Chapter 17

A Linked Structure

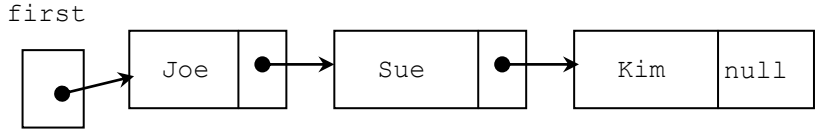
This chapter demonstrates the `OurList` interface implemented with a class that uses a linked structure rather than the array implementation of the previous chapter. The linked version introduces another data structure for implementing collection classes: the singly linked structure.

Goals

- Show a different way to elements in a collection class
- See how nodes can be linked
- Consider the differences from arrays in order to such as sequencing through elements that are no longer in contiguous memory

17.1 A Linked Structure

A collection of elements can be stored within a linked structure by storing a reference to elements in a node and a reference to another node of the same type. The next node in a linked structure also stores a reference to data and a reference to yet another node. There is at least one variable to locate the beginning, which is named `first` here



A linked structure with three nodes

Each node is an object with two instance variables:

1. A reference to the data of the current node ("Joe", "Sue", and "Kim" above)
2. A reference to the next element in the collection, indicated by the arrows

The node with the reference value of `null` indicates the end of the linked structure. Because there is precisely one link from every node, this is a singly linked structure. (Other linked structures have more than one link to other nodes.)

A search for an element begins at the node referenced by the external reference `first`. The second node can be reached through the link from the first node. Any node can be referenced in this sequential fashion. The search stops at the null terminator, which indicates the end. These nodes may be located anywhere in available

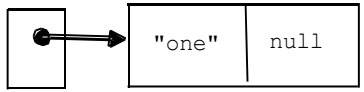
memory. The elements are not necessarily contiguous memory locations as with arrays. Interface `OurList` will now be implemented using many instances of the private inner class named `Node`.

```
/**
 * OurLinkedList is a class that uses an singly linked structure to
 * store a collection of elements as a list. This is a growable coll-
 * ection that uses a linked structure for the backing data storage.
 */
public class OurLinkedList<E> implements OurList<E> {
  // This private inner class is accessible only within OurLinkedList.
  // Instances of class Node will store a reference to the same
  // type used to construct an OurLinkedList<Type>.
  private class Node  {
    // These instance variables can be accessed within OurLinkedList<E>
    private E data;
    private Node next;

    public Node(E element) {
      data = element;
      next = null;
    }
  } // end class Node

  // TBA: OurLinkedList instance variables and methods

} // end class OurLinkedList
```

The `Node` instance variable `data` is declared as Object in order to allow any type of element to be stored in a node. The instance variable named `next` is of type `Node`. This allows one `Node` object to refer to another instance of the same `Node` class. Both of these instance variables will be accessible from the methods of the enclosing class (`OurLinkedList`) even though they have `private` access.

We will now build a linked structure storing a collection of three `String` objects. We let the `Node` reference `first` store a reference to a Node with `"one"` as the data.

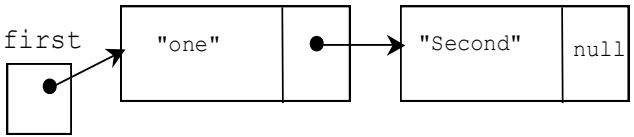
```
// Build the first node and keep a reference to this first node
Node first = new Node("one");
```




```
public Node(Object objectReference) {
  data = objectReference;
  next = null;
}
```

The following construction stores a reference to the string `"second"`. However, this time, the reference to the new `Node` object is stored into the `next` field of the `Node` referenced by `first`. This effectively adds a new node to the end of the list.

```
// Construct a second node referenced by the first node's next
first.next = new Node("Second");
```

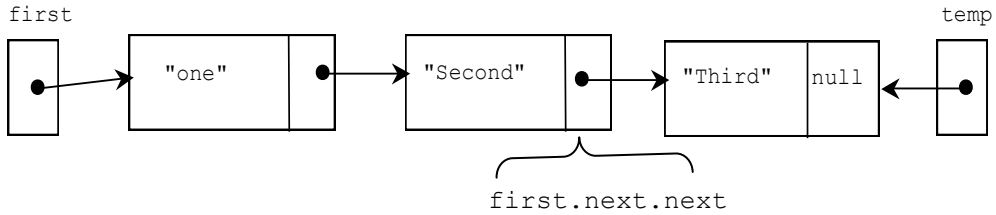


The code above directly assigned a reference to the next instance variable. This unusual direct reference to a private instance variable makes the implementation of `OurLinkedList` than having a separate class as some textbooks use. Since `Node` is intimately tied to this linked structure — and it has been made an inner class — you will see many permitted assignments to both of `Node`'s private instance variables, `data` and `next`.

This third construction adds a new `Node` to the end of the list. The `next` field is set to refer to this new node by referencing it with the dot notation `first.next.next`.

```
// Construct a third node referenced by the second node's next
Node temp = new Node("Third");
// Replace null with the reference value of temp (pictured as an arrow)
first.next.next = temp;
```

The following picture represents this hard coded (not general) list:

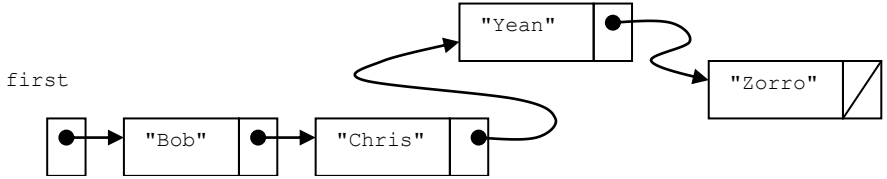


The `Node` reference variable named `first` is not an internal part of the linked structure. The purpose of `first` is to find the beginning of the list so algorithms can find an insertion point for a new element, for example.

In a singly linked structure, the instance variable `data` of each `Node` refers to an object in memory, which could be of any type. The instance variable `next` of each `Node` object references another node containing the next element in the collection. One of these `Node` objects stores `null` to mark the end of the linked structure. The `null` value should only exist in the last node.

Self-Check

Use this linked structure to answer the questions that follow.



17-1 What is the value of `first.data`?

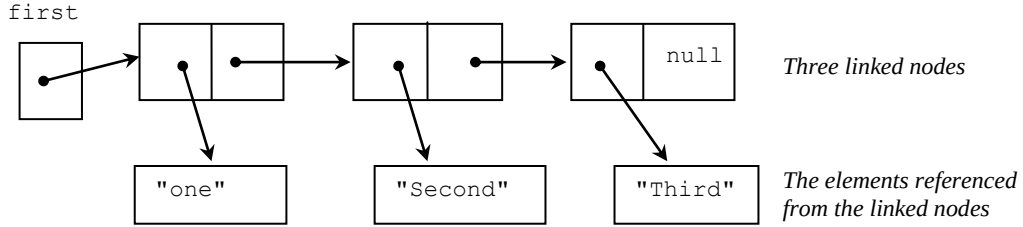
17-2 What is the value of `first.next.data`?

17-3 What is the value of `first.next.next.next.data`?

17-4 What is the value of `first.next.next.next`?

Each node stores a reference to the element

A linked structure would be pictured more accurately with the `data` field shown to reference an object somewhere else in memory.



However, it is more convenient to show linked structures with the value of the element written in the node, especially if the elements are strings. This means that even though both parts store a reference value (exactly four bytes of memory to indicate a reference to an object), these structures are often pictured with a box dedicated to the data value, as will be done in the remainder of this chapter. The reference values, pictured as arrows, are important. If one of these links becomes misdirected, the program will not be able to find elements in the list.

Traversing a Linked Structure

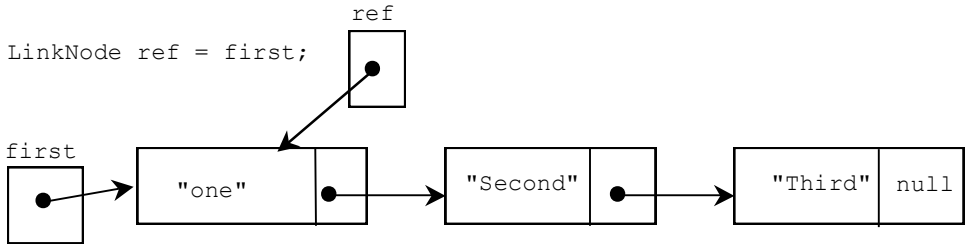
Elements in a linked structure can be accessed in a sequential manner. Analogous to a changing `int` subscript to reference all elements in an array, a changing `Node` variable can reference all elements in a singly linked structure. In the following `for` loop, the `Node` reference begins at the first node and is updated with `next` until it becomes `null`.

```
for (Node ref = first; ref != null; ref = ref.next) {
  System.out.println(ref.data.toString());
```

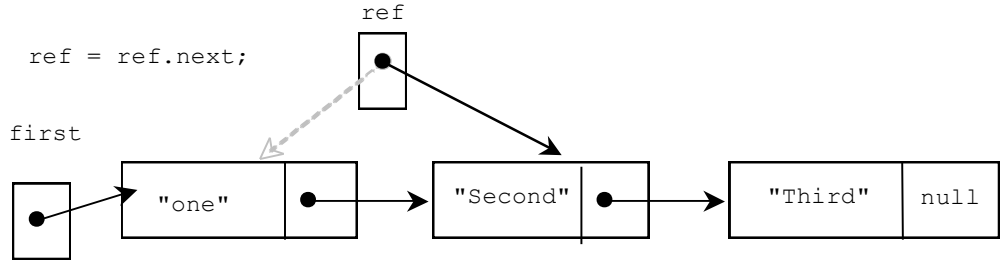
Output

```
one
Second
Third
```

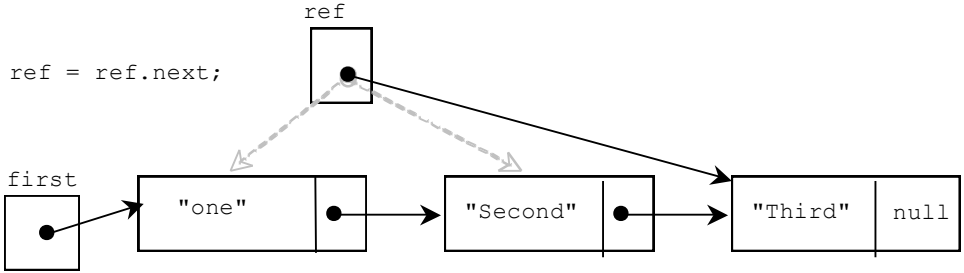
When the loop begins, `first` is not `null`, thus neither is `ref`. The `Node` object `ref` refers to the first node.



At this point, `ref.data` returns a reference to the object referenced by the data field—in this case, the string "one". To get to the next element, the `for` loop updates the external pointer `ref` to refer to the next node in the linked structure. The first assignment of `ref = ref.next` sets `ref` to reference the second node.



At the end of the next loop iteration, `ref = ref.next` sets `ref` to reference the third node.



And after one more `ref = ref.next`, the external reference named `ref` is assigned `null`.

At this point, the `for` loop test `ref != null` is `false`. The traversal over this linked structure is complete.

With an array, the `for` loop could be updating an integer subscript until the value is beyond the index of the last meaningful element (`index == n` for example). With a linked structure, the `for` loop updates an external reference (a `Node` reference named `ref` here) so it can reference all nodes until it finds the `next` field to be `null`.

This traversal represents a major difference between a linked structure and an array. With an array, subscript `[2]` directly references the third element. This random access is very quick and it takes just one step. With a linked structure, you must often use sequential access by beginning at the first element and visiting all

the nodes that precede the element you are trying to access. This can make a difference in the runtime of certain algorithms and drive the decision of which storage mechanism to use.

17.2 Implement **`OurList`** with a Linked Structure

Now that the inner `private class Node` exists, consider a class that implements `OurList`. This class will provide the same functionality as `OurArrayList` with a different data structure. The storage mechanism will be a collection of `Node` objects. The algorithms will change to accommodate this new underlying storage structure known as a singly linked structure. The collection class that implements ADT OurList along with its methods and linked structure is known as a **linked list**.

This `OurLinkedList` class uses an inner Node class with two additional constructors (their use will be shown later). It also needs the instance variable first to mark the beginning of the linked structure.

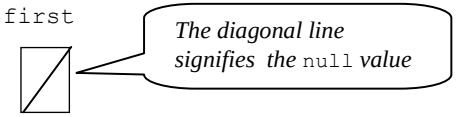
```
// A type-safe Collection class to store a list of any type element
public class OurLinkedList<E> implements OurList<E> {

  // This private inner class is only known within OurLinkedList.
  // Instances of class Node will store a reference to an
  // element and a reference to another instance of Node.
  private class Node  {

    // Store one element of the type specified during construction
    private E data;
    // Store a reference to another node with the same type of data
    private Node next;

    // Allows Node n = new Node();
    public Node() {
      data = null;
      next = null;
    }
```

```
    // Allows Node n = new Node("Element");
    public Node(E element) {
      data = element;
      next = null;
    }

    // Allows Node n = new Node("Element", first);
    public Node(E element, Node nextReference) {
      data = element;
      next = nextReference;
    }

  } /////// end inner class Node //////////

  // Instance variables for OurLinkedList
  private Node first;

  private int size;

  // Construct an empty list with size 0
  public OurLinkedList() {
    first = null;
    size = 0;
  }

  // more to come …
}
```

After construction, the picture of memory shows `first` with a `null` value written as a diagonal line.

```
  OurLinkedList<String> list = new OurLinkedList<String>();
```

An empty list:



The first method `isEmpty` returns true when first is null.

```
/**
 * Return true when no elements are in this list
 */
public boolean isEmpty() {
  return first == null;
}
```

Adding Elements to a Linked Structure

This section explores the algorithms to add to a linked structure in the following ways:

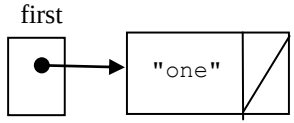
- Inserting an element at the beginning of the list
- Inserting an element at the end of the list
- Inserting an element anywhere in the list at a given position

To insert an element as the first element in a linked list that uses an external reference to the first node, the algorithm distinguishes these two possibilities:

1. the list is empty
2. the list is not empty

If the list is empty, the insertion is relatively easy. Consider the following code that inserts a new `String` object at index zero of an empty list. A reference to the new `Node` object with `"one"` will be assigned to `first`.

```
OurLinkedList<String> stringList = new OurLinkedList<String>();
stringList.addFirst("one");
```




```
/** Add an element to the beginning of this list.
  * O(1)
  * @param element The new element to be added at index 0.
  */
  public void addFirst(E element) {
    // The beginning of an addFirst operation
    if (this.isEmpty()) {
      first = new Node(element);
      // ...
```

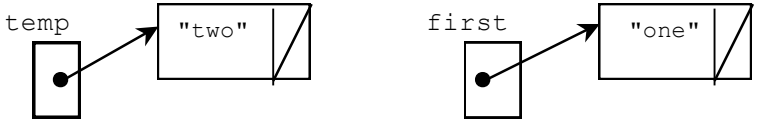
When the list is not empty, the algorithm must still make the insertion at the beginning; `first` must still refer to the new first element. You must also take care of linking the new element to the rest of the list. Otherwise, you lose the entire list! Consider adding a new first element (to a list that is not empty) with this message:

```
stringList.addFirst("two");
```

This can be accomplished by constructing a new `Node` with the zero-argument constructor that sets both `data` and `next` to `null`. Then the reference to the soon to be added element is stored in the data field (again `E` can represent any reference type).

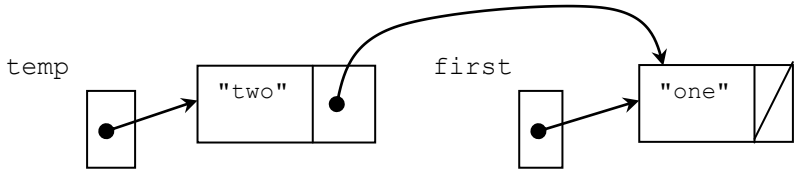
```
    else {
      // the list is NOT empty
      Node temp = new Node(); // data and next are null
      temp.data = element;    // Store reference to element
```

There are two lists now, one of which is temporary.



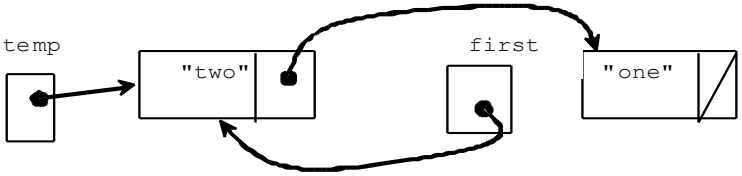
The following code links the node that is about to become the new `first` so that it refers to the element that is about to become the second element in the linked structure.

```
      temp.next = first; // 2 Nodes refer to the node with "one"
    }
```

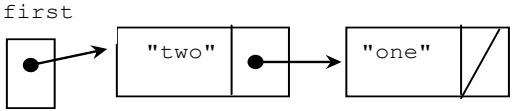


Now move `first` to refer to the `Node` object referenced by `first` and increment `size`.

```
      first = temp;
    }  // end method addFirst
    size++;
  } // end addFirst
```



After `"two"` is inserted at the front, the local variable `temp` is no longer needed in the picture. The list can also be drawn like this since the local variable `temp` will no longer exist after the method finishes:



This `size` method can now return the number of elements in the collection (providing the other `add` methods also increment size when appropriate).

```
/**
  * Return the number of elements in this list
  */
 public int size() {
   return size;
 }
```

Self-Check

17-5 Draw a picture of memory after each of the following sets of code executes:

a.
```
OurLinkedList<String> aList = new OurLinkedList<String>();
```

b.
```
OurLinkedList<String> aList = new OurLinkedList<String>();
aList.addFirst("Chris");
```

c.
```
OurLinkedList<Integer> aList = new OurLinkedList<Integer>();
aList.addFirst(1);
aList.addFirst(2);
```

`addFirst(E)` again

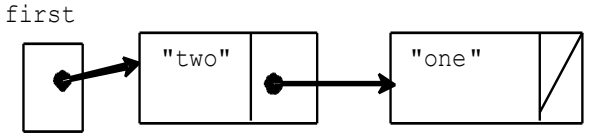
The `addFirst` method used an `if…else` statement to check if the reference to the beginning of the list needed to be changed. Then several other steps were required. Now imagine changing the `addFirst` method using the two-argument constructor of the `Node` class.

```
public Node(Object element, Node nextReference) {
   data = element;
   next = nextReference;
}
```

To add a new node at the beginning of the linked structure, you can initialize both `Node` instance variables. This new two-argument constructor takes a `Node` reference as a second argument. The current value of `first` is stored into the new `Node`'s `next` field. The new node being added at index 0 now links to the old `first`.

```
/** Add an element to the beginning of this list.
  * @param element The new element to be added at the front.
  * Runtime O(1)
  */
 public void addFirst(E element) {
   first = new Node(element, first);
   size++;
 }
```

To illustrate, consider the following message to add a third element to the existing list of two nodes (with "two" and "one"): `stringList.addFirst("tre");`



The following initialization executes in `addFirst`:

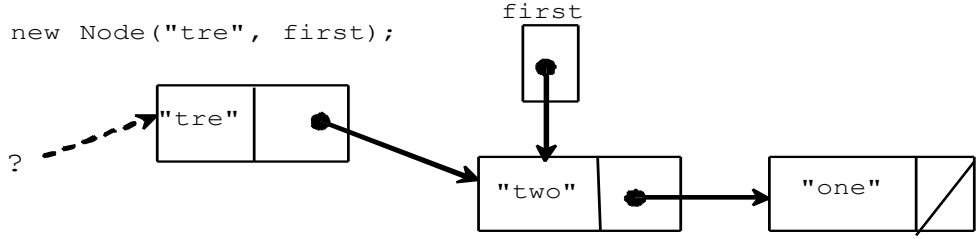
```
first = new Node(element, first);
```

This invokes the two-argument constructor of `class Node`:

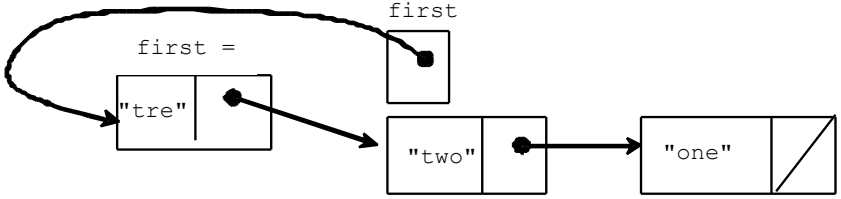
```
public Node(Object element, Node nextReference) {
  data = element;
  next = nextReference;
}
```

This constructor generates the `Node` object pictured below with a reference to `"tre"` in `data`. It also stores the value of `first` in its `next` field. This means the new node (with `"tre"`) has its `next` field refer to the old

`first` of the list.

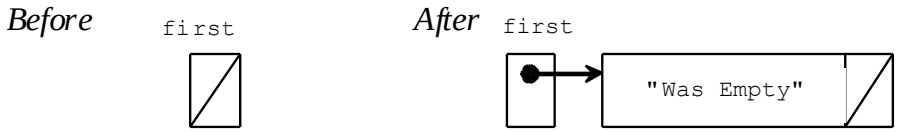


Then after the construction is finished, the reference to the new `Node` is assigned to `first`. Now `first` refers to the new `Node` object. The element `"tre"` is now at index 0.



The following code illustrates that `addFirst` will work even when the list is empty. You end up with a new `Node` whose reference instance variable next has been set to `null` and where `first` references the only element in the list.

```
OurLinkedList<String> anotherList = new OurLinkedList<String>();
anotherList.addFirst("Was Empty");
```



Since the `addFirst` method essentially boils down to two assignment statements, no matter how large the list, `addFirst` is O(1).

E get(int)

OurList specifies a get method that emulates the array square bracket notation `[]` for getting a reference to a specific index. This implementation of the `get` method will throw an `IllegalArgumentException` if the argument `index` is outside the range of `0` through `size()-1`. This avoids returning `null` or other meaningless data during a `get` when the index is out of range.

```
/**
  * Return a reference to the element at index getIndex
  * O(n)
  */
 public E get(int getIndex) {
 // Verify insertIndex is in the range of 0..size()-1
 if (getIndex < 0 || getIndex >= this.size())
   throw new IllegalArgumentException("" + getIndex);
```

Finding the correct node is not the direct access available to an array. A loop must iterate through the linked structure.

```
    Node ref = first;
    for (int i = 0; i < getIndex; i++)
      ref = ref.next;
    return ref.data;
  } // end get
```

When the temporary external reference `ref` points to the correct element, the data of that node will be returned. It is now possible to test the `addFirst` and `get` methods. First, let's make sure the method throws an exception when the index to `get` is out of range. First we'll try `get(0)` on an empty list.

```
@Test(expected = IllegalArgumentException.class)
public void testGetExceptionWhenEmpty() {
  OurLinkedList<String> list = new OurLinkedList<String>();
  list.get(0); // We want get(0) to throw an exception
}
```

Another test method ensures that the indexes just out of range do indeed throw exceptions.

```
@Test(expected = IllegalArgumentException.class)
public void testGetExceptionWhenIndexTooBig() {
  OurLinkedList<String> list = new OurLinkedList<String>();
  list.addFirst("B");
  list.addFirst("A");
  list.get(2); // should throw an exception
}


@Test(expected = IllegalArgumentException.class)
public void testGetExceptionWhenIndexTooSmall () {
  OurLinkedList<String> list = new OurLinkedList<String>();
  list.addFirst("B");
  list.addFirst("A");
  list.get(-1); // should throw an exception
}
```

This test for `addFirst` will help to verify it works correctly while documenting the desired behavior.

```
@Test
public void testAddFirstAndGet() {
  OurLinkedList<String> list = new OurLinkedList<String>();
  list.addFirst("A");
  list.addFirst("B");
  list.addFirst("C");
  // Assert that all three can be retrieved from the expected index
  assertEquals("C", list.get(0));
  assertEquals("B", list.get(1));
  assertEquals("A", list.get(2));
}
```

Self-Check

17-6 Which one of the following assertions would fail: a, b, c, or d?

```
OurLinkedList<String> list = new OurLinkedList<String>();
list.addFirst("El");
list.addFirst("Li");
list.addFirst("Jo");
list.addFirst("Cy");
assertEquals("El", list.get(3));  // a.
assertEquals("Li", list.get(2));  // b.
assertEquals("JO", list.get(1));  // c.
assertEquals("Cy", list.get(0));  // d.
```

`String toString()`

Programmers using an `OurLinkedList` object may be interested in getting a peek at the current state of the list or finding an element in the list. To do this, the list will also have to be traversed.

This algorithm in `toString` begins by storing a reference to the first node in the list and updating it until it reaches the desired location. A complete traversal begins at the node reference by first and ends at the last node (where the `next` field is `null`). The loop traverses the list until `ref` becomes `null`. This is the only `null` value stored in a next field in any proper list. The null value denotes the end of the list.

```
/**
  * Return a string with all elements in this list.
  * @returns One String that concatenation of toString versions of all
  * elements in this list separated by ", " and bracketed with "[  ]".
  */
  public String toString() {
    String result = "[";
    if (! this.isEmpty()) { // There is at least one element
      // Concatenate all elements except the last
      Node ref = first;
      while (ref.next != null) {
        // Concatenate the toString version of each element
        result = result + ref.data.toString() + ", ";
        // Bring loop closer to termination
        ref = ref.next;
      }
      // Concatenate the last element (if size > 0) but without ", "
      result += ref.data.toString();
    }
    // Always concatenate the closing square bracket
    result += "]";
    return result;
  }
```

Notice that each time through the `while` loop, the variable `ref` changes to reference the `next` element. The loop keeps going as long as `ref` does not refer to the last element (`ref.next != null`) .

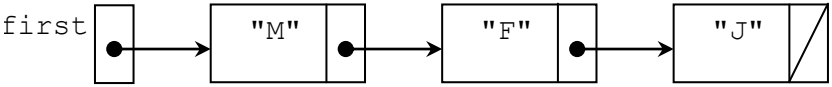
Modified versions of the `for` loop traversal will be used to insert an element into a linked list at a specific index, to find a specific element, and to remove elements.

The **`add(int, E)`** Method

Suppose a linked list has the three strings `"M"`, `"F"`, and `"J"`:

```
OurLinkedList<String> list = new OurLinkedList<String>();
list.add(0, "M");
list.add(1, "F");
list.add(2, "J");
assertEquals("[M, F, J]", list.toString());
```

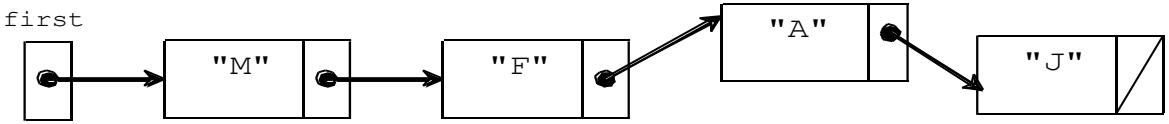
The linked structure generated by the code above would look like this:



This message inserts a fourth string into the 3rd position, at index 2, where `"J"` is now:

```
list.add(2, "A"); // This has zero based indexing--index 2 is 3rd spot
assertEquals("[M, F, A, J]", list.toString());
```

Since the three existing nodes do not necessarily occupy contiguous memory locations in this list, the elements in the existing nodes need not be shifted as did the array data structure. However, you will need a loop to count to the insertion point. Once again, the algorithm will require a careful adjustment of links in order to insert a new element. Below, we will see how to insert `"A"` at index 2.



The following algorithm inserts an element into a specific location in a linked list. After ensuring that the index is in the correct range, the algorithm checks for the special case of inserting at index 0, where the external reference `first` must be adjusted.

```
if the index is out of range
    throw an exception
else if the new element is to be inserted at index 0
  addFirst(element)
else {
  Find the place in the list to insert
  construct a new node with the new element in it
  adjust references of existing Node objects to accommodate the insertion
}
```

This algorithm is implemented as the `add` method with two arguments. It requires the index where that new element is to be inserted along with the object to be inserted. If either one of the following conditions exist, the index is out of range:

1. a negative index
2. an index greater than the size() of the list

The `add` method first checks if it is appropriate to throw an exception — when `insertIndex` is out of range.

```
/** Place element at the insertIndex specified.
  * Runtime: O(n)
  * @param element The new element to be added to this list
  * @param insertIndex The location where the new element will be added
  * @throws IllegalArgumentException if insertIndex is out of range
  */
  public void add(int insertIndex, E element) {
    // Verify insertIndex is in the range of 0..size()-1
    if (insertIndex < 0 || insertIndex > this.size())
      throw new IllegalArgumentException("" + insertIndex);
```

The method throws an `IllegalArgumentException` if the argument is less than zero or greater than the number of elements. For example, when the size of the list is 4, the only legal arguments would be 0, 1, 2, or 3 and 4 (inserts at the end of the list). For example, the following message generates an exception because the largest index allowed with "insert element at" in a list of four elements is 4.

```
list.add(5, "Y");
```

Output

```
java.lang.IllegalArgumentException: 5
```

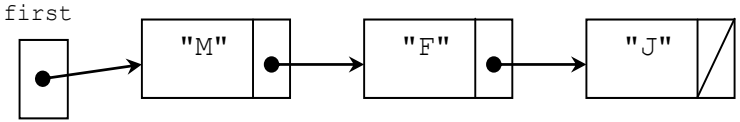
If `insertIndex` is in range, the special case to watch for is if the `insertAtIndex` equals 0. This is the one case when the external reference `first` must be adjusted.

```
  if (insertIndex == 0) {
    // Special case of inserting before the first element.
    addFirst(element);
  }
```

The instance variable `first` must be changed if the new element is to be inserted before all other elements. It is not enough to simply change the local variables. The `addFirst` method shown earlier conveniently takes the correct steps when `insertIndex==0`.

If the `insertIndex` is in range, but not 0, the method proceeds to find the correct insertion point. Let's return to a list with three elements, built with these messages:

```
OurLinkedList<String> list = new OurLinkedList<String>();
list.add(0, "M");
list.add(1, "F");
list.add(2, "J");
```



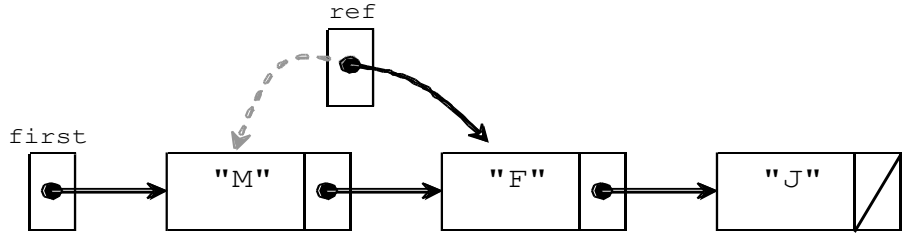
This message inserts a new element at index 2, after `"F"` and before `"J"`.

```
list.add(2, "A"); // We're using zero based indexing, so 2 is 3rd spot
```

This message causes the `Node` variable `ref` (short for reference) to start at `first` and get. This external reference gets updated in the `for` loop.

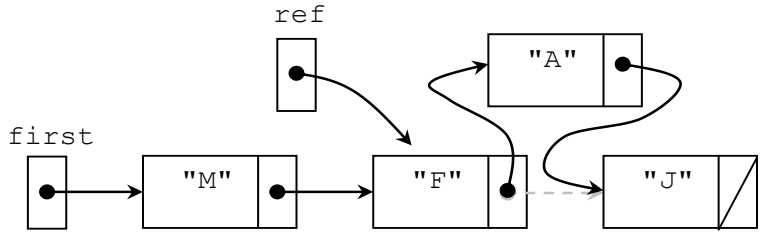
```
else {
  Node ref = first;
  for (int index = 1; index < insertIndex; index++) {
    // Stop when ref refers to the node before the insertion point
    ref = ref.next;
  }
  ref.next = new Node(element, ref.next);
}
```

The loop leaves `ref` referring to the node before the node where the new element is to be inserted. Since this is a singly linked list (and can only go forward from `first` to back) there is no way of going in the opposite direction once the insertion point is passed. So, the loop must stop when ref refers to the node *before* the insertion point.



The insertion can now be made with the `Node` constructor that takes a `Node` reference as a second argument.

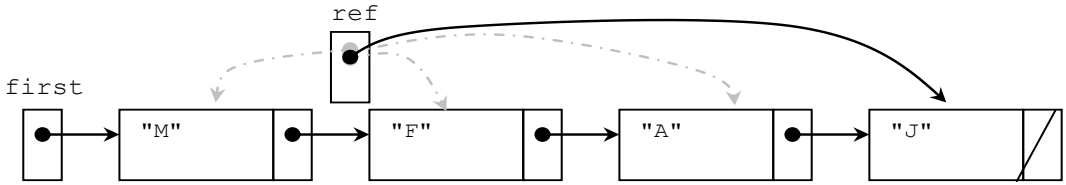
```
ref.next = new Node(element, ref.next);
```



Inserting at index 2

Consider the insertion of an element at the end of a linked list. The `for` loop advances ref until it refers to the last node in the list, which currently has the element "J". The following picture provides a trace of ref using this message

```
list.add(list.size(), "LAST");
```

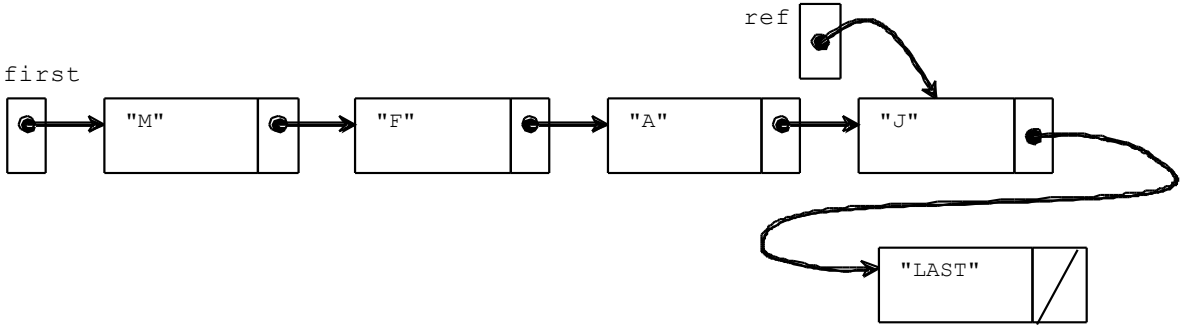


If the list has 1,000 elements, this loop requires 999 (or in general n-1) operations.

```
for (int index = 1; index < insertIndex - 1; index++) {
  ref = ref.next;
}
```

Once the insertion point is found, with `ref` pointing to the correct node, the new element can be added with one assignment and help from the `Node` class.

```
ref.next = new Node(element, ref.next);
```



The new node's `next` field becomes `null` in the `Node` constructor. This new node, with `"LAST"` in it, marks the new end of this list.

Self-Check

17-7 Which of the `add` messages (there may be more than one) would throw an exception when sent immediately after the message `list.add(0, 4);`?

```
OurLinkedList<Integer> list = new OurLinkedList<Integer> ();
list.add(0, 1);
list.add(0, 2);
list.add(0, 3);
list.add(0, 4);
```

a. `list.add(-1, 5);`
b. `list.add(3, 5);`
c. `list.add(5, 5);`
d. `list.add(4, 5);`

addLast

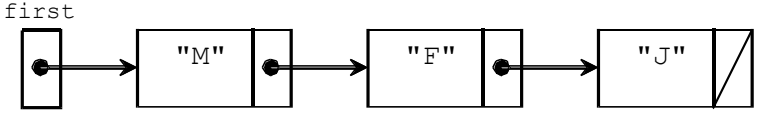
The `addLast` method is easily implemented in terms of `add`. It could have implemented the same algorithm separately, however it is considered good design to avoid repeated code and use an existing method if possible.

```
/**
  * Add an element to the end of this list.
  * Runtime: O(n)
  * @param element The element to be added as the new end of the list.
  */
  public void addLast(E element) {
    // This requires n iterations to determine the size before the
    // add method loops size times to get a reference to the last
    // element in the list. This is n + n operations, which is O(n).
    add(size(), element);
  }
```

The `addLast` algorithm can be modified to run O(1) by adding an instance variable that maintains an external reference to the last node in the linked list. Modifying this method is left as a programming exercise.

Removing from a Specific Location: **`removeElementAt(int)`**

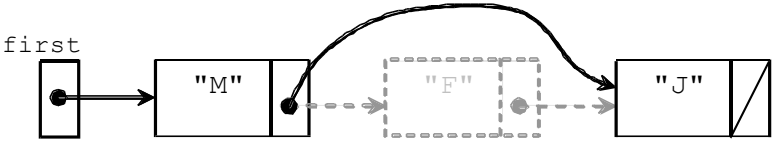
Suppose a linked list has these three elements:



Removing the element at index 1 is done with a `removeElementAt` message.

```
assertEquals("[M, F, J]", list.toString());
list.removeElementAt(1);
assertEquals("[M, J]", list.toString());
```

The linked list should look like this after the node with "F" is reclaimed by Java's garbage collector. There are no more references to this node, so it is no longer needed.



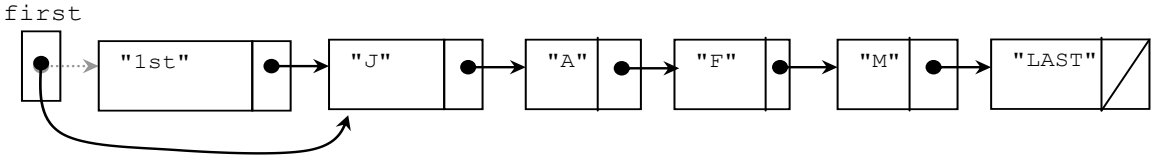
Assuming the index of the element to be removed is in the correct range of `0` through `size()-1`, the following algorithm should work with the current implementation:

```
if removal index is out of range
    throw an exception
else if the removal is the node at the first
  change first to refer to the second element  (or make the list empty if size()==1)
else  {
  Get a reference to the node before the node to be removed
  Send the link around the node to be removed
}
```

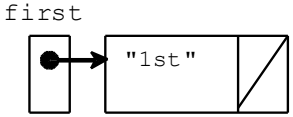
A check is first made to avoid removing elements that do not exist, including removing index 0 from an empty list. Next up is checking for the special case of removing the first node at index 0 ("one" in the structure below).

Simply send `first.next` "around" the first element so it references the second element. The following assignment updates the external reference first to refer to the next element.

```
first = first.next;
```

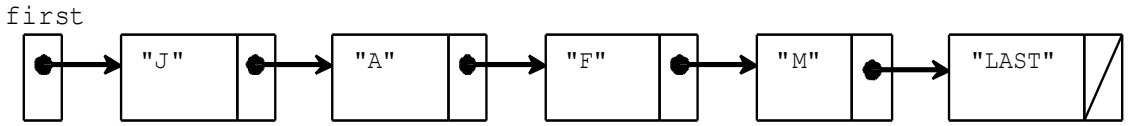


This same assignment will also work when there is only one element in the list.



With the message `list.removeElementAt(0)` on a list of size 1, the old value of first is replaced with `null`, making this an empty list.

Now consider `list.removeElementAt(2)` (`"F"`) from the following list:

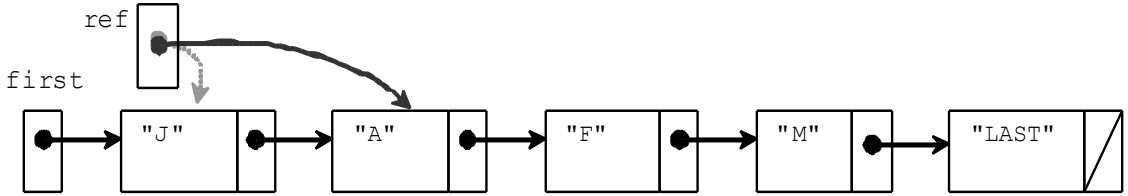


The following assignment has the `Node` variable `ref` refer to the same node as `first`:

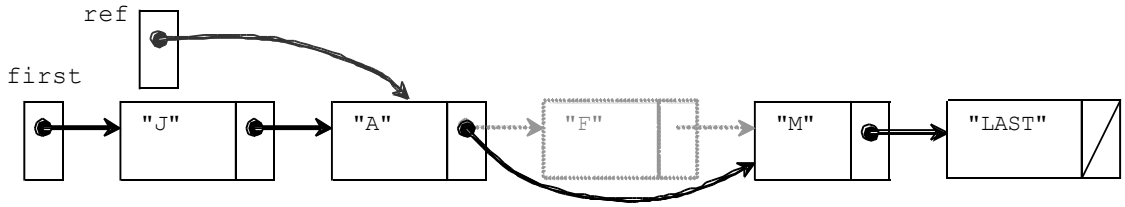
```
Node ref = first;
```

`ref` then advances to refer to the node just *before* the node to be removed.

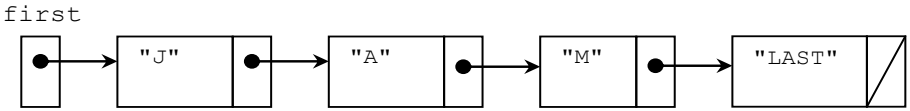
```
for (int index = 1; index < removalIndex; index++) // 1 iteration only
  ref = ref.next;
```



Then the node at index 1 ("A") will have its `next` field adjusted to move around the node to be removed ("F"). The modified list will look like this:



Since there is no longer a reference to the node with "F", the memory for that node will be reclaimed by Java's garbage collector. When the method is finished, the local variable `ref` also disappears and the list will look like this:



The `removeElementAt` method is left as a programming exercise.

Deleting an element from a Linked List: `remove`

When deleting an element from a linked list, the code in this particular class must recognize these two cases:

1. Deleting the first element from the list (a special case again)
2. Deleting an interior node from the list (including the last node)

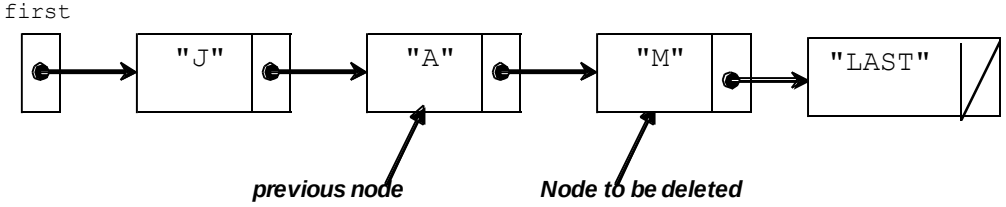
When deleting the first node from a linked list, care must be taken to ensure that the rest of the list is not destroyed. The adjustment to `first` is again necessary so that all the other methods work (and the object is not

in a corrupt state). This can be accomplished by shifting the reference value of `first` to the second element of the list (or to `null` if there is only one element). One assignment will do this:

```
first = first.next;
```

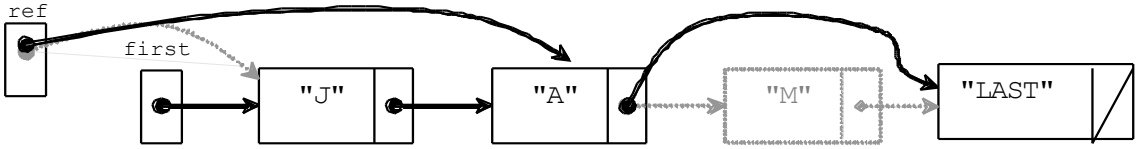
Now consider removing a specific element that may or may not be stored in an interior node. As with `removeElementAt`, the code will look to place a reference to the node just *before* the node to be removed. So to remove `"M"`, the link to the node before M is needed. This is the node with `"A"`.

```
list.remove("M");
```



At this point, the `next` field in the node with "A" can be "sent around" the node to be removed ("M"). Assuming the `Node` variable named ref is storing the reference to the node before the node to be deleted, the following assignment effectively removes a node and its element "M" from the structure:

```
ref.next = ref.next.next;
```



Deleting a specific internal node, in this case "M"

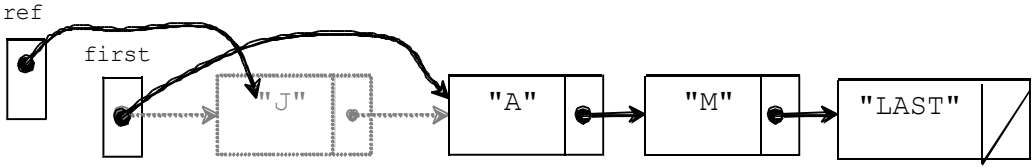
This results in the removal of an element from the interior of the linked structure. The memory used to store the node with "M" will be reclaimed by Java's garbage collector.

The trick to solving this problem is comparing the data that is one node ahead. Then you must make a reference to the node before the found element (assuming it exists in the list). The following code does just that. It removes the first occurrence of the `objectToRemove` found in the linked list. It uses the class's `equals` method to make sure that the element located in the node equals the state of the object that the message intended to remove. First, a check is made for an empty list.

```
/** Remove element if found using the equals method for type E.
  * @param The object to remove from this list if found
  */
  public boolean remove(E element) {
    boolean result = true;

    // Don't attempt to remove an element from an empty list
    if (this.isEmpty())
      result = false;
```

The following code takes care of the special case of removing the first element when the list is not empty:6

```
else {
  // If not empty, begin to search for an element that equals obj
  // Special case: Check for removing first element
  if (first.data.equals(element))
    first = first.next;
```



Checking for these special cases has an added advantage. The algorithm can now assume that there is at least one element in the list. It can safely proceed to look one node ahead for the element to remove. A `while` loop traverses the linked structure while comparing `objectToRemove` to the `data` in the node one element ahead. This traversal will terminate when either of these two conditions occur:

1. The end of the list is reached.
2. An item in the next node `equals` the element to be removed.

The algorithm assumes that the element to be removed is in index 1 through `size()-1` (or it's not there at all). This allows the `Node` variable named `ref` to "peek ahead" one node. Instead of comparing `objectToRemove` to `ref.data`, `objectToRemove` is compared to `ref.next.data`.

```
else {
  // Search through the rest of the list
  Node ref = first;
  // Look ahead one node
  while ((ref.next != null) && !(element.equals(ref.next.data)))
    ref = ref.next;
```

This `while` loop handles both loop termination conditions. The loop terminates when `ref`'s `next` field is `null` (the first expression in the loop test). The loop will also terminate when the next element (`ref.next.data`) in the list `equals(objectToRemove)`, the element to be removed. Writing the test for `null` before the equals message avoids `null` pointer exceptions. Java's guaranteed short circuit boolean evaluation will not let the expression after && execute when the first subexpression (`ref.next != null`) is false.

Self-Check

17-8 What can happen if the subexpressions in the loop test above are reversed?

```
while (!(objectToRemove.equals(ref.next.data))
         && (ref.next != null))
```

At the end of this loop, `ref` would be pointing to one of two places:

1. the node just before the node to be removed, or
2. the last element in the list.

In the latter case, no element "equaled" `objectToRemove`. Because there are two ways to terminate the loop, a test is made to see if the removal element was indeed in the list. The link adjustment to remove a node executes only if the loop terminated before the end of the list was reached. The following code modifies the list only if `objectToRemove` was found.

```
// Remove node if found (ref.next != null).  However, if
// ref.next is null, the search stopped at end of list.
if (ref.next == null)
  return false;  // Got to the end without finding element
else {
  ref.next = ref.next.next;
```

```
      return true;
    }
  } // end remove(E element)
```

Self-Check

17-9 In the space provided, write the expected value that would make the assertions pass:

```
OurLinkedList<String> list = new OurLinkedList<String>();
list.addLast("A");
list.insertElementAt(0, "B");
list.addFirst("C");
assertEquals(_______________, list.toString());  // a.
list.remove("B");
assertEquals(_______________, list.toString());  // b.
list.remove("A");
assertEquals(_______________, list.toString());  // c.
list.remove("Not Here");
assertEquals(_______________, list.toString());  // d.
list.remove("C");
assertEquals(_______________, list.toString());  // e.
```

17-10 What must you take into consideration when executing the following code?

```
if (current.data.equals("CS 127B"))
  current.next.data = "CS 335";
```

17.3 When to use Linked Structures

The one advantage of a linked implementation over an array implementation may be constrained to the growth and shrink factor when adding elements. With an array representation, growing an array during `add` and shrinking an array during `remove` requires an additional temporary array of contiguous memory be allocated. Once all elements are copied, the memory for the temporary array can be garbage collected. However, for that moment, the system has to find a large contiguous block of memory. In a worst case scenario, this could potentially cause an `OutOfMemoryException`.

When adding to a linked list, the system allocates the needed object and reference plus an additional 4 bytes overhead for the `next` reference value. This may work on some systems better than an array implementation, but it is difficult to predict which is better and when.

The linked list implementation also may be more time efficient during inserts and removes. With an array, removing the first element required n assignments. Removing from a linked list requires only one assignment. Removing an internal node may also run a bit faster for a linked implementation, since the worst case rarely occurs. With an array, the worst case always occurs—n operations are needed no matter which element is being removed. With a linked list, it may be more like n/2 operations.

Adding another external reference to refer to the last element in a linked list would make the addLast method run O(1), which is as efficient as an array data structure. A linked list can also be made to have links to the node before it to allow two-way traversals and faster removes — a doubly linked list. This structure could be useful in some circumstances.

A good place to use linked structures will be shown in the implementation of the stack and queue data structures in later chapters. In both collections, access is limited to one or both ends of the collection. Both grow and shrink frequently, so the memory and time overhead of shifting elements are avoided (however, an array can be used as efficiently with a few tricks).

Computer memory is another thing to consider when deciding which implementation of a list to use. If an array needs to be "grown" during an `add` operation, for a brief time there is a need for twice as many reference values. Memory is needed to store the references in the original array. An extra temporary array is also needed. For example, if the array to be grown has an original capacity of 50,000 elements, there will be a need for an additional 200,000 bytes of memory until the references in the original array are copied to the temporary array. Using a linked list does not require as much memory to grow. The linked list needs as many references as the array does for each element, however at grow time the linked list can be more efficient in terms of memory (and time). The linked list does not need extra reference values when it grows.

Consider a list of 10,000 elements. A linked structure implementation needs an extra reference value (`next`) for every element. That is overhead of 40,000 bytes of memory with the linked version. An array-based implementation that stores 10,000 elements with a capacity of 10,000 uses the same amount of memory. Imagine the array has 20 unused array locations — there would be only 80 wasted bytes. However, as already mentioned, the array requires double the amount of overhead when growing itself. Linked lists provide the background for another data structure called the binary tree structure in a later chapter.

When not to use Linked Structures

If you want quick access to your data, a linked list will not be that helpful when the size of the collection is big. This is because accessing elements in a linked list has to be done sequentially. To maintain a fixed list that has to be queried a lot, the algorithm needs to traverse the list each time in order to get to the information. So if a lot of `set` and `gets` are done, the array version tends to be faster. The access is O(1) rather than O(n). Also, if you have information in an array that is sorted, you can use the more efficient binary search algorithm to locate an element.

A rather specific time to avoid linked structures (or any dynamic memory allocations) is when building software for control systems on aircraft. The United States Federal Aviation Association (FAA) does not allow it because it's not safe to have a airplane system run out of memory in flight. The code must work with fixed

arrays. All airline control code is carefully reviewed to ensure that allocating memory at runtime is not present. With Java, this would mean there could never be any existence of `new`.

One reason to use the linked version of a list over an array-based list is when the collection is very large and there are frequent add and removal messages that trigger the grow array and shrink array loops. However, this could be adjusted by increasing the `GROW_SHRINK_INCREMENT` from 20 to some higher number. Here is a comparison of the runtimes for the two collection classes implemented over this and the previous chapter.

	`OurArrayList`	`OurLinkedList`
get and set	O(1)	O(n)
remove removeElementAt	O(n)	O(n)
find[1]	O(n)	O(n)
add(int index, Object el)	O(n)	O(n)
size[2]	O(1)	O(n)
addFirst	O(n)	O(1)
addLast[3]	O(1)	O(n)

One advantage of arrays is the `get` and `set` operations of `OurArrayList` are an order of magnitude better than the linked version. So why study singly linked structures?

---

[1] find could be improved to O(log n) if the data structure is changed to an ordered and sorted list.

[2] size could be improved to O(1) if the `SimpleLinkedList` maintained a separate instance variable for the number of element in the list (add 1during inserts subtract 1 during successful removes)

[3] addLast with SimpleLinkedList could be improved by maintaining an external reference to the last element in the list.

1. The linked structure is a more appropriate implementation mechanism for the stacks and queues of the next chapter.
2. Singly linked lists will help you to understand how to implement other powerful and efficient linked structures (trees and skip lists, for example).

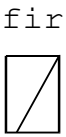
When a collection is very large, you shouldn't use either of the collection classes shown in this chapter, or even Java's `ArrayList` or `LinkedList` classes in the `java.util` package. There are other data structures such as hash tables, heaps, and trees for large collections. These will be discussed later in this book. In the meantime, the implementations of a list interface provided insights into the inner workings of collections classes and two storage structures. You have looked at collections from both sides now.

Self-Check

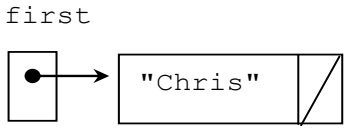
17-11 Suppose you needed to organize a collection of student information for your school's administration. There are approximately 8,000 students in the university with no significant growth expected in the coming years. You expect several hundred lookups on the collection everyday. You have only two data structures to store the data, and array and a linked structure. Which would you use? Explain.

Answers to Self-Checks

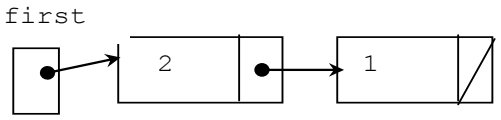
17-1 first.data.equals("Bob")
17-2 first.next.data ("Chris");
17-3 first.next.next.next.data.equals("Zorro");
17-4 first.next.next.next refers to a Node with a reference to "Zorro" and null in its next field.
17-5 drawing of memory
a.



b.



c.



17-6 c would fail ("JO" should be "Jo")

17-7 which would throw an exception

-a- IndexOutOfBoundsException

-c- the largest valid index is currently 4

-d- Okay since the largest index can be the size, which is 4 in this case

17-8 if switched, ref would move one Node too far and cause a NullPointerException

17-9 assertions - listed in correct order

```
OurLinkedList<String> list = new OurLinkedList<String>();
list.addLast("A");
list.insertElementAt(0, "B");
list.addFirst("C");

assertEquals(__"[C, B, A]"__, list.toString()); // a.

list.remove("B");
assertEquals(___"[C, A]"__, list.toString());  // b.

list.remove("A");
assertEquals(____"[C]"_____, list.toString()); // c.

list.remove("Not Here");
assertEquals(____"[C]"_____, list.toString()); // d.

list.remove("C");
assertEquals(_____"[]"_____, list.toString()); // e.
```

17-10 Whether or not a node actually exists at current.next. It could be null.

17-11 An array so the more efficient binary search could be used rather than the sequential search necessary with a linked structure.