Chapter 9

Lists

Concepts:
- ▷ The list abstraction
- ▷ Singly linked lists
- ▷ Doubly linked lists
- ▷ Circular lists
- ▷ Vectors as lists

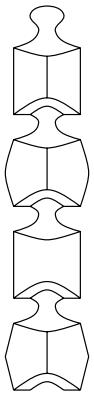
He's makin' a list
and checkin' it twice!
—Haven Gillespie

IMAGINE YOU ARE TO WRITE A ROBUST PROGRAM to handle varying amounts of data. An inventory program is a classic example. The same inventory program might be used to keep track of either tens of items, or millions. To support such applications, `Vectors` and arrays are not ideal. As they reach their capacity they must be expanded. Either this happens manually, as with arrays, or automatically, as with `Vectors`. In either case the penalty for growing the structure is the same over the growth of the structure. With a `Vector`, for example, when the structure must double in size, the cost of adding an element is proportional to the size of the `Vector`.

In this chapter, we develop the concept of a linked list. A *linked list* is a *dynamic structure* that grows and shrinks exactly when necessary and whose elements may be added in constant time. There is some cost for this dynamic behavior, however. As with `Vectors` and arrays, each of the elements of a linked-list has an associated index, but the elements of many linked list implementations cannot be efficiently accessed out of order or *accessed randomly*. Despite this one inefficiency, linked lists provide an important building block for the design of many effective data structures.

An analogy for linked lists is a child's string of snap-together beads. As we grow the string of beads, we attach and detach new beads on either the front (*head*) or rear (*tail*). Since there are two modifying operations that we can perform (*add* or *remove*) and two ends (at the location of the *first* or *last* element) there are four operations that change the length of the structure at the end.

We may also wish to perform operations on the internal portion of the list. For example, we may want to test for inclusion (*Is* there a red bead?) or extract an element (*Remove* a red bead!). These operations require a traversal of the linked list from one of the two ends.



If you have never seen these, visit your niece.

List

Now, let's see what the Java description of a list looks like:

```
public interface List<E> extends Structure<E>
{
    public int size();
    // post: returns number of elements in list

    public boolean isEmpty();
    // post: returns true iff list has no elements

    public void clear();
    // post: empties list

    public void addFirst(E value);
    // post: value is added to beginning of list

    public void addLast(E value);
    // post: value is added to end of list

    public E getFirst();
    // pre: list is not empty
    // post: returns first value in list

    public E getLast();
    // pre: list is not empty
    // post: returns last value in list

    public E removeFirst();
    // pre: list is not empty
    // post: removes first value from list

    public E removeLast();
    // pre: list is not empty
    // post: removes last value from list

    public E remove(E value);
    // post: removes and returns element equal to value
    //       otherwise returns null

    public void add(E value);
    // post: value is added to tail of list

    public E remove();
    // pre: list has at least one element
    // post: removes last value found in list

    public E get();
    // pre: list has at least one element
    // post: returns last value found in list
```

```
    public boolean contains(E value);
    // pre: value is not null
    // post: returns true iff list contains an object equal to value

    public int indexOf(E value);
    // pre: value is not null
    // post: returns (0-origin) index of value,
    //   or -1 if value is not found

    public int lastIndexOf(E value);
    // pre: value is not null
    // post: returns (0-origin) index of value,
    //   or -1 if value is not found

    public E get(int i);
    // pre: 0 <= i < size()
    // post: returns object found at that location

    public E set(int i, E o);
    // pre: 0 <= i < size()
    // post: sets ith entry of list to value o;
    //    returns old value

    public void add(int i, E o);
    // pre: 0 <= i <= size()
    // post: adds ith entry of list to value o

    public E remove(int i);
    // pre: 0 <= i < size()
    // post: removes and returns object found at that location

    public Iterator<E> iterator();
    // post: returns an iterator allowing
    //   ordered traversal of elements in list
}
```

Again, because this structure is described as an `interface` (as opposed to a `class`) Java understands this to be a *contract* describing the methods that are required of lists. We might think of an interface as being a "structural precondition" describing the outward appearance of any "listlike" class. If we write our code in terms of this interface, we may only invoke methods specified within the contract.

Note that the `List` interface is an extension of the `Structure` interface that we have seen earlier, in Section 1.8. Thus, every `List` is also a `Structure`—a structure that supports operations like `add` and `remove` as well as other size-related methods. We will see, over the course of this text, several abstract types that may serve as `Structures`.

The interface, along with pre- and postconditions, makes many of the *implementation-independent* decisions about the semantics of associated structures.

When we develop specific implementations, we determine the *implementation-specific* features of the structure, including its performance. When we compare specific implementations, we compare their performance in terms of space and time. Often, performance can be used to help us select among different implementations for a specific use.

9.1 Example: A Unique Program

As an example of how we might use lists, we write a program that writes out the input with duplicate lines removed. The approach is to store each of the unique lines in a structure (`lines`) as they are printed out. When new lines are read in, they are compared against the existing list of unique, printed lines. If the current line (`current`) is not in the list, it is added. If the current line is in the list, it is ignored.

Unique

```
public static void main(String[] args)
{
    // input is read from System.in
    Scanner s = new Scanner(System.in);
    String current;                  // current line
    // list of unique lines
    List<String> lines = new SinglyLinkedList<String>();
    // read a list of possibly duplicated lines
    while (s.hasNextLine()) {
        current = s.nextLine();
        // check to see if we need to add it
        if (!lines.contains(current)) {
            System.out.println(current);
            lines.add(current);
        }
    }
}
```

In this example we actually construct a particular type of list, a `SinglyLinkedList`. The details of that implementation, discussed in the next section, are not important to us because `lines` is declared to be a generic interface, a `List`. Accessing data through the `lines` variable, we are only allowed to invoke methods found in the `List` interface. On the other hand, if we are able to cast our algorithms in terms of `Lists`, any implementation of a `List` will support our program.

When given input

```
madam
I'm
Adam!
...
Adam!
```

```
I'm
Ada!
...
mad
am I...
madam
```

the program generates the following output:

```
madam
I'm
Adam!
...
Ada!
mad
am I...
```

Because there is no practical limit (other than the amount of memory available) on the length of a list, there is no practical limit on the size of the input that can be fed to the program. The list interface does not provide any hint of how the list is actually implemented, so it is difficult to estimate the performance of the program. It is likely, however, that the `contains` method—which is likely to have to consider every existing element of the list—and the `add` method—which might have to pass over every element to find its correct position—will govern the complexity of the management of this `List`. As we consider implementations of `Lists`, we should keep the performance of programs like `Unique` in mind.

9.2 Example: Free Lists

In situations where a pool of resources is to be managed, it is often convenient to allocate a large number and keep track of those that have not been allocated. This technique is often used to allocate chunks of physical memory that might eventually be allocated to individual applications or printers from a pool that might be used to service a particular type of printing request.

The following application maintains rental contracts for a small parking lot. We maintain each parking space using a simple class, `Space`:

ParkingLot

```
class Space
{   // structure describing parking space
    public final static int COMPACT = 0; // small space
    public final static int MINIVAN = 1; // medium space
    public final static int TRUCK = 2;   // large space
    protected int number;       // address in parking lot
    protected int size;         // size of space
    public Space(int n, int s)
    // post: construct parking space #n, size s
    {
```

```
        number = n;
        size = s;
    }
    public boolean equals(Object other)
    // pre: other is not null
    // post: true iff spaces are equivalent size
    {
        Space that = (Space)other;
        return this.size == that.size;
    }
}
```

The lot consists of 10 spaces of various sizes: one large, six medium, and three small. Renters may rent a space if one of appropriate size can be found on the free list. The `equals` method of the `Space` class determines an appropriate match. The `rented` list maintains `Associations` between names and space descriptions. The following code initializes the free list so that it contains all the parking spaces, while the `rented` list is initially empty:

```
List<Space> free = new SinglyLinkedList<Space>();  // available
List<Association<String,Space>> rented =
    new SinglyLinkedList<Association<String,Space>>(); // rented spaces
for (int number = 0; number < 10; number++)
{
    if (number < 3) // three small spaces
        free.add(new Space(number,Space.COMPACT));
    else if (number < 9) // six medium spaces
        free.add(new Space(number,Space.MINIVAN));
    else // one large space
        free.add(new Space(number,Space.TRUCK));
}
```

The main loop of our program reads in commands from the keyboard—either `rent` or `return`:

```
Scanner s = new Scanner(System.in);
while (s.hasNext())
{
    String command = s.next(); // rent/return
      ...
}
System.out.println(free.size()+" slots remain available.");
```

Within the loop, when the `rent` command is entered, it is followed by the size of the space needed and the name of the renter. This information is used to construct a contract:

```
Space location;
if (command.equals("rent"))
{   // attempt to rent a parking space
```

```
        String size = s.next();
        Space request;
        if (size.equals("small"))
            request = new Space(0,Space.COMPACT);
        else if (size.equals("medium"))
            request = new Space(0,Space.MINIVAN);
        else request = new Space(0,Space.TRUCK);
        // check free list for appropriate-sized space
        if (free.contains(request))
        {   // a space is available
            location = free.remove(request);
            String renter = s.next(); // to whom?
            // link renter with space description
            rented.add(new Association<String,Space>(renter,location));
            System.out.println("Space "+location.number+" rented.");
        } else {
            System.out.println("No space available. Sorry.");
        }
    }
```

Notice that when the `contains` method is called on a `List`, a dummy element is constructed to specify the type of object sought. When the dummy item is used in the `remove` command, the actual item removed is returned. This allows us to maintain a single copy of the object that describes a single parking space.

When the spaces are returned, they are returned by name. The contract is looked up and the associated space is returned to the free list:

```
    Space location;
    if (command.equals("return")){
        String renter = s.next(); // from whom?
        // template for finding "rental contract"
        Association<String,Space> query = new Association<String,Space>(renter);
        if (rented.contains(query))
        {   // contract found
            Association<String,Space> contract =
                rented.remove(query);
            location = contract.getValue(); // where?
            free.add(location); // put in free list
            System.out.println("Space "+location.number+" is now free.");
        } else {
            System.out.println("No space rented to "+renter);
        }
    }
```

Here is a run of the program:

```
       rent small Alice
    Space 0 rented.
       rent large Bob
    Space 9 rented.
```

```
   rent small Carol
Space 1 rented.
   return Alice
Space 0 is now free.
   return David
No space rented to David
   rent small David
Space 2 rented.
   rent small Eva
Space 0 rented.
   quit
6 slots remain available.
```

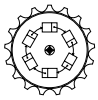
Notice that when Alice's space is returned, it is not immediately reused because the free list contains other small, free spaces. The use of `addLast` instead of `addFirst` (or the equivalent method, `add`) would change the reallocation policy of the parking lot.

We now consider an abstract base class implementation of the `List` interface.

9.3 Partial Implementation: Abstract Lists

Although we don't have in mind any particular implementation, there are some pieces of code that may be written, given the little experience we already have with the use of `Lists`.

For example, we realize that it is useful to have a number of synonym methods for common operations that we perform on `Lists`. We have seen, for example, that the `add` method is another way of indicating we want to add a new value to one end of the `List`. Similarly, the parameterless `remove` method performs a `removeLast`. In turn `removeLast` is simply a shorthand for removing the value found at location `size()-1`.



AbstractList

```
public abstract class AbstractList<E>
    extends AbstractStructure<E> implements List<E>
{
    public AbstractList()
    // post: does nothing
    {
    }

    public boolean isEmpty()
    // post: returns true iff list has no elements
    {
        return size() == 0;
    }

    public void addFirst(E value)
    // post: value is added to beginning of list
```

```
{
    add(0,value);
}

public void addLast(E value)
// post: value is added to end of list
{
    add(size(),value);
}

public E getFirst()
// pre: list is not empty
// post: returns first value in list
{
    return get(0);
}

public E getLast()
// pre: list is not empty
// post: returns last value in list
{
    return get(size()-1);
}

public E removeFirst()
// pre: list is not empty
// post: removes first value from list
{
    return remove(0);
}

public E removeLast()
// pre: list is not empty
// post: removes last value from list
{
    return remove(size()-1);
}

public void add(E value)
// post: value is added to tail of list
{
    addLast(value);
}

public E remove()
// pre: list has at least one element
// post: removes last value found in list
{
    return removeLast();
}
```

```
    public E get()
    // pre: list has at least one element
    // post: returns last value found in list
    {
        return getLast();
    }

    public boolean contains(E value)
    // pre: value is not null
    // post: returns true iff list contains an object equal to value
    {
        return -1 != indexOf(value);
    }
}
```

Position-independent operations, like `contains`, can be written in an implementation-independent manner. To see if a value is contained in a `List` we could simply determine its index with the `indexOf` method. If the value returned is -1, it was not in the list, otherwise the list contains the value. This approach to the implementation does not reduce the cost of performing the `contains` operation, but it *does* reduce the cost of *implementing* the `contains` operation: once the `indexOf` method is written, the `contains` method will be complete. When we expect that there will be multiple implementations of a class, supporting the implementations in the abstract base class can be cost effective. If improvements can be made on the generic code, each implementation has the option of providing an alternative version of the method.

Notice that we provide a parameterless constructor for `AbstractList` objects. Since the class is declared `abstract`, the constructor does not seem necessary. If, however, we write an implementation that extends the `AbstractList` class, the constructors for the implementation implicitly call the parameterless constructor for the `AbstractList` class. That constructor would be responsible for initializing any data associated with the `AbstractList` portion of the implementation. In the examples of the last chapter, we saw the `AbstractGenerator` initialized the `current` variable. Even if there is no class-specific data—as is true with the `AbstractList` class—it is good to get in the habit of writing these simple constructors.

We now consider a number of implementations of the `List` type. Each of these implementations is an extension of the `AbstractList` class. Some inherit the methods provided, while others override the selected method definitions to provide more efficient implementation.

9.4 Implementation: Singly Linked Lists

Dynamic memory is allocated using the `new` operator. Java programmers are accustomed to using the `new` operator whenever classes or arrays are to be allocated. The value returned from the `new` operator is a *reference* to the new object.

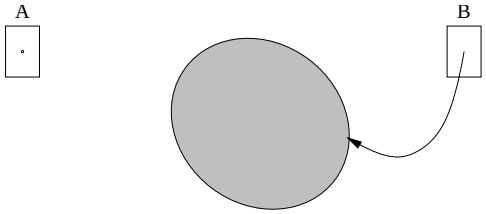
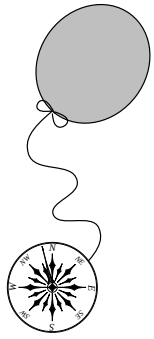


Figure 9.1 Pictures of a `null` reference (left) and a non-`null` reference to an instance of a class (right).

Thus, whenever we declare an instance of a class, we are actually declaring a reference to one of those objects. Assignment of references provides multiple variables with access to a single, shared instance of an object.

An instance of a class is like a helium-filled balloon. The balloon is the object being allocated. The string on the balloon is a convenient handle that we can use to hold onto with a hand. Anything that holds onto the string is a *reference*. Assignment of references is similar to asking another hand to "hold the balloon I'm holding." To not reference anything (to let go of the balloon) we can assign the reference the value `null`. If nothing references the balloon, then it floats away and we can no longer get access to the instance. When memory is not referenced in any way, it is recycled automatically by a *garbage collector*.

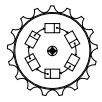


Principle 10 *When manipulating references, draw pictures.*

In this text, we will draw references as arrows pointing to their respective objects (Figure 9.1). When a reference is not referencing anything, we draw it as a dot. Since references can only be in one of two states—pointing to nothing or pointing to an object—these are the only pictures we will ever draw.

One approach to keeping track of arbitrarily large collections of objects is to use a *singly linked list* to dynamically allocate each chunk of memory "on the fly." As the chunks of memory are allocated, they are linked together to form the entire structure. This is accomplished by packaging with each user object a reference to the next object in the chain. Thus, a list of 10 items contains 10 elements, each of which contains a value as well as another element reference. Each element references the next, and the final element does not reference anything: it is assigned `null` (see Figure 9.2). Here, an implementation of a `Node` contains an additional reference, `nextElement`:

First garbage, now flies!



`Node`

```
public class Node<E>
{
    protected E data; // value stored in this element
    protected Node<E> nextElement; // ref to next
```

```
    public Node(E v, Node<E> next)
    // pre: v is a value, next is a reference to remainder of list
    // post: an element is constructed as the new head of list
    {
        data = v;
        nextElement = next;
    }

    public Node(E v)
    // post: constructs a new tail of a list with value v
    {
        this(v,null);
    }

    public Node<E> next()
    // post: returns reference to next value in list
    {
        return nextElement;
    }

    public void setNext(Node<E> next)
    // post: sets reference to new next value
    {
        nextElement = next;
    }

    public E value()
    // post: returns value associated with this element
    {
        return data;
    }

    public void setValue(E value)
    // post: sets value associated with this element
    {
        data = value;
    }
}
```

When a list element is constructed, the value provided is stored away in the object. Here, `nextElement` is a reference to the next element in the list. We access the `nextElement` and `data` fields through `public` methods to avoid accessing protected fields. Notice that, for the first time, we see a self-referential data structure: the `Node` object has a reference to a `Node`. This is a feature common to structures whose size can increase dynamically. This class is declared `public` so that anyone can construct `Node`s.

We now construct a new class that *implements* the `List` interface by extending the `AbstractList` base class. For that relation to be complete, it is necessary to provide a complete implementation of each of the methods promised by the

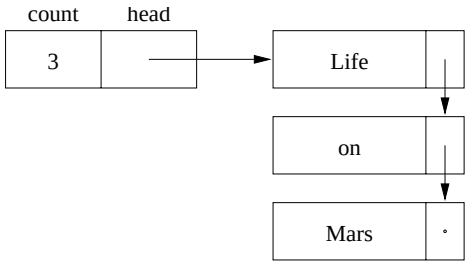


Figure 9.2 A nonempty singly linked list.

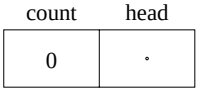
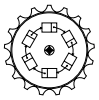


Figure 9.3 An empty singly linked list.

interface. Failure to implement *any* of the methods leaves the implementation incomplete, leaving the class *abstract*.

Our approach will be to maintain, in `head`, a reference to the first element of the list in a protected field (Figure 9.2). This initial element references the second element, and so on. The final element has a `null`-valued `next` reference. If there are no elements, `head` contains a `null` reference (Figure 9.3). We also maintain an integer that keeps track of the number of elements in the list. First, as with all classes, we need to specify protected data and a constructor:



`SinglyLinked-List`

```
protected int count;                    // list size
protected Node<E> head; // ref. to first element

public SinglyLinkedList()
// post: generates an empty list
{
    head = null;
    count = 0;
}
```

This code sets the `head` reference to `null` and the `count` field to 0. Notice that, by the end of the constructor, the list is in a consistent state.

Principle 11 *Every public method of an object should leave the object in a consistent state.*

What constitutes a "consistent state" depends on the particular structure, but in most cases the concept is reasonably clear. In the `SinglyLinkedList`, the constructor constructs a list that is empty.

The size-oriented methods are simply written in terms of the `count` identifier. The `size` method returns the number of elements in the list.

```
public int size()
// post: returns number of elements in list
{
    return count;
}
```

Recall that the `isEmpty` method described in the `AbstractList` class simply returns whether or not the `size` method would return 0. There's a great advantage to calling the `size` method to implement `isEmpty`: if we ever change the implementation, we need only change the implementation of `size`.

Both of these methods could avoid referencing the `count` field, by traversing each of the `next` references. In this alternative code we use the analogy of a *finger* referencing each of the elements in the list. Every time the finger references a new element, we increment a counter. The result is the number of elements. This time-consuming process is equivalent to constructing the information stored explicitly in the `count` field.

```
public int size()
// post: returns number of elements in list
{
    // number of elements we've seen in list
    int elementCount = 0;
    // reference to potential first element
    Node<E> finger = head;

    while (finger != null) {
        // finger references a new element, count it
        elementCount++;
        // reference possible next element
        finger = finger.next();
    }
    return elementCount;
}
```

Note that `isEmpty` does not need to change.[1] It is early verification that the interface for `size` helps to hide the implementation.

The decision between the two implementations has little impact on the user of the class, as long as both implementations meet the postconditions. Since the user is insulated from the details of the implementation, the decision can be made *even after applications have been written*. If, for example, an environment is memory-poor, it might be wise to avoid the use of the `count` field and

[1] In either case, the method `isEmpty` could be written more efficiently, checking a `null head` reference.

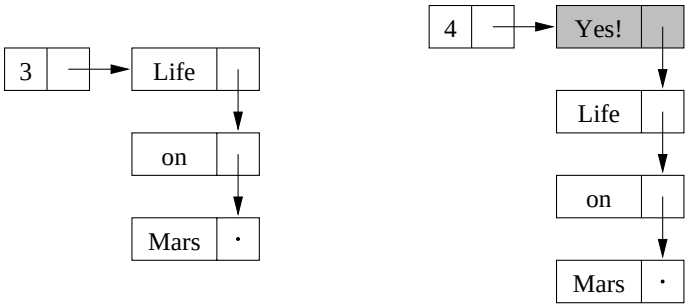


Figure 9.4 A singly linked list before and after the call to `addFirst`. Shaded value is added to the list. The `removeFirst` method reverses this process and returns value.

instead traverse the list to determine the number of elements by counting them. If, however, a machine is slow but memory-rich, then the first implementation would be preferred. Both implementations could be made available, with the user selecting the appropriate design, based on broad guidelines (e.g., memory versus speed). If this trade-off does not appear dramatic, you might consider Problem 9.10. We also discuss space–time trade-offs in more detail in Chapter 10.

Let us now consider the implementation of the methods that manipulate items at the head of the list (see Figure 9.4). First, to add an element at the head of the list, we simply need to create a new `Node` that has the appropriate value and references the very first element of the list (currently, `head`). The head of the new list is simply a reference to the new element. Finally, we modify the `count` variable to reflect an increase in the number of elements.

```
public void addFirst(E value)
// post: value is added to beginning of list
{
    // note order that things happen:
    // head is parameter, then assigned
    head = new Node<E>(value, head);
    count++;
}
```

Removing a value should simply perform the reverse process. We copy the reference[2] to a temporary variable where it can be held for return, and then we simply move the head reference down the list. Once completed, the value is returned.

[2] Remember: The assignment operator *does not* copy the value, just the reference. If you want a reference to a *new* element, you should use the `new` operator and explicitly create a new object to be referenced.

```
public E removeFirst()
// pre: list is not empty
// post: removes and returns value from beginning of list
{
    Node<E> temp = head;
    head = head.next(); // move head down list
    count--;
    return temp.value();
}
```

Notice that `removeFirst` returns a value. Why not? Since `addFirst` "absorbs" a value, `removeFirst` should do the reverse and "emit" one. Typically, the caller will not dispose of the value, but re-insert it into another data structure. Of course, if the value is not desired, the user can avoid assigning it a variable, and it will be garbage-collected at some later time. Since we think of these two operations as being inverses of each other, it is only natural to have them balance the consumption of objects in this way.

Principle 12 *Symmetry is good.*

One interesting exception to Principle 12 only occurs in languages like Java, where a garbage collector manages the recycling of dynamic memory. Clearly, `addFirst` must construct a new element to hold the value for the list. On the other hand, `removeFirst` does not explicitly *get rid* of the element. This is because after `removeFirst` is finished, there are no references to the element that was just removed. Since there are no references to the object, the garbage collector can be assured that the object can be recycled. All of this makes the programmer a little more lax about thinking about when memory has been logically freed. In languages without garbage collection, a "dispose" operation must be called for any object allocated by a `new` command. Forgetting to dispose of your garbage properly can be a rude shock, causing your program to run out of precious memory. We call this a *memory leak*. Java avoids all of this by collecting your garbage for you.

There's one more method that we provide for the sake of completeness: `getFirst`. It is a *nondestructive* method that returns a reference to the first value in the list; the list is not modified by this method; we just get access to the data:

```
public E getFirst()
// pre: list is not empty
// post: returns first value in list
{
    return head.value();
}
```

Next, we must write the methods that manipulate the tail of the list (see Figure 9.5). While the interface makes these methods appear similar to those that manipulate the head of the list, our implementation has a natural bias against tail-oriented methods. Access through a single reference to the head

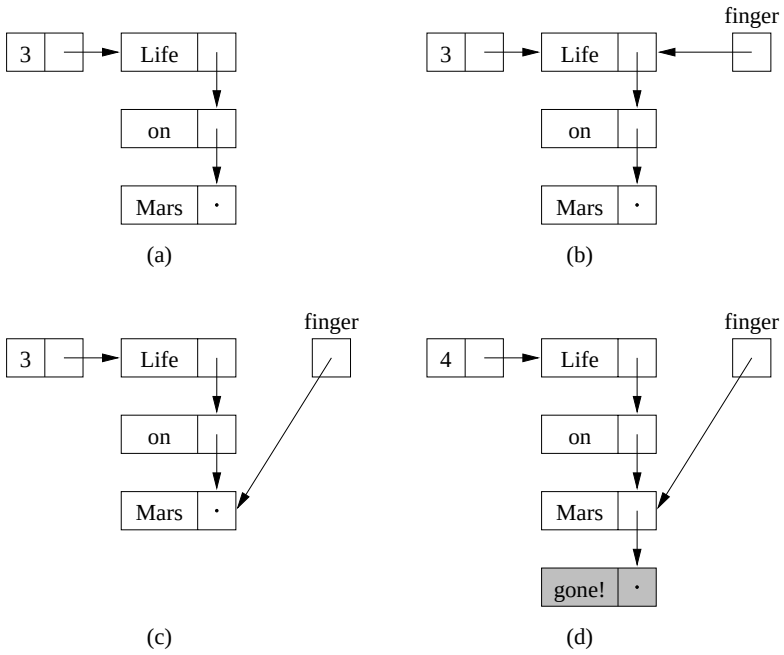


Figure 9.5 The process of adding a new value (shaded) to the tail of a list. The `finger` reference keeps track of progress while searching for the element whose reference must be modified.

---

of the list makes it difficult to get to the end of a long singly linked list. More "energy" will have to be put into manipulating items at the tail of the list. Let's see how these methods are implemented:

```
public void addLast(E value)
// post: adds value to end of list
{
    // location for new value
    Node<E> temp = new Node<E>(value,null);
    if (head != null)
    {
        // pointer to possible tail
        Node<E> finger = head;
        while (finger.next() != null)
        {
            finger = finger.next();
        }
        finger.setNext(temp);
    } else head = temp;
```

```
    count++;
}

public E removeLast()
// pre: list is not empty
// post: removes last value from list
{
    Node<E> finger = head;
    Node<E> previous = null;
    Assert.pre(head != null,"List is not empty.");
    while (finger.next() != null) // find end of list
    {
        previous = finger;
        finger = finger.next();
    }
    // finger is null, or points to end of list
    if (previous == null)
    {
        // has exactly one element
        head = null;
    }
    else
    {
        // pointer to last element is reset
        previous.setNext(null);
    }
    count--;
    return finger.value();
}
```

Each of these (complex) methods uses the finger-based list traversal technique. We reference each element of the list, starting at the top and moving downward, until we finally reach the tail. At that point we have constructed the desired reference to the end of the list, and we continue as we would have in the head-manipulating methods. We have to be aware of one slight problem that concerns the very simplest case—when the list is empty. If there are no elements, then `finger` never becomes non-`null`, and we have to write special code to manipulate the `head` reference.

To support the `add` and `remove` methods of the `Structure` (and thus `List`) interface, we had them call the `addLast` and `removeLast` methods, respectively. Given their expense, there might be a good argument to have them manipulate values at the head of the list, but that leads to an inconsistency with other potential implementations. The correct choice in design is not always obvious.

Several methods potentially work in the context of the middle of lists—including `contains` and `remove`. Here, the code becomes particularly tricky because we cannot depend on lists having any values, and, for `remove`, we must carefully handle the boundary cases—when the elements are the first or last elements of the list. Errors in code usually occur at these difficult points, so it is important to make sure they are tested.

Principle 13 *Test the boundaries of your structures and methods.*

Here is the code for these methods:

```
public boolean contains(E value)
// pre: value is not null
// post: returns true iff value is found in list
{
    Node<E> finger = head;
    while (finger != null &&
           !finger.value().equals(value))
    {
        finger = finger.next();
    }
    return finger != null;
}

public E remove(E value)
// pre: value is not null
// post: removes first element with matching value, if any
{
    Node<E> finger = head;
    Node<E> previous = null;
    while (finger != null &&
           !finger.value().equals(value))
    {
        previous = finger;
        finger = finger.next();
    }
    // finger points to target value
    if (finger != null) {
        // we found element to remove
        if (previous == null) // it is first
        {
            head = finger.next();
        } else {              // it's not first
            previous.setNext(finger.next());
        }
        count--;
        return finger.value();
    }
    // didn't find it, return null
    return null;
}
```

In the `contains` method we call the value's `equals` method to test to see if the values are logically equal. Comparing the values with the == operator checks to see if the references are the same (i.e., that they are, in fact, the same object). We are interested in finding a logically equal object, so we invoke the object's `equals` method.

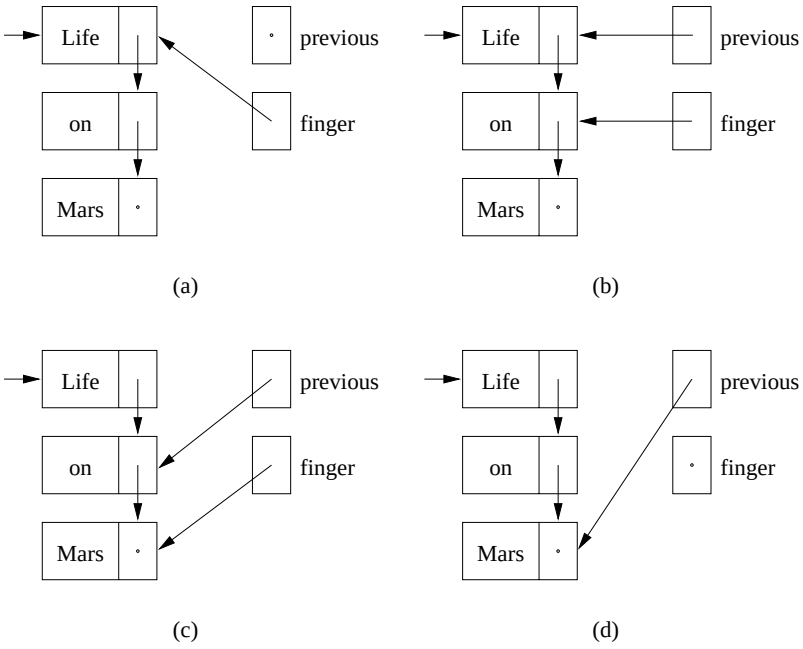


Figure 9.6 The relation between `finger` and `previous`. The target element is (a) the head of the list, (b) in the middle, (c) at the tail, or (d) not present.

Some fancy reference manipulation is needed in any routine that removes an element from the list. When we find the target value, the `finger` variable has moved too far down to help with removing the element. By the time `finger` references the element holding the target value, we lose the reference to the previous element—precisely the element that needs to have its `next` reference reset when the value is removed. To avoid this difficulty, we keep another reference, local to the particular method, that is either `null` or references the element just before `finger`. When (and if) we find a value to be removed, the element to be fixed is referenced by `previous` (Figure 9.6). Of course, if `previous` is `null`, we must be removing the first element, and we update the `head` reference. All of this can be very difficult to write correctly, which is another good reason to write it carefully once and reuse the code whenever possible (see Principle 2, *Free the future: Reuse code*).

One final method with subtle behavior is the `clear` method. This removes all the elements from the list. In Java, this is accomplished by clearing the reference to the head and adjusting the list size:

```
public void clear()
// post: removes all elements from list
{
    head = null;
    count = 0;
}
```

All that happens is that `head` stops referencing the list. Instead, it is explicitly made to reference nothing. What happens to the elements of the list? When the garbage collector comes along, it notices that the first element of the former list is not referenced by anything—after all it was only referenced by `head` before. So, the garbage collector collects that first element as garbage. It is pretty easy to see that if anything is referenced *only by* garbage, it *is* garbage. Thus, the second element (as well as the value referenced by the first element) will be marked as garbage, and so forth. This cascading identification of garbage elements is responsible for recycling all the elements of the list and, potentially, the `Objects` they reference. (If the list-referenced objects are referenced outside of the list, they *may* not be garbage after all!)

You are what references you.

We have left to this point the implementation of general methods for supporting indexed versions of `add` and `remove`. These routines insert and remove values found at particular offsets from the beginning of this list. Careful inspection of the `AbstractList` class shows that we have chosen to implement `addFirst` and similar procedures in terms of the generic `add` and `remove` routines. We have, however, already seen quite efficient implementations of these routines. Instead, we choose to make use of the end-based routines to handle special cases of the generic problem.

Here, we approach the adding of a value to the middle of a list. An index is passed with a value and indicates the desired index of the value in the augmented list. A finger keeps track of our progress in finding the correct location.

```
public void add(int i, E o)
// pre: 0 <= i <= size()
// post: adds ith entry of list to value o
{
    Assert.pre((0 <= i)  && (i <= size()),
               "Index in range.");
    if (i == size()) {
        addLast(o);
    } else if (i == 0) {
        addFirst(o);
    } else {
        Node<E> previous = null;
        Node<E> finger = head;
        // search for ith position, or end of list
        while (i > 0)
        {
            previous = finger;
            finger = finger.next();
            i--;
        }
        // create new value to insert in correct position
        Node<E> current =
            new Node<E>(o,finger);
        count++;
        // make previous value point to new value
        previous.setNext(current);
    }
}
```

Some thought demonstrates that the general code can be considerably simplified if the boundary cases (adding near the ends) can be handled directly. By handling the head and tail cases we can be sure that the new value will be inserted in a location that has a non-`null` previous value, as well as a non-`null` next value. The loop is simpler, then, and the routine runs considerably faster.

A similar approach is used in the indexed `remove` routine:

```
public E remove(int i)
// pre: 0 <= i < size()
// post: removes and returns object found at that location
{
    Assert.pre((0 <= i) && (i < size()),
               "Index in range.");
    if (i == 0) return removeFirst();
    else if (i == size()-1) return removeLast();
    Node<E> previous = null;
    Node<E> finger = head;
    // search for value indexed, keep track of previous
    while (i > 0)
    {
        previous = finger;
```

```
            finger = finger.next();
            i--;
        }
        // in list, somewhere in middle
        previous.setNext(finger.next());
        count--;
        // finger's value is old value, return it
        return finger.value();
    }
```

Exercise 9.1 *Implement the indexed* `set` *and* `get` *routines. You may assume the existence of* `setFirst`*,* `setLast`*,* `getFirst`*, and* `getLast`*.*

We now consider another implementation of the list interface that makes use of two references per element. *Swoon!*

9.5 Implementation: Doubly Linked Lists

In Section 9.4, we saw indications that some operations can take more "energy" to perform than others, and expending energy takes time. Operations such as modifying the tail of a singly linked list can take significantly longer than those that modify the head. If we, as users of lists, expect to modify the tail of the list frequently, we might be willing to make our code more complex, or use more space to store our data structure if we could be assured of significant reductions in time spent manipulating the list.

We now consider an implementation of a *doubly linked list*. In a doubly linked list, each element points not only to the next element in the list, but also to the previous element (see Figure 9.7). The first and last elements, of course, have `null` `previousElement` and `nextElement` references, respectively.

In addition to maintaining a second reference within each element, we will also consider the addition of a reference to the `tail` of the list (see Figure 9.8). This one reference provides us direct access to the end of the list and has the potential to improve the `addLast` and `removeLast` methods.

A cursory glance at the resulting data structure identifies that it is more *symmetric* with respect to the head and tail of the list. Writing the tail-related methods can be accomplished by a simple rewriting of the head-related methods. Symmetry is a powerful concept in the design of complex structures; if something is asymmetric, you should step back and ask yourself why.

Principle 14 *Question asymmetry.*

We begin by constructing a `DoublyLinkedNode` structure that parallels the `Node`. The major difference is the addition of the `previous` reference that refers to the element that occurs immediately before this element in the doubly linked list. One side effect of doubling the number of references is that we duplicate some of the information.

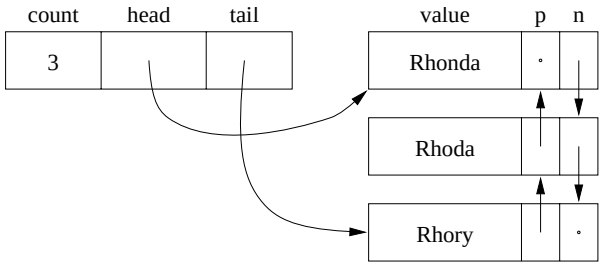


Figure 9.7 A nonempty doubly linked list.

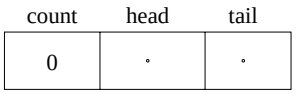


Figure 9.8 An empty doubly linked list.

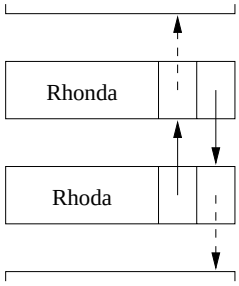
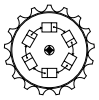


Figure 9.9 Rhonda's next reference duplicates Rhoda's previous reference.

If we look at two adjacent elements `Rhonda` and `Rhoda` in a doubly linked list, their mutual adjacency is recorded in two references (Figure 9.9): `Rhonda`'s `nextElement` reference refers to `Rhoda`, while `Rhoda`'s `previousElement` reference refers to `Rhonda`.Whenever one of the references is modified, the other must be modified also. When we construct a new `DoublyLinkedNode`, we set both the `nextElement` and `previousElement` references. If either is non-`null`, a reference in the newly adjacent structure must be updated. *If we fail to do this, the data structure is left in an inconsistent state.*

Say that *twice!*

Here's the code:



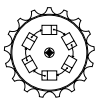
`DoublyLinked-Node`

```
protected E data;
protected DoublyLinkedNode<E> nextElement;
protected DoublyLinkedNode<E> previousElement;

public DoublyLinkedNode(E v,
                        DoublyLinkedNode<E> next,
                        DoublyLinkedNode<E> previous)
{
    data = v;
    nextElement = next;
    if (nextElement != null)
        nextElement.previousElement = this;
    previousElement = previous;
    if (previousElement != null)
        previousElement.nextElement = this;
}

public DoublyLinkedNode(E v)
// post: constructs a single element
{
    this(v,null,null);
}
```

Now we construct the class describing the doubly linked list, proper. As with any implementation of the list interface, it is necessary for our new `Doubly-LinkedList` to provide code for each method not addressed in the `AbstractList` class. The constructor simply sets the `head` and `tail` references to `null` and the `count` to 0—the state identifying an empty list:



`DoublyLinked-List`

```
protected int count;
protected DoublyLinkedNode<E> head;
protected DoublyLinkedNode<E> tail;

public DoublyLinkedList()
// post: constructs an empty list
{
    head = null;
    tail = null;
    count = 0;
}
```

Many of the fast methods of `SinglyLinkedLists`, like `addFirst`, require only minor modifications to maintain the extra references.

```
public void addFirst(E value)
// pre: value is not null
// post: adds element to head of list
{
    // construct a new element, making it head
    head = new DoublyLinkedNode<E>(value, head, null);
    // fix tail, if necessary
    if (tail == null) tail = head;
    count++;
}
```

The payoff for all our extra references comes when we implement methods like those modifying the tail of the list:

```
public void addLast(E value)
// pre: value is not null
// post: adds new value to tail of list
{
    // construct new element
    tail = new DoublyLinkedNode<E>(value, null, tail);
    // fix up head
    if (head == null) head = tail;
    count++;
}

public E removeLast()
// pre: list is not empty
// post: removes value from tail of list
{
    Assert.pre(!isEmpty(),"List is not empty.");
    DoublyLinkedNode<E> temp = tail;
    tail = tail.previous();
    if (tail == null) {
        head = null;
    } else {
        tail.setNext(null);
    }
    count--;
    return temp.value();
}
```

Here, it is easy to see that head- and tail-based methods are textually similar, making it easier to verify that they are written correctly. Special care needs to be taken when these procedures handle a list that newly becomes either empty or not empty. In these cases, *both* the `head` and `tail` references must be modified to maintain a consistent view of the list. Some people consider the careful manipulation of these references so time-consuming and error-prone that they dedicate an unused element that permanently resides at the head of the list. It

is never seen or modified by the user, and it can simplify the code. Here, for example, are the `addLast` and `removeLast` methods for this type of list:

```
public void addLast(E value)
{
    // construct new element
    tail = new DoublyLinkedNode<E>(value, null, tail);
    count++;
}

public E removeLast()
{
    Assert.pre(!isEmpty(),"List is not empty.");
    DoublyLinkedNode<E> temp = tail;
    tail = tail.previous();
    tail.setNext(null);
    count--;
    return temp.value();
}
```

The reserved-element technique increases the amount of space necessary to store a `DoublyLinkedList` by the size of a single element. The choice is left to the implementor and is another example of a time–space trade-off.

Returning to our original implementation, we note that `remove` is simplified by the addition of the `previous` reference:

```
public E remove(E value)
// pre: value is not null.  List can be empty
// post: first element matching value is removed from list
{
    DoublyLinkedNode<E> finger = head;
    while (finger != null &&
           !finger.value().equals(value))
    {
        finger = finger.next();
    }
    if (finger != null)
    {
        // fix next field of element above
        if (finger.previous() != null)
        {
            finger.previous().setNext(finger.next());
        } else {
            head = finger.next();
        }
        // fix previous field of element below
        if (finger.next() != null)
        {
            finger.next().setPrevious(finger.previous());
        } else {
```

```
            tail = finger.previous();
        }
        count--;            // fewer elements
        return finger.value();
    }
    return null;
}
```

Because every element keeps track of its previous element, there is no difficulty in finding it from the element that is to be removed. Of course, once the removal is to be done, several references need to be updated, and they must be assigned carefully to avoid problems when removing the first or last value of a list.

The `List` interface requires the implementation of two index-based methods called `indexOf` and `lastIndexOf`. These routines return the index associated with the first (or last) element that is equivalent to a particular value. The `indexOf` method is similar to the implementation of `contains`, but it returns the index of the element, instead of the element itself. For `DoublyLinkedLists`, the `lastIndexOf` method performs the same search, *but starts at the tail of the list*. It is, essentially, the mirror image of an `indexOf` method.

```
public int lastIndexOf(E value)
// pre: value is not null
// post: returns the (0-origin) index of value,
//   or -1 if value is not found
{
    int i = size()-1;
    DoublyLinkedNode<E> finger = tail;
    // search for last matching value, result is desired index
    while (finger != null && !finger.value().equals(value))
    {
        finger = finger.previous();
        i--;
    }
    if (finger == null)
    {   // value not found, return indicator
        return -1;
    } else {
        // value found, return index
        return i;
    }
}
```

9.6 Implementation: Circularly Linked Lists

Careful inspection of the singly linked list implementation identifies one seemingly unnecessary piece of data: the final reference of the list. This reference is always `null`, but takes up as much space as any varying reference. At the same time, we were motivated to add a tail reference in the doubly linked list to help

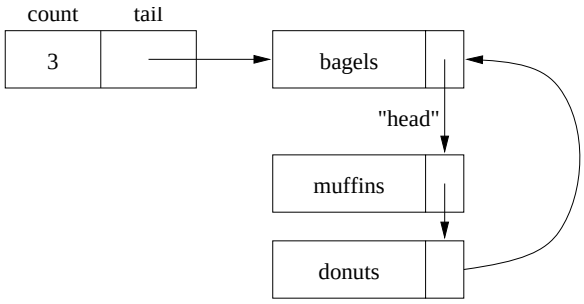
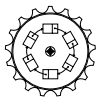


Figure 9.10 A nonempty circularly linked list.

us access either end of the list with equal ease. Perhaps we could use the last reference as the extra reference we need to keep track of one of the ends!

The tail wags the dog.

Here's the technique: Instead of keeping track of both a head and a tail reference, we explicitly keep only the reference to the tail. Since this element would normally have a `null` reference, we use that reference to refer, implicitly, to the head (see Figure 9.10). This implementation marries the speed of the `DoublyLinkedList` with the space needed by the `SinglyLinkedList`. In fact, we are able to make use of the `Node` class as the basis for our implementation. To build an empty list we initialize the tail to `null` and the `count` to 0:



`CircularList`

```
protected Node<E> tail;
protected int count;

public CircularList()
// pre: constructs a new circular list
{
    tail = null;
    count = 0;
}
```

Whenever access to the head of the list is necessary, we use `tail.next()`, instead.[3] Thus, methods that manipulate the head of the list are only slight modifications of the implementations we have seen before for singly and doubly linked lists. Here is how we add a value to the head of the list:

```
public void addFirst(E value)
// pre: value non-null
// post: adds element to head of list
{
```

[3] This longhand even works in the case when there is exactly one element, since its next reference points to itself.

```
    Node<E> temp = new Node<E>(value);
    if (tail == null) { // first value added
        tail = temp;
        tail.setNext(tail);
    } else { // element exists in list
        temp.setNext(tail.next());
        tail.setNext(temp);
    }
    count++;
}
```

Now, to add an element to the end of the list, we first add it to the head, and then "rotate" the list by moving the `tail` down the list. The overall effect is to have added the element to the tail!

```
public void addLast(E value)
// pre: value non-null
// post: adds element to tail of list
{
    // new entry:
    addFirst(value);
    tail = tail.next();
}
```

The "recycling" of the tail reference as a new head reference does not solve all our problems. Careful thought will demonstrate that the removal of a value from the tail of the list remains a difficult problem. Because we only have access to the tail of the list, and not the value that precedes it, it is difficult to remove the final value. To accomplish this, we must iterate through the structure, looking for an element that refers to the same element as the tail reference.

```
public E removeLast()
// pre: !isEmpty()
// post: returns and removes value from tail of list
{
    Assert.pre(!isEmpty(),"list is not empty.");
    Node<E> finger = tail;
    while (finger.next() != tail) {
        finger = finger.next();
    }
    // finger now points to second-to-last value
    Node<E> temp = tail;
    if (finger == tail)
    {
        tail = null;
    } else {
        finger.setNext(tail.next());
        tail = finger;
    }
```

```
        count--;
        return temp.value();
    }
```

There are two approaches to improving the performance of this operation. First, we could reconsider the `previous` links of the doubly linked list. There's not much advantage to doing this, and if we did, we could then keep the head reference instead of the tail reference. The second technique is to point instead to the element before the tail; that is the subject of Problem 9.11.

9.7 Implementation: Vectors

Careful inspection of the `List` interface makes it clear that the `Vector` class actually implements the `List` interface. Thus, we can augment the `Vector` definition with the phrase `implements List`.

With such varied implementations, it is important to identify the situations where each of the particular implementations is most efficient. As we had noted before, the `Vector` is a good random access data structure. Elements in the middle of the `Vector` can be accessed with little overhead. On the other hand, the operations of adding or removing a value from the front of the `Vector` are potentially inefficient, since a large number of values must be moved in each case.

In contrast, the dynamically allocated lists manipulate the head of the list quite efficiently, but do not allow the random access of the structure without a significant cost. When the tail of a dynamically allocated list must be accessed quickly, the `DoublyLinkedList` or `CircularList` classes should be used.

Exercise 9.2 *In Exercise 6.3 (see page 144) we wrote a version of* `insertionSort` *that sorts* `Vectors`*. Follow up on that work by making whatever modification would be necessary to have the* `insertionSort` *work on any type of* `List`*.*

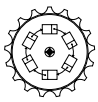
9.8 List Iterators

The observant reader will note that all classes that implement the `Structure` class (see page 24) are required to provide an `iterator` method. Since the `List` interface extends the `Structure` interface, all `Lists` are required to implement an `iterator` method. We sketch the details of an `Iterator` over `SinglyLinkedLists` here. Implementations of other `List`-based iterators are similar.

When implementing the `VectorIterator` it may be desirable to use only methods available through the `Vector`'s public interface to access the `Vector`'s data. Considering the `List` interface—an interface biased toward manipulating the ends of the structure—it is not clear how a traversal might be accomplished without disturbing the underlying `List`. Since several `Iterators` may be active on a single `List` at a time, it is important not to disturb the host structure.

As a result, efficient implementations of `ListIterator`s must make use of the protected fields of the `List` object.

The `SinglyLinkedListIterator` implements all the standard `Iterator` methods. To maintain its positioning within the `List`, the iterator maintains two references: the head of the associated list and a reference to the current node. The constructor and initialization methods appear as follows:



SinglyLinked-
ListIterator

```
protected Node<E> current;
protected Node<E> head;

public SinglyLinkedListIterator(Node<E> t)
// post: returns an iterator that traverses a linked list
{
    head = t;
    reset();
}

public void reset()
// post: iterator is reset to beginning of traversal
{
    current = head;
}
```

When called by the `SinglyLinkedList`'s `iterator` method, the protected `head` reference is passed along. The constructor caches away this value for use in `reset`. The `reset` routine is then responsible for initializing `current` to the value of `head`. The `Iterator` is able to refer to the `Node`s because both structures are in the same package.

The value-returning routines visit each element and "increment" the `current` reference by following the `next` reference:

```
protected Node<E> current;
protected Node<E> head;

public boolean hasNext()
// post: returns true if there is more structure to be viewed:
//       i.e., if value (next) can return a useful value.
{
    return current != null;
}

public E next()
// pre: traversal has more elements
// post: returns current value and increments iterator
{
    E temp = current.value();
    current = current.next();
    return temp;
}
```

The traversal is finished when the `current` reference "falls off" the end of the `List` and becomes `null`.

Observe that the `Iterator` is able to develop references to values that are not accessible through the public interface of the underlying `List` structure. While it is of obvious utility to access the middle elements of the `List`, these references could be used to modify the associated `List` structure. If the objects referred to through the `Iterator` are modified, this underlying structure could become corrupted. One solution to the problem is to return copies or *clones* of the current object, but then the references returned are not really part of the `List`. The best advice is to think of the values returned by the `Iterator` as *read-only*.

Principle 15 *Assume that values returned by iterators are read-only.*

9.9 Conclusions

In this chapter we have developed the notion of a list and three different implementations. One of the features of the list is that as each of the elements is added to the list, the structure is expanded dynamically, using dynamic memory. To aid in keeping track of an arbitrarily large number of chunks of dynamic memory, we allocate, with each chunk, at least one reference for keeping track of logically nearby memory.

Although the description of the interface for lists is quite detailed, none of the details of any particular implementation show through the interface. This approach to designing data structures makes it less possible for applications to depend on the peculiarities of any particular implementation, making it more likely that implementations can be improved without having to reconsider individual applications.

Finally, as we investigated each of the three implementations, it became clear that there were certain basic trade-offs in good data structure design. Increased speed is often matched by an increased need for space, and an increase in complexity makes the code less maintainable. We discuss these trade-offs in more detail in upcoming chapters.

Self Check Problems

Solutions to these problems begin on page 446.

9.1 What are the essential distinctions between the `List` types and the `Vector` implementation?

9.2 Why do most `List` implementations make use of one or more references for each stored value?

9.3 How do we know if a structure qualifies as a `List`?

9.4 If class `C` extends the `SinglyLinkedList` class, is it a `SinglyLinkedList`? Is it a `List`? Is it an `AbstractList`? Is it a `DoublyLinkedList`?

9.5 The `DoublyLinkedList` class has elements with two pointers, while the `SinglyLinkedList` class has elements with one pointer. Is `DoublyLinkedList` a `SinglyLinkedList` with additional information?

9.6 Why do we have a `tail` reference in the `DoublyLinkedList`?

9.7 Why don't we have a `tail` reference in the `SinglyLinkedList`?

9.8 The `ListVector` implementation of a `List` is potentially slow? Why might we use it, in any case?

9.9 The `AbstractList` class does not make use of any element types or references. Why?

9.10 If you use the `add` method to add an element to a `List`, to which end does it get added?

9.11 The `get` and `set` methods take an integer index. Which element of the list is referred to by index 1?

Problems

Solutions to the odd-numbered problems begin on page 471.

9.1 When considering a data structure it is important to see how it works in the *boundary cases*. Given an empty `List`, which methods may be called without violating preconditions?

9.2 Compare the implementation of `getLast` for each of the three `List` types we have seen in this chapter.

9.3 From within Java programs, you may access information on the Web using URL's (uniform resource locators). Programmers at MindSlave software (working on their new NetPotato browser) would like to keep track of a potentially large bookmark list of frequently visited URL's. It would be most useful if they had arbitrary access to the values saved within the list. Is a `List` an appropriate data structure? (Hint: If not, why?)

9.4 Write a `List` method, `equals`, that returns `true` exactly when the elements of two lists are pair-wise equal. Ideally, your implementation should work for any `List` implementation, without change.

9.5 Write a method of `SinglyLinkedList`, called `reverse`, that reverses the order of the elements in the list. This method should be *destructive*—it should modify the list upon which it acts.

9.6 Write a method of `DoublyLinkedList`, called `reverse`, that reverses the order of the elements in the list. This method should be destructive.

9.7 Write a method of `CircularList`, called `reverse`, that reverses the order of the element in the list. This method should be destructive.

9.8 Each of the n references in a singly linked list are needed if we wish to remove the final element. In a doubly linked list, are each of the additional n `previous` references necessary if we want to remove the tail of the list in constant time? (Hint: What would happen if we mixed `Nodes` and `DoublyLinkedNodes`?)

9.9 Design a method that inserts an object into the middle of a `CircularList`.

9.10 Which implementation of the `size` and `isEmpty` methods would you use if you had the potential for a million-element list. (Consider the problem of keeping track of the alumni for the University of Michigan.) How would you choose if you had the potential for a million small lists. (Consider the problem of keeping track of the dependents for each of a million income-tax returns.)

9.11 One way to make all the circular list operations run quickly is to keep track of the element that points to the last element in the list. If we call this `penultimate`, then the tail is referenced by `penultimate.next`, and the head by `penultimate.next.next`. What are the disadvantages of this?

9.12 Suppose we read n integers $1, 2, \ldots, n$ from the input, in order. Flipping a coin, we add each new value to either the head or tail of the list. Does this shuffle the data? (Hint: See Problem 6.18.)

9.13 Measure the performance of `addFirst`, `remove(Object)`, and `removeLast` for each of the three implementations (you may include `Vectors`, if you wish). Which implementations perform best for small lists? Which implementations perform best for large lists?

9.14 Consider the implementation of an `insertionSort` that works on `Lists`. (See Exercises 6.3 and 9.2.) What is the worst-case performance of this sort? Be careful.

9.15 Implement a recursive version of the `size` method for `SinglyLinkedLists`. (Hint: A wrapper may be useful.)

9.16 Implement a recursive version of the `contains` method for `SinglyLinkedLists`.

9.17 Suppose the `add` of the `Unique` program is replaced by `addFirst` and the program is run on (for example) the first chapter of Mark Twain's *Tom Sawyer*. Why does the modified program run as much as 25 percent *slower* than the program using the `add` (i.e., `addLast`) method? (Hint: Mark Twain didn't write randomly.)

9.18 Describe an implementation for an iterator associated with `CircularLists`.

9.10 Laboratory: Lists with Dummy Nodes

Objective. To gain experience implementing `List`-like objects.

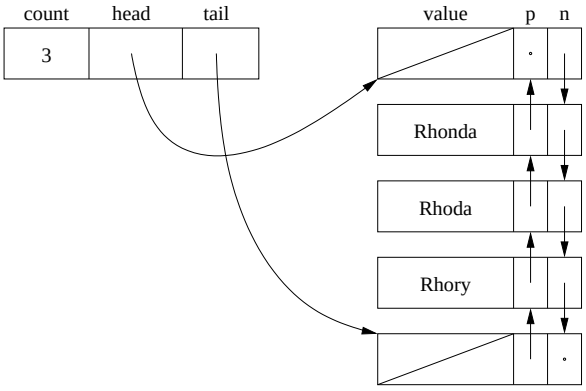
Discussion. Anyone attempting to understand the workings of a doubly linked list understands that it is potentially difficult to keep track of the references. One of the problems with writing code associated with linked structures is that there are frequently *boundary cases*. These are special cases that must be handled carefully because the "common" path through the code makes an assumption that does not hold in the special case.

Take, for example, the `addFirst` method for `DoublyLinkedLists`:

```
public void addFirst(E value)
// pre: value is not null
// post: adds element to head of list
{
    // construct a new element, making it head
    head = new DoublyLinkedNode<E>(value, head, null);
    // fix tail, if necessary
    if (tail == null) tail = head;
    count++;
}
```

The presence of the `if` statement suggests that sometimes the code must reassign the value of the `tail` reference. Indeed, if the list is empty, the first element must give an initial non-`null` value to `tail`. Keeping track of the various special cases associated with a structure can be very time consuming and error-prone.

One way that the complexity of the code can be reduced is to introduce *dummy nodes*. Usually, there is one dummy node associated with each external reference associated with the structure. In the `DoublyLinkedList`, for example, we have two references (`head` and `tail`); both will refer to a dedicated dummy node:



These nodes appear to the code to be normal elements of the list. In fact, they do not hold any useful data. They are completely hidden by the abstraction of the data structure. They are *transparent*.

Because most of the boundary cases are associated with maintaining the correct values of external references and because these external references are now "hidden" behind their respective dummy nodes, most of the method code is simplified. This comes at some cost: the dummy nodes take a small amount of space, and they must be explicitly stepped over if we work at either end of the list. On the other hand, the total amount of code to be written is likely to be reduced, and the running time of many methods decreases if the special condition testing would have been expensive.

Procedure. In this lab we will extend the `DoublyLinkedList`, building a new class, `LinkedList`, that makes use of two dummy nodes: one at the head of the list, and one at the end.

`LinkedList`

You should begin taking a copy of the `LinkedList.java` starter file. This file simply declares `LinkedList` to be an extension of the `structure` package's `DoublyLinkedList` class. The code associated with each of the existing methods is similar to the code from `DoublyLinkedList`. You should replace that code with working code that makes use of two dummy nodes:

1. First, recall that the three-parameter constructor for `DoublyLinkedList-Elements` takes a value and two references—the nodes that are to be `next` and `previous` to this new node. That constructor will also update the `next` and `previous` nodes to point to the newly constructed node. You may find it useful to use the one-parameter constructor, which builds a node with `null` `next` and `previous` references.

2. Replace the constructor for the `LinkedList`. Instead of constructing `head` and `tail` references that are `null`, you should construct two dummy nodes; one node is referred to by `head` and the other by `tail`. These dummy nodes should point to each other in the natural way. Because these dummy nodes replace the `null` references of the `DoublyLinkedList` class, we will not see any need for `null` values in the rest of the code. Amen.

3. Check and make necessary modifications to `size`, `isEmpty`, and `clear`.

4. Now, construct two important `protected` methods. The method `insert-After` takes a `value` and a reference to a node, `previous`. It inserts a new node with the value `value` that directly follows `previous`. It should be declared `protected` because we are not interested in making it a formal feature of the class. The other method, `remove`, is given a reference to a node. It should unlink the node from the linked list and return the value stored in the node. You should, of course, assume that the node removed is not one of the dummy nodes. These methods should be simple with no `if` statements.

5. Using `insertAfter` and `remove`, replace the code for `addFirst`, `addLast`, `getFirst`, `getLast`, `removeFirst`, and `removeLast`. These methods should be very simple (perhaps one line each), with no `if` statements.

6. Next, replace the code for the indexed versions of methods `add`, `remove`, `get`, and `set`. Each of these should make use of methods you have already written. They should work without any special `if` statements.

7. Finally, replace the versions of methods `indexOf`, `lastIndexOf`, and `contains` (which can be written using `indexOf`), and the `remove` method that takes an object. Each of these searches for the location of a value in the list and then performs an action. You will find that each of these methods is simplified, making no reference to the `null` reference.

Thought Questions. Consider the following questions as you complete the lab:

1. The three-parameter constructor for `DoublyLinkedNodes` makes use of two `if` statements. Suppose that you replace the calls to this constructor with the one-parameter constructor and manually use `setNext` and `setPrevious` to set the appropriate references. The `if` statements disappear. Why?

2. The `contains` method can be written making use of the `indexOf` method, but not the other way around. Why?

3. Notice that we could have replaced the method `insertAfter` with a similar method, `insertBefore`. This method inserts a new value *before* the indicated node. Some changes would have to be made to your code. There does not appear, however, to be a choice between versions of `remove`. Why is this the case? (Hint: Do you ever pass a dummy node to `remove`?)

4. Even though we don't need to have the special cases in, for example, the indexed version of `add`, it is desirable to handle one or more cases in a special way. What are the cases, and why is it desirable?

5. Which file is bigger: your final result source or the original?

Notes: