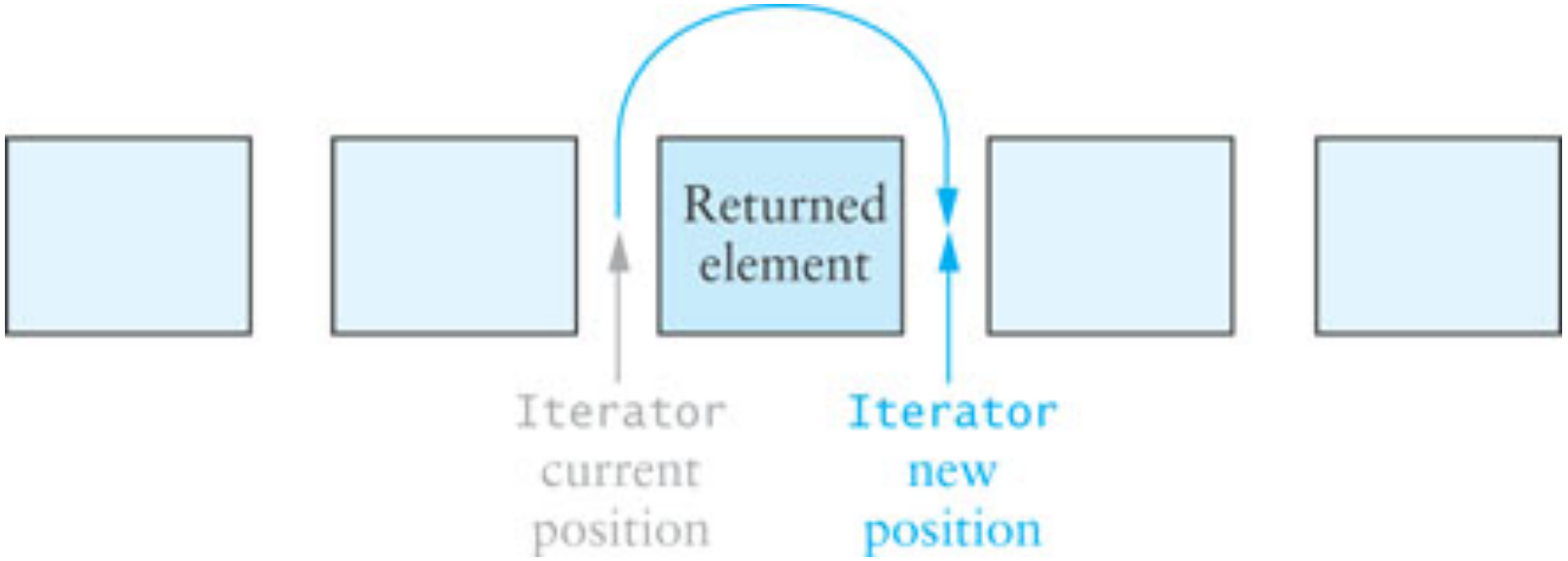
`Iterator` Interface

- The `Iterator` interface is defined in `java.util`
- The `List` interface declares the method `iterator` which returns an `Iterator` object that iterates over the elements of that list

Method	Behavior
`boolean hasNext()`	Returns `true` if the `next` method returns a value.
`E next()`	Returns the next element. If there are no more elements, throws the `NoSuchElementException`.
`void remove()`	Removes the last element returned by the `next` method.

`Iterator` **Interface** (cont.)

- An `Iterator` is conceptually *between* elements; it does not refer to a particular object at any given time



`Iterator` **Interface** (cont.)

- In the following loop, we process all items in `List<Integer>` through an `Iterator`

```
Iterator<Integer> iter = aList.iterator();
while (iter.hasNext()) {
  int value = iter.next();
  // Do something with value
  ...
}
```

Enhanced `for` Statement

- Java 5.0 introduced an enhanced `for` statement
- The enhanced `for` statement creates an `Iterator` object and implicitly calls its `hasNext` and `next` methods
- Other `Iterator` methods, such as `remove`, are not available

Enhanced `for` Statement (cont.)

- The following code counts the number of times `target` occurs in `myList` (type `LinkedList<String>`)

```
count = 0;
for (String nextStr : myList) {
  if (target.equals(nextStr)) {
    count++;
  }
}
```

Enhanced `for` Statement (cont.)

- The enhanced for statement can also be used with arrays, in this case, `chars` or type `char[]`

```
for (char nextCh : chars) {
  System.out.println(nextCh);
}
```

Implementation of a Double-Linked List Class

Section 2.8

- Makes heavy use of iterators
- Can also be implemented using previous/next references

KWLinkedList

- We will define a `KWLinkedList` class which implements some of the methods of the `List` interface
- The `KWLinkedList` class is for demonstration purposes only; Java provides a standard `LinkedList` class in `java.util` which you should use in your programs (after this course)

Data Field	Attribute
`private Node<E> head`	A reference to the first item in the list
`private Node<E> tail`	A reference to the last item in the list
`private int size`	A count of the number of items in the list

KWLinkedList (cont.)


```
import java.util.*;

/** Class KWLinkedList implements a double linked list
 * and a ListIterator. */

public class KWLinkedList <E> {
    // Data Fields
    private Node <E> head = null;

    private Node <E> tail = null;

    private int size = 0;

    . . .
```

`Add` **Method**

1. Obtain a reference, `nodeRef`, to the node at position `index`
2. Insert a new `Node` containing `obj` before the node referenced by `nodeRef`

To use a `ListIterator` object to implement add:

1. Obtain an iterator that is positioned just before the `Node` at position `index`
2. Insert a new `Node` containing `obj` before the `Node` currently referenced by this iterator

Add **Method**


```
/** Add an item at the specified index.
    @param index The index at which the object is
           to be inserted
    @param obj The object to be
           inserted
    @throws
       IndexOutOfBoundsException
           if the index is out of range
           (i < 0 || i > size())
*/
public void add(int index, E obj) {
  listIterator(index).add(obj);
}
```

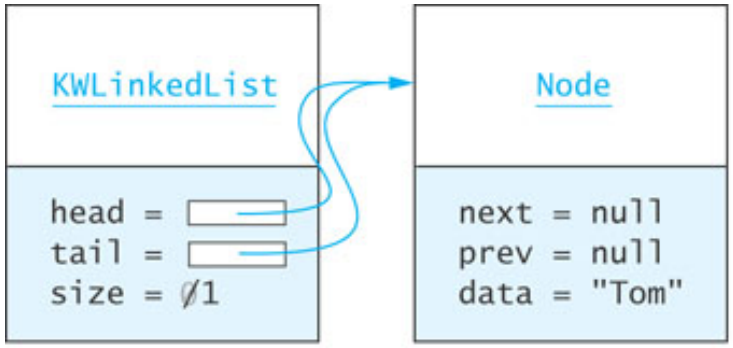
Other Add and Get Methods


```
public void addFirst(E item) {
  add(0, item);
}

public void addLast(E item) {
  add(size, item);
}

public E getFirst() {
  return head.data;
}

public E getLast() {
  return tail.data;
}
```

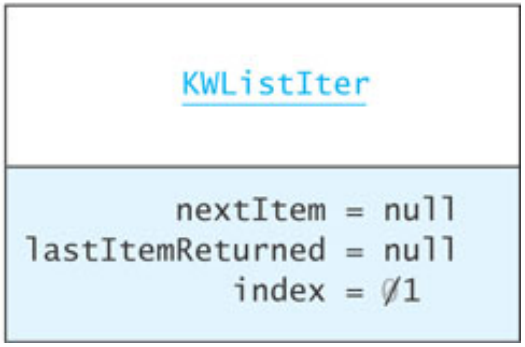
The `Add` Method

- When adding, there are four cases to address:
  - Add to an empty list
  - Add to the head of the list
  - Add to the tail of the list
  - Add to the middle of the list

Adding to an Empty List

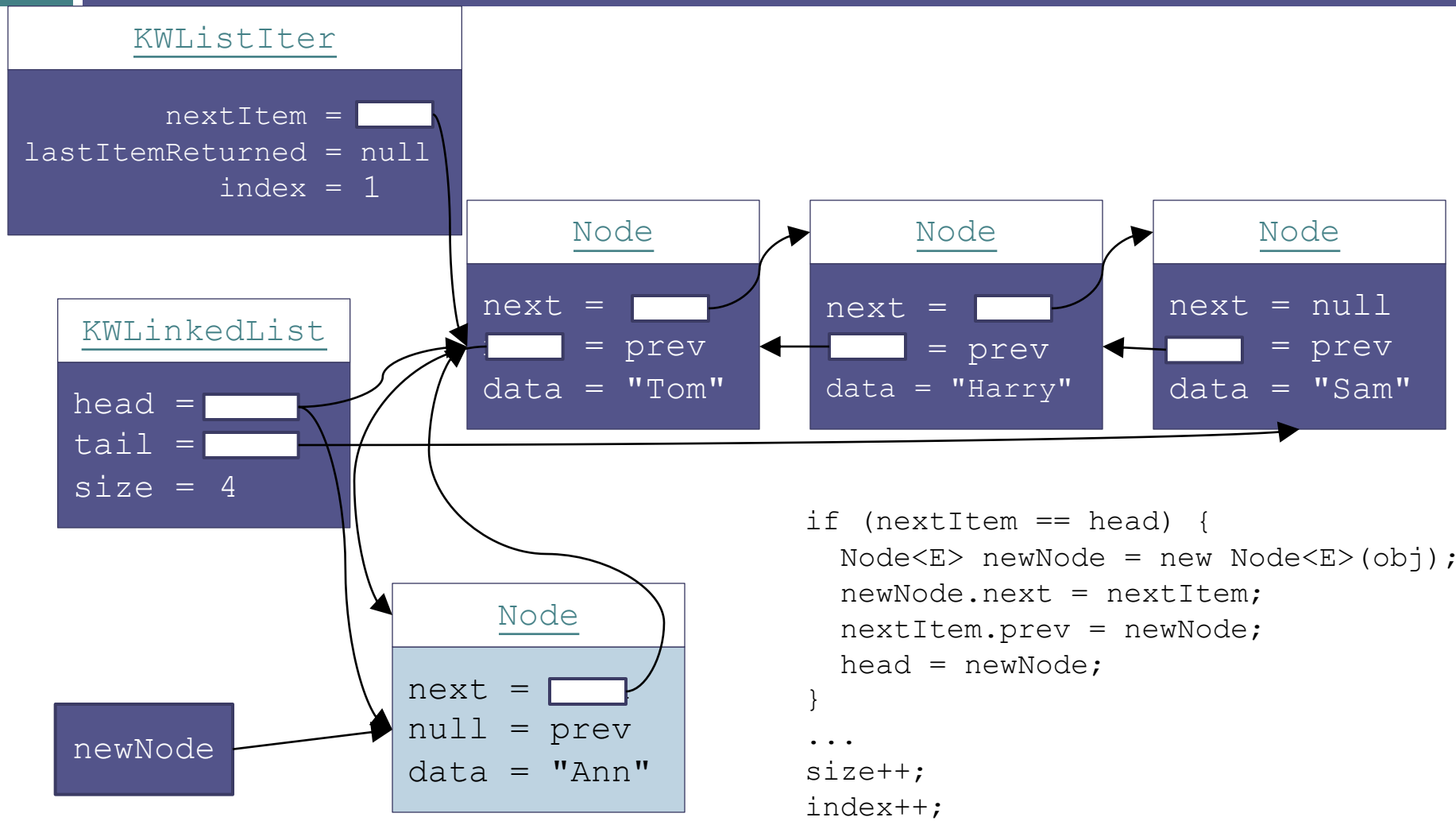


(after insertion)



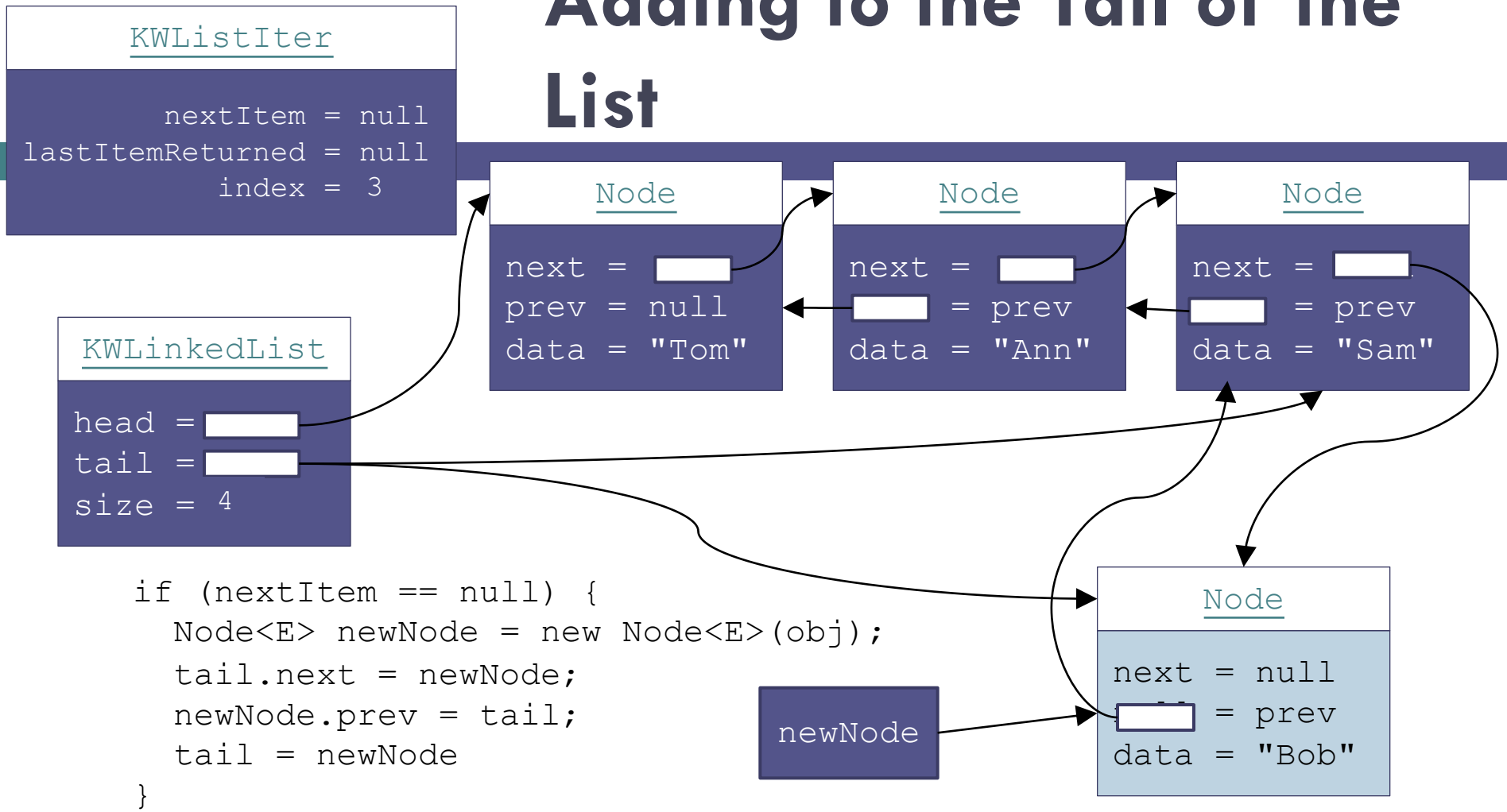

```
if (head == null) {
   head = new Node<E>(obj);
   tail = head;
}
...
size++
```

Adding to the Head of the List



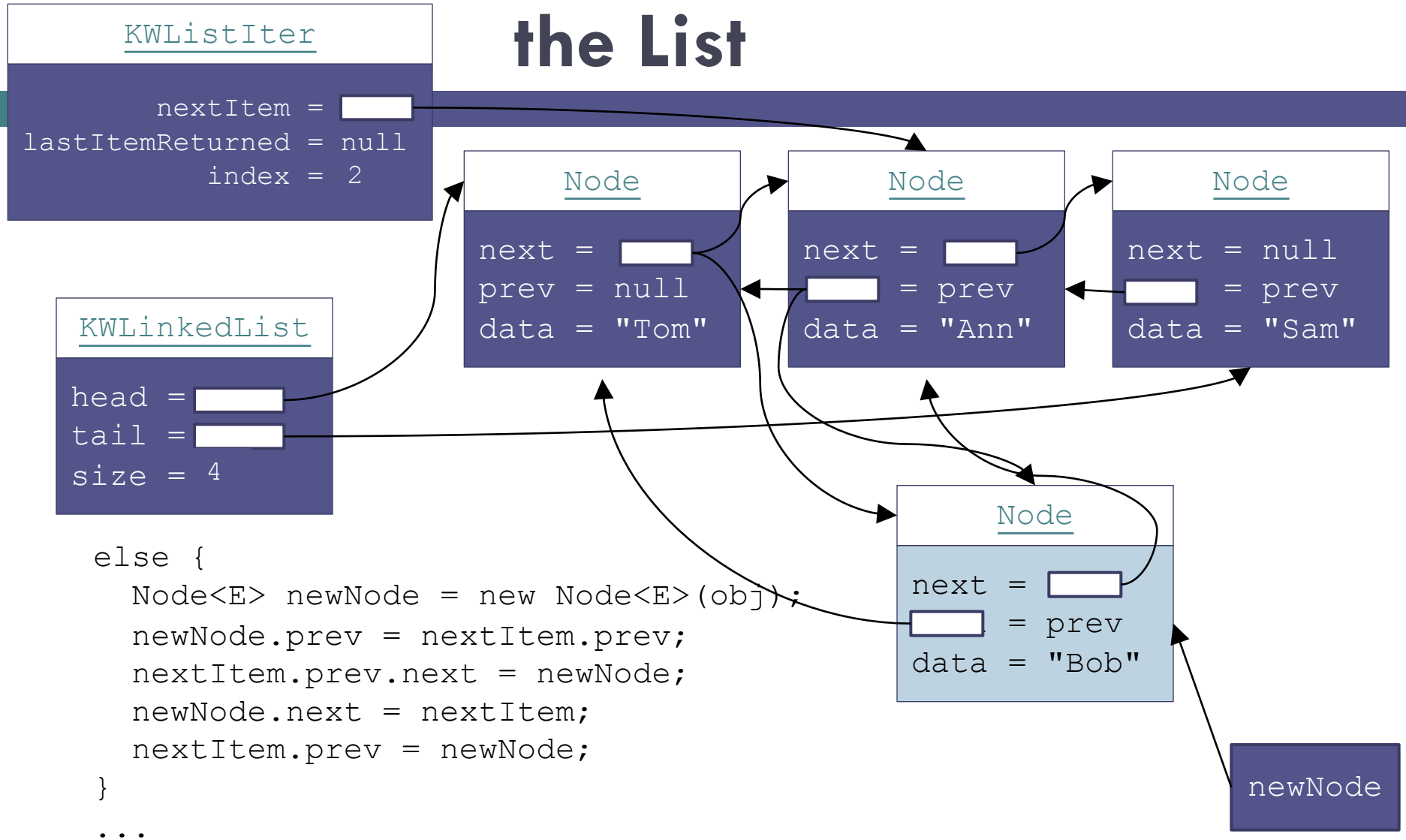

```
if (nextItem == head) {
  Node<E> newNode = new Node<E>(obj);
  newNode.next = nextItem;
  nextItem.prev = newNode;
  head = newNode;
}
...
size++;
index++;
```

Adding to the Tail of the List




```
if (nextItem == null) {
  Node<E> newNode = new Node<E>(obj);
  tail.next = newNode;
  newNode.prev = tail;
  tail = newNode
}
...
size++;
index++;
```

Adding to the Middle of the List




```
else {
  Node<E> newNode = new Node<E>(obj);
  newNode.prev = nextItem.prev;
  nextItem.prev.next = newNode;
  newNode.next = nextItem;
  nextItem.prev = newNode;
}
...
size++;
index++;
```

Inner Classes: Static and Nonstatic

- `KWLinkedList` contains two inner classes:
  - `Node<E>` is declared static: there is no need for it to access the data fields of its parent class, `KWLinkedList`
  - `KWListIter` cannot be declared static because its methods access and modify data fields of `KWLinkedList`'s parent object which created it
- An inner class which is not static contains an implicit reference to its parent object and can reference the fields of its parent object

The Collections Framework Design

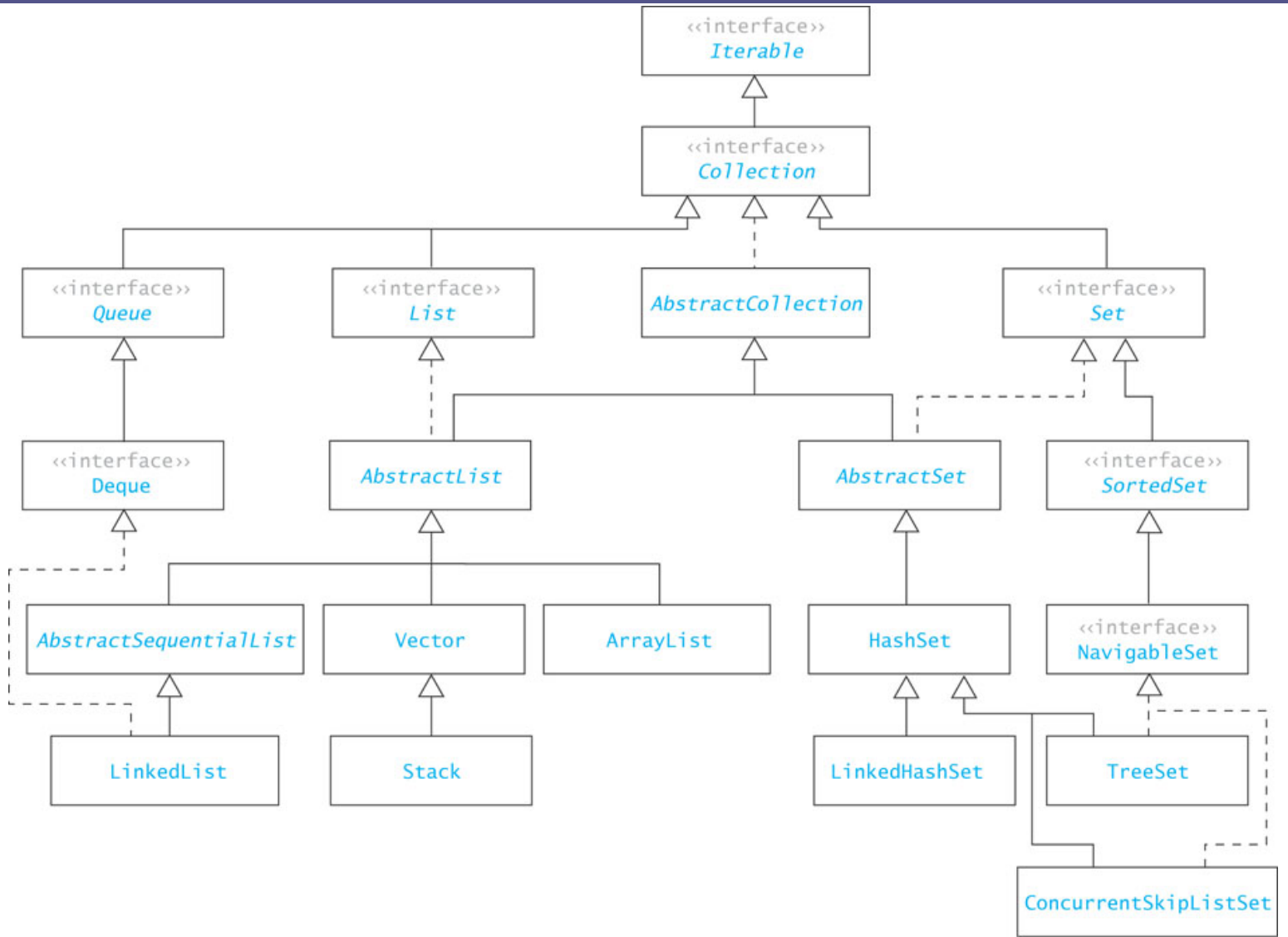
Section 2.9

The `Collection` Interface

- Specifies a subset of methods in the `List` interface, specifically excluding
  - `add(int, E)`
  - `get(int)`
  - `remove(int)`
  - `set(int, E)`

  but including
  - `add(E)`
  - `remove(Object)`
  - the `iterator` method

The Collection Framework



Common Features of Collections

- Collections
  - grow as needed
  - hold references to objects
  - have at least two constructors: one to create an empty collection and one to make a copy of another collection

Application of the `LinkedList` Class

Section 2.10

An Application: Ordered Lists

- We want to maintain a list of names in alphabetical order at all times
- **Approach**
  - Develop an `OrderedList` class (which can be used for other applications)
  - Implement a `Comparable` interface by providing a `compareTo(E)` method
  - Use a `LinkedList` class as a component of the `OrderedList`
    - if `OrderedList` extended `LinkedList`, the user could use `LinkedList`'s `add` methods to add an element out of order

Class Diagram for `OrderedList`

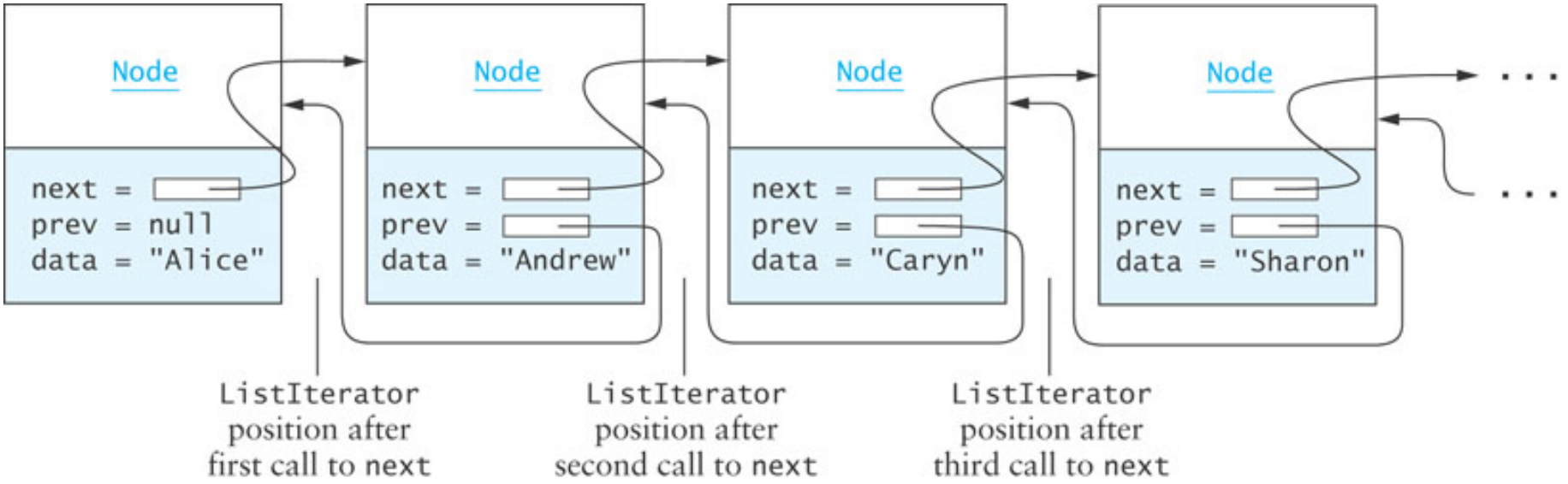
Design

Data Field	Attribute
`private LinkedList<E> theList`	A linked list to contain the data.
Method	**Behavior**
`public void add(E obj)`	Inserts `obj` into the list preserving the list's order.
`public Iterator iterator()`	Returns an `Iterator` to the list.
`public E get(int index)`	Returns the object at the specified position.
`public int size()`	Returns the size of the list.
`public E remove(E obj)`	Removes first occurrence of `obj` from the list.

Inserting into an `OrderedList`

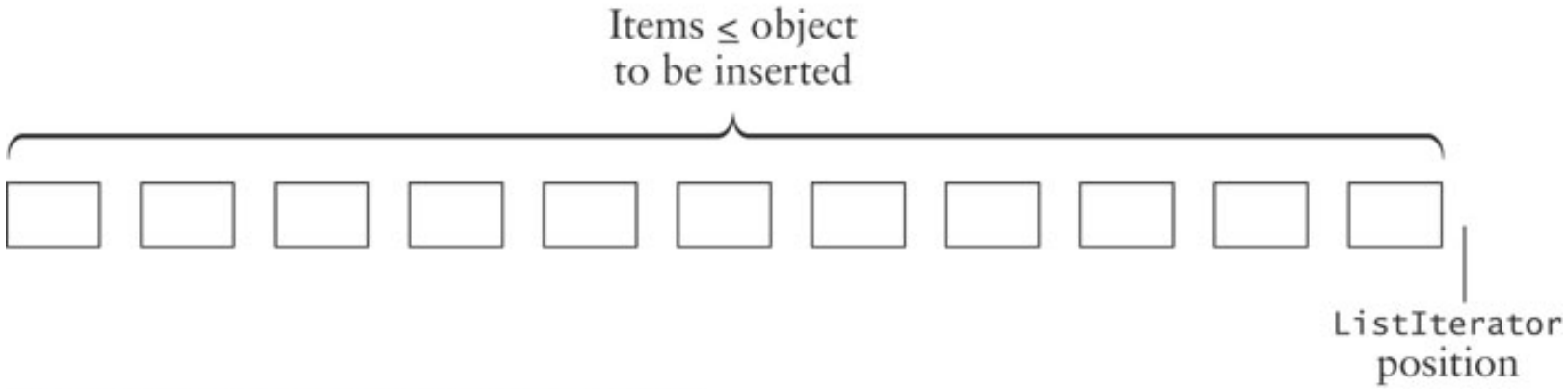
- Strategy for inserting new element `e`:
  - Find first item > `e`
  - Insert `e` before that item
- Refined with an iterator:
  - Create `ListIterator` that starts at the beginning of the list
  - While the `ListIterator` is not at the end of the list and `e` >= the next item
    - Advance the `ListIterator`
  - Insert `e` before the current `ListIterator` position

Inserting Diagrammed

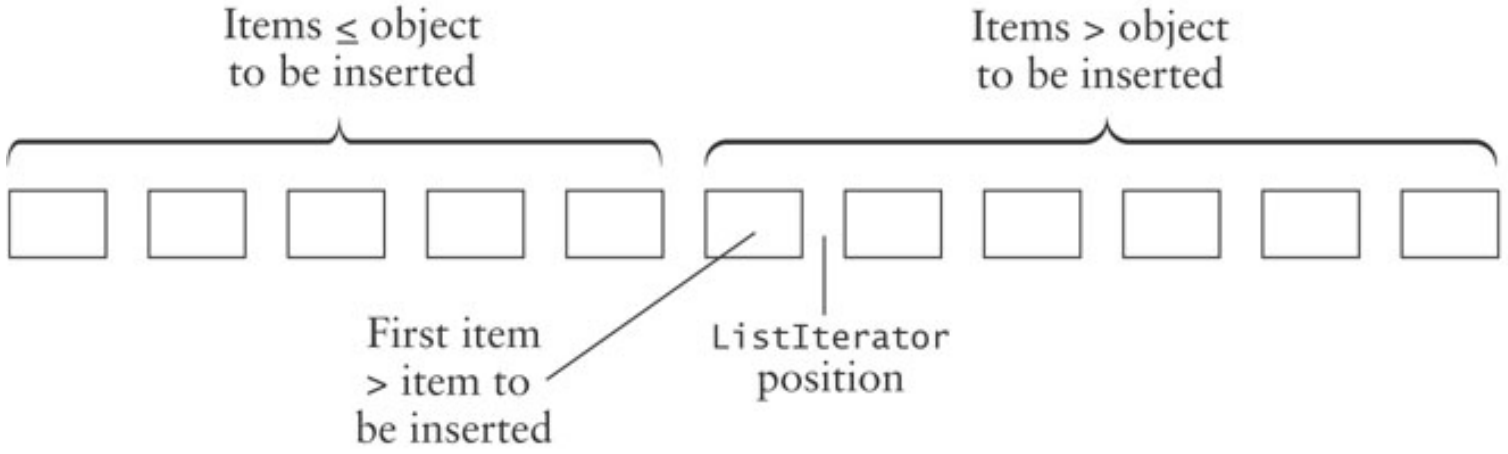


Inserting Diagrammed (cont.)

Case 1: Inserting at the end of a list



Case 2: Inserting in the middle of a list



OrderedList.add


```
public void add (E e) {
  ListIterator<E> iter = theList.listIterator();
  while (iter.hasNext()) {
    if (e.compareTo(iter.next()) < 0) {
      // found element > new one
      iter.previous();  // back up by one
      iter.add(e);      // add new one
      return;           // done
     }
  }
  iter.add(e);  // will add at end
}
```

Using Delegation to Implement the Other Methods


```
public E get (int index) {
   return theList.get(index);
}
public int size () {
   return theList.size();
}
public E remove (E e) {
   return theList.remove(e);
}
```

Testing `OrderedList`

- To test an `OrderedList`,
  - store a collection of randomly generated integers in an `OrderedList`
  - test insertion at beginning of list: insert a negative integer
  - test insertion at end of list: insert an integer larger than any integer in the list
  - create an iterator and iterate through the list, displaying an error if any element is smaller than the previous element
  - remove the first element, the last element, and a middle element, then traverse to show that order is maintained

Testing `OrderedList` (cont.)


```
Class TestOrderedList

import java.util.*;

public class TestOrderedList {
      /** Traverses ordered list and displays each
      element.
      Displays an error message if an element is out of
      order.
      @param testList An ordered list of integers
      */
```

Testing `OrderedList` (cont.)


```
public static void traverseAndShow(OrderedList<Integer>
                testList) {
        int prevItem = testList.get(0);

        // Traverse ordered list and display any value that
        // is out of order.
        for (int thisItem : testList) {
                System.out.println(thisItem);
                    if (prevItem > thisItem)
                       System.out.println("*** FAILED, value is "
                                          + thisItem);
                  prevItem = thisItem;
        }
}
```

Think of how this can be done using references

Testing `OrderedList` (cont.)


```
public static void main(String[] args) {
      OrderedList<Integer> testList = new
                                    OrderedList<Integer>();
      final int MAX_INT = 500;
      final int START_SIZE = 100;

      // Create a random number generator.
      Random random = new Random();
      for (int i = 0; i < START_SIZE; i++) {
            int anInteger = random.nextInt(MAX_INT);
            testList.add(anInteger);
      }
```

Testing `OrderedList` (cont.)


```
// Add to beginning and end of list.
testList.add(-1);
testList.add(MAX_INT + 1);
traverseAndShow(testList); // Traverse and display.

// Remove first, last, and middle elements.
Integer first = testList.get(0);
Integer last = testList.get(testList.size() - 1);
```

Testing `OrderedList` (cont.)


```
    Integer middle = testList.get(testList.size() / 2);
    testList.remove(first);
    testList.remove(last);
    testList.remove(middle);
    traverseAndShow(testList); // Traverse and display.
    }
}
```