

```
        return first == null;
    }
}
```

So far it is straightforward. In an empty queue, both **first** and **last** are null. To check whether a list is empty, we only have to check one of them.

add is a little more complicated because we have to deal with several special cases.

```
public void add (Object obj) {
    Node node = new Node (obj, null);
    if (last != null) {
        last.next = node;
    }
    last = node;
    if (first == null) {
        first = last;
    }
}
```

The first condition checks to make sure that **last** refers to a node; if it does then we have to make it refer to the new node.

The second condition deals with the special case where the list was initially empty. In this case both **first** and **last** refer to the new node.

remove also deals with several special cases.

```
public Object remove () {
    Node result = first;
    if (first != null) {
        first = first.next;
    }
    if (first == null) {
        last = null;
    }
    return result;
}
```

The first condition checks whether there were any nodes in the queue. If so, we have to copy the **next** node into **first**. The second condition deals with the special case that the list is now empty, in which case we have to make **last** null.

As an exercise, draw diagrams showing these operations in both the normal case and in the special cases, and convince yourself that they are correct.

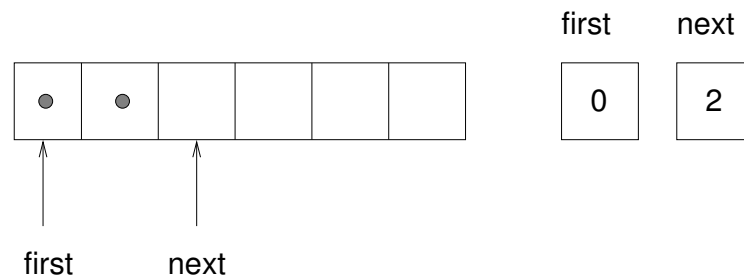
Clearly, this implementation is more complicated than the veneer implementation, and it is more difficult to demonstrate that it is correct. The advantage is that we have achieved the goal: **add** and **remove** are constant time operations.

16.4 Circular buffer

Another common implementation of a queue is a **circular buffer**. “Buffer” is a general name for a temporary storage location, although it often refers to an array, as it does in this case. What it means to say a buffer is “circular” should become clear in a minute.

The implementation of a circular buffer is similar to the array implementation of a stack in Section 15.12. The queue items are stored in an array, and we use indices to keep track of where we are in the array. In the stack implementation, there was a single index that pointed to the next available space. In the queue implementation, there are two indices: **first** points to the space in the array that contains the first customer in line and **next** points to the next available space.

The following figure shows a queue with two items (represented by dots).



There are two ways to think of the variables **first** and **last**. Literally, they are integers, and their values are shown in boxes on the right. Abstractly, though, they are indices of the array, and so they are often drawn as arrows pointing to locations in the array. The arrow representation is convenient, but you should remember that the indices are not references; they are just integers.

Here is an incomplete array implementation of a queue:

```
public class Queue {
    public Object[] array;
    public int first, next;

    public Queue () {
        array = new Object[128];
        first = 0;
        next = 0;
    }

    public boolean isEmpty () {
        return first == next;
    }
}
```

The instance variables and the constructor are straightforward, although again we have the problem that we have to choose an arbitrary size for the array.

Later we will solve that problem, as we did with the stack, by resizing the array if it gets full.

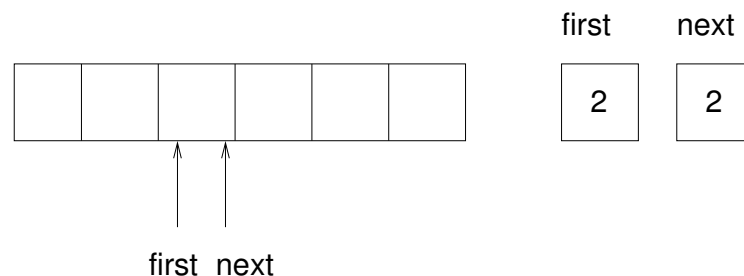
The implementation of `isEmpty` is a little surprising. You might have thought that `first == 0` would indicate an empty queue, but that neglects the fact that the head of the queue is not necessarily at the beginning of the array. Instead, we know that the queue is empty if `head` equals `next`, in which case there are no items left. Once we see the implementation of `add` and `remove`, that condition will make more sense.

```
public void add (Object item) {
    array[next] = item;
    next++;
}

public Object remove () {
    Object result = array[first];
    first++;
    return result;
}
```

`add` looks very much like `push` in Section 15.12; it puts the new item in the next available space and then increments the index.

`remove` is similar. It takes the first item from the queue and then increments `first` so it refers to the new head of the queue. The following figure shows what the queue looks like after both items have been removed.



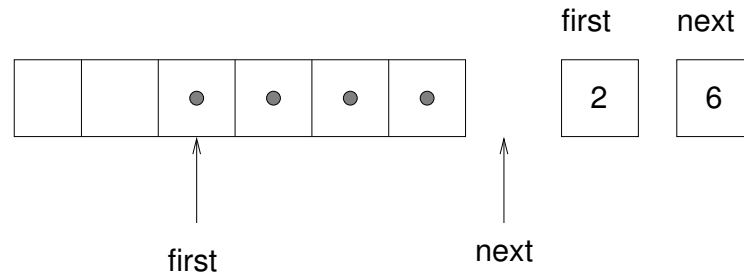
It is always true that `next` points to an available space. If `first` catches up with `next` and points to the same space, then `first` is referring to an “empty” location, and the queue is empty. I put “empty” in quotation marks because it is possible that the location that `first` points to actually contains a value (we do nothing to ensure that empty locations contain `null`); on the other hand, since we know the queue is empty, we will never read this location, so we can think of it, abstractly, as empty.

Exercise 16.1 Modify `remove` so that it returns `null` if the queue is empty.

The next problem with this implementation is that eventually it will run out of space. When we add an item we increment `next` and when we remove an item

we increment **first**, but we never decrement either. What happens when we get to the end of the array?

The following figure shows the queue after we add four more items:



The array is now full. There is no “next available space,” so there is nowhere for **next** to point. One possibility is that we could resize the array, as we did with the stack implementation. But in that case the array would keep getting bigger regardless of how many items were actually in queue. A better solution is to wrap around to the beginning of the array and reuse the spaces there. This “wrap around” is the reason this implementation is called a circular buffer.

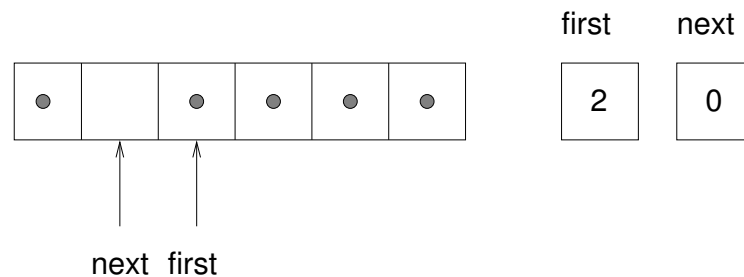
One way to wrap the index around is to add a special case whenever we increment an index:

```
next++;
if (next == array.length) next = 0;
```

A fancy alternative is to use the modulus operator:

```
next = (next + 1) % array.length;
```

Either way, we have one last problem to solve. How do we know if the queue is *really* full, meaning that we cannot add another item? The following figure shows what the queue looks like when it is “full.”



There is still one empty space in the array, but the queue is full because if we add another item, then we have to increment **next** such that **next == first**, and in that case it would appear that the queue was empty!

To avoid that, we sacrifice one space in the array. So how can we tell if the queue is full?

```
if ((next + 1) % array.length == first)
```

And what should we do if the array is full? In that case resizing the array is probably the only option.

Exercise 16.2 Write an implementation of a queue using a circular buffer that resizes itself when necessary.

16.5 Priority queue

The Priority Queue ADT has the same interface as the Queue ADT, but different semantics. The interface is:

constructor: Create a new, empty queue.

add: Add a new item to the queue.

remove: Remove and return an item from the queue. The item that is returned is the one with the highest priority.

isEmpty: Check whether the queue is empty.

The semantic difference is that the item that is removed from the queue is not necessarily the first one that was added. Rather, it is whatever item in the queue has the highest priority. What the priorities are, and how they compare to each other, are not specified by the Priority Queue implementation. It depends on what the items are that are in the queue.

For example, if the items in the queue have names, we might choose them in alphabetical order. If they are bowling scores, we might choose from highest to lowest, but if they are golf scores, we would go from lowest to highest.

So we face a new problem. We would like an implementation of Priority Queue that is generic—it should work with any kind of object—but at the same time the code that implements Priority Queue needs to have the ability to compare the objects it contains.

We have seen a way to implement generic data structures using `Objects`, but that does not solve this problem, because there is no way to compare `Objects` unless we know what type they are.

The answer lies in a Java feature called a **metaclass**.

16.6 Metaclass

A metaclass is a set of classes that provide a common set of methods. The metaclass definition specifies the requirements a class must satisfy to be a member of the set.

Often metaclasses have names that end in “able” to indicate the fundamental capability the metaclass requires. For example, any class that provides a method named `draw` can be a member of the metaclass named `Drawable`. Any class that contains a method named `start` can be a member of the metaclass `Runnable`.

Java provides a built-in metaclass that we can use in an implementation of a Priority Queue. It is called `Comparable`, and it means what it says. Any class that belongs to the `Comparable` metaclass has to provide a method named `compareTo` that compares two objects and returns a value indicating whether one is larger or smaller than the other, or whether they are the same.

Many of the built-in Java classes are members of the `Comparable` metaclass, including numeric wrapper classes like `Integer` and `Double`.

In the next section I will show how to write an ADT that manipulates a metaclass. Then we will see how to write a new class that belongs to an existing metaclass. In the next chapter we will see how to define a new metaclass.

16.7 Array implementation of Priority Queue

In the implementation of the Priority Queue, every time we specify the type of the items in the queue, we specify the metaclass `Comparable`. For example, the instance variables are an array of `Comparables` and an integer:

```
public class PriorityQueue {
    private Comparable[] array;
    private int index;
}
```

As usual, `index` is the index of the next available location in the array. The instance variables are declared `private` so that other classes cannot have direct access to them.

The constructor and `isEmpty` are similar to what we have seen before. The initial size of the array is arbitrary.

```
public PriorityQueue () {
    array = new Comparable [16];
    index = 0;
}

public boolean isEmpty () {
    return index == 0;
}
```

`add` is similar to `push`:

```
public void add (Comparable item) {
    if (index == array.length) {
        resize ();
    }
}
```

```

        array[index] = item;
        index++;
    }

```

The only substantial method in the class is `remove`, which has to traverse the array to find and remove the largest item:

```

    public Comparable remove () {
        if (index == 0) return null;

        int maxIndex = 0;

        // find the index of the item with the highest priority
        for (int i=1; i<index; i++) {
            if (array[i].compareTo (array[maxIndex]) > 0) {
                maxIndex = i;
            }
        }
        Comparable result = array[maxIndex];

        // move the last item into the empty slot
        index--;
        array[maxIndex] = array[index];
        return result;
    }

```

As we traverse the array, `maxIndex` keeps track of the index of the largest element we have seen so far. What it means to be the “largest” is determined by `compareTo`. In this case the `compareTo` method is provided by the `Integer` class, and it does what we expect—larger (more positive) numbers win.

16.8 A Priority Queue client

The implementation of Priority Queue is written entirely in terms of `Comparable` objects, but there is no such thing as a `Comparable` object! Go ahead, try to create one:

```

    Comparable comp = new Comparable ();           // ERROR

```

You’ll get a compile-time message that says something like “`java.lang.Comparable` is an interface. It can’t be instantiated.” In Java, metaclasses are called **interfaces**. I have avoided this word so far because it also means several other things, but now you have to know.

Why can’t metaclasses be instantiated? Because a metaclass only specifies requirements (you must have a `compareTo` method); it does not provide an implementation.

To create a `Comparable` object, you have to create one of the objects that belongs to the `Comparable` set, like `Integer`. Then you can use that object anywhere a `Comparable` is called for.

```
PriorityQueue pq = new PriorityQueue ();
Integer item = new Integer (17);
pq.add (item);
```

This code creates a new, empty Priority Queue and a new Integer object. Then it adds the Integer into the queue. `add` is expecting a `Comparable` as a parameter, so it is perfectly happy to take an `Integer`. If we try to pass a `Rectangle`, which does not belong to `Comparable`, we get a compile-time message like, “Incompatible type for method. Explicit cast needed to convert `java.awt.Rectangle` to `java.lang.Comparable`.”

That’s the compiler telling us that if we want to make that conversion, we have to do it explicitly. We might try to do what it says:

```
Rectangle rect = new Rectangle ();
pq.add ((Comparable) rect);
```

But in that case we get a run-time error, a `ClassCastException`. When the `Rectangle` tries to pass as a `Comparable`, the run-time system checks whether it satisfies the requirements, and rejects it. So that’s what we get for following the compiler’s advice.

To get items out of the queue, we have to reverse the process:

```
while (!pq.isEmpty ()) {
    item = (Integer) pq.remove ();
    System.out.println (item);
}
```

This loop removes all the items from the queue and prints them. It assumes that the items in the queue are `Integers`. If they were not, we would get a `ClassCastException`.

16.9 The Golfer class

Finally, let’s look at how we can make a new class that belongs to `Comparable`. As an example of something with an unusual definition of “highest” priority, we’ll use golfers:

```
public class Golfer implements Comparable {
    String name;
    int score;

    public Golfer (String name, int score) {
        this.name = name;
        this.score = score;
    }
}
```

The class definition and the constructor are pretty much the same as always; the difference is that we have to declare that `Golfer` implements `Comparable`. In

this case the keyword `implements` means that `Golfer` implements the interface specified by `Comparable`.

If we try to compile `Golfer.java` at this point, we get something like “class `Golfer` must be declared abstract. It does not define `int compareTo(java.lang.Object)` from interface `java.lang.Comparable`.” In other words, to be a `Comparable`, `Golfer` has to provide a method named `compareTo`. So let’s write one:

```
public int compareTo (Object obj) {
    Golfer that = (Golfer) obj;

    int a = this.score;
    int b = that.score;

    // for golfers, low is good!
    if (a<b) return 1;
    if (a>b) return -1;
    return 0;
}
```

Two things here are a little surprising. First, the parameter is an `Object`. That’s because in general the caller doesn’t know what type the objects are that are being compared. For example, in `PriorityQueue.java` when we invoke `compareTo`, we pass a `Comparable` as a parameter. We don’t have to know whether it is an `Integer` or a `Golfer` or whatever.

Inside `compareTo` we have to convert the parameter from an `Object` to a `Golfer`. As usual, there is a risk when we do this kind of cast: if we cast to the wrong type we get an exception.

Finally, we can create some golfers:

```
Golfer tiger = new Golfer ("Tiger Woods", 61);
Golfer phil = new Golfer ("Phil Mickelson", 72);
Golfer hal = new Golfer ("Hal Sutton", 69);
```

And put them in the queue:

```
pq.add (tiger);
pq.add (phil);
pq.add (hal);
```

When we pull them out:

```
while (!pq.isEmpty ()) {
    golfer = (Golfer) pq.remove ();
    System.out.println (golfer);
}
```

They appear in descending order (for golfers):

```
Tiger Woods    61
Hal Sutton     69
Phil Mickelson 72
```

When we switched from `Integers` to `Golfers`, we didn't have to make any changes in `PriorityQueue.java` at all. So we succeeded in maintaining a barrier between `PriorityQueue` and the classes that use it, allowing us to reuse the code without modification. Furthermore, we were able to give the client code control over the definition of `compareTo`, making this implementation of `PriorityQueue` more versatile.

16.10 Glossary

queue: An ordered set of objects waiting for a service of some kind.

queueing discipline: The rules that determine which member of a queue is removed next.

FIFO: "first in, first out," a queueing discipline in which the first member to arrive is the first to be removed.

priority queue: A queueing discipline in which each member has a priority determined by external factors. The member with the highest priority is the first to be removed.

Priority Queue: An ADT that defines the operations one might perform on a priority queue.

veneer: A class definition that implements an ADT with method definitions that are invocations of other methods, sometimes with simple transformations. The veneer does no significant work, but it improves or standardizes the interface seen by the client.

performance hazard: A danger associated with a veneer that some of the methods might be implemented inefficiently in a way that is not apparent to the client.

constant time: An operation whose run time does not depend on the size of the data structure.

linear time: An operation whose run time is a linear function of the size of the data structure.

linked queue: An implementation of a queue using a linked list and references to the first and last nodes.

circular buffer: An implementation of a queue using an array and indices of the first element and the next available space.

metaclass: A set of classes. The metaclass specification lists the requirements a class must satisfy to be included in the set.

interface: The Java word for a metaclass. Not to be confused with the more broad meaning of the word interface.

16.11 Exercises

Exercise 16.3 This question is based on Exercise 9.3.

Write a `compareTo` method for the `Rational` class that would allow `Rational` to implement `Comparable`. Hint: don't forget that the parameter is an `Object`.

Exercise 16.4 Write a class definition for `SortedList`, which extends `LinkedList`. A `SortedList` is similar to a `LinkedList`; the difference is that the elements have to be `Comparable`, and the list is sorted in decreasing order.

Write an object method for `SortedList` called `add` that takes a `Comparable` as a parameter and that adds the new object into the list, at the appropriate place so that the list stays sorted.

If you want, you can write a helper method in the `Node` class.

Exercise 16.5 Write an object method for the `LinkedList` class named `maximum` that can be invoked on a `LinkedList` object, and that returns the largest cargo object in the list, or null if the list is empty.

You can assume that every cargo element belongs to a class that belongs to the meta-class `Comparable`, and that any two elements can be compared to each other.

Exercise 16.6 Write an implementation of a Priority Queue using a linked list. There are three ways you might proceed:

- A Priority Queue might contain a `LinkedList` object as an instance variable.
- A Priority Queue might contain a reference to the first `Node` object in a linked list.
- A Priority Queue might extend (inherit from) the existing `LinkedList` class.

Think about the pros and cons of each and choose one. Also, you can choose whether to keep the list sorted (slow add, fast remove) or unsorted (slow remove, fast add).

Exercise 16.7 An event queue is a data structure that keeps track of a set of events, where each event has a time associated with it. The ADT is:

constructor: make a new, empty event queue

add: put a new event in the queue. The parameters are the event, which is an `Object`, and the time the event occurs, which is a `Date` object. The event `Object` must not be null.

nextTime: return the `Date` at which the next event occurs, where the "next" event is the one in the queue with the earliest time. Do not remove the event from the queue. Return null if the queue is empty.

nextEvent: return the next event (an `Object`) from the queue and remove it from the queue. Return null if the queue is empty.

The `Date` class is defined in `java.util` and it implements `Comparable`. According to the documentation, its `compareTo` method returns "the value 0 if the argument `Date` is equal to this `Date`; a value less than 0 if this `Date` is before the `Date` argument; and a value greater than 0 if this `Date` is after the `Date` argument."

Write an implementation of an event queue using the `PriorityQueue` ADT. You should not make any assumptions about how the `PriorityQueue` is implemented.

HINT: create a class named `Event` that contains a `Date` and an event `Object`, and that implements `Comparable` appropriately.

Chapter 17

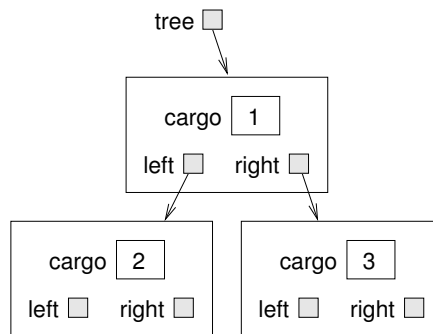
Trees

17.1 A tree node

Like lists, trees are made up of nodes. A common kind of tree is a **binary tree**, in which each node contains a reference to two other nodes (possibly null). The class definition looks like this:

```
public class Tree {  
    Object cargo;  
    Tree left, right;  
}
```

Like list nodes, tree nodes contain cargo: in this case a generic `Object`. The other instance variables are named `left` and `right`, in accordance with a standard way to represent trees graphically:



The top of the tree (the node referred to by `tree`) is called the **root**. In keeping with the tree metaphor, the other nodes are called branches and the nodes at the tips with null references are called **leaves**. It may seem odd that we draw the picture with the root at the top and the leaves at the bottom, but that is not the strangest thing.

To make things worse, computer scientists mix in yet another metaphor: the family tree. The top node is sometimes called a **parent** and the nodes it refers to are its **children**. Nodes with the same parent are called **siblings**, and so on.

Finally, there is also a geometric vocabulary for talking about trees. I already mentioned left and right, but there is also “up” (toward the parent/root) and down (toward the children/leaves). Also, all the nodes that are the same distance from the root comprise a **level** of the tree.

I don’t know why we need three metaphors for talking about trees, but there it is.

17.2 Building trees

The process of assembling tree nodes is similar to the process of assembling lists. We have a constructor for tree nodes that initializes the instance variables.

```
public Tree (Object cargo, Tree left, Tree right) {
    this.cargo = cargo;
    this.left = left;
    this.right = right;
}
```

We allocate the child nodes first:

```
Tree left = new Tree (new Integer(2), null, null);
Tree right = new Tree (new Integer(3), null, null);
```

We can create the parent node and link it to the children at the same time:

```
Tree tree = new Tree (new Integer(1), left, right);
```

This code produces the state shown in the previous figure.

17.3 Traversing trees

The most natural way to traverse a tree is recursively. For example, to add up all the integers in a tree, we could write this class method:

```
public static int total (Tree tree) {
    if (tree == null) return 0;
    Integer cargo = (Integer) tree.cargo;
    return cargo.intValue() + total (tree.left) + total (tree.right);
}
```

This is a class method because we would like to use `null` to represent the empty tree, and make the empty tree the base case of the recursion. If the tree is empty, the method returns 0. Otherwise it makes two recursive calls to find the total value of its two children. Finally, it adds in its own cargo and returns the total.

Although this method works, there is some difficulty fitting it into an object-oriented design. It should not appear in the `Tree` class because it requires the

cargo to be `Integer` objects. If we make that assumption in `Tree.java` then we lose the advantages of a generic data structure.

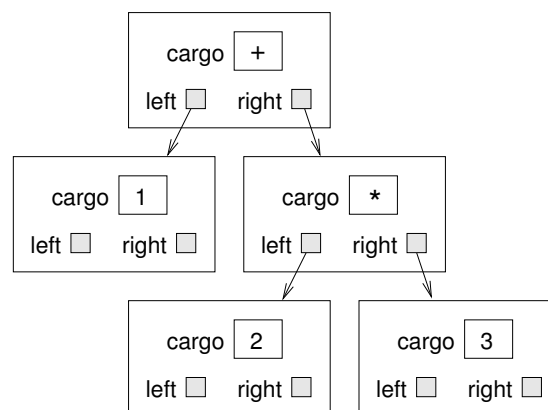
On the other hand, this code accesses the instance variables of the `Tree` nodes, so it “knows” more than it should about the implementation of the tree. If we change that implementation later this code will break.

Later in this chapter we will develop ways to solve this problem, allowing client code to traverse trees containing any kinds of objects without breaking the abstraction barrier between the client code and the implementation. Before we get there, let’s look at an application of trees.

17.4 Expression trees

A tree is a natural way to represent the structure of a mathematical expression. Unlike other notations, it can represent the computation unambiguously. For example, the infix expression $1 + 2 * 3$ is ambiguous unless we know that the multiplication happens before the addition.

The following figure represents the same computation:



The nodes can be operands like 1 and 2 or operators like + and *. Operands are leaf nodes; operator nodes contain references to their operands (all of these operators are **binary**, meaning they have exactly two operands).

Looking at this figure, there is no question what the order of operations is: the multiplication happens first in order to compute the first operand of the addition.

Expression trees like this have many uses. The example we are going to look at is translation from one format (postfix) to another (infix). Similar trees are used inside compilers to parse, optimize and translate programs.

17.5 Traversal

I already pointed out that recursion provides a natural way to traverse a tree. We can print the contents of an expression tree like this:

```
public static void print (Tree tree) {
    if (tree == null) return;
    System.out.print (tree + " ");
    print (tree.left);
    print (tree.right);
}
```

In other words, to print a tree, first print the contents of the root, then print the entire left subtree, then print the entire right subtree. This way of traversing a tree is called a **preorder**, because the contents of the root appear before the contents of the children.

For the example expression the output is `+ 1 * 2 3`. This is different from both postfix and infix; it is a new notation called **prefix**, in which the operators appear before their operands.

You might suspect that if we traverse the tree in a different order we get the expression in a different notation. For example, if we print the subtrees first, and then the root node:

```
public static void printPostorder (Tree tree) {
    if (tree == null) return;
    printPostorder (tree.left);
    printPostorder (tree.right);
    System.out.print (tree + " ");
}
```

We get the expression in postfix `(1 2 3 * +)`! As the name of the method implies, this order of traversal is called **postorder**. Finally, to traverse a tree **inorder**, we print the left tree, then the root, then the right tree:

```
public static void printInorder (Tree tree) {
    if (tree == null) return;
    printInorder (tree.left);
    System.out.print (tree + " ");
    printInorder (tree.right);
}
```

The result is `1 + 2 * 3`, which is the expression in infix.

To be fair, I have to point out that I omitted an important complication. Sometimes when we write an expression in infix we have to use parentheses to preserve the order of operations. So an inorder traversal is not quite sufficient to generate an infix expression.

Nevertheless, with a few improvements, the expression tree and the three recursive traversals provide a general way to translate expressions from one format to another.

17.6 Encapsulation

As I mentioned before, there is a problem with the way we have been traversing trees: it breaks down the barrier between the client code (the application that uses the tree) and the provider code (the `Tree` implementation). Ideally, tree code should be general; it shouldn't know anything about expression trees. And the code that generates and traverses the expression tree shouldn't know about the implementation of the trees. This design criterion is called **object encapsulation** to distinguish it from the encapsulation we saw in Section 6.6, which we might call **method encapsulation**.

In the current version, the `Tree` code knows too much about the client. Instead, the `Tree` class should provide the general capability of traversing a tree in various ways. As it traverses, it should perform operations on each node that are specified by the client.

To facilitate this separation of interests, we will create a new metaclass, called `Visitable`. The items stored in a tree will be required to be visitable, which means that they define a method named `visit` that does whatever the client wants done to each node. That way the `Tree` can perform the traversal and the client can perform the node operations.

Here are the steps we have to perform to wedge a metaclass between a client and a provider:

1. Define a metaclass that specifies the methods the provider code will need to invoke on its components.
2. Write the provider code in terms of the new metaclass, as opposed to generic `Objects`.
3. Define a class that belongs to the metaclass and that implements the required methods as appropriate for the client.
4. Write the client code to use the new class.

The next few sections demonstrate these steps.

17.7 Defining a metaclass

There are actually two ways to implement a metaclass in Java, as an **interface** or as an **abstract class**. The differences between them aren't important for now, so we'll start by defining an interface.

An interface definition looks a lot like a class definition, with two differences:

- The keyword `class` is replaced with `interface`, and
- The method definitions have no bodies.

An interface definition specifies the methods a class has to implement in order to be in the metaclass. The specification includes the name, parameter types, and return type of each method.

The definition of `Visitable` is

```
public interface Visitable {  
    public void visit ();  
}
```

That's it! The definition of `visit` looks like any other method definition, except that it has no body. This definition specifies that any class that implements `Visitable` has to have a method named `visit` that takes no parameters and that returns `void`.

Like other class definitions, interface definitions go in a file with the same name as the class (in this case `Visitable.java`).

17.8 Implementing a metaclass

If we are using an expression tree to generate infix, then “visiting” a node means printing its contents. Since the contents of an expression tree are tokens, we'll create a new class called `Token` that implements `Visitable`

```
public class Token implements Visitable {  
    String str;  
  
    public Token (String str) {  
        this.str = str;  
    }  
  
    public void visit () {  
        System.out.print (str + " ");  
    }  
}
```

When we compile this class definition (which is in a file named `Token.java`), the compiler checks whether the methods provided satisfy the requirements specified by the metaclass. If not, it will produce an error message. For example, if we misspell the name of the method that is supposed to be `visit`, we might get something like, “class `Token` must be declared abstract. It does not define void `visit()` from interface `Visitable`.” This is one of many error messages where the solution suggested by the compiler is wrong. When it says the class “must be declared abstract,” what it means is that you have to fix the class so that it implements the interface properly. Sometimes I think the people who write these messages should be beaten.

The next step is to modify the parser to put `Token` objects into the tree instead of `Strings`. Here is a small example:

```
String expr = "1 2 3 * +";
StringTokenizer st = new StringTokenizer (expr, " +-*/", true);
String token = st.nextToken();
Tree tree = new Tree (new Token (token), null, null));
```

This code takes the first token in the string and wraps it in a `Token` object, then puts the `Token` into a tree node. If the `Tree` requires the cargo to be `Visitable`, it will convert the `Token` to be a `Visitable` object. When we remove the `Visitable` from the tree, we will have to cast it back into a `Token`.

Exercise 17.1 Write a version of `printPreorder` called `visitPreorder` that traverses the tree and invokes `visit` on each node in preorder.

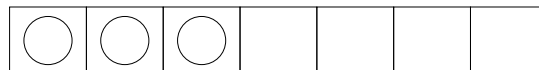
The flow of execution for methods like `visitPreorder` is unusual. The client invokes a method provided by the `Tree` implementation, and then the tree implementation invokes a method provided by the client. This pattern is called a **callback**; it is a good way to make provider code more general without breaking down the abstraction barrier.

17.9 The Vector class

The `Vector` is a built-in Java class in the `java.util` package. It is an implementation of an array of `Objects`, with the added feature that it can resize itself automatically, so we don't have to.

Before using the `Vector` class, you should understand a few concepts. Every `Vector` has a capacity, which is the amount of space that has been allocated to store values, and a size, which is the number of values that are actually in the vector.

The following figure is a simple diagram of a `Vector` that contains three elements, but it has a capacity of seven.



There are two sets of methods for accessing the elements of a vector. They provide different semantics and different error-checking capabilities, and they are easy to get confused.

The simpler accessor methods are `get` and `set`, which provide semantics similar to the array index operator `[]`. `get` takes an integer index and returns the element at the indicated position. `set` takes an index and an element, and stores the new element at the indicated position, replacing the existing element.

`get` and `set` do not change the size of the vector (number of elements). It is the responsibility of the client code to make sure that the vector has sufficient size before invoking `set` or `get`. The `size` method returns the number of elements

in the `Vector`. If you try to access an element that does not exist (in this case the elements with indices 3 through 6), you will get an `ArrayIndexOutOfBoundsException` exception.

The other set of methods includes several versions of `add` and `remove`. These methods change the size of the `Vector` and, if necessary, the capacity. One version of `add` takes an element as a parameter and adds it to the end of the `Vector`. This method is safe in the sense that it will not cause an exception.

Another version of `add` takes an index and an element and, like `set`, it puts the new element at the given position. The difference is that `add` doesn't replace the existing element; it increases the size of the `Vector` and shifts elements to the right to make room for the new one. Thus, the invocation `v.add(0, elt)` add the new element at the beginning of the `Vector`. Unfortunately, this method is neither safe nor efficient; it can cause an `ArrayIndexOutOfBoundsException` exception and, in most implementations, it is linear time (proportional to the size of the `Vector`).

Most of the time the client doesn't have to worry about capacity. Whenever the size of the `Vector` changes, the capacity is updated automatically. For performance reasons, some applications take control of this function, which is why there are additional methods for increasing and decreasing capacity.

Because the client code has no access to the implementation of a vector, it is not clear how we should traverse one. Of course, one possibility is to use a loop variable as an index into the vector:

```
for (int i=0; i<v.size(); i++) {  
    System.out.println (v.get(i));  
}
```

There's nothing wrong with that, but there is another way that serves to demonstrate the `Iterator` class. Vectors provide a method named `iterator` that returns an `Iterator` object that makes it possible to traverse the vector.

17.10 The Iterator class

`Iterator` is an interface in the `java.util` package. It specifies three methods:

hasNext: Does this iteration have more elements?

next: Return the next element, or throw an exception if there is none.

remove: Remove the most recent element from the data structure we are traversing.

The following example uses an iterator to traverse and print the elements of a vector.

```

    Iterator it = vector.iterator ();
    while (it.hasNext ()) {
        System.out.println (it.next ());
    }

```

Once the `Iterator` is created, it is a separate object from the original `Vector`. Subsequent changes in the `Vector` are not reflected in the `Iterator`. In fact, if you modify the `Vector` after creating an `Iterator`, the `Iterator` becomes invalid. If you access the `Iterator` again, it will cause a `ConcurrentModification` exception.

In a previous section we used the `Visitable` metaclass to allow a client to traverse a data structure without knowing the details of its implementation. Iterators provide another way to do the same thing. In the first case, the provider performs the iteration and invokes client code to “visit” each element. In the second case the provider gives the client an object that it can use to select elements one at a time (albeit in an order controlled by the provider).

Exercise 17.2 Write a class named `PreIterator` that implements the `Iterator` interface, and write a method named `preorderIterator` for the `Tree` class that returns a `PreIterator` that selects the elements of the `Tree` in preorder.

HINT: The easiest way to build an `Iterator` is to put elements into a `Vector` in the order you want and then invoke `iterator` on the `Vector`.

17.11 Glossary

binary tree: A tree in which each node refers to 0, 1, or 2 dependent nodes.

root: The top-most node in a tree, to which no other nodes refer.

leaf: A bottom-most node in a tree, which refers to no other nodes.

parent: The node that refers to a given node.

child: One of the nodes referred to by a node.

level: A set of nodes equidistant from the root.

prefix notation: A way of writing a mathematical expression with each operator appearing before its operands.

preorder: A way to traverse a tree, visiting each node before its children.

postorder: A way to traverse a tree, visiting the children of each node before the node itself.

inorder: A way to traverse a tree, visiting the left subtree, then the root, then the right subtree.

class variable: A `static` variable declared outside of any method. It is accessible from any method.

binary operator: An operator that takes two operands.

object encapsulation: The design goal of keeping the implementations of two objects as separate as possible. Neither class should have to know the details of the implementation of the other.

method encapsulation: The design goal of keeping the interface of a method separate from the details of its implementation.

callback: A flow of execution where provider code invokes a method provided by the client.

17.12 Exercises

Exercise 17.3

- What is the value of the postfix expression `1 2 + 3 *`?
- What is the postfix expression that is equivalent to the infix expression `1 + 2 * 3`?
- What is the value of the postfix expression `17 1 - 5 /`, assuming that `/` performs integer division?

Exercise 17.4 The height of a tree is the longest path from the root to any leaf. Height can be defined recursively as follows:

- The height of a null tree is 0.
- The height of a non-null tree is `1 + max (leftHeight, rightHeight)`, where `leftHeight` is the height of the left child and `rightHeight` is the height of the right child.

Write a method named `height` that calculates the height of the `Tree` provided as a parameter.

Exercise 17.5 Imagine we define a `Tree` that contains `Comparable` objects as cargo:

```
public class ComparableTree {
    Comparable cargo;
    Tree left, right;
}
```

Write a `Tree` class method named `findMax` that returns the largest cargo in the tree, where “largest” is defined by `compareTo`.

Exercise 17.6 A binary search tree is a special kind of tree where, for every node `N`:

all the cargo in the left subtree of `N` < the cargo in node `N`

and

the cargo in node $N <$ all the cargo in the right subtree of N

Using the following class definition, write an object method called `contains` that takes an `Object` as an argument and that returns true if the object appears in the tree or false otherwise. You can assume that the target object and all the objects in the tree are `Comparable`.

```
public class SearchTree {
    Comparable cargo;
    SearchTree left, right;
}
```

Exercise 17.7 In mathematics, a **set** is a collection of elements that contains no duplicates. The interface `java.util.Set` is intended to model a mathematical set. The methods it requires are `add`, `contains`, `containsAll`, `remove`, `size`, and `iterator`.

Write a class called `TreeSet` that extends `SearchTree` and that implements `Set`. To keep things simple, you can assume that `null` does not appear in the tree or as an argument to any of the methods.

Exercise 17.8 Write a method called `union` that takes two `Sets` as parameters and returns a new `TreeSet` that contains all the elements that appear in either `Set`.

You can add this method to your implementation of `TreeSet`, or create a new class that extends `java.util.TreeSet` and provides `union`.

Exercise 17.9 Write a method called `intersection` that takes two `Sets` as parameters and returns a new `TreeSet` that contains all the elements that appear in both `Sets`.

`union` and `intersection` are generic in the sense that the parameters can be any type in the metaclass `Set`. The two parameters don't even have to be the same type.

Exercise 17.10 One of the reasons the `Comparable` interface is useful is that it allows an object type to specify whatever ordering is appropriate. For types like `Integer` and `Double`, the appropriate ordering is obvious, but there are lots of examples where the ordering depends on what the objects are supposed to represent. In golf, for example, a low score is better than a high score; if we compare two `Golfer` objects, the one with the lower score wins.

- a. Write a definition of a `Golfer` class that contains a name and an integer score as instance variables. The class should implement `Comparable` and provide a `compareTo` method that gives higher priority to the lower score.
- b. Write a program that reads a file containing the names and scores of a set of golfers. It should create `Golfer` objects, put them in a `Priority Queue` and then take them out and print them. They should appear in descending order of priority, which is increasing order by score.

Tiger Woods	61
Hal Sutton	69
Phil Mickelson	72
Allen Downey	158

HINT: See Section C for code that reads lines from a file.

Exercise 17.11 Write an implementation of a Stack using a Vector. Think about whether it is better to push new elements onto the beginning or the end of the Vector.

Chapter 18

Heap

18.1 Array implementation of a tree

What does it mean to “implement” a tree? So far we have only seen one implementation of a tree, a linked data structure similar to a linked list. But there are other structures we would like to identify as trees. Anything that can perform the basic set of tree operations should be recognized as a tree.

So what are the tree operations? In other words, how do we define the Tree ADT?

constructor: Build an empty tree.

getLeft: Return the left child of this node.

getRight: Return the right child of this node.

getParent: Return the parent of this node.

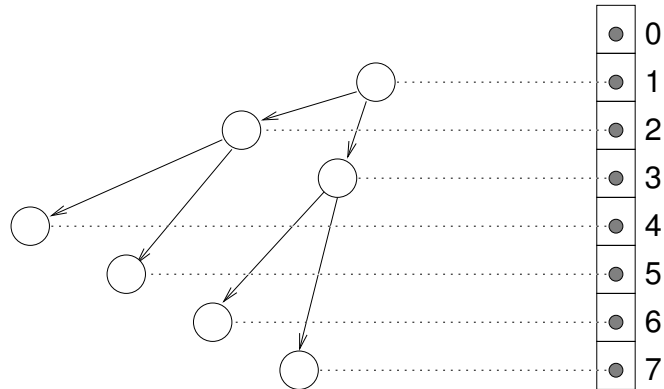
getCargo: Return the cargo object from this node.

setCargo: Assign a cargo object to this node (and create the node, if necessary).

In the linked implementation, the empty tree is represented by the special value `null`. `getLeft` and `getRight` are performed by accessing the instance variables of the node, as are `getCargo` and `setCargo`. We have not implemented `getParent` yet (you might think about how to do it).

There is another implementation of trees that uses arrays and indices instead of objects and references. To see how it works, we will start by looking at a hybrid implementation that uses both arrays and objects.

This figure shows a tree like the ones we have been looking at, although it is laid out at an angle. At the right there is an array of references that refer to the cargo in the nodes.



Each node in the tree has a unique index. Furthermore, the indices have been assigned to the nodes according to a deliberate pattern, in order to achieve the following results:

1. The left child of the node with index i has index $2i$.
2. The right child of the node with index i has index $2i + 1$.
3. The parent of the node with index i has index $i/2$ (rounded down).

Using these formulas, we can implement `getLeft`, `getRight` and `getParent` just by doing arithmetic; we don't have to use the references at all!

Since we don't use the references, we can get rid of them, which means that what used to be a tree node is now just cargo and nothing else. That means we can implement the tree as an array of cargo objects; we don't need tree nodes.

Here's what one implementation looks like:

```
public class ArrayTree {
    Object[] array;
    int size;

    public ArrayTree () {
        array = new Object [128];
    }
}
```

No surprises so far. The only instance variable is the array of `Objects` that contains the tree's cargo. The constructor initializes the array with an arbitrary initial capacity; the result is an empty tree.

Here is the simplest implementation of `getCargo` and `setCargo`.

```
public Object getCargo (int i) {
    return array[i];
}

public void setCargo (int i, Object obj) {
    array[i] = obj;
}
```

These methods don't do any error-checking, so if the parameter is wrong, they might generate an `ArrayIndexOutOfBoundsException` exception.

The implementation of `getLeft`, `getRight` and `getParent` is just arithmetic:

```
public int getLeft (int i) { return 2*i; }
public int getRight (int i) { return 2*i + 1; }
public int parent (int i) { return i/2; }
```

Finally we are ready to build a tree. In another class (the client), we would write

```
ArrayTree tree = new ArrayTree ();
tree.setCargo (1, "cargo for root");
```

The constructor builds an empty tree. Invoking `setCargo` puts the string "cargo for root" into the root node.

To add children to the root nodes:

```
tree.setCargo (tree.getLeft(1), "cargo for left");
tree.setCargo (tree.getRight(1), "cargo for right");
```

In the tree class we could provide a method that prints the contents of the tree in preorder.

```
public void print (int i) {
    Object cargo = tree.getCargo (i);
    if (cargo == null) return;
    System.out.println (cargo);
    print (getRight (i));
    print (getLeft (i));
}
```

To invoke this method, we have to pass the index of the root as a parameter.

```
tree.print (1);
```

The output is

```
cargo for root
cargo for left
cargo for right
```

This implementation provides the basic operations that define a tree. As I pointed out, the linked implementation of a tree provides the same operations, but the syntax is different.

In some ways, the array implementation is a bit awkward. For one thing, we assume that `null` cargo indicates a non-existent node, but that means that we can't put a `null` object in the tree as cargo.

Another problem is that subtrees aren't represented as objects; they are represented by indices into the array. To pass a tree node as a parameter, we have to pass a reference to the tree object and an index into the array. Finally, some operations that are easy in the linked implementation, like replacing an entire subtree, are harder in the array implementation.

On the other hand, this implementation saves space, since there are no links between the nodes, and there are several operations that are easier and faster in the array implementation. It turns out that these operations are just the ones we want to implement a Heap.

A Heap is an implementation of the Priority Queue ADT that is based on the array implementation of a Tree. It turns out to be more efficient than the other implementations we have seen.

To prove this claim, we will proceed in a few steps. First, we need to develop ways of comparing the performance of various implementations. Next, we will look at the operations Heaps perform. Finally, we will compare the Heap implementation of a Priority Queue to the others (arrays and lists) and see why the Heap is considered particularly efficient.

18.2 Performance analysis

When we compare algorithms, we would like to have a way to tell when one is faster than another, or takes less space, or uses less of some other resource. It is hard to answer those questions in detail, because the time and space used by an algorithm depend on the implementation of the algorithm, the particular problem being solved, and the hardware the program runs on.

The objective of this section is to develop a way of talking about performance that is independent of all of those things, and only depends on the algorithm itself. To start, we will focus on run time; later we will talk about other resources.

Our decisions are guided by a series of constraints:

1. First, the performance of an algorithm depends on the hardware it runs on, so we usually don't talk about run time in absolute terms like seconds. Instead, we usually count the number of abstract operations the algorithm performs.
2. Second, performance often depends on the particular problem we are trying to solve – some problems are easier than others. To compare algorithms, we usually focus on either the worst-case scenario or an average (or common) case.
3. Third, performance depends on the size of the problem (usually, but not always, the number of elements in a collection). We address this dependence explicitly by expressing run time as a function of problem size.
4. Finally, performance depends on details of the implementation like object allocation overhead and method invocation overhead. We usually ignore these details because they don't affect the rate at which the number of abstract operations increases with problem size.

To make this process more concrete, consider two algorithms we have already seen for sorting an array of integers. The first is **selection sort**, which we saw in Section 12.3. Here is the pseudocode we used there.

```
selectionsort (array) {
    for (int i=0; i<array.length; i++) {
        // find the lowest item at or to the right of i
        // swap the ith item and the lowest item
    }
}
```

To perform the operations specified in the pseudocode, we wrote helper methods named `findLowest` and `swap`. In pseudocode, `findLowest` looks like this

```
// find the index of the lowest item between
// i and the end of the array

findLowest (array, i) {
    // lowest contains the index of the lowest item so far
    lowest = i;
    for (int j=i+1; j<array.length; j++) {
        // compare the jth item to the lowest item so far
        // if the jth item is lower, replace lowest with j
    }
    return lowest;
}
```

And `swap` looks like this:

```
swap (i, j) {
    // store a reference to the ith card in temp
    // make the ith element of the array refer to the jth card
    // make the jth element of the array refer to temp
}
```

To analyze the performance of this algorithm, the first step is to decide what operations to count. Obviously, the program does a lot of things: it increments `i`, compares it to the length of the deck, it searches for the largest element of the array, etc. It is not obvious what the right thing is to count.

It turns out that a good choice is the number of times we compare two items. Many other choices would yield the same result in the end, but this is easy to do and we will find that it allows us to compare the sorting algorithms most easily.

The next step is to define the “problem size.” In this case it is natural to choose the size of the array, which we’ll call n .

Finally, we would like to derive an expression that tells us how many abstract operations (in this case, comparisons) we have to do, as a function of n .

We start by analyzing the helper methods. `swap` copies several references, but it doesn’t perform any comparisons, so we ignore the time spent performing

swaps. `findLowest` starts at `i` and traverses the array, comparing each item to `lowest`. The number of items we look at is $n - i$, so the total number of comparisons is $n - i - 1$.

Next we consider how many times `findLowest` gets invoked and what the value of i is each time. The last time it is invoked, i is $n - 2$ so the number of comparisons is 1. The previous iteration performs 2 comparisons, and so on. During the first iteration, i is 0 and the number of comparisons is $n - 1$.

So the total number of comparisons is $1 + 2 + \dots + n - 1$. This sum is equal to $n^2/2 - n/2$. To describe this algorithm, we would typically ignore the lower order term ($n/2$) and say that the total amount of work is proportional to n^2 . Since the leading order term is quadratic, we might also say that this algorithm is **quadratic time**.





18.3 Analysis of mergesort

In Section 12.6 I claimed that mergesort takes time that is proportional to $n \log n$, but I didn't explain how or why. Now I will.

Again, we start by looking at pseudocode for the algorithm. For mergesort, it's

```
mergeSort (array) {
    // find the midpoint of the array
    // divide the array into two halves
    // sort the halves recursively
    // merge the two halves and return the result
}
```

At each level of the recursion, we split the array in half, make two recursive calls, and then merge the halves. Graphically, the process looks like this:

	# arrays	items per array	# merges	comparisons per merge	total work
	1	n	1	$n-1$	$\sim n$
	2	$n/2$	2	$n/2-1$	$\sim n$
⋮	⋮	⋮	⋮	⋮	⋮
	$n/2$	2	$n/2$	$2-1$	$\sim n$
	n	1	0	0	

Each line in the diagram is a level of the recursion. At the top, a single array divides into two halves. At the bottom, n arrays with one element each are merged into $n/2$ arrays with 2 elements each.

The first two columns of the table show the number of arrays at each level and the number of items in each array. The third column shows the number of merges that take place at each level of recursion. The next column is the one