Chapter 11

Ordered Structures

Concepts:

- ▷ The `java.lang.Comparable` interface
- ▷ The `java.util.Comparator`
- ▷ The `OrderedStructure` interface
- ▷ The `OrderedVector`
- ▷ The `OrderedList`

"Make no mistake about it.
A miracle has happened. . .
we have no ordinary pig."
"Well," said Mrs. Zuckerman,
"it seems to me you're a little off.
It seems to me we have
no ordinary spider*."*
—Elwyn Brooks White

WE HAVE MADE NO ASSUMPTIONS about the type of data we store within our structures—so far. Instead, we have assumed only that the data referenced are a subclass of the type `Object`. Recall that *all* classes are subtypes of `Object` in Java, so this is hardly a constraint. Data structures serve a purpose, often helping us perform tasks more complex than "just holding data." For example, we used the `Stack` and `Queue` classes to *guide* a search through search space in the previous chapter.

One important use of data structures is to help keep data in order—the smallest value in the structure might be stored close to the front, while the largest value would be stored close to the rear. Once a structure is ordered it becomes potentially useful as a mechanism for sorting: we simply insert our possibly unordered data into the structure and then extract the values in order. To do this, however, it is necessary to *compare* data values to see if they are in the correct order. In this chapter we will discuss approaches to the various problems associated with maintaining ordered structures. First we review material we first encountered when we considered sorting.

11.1 Comparable Objects Revisited

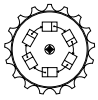
In languages like C++ it is possible to *override* the comparison operators (<, >, ==, etc.). When two objects are compared using these operators, a user-written method is called. Java does not support overriding of built-in operators. Thus, it is useful to come up with a convention for supporting *comparable* data.

First, let's look closely at the interface for Java's `Object`. Since every class inherits and extends the interface of the `Object` class, each of its methods may be applied to any class. For example, the `equals` method allows us to check

if an `Object` is logically equal to another `Object`. In contrast, the `==` operator compares two *references* to see if they refer to the same *instance* of an object.

By default, the `equals` function returns `true` whenever two references point to exactly the same object. This is often not the correct comparison—often we wish to have different instances of the same type be equal—so the class designer should consider rewriting it as a class-specific method.

For our purposes, we wish to require of comparable classes a method that determines the relative order of objects. How do we *require* this? Through an interface! Since an interface is a contract, we simply wrap the `compareTo` method in a language-wide interface, `java.lang.Comparable`:



Comparable

```
public interface Comparable<T>
{
    public int compareTo(T that);
}
```

This is pretty simple: When we want to compare two objects of type `T`, we simply call the `compareTo` method of one on another. Now, if we require that an object be a `Comparable` object, then we know that it may be compared to similarly typed data using the `compareTo` method.

11.1.1 Example: Comparable Ratios

Common types, such as `Integers` and `Strings`, include a `compareTo` method. In this section we add methods to make the `Ratio` class comparable. Recall that a `Ratio` has the following interface:

Ratio

```
public class Ratio
       implements Comparable<Ratio>
{
    public Ratio(int top, int bottom)
    // pre: bottom != 0
    // post: constructs a ratio equivalent to top::bottom

    public int getNumerator()
    // post: return the numerator of the fraction

    public int getDenominator()
    // post: return the denominator of the fraction

    public double getValue()
    // post: return the double equivalent of the ratio

    public Ratio add(Ratio other)
    // pre: other is nonnull
    // post: return new fraction--the sum of this and other

    public String toString()
```

```
    // post: returns a string that represents this fraction.

    public int compareTo(Ratio that)
    // pre: other is non-null and type Ratio
    // post: returns value <, ==, > 0 if this value is <, ==, > that

    public boolean equals(Object that)
    // pre: that is type Ratio
    // post: returns true iff this ratio is the same as that ratio
}
```

A `Ratio` is constructed by passing it a pair of integers. These integers are cached away—we cannot tell how—where they can later be used to compare their ratio with another `Ratio`. The protected data and the constructor that initializes them appear as follows:

```
protected int numerator;   // numerator of ratio
protected int denominator; // denominator of ratio

public Ratio(int top, int bottom)
// pre: bottom != 0
// post: constructs a ratio equivalent to top::bottom
{
    numerator = top;
    denominator = bottom;
    reduce();
}
```

We can see, now, that this class has a pair of protected `int`s to hold the values. Let us turn to the `compareTo` method. The `Comparable` interface for a type `Ratio` declares the `compareTo` method to take a `Ratio` parameter, so we expect the parameter to be a `Ratio`; it's not useful to compare `Ratio` types to non-`Ratio` types. Implementation is fairly straightforward:

```
public int compareTo(Ratio that)
// pre: other is non-null and type Ratio
// post: returns value <, ==, > 0 if this value is <, ==, > that
{
    return this.getNumerator()*that.getDenominator()-
           that.getNumerator()*this.getDenominator();
}
```

The method checks the order of two ratios: the values stored within `this` ratio are compared to the values stored in `that` ratio and an integer is returned. The relationship of this integer to zero reprepresents the relationship between `this` and `that`. The *magnitude* of the result is unimportant (other than being zero or non-zero).

We can now consider the `equals` method:

```
public boolean equals(Object that)
// pre: that is type Ratio
// post: returns true iff this ratio is the same as that ratio
{
    return compareTo((Ratio)that) == 0;
}
```

Conveniently, the `equals` method can be cast in terms of the `compareTo` method. For the `Ratio` class, the `compareTo` method is not much more expensive to compute than the `equals`, so this "handing off" of the work does not cost much. For more complex classes, the `compareTo` method may be so expensive that a consistent `equals` method can be constructed using independently considered code. In either case, it is important that `equals` return `true` exactly when the `compareTo` method returns 0.

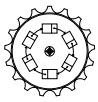
Note also that the parameter to the `equals` method is declared as an `Object`. If it is not, then the programmer is writing a *new* method, rather than overriding the default method inherited from the `Object` class. Because `compareTo` compares `Ratio` types, we must *cast* the type of `that` to be a `Ratio`. If `that` is a `Ratio` (or a subclass), the compare will work. If not, then a cast error will occur at this point. Calling `compareTo` is the correct action here, since equal `Ratios` may appear in different object instances. Direct comparison of references is not appropriate. Failure to correctly implement the `equals` (or `compareTo`) method can lead to very subtle logical errors.

Principle 17 *Declare parameters of overriding methods with the most general types possible.*

To reiterate, failure to correctly declare these methods as generally as possible makes it less likely that Java will call the correct method.

11.1.2 Example: Comparable Associations

Let us return now to the idea of an `Association`. An `Association` is a key-value pair, bound together in a single class. For the same reasons that it is sometimes nice to be able to compare integers, it is often useful to compare `Associations`. Recall that when we constructed an `Association` we took great care in defining the `equals` operator to work on just the `key` field of the `Association`. Similarly, when we extend the concept of an `Association` to its `Comparable` equivalent, we will have to be just as careful in constructing the `compareTo` method.



`Comparable-Association`

Unlike the `Ratio` class, the `ComparableAssociation` can be declared an extension of the `Association` class. The outline of this extension appears as follows:

```
public class ComparableAssociation<K extends Comparable<K>,V>
    extends Association<K,V>
    implements Comparable<ComparableAssociation<K,V>>
{
```

```
    public ComparableAssociation(K key)
    // pre: key is non-null
    // post: constructs comparable association with null value

    public ComparableAssociation(K key, V value)
    // pre: key is non-null
    // post: constructs association between a comparable key and a value

    public int compareTo(ComparableAssociation<K,V> that)
    // pre: other is non-null ComparableAssociation
    // post: returns integer representing relation between values
}
```

Notice that there are very few methods. Since `ComparableAssociation` is an *extension* of the `Association` class, all the methods written for `Association` are available for use with `ComparableAssociations`. The only additions are those shown here. Because one of the additional methods is the `compareTo` method, it meets the specification of what it means to be `Comparable`; thus we claim it implements the `Comparable` interface.

Let's look carefully at the implementation. As with the `Association` class, there are two constructors for `ComparableAssociations`. The first constructor initializes the `key` and sets the `value` reference to `null`, while the second initializes both `key` and `value`:

```
public ComparableAssociation(K key)
// pre: key is non-null
// post: constructs comparable association with null value
{
    this(key,null);
}

public ComparableAssociation(K key, V value)
// pre: key is non-null
// post: constructs association between a comparable key and a value
{
    super(key,value);
}
```

Remember that there are two special methods available to constructors: `this` and `super`. The `this` method calls another constructor with a different set of parameters (if the parameters are not different, the constructor could be recursive!). We write one very general purpose constructor, and any special-purpose constructors call the general constructor with reconsidered parameter values. The `super` method is a means of calling the constructor for the superclass—the class we are extending—`Association`. The second constructor simply calls the constructor for the superclass. The first constructor calls the second constructor (which, in turn, calls the superclass's constructor) with a `null value` field. All of this is necessary to be able to construct `ComparableAssociations` using the `Association`'s constructors.

Now, the compareTo method requires some care:

```
public int compareTo(ComparableAssociation<K,V> that)
// pre: other is non-null ComparableAssociation
// post: returns integer representing relation between values
{
    return this.getKey().compareTo(that.getKey());
}
```

Because the compareTo method must implement the Comparable interface between ComparableAssociations, its parameter is an ComparableAssociation. Careful thought suggests that the relationship between the associations is completely determined by the relationship between their respective keys.

Since ComparableAssociations are associations with comparable keys, we know that the key within the association has a compareTo method. Java would not able to deduce this on its own, so we must give it hints, by declaring the type parameter K to be any type that implements Comparable<K>. The declaration, here, is essentially a catalyst to get Java to verify that a referenced object has certain type characteristics. In any case, we get access to both keys through independent variables. These allow us to make the comparison by calling the compareTo method on the comparable objects. Very little logic is directly encoded in these routines; we mostly make use of the prewritten code to accomplish what we need.

In the next few sections we consider features that we can provide to existing data structures, provided that the underlying data are known to be comparable.

11.2 Keeping Structures Ordered

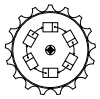
We can make use of the natural ordering of classes suggested by the compareTo method to organize our structure. Keeping data in order, however, places significant constraints on the type of operations that should be allowed. If a comparable value is added to a structure that orders its elements, the relative position of the new value is determined by the data, not the structure. Since this placement decision is predetermined, ordered structures have little flexibility in their interface. It is not possible, for example, to insert data at random locations. While simpler to use, these operations also tend to be more costly than their unordered counterparts. Typically, the increased *energy* required is the result of an increase in the friction associated with decisions needed to accomplish add and remove.

The implementation of the various structures we see in the remainder of this chapter leads to simpler algorithms for sorting, as we will see in Section 11.2.3.

11.2.1 The OrderedStructure Interface

Recall that a Structure is any traversable structure that allows us to add and remove elements and perform membership checks (see Section 1.8). Since the

`Structure` interface also requires the usual size-related methods (e.g., `size`, `isEmpty`, `clear`), none of these methods actually requires that the data within the structure be kept in order. To ensure that the structures we create order their data (according to their native ordering), we make them abide by an extended interface—an `OrderedStructure`:



`Ordered-Structure`

```
public interface OrderedStructure<K extends Comparable<K>>
         extends Structure<K>
{
}
```

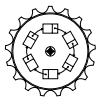
The emperor wears no clothes!

Amazingly enough we have accomplished something for almost nothing! Actually, what is happening is that we are using the *type* to store the fact that the data can be kept in sorted order. Encoded in this, of course, is that we are working with `Comparable` values of generic type `K`.

11.2.2 The Ordered Vector and Binary Search

We can now consider the implementation of an ordered `Vector` of values. Since it implements an `OrderedStructure`, we know that the order in which elements are added does not directly determine the order in which they are ultimately removed. Instead, when elements are added to an `OrderedVector`, they are kept ascending in their natural order.

Constructing an ordered `Vector` requires little more than allocating the underlying vector:



`OrderedVector`

```
public class OrderedVector<E extends Comparable<E>>
    extends AbstractStructure<E>
    implements OrderedStructure<E>
{
    public OrderedVector()
    // post: constructs an empty, ordered vector
    {
        data = new Vector<E>();
    }
}
```

Rather obviously, if there are no elements in the underlying `Vector`, then all of the elements are in order. Initially, at least, the structure is in a consistent state. We must always be mindful of consistency.

Because finding the correct location for a value is important to both adding and removing values, we focus on the development of an appropriate search technique for `OrderedVectors`. This process is much like looking up a word in a dictionary, or a name in a phone book (see Figure 11.1). First we look at the value halfway through the `Vector` and determine if the value for which we are looking is bigger or smaller than this *median*. If it is smaller, we restart our search with the left half of the structure. If it is bigger, we restart our search with the right half of the `Vector`. Whenever we consider a section of the `Vector`

consisting of a single element, the search can be terminated, with the success of the search dependent on whether or not the indicated element contains the desired value. This approach is called *binary search*.

We present here the code for determining the index of a value in an `OrderedVector`. Be aware that if the value is not in the `Vector`, the routine returns the ideal location to insert the value. This may be a location that is outside the `Vector`.

```
protected int locate(E target)
{
    Comparable<E> midValue;
    int low = 0;  // lowest possible location
    int high = data.size(); // highest possible location
    int mid = (low + high)/2; // low <= mid <= high
    // mid == high iff low == high
    while (low < high) {
        // get median value
        midValue = data.get(mid);
        // determine on which side median resides:
        if (midValue.compareTo(target) < 0) {
            low = mid+1;
        } else {
            high = mid;
        }
        // low <= high
        // recompute median index
        mid = (low+high)/2;
    }
    return low;
}
```

For each iteration through the loop, `low` and `high` determine the bounds of the `Vector` currently being searched. `mid` is computed to be the middle element (if there are an even number of elements being considered, it is the leftmost of the two middle elements). This middle element is compared with the parameter, and the bounds are adjusted to further constrain the search. Since the portion of the `Vector` participating in the search is roughly halved each time, the total number of times around the loop is approximately $O(\log n)$. This is a considerable improvement over the implementation of the `indexOf` method for `Vector`s of arbitrary elements—that routine is *linear* in the size of the structure.

Notice that `locate` is declared as a `protected` member of the class. This makes it impossible for a user to call directly, and makes it more difficult for a user to write code that depends on the underlying implementation. To convince yourself of the utility of this, both `OrderedStructures` of this chapter have exactly the same interface (so these two data types can be interchanged), but they are completely different structures. If the `locate` method were made public, then code could be written that makes use of this `Vector`-specific method, and it would be impossible to switch implementations.

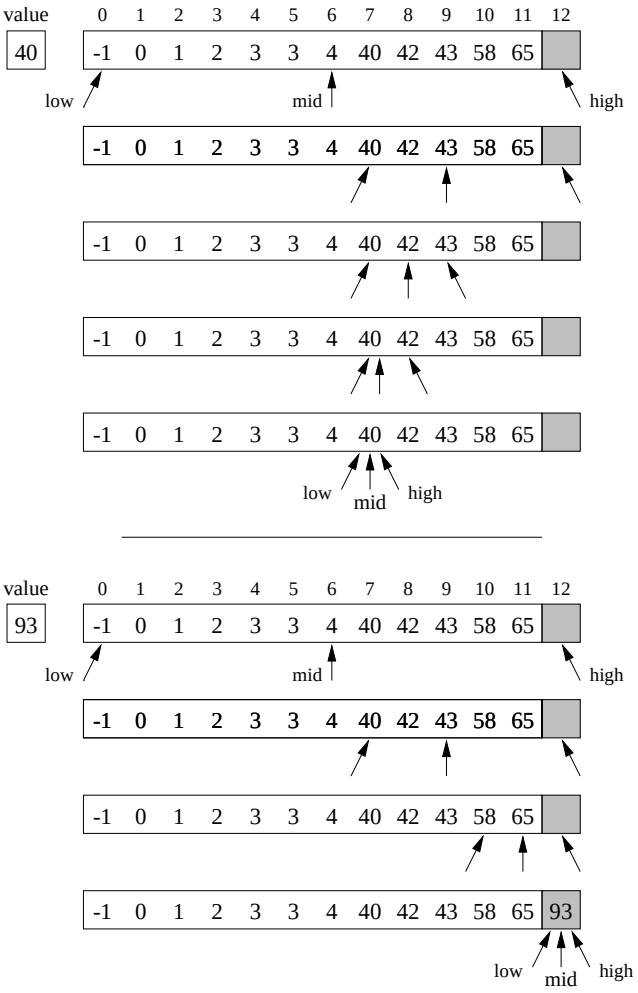


Figure 11.1 Finding the correct location for a comparable value in an ordered array. The top search finds a value in the array; the bottom search fails to find the value, but finds the correct point of insertion. The shaded area is not part of the `Vector` during search.

Implementation of the `locate` method makes most of the nontrivial `OrderedVector` methods more straightforward. The `add` operator simply adds an element to the `Vector` in the position indicated by the `locate` operator:

```
public void add(E value)
// pre: value is non-null
// post: inserts value, leaves vector in order
{
    int position = locate(value);
    data.add(position,value);
}
```

It is interesting to note that the cost of looking up the value is $O(\log n)$, but the `insertElementAt` for relatively "small" values *can* take $O(n)$ time to insert. Thus, the worst-case (and expected—see Problem 11.6) time complexity of the `add` operation is $O(n)$, linear in the size of the structure. In reality, for large `Vectors`, the time required to find the index of a value using the `OrderedVector` method is significantly reduced over the time required using the underlying `Vector` method. If the cost of comparing two objects exceeds the cost of assigning one object to another, the use of binary search can be expected to reduce the cost of the `add` operation by as much as a factor of 2.

Both `contains` and `remove` can also make use of the `locate` operator. First, we consider testing to see if an element is contained by the `OrderedVector`:

```
public boolean contains(E value)
// pre: value is non-null
// post: returns true if the value is in the vector
{
    int position = locate(value);
    return (position < size()) &&
           data.get(position).equals(value);
}
```

We simply attempt to find the item in the `Vector`, and if the location returned contains the value we desire, we return `true`; otherwise we return `false`. Since `locate` takes $O(\log n)$ time and the check for element equality is constant, the total complexity of the operation is $O(\log n)$. The `Vector` version of the same operation is $O(n)$ time. This is a considerable improvement.

The `return` statement, you will note, returns the result of a *logical and* (`&&`) operator. This is a *short-circuiting* operator: if, after evaluating the left half of the expression, the ultimate value of the expression is known to be false, then the second expression is not evaluated. That behavior is used here to avoid calling the `get` operator with a `position` that might exceed the size of the structure, that is, the length of the `Vector`. This is a feature of many languages, but a potential trap if you ever consider reordering your boolean expressions.

Removing a value from an `OrderedVector` involves finding it within the `Vector` and then explicitly extracting it from the structure:

```
public E remove(E value)
// pre: value is non-null
// post: removes one instance of value, if found in vector
{
    if (contains(value)) {
        // we know value is pointed to by location
        int position = locate(value);
        // since vector contains value, position < size()
        // keep track of the value for return
        E target = data.get(position);
        // remove the value from the underlying vector
        data.remove(position);
        return target;
    }
    return null;
}
```

Like `add`, the operation has complexity $O(n)$. But it executes faster than its `Vector` equivalent, `removeElement`.

Note that by keeping the elements sorted, we have made adding and removing an element from the `OrderedVector` relatively symmetric: the expected complexity of each method is $O(n)$. Yet, in the underlying `Vector`, an `addElement` operation takes constant time, while the `removeElement` operation takes $O(n)$ time.

Extracting values in order from an `OrderedStructure` is accomplished by an iterator returned from the `elements` method. Because the elements are stored in the correct order in the `Vector`, the method need only return the value of the `Vector`'s `iterator` method:

```
public Iterator<E> iterator()
{
    return data.iterator();
}
```

The ease of implementing this particular method reassures us that our layout of values within the vector (in ascending order) is appropriate. The rest of the `OrderedVector` operators repackage similar operators from the `Vector` class:

```
public boolean isEmpty()
// post: returns true if the OrderedVector is empty
{
    return data.size() == 0;
}

public void clear()
// post: vector is emptied
{
    data.setSize(0);
}
```

```
public int size()
// post: returns the number of elements in vector
{
    return data.size();
}
```

This "repackaging" brings up a point: *Why is it necessary?* If one were to, instead, consider the `OrderedVector` to be an *extension* of the `Vector` class, much of this repackaging would be unnecessary, because each of the repackaged methods could be inherited, and those—like `add`, `contains`, and `remove`—that required substantial reconsideration could be rewritten overriding the methods provided in the underlying `Vector` class.

That's all true! There's one substantial drawback, however, that is uncovered by asking a simple question: *Is an* `OrderedVector` *suitably used wherever a* `Vector` *is used?* The answer is: *No!* Consider the following hypothetical code that allocates an `OrderedVector` for use as a `Vector`:

```
static void main(String args[])
{
    OrderedVector<String> v = new OrderedVector<String>();

    v.add("Michael's Pizza");
    v.add(1,"Cozy Pizza");
    v.add(0,"Hot Tomatoes Pizza");;
}
```

First, the `add` methods are not methods for `OrderedVectors`. Assuming this could be done, the semantics become problematic. We are inserting elements at specific locations within a `Vector`, but it is really an `OrderedVector`. The values inserted violate the ordering of elements and the postconditions of the `add` method of the `OrderedVector`.

We now consider a simple application of `OrderedStructures`—sorting.

11.2.3 Example: Sorting Revisited

Now that we have seen the implementation of an `OrderedStructure`, we can use these structures to sort comparable values. (If values are not comparable, it is hard to see how they might be sorted, but we will see an approach in Section 11.2.4.) Here is a program to sort integers appearing on the input:

```
public static void main(String[] args)
{
    Scanner s = new Scanner(System.in);
    OrderedStructure<Integer> o = new OrderedVector<Integer>();
    // read in integers
    while (s.hasNextInt())
    {
        o.add(s.nextInt());
    }
    // and print them out, in order
    for (Integer i : o)
    {
        System.out.println(i);
    }
}
```

Sort

In this simple program a sequence of numbers is read from the input stream. Each number is placed within an `Integer` that is then inserted into the `OrderedStructure`, in this case an `OrderedVector`. The insertion of this value into the `Vector` may involve moving, on average, $\frac{n}{2}$ elements out of the way. As the n values are added to the `Vector`, a total of $O(n^2)$ values have to be moved. The overall effect of this loop is to perform insertion sort! Once the values have been inserted in the ordered structure, we can use an iterator to traverse the `Vector` in order and print out the values in order. If the `OrderedVector` is substituted with any structure that meets the `OrderedStructure` interface, similar results are generated, but the performance of the sorting algorithm is determined by the complexity of insertion.

Now, what should happen if we don't have a `Comparable` data type?

11.2.4 A Comparator-based Approach

Sometimes it is not immediately obvious how we should generally order a specific data type, or we are hard-pressed to commit to one particular ordering for our data. In these cases we find it useful to allow ordered structures to be ordered in alternative ways. One approach is to have the ordered structure keep track of a `Comparator` that can be used when the `compareTo` method does not seem appropriate. For example, when constructing a list of `Integer` values, it may be useful to have them sorted in descending order.

The approach seems workable, but somewhat difficult when a comparison needs to actually be made. We must, at that time, check to see if a `Comparator` has somehow been associated with the structure and make either a `Comparator`-based `compare` *or* a class-based `compareTo` method call. We can greatly simplify the code if we assume that a `Comparator` method can always be used: we construct a `Comparator`, the `structure` package's `NaturalComparator`, that calls the `compareTo` method of the particular elements and returns that value for `compare`:

Natural-
Comparator

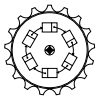
```
import java.util.Comparator;

public class NaturalComparator<E extends Comparable<E>>
                 implements Comparator<E>
{
    public int compare(E a, E b)
    // pre: a, b non-null, and b is same type as a
    // post: returns value <, ==, > 0 if a <, ==, > b
    {
        return a.compareTo(b);
    }

    public boolean equals(Object b)
    // post: returns true if b is a NaturalComparator
    {
        return (b != null) && (b instanceof NaturalComparator);
    }
}
```

The `NaturalComparator`, can then serve as a default comparison method in classes that wish to make exclusive use of the `Comparator`-based approach.

To demonstrate the power of the `Comparator`-based approach we can develop a notion of `Comparator` composition: one `Comparator` can be used to modify the effects of a *base* `Comparator`. Besides the `NaturalComparator`, the `structure` package also provides a `ReverseComparator` class. This class keeps track of its base `Comparator` in a protected variable, `base`. When a `ReverseComparator` is constructed, another `Comparator` can be passed to it to reverse. Frequently we expect to use this class to reverse the natural order of values, so we provide a parameterless constructor that forces the base `Comparator` to be `NaturalComparator`:



Reverse-
Comparator

```
protected Comparator<E> base; // comparator whose ordering is reversed

public ReverseComparator()
// post: constructs a comparator that orders in reverse order
{
    base = new NaturalComparator<E>();
}

public ReverseComparator(Comparator<E> base)
// post: constructs a Comparator that orders in reverse order of base
{
    this.base = base;
}
```

We are now ready to implement the comparison method. We simply call the `compare` method of the `base` `Comparator` and reverse its sign. This effectively reverses the relation between values.

```
public int compare(E a, E b)
```

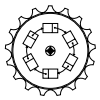
```
// pre: a, b non-null, and b is of type of a
// post: returns value <, ==, > 0 if a <, ==, > b
{
    return -base.compare(a,b);
}
```

Note that formerly equal values are still equal under the `ReverseComparator` transformation.

We now turn to an implementaton of an `OrderedStructure` that makes exclusive use of `Comparators` to keep its elements in order.

11.2.5 The Ordered List

Arbitrarily inserting an element into a list is difficult, since it requires moving to the middle of the list to perform the addition. The lists we have developed are biased toward addition and removal of values from their ends. Thus, we choose to use the underlying structure of a `SinglyLinkedList` to provide the basis for our `OrderedList` class. As promised, we will also support orderings through the use of `Comparators`. First, we declare the class as an implementation of the `OrderedStructure` interface:



OrderedList

```
public class OrderedList<E extends Comparable<E>>
    extends AbstractStructure<E> implements OrderedStructure<E>
```

The instance variables describe a singly linked list as well as a `Comparator` to determine the ordering. The constructors set up the structure by initializing the protected variables using the `clear` method:

```
protected Node<E> data; // smallest value
protected int count;        // number of values in list
protected Comparator<? super E> ordering;   // the comparison function

public OrderedList()
// post: constructs an empty ordered list
{
    this(new NaturalComparator<E>());
}

public OrderedList(Comparator<? super E> ordering)
// post: constructs an empty ordered list ordered by ordering
{
    this.ordering = ordering;
    clear();
}

public void clear()
// post: the ordered list is empty
{
    data = null;
```

```
        count = 0;
    }
```

Again, the advantage of this technique is that changes to the initialization of the underlying data structure can be made in one place within the code.

By default, the `OrderedList` keeps its elements in the order determined by the `compareTo` method. The `NaturalOrder` comparator does precisely that. If an alternative ordering is desired, the constructor for the `OrderedList` can be given a `Comparator` that can be used to guide the ordering of the elements in the list.

To warm up to the methods that we will soon have to write, let's consider implementation of the `contains` method. It uses the finger technique from our work with `SinglyLinkedLists`:

```
    public boolean contains(E value)
    // pre: value is a non-null comparable object
    // post: returns true iff contains value
    {
        Node<E> finger = data; // target
        // search down list until we fall off or find bigger value
        while ((finger != null) &&
               ordering.compare(finger.value(),value) < 0)
        {
            finger = finger.next();
        }
        return finger != null && value.equals(finger.value());
    }
```

This code is very similar to the *linear search* `contains` method of the `SinglyLinkedList` class. However, because the list is always kept in order, it can stop searching if it finds an element that is larger than the desired element. This leads to a behavior that is linear in the size of the list, but in the case when a value is not in the list, it terminates—on average—halfway down the list. For programs that make heavy use of looking up values in the structure, this can yield dramatic improvements in speed.

Note the use of the `compare` method in the `ordering Comparator`. No matter what order the elements have been inserted, the `ordering` is responsible for keeping them in the order specified.

Exercise 11.1 *What would be necessary to allow the user of an* `OrderedStructure` *to provide an alternative ordering* during *the lifetime of a class? This method might be called* `sortBy` *and would take a* `Comparator` *as its sole parameter.*

Now, let us consider the addition of an element to the `OrderedList`. Since the elements of the `OrderedList` are kept in order constantly, we must be careful to preserve that ordering after we have inserted the value. Here is the code:

```
public void add(E value)
// pre: value is non-null
// post: value is added to the list, leaving it in order
{
    Node<E> previous = null; // element to adjust
    Node<E> finger = data;   // target element
    // search for the correct location
    while ((finger != null) &&
           ordering.compare(finger.value(),value) < 0)
    {
        previous = finger;
        finger = finger.next();
    }
    // spot is found, insert
    if (previous == null) // check for insert at top
    {
        data = new Node<E>(value,data);
    } else {
        previous.setNext(
           new Node<E>(value,previous.next()));
    }
    count++;
}
```

Here we use the finger technique with an additional `previous` reference to help the insertion of the new element. The first loop takes, on average, linear time to find a position where the value can be inserted. After the loop, the `previous` reference refers to the element that will refer to the new element, or is `null`, if the element should be inserted at the head of the list. Notice that we use the `Node` methods to ensure that we reuse code that works and to make sure that the elements are constructed with reasonable values in their fields.

One of the most common mistakes made is to forget to do important bookkeeping. Remember to increment `count` when inserting a value and to decrement `count` when removing a value. When designing and implementing structures, it is sometimes useful to look at each method from the point of view of each of the bookkeeping variables that you maintain.

Principle 18 *Consider your code from different points of view.*

Removing a value from the `OrderedList` first performs a check to see if the value is included, and then, if it is, removes it. When removing the value, we return a reference to the value found in the list.

```
public E remove(E value)
// pre: value is non-null
// post: an instance of value is removed, if in list
{
    Node<E> previous = null; // element to adjust
    Node<E> finger = data;   // target element
```

```
        // search for value or fall off list
        while ((finger != null) &&
               ordering.compare(finger.value(),value) < 0)
        {
            previous = finger;
            finger = finger.next();
        }
        // did we find it?
        if ((finger != null) && value.equals(finger.value())) {
            // yes, remove it
            if (previous == null)  // at top?
            {
                data = finger.next();
            } else {
                previous.setNext(finger.next());
            }
            count--;
            // return value
            return finger.value();
        }
        // return nonvalue
        return null;
    }
```

Again, because the `SinglyLinkedListIterator` accepts a `SinglyLinkedListElement` as its parameter, the implementation of the `OrderedList`'s `iterator` method is particularly simple:

```
    public Iterator<E> iterator()
    {
        return new SinglyLinkedListIterator<E>(data);
    }
```

The remaining size-related procedures follow those found in the implementation of `SinglyLinkedLists`.

11.2.6 Example: The Modified Parking Lot

Renter—ambiguous noun: (1) one who rents from others, (2) one who rents to others.

In Section 9.2 we implemented a system for maintaining rental contracts for a small parking lot. With our knowledge of ordered structures, we now return to that example to incorporate a new feature—an alphabetical listing of contracts.

As customers rent spaces from the parking office, contracts are added to a generic list of associations between renter names and lot assignments. We now change that structure to reflect a better means of keeping track of this information—an ordered list of comparable associations. This structure is declared as an `OrderedStructure` but is assigned an instance of an `OrderedList`:

```
OrderedStructure<ComparableAssociation<String,Space>> rented =
   new OrderedList<ComparableAssociation<String,Space>>(); // rented spaces
```

When a renter fills out a contract, the name of the renter and the parking space information are bound together into a single `ComparableAssociation`:

```
String renter = r.readString();
// link renter with space description
rented.add(new ComparableAssociation<String,Space>(renter,location));
System.out.println("Space "+location.number+" rented.");
```

ParkingLot2

Notice that the renter's name is placed into a `String`. Since `Strings` support the `compareTo` method, they implement the `Comparable` interface. The default ordering is used because the call to the constructor did not provide a specific ordering.

At this point, the `rented` structure has all contracts sorted by name. To print these contracts out, we accept a new command, `contracts`:

```
if (command.equals("contracts"))
{   // print out contracts in alphabetical order
    for (ComparableAssociation<String,Space> contract : rented) {
        // extract contract from iterator
        // extract person from contract
        String person = contract.getKey();
        // extract parking slot description from contract
        Space slot = contract.getValue();
        // print it out
        System.out.println(person+" is renting "+slot.number);

    }
}
```

An iterator for the `OrderedStructure` is used to retrieve each of the `ComparableAssociations`, from which we extract and print the renters' names in alphabetical order. Here is an example run of the program (the user's input is indented):

```
   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.
```

Note that, for each of the requests for contracts, the contracts are listed in alphabetical order. This example is particularly interesting since it demonstrates that use of an ordered structure eliminates the need to sort the contracts before they are printed each time and that the interface meshes well with software that doesn't use ordered structures. While running an orderly parking lot can be a tricky business, it is considerably simplified if you understand the subtleties of ordered structures.

Exercise 11.2 *Implement an alternative* `Comparator` *that compares two parking spaces, based on slot numbers. Demonstrate that a single line will change the order of the records in the* `ParkingLot2` *program.*

11.3 Conclusions

Computers spend a considerable amount of time maintaining ordered data structures. In Java we described an ordering of data values using the comparison operator, `compareTo`, or a `Comparator`. Objects that fail to have an operator such as `compareTo` cannot be totally ordered in a predetermined manner. Still, a `Comparator` might be constructed to suggest an ordering between otherwise incomparable values. Java enforces the development of an ordering using the `Comparable` interface—an interface that simply requires the implementation of the `compareTo` method.

Once data values may be compared and put in order, it is natural to design a data structure that keeps its values in order. Disk directories, dictionaries, filing cabinets, and zip-code ordered mailing lists are all obvious examples of abstract structures whose utility depends directly on their ability to efficiently maintain a consistently ordered state. Here we extend various unordered structures in a way that allows them to maintain the natural ordering of the underlying data.

Self Check Problems

Solutions to these problems begin on page 448.

11.1 What is the primary feature of an `OrderedStructure`?

11.2 How does the user communicate the order in which elements are to be stored in an `OrderedStructure`?

11.3 What is the difference between a `compareTo` method and a comparator with a `compare` method?

11.4 Are we likely to find two objects that are equal (using their `equals` method) to be close together in an `OrderedStructure`?

11.5 Is an `OrderedVector` a `Vector`?

11.6 Is it reasonable to have an `OrderedStack`, a class that implements both the `Stack` and `OrderedStructure` interfaces?

11.7 People queue up to enter a movie theater. They are stored in an `OrderedStructure`. How would you go about comparing two people?

11.8 Sally and Harry implement two different `compareTo` methods for a class they are working on together. Sally's `compareTo` method returns -1, 0, or $+1$, depending on the relationship between two objects. Harry's `compareTo` method returns -6, 0, or $+3$. Which method is suitable?

11.9 Sally and Harry are working on an implementation of `Coin`. Sally declares the sole parameter to her `compareTo` method as type `Object`. Harry knows the `compareTo` method will always be called on objects of type `Coin` and declares the parameter to be of that type. Which method is suitable for storing `Coin` objects in a `OrderedVector`?

Problems

Solutions to the odd-numbered problems begin on page 475.

11.1 Describe the contents of an `OrderedVector` after each of the following values has been added: 1, 9, 0, -1, and 3.

11.2 Describe the contents of an `OrderedList` after each of the following values has been added: 1, 9, 0, -1, and 3.

11.3 Suppose duplicate values are added to an `OrderedVector`. Where is the oldest value found (with respect to the newest)?

11.4 Suppose duplicate values are added to an `OrderedList`. Where is the oldest value found (with respect to the newest)?

11.5 Show that the expected insertion time of an element into an `OrderedList` is $O(n)$ in the worst case.

11.6 Show that the expected insertion time of an element into an `OrderedVector` is $O(n)$.

11.7 Under what conditions would you use an `OrderedVector` over an `OrderedList`?

11.8 At what point does the Java environment complain about your passing a non-`Comparable` value to an `OrderedVector`?

11.9 Write the `compareTo` method for the `String` class.

11.10 Write the `compareTo` method for a class that is to be ordered by a field, `key`, which is a `double`. Be careful: The result of `compareTo` must be an `int`.

11.11 Write the `compareTo` method for a class describing a person whose name is stored as two `Strings`: `first` and `last`. A person is "less than" another if they appear before the other in a list alphabetized by last name and then first name (as is typical).

11.12 Previous editions of the `structures` package opted for the use of a `lessThan` method instead of a `compareTo` method. The `lessThan` method would return `true` exactly when one value was `lessThan` another. Are these approaches the same, or is one more versatile?

11.13 Suppose we consider the use of an `OrderedStructure` `get` method that takes an integer i. This method returns the ith element of the `OrderedStructure`. What are the best- and worst-case running times for this method on `OrderedVector` and `OrderedList`?

11.14 Your department is interested in keeping track of information about majors. Design a data structure that will maintain useful information for your department. The roster of majors, of course, should be ordered by last name (and then by first, if there are multiple students with the same last name).

11.4 Laboratory: Computing the "Best Of"

Objective. To efficiently select the largest k values of n.

Discussion. One method to select the largest k values in a sequence of n is to sort the n values and to look only at the first k. (In Chapter 13, we will learn of another technique: insert each of the n values into a max-heap and extract the first k values.) Such techniques have two important drawbacks:

- The data structure that keeps track of the values must be able to hold $n >> k$ values. This may not be possible if, for example, there are more data than may be held easily in memory.

- The process requires $O(n \log n)$ time. It should be possible to accomplish this in $O(n)$ time.

One way to reduce these overheads is to keep track of, at all times, the best k values found. As the n values are passed through the structure, they are only remembered if they are potentially one of the largest k values.

Procedure. In this lab we will implement a `BestOf OrderedStructure`. The constructor for your `BestOf` structure should take a value k, which is an upper bound on the number of values that will be remembered. The default constructor should remember the top 10.

An `add` method takes an `Object` and adds the element (if reasonable) in the correct location. The `get(i)` method should return the ith largest value encountered so far. The `size` method should return the number of values currently stored in the structure. This value should be between 0 and k, inclusive. The `iterator` method should return an `Iterator` over all the values. The `clear` method should remove all values from the structure.

Here are the steps that are necessary to construct and test this data structure:

1. Consider the underlying structure carefully. Because the main considerations of this structure are size and speed, it would be most efficient to implement this using a fixed-size array. We will assume that here.

2. Implement the `add` method. This method should take the value and, like a pass of insertion sort, it should find the correct location for the new value. If the array is full and the value is no greater than any of the values, nothing changes. Otherwise, the value is inserted in the correct location, possibly dropping a smaller value.

3. Implement the `get(i)`, `size`, and `clear` methods.

4. Implement the `iterator` method. A special `AbstractIterator` need not be constructed; instead, values can be added to a linear structure and the result of *that* structure's `iterator` method is returned.

To test your structure, you can generate a sequence of integers between 0 and $n-1$. By the end, only values $n-k \ldots n-1$ should be remembered.

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

1. What is the (big-O) complexity of one call to the `add` method? What is the complexity of making n calls to `add`?
2. What are the advantages and disadvantages of keeping the array sorted at all times? Would it be more efficient to, say, only keep the smallest value in the first slot of the array?
3. Suppose that $f(n)$ is defined to be $n/2$ if n is even, and $3n+1$ if n is odd. It is known that for small values of n (less than 10^{40}) the sequence of values generated by repeated application of f starting at n eventually reaches 1. Consider all the sequences generated from $n < 10,000$. What are the maximum values encountered?
4. The `BestOf` structure can be made more general by providing a third constructor that takes k and a `Comparator`. The comparisons in the `BestOf` class can now be recast as calls to the `compare` method of a `Comparator`. When a `Comparator` is not provided, an instance of the `structure` package's `NaturalComparator` is used, instead. Such an implementation allows the `BestOf` class to order non-`Comparable` values, and `Comparable` values in alternative orders.

Notes: