Chapter 12

Interfaces

Goals

- Understand what it means to implement a Java interface
- Use the Comparable interface to have any type of elements sorted and binary searched

12.1 Java Interfaces

Java has 422 interfaces. Of the 1,732 Java classes, 646 classes or 37%, implement one or more interfaces. Considering the large number of interfaces in Java and the high percentage of Java classes that implement the interfaces, interfaces should be considered to be an important part of the Java programming language. Interfaces are used for several reasons.

- guarantee a class has a particular set of methods and catch errors at compiletme rather than runtime.
- implement the same behavior with different algorithms and data structures where one may be better in some circumstances, and the other better in other circumstances.
- treat a variety of types as the same type where the same message results in different behavior
- provide programming projects that guarantee the required methods have the required method signature
- in larger projects, develop software using an interface before the completed implementation

You will not be asked to write the interfaces themselves (like the `TimeTalker` interface below). Instead, you will be asked to write a class that implements an `interface`. The interface will be given to you in programming projects.

A Java `interface` begins with a heading that is similar to a Java `class` heading except the keyword `interface` is used. A Java `interface` cannot have constructors or instance variables. A Java interface will be implemented by one or more Java classes that add instance variables and have their own constructors. A Java `interface` specifies the method headings that someone decided would represent what all instances of the class must be able to do.

Here is a sample Java interface that has only one method. Although not very useful, it provides a simple first example of an interface and the classes that implement the interface in different ways.

```
// File name: TimeTalker.java
// An interface that will be implemented by several classes
public interface TimeTalker {
  // Return a representation of how each implementing class tells time
  public String tellMeTheTime();
}
```

For each `interface`, there are usually two or more classes that implement it. Here are three classes that implement the `TimeTalker` interface. One instance variable has been added to store the name of any `TimeTalker`. A constructor was also needed to initialize this instance variable with anybody's name.

```
// File name: FiveYearOld.java
// Represent someone who cannot read time yet.
public class FiveYearOld implements TimeTalker {
  String name;

  public FiveYearOld(String name) {
    name = name;
  }

  public String tellMeTheTime() {
    return name + " says it's morning time";
  }
}
```

```
// File name: DeeJay.java
// A "hippy dippy" DJ who always mentions the station
public class DeeJay implements TimeTalker {
  String name;

  public DeeJay(String name) {
    name = name;
  }

  public String tellMeTheTime() {
    return name + " says it's 7 oh 7 on your favorite oldies station";
  }
}
```

```
// File name: FutureOne.java
// A "Star-Trekker" who speaks of star dates
public class FutureOne implements TimeTalker {
  String name;

  public FutureOne(String name) {
    name = name;
  }

  public String tellMeTheTime() {
    return name + " says it's star date 78623.23";
  }
}
```

One of several reasons that Java has interfaces is to allow many classes to be treated as the same type. To demonstrate that this is possible, consider the following code that stores references to three different types of objects as `TimeTalker` variables.

```
// These three objects can be referenced by variables
// of the interface type that they implement.
TimeTalker youngOne = new FiveYearOld("Pumpkin");
TimeTalker dj = new DeeJay("WolfMan Jack");
TimeTalker captainKirk = new FutureOne("Jim");

System.out.println(youngOne.tellMeTheTime());
System.out.println(dj.tellMeTheTime());
System.out.println(captainKirk.tellMeTheTime());
```

Output

```
Pumpkin says it's morning time
WolfMan Jack says it's 7 oh 7 on your favorite oldies station
Jim says it's star date 78623.23
```

Because each class implements the `TimeTalker` interface, references to instances of these three classes—`FiveYearOld`, `DeeJay`, and `FutureOne`—can be stored in the reference type variable `TimeTalker`. They can all be considered to be of type `TimeTalker`. However, the same message to the three different classes of `TimeTalker` objects results in three different behaviors. The same message executes three different methods in three different classes.

A Java interface specifies the exact method headings of the classes that need to be implemented. These interfaces capture design decisions—what instances of the class should be able to do—that were made by a team of programmers.

Self-Check

12-1 Write two classes that implement this interface:

```
public interface BarnyardAnimal {
  public String sound();
}
```

The following program must generate the output exactly as shown.

```
public class OldMacDonald {
  public static void main(String[] args) {
    // Supply the sound you would like the BarnyardAnimal(as a String)
    BarnyardAnimal a1 = new Chicken("cluck");
    BarnyardAnimal a2 = new Cow("moo");

    System.out.println("With a " + a1.sound() + " " + a1.sound() + " here,");
    System.out.println("and a " + a2.sound() + " " + a2.sound() + " there.");
    System.out.println("Here a " + a1.sound() + ",");
    System.out.println("there a " + a2.sound());
    System.out.println("everywhere a " + a2.sound() + " " + a1.sound() + ".");
  }
}
```

Output

```
With a cluck cluck here,
and a moo moo there.
Here a cluck,
there a moo,
everywhere a moo cluck.
```

12.2 The Comparable Interface

The `compareTo` method has been shown to compare two String objects to see if one was less than, greater than, or equal to another. This section introduces a general way to compare any objects with the same `compareTo` message. This is accomplished by having a class implement the `Comparable` interface. Java's `Comparable` interface has just one method—`compareTo`.

```
public interface Comparable<T> {

/*
 * Returns a negative integer, zero, or a positive integer when this object is
 * less than, equal to, or greater than the specified object, respectively.
 */
  public int compareTo(T other);
}
```

The angle brackets represent a new syntactical element that specifies the type to be compared. Since any class can implement the `Comparable` interface, the `T` in `<T>` will be replaced with the class name that implements the interface. This ensures the objects being compared are the same class. This example shows how one class may implement the `Comparable<T>` interface.

```
public class AsSmallAsPossible implements Comparable<AsSmallAsPossible> {

  public int compareTo(AsSmallAsPossible other) {
    // TODO Auto-generated method stub
    return 0;
  }
}
```

With this incomplete type only shown to demonstrate a class implementing an interface (it is useless otherwise), all `AsSmallAsPossible` objects would be considered equal since the `compareTo` always returns 0.

When a class implements the `Comparable` interface, instances of that class are guaranteed to understand the `compareTo` message. Some Java classes already implement `Comparable`.[1] Additionally, any class you write may also implement the `Comparable` interface. For example, to sort or binary search an array of `BankAccount` objects, `BankAccount` can be made to implement `Comparable<BankAccount>`. Then two `BankAccount` objects in an array named `data` can be compared to see if one is less than another:

```
  // . . . in the middle of selection sort . . .
  // Then compare all of the other elements, looking for the smallest
```

[1] Some of the Java classes that implement the `Comparable` interface are `Character`, `File`, `Long`, `ObjectStreamField`, `Short`, `String`, `Float`, `Integer`, `Byte`, `Double`, `BigInteger`, `BigDecimal`, `Date`, and `CollationKey`.

```
for(int index = top + 1; index < data.length; index++) {
  if(data[index].compareTo(data[indexOfSmallest]) < 0 )
    indexOfSmallest = index;
}
// . . .
```

The Boolean expression highlighted above is true if `data[index]` is "less than" `data[smallestIndex]`. What "less than" means depends upon how the programmer implements the `compareTo` method. The `compareTo` method defines what is known as the "natural ordering" of objects. For strings, it is alphabetic ordering. For `Double` and `Integer` it is what we already know: 3 < 4 and 1.2 > 1.1, for example. In the case of `BankAccount`, we will see that one `BankAccount` can be made to be "less than" another when its ID precedes the other's ID alphabetically. The same code could be used to sort any type of objects as long as the class implements the `Comparable` interface. Once sorted, the same binary search method could be used for any arry of objects, as long as the objects implement the comparable interface.

To have a new type fit in with this general method for comparing two objects, first change the class heading so the type implements the `Comparable` interface.

```
public class BankAccount implements Comparable<BankAccount> {
```

The `<T>` in the `Comparable` interface becomes `<BankAccount>`. If the class name were `String`, the heading would use `<String>` as in

```
public class String implements Comparable<String> {
```

Adding **`implements`** is not enough. An attempt to compile the class without adding the `compareTo` method results in this compile time error:

```
The type BankAccount must implement the inherited abstract method
Comparable<BankAccount>.compareTo(BankAccount)
```

Adding the `compareTo` method to the `BankAccount` class resolves the compile time error. The heading *must* match the method in the interface. So the method must return an int. And the `compareTo` method *really should* return zero, a negative, or a positive integer to indicate if the object before the dot (the receiver of the message) is equal to, less than, or greater than the object passed as the argument. This desired behavior is indicated in the following test method.

```
@Test
public void testCompareTo() {
  BankAccount b1 = new BankAccount("Chris", 100.00);
  BankAccount b2 = new BankAccount("Kim", 100.00);
  // Note: The natural ordering is based on IDs, the balance is ignored
  assertTrue(b1.compareTo(b1) == 0); // "Chris" == "Chris"
  assertTrue(b1.compareTo(b2) < 0);  // "Chris" < "Kim"
  assertTrue(b2.compareTo(b1) > 0);  // "Kim" > "Chris"
}
```

Since the `Comparable` interface was designed to work with any Java class, the `compareTo` method must have a parameter of that same class.

```
/**
  * This method allows for comparison of two BankAccount objects.
```

```
   *
   * @param other is the object being compared to this BankAccount.
   *
   * @return a negative integer if this object has an ID that alphabetically
   * precedes other (less than), 0 if the IDs are equals, or a positive
   * integer if this object follows other alphabetically (greater than).
   */
  public int compareTo(BankAccount other) {
    if (this.getID().compareTo(other.getID()) == 0)
      return 0;  // This object "equals" other
    else if (this.getID().compareTo(other.getID()) < 0)
      return -1; // This object < other
    else
      return +1; // This object > other
  }
```

Or since the String `compareTo` method exists, this particular `compareTo` method can be written more simply as follows

```
  public int compareTo(BankAccount other) {
    return this.getID().compareTo(other.getID());
  }
```

The Implicit Parameter `this`

The code shown above has `getID()` messages sent to **`this`**. In Java, the keyword **`this`** is a reference variable that allows an object to refer to itself. When **`this`** is the receiver of a message, the object is using its own internal state (instances variables). Because an object sends messages to itself so frequently, Java provides a shortcut: **`this`** and the dot are not really necessary before the method name. Whereas the keyword **`this`** is sometimes required, it was not really necessary in the code above. It was used only to distinguish the two objects being compared. Therefore the method could also be written as follows:

```
  public int compareTo(BankAccount other) {
    return getID().compareTo(other.getID()); // "this." removed
   }
```

It's often a matter of taste of when to use **`this`**. You rarely need **`this`**, but **`this`** sometimes clarifies things. Here is an example where **`this`** is needed. Since the constructor parameters have the same names as the instance variables, the assignments currently have no effect.

```
public class BankAccount implements Comparable<BankAccount> {

  // Instance variables that every BankAccount object will maintain.
  private String ID;
   private double balance;

  public BankAccount(String ID, double balance) {
    ID = ID;
    balance = balance;
  }
```

If you really want to name instance variables the same as the constructor parameters, you must write **`this`** to distinguish the two. With the code above, the instance variables will never change

to the expected values of the arguments. To fix this error, add **`this`** to distinguish instance variables from parameters.

```
  public BankAccount(String ID, double balance) {
    this.ID = ID;
    this.balance = balance;
  }
```

Self-Check

12-2 Modify the `compareTo` method to define the natural ordering of `BankAccount` to be based on balances rather than IDs. One account is less than another if the balance is less than the other. The following assertions must pass.

```
@Test
  public void testCompareTo() {
    BankAccount b1 = new BankAccount("Chris", 111.11);
    BankAccount b2 = new BankAccount("Chris", 222.22);
    // Note: The natural ordering is based on the balance. IDs are ignored.
    assertTrue(b1.compareTo(b1) == 0); // 111.11 == 111.11
    assertTrue(b1.compareTo(b2) < 0);  // 111.11  < 222.22
    assertTrue(b2.compareTo(b1) > 0);  // 222.22 > 111.11
  }
```

Answers to Self-Check Questions

12-1

```
public class Chicken implements BarnyardAnimal {
  private String mySound;

  public Chicken(String sound) {
    mySound = sound;
  }

  public String sound() {
    return mySound;
  }
}

public class Cow implements BarnyardAnimal {
  private String mySound;

  public Cow(String sound) {
    mySound = sound;
  }

  public String sound() {
    return mySound;
  }
}
```

12-2

```
  public int compareTo(BankAccount other) {
    double thisObjectsPennies = 100 * this.getBalance();
    double theOtherObjectsPennies = 100 * other.getBalance();
    return (int) thisObjectsPennies - (int) theOtherObjectsPennies;
  }
```

Chapter 13

Collection Considerations

Goals

- Introduce class Object and inheritance
- Show how one collection can store any type of element using `Object[]`
- Show how one collection can store any type of element using generics
- Understand the three storage structures used in this textbook: arrays, singly linked, trees
- Distinguish Abstract Data Types, Data Structures and Collection Classes

---

13.1 The `Object` class

The `StringBag` class shown of Chapter 9 could only store one type of element. It is desirable to have a collection class to store any type. Java provides at least two approaches:

1. Store reference to `Object` objects rather than just String (this section)
2. Use Java generics (later in this chapter)

We'll consider the first option now, which requires knowledge of Java's `Object` class, inheritance, and casting.

Java's `Object` class has one constructor (no arguments) and 11 methods, including `equals` and `toString`. All classes in Java extend the `Object` class or another class that extends `Object`. There is no exception to this. All classes inherit the methods of the `Object` class.

One class inherits the methods and instance variables of another class with the keyword `extends`. For example, the following class heading explicitly states that the `EmptyClass` class inherits all of the method of class `Object`. If a class heading does not have `extends`, the compiler automatically adds `extends Object`.

```
// This class extends Object even if extends Object is omitted
public class EmptyClass extends Object {
}
```

Even though this `EmptyClass` defines no methods, a programmer can construct and send messages to `EmptyClass` objects. This is possible because a class that extends `Object` inherits (obtains) `Object`'s methods. A class that extends another is called a subclass. Here are three of the methods that `EmptyClass` inherits from the `Object` class:

Two Methods of the `Object` Class

- **`toString`** returns a `String` that is the class name concatenated with the at symbol (@) and a hexadecimal (base 16) number related to locating the object at runtime.
- **`equals`** returns `true` if both the receiver and argument reference the same object.

Additionally, a class that does not declare a constructor is automatically given a default constructor. This is to ensure that constructor for Object gets invoked. The following code is equivalent to that shown above

```
// This class extends Object implicitly
public class EmptyClass {

  public EmptyClass() {
    // Explicitly call the constructor of the superclass, which is Object
    super();
  }
}
```

This following code shows these two methods used by a class that extends class `Object`.

```
  EmptyClass one = new EmptyClass();
  EmptyClass two = new EmptyClass();

  assertFalse(one.equals(two)); // passes
  System.out.println(two.toString());
  System.out.println(one.toString());

  // The variable two will now reference the same object as one
  one = two;
  assertTrue(one.equals(two));
  System.out.println("after assignment->");
  System.out.println(two.toString());
  System.out.println(one.toString());
```

Output

```
EmptyClass@8813f2
EmptyClass@1d58aae
after assignment->
EmptyClass@8813f2
EmptyClass@8813f2
```

The `Object` class captures methods that are common to all Java objects. Java makes sure that all classes extend the `Object` class because there are several things that all objects must be capable of in order to work with Java's runtime system. For example, `Object`'s constructor gets invoked for every object construction to help allocate computer memory for the object at runtime. The class also has methods to allow different processes to run at the same time, allowing applications such as Web browsers to be more efficient. Java programs can download several files while browsing elsewhere, while creating another image, and so on.

One-way Assignment Compatibility

Because all classes extend Java's `Object` class, a reference to any type of object can be assigned to an `Object` reference variable. For example, consider the following valid code that assigns a

`String` object and an `EmptyClass` object to two different reference variables of type `Object`:

```
String aString = new String("first");
// Assign a String reference to an Object reference
Object obj1 = aString;

EmptyClass one = new EmptyClass();
// Assign an EmptyClass reference to an Object reference
Object obj2 = one;
```

Java's one-way assignment compatibility means that you can assign a reference variable to the class that it extends. However, you cannot directly assign in the other direction. For example, an `Object` reference cannot be directly assigned to a `String` reference.

```
Object obj = new Object();
String str = obj;
```

Type mismatch: cannot convert from Object to String

This compile time error occurs because the compiler recognizes that `obj` is a reference to an `Object` that cannot be assigned down to a `String` reference variable.

13.2 A Generic Collection

This `GenericList` class uses `Object` parameters in the `add` and `remove` methods, an `Object` return type in the `get` method, and an array of `Object`s as the instance variable to store any type element that can be assigned to `Object` (which is any Java type).

```
public class GenericList {

  private Object[] elements;
  private int n;

  public GenericList() {
    elements = new Object[10];
  }

  public void add(Object elementToAdd) {
    // . . .
  }

  public boolean remove(Object elementToRemove) {
    // . . .
  }

  public Object get(int index) {
    // . . .
  }
}
```

This design allows one class to store collections with any type of elements:

```
@Test
public void testGenericity() {
  GenericList names = new GenericList();
  names.add("Kim");
  names.add("Devon");
```

```
    GenericList accounts = new GenericList();
    accounts.add(new BankAccount("Speilberg", 1942));

    GenericList numbers = new GenericList();
    numbers.add(12.3);
    numbers.add(4.56);
  }
```

In such a class, a method that returns a value would have to return an `Object` reference.

```
  public Object get(int index) {
    return elements[index];
  }
```

This approach requires a cast. You have to know the type stored.

```
  @Test
  public void testGet() {
    GenericListA strings = new GenericListA();
    strings.add("A");
    strings.add("B");
    String firstElement = (String) strings.get(0);
    assertEquals("A", firstElement);
  }
```

With this approach, programmers always have to cast, something Java software developers had been complaining about for years (before Java 5). With this approach, you also have to be wary of runtime exceptions. For example, even though the following code compiles, when the test runs, a runtime error occurs.

```
  @Test
  public void testGet() {
    GenericList strings = new GenericList();
    strings.add("A");
    strings.add("B");

    // Using Object objects for a generic collection is NOT type safe.
    // Any type of object can be added accidentally (not usually desirable).
    strings.add(123);

    // The attempt to send cast to all elements fails when index == 2:
    for (int index = 0; index < 3; index++) {
      String theString = (String) strings.get(index);
      System.out.println(theString.toLowerCase());
    }
  }
```

java.lang.ClassCastException: java.util.Integer

`strings.get(2)` returns a reference to an integer, which the runtime treats as a `String` in the cast. A `ClassCastException` occurs because a String cannot be cast to an integer. In a later section, Java Generics will be shown as a way to have a collection store a specific type. One collection class is all that is needed, but the casting and runtime error will dissapear.

Self-Check

13-1 Which statements generate compiletime errors?

```
    Object anObject = "3";            // a.
    int anInt = anObject;             // b.
    String aString = anObject;        // c.
    anObject = new Object();          // d.
```

13-2 Which letters represent valid assignment statements that compile?

```
Object obj = "String";            // a.
String str = (String)obj;         // b.
Object obj2 = new Point(3, 4);    // c.
Point p = (String)obj2;           // d.
```

13-3 Which statements generate compile time errors?

```
Object[] elements = new Object[5]; // a.
elements[0] = 12;                  // b.
elements[1] = "Second";            // c.
elements[2] = 4.5;                 // d.
elements[3] = new Point(5, 6);     // e.
```

13.3 ADTs, Collection Classes, and Data Structures

A collection is an object to store, retrieve, and manipulate a collection of objects in some meaningful way. Collection classes have the following characteristics:

- The main responsibility of a collection class is to store a collection of values.
- Elements may be added and removed from the collection.
- A collection class allows clients to access the individual elements.
- A collection class may have search-and-sort operations for locating a particular element.
- Some collections allow duplicate elements while other collections do not.
- Some collections are naturally ordered while other collections are not.

Some collections are designed to store large amounts of information where any element may need to be found quickly—to find a phone listing, for example. Other collections are designed to have elements that are added, removed, and changed frequently—an inventory system, for example. Some collections store elements in a first in first out basis—such as a queue of incoming packets on an Internet router. Other collections are used to help build computer systems—a stack that manages method calls, for example

Collection classes support many operations. The following is a short list of operations that are common to many collection classes:

- **add** Place an object into a collection (insertion point varies).
- **find** Get a reference to an object in a collection so you can send messages to it.
- **remove** Take the object out of the collection (extraction point varies).

The Java class is a convenient way to encapsulate algorithms and store data in one module. In addition to writing the class and method headings, decisions have to be made about what data structures to use.

Data Structures

A data structure is a way of storing data on a computer so it can be used efficiently. There are several structures available to programmers for storing data. Some are more appropriate than others, depending on how you need to manage your information. Although not a complete list, here are some of the storage structures you have or will see in this book:

1. arrays
2. linked structure
3. tables

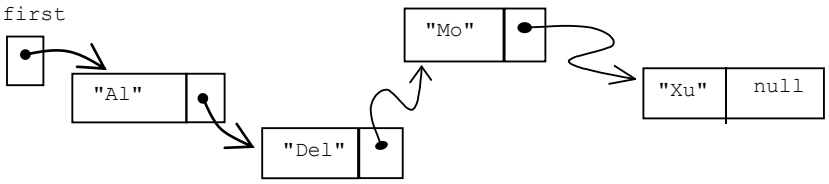
Data can be stored in contiguous memory with arrays or in non-contiguous memory with linked structures. Arrays allow you to reserve memory where each element can be physically located next to its predecessor and successor. Any element can be directly changed and accessed through an index.

```
String[] data = new String[5];
data[0] = "Al";
data[1] = "Di";
data[2] = "Mo";
data[3] = "Xu";
```

`data` (where data.length == 5):

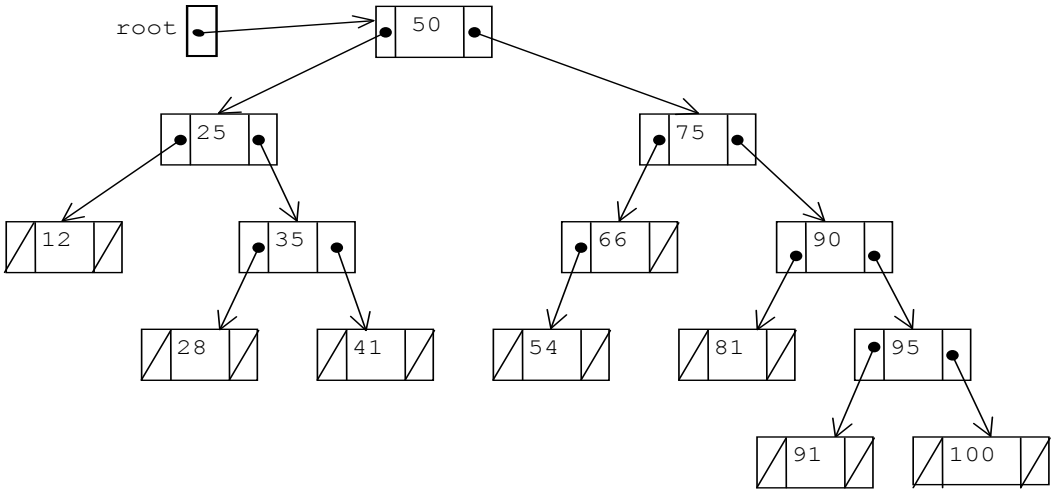
[0]	[1]	[2]	[3]	[4]
"Al"	"Di"	"Mo"	"Xu"	null

A linked structure contains nodes that store a reference to an element and a reference to another node. The reference to another node may be `null` to indicate there are no more elements stored.



You will see that both of these storage mechanisms — arrays and linked structures — implement the same List ADT in subsequent chapters.

You will also see how binary trees store data that can be added, removed, and found more quickly. The following picture indicates a linked structure where each node contains a reference to data (integers are used here to save space) and references to two other nodes, or to nothing (shown as diagonal lines).



You will also see another storage mechanism that uses a key—such as a student ID, employee number, or Social Security Number—to find the value very quickly.

Array Index	Key	Value mapped to the key *null or 5 instance variables shown*
[0]	"1023"	"Devon" 40.0 10.50 1 'S'
[1]	null	null
[2]	"5462"	"Al" 42.5 12.00 2 'M'
[3]	null	null
[4]	"3343"	"Ali" 20.0 9.50 0 'S'
…		
[753]	"0930"	"Chris" 0.0 13.50 1 'S'

One focal point for the remainder of this textbook is collections that use different data structures to implement abstract data types.

Abstract Data Types

An **abstract data type** (ADT) describes a set of data values and associated operations that are precisely specified independent of any particular implementation. An abstract data type can be specified using axiomatic semantics. For example, here is the Bag ADT as described by the National Institute of Standards and Technology (NIST)[2].

Bag

Definition: An unordered collection of values that may have duplicates.

[2] http://www.nist.gov/dads/HTML/bag.html

Formal Definition: A bag has a single query function, occurencesOf(v, B), which tells how many copies of an element are in the bag, and two modifier functions, add(v, B) and remove(v, B). These may be defined with axiomatic semantics as follows.

1. new() returns a bag
2. occurencesOf(v, new()) = 0
3. occurencesOf(v, add(v, B)) = 1 + occurencesOf(v, B) [3]
4. occurencesOf(v, add(u, B)) = occurencesOf(v, B) if $v \neq u$
5. remove(v, new()) = new()
6. remove(v, add(v, B)) = B
7. remove(v, add(u, B)) = add(u, remove(v, B)) if $v \neq u$

where B is a bag and u and v are elements.

The predicate isEmpty(B) may be defined with the following additional axioms:

8. isEmpty(new()) = true
9. isEmpty(add(v, B)) = false

Also known as multi-set.
Note: A bag, or multi-set, is a set where values may be repeated. Inserting 2, 1, 2 into an empty set gives the set {1, 2}. Inserting those values into an empty bag gives {1, 2, 2}.

Although an abstract data type describes how each operation affects data, an ADT

- does not specify how data will be stored
- does not specify the algorithms needed to accomplish the listed operations
- is not dependent on a particular programming language or computer system.

An ADT can also be described as a Java interface that specifies the operations and JUnit assertions to specify behavior.

ADTs Described with Java Interfaces and JUnit Assertions

The Java `interface` introduced in the preceding chapter can also be used to specify an abstract data type. For example, the following Java interface specifies the operations for a Bag ADT as method headings. The `new` operation from NIST's Bag ADT is not included here simply because

[3] This definition shows a bag B is passed as an argument. In an object-oriented language you send a message to an object of type B as in `aBag.add("New Element);` rather than `add("New Element", aBag);`

the Java interface does not allow constructor to be specified in an interface. We will use `new` later when there is some class that implements the interface. The syntactic element <E> will require the class to specify the type of element to be stored.

```
/**
 * This interface specifies the methods for a Bag ADT. It is designed to be
 * generic so any type of element can be stored.
 */
public interface Bag<E> {

  /**
   * Add element to this Bag.
   *
   * @param element: The element to insert.
   */
  public void add(E element);

  /**
   * Determine if there are any elements in this bag.
   *
   * @return False if there is one or more elements in this bag.
   */
  public boolean isEmpty();

  /**
   * Return how many elements match element according to the equals method of
   * the type specified at construction.
   *
   * @param element The element to count in this Bag.
   */
  public int occurencesOf(E element);

  /**
   * Remove the first occurrence of element and return true if found. Otherwise
   * leave this Bag unchanged.
   *
   * @param element: The element to remove
   * @return true if element was removed, false if element was not found.
   */
  public boolean remove(E element);
}
```

Like an ADT specified with axiomatic expressions, an `interface` does not specify the data structure and the algorithms to add, remove, or find elements. Comments and well-named identifiers imply the behavior of the operations; however the behavior can be made more explicit with assertions. For example, the assertions like those shown in the following test method help describe the behavior of `add` and `occurencesOf`. This code assumes that a class named `ArrayBag` implements `interface Bag<E>`.

```
  @Test
  public void testOccurencesOf() {
    Bag<String> names = new ArrayBag<String>();

    // A new bag has no occurrences of any string
    assertEquals(0, names.occurencesOf("NOT here"));
    names.add("Sam");
    names.add("Devon");
    names.add("Sam");
    names.add("Sam");
    assertEquals(0, names.occurencesOf("NOT Here"));
```

```
    assertEquals(1, names.occurencesOf("Devon"));
    assertEquals(3, names.occurencesOf("Sam"));
  }
```

One Class for all Types

The Bag ADT from NIST does not specify what type of elements can be stored. The Bag interface does. Any class that implements the Bag interface can store any type of element. This is specified with the parameter E in `add`, `occurencesOf`, and `remove`. The compiler replaces the type specified in a construction wherever E is found. So, when a Bag is constructed to store BankAccount objects, you can add BankAccount objects. At compiletime, the parameter E in `public void add(E element)` is considered to be BankAccount.

```
Bag<BankAccount> accounts = new ArrayBag<BankAccount>();
accounts.add(new BankAccount("Chris", 100.00));
accounts.add(new BankAccount("Skyler", 2802.67));
accounts.add(new BankAccount("Sam", 31.13));
```

When a `Bag` is constructed to store `Point` objects, you can add `Point` objects. At compile time, the parameter `E in public void add(E element)` is considered to be `Point`.

```
Bag<Point> points = new ArrayBag<Point>();
points.add(new Point(2, 3));
points.add(new Point(0, 0));
points.add(new Point(-2, -1));
```

Since Java 5.0, you can add primitive values. This is possible because integer literals such as `100` are autoboxed as new `Integer(100)`. At compiletime, the parameter `E` in `public void add(E element)` is considered to be `Integer` for the Bag `ints`.

```
Bag<Integer> ints = new ArrayBag<Integer>();
ints.add(100);
ints.add(95);
ints.add(new Integer(95));
```

One of the advantages of using generics is that you cannot accidentally add the wrong type of element. Each of the following attempts to add the element that is not of type E for that instance results in a compiletime error.

```
ints.add(4.5);
ints.add("Not a double");
points.add("A String is not a Point");
accounts.add("A string is not a BankAccount");
```

Implementing the Bag interface

A JUnit test describes the behavior of methods in a very real sense by sending messages and making assertions about the state of a `Bag` object. The assertions shown describe operations of the Bag ADT in a concrete fashion. We'll write a test and then the code to make it compile. We'll also

need a data structure to store the elements. We'll use an array in this collection class and choose the name `ArrayBag`.

The first assertion shows that a bag is empty immediately after `new` and no longer empty after one `add` (see axioms 8 and 9 above).

```
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import org.junit.Test;

// This unit test shows another view of the bag ADT, this time as a unit test
public class ArrayBagTest {

  @Test
  public void testIsEmptyWithOneAdd() {
    Bag<String> names = new ArrayBag<String>();
    assertTrue(names.isEmpty());
    names.add("Kim");
    assertFalse(names.isEmpty());
  }

    // More to come . . .
}
```

If any assertion fails, either that assertion is stated incorrectly or a method such as `isEmpty`, `add` or the constructor (`new`) may be wrong. In any case, you will know something is wrong with the concrete implementation or testing of the new type (a Java class). Here is the start of a generic class that implements an interface.

```
// An ArrayBag object can store elements as a multi-set where elements can
// "equals" each other.  There is no specific order to the elements.
public class ArrayBag<E> implements Bag<E> {

  // --Instance variables
  private Object[] data;
  private int n;

  // Construct an empty bag that can store any type of element.
  public ArrayBag() {
    data = new Object[20];
    n = 0;
  }


   // Return true if there are 0 elements in this bag.
   public boolean isEmpty() {
    return n == 0;
  }

  // Add element to this bag
  public void add(E element) {
    data[n] = element;
    n++;
  }

  public int occurencesOf(E element) {
    // TODO Change this method stub to a working method
    return 0;
  }

  public boolean remove(E element) {
```

```
    // TODO Change this method stub to a working method
    return false;
  }
}
```

There are three new things shown in this new class:

1. `ArrayBag` uses an array of `Object` references to store the elements. Since we want one class to store many different types, it cannot be one particular type. The array could also be declared to be type `E`. `Object` was chosen because at runtime that is exactly the type it will be: an array of `Object`. (It also avoids an ugly cast and a warning.)
2. Since the `ArrayBag<E>` class implements the `Bag<E>` interface, all methods from the interface must exist for class to compile. Two methods are still stubs, just enough to make it compile, but not yet working as specified: `occurencesOf` always returns 0 and `remove` does not do anything other than return false. (`occurencesOf` will be discussed next, but it will be your job to write the appropriate code for the `remove` method.)
3. The big-O runtimes are added as comments to describe the runtime behavior for each.

occurencesOf

The next test method provides another way to indicate that after adding an element with the value "Kim", the Bag has 1 more of that value than before (see axiomatic expression 3 above). It also ensures there are none of a particular value in a new Bag (see axiomatic expression 2 above).

```
@Test
public void testOccurencesOfWithOneElement() {
  // This code assumes ArrayBag<E> is a class that implements Bag<E>
  Bag<String> names = new ArrayBag<String>();
  assertEquals(0, names.occurencesOf("Kim"));
  names.add("Kim");
  assertEquals(1, names.occurencesOf("Kim"));
}
```

Another test method verifies that duplicate elements are allowed.

```
@Test
public void testOccurencesOf() {
  Bag<String> names = new ArrayBag<String>();
  names.add("Sam");
  names.add("Devon");
  names.add("Sam");
  names.add("Sam");
  assertEquals(1, names.occurencesOf("Devon"));
  assertEquals(3, names.occurencesOf("Sam"));
}
```

Since there is no specified ordering to `Bag` objects, the element passed as an argument may be located at any index. Also, a value that equals the argument may occur more than once. Thus each element in indexes 0..n-1 must be compared. It makes the most sense to use the `equals` method, assuming `equals` has been overridden to compare the state of two objects rather than the reference values.

```
// Return how many objects currently in the bag match an element, using
```

```
// the equals method of whatever type of element is stored in this bag.
public int occurencesOf(E element) {
  int result = 0;
  for (int i = 0; i < n; i++) {
    if (element.equals(data[i]))
      result++; // Found one that equals element
  }
  return result;
}
```

Self-Check

13-4 Write a test method for `remove`.

13-5 Implement the `remove` method as if you were writing it inside the `ArrayBag` class.

13.4 Collections of Primitive Types

Collections of the primitive types such `int`, `double`, `char` can also be stored in a generic class. The type parameter must be one of Java's "wrapper" classes. Java has a "wrapper" class for each primitive type named `Integer`, `Double`, `Character`, `Boolean`, `Long`, `Short`, and `Float`. A wrapper class does little more than allow a primitive value to be viewed as a reference type.

The construction of a generic class requires a reference type. We must use `Character` rather than `char` and `Integer` rather than `int`.

```
Bag<Integer> bagOfInts = new ArrayBag<Integer>();
```

With the parameter set to be `Integer`, integer values can be added like this:

```
bagOfInts.add(new Integer(1));
```

Integer values can also be added like this:

```
bagOfInts.add(123);
bagOfInts.add(-99);
```

Java 5.0 allows primitive integers to be treated like objects through a process known as boxing.

Auto Boxing / UnBoxing

Before Java 5, to treat primitive type as reference types, programmers were required to "box" primitive values in their wrapper class.

```
Integer anInt = new Integer(123);
```

To convert from reference type back to a primitive type, programmers were required to "unbox" by asking the `Integer` object for its `intValue` like this:

```
int primitiveInt = anInt.intValue();
```

In Java 5.0, this boxing and unboxing is done automatically.

```
Integer anotherInt = 123;  // boxed automatically
int anotherPrimitiveInt = anotherInt;  // unboxed automatically
```

This allows a generic class to be instantiated with reference type parameters such as `Integer` so primitive ints can be added. The boxing occurs automatically during the add as shown in the comment.

```
@Test
public void testAddAndOccurencesOf() {
  Bag<Integer> bagOfInts = new ArrayBag<Integer>();

  // Java boxes 5 as new Integer(5) automatically.
  bagOfInts.add(5);
  bagOfInts.add(7);
  bagOfInts.add(5);
}
```

Auto unboxing comes in handy when a parameterized collection has a method that returns the type of object specified at construction. For example, if a get method had a return type of `Object` like this:

```
public Object get(int atIndex)
```

a cast like this would be required:

```
Integer anInt = (Integer)bagOfInts.get(0);
```

However, with a generic parameter E and the return type of E,

```
public E get(int atIndex)
```

the cast is not required

```
Integer anInt = bagOfInts.get(0);
```

In fact, because Java 5 will unbox automatically, the return type of E when E is Integer allows the return value to also be stored into a primitive variable:

```
int primitiveInt = list.get(0);  // No intValue required, auto-unboxed
```

Most of the collection classes throughout the remainder of this textbook will use parameterized classes like.

```
public class GenericList<E> {

  private Object[] data = new Object[100];

  public void addElementAt(int index, E element) { /* */ }

  public E get(int index) { /* */ }
}
```

Java 5 generics requires extra syntax when construct a collection (two sets of angle brackets and the type to be stored twice), however the benefits include much less casting syntax, we can have collections of primitives, and we gain the type safety that comes from allowing the one type to element to be maintained by the collection.

Self-Check

13-6 Place a check mark √ in the comment after assignment statement that compiles (or leave blank).

```
Object anObject = new Object();
String aString = "abc";
Integer anInteger = new Integer(5);
anObject = aString;    // _______
anInteger = aString;   // _______
anObject = anInteger;  // _______
anInteger = anObject;  // _______
```

13-7 Place a check mark √ in the comment after assignment statement that compiles (or leave blank).

```
Object anObject = new String("abc");
Object anotherObject = new Integer(50);
Integer n = (Integer) anObject;          // _______
String s = (String) anObject;            // _______
anObject = anotherObject;                // _______
String another = (String) anotherObject; // _______
Integer anotherInt = (Integer) anObject; // _______
```

Answers to Self-Check Questions

13-1 which have errors? b and c
-b cannot assign an Object to an Integer
-c cannot assign an Object to a String even when it references a String

13-2 a, b, and c. In d, the compiler notices the attempt to store a String into a Point?

```
Point p = (String)obj2;          // d.
```

13-3 None

13-4 One possible test method for remove:

```
@Test
public void testRemove() {
  Bag<String> names = new ArrayBag<String>();
  names.add("Sam");
  names.add("Chris");
  names.add("Devon");
  names.add("Sandeep");

  assertFalse(names.remove("Not here"));

  assertTrue(names.remove("Sam"));
  assertEquals(0, names.occurencesOf("Sam"));
  // Attempt to remove after remove
  assertFalse(names.remove("Sam"));
  assertEquals(0, names.occurencesOf("Sam"));

  assertEquals(1, names.occurencesOf("Chris"));
  assertTrue(names.remove("Chris"));
  assertEquals(0, names.occurencesOf("Chris"));

  assertEquals(1, names.occurencesOf("Sandeep"));
  assertTrue(names.remove("Sandeep"));
  assertEquals(0, names.occurencesOf("SanDeep"));

  // Only 1 left
  assertEquals(1, names.occurencesOf("Devon"));
```

```
    assertTrue(names.remove("Devon"));
    assertEquals(0, names.occurencesOf("Devon"));
  }
```

13-5 remove method in the ArrayBag class

```
  // Remove element if it is in this bag, otherwise return false.
  public boolean remove(E element) {
    for (int i = 0; i < n; i++) {
      if (element.equals(data[i])) {
        // Replace with the element at the send
        data[i] = data[n - 1];
        // Reduce the size
        n--;
        return true; // Found--end the search
      }
    }
    // Did not find element "equals" anything in this bag.
    return false;
  }
```

13-6

```
    anObject = aString;    // __√___
    anInteger = aString;   // Can't assign String to Integer
    anObject = anInteger;  // __√___
    anInteger = anObject;  // Can;t assign Obect to Integer
```

13-7 All compile

Chapter 14

Algorithm Analysis

Goals

- Analyze algorithms
- Understand some classic searching and sorting algorithms
- Distinguish runtime order: O(1), O(n), O(n log n), and $O(n^2)$

14.1 Algorithm Analysis

This chapter introduces a way to investigate the efficiency of algorithms. Examples include searching for an element in an array and sorting elements in an array. The ability to determine the efficiency of algorithms allows programmers to better compare them. This helps when choosing a more efficient algorithm when implementing data structures.

An **algorithm** is a set of instructions that can be executed in a finite amount of time to perform some task. Several properties may be considered to determine if one algorithm is better than another. These include the amount of memory needed, ease of implementation, robustness (the ability to properly handle exceptional events), and the relative efficiency of the runtime.

The characteristics of algorithms discussed in this chapter relate to the number of operations required to complete an algorithm. A tool will be introduced for measuring anticipated runtimes to allow comparisons. Since there is usually more than one algorithm to choose from, these tools help programmers answers the question: "Which algorithm can accomplish the task more efficiently?"

Computer scientists often focus on problems related to the efficiency of an algorithm: Does the algorithm accomplish the task fast enough? What happens when the number of elements in the collection grows from one thousand to one million? Is there an algorithm that works better for storing a collection that is searched frequently? There may be two different algorithms that accomplish the same task, but all other

things being equal, one algorithm may take much longer than another when implemented and run on a computer.

Runtimes may be reported in terms of actual time to run on a particular computer. For example, SortAlgorithmOne may require 2.3 seconds to sort 2000 elements while SortAlgorithmTwo requires 5.7 seconds. However, this time comparison does not ensure that SortAlgorithmOne is better than SortAlgorithmTwo. There could be a good implementation of one algorithm and a poor implementation of the other. Or, one computer might have a special hardware feature that SortAlgorithmOne takes advantage of, and without this feature SortAlgorithmOne would not be faster than SortAlgorithmTwo. Thus the goal is to compare algorithms, not programs. By comparing the actual running times of SortAlgorithmOne and SortAlgorithmTwo, programs are being considered—not their algorithms. Nonetheless, it can prove useful to observe the behavior of algorithms by comparing actual runtimes — the amount of time required to perform some operation on a computer. The same tasks accomplished by different algorithms can be shown to differ dramatically, even on very fast computers. Determining how long an algorithm takes to complete is known as algorithm analysis.

Generally, the larger the size of the problem, the longer it takes the algorithm to complete. For example, searching through 100,000 elements requires more operations than searching through 1,000 elements. In the following discussion, the variable **n** will be used to suggest the "*n*umber of things".

We can study algorithms and draw conclusions about how the implementation of the algorithm will behave. For example, there are many sorting algorithms that require roughly n^2 operations to arrange a list into its natural order. Other algorithms can accomplish the same task in $n * \log_2 n$ operations. There can be a large difference in the number of operations needed to complete these two different algorithms when n gets very large.

Some algorithms don't grow with n. For example, if a method performs a few additions and assignment operations, the time required to perform these operations does not change when n increases. These instructions are said to run in *constant time*. The number of operations can be described as a constant function f(n) = k, where k is a constant.

Most algorithms do not run in constant time. Often there will be a loop that executes more operations in relation to the size of the data variable such as searching for an element in a collection, for example. The more elements there are to locate, the longer it can take.

Computer scientists use different notations to characterize the runtime of an algorithm. The three major notations O(n), Ω(n), and Θ(n) are pronounced “big-O”, “big-Omega”, and “big-Theta”, respectively. The big-O measurement represents the upper bound on the runtime of an algorithm; the algorithm will never run slower than the specified time. Big-Omega is symmetric to big-O. It is a lower bound on the running time of an algorithm; the algorithm will never run faster than the specified time. Big-Theta is the tightest bound that can be established for the runtime of an algorithm. It occurs when the big-O and Omega running times are the same, therefore it

is known that the algorithm will never run faster or slower then the time specified. This textbook will introduce and use only big-O.

When using notation like big-O, the concern is the *rate of growth* of the function instead of the precise number of operations. When the size of the problem is small, such as a collection with a small size, the differences between algorithms with different runtimes will not matter. The differences grow substantially when the size grows substantially.

Consider an algorithm that has a cost of $n^2 + 80n + 500$ statements and expressions. The upper bound on the running time is $O(n^2)$ because the larger growth rate function dominates the rest of the terms. The same is true for coefficients and constants. For very small values of n, the coefficient 80 and the constant 500 will have a greater impact on the running time. However, as the size grows, their impact decreases and the highest order takes over. The following table shows the growth rate of all three terms as the size, indicated by n, increases.

Function growth and weight of terms as a percentage of all terms as n increases

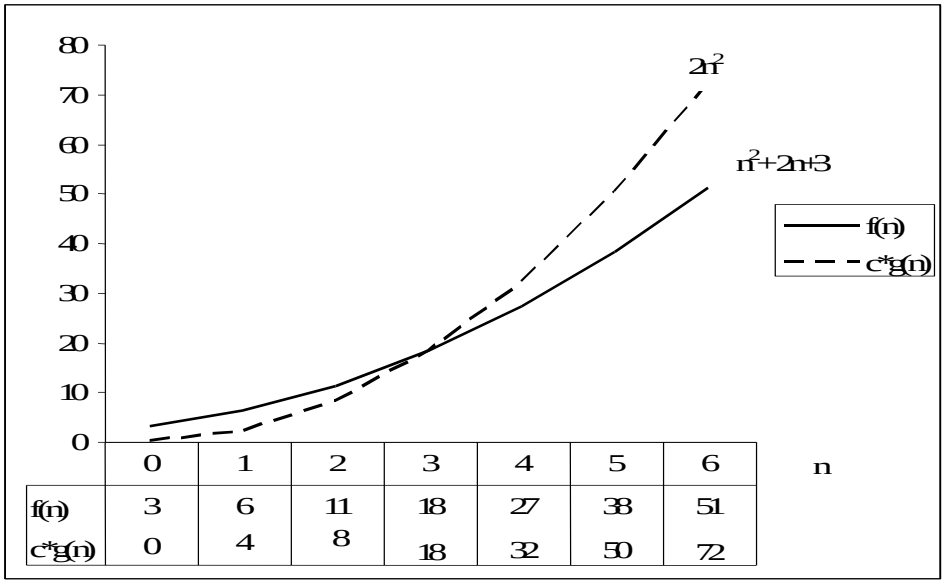
n	f(n)	n^2	80n	500
10	1,400	100 (7%)	800 (57%)	500 (36%)
100	18,500	10,000 (54%)	8,000 (43%)	500 (3%)
1000	1,0805,000	1,000,000 (93%)	80,000 (7%)	500 (0%)
10000	100,800,500	100,000,000 (99%)	800,000 (1%)	500 (0%)

This example shows that the constant 500 has 0% impact (rounded) on the running time as n increases. The weight of this constant term shrinks to near 0%. The term 80n has some impact, but certainly not as much as the term n^2, which raises n to the 2nd power. Asymptotic notation is a measure of runtime complexity when n is large. Big-O ignores constants, coefficients, and lower growth terms.

14.2 Big-O Definition

The big-O notation for algorithm analysis has been introduced with a few examples, but now let's define it a little further. We say that f(n) is O(g(n)) if and only if there exist two positive constants c and N such that $f(n) <= c*g(n)$ for all $n > N$. We say that g(n) is an asymptotic upper bound for f(n). As an example, consider this graph where $f(n) = n^2 + 2n + 3$ and $g(n) = c*n^2$

Show that $f(n) = n^2 + 2n + 3$ is $O(n^2)$



	0	1	2	3	4	5	6
f(n)	3	6	11	18	27	38	51
c*g(n)	0	4	8	18	32	50	72

To fulfill the definition of big-O, we only find constants c and N at the point in the graph where c*g(n) is greater than f(n). In this example, this occurs when c is picked to be 2.0 and N is 4. The above graph shows that if n < N, the function g is at or below the graph of f. In this example, when n ranges from 0 through 2, g(n) < f(n). c*g(n) is equal to f(n) when c is 2 and n is 3 ($2*3^2$ == 18 as does $3^2+2*3+3$). And for all n >= 4, f(n) <= c*g(n). Since g(n) is larger than f(n) when c is 2.0 and N >= 4, it can be said that f(n) is O(g(n)). More specifically, f(n) is $O(n^2)$.

The g(n) part of these charts could be any of the following common big-O expressions that represent the upper bound for the runtime of algorithms:

Big-O expressions and commonly used names

O(1) *constant (an increase in the amount of data (n) has no effect)*

O(log n) *logarithmic (operations increase once each time n doubles)*

O(n) *linear*

O(n log n) *n log n*

O(n^2) *quadratic*

O(n^3) *cubic*

O(2^n) *exponential*

Properties of Big-O

When analyzing algorithms using big-O, there are a few properties that will help to determine the upper bound of the running time of algorithms.

Property 1, coefficients: If f(n) is x * g(n) then f(n) is O(g(n))
This allows the coefficient (x) to be dropped.
Example:
f(n) = 100*g(n)
then f(n) is O(n)

Property 2, sum: If $f_1(n)$ is O(g(n)) and $f_2(n)$ is O(g(n)) then $f_1(n) + f_2(n)$ is O(g(n))
This property is useful when an algorithm contains several loops of the same order.
Example:
$f_1(n)$ is O(n)
$f_2(n)$ is O(n)
then $f_1(n) + f_2(n)$ is O(n) + O(n), which is O(n)

Property 3, sum: If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$. This property works because we are only concerned with the term of highest growth rate.
Example:
$f_1(n)$ is $O(n^2)$
$f_2(n)$ is O(n)
so $f_1(n) + f_2(n) = n^2 + n$, which is $O(n^2)$

Property 4, multiply: If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$ then $f_1(n) * f_2(n)$ is $O(g_1(n) * g_2(n))$. This property is useful for analyzing segments of an algorithm with nested loops.
Example:
$f_1(n)$ is $O(n^2)$
$f_2(n)$ is O(n)
then $f_1(n) * f_2(n)$ is $O(n^2) * O(n)$, which is $O(n^3)$

14.3 Counting Operations

We now consider one technique for analyzing the runtime of algorithms—approximating the number of operations that would execute with algorithms written in Java. This is the *cost* of the code. Let the cost be defined as the total number of operations that would execute in the worst case. The operations we will measure may be assignment statements, messages, and logical expression evaluations, all with a cost of 1. This is very general and does not account for the differences in the number of machine instructions that actually execute. The cost of each line of code is shown in comments. This analysis, although not very exact, is precise enough for this illustration. In the following code, the first three statements are assignments with a cost of 1 each.

Example 1

```
int n = 1000;                    // 1 instruction
int operations = 0;              // 1
int sum = 0;                     // 1
for (int j = 1; j <= n; j++) {   // 1 + (n+1) +  n
  operations++;                  // n
  sum += j;                      // n
}
```

The loop has a logical expression $j \leq n$ that evaluates $n + 1$ times. (The last time it is false.) The increment j++ executes n times. And both statements in the body of the loop execute n times. Therefore the total number of operations $f(n) = 1 + 1 + 1 + 1 + (n+1) + n + n + n = 4n + 5$. To have a runtime O(n), we must find a real constant c and an integer constant N such that $4n + 5 \leq cN$ for all $N > n$. There are an infinite set of values to choose from, for example $c = 6$ and $N = 3$, thus $17 \leq 18$. This is also true for all $N > 3$, such as when $N = 4$ ($21 \leq 24$) and when $N = 5$ ($25 < 30$). A simpler way to determine the runtime is to drop the lower order term (the constant 5) and the coefficient 4.

Example 2

A sequence of statements that does not grow with n is O(1) (constant time). For example, the following algorithm (implemented as Java code) that swaps two array elements has the same runtime for any sized array. $f(n) = 3$, which is O(1).

```
private void swap(String[] array, int left, int right)  {
  String temp = array[left];      // 1
  array[left] = array[right];     // 1
  array [right] = temp;           // 1
}
```

To have a runtime O(1), we must find a real constant c and an integer constant N such that $f(n) = 3 \leq cN$. For example, $c = 2$ and $N = 3$ ($3 \leq 6$).

Example 3

The following code has a total cost of $6n + 3$, which after dropping the coefficient 6 and the constant 3, is O(n).

```
// Print @ for each n
for (int i = 0; i < 2 * n; i++)  // 1 + (2n+1) + 2n
  System.out.print("@");         // 2n+1
```

To have a runtime O(n), we must find a real constant c and an integer constant N such that $f(n) = 2n+1 \leq cN$. For example, $c = 4$ and $N = 3$ ($7 \leq 12$).

Example 4

Algorithms under analysis typically have one or more loops. Instead of considering the comparisons and increments in the loop added to the number of times each instruction inside the body of the loop executes, we can simply consider how often the loop repeats. A few assignments before or after the loop amount to a small constant that can

be dropped. The following loop, which sums all array elements and finds the largest, has a total cost of about 5n + 1. The runtime once again, after dropping the coefficient 5 and the constant 1, is O(n).

```
double sum = 0.0;              // 1
double largest = a[0];         // 1
for (int i = 1; i < n; i++) {  // 1 + n + (n-1)
  sum += a[i];                 // n-1
  if (a[i] > largest)          // n-1
    largest = a[i];            // n-1, worst case: a[i] > largest always
  }
```

Example 5

In this next example, two loops execute some operation n times. The total runtime could be described as O(n) + O(n). However, a property of big O is that the sum of the same orders of magnitude is in fact that order of magnitude (see big-O properties below). So the big-O runtime of this algorithm is O(n) even though there are two individual `for` loops that are O(n).

```
// f(n) = 3n + 5 which is O(n)
// Initialize n array elements to random integers from 0 to n-1
int n = 10;                                              // 1
int[] a = new int[n];                                    // 1
java.util.Random generator = new java.util.Random();     // 1
for (int i = 0; i < n; i++)                              // 2n + 2
  a[i] = generator.nextInt(n);                           // n

// f(n) = 7n + 3 which is O(n)
// Rearrange array so all odd integers in the lower indexes
int indexToPlaceNextOdd = 0;              // 1
for (int j = 0; j < n; j++) {             // 2n + 2
  if (a[j] % 2 == 1) {                    // n: worst case always  odd
    // Swap the current element into
    // the sub array of odd integers
    swap(a, j, indexToPlaceNextOdd);      // n
    indexToPlaceNextOdd++;                // n
  }
}
```

To reinforce that O(n) + O(n) is still O(n), all code above can be counted as f(n)= 10n + 8, which is O(n). To have a runtime O(n), use c = 12 and N = 4 where $10n+8 \leq cN$, or $48 \leq 48$.

Example 6

The runtime of nested loops can be expressed as the product of the loop iterations. For example, the following inner loop executes (n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1 times, which is n/2 operations. The outer loop executes the inner loop n-1 times. The inner loop executes (n-1)*(n/2) twice, which is $n^2 - n$ operations. Add the others to get f(n) = $3n^2+4n-2$. After dropping the coefficient from n^2 and the lower order terms 4n and –2, the runtime is $O(n^2)$.

```
// Rearrange arrays so integers are sorted in ascending order
for (int top = 0; top < n - 1; top++) {        // 2n + 1
  int smallestIndex = top;                     //  n - 1
```

```
    for (int index = top; index < n; index++) {  // (n-1)*(2n)
      if (a[index] < a[smallestIndex])           // (n-1)*(n/2)
        smallestIndex = index;                   // (n-1)*(n/2) at worst
    }
    // Swap smallest to the top index
    swap(a, top, smallestIndex);                 // 3n
  }
```

To have a runtime $O(n^2)$, use c = 4 and N = 4 where $3n^2 + 4n - 2 \leq cN$, or $62 \leq 64$.

Example 7

If there are two or more loops, the longest running loop takes precedence. In the following example, the entire algorithm is $O(n^2) + O(n)$. The maximum of these two orders of magnitudes is $O(n^2)$.

```
int operations = 0;                          // 1
int n = 10000;                               // 1
// The following code runs O(n*n)
for (int j = 0; j < n; j++)                  // 2n+2
  for (int k = 0; k < n; k++)                // n*(2n+2)
    operations++;                            // n*(2n+2)

// The following code runs O(n)
for (int i = 0; i < n; i++)                  // 2n+2
  operations++;                              // n
```

Since $f(n) = 4n^2+9n+6 < cn^2$ for c = 6.05 when N = 5, f(n) is $O(n^2)$.

Tightest Upper Bound

Since big-O notation expresses the notion that the algorithm will take no longer to execute than this measurement, it could be said, for example, that sequential search is $O(n^2)$ or even $O(2^n)$. However, the notation is only useful by stating the runtime as a tight upper bound. The tightest upper bound is the lowest order of magnitude that still satisfies the upper bound. Sequential search is more meaningfully characterized as O(n).

Big-O also helps programmers understand how an algorithm behaves as n increases. With a linear algorithm expressed as O(n), as n doubles, the number of operations doubles. As n triples, the number of operations triples. Sequential search through a list of 10,000 elements takes 10,000 operations in the worst case. Searching through twice as many elements requires twice as many operations. The runtime can be predicted to take approximately twice as long to finish on a computer.

Here are a few algorithms with their big-O runtimes.

- Sequential search (shown later) is O(n)
- Binary search (shown later) is O(log n)
- Many sorting algorithms are $O(n^2)$ — one of these was shown above and will be explained in detail later

- Some faster sort algorithms are O(n log n) — one of these (Quicksort) is described in a later chapter on Recursion.
- Matrix multiplication is $O(n^3)$

Self-Check

14-1 Arrange these functions by order of growth from highest to lowest

$100*n^2$ 1000 2^n $10*n$ n^3 $2*n$

14-2 Which term dominates this function when n gets really big, n^2, 10n, or 100?

$n^2 + 10n + 100$

14-3. When n = 500 in the function above, what percentage of the function is the term?

14-4 Express the tightest upper bound of the following loops in big-O notation.

a)
```
int sum = 0;
int n = 100000;
```

b)
```
int sum = 0;
for (int j = 0; j < n; j++)
  for (int k = 0; k < n; k++)
     sum += j * k;
```

c)
```
for (int j = 0; j < n; j++)
  for (int k = 0; k < n; k++)
    for (int l = 0; l < n; l++)
      sum += j * k * l;
```

d)
```
for (int j = 0; j < n; j++)
  sum++;
for (int j = 0; j < n; j++)
  sum--;
```

e)
```
for (int j = 0; j < n; j++)
  sum += j;
```

f)
```
for (int j = 1; j < n; j *= 2)
  sum += j;
```

14.4 Search Algorithms with Different Big-Os

A significant amount of computer processing time is spent searching. An application might need to find a specific student in the registrar's database. Another application might need to find the occurrences of the string "data structures" on the Internet. When a collection contains many, many elements, some of the typical operations on data structures—such as searching—may become slow. Some algorithms result in programs that run more quickly while other algorithms noticeably slow down an application.

Sequential Search

Consider the following algorithm to search for an element in an indexed collection.

```
sequentially compare all elements, from index 0 to size-1 {
    if searchID equals ID of the object , return a reference to that object
```

}
return null because searchID does not match any elements from index 0..size-1

This algorithm starts by considering the first element in the list. If there is no match, the second element is compared, then the third, up until the last element. If the element being sought is found, the search terminates.

Because the elements are searched one after another, in sequence, this algorithm is called **sequential search**. Now for a concrete example, consider searching an array of BankAccounts (referenced by accountList) for a `BankAccount` with a matching ID.

```
public BankAccount findAccountWithID(String accountID) {
  for (int index = 0; index < mySize; index++) {
    if (accountID.equals(accountList[index].getID()))
      return accountList[index];
  }
  return null;
}
```

In this example f(n) = 3n+2, so sequential search is O(n)

This function describes the worst case. The loop does not always actually execute n times. If the `searchID` equals `accounts[index].getID()`, only one comparison would be necessary. If `searchID` matches the `getID()` of the last element in the array, n comparisons would be necessary—one comparison for each array element. These two extremes are called the best and worst cases of the algorithm. The big-O notation represents the upper bound, or the worst case.

Binary Search

This section considers a search algorithm that has a "better" big-O runtime with a tight upper bound of O(log n). In a moment, you will see an experiment which shows the difference in runtime efficiencies between sequential search and the faster binary search.

The binary search algorithm accomplishes the same task as sequential search, however binary search finds things more quickly. One of its preconditions is that the array must be sorted. Half of the elements can be eliminated from the search every time a comparison is made. This is summarized in the following algorithm:

***Algorithm:* Binary Search, use with sorted collections that can be indexed**

while the element is not found and it still may be in the array {
Determine the position of the element in the middle of the array
If array[middle] equals the search string
return the index
If the element in the middle is not the one being searched for:

remove the half of the sorted array that cannot contain the element
}

Each time the search element is compared to one array element, the binary search effectively eliminates half the remaining array elements from the search. By contrast, the sequential search would only eliminate one element from the search field for each comparison. Assuming an array of strings is sorted in alphabetic order, sequentially searching for `"Ableson"` might not take long since `"Ableson"` is likely to be located at a low index in the array. On the other hand, sequentially searching for `"Zevon"` would take much more time if the array were very big (in the millions). The sequential search algorithm must first compare all names beginning with A through Y before arriving at any names beginning with Z. On the other hand, binary search gets to `"Zevon"` much more quickly by cutting the array in half each time. When `n` is very large, binary search is much faster. The binary search algorithm has the following preconditions:

- The array must be sorted (in ascending order for now).
- The indexes that reference the first and last elements must represent the entire range of meaningful elements.

The element in the middle of the array is referenced by computing the array index that is halfway between the first and last meaningful indexes. This is the average of the two indexes representing the first and last elements in the array. These three indexes are referred to here as `left`, `mid`, and `right`.

```
public BankAccount findAccountWithID(String accountID) {
  int left = 0;
  int right = mySize-1;

  while (left <= right)  {
    int mid = (left + right) / 2;
    if (accountID.equals(accountList[mid].getID()))
      return accountList[mid];
    else if (accountID.compareTo(accountList[mid].getID()) > 0)
      left = mid + 1;
    else
      right = mid - 1;
  }
  return null;  // not found
}
```

As the search begins, one of three things can happen

- The element at the middle index of the array equals `searchString`—the search is complete.
- `accountID` is less than (alphabetically precedes) the middle element. The second half of the array can be eliminated from the search field.
- `accountID` is greater than (alphabetically follows) the middle element. The first half of the array can be eliminated from the search field.

With binary search, the best case is one comparison (when the element is found right away). The worst case occurs when target is not in the array. At each pass, the "live" portion of the array is narrowed to half the previous size.

The binary search algorithm can be more efficient than the sequential search. Whereas sequential search only eliminates one element from the search per comparison, binary search eliminates half of the array elements for each comparison. For example, when n==1024, a binary search eliminates 512 elements from further search for the first comparison, 256 during a second comparison, then 128, 64, 32, 16, 4, 2, and 1.

When n is small, the binary search algorithm does not see a gain in terms of speed. However when n gets large, the difference in the time required to search for an element can make the difference between selling the software and having it unmarketable. Consider how many comparisons are necessary when n grows by powers of two. Each doubling of n would require potentially twice as many loop iterations for sequential search. However, the same doubling of n would only require potentially one more comparison for binary search.

Maximum number of comparisons for two different search algorithms

Power of 2	n	Sequential Search	Binary Search
2^2	4	4	2
2^4	16	16	4
2^8	128	128	8
2^{12}	4,096	4,096	12
2^{24}	16,777,216	16,777,216	24

As n gets very large, sequential search has to do a lot more work. The numbers above represent the maximum number of iterations necessary to search for an element. The difference between 24 comparisons and almost 17 million comparisons is quite dramatic, even on a fast computer. Let us analyze the binary search algorithm by asking, "How fast is Binary Search?"

The best case is when the element being searched for is in the middle—one iteration of the loop. The upper bound occurs when the element being searched for is not in the array. Each time through the loop, the "live" portion of the array is narrowed to half the previous size. The number of elements to consider each time through the loop begins with n elements (the size of the collection) and proceeds like this: n/2, n/4, n/8, ... 1. Each term in this series represents one comparison (one loop iteration). So the question is "How long does it take to get to 1?" This will be the number of times through the loop.

Another way to look at this is to begin to count at 1 and double this count until the number *k* is greater than or equal to n.

1, 2, 4, 8, 16, ... , k >= n or $2^0, 2^1, 2^2, 2^3, 2^4, \ldots, 2^c >= n$

The length of this series is c+1. The number of loop iterations can be stated as “2 to what power *c* is greater than or equal to n?” Here are a few examples:

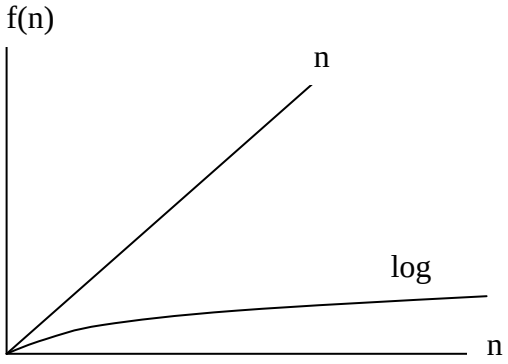
if n is 2, *c* is 1
if n is 4, *c* is 2
if n is 5, *c* is 3
if n is 100, *c* is 7
if n is 1024, *c* is 10
if n is 16,777,216, *c* is 24

In general, as the number of elements to search (n) doubles, binary search requires only one more iteration to effectively remove half of the array elements from the search. The growth of this function is said to be **logarithmic**. Binary search is O(log n). The base of the logarithm (2) is not written, for two reasons:

- The difference between $\log_2 n$ and $\log_3 n$ is a constant factor and constants are not a concern.
- The convention is to use base 2 logarithms.

The following graph illustrates the difference between linear search, which is O(n), and binary search, which takes at most $\log_2 n$ comparisons.

Comparing O(n) to O(log n)



To further illustrate, consider the following experiment: using the same array of objects, search for every element in that array. Do this using both linear search and binary search. This experiment searches for every single list element. There is one O(n) loop that calls the binary search method with an O(log n) loop. Therefore, the time to search for every element using binary search indicates an algorithm that is O(n log n).

Searching for every element in the array (1.3 gig processor, 512 meg RAM):

	Binary Search for every element O(n log n)				Sequential search for every element $O(n^2)$		
n	Binary Search #operations	average operations per search	total time in seconds		Sequential Search #operations	average operations per search	total time in seconds
100	588	5.9	0.00		5,050	50.5	0.0
1,000	9,102	9.1	0.01		500,500	500.5	0.3

10,000	124,750	12.4	0.30		5,0005,000	5,000.5	4.7
100,000	1,581,170	15.8	2.40		5,000,050,000	50,000.5	1,168.6

The time for sequential search also reflects a search for every element in the list. An O(n) loop calls a method that in turn has a loop that executes operations as follows (searching for the first element requires 1 comparison, searching for the second element requires 2 comparisons, and searching for the last two elements requires n-1 and n operations).

1 + 2 + 3 +, ... + n-2 + n-1 + n

This sequence adds up to the sum of the first n integers: n(n+1)/2. So when n is 100, 100(100+1)/2 = 5050 operations are required. The specific number of operations after removing the coefficient 1/2 is n*(n+1). Sequentially searching for every element in a list of size n is $O(n^2)$. Notice the large difference when n is 100,000: 1,168 seconds for the $O(n^2)$ algorithm compared to 4.5 seconds for the O(n log n) operation.

One advantage of sequential search is that it does not require the elements to be in sorted order. Binary search does have this precondition. This should be considered when deciding which searching algorithm to use in a program. If the program is rarely going to search for an item, then the overhead associated with sorting before calling binary search may be too costly. However, if the program is mainly going to be used for searching, then the time expended sorting the list may be made up with repeated searches.

14.5 Example Logarithm Functions

Here are some other applications that help demonstrate how fast things grow if doubled and how quickly something shrinks if repeatedly halved.

1. Guess a Number between 1 and 100

Consider a daily number game that asks you to guess some number in the range of 1 through 1000. You could guess 1, then 2, then 3, all the way up to 1000. You are likely to guess this number in 500 tries, which grows in a linear fashion. Guessing this way for a number from 1 to 10,000 would likely require 10 times as many tries. However, consider what happens if you are told your guess is either too high, too low, or just right

Try the middle (500), you could be right. If it is too high, guess a number that is near the middle of 1..499 (250). If your initial guess was too low, check near middle of 501..1000 (750). You should find the answer in $2^c >= 1000$ tries. Here, c is 10. Using the base 2 logarithm, here is the maximum number of tries needed to guess a number in a growing range.

from 1..250, a maximum of $2^c >= 250$, c == 8
from 1..500, a maximum of $2^c >= 500$, c == 9
from 1..1000, a maximum of $2^c >= 1000$, c == 10

2. Layers of Paper to Reach the Moon

Consider how quickly things grow when doubled. Assuming that an extremely large piece of paper can be cut in half, layered, and cut in half again as often as you wish, how many times do you need to cut and layer until paper thickness reaches the moon? Here are some givens:

4. paper is 0.002 inches thick
5. distance to moon is 240,000 miles
6. 240,000 * 5,280 feet per mile * 12 inches per foot = 152,060,000,000 inches to the moon

3. Storing Integers in Binary Notation

Consider the number of bits required to store a binary number. One bit represents two unique integer values: 0 and 1. Two bits can represent the four integer values 0 through 3 as 00, 01, 10, and 11. Three bits can store the eight unique integer values 0 through 7 as 000, 001, 010, ... 111. Each time one more bit is used twice as many unique values become possible.

4. Payoff for Inventing Chess

It is rumored that the inventor of chess asked the grateful emperor to be paid as follows: 1 grain of rice on the first square, 2 grains of rice on the next, and double the grains on each successive square. The emperor was impressed until later that month he realized that the 2^{64} grains of rice on the 64th square would be enough rice to cover the earth's surface.

14.6 A Sorting Algorithm that is $O(n^2)$

The elements of a collection are often arranged into either an ascending or descending order through a process known as *sorting*. For example, an array of test scores is sorted into descending order by rearranging the tests from highest to lowest. An array of strings sorted in ascending order establishes an alphabetized list ("A" precedes "B", "B" is less than "C"). To sort an array, the elements must be compared. For `int` and `double`, < or > will suffice. For `String`, `Integer` and `Double` objects, the `compareTo` method is used. For example, if one object can be less than another object of the same class, then an array of those objects can be sorted. For example, `("A".compareTo("B") < 0)` is a valid expression. Therefore, an array of primitive `int` values or `String` objects can be sorted. Only a few Java classes define a `compareTo` method. However, any class written by a programmer can add a `compareTo` method, and a collection of those classes of objects can be sorted (i.e., have the class `implement Comparable` and define what `compareTo` means).

There are many sorting algorithms. The relatively simple **selection sort** is presented here, even though others algorithms exist that are more efficient (run faster). The goal is to arrange an array of numbers into ascending order (also known as the natural ordering).

Object Name	Unsorted Array	Sorted Array
data[0]	86.0	62.0
data[1]	92.0	79.0
data[2]	100.0	86.0
data[3]	62.0	92.0
data[4]	79.0	100.0

With the selection sort algorithm, the smallest number must end up in `data[0]` and the smallest in `data[4]`. In general, an array `x` of size `n` is sorted in ascending order if x[j] <= x[j+1] for j = 0 through n-2.

The selection sort begins by locating the smallest element in the array from the first (`data[0]`) through the last (`data[4]`). The smallest element, `data[3]` in this array, will then be swapped with the top element, `data[0]`. Once this is done, the array is sorted at least through the first element.

Place smallest value in the "top" position (index 0)

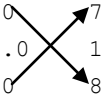
top == 0	Before	After	Sorted
data[0]	86.0	62.0	⇐
data[1]	92.0	92.0	
data[2]	100.0	100.0	
data[3]	62.0	86.0	
data[4]	79.0	79.0	

***Algorithm**:* Find the smallest element in the array and swap with the topmost element

```
top = 0
// At first, assume that the first element is the smallest
indexOfSmallest = top
// Check the rest of the array (data[top+1] through data[n-1])
for index ranging from top+1 through n-1
    if data[index] < data[indexOfSmallest] then
        indexOfSmallest = index
// Place smallest element into the first position and also place the first array
// element into the location where the smallest array element was located
swap data[indexOfSmallest] with data[top]
```

The same algorithm can be used to place the second smallest element into `data[1]`. The second traversal must begin at a new "top" of the array—index 1 rather than 0. This is accomplished by incrementing `top` from 0 to 1. Now a second traversal of the array begins at the second element rather than the first, and the smallest element in the unsorted portion of the array will be swapped with the second element. A second traversal of the array ensures the first two elements are in order. In this example array, `data[4]` is swapped with `data[1]` and the array is sorted through the first two elements:

top == 1	Before	After	Sorted
data[0]	62.0	62.0	⇐
data[1]	92.0	79.0	⇐
data[2]	100.0	100.0	
data[3]	86.0	86.0	
data[4]	79.0	92.0	



This process repeats a total of n-1 times—for all but the last element. It is unnecessary to find the smallest element in a sub array of size 1. This n^{th} element must be the smallest (or equal to the smallest) since all elements preceding the last element are already sorted. An outer loop changes top from 0 through n-2 inclusive.

Algorithm: Selection Sort

```
for top ranging from 0 through n - 2  {
   indexOfSmallest = top
   for index ranging from top + 1 through n - 1      {
       if data[index] < data[indexOfSmallest] then
          indexOfSmallest = index
   }
   swap data[indexOfSmallest] with data[top]
}
```

Self-Check

14-5 What is the runtime of selection sort in Big-O?

14-6 What change(s) would need to be made to the algorithm above to sort an array so it is in descending order?

14-7 What method can be used to compare any two objects (not primitives)

14-8 What is the runtime to reverse all array elements in Big-O?

14-9 What is the runtime to remove an element from a sorted array of ints in Big-O?

Answers to Self-Check Questions

14-1 order of growth, highest to lowest

-1 2^n (2 to the nth power)
-2 n^3
-3 $100*n^2$
-4 10*n
-5 2*n
-6 1000

14-2 n^2 dominates the function

14-3 percentage of the function
n^2 = 98%
10n = 1.96%
100 = 0.0392%

14-4 tightest upper bounds
-a O(1)
-b $O(n^2)$ On the order of n squared
-c $O(n^3)$
-d O(n)
-e O(n)
-f O(log n)

14-5 $O(n^2)$

14-6 change < to >

14-7 compareTo

14-8 run time of reversal = O(n) (can perform 0.5 * n swaps)

14-9 run time for removal = O(n) (may have to shift from 1 up to n-1 elements

Chapter 15

A List ADT

This chapter defines an interface that will represent a List abstract data type. The class that implements this interface uses an array data structure to store the elements. In the next chapter, we will see how this interface can also be implemented using the linked structure shown in the previous chapter.

Goals

- Introduce the List ADT
- Implement an interface
- Have methods throw exceptions and test that the methods do

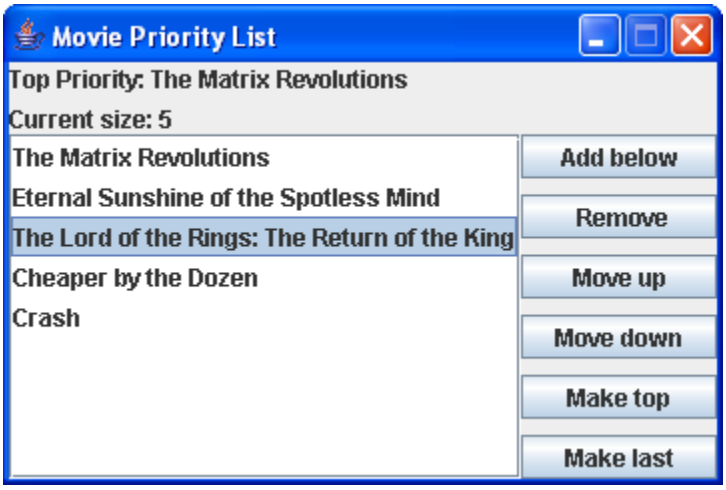
15.1 A List ADT

A **list** is a collection where each element has a specific position—each element has a distinct predecessor (except the first) and a distinct successor (except the last). A list allows access to elements through an index. The list interface presented here supports operations such as the following:

7. add, get, or remove an element at specific location in the list
8. find or remove an element with a particular characteristic

From an *application* point of view, a list may store a collection of elements where the index has some importance. For example, the following interface shows one view of a list that stores a collection of DVDs to order. The DVD at index 0, “The Matrix Revolutions”, has the top priority. The DVD at index 4 has a lower priority than the DVD at index 3. By moving any “to do” item up or down in the

list, users reprioritize what movies to get next. Users are able to add and remove DVDs or rearrange priorities.



From an *implementation* point of view, your applications could simply use an existing Java collection class such as `ArrayList<E>` or `LinkedList<E>`. As is customary in a second level course in computer science, we will be implementing our own, simpler version, which will

- enhance your ability to use arrays and linked structures (required in further study of computing).
- provide an opportunity to further develop programming skills: coding, testing, and debugging.
- help you understand how existing collection classes work, so you can better choose which one to use in programs.

Specifying ADTs as Java Interfaces

To show the inner workings of a collection class (first with an array data structure, and then later with a linked structure), we will have the same interface implemented by two different classes. This interface, shown below, represents one abstract view of a list that was designed to support the goals mentioned above.

The interface specifies that implementing classes must be able to store any type of element through Java generics—`List<E>`, rather than `List`. One alternative to this design decision is to write a List class each time you need a new list of a different type (which could be multiple classes that are almost the same). You could implement a class for each type of the following objects:

```
// Why implement a new class for each type?
StringList stringList = new StringList();
BankAccountList bankAccountList = new BankAccountList();
DateList dateList = new DateList();
```

Another alternative was shown with the `GenericList` class shown in the previous chapter. The method heading that adds an element would use an `Object` parameter and the `get` method to return an element would have an `Object` return type.

```
// Add any reference type of element (no primitive types)
```

```
public void add(Object element);

// Get any reference type of element (no primitive types)
public Object get(int index);
```

Collections of this type require the extra effort of casting when getting an element. If you wanted a collection of primitives, you would have to wrap them. Additionally, these types of collections allow you to add any mix of types. The output below also shows that runtime errors can occur because any reference type can be added as an element. The compiler approves this code, but an exception is thrown at runtime.

```
GenericList list = new GenericList();
list.add("Jody");
list.add(new BankAccount("Kim", 100));

for (int i = 0; i < list.size(); i++) {
  String element = (String) list.get(i); // cast required
  System.out.println(element.toUpperCase());
}
```

Output:

```
JODY
Exception in thread "main" java.lang.ClassCastException: BankAccount
```

The preferred option is to focus on classes that have a type parameter in the heading like this

```
public class OurList<E>  // E is a type parameter
```

Now `E` represents the type of elements to that can be stored in the collection. Generic classes provide the same services as the raw type equivalent with the following advantages:

- require less casting
- can store collections of any type, including primitives (at least give the appearance of)
- generate errors at compile time when they are much easier to deal with
- this approach is used in the new version of Java's collection framework

Generic collections need a type argument at construction to let the compiler know which type E represents. When an `OurList` object is constructed with a `<String>` type argument, every occurrence of `E` in the class will be seen as `String`.

```
// Add a type parameter such as <E> and implement only one class
OurList<String> sl = new OurArrayList<String>();
OurList<BankAccount> bl = new OurArrayList<BankAccount>();
OurList<Integer> dl = new OurArrayList<Integer>();
```

Now an attempt to add a `BankAccount` to a list constructed to only store strings

```
sl.add(0, new BankAccount("Jody", 100));
```

results in this compiletime error:

The method add(int, String) in the type OurList<String> is not applicable for the arguments (int, BankAccount)

15.2 A List ADT Specified as a Java interface

Interface `OurList` specifies a reduced version of Java's `List` interface (7 methods instead of 25). By design, these methods match the methods of the same name found in the two Java classes that implement Java's `List` interface: ArrayList<E> and LinkedList<E>.

```
/**
 * This interface specifies the methods for a generic List ADT.
 * It is designed to be generic so any type element can be stored.
 * It will be implemented with an array in this chapter and then a
 * linked structure in the chapter that follows. These 7 methods
 * are a subset of the 25 methods specified in java.util.List<E>
 */
public interface OurList<E>  {

  // Return the number of elements currently in the list
  public int size();

  // Insert an element at the specified location
  // Precondition: insertIndex >= 0 and insertIndex <= size()
  public void add(int insertIndex, E element) throws IllegalArgumentException;

  // Get the element stored at a specific index
  // Precondition: insertIndex >= 0 and insertIndex < size()
  public E get(int getIndex) throws IllegalArgumentException;

  // Replace the element at a specific index with element
  // Precondition: insertIndex >= 0 and insertIndex < size()
  public void set(int insertIndex, E element) throws IllegalArgumentException;

  // Return a reference to element in the list or null if not found.
  public E find(E search);

  // Remove element specified by removalIndex if it is in range
  // Precondition: insertIndex >= 0 and insertIndex < size()
  public void removeElementAt(int removalIndex) throws IllegalArgumentException;

  // Remove the first occurrence of element and return true or if the
  // element is not found leave the list unchanged and return false
  public boolean remove(E element);
}
```

OurArrayList<E> implements OurList<E>

The following class implements `OurList` using an array as the structure to store elements. The constructor ensures the array has the capacity to store 10 elements. (The capacity can change). Since `n` is initially set to 0, the list is initially empty.

```
public class OurArrayList<E> implements OurList<E> {

 /**
   * A class constant to set the capacity of the array.
   * The storage capacity increases if needed during an add.
   */
  public static final int INIITIAL_CAPACITY = 10;

 /**
   * A class constant to help control thrashing about when adding and
   * removing elements when the capacity must increase or decrease.
```

```
   */
  public static final int GROW_SHRINK_INCREMENT = 20;

  // --Instance variables
  private Object[] data; // Use an array data structure to store elements
  private int n;         // The number of elements (not the capacity)

  /**
   * Construct an empty list with an initial default capacity.
   * This capacity may grow and shrink during add and remove.
   */
  public OurArrayList() {
    data = new Object[INIITIAL_CAPACITY];
    n = 0;
  }
```

Whenever you are making a generic collection, the type parameter (such as <E>) does not appear in the constructor. Since the compiler does not know what the array element type will be in the future, it is declared to be an array of Objects so it can store any reference type.

The initial capacity of OurList object was selected as 10, since this is the same as Java's `ArrayList<E>`. This class does not currently have additional constructors to start with a larger capacity, or a different grow and shrink increment, as does Java's `ArrayList`. Enhancing this class in this manner is left as an exercise.

size

The `size` method returns the number of elements in the list which, when empty, is zero.

```
  public void testSizeWhenEmpty() {
    OurList<String> emptyList = new OurArrayList<String>();
    assertEquals(0, emptyList.size());
  }
```

Because returning an integer does not depend on the number of elements in the collection, the `size` method executes in constant time.

```
  /**
   * Accessing method to determine how many elements are in this list.
   * Runtime: O(1)
   * @returns the number of elements in this list.
   */
  public int size() {
    return n;
  }
```

get

OurList specifies a get method that emulates the array square bracket notation `[]` for getting a reference to a specific index. This implementation of the get method will throw an `IllegalArgumentException` if argument `index` is outside the range of `0` through `size()-1`. Although not specified in the interface, this design decision will cause the correct exception to be thrown in the correct place, even if the index is in the capacity bounds of the array. This avoids returning null or other meaningless data during a “get” when the index is in the range of 0 through `data.length-1` inclusive.

```
  /**
   * Return a reference to the element at the given index.
   * This method acts like an array with [] except an exception
```

```
  * is thrown if index >= size().
  * Runtime: O(1)
  * @returns Reference to object at index if 0 <= index < size().
  * @throws IllegalArgumentException when index<0 or index>=size().
  */
  public E get(int index) throws IllegalArgumentException {
    if (index < 0 || index >= size())
      throw new IllegalArgumentException("" + index);

    return data[index];
  }
```

Exception Handling

When programs run, errors occur. Perhaps an arithmetic expression results in division by zero, or an array subscript is out of bounds, orto read from a file with a name that simply does not exist. Or perhaps, the `get` method receives an argument 5 when the size was 5. These types of errors that occur while a program is running are known as exceptions.

The `get` method throws an exception to let the programmer using the method know that an invalid argument was passed during a message. At that point, the program terminates indicating the file name, the method name, and the line number where the exception was thrown. When `size` is 5 and the argument 5 is passed, the `get` method throws the exception and Java prints this information:

```
java.lang.IllegalArgumentException: 5
at OurArrayListTest.get(OurArrayListTest.java:108)
```

Programmers have at least two options for dealing with these types of errors:

- Ignore the exception and let the program terminate
- Handle the exception

Java allows you to *try* to execute methods that may throw an exception. The code exists in a `try` block—the keyword `try` followed by the code wrapped in a block, `{ }`.

```
try {
    code that may throw an exception when an exception is thrown
}
catch(Exception anException) {
    code that executes only if an exception is thrown from code in the above try block.
}
```

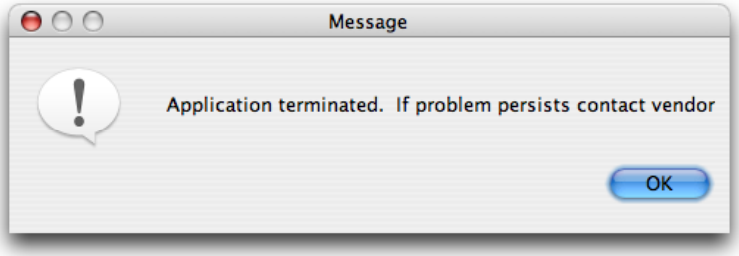
A `try` block must be followed by a `catch` block—the keyword `catch` followed by the anticipated exception as a parameter and code wrapped in a block. The `catch` block contains the code that executes when the code in the `try` block causes an exception to be thrown (or called a method that throws an exception).

Because all exception classes extend the `Exception` class, the type of exception in as the parameter to catch could always be `Exception`. In this case, the `catch` block would catch any type of exception that can be thrown. However, it is recommended that you use the specific exception that is expected to be thrown by the code in the `try` block, such as `IllegalArgumentException`.

The following example will always throw an exception since the list is empty. Any input by the user for `index` will cause the get method to throw an exception.

```
Scanner keyboard = new Scanner(System.in);
OurArrayList<String> list = new OurArrayList<String>();
```

```
int index = keyboard.nextInt();

try {
  String str = list.get(index); // When size==0, get always throws an exception
}
catch (IllegalArgumentException iobe) {
  JOptionPane.showMessageDialog(null, "Application terminated. "
                          + " If problem persists contact vendor");
}
```



If the size were greater than 0, the user input may or may not cause an exception to be thrown.

To successfully handle exceptions, a programmer must know if a method might throw an exception, and if so, the type of exception. This is done through documentation in the method heading.

```
public E get(int index) throws IllegalArgumentException {
```

A programmer has the option to put a call to get in a `try` block or the programmer may call the method without placing it in a `try` block. The option comes from the fact that `IllegalArgumentException` is a `RuntimeException` that needs not be handled. Exceptions that don't need to be handled are called unchecked exceptions. The unchecked exception classes are those that extend `RuntimeException`, plus any `Exception` that you write that also extends `RuntimeException`. The unchecked exceptions include the following types (this is not a complete list):

- `ArithmeticException`
- `ClassCastException`
- `IllegalArgumentException`
- `IllegalArgumentException`
- `NullPointerException`

Other types of exceptions require that the programmer handle them. These are called checked exceptions. There are many checked exceptions when dealing with file input/output and networking that *must* be surrounded by a try catch. For example when using the `Scanner` class to read input from a file, the constructor needs a `java.io.File` object. Because that constructor can throw a `FileNotFoundException`, the Scanner must be constructed in a try bock.

```
Scanner keyboard = new Scanner(System.in);
String fileName = keyboard.nextLine();
Scanner inputFile = null;
try {
  // Throws exception if file with the input name can not be found
  inputFile = new Scanner(new File(filename));
}
catch (FileNotFoundException fnfe) {
  JOptionPane.showMessageDialog(null, "File not found: '" +
                                      fileName + "'");
```

```
}
```

Output assuming the user entered *WrongNameWhoops.data* and that file name does not exist.

Self-Check

15-1 Which of the following code fragments throws an exception?

-a
```
int j = 7 / 0;
```

-b
```
String[] names = new String[5];
names[0] = "Chris";
System.out.println(names[1].toUpperCase());
```

-c
```
String[] names;
names[0] = "Kim";
```

15-2 Write a method that reads and prints all the lines in the file.

Testing that the Method throws the Exception

The `get` method is supposed to throw an exception when the index is out of bounds. To make sure this is happening, the following test method will fail if the `get` method does *not* throw an exception when it is expected:

```
@Test
public void testEasyGetException() {
  OurArrayList<String> list = new OurArrayList<String>();
  try {
    list.get(0); // We want get to throw an exception . . .
    fail();      // Show the red bar only if get did NOT throw the exception
  }
  catch (IllegalArgumentException iobe) {
    // . . . and then skip fail() to execute this empty catch block
  }
}
```

This rather elaborate way of testing—to make sure a method throws an exception without shutting down the program—depends on the fact that the empty catch block will execute rather than the `fail` method. The `fail` method of class TestCase automatically generates a failed assertion. The assertion will fail only when your method does not throw an exception at the correct time.

JUnit now provides an easier technique to ensure a method throws an exception. The `@Test` annotation takes a parameter, which can be the type of the Exception that the code in the test method should throw. The following test method will fail if the `get` method does *not* throw an exception when it is expected:

```
@Test(expected = IllegalArgumentException.class)
public void testEasyGetException() {
  OurArrayList<String> list = new OurArrayList<String>();
```

```
    // We want get to ensure this does throws an exception.
    list.get(0);
  }
```

We will use this shorter technique.

add(int, E)

An element of any type can be inserted into any index as long as it in the range of 0 through size() inclusive. Any element added at 0 is the same as adding it as the first element in the list.

```
  @Test
  public void testAddAndGet() {
    OurList<String> list = new OurArrayList<String>();
    list.add(0, "First");
    list.add(1, "Second");
    list.add(0, "New first");
    assertEquals(3, list.size());
    assertEquals("New first", list.get(0));
    assertEquals("First", list.get(1));
    assertEquals("Second", list.get(2));
  }

  @Test(expected = IllegalArgumentException.class)
  public void testAddThrowsException() {
    OurArrayList<String> list = new OurArrayList<String>();
    list.add(1, "Must start with 0");
  }
```

The `add` method first checks to ensure the parameter `insertIndex` is in the correct range. If it is out of range, the method throws an exception.

```
 /**
   * Place element at insertIndex.
   * Runtime: O(n)
   *
   * @param element The new element to be added to this list
   * @param insertIndex The location to place the new element.
   * @throws IllegalArgumentException if insertIndex is out of range.
   */
  public void add(int insertIndex, E element) throws IllegalArgumentException {
    // Throw exception if insertIndex is not in range
    if (insertIndex < 0 || insertIndex > size())
      throw new IllegalArgumentException("" + insertIndex);

    // Increase the array capacity if necessary
    if (size() == data.length)
      growArray();

    // Slide all elements right to make a hole at insertIndex
    for (int index = size(); index > insertIndex; index--)
      data[index] = data[index - 1];

    // Insert element into the "hole" and increment n.
    data[insertIndex] = element;
    n++;
  }
```

If the index is in range, the method checks if the array is full. If so, it calls the private helper method `growArray` (shown later) to increase the array capacity. A `for` loop then slides the array elements one index to the right to make a "hole" for the new element. Finally, the reference to the new element gets inserted into the array and n (size) increases by +1. Here is a picture of the array after five elements are added with the following code:

```
OurList<String> list = new OurArrayList<String>();
list.add(0, "A");
list.add(1, "B");
list.add(2, "C");
list.add(3, "D");
list.add(4, "E");
```

n: 5
data:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
"A"	"B"	"C"	"D"	"E"	null	null	null	null	null

At this point, an `add(0, "F")` would cause all array elements to slide to the right by one index. This leaves a "hole" at index 0, which is actually an unnecessary reference to the `String` object `"A"` in `data[0]`. This is okay, since this is precisely where the new element `"F"` should be placed.

n: 5
data:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
"A"	**"A"**	**"B"**	**"C"**	**"D"**	**"E"**	null	null	null	null

After storing a reference to `"F"` in `data[0]` and increasing `n`, the instance variables should look like this:

n: **6**
data:

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
"F"	"A"	"B"	"C"	"D"	"E"	null	null	null	null

An `add` operation may require that every reference slide to the bordering array location. If there are 1,000 elements, the loop executes 1,000 assignments. Assuming that an insertion is as likely to be at index 1 as at index n-1 or any other index, the loop will likely average n/2 assignments to make a "hole" for the new element. With the possibility of `growArray` executing O(n), `add`, for all other cases, f(n) = n/2 + n or 1.5n. After dropping the coefficient 1.5, the runtime of `add` would still be O(n). The tightest upper bound is still O(n) even if `growArray` is called, since it too is O(n).

`growArray()`, a private helper method

If there is not enough room to insert the element, the array capacity will increase by sending a `growArray` message before the insertion. `OurArrayList` allocates another `GROW_SHRINK_INCREMENT` number of array locations when an `add` message discovers that there is no more room to add a new element. The `growArray` method changes the instance variable `data` to reference a new array a larger capacity by calling `growArray`. It leaves the original contents (`x[0]` through `x[n-1]`) intact, with the elements in the same indexes as before. With an array implementation, the object should grow and shrink the array at the appropriate times. If the list

frequently changes in size, the `add` and `remove` methods can ensure that not too much memory is wasted at one time.

```
// Make the array have greater capacity to store more elements.
// The original elements are retained in indexes 0..size()-1.
private void growArray() {
 Object[] temp = new Object[data.length + GROW_SHRINK_INCREMENT];
 for (int j = 0; j < size(); j++) {
   temp[j] = data[j];
 }
 data = temp;
 // Reference to temp disappears--that memory will be garbage collected
}
```

When the array data is filled to capacity, the instance variables look like this:

```
n: 10
data (data.length == 10):
```

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
"F"	"A"	"B"	"C"	"D"	"E"	"G"	"H"	"I"	"J"

During the message add(10, "Z"); the `add` method sends a growArray message to itself to make the instance variables look like this:

```
n: 11
data (data.length == 30):
```

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	**[10]**	[11]	[12]	…	…	[28]	**[29]**
"F"	"A"	"B"	"C"	"D"	"E"	"G"	"H"	"I"	"J"	"Z"	null	null	…	…	null	**null**

`toString()`

Although not specified in interface `OurList`, each Java class should implement a `toString` method to provide an easy way to visualize the state of its objects. This `toString` method overrides the Object `toString` method. The output will look just like the output from Java's collection classes. Square brackets surround all elements that are separated by commas.

Although it is not necessary to test `toString` methods, the following test shows the desired return result from `toString`.

```
@Test
public void testToString() {
  OurList<String> list = new OurArrayList<String>();
  assertEquals("[]", list.toString());

  list.add(0, "A");
  assertEquals("[A]", list.toString());

  list.add(1, "B");
  assertEquals("[A, B]", list.toString());

  list.add(0, "1st");
  assertEquals("[1st, A, B]", list.toString());
}
```

Since the toString method concatenates all n elements in the list, the runtime of this method grows with n, and so is O(n).

```
/**
```

```
   * Return a string with all elements in this list.
   * Runtime: O(n)
   * @returns A String that is the concatenation of the toString version
   * of all elements separated by ", " and bracketed with "[" and "]".
   */
   @Override
   public String toString() {
     String result = "[";
     // Concatenate all but the last
     for (int index = 0; index < size() - 1 ; index++) {
       result = result + data[index].toString() + ", ";
     }
     if (size() > 0) { // Avoid placing , after the last element.
       result += data[size()-1];
     }
     result += "]";   // Always concatenate the closing square bracket
     return result;
   }
```

set(int, E)

OurList specifies the set method to emulate the array square bracket notation `[]` for setting an element at a specific index. Like the get method, set will also throw an `IllegalArgumentException` if the `index` is outside the range of `0` through `size()-1`; and for the same reasons. Since the size of the list does not affect the runtime of these two algorithms, `get` and `set` are O(1).

```
  /** Change the element referenced by index in this list.
   * @param index Location where a newElement replaces existing one.
   * @param newElement A reference to the object to be inserted.
   * @throws IllegalArgumentException if index < 0 or index >= size().
   */
  public void set(int index, E newElement) throws IllegalArgumentException {
    if (index < 0 || index >= size())
      throw new IllegalArgumentException("" + index);
    else
      data[index] = newElement;
  }
```

The following test method will generate a failure if the `set` method does not throw an exception when expected:

```
@Test(expected = IllegalArgumentException.class)
  public void testThrowExceptionsOnAnEmptyList() {
    OurList<String> list = new OurArrayList<String>();
    list.set(0, "Cannot change any element in an empty list");
  }
```

Summarize the Behavior of `OurArrayList` so Far

The following test method demonstrates the methods just described. It must be assumed that the other methods of interface `OurList` have already been implemented at least as stubs (correct headings, but only returns) because `OurArrayList` will not compile until it implements all seven method headings of interface `OurList`.

```
  @Test
  public void testOurArrayListAddsGetSetAndToString() {
    OurList<String> list = new OurArrayList<String>();
    list.add(0, "A");             // [A]
    list.add(1, "B");             // [A, B]
```

```
    assertEquals("[A, B]", list.toString());

    // Insert elements at indexes 0, 3, and list.size()
    list.add(0, "1st");             // [1st, A, B]
    list.add(list.size(), "4th");   // [1st, A, B, 4th]
    assertEquals("[1st, A, B, 4th]", list.toString());

    // Change two elements with set
    list.set(1, "2nd");
    list.set(2, "3rd");
    assertEquals("[1st, 2nd, 3rd, 4th]", list.toString());

    // Check every element to ensure get returns the correct value
    assertEquals("1st", list.get(0));
    assertEquals("2nd", list.get(1));
    assertEquals("3rd", list.get(2));
    assertEquals("4th", list.get(3));
}
```

Self-Check

15-3 What is the runtime of the test method above, assuming `assertEquals` is O(1)?

15-4 Which assertEquals will pass: a, b, or c?

```
OurList<String> list = new OurArrayList<String>();
list.add(0, "A");
list.add(0, "B");
list.add(0, "C");
assertEquals("B", list.get(0));  // a.
assertEquals("B", list.get(1));  // b.
assertEquals("B", list.get(2));  // c.
```

find

The `find` method returns a reference to the first element in the list that matches the argument. It uses the `equals` method that is defined for that type. The `find` method returns `null` if the argument does not match any list element, again using the `equals` method for the type of elements being stored. Any class of objects that you store should override the `equals` method such that the state of the objects are compared rather than references.

Searching for a `String` in a list of strings is easy, since the `String` equals method does compare the state of the object. You can simply ask to get a reference to a `String` by supplying the string you seek as an argument.

```
@Test
public void testFindWithStrings() {
  OurList<String> list = new OurArrayList<String>();
  list.add(0, "zero");
  list.add(1, "one");
  list.add(2, "two");
  assertNotNull(list.find("zero"));
  assertNotNull(list.find("one"));
  assertNotNull(list.find("two"));
}
```

A test should also exist to make sure `null` is returned when the string does not exist in the list

```
@Test
public void testFindWhenNotHere() {
  OurList<String> names = new OurArrayList<String>();
```

```
    names.add(0, "Jody");
    names.add(1, "Devon");
    names.add(2, "Nar");
    assertNull(names.find("Not Here"));
  }
```

However, for most other types, searching through an `OurArrayList` object (or an `ArrayList` or `LinkedList` object) requires the creation of a faked temporary object that "`equals`" the object that you wish to query or whose state you wish to modify. Consider the following test that establishes a small list for demonstration purposes. Using a small list of BankAccounts, the following code shows a deposit of 100.00 made to one of the accounts.

```
@Test
  public void testDepositInList() {
    OurList<BankAccount> accountList = new OurArrayList<BankAccount>();
    accountList.add(0, new BankAccount("Joe", 0.00));
    accountList.add(1, new BankAccount("Ali", 1.00));
    accountList.add(2, new BankAccount("Sandeep", 2.00));

    String searchID = "Ali";
    BankAccount searchAccount = new BankAccount(searchID, -999);
    BankAccount ref = accountList.find(searchAccount);
    assertNotNull(ref);
    ref.deposit(100.00);
    // Make sure the correct element was really changed
    ref = accountList.find(searchAccount);
    assertEquals(101.00, ref.getBalance(), 1e-12);
  }
```

The code constructs a "fake" reference (`searchAccount`) to be compared to elements in the list. This temporary instance of `BankAcccount` exists solely for aiding the search process. To make an appropriate temporary search object, the programmer must know how the equals method returns true for this type when the IDs match exactly. (You may need to consult the documentation for `equals` methods of other type.) The temporary search account need only have the ID of the searched-for object—`equals` ignores the balance. They do not need to match the other parts of the real object's state.[4] The constructor uses an initial balance of -999 to emphasize that the other parameter will not be used in the search.

The `find` method uses the sequential search algorithm to search the unsorted elements in the array structure. Therefore it runs O(n).

```
/**
  * Return a reference to target if it "equals" an element, or null if
  * it is not found. Since you will rarely be requesting a reference to
  * an object that you already have a reference to, this method will
  * often be called with a "fake" object that has the correct state to
  * equal the real object being sought. For example
  *    list.find(new JukeboxAccount("SearchID", -999));
  * returns a reference to the first element in this list that has the
  * ID of "SearchID". The other argument -999 is not used in the
  * equals method for the type (BankAccount in this example)
  *
```

[4] This is typical when searching through indexed collections. However, there are better ways to do the same thing. Other collections presented later will map a key to a value. All the programmer needs to worry about is the key, such as an account number or student ID. There is no need to construct a temporary object or worry about how a particular `equals` method works for many different classes of objects.

```
   * Runtime: O(n)
   *
   * @param target The object that will be compared to list elements.
   * @returns Reference to first object that equals target (or null).
   */
   public E find(E target) {
     // Get index of first element that equals target
     for (int index = 0; index < n; index++)  {
       if (target.equals(data[index]))
         return data[index];
     }
     return null; // Did not find target in this list
   }
```

The following test method builds a list of two `BankAccount` objects and asserts that both can be successfully found.

```
  @Test
  public void testFindWithBankAccounts() {
    // Set up a small list of BankAccounts
    OurList<BankAccount> list = new OurArrayList<BankAccount>();
    list.add(0, new BankAccount("zero", 0.00));
    list.add(1, new BankAccount("one", 1.00));

    // Find one
    BankAccount fakedToFind = new BankAccount("zero", -999);
    BankAccount withTheRealBalance = list.find(fakedToFind);

    // The following assertions expect a reference to the real account
    assertNotNull(withTheRealBalance);
    assertEquals("zero", withTheRealBalance.getID());
    assertEquals(0.00, withTheRealBalance.getBalance(), 1e-12);

    // Find the the other
    fakedToFind = new BankAccount("one", +234321.99);
    withTheRealBalance = list.find(fakedToFind);

    // The following assertions expect a reference to the real account
    assertNotNull(withTheRealBalance);
    assertEquals("one", withTheRealBalance.getID());
    assertEquals(1.00, withTheRealBalance.getBalance(), 1e-12);
  }
```

And of course we should make sure the `find` method returns null when the object does not "equals" any element in the list:

```
@Test
  public void testFindWhenElementIsNotThere() {
    OurList<BankAccount> list = new OurArrayList<BankAccount>();
    list.add(0, new BankAccount("zero", 0.00));
    list.add(1, new BankAccount("one", 1.00));
    list.add(2, new BankAccount("two", 2.00));
    BankAccount fakedToFind = new BankAccount("Not Here", 0.00);
    // The following assertions expect a reference to the real account
    assertNull(list.find(fakedToFind));
  }
```

1. removeElementAt(int) **and** remove(E)

The `removeElementAt` method will first check to make sure the removal `index` is in range. Instead of sliding elements to the right to make a hole as in `add`, `removeElementAt` will slide all successive

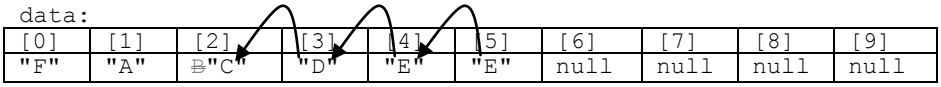
elements one index to the left. The first loop iteration destroys the reference to the element to be removed. The loop continues until the last element has been moved to the old `size()-1`. Then `n` decreases by 1. Here is a picture of an array based list object when it has six elements (`"F"` is first, and `"E"` is last):

`n: 6`
`data:`

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
"F"	"A"	"B"	"C"	"D"	"E"	null	null	null	null

The message `removeElementAt(2)` ("B") would shift all elements from index 2 through size-1 one location to the left. In this case, there are now two references to a `String` object with the value "E".

`n: 5`
`data:`



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
"F"	"A"	~~"B"~~ "C"	"D"	"E"	"E"	null	null	null	null

The unnecessary reference to "E" in `data[5]` could be set to `null`, but it is not necessary. Since there still is a reference to the String object "E", the object would remain anyway. The "extra" reference to "E" (in data[5] above) would effectively be removed from the list simply by decreasing `n` by 1.

The remove method could now use the tested `removeElement`. Here is one algorithm for `remove` that will be called with a message such as `list.remove("B")`.

Algorithm to remove an element that "equals" target

```
index = indexOf(target)
if (index >= 0)
  removeElementAt(index)
```

In this algorithm, the remove method first asks the existing `indexOf` method for the index of the object to be removed. If the object is not found, the collection remains unchanged. If the element is in the list, a `removeElementAt` message removes the element from that index in the list.

Whereas the `add` method of `OurArrayList` increased the capacity of the array instance variable `data`, a similar method should decrease the capacity when appropriate.

Self-Check

15-5 What is the tightest Big O runtime of the `removeElementAt` algorithm?

15-6 What is the tightest Big O runtime of the `remove` operation?

15-7 What would be the tightest Big O runtime of a `removeAll` algorithm that removes the element in index 0 as long as there is an element? Here is the code that accomplishes this:

```
public void removeAll() {
   while(! list.isEmpty())
     list.removeElementAt(0);
}
```

15-8 What would be the runtime of a new `removeAll` method that effectively removes all elements like this:

```
public void removeAll() {
   data = new Object[INIITIAL_CAPACITY];
   n = 0;
}
```

15-9 Which algorithm would be preferred if you assume someone will use `ArrayList` objects with tens of thousands of elements: the algorithm that is O(n) or the algorithm that is $O(n^2)$?

15-10 Write the `removeElementAt` method as if it were in class `OurArrayList`. Make sure the method throws an `IllegalArgumentException` if the index is not in the correct range.

```
public void removeElementAt(int index) {
```

Answers to Self-Check Questions

15-1 which throws an exception? a, b, and c

-a- / by 0

-b- NullPointerException because `names[1]` is `null`

-c- No runtime error, but this is a compiletime error because names is not initialized

15-2 read in from a file

```
public void readAndPrint(String fileName) {
  Scanner inputFile = null; // To avoid compiletime error later
  try {
    inputFile = new Scanner(new File(fileName));
  }
  catch (FileNotFoundException e) {
    e.printStackTrace();
  }

  while (inputFile.hasNextLine()) {
     System.out.println(inputFile.nextLine());
  }
}
```

15-3 O(n) since add is a linear operation.

15-4 b. The list has [C, B, A]

15-5 removeElementAt is O(n);

15-6 remove is O(n)

15-7 $O(n^2)$ removeAll has an O(n) operation preformed n/2 times

15-8 O(1) Java initializes all 20 elements of data, but n is not a factor

15-9 most likely: O(n)

15-10

```
public void removeElementAt(int removalIndex)
                                          throws IllegalArgumentException {
   if (removalIndex < 0 || removalIndex >= size)
     throw new IllegalArgumentException();

   for(int i = removalIndex; i < size-1; i++)
     elements[i] = elements[i + 1];
  }
```

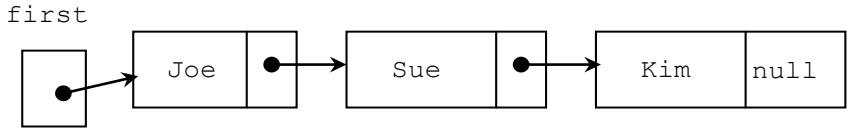
Chapter 16

Linked Structures

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

16.1 A Linked Structure

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



A linked structure with three nodes

Each node is an object with two instance variables:

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

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

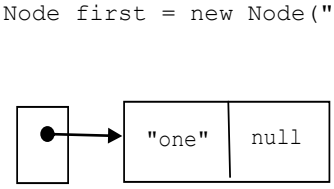
A search for an element begins at the node referenced by the external reference `first`. The second node can be reached through the link from the first node. Any node can be referenced in this sequential fashion. The search stops at the null terminator, which indicates the end. These nodes may be located anywhere in available memory. The elements are not necessarily contiguous memory locations as with arrays. Interface `OurList` will now be implemented using many instances of the private inner class named `Node`.

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

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

  // TBA: OurLinkedList instance variables and methods

} // end class OurLinkedList
```

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

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

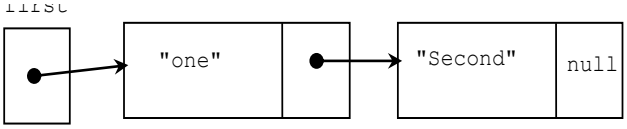
```
// Build the first node and keep a reference to this first node
Node first = new Node("one");
```




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

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

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



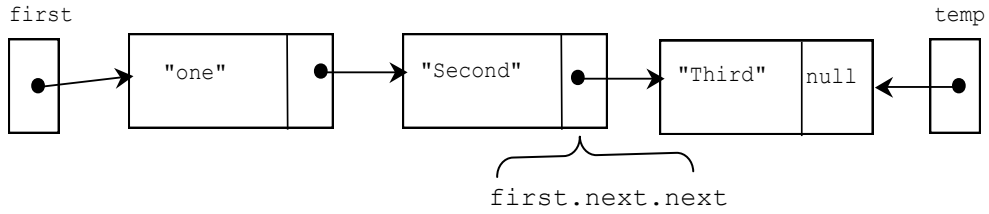
The code above directly assigned a reference to the next instance variable. This unusual direct reference to a private instance variable makes the implementation of `OurLinkedList` than having a

separate class as some textbooks use. Since `Node` is intimately tied to this linked structure — and it has been made an inner class — you will see many permitted assignments to both of `Node`'s private instance variables, `data` and `next`.

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

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

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

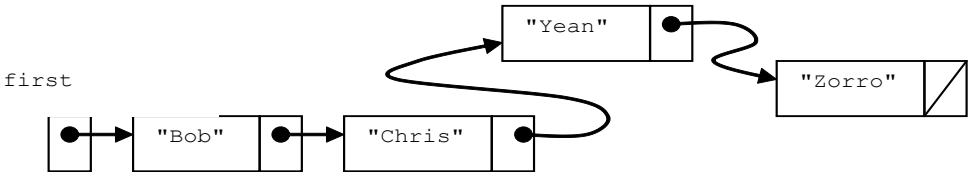


The `Node` reference variable named `first` is not an internal part of the linked structure. The purpose of `first` is to find the beginning of the list so algorithms can find an insertion point for a new element, for example.

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

Self-Check

Use this linked structure to answer the questions that follow.



16-1 What is the value of `first.data`?

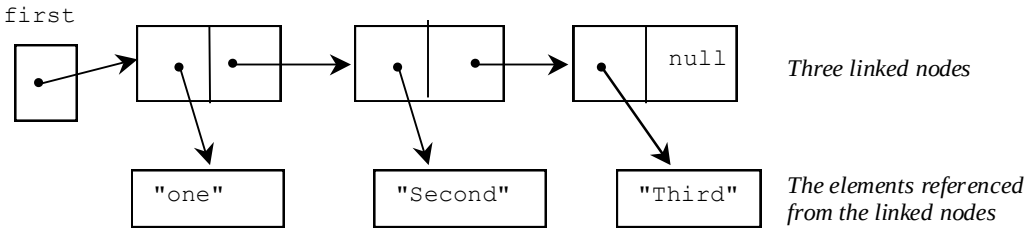
16-2 What is the value of `first.next.data`?

16-3 What is the value of `first.next.next.next.data`?

16-4 What is the value of `first.next.next.next`?

Each node stores a reference to the element

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



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

Traversing a Linked Structure

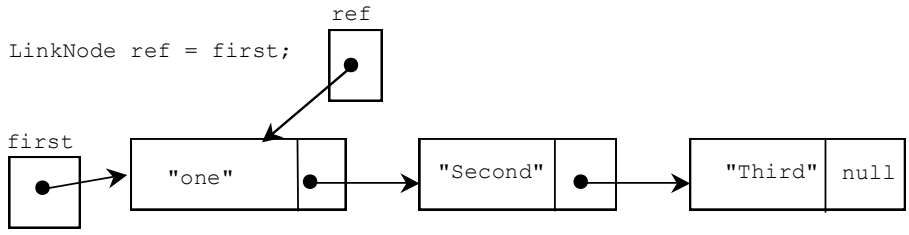
Elements in a linked structure can be accessed in a sequential manner. Analogous to a changing `int` subscript to reference all elements in an array, a changing `Node` variable can reference all elements in a singly linked structure. In the following `for` loop, the `Node` reference begins at the first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

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

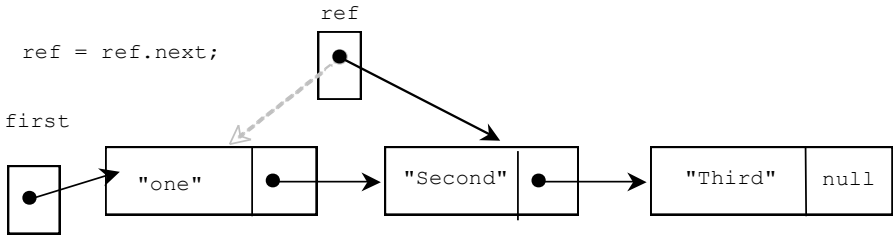
Output

```
one
Second
Third
```

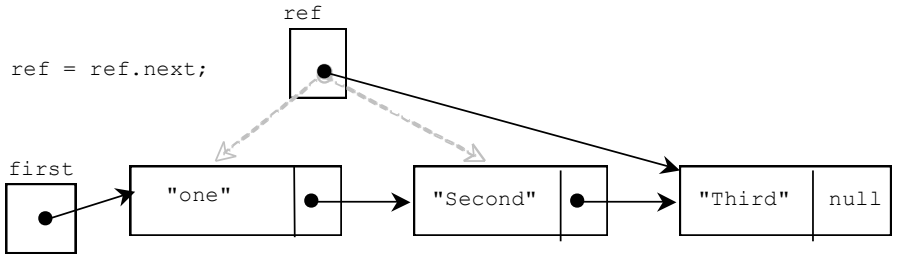
When the loop begins, `first` is not `null`, thus neither is `ref`. The `Node` object `ref` refers to the first node.



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



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



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

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

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

This traversal represents a major difference between a linked structure and an array. With an array, subscript `[2]` directly references the third element. This random access is very quick and it takes just one step. With a linked structure, you must often use sequential access by beginning at the first element and visiting all the nodes that precede the element you are trying to access. This can make a difference in the runtime of certain algorithms and drive the decision of which storage mechanism to use.

16.2 Implement `OurList` using a Linked Structure

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

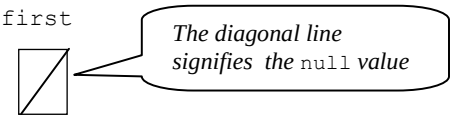
This `OurLinkedList` class uses an inner Node class with two additional constructors (their use will be shown later). It also needs the instance variable first to mark the beginning of the linked structure.

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

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

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

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

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

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

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

  // Instance variables for OurLinkedList
  private Node first;

  private int size;

  // Construct an empty list with size 0
  public OurLinkedList() {
    first = null;
    size = 0;
  }
  // more to come …
}
```

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

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

Picture of an empty list:



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

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

Adding Elements to a Linked Structure

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

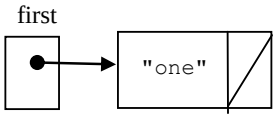
- Inserting an element at the beginning of the list
- Inserting an element at the end of the list
- Inserting an element anywhere in the list at a given position

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

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

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

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




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

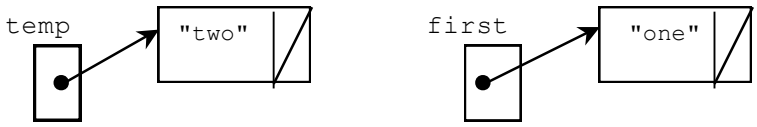
When the list is not empty, the algorithm must still make the insertion at the beginning; `first` must still refer to the new first element. You must also take care of linking the new element to the rest of the list. Otherwise, you lose the entire list! Consider adding a new first element (to a list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

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

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

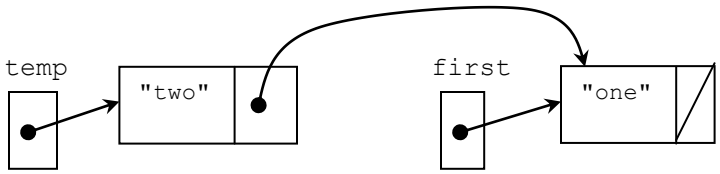
```
  else {
    // the list is NOT empty
    Node temp = new Node(); // data and next are null
    temp.data = element;    // Store reference to element
```

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



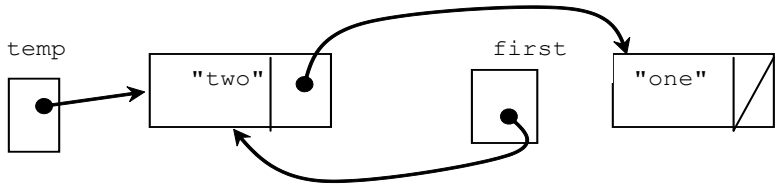
The following code links the node that is about to become the new `first` so that it refers to the element that is about to become the second element in the linked structure.

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

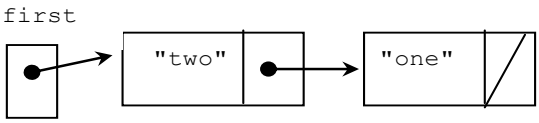


Now move `first` to refer to the `Node` object referenced by `first` and increment `size`.

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



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



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

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

Self-Check

16-5 Draw a picture of memory after each of the following sets of code executes:

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

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

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

`addFirst` again

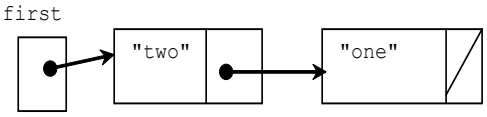
The `addFirst` method used an `if…else` statement to check if the reference to the beginning of the list needed to be changed. Then several other steps were required. Now imagine changing the `addFirst` method using the two-argument constructor of the `Node` class.

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

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

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

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



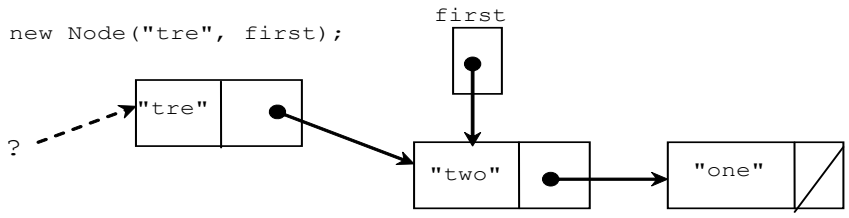
The following initialization executes in `addFirst`:

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

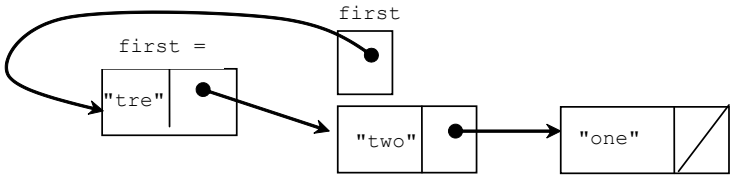
This invokes the two-argument constructor of `class Node`:

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

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

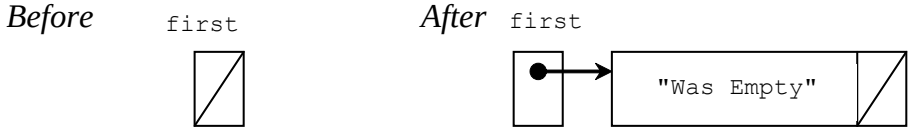


Then after the construction is finished, the reference to the new `Node` is assigned to `first`. Now `first` refers to the new `Node` object. The element `"tre"` is now at index 0.



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

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



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

`get`

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

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

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

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

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

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

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

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

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

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

```
@Test
public void testAddFirstAndGet() {
  OurLinkedList<String> list = new OurLinkedList<String>();
  list.addFirst("A");
  list.addFirst("B");
  list.addFirst("C");

  // Assert that all three can be retrieved from the expected index
  assertEquals("C", list.get(0));
  assertEquals("B", list.get(1));
  assertEquals("A", list.get(2));
}
```

Self-Check

16-6 Which one of the following assertions would fail: a, b, c, or d?

```
OurLinkedList<String> list = new OurLinkedList<String>();
list.addFirst("El");
list.addFirst("Li");
list.addFirst("Jo");
list.addFirst("Cy");
```

```
      assertEquals("El", list.get(3));  // a.
      assertEquals("Li", list.get(2));  // b.
      assertEquals("JO", list.get(1));  // c.
      assertEquals("Cy", list.get(0));  // d.
```

A `toString` method

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

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

```
/**
  * Return a string with all elements in this list.
  * @returns One String that is the concatenation of the toString version of all
  * elements in this list separated by ", " and bracketed with "[" and "]".
  */
  public String toString() {
    String result = "[";

    if (! this.isEmpty()) { // There is at least one element
      // Concatenate all elements except the last
      Node ref = first;
      while (ref.next != null) {
        // Concatenate the toString version of each element
        result = result + ref.data.toString() + ", ";

        // Bring loop closer to termination
        ref = ref.next;
      }
      // Concatenate the last element (if size > 0) but without ", "
      result += ref.data.toString();
    }
    // Always concatenate the closing square bracket
    result += "]";
    return result;
  }
```

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

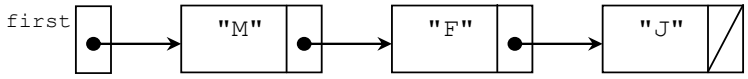
Modified versions of the `for` loop traversal will be used to insert an element into a linked list at a specific index, find a specific element, and remove elements.

The `add` Method

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

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

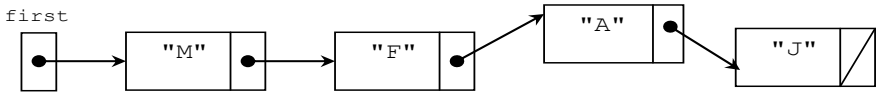
The linked structure generated by the code above would look like this:



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

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

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



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

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

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

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

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

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

```
    if (insertIndex < 0 || insertIndex > this.size())
      throw new IllegalArgumentException("" + insertIndex);
```

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

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

Output

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

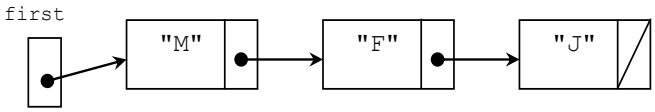
If `insertIndex` is in range, the special case to watch for is if the `insertAtIndex` equals 0. This is the one case when the external reference `first` must be adjusted.

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

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

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

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



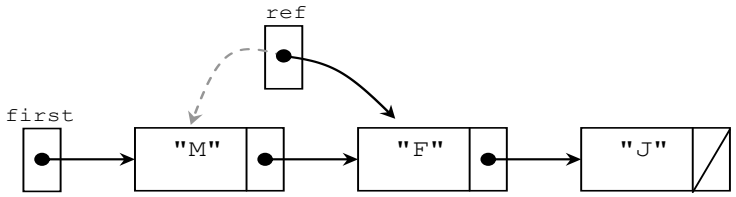
This message inserts a new element at index 2, after `"F"` and before `"J"`.

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

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

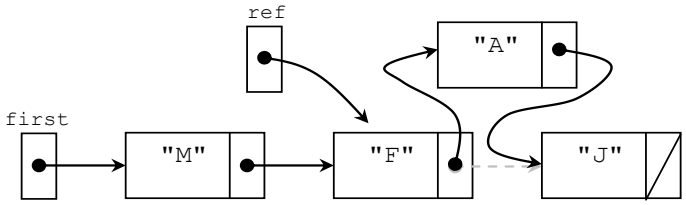
```
else {
  Node ref = first;
  for (int index = 1; index < insertIndex; index++) {
    // Stop when ref refers to the node before the insertion point
    ref = ref.next;
  }
  ref.next = new Node(element, ref.next);
}
```

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



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

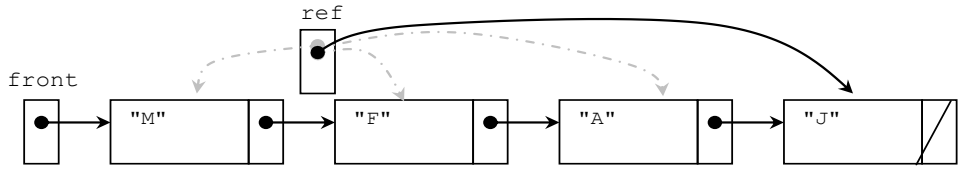
```
ref.next = new Node(element, ref.next);
```



Inserting at index 2

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

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

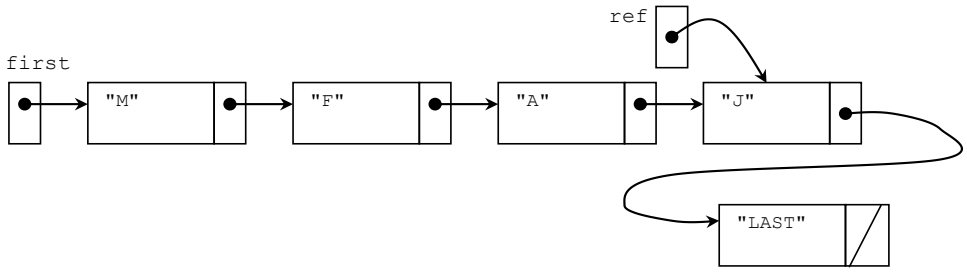


If the list has 1,000 elements, this loop requires 999 (or in general n-1) operations.

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

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

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



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

Self-Check

16-7 Which of the `add` messages (there may be more than one) would throw an exception when sent immediately after the message `list.add(0, 4);`?

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

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

addLast

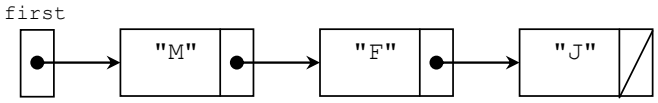
The `addLast` method is easily implemented in terms of `add`. It could have implemented the same algorithm separately, however it is considered good design to avoid repeated code and use an existing method if possible.

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

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

Removing from a Specific Location: `removeElementAt`

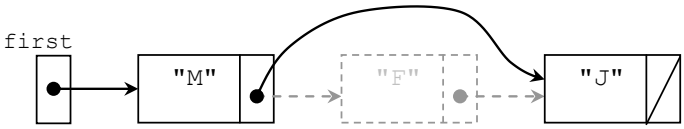
Suppose a linked list has these three elements:



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

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

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

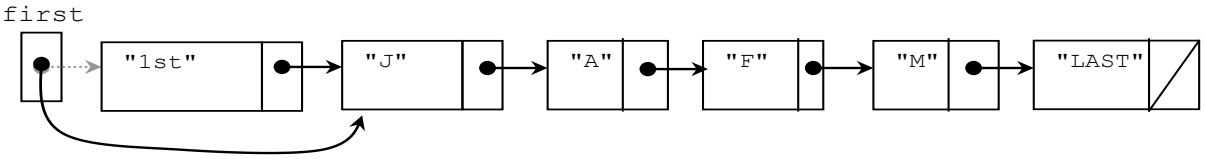


Assuming the index of the element to be removed is in the correct range of `0` through `size()-1`, the following algorithm should work with the current implementation:

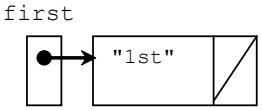
```
if removal index is out of range
    throw an exception
else if the removal is the node at the first
  change first to refer to the second element  (or make the list empty if size()==1)
else  {
  Get a reference to the node before the node to be removed
  Send the link around the node to be removed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

A check is first made to avoid removing elements that do not exist, including removing index 0 from an empty list. Next up is checking for the special case of removing the first node at index 0 ("one" in the structure below). Simply send `first.next` "around" the first element so it references the second element. The following assignment updates the external reference first to refer to the next element.

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

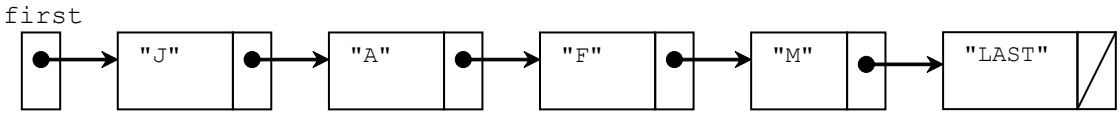


This same assignment will also work when there is only one element in the list.



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

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

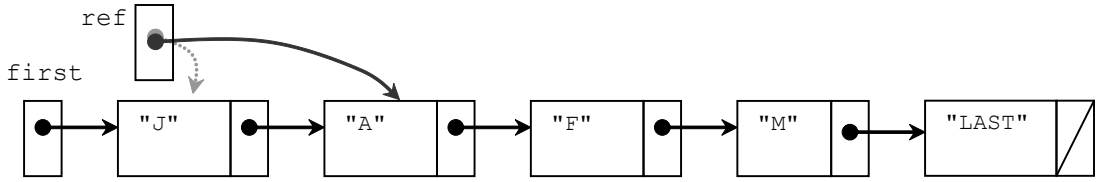


The following assignment has the `Node` variable `ref` refer to the same node as `first`:

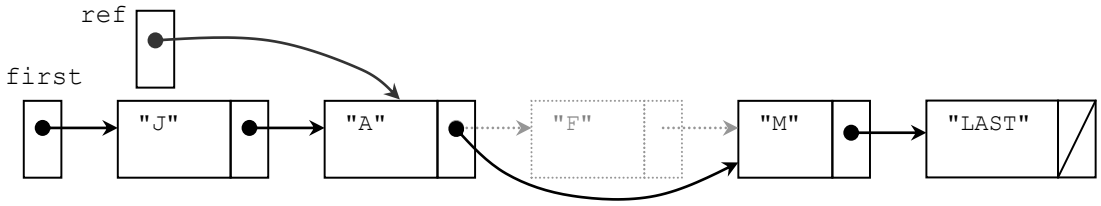
```
Node ref = first;
```

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

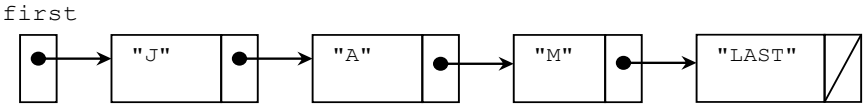
```
for (int index = 1; index < removalIndex; index++) // 1 iteration only
   ref = ref.next;
```



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



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



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

Deleting an element from a Linked List: `remove`

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

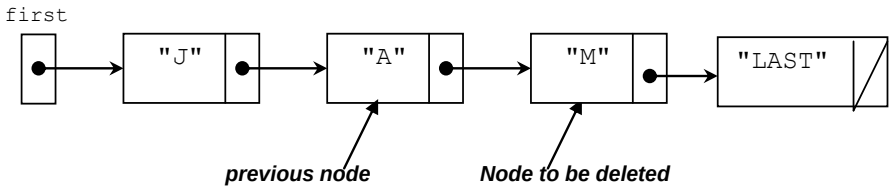
1. Deleting the first element from the list (a special case again)
2. Deleting an interior node from the list (including the last node)

When deleting the first node from a linked list, care must be taken to ensure that the rest of the list is not destroyed. The adjustment to `first` is again necessary so that all the other methods work (and the object is not in a corrupt state). This can be accomplished by shifting the reference value of `first` to the second element of the list (or to `null` if there is only one element). One assignment will do this:

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

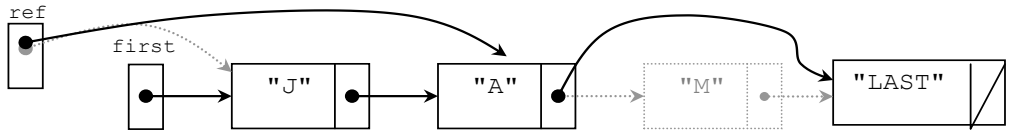
Now consider removing a specific element that may or may not be stored in an interior node. As with `removeElementAt`, the code will look to place a reference to the node just *before* the node to be removed. So to remove `"M"`, the link to the node before M is needed. This is the node with `"A"`.

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



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

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



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

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

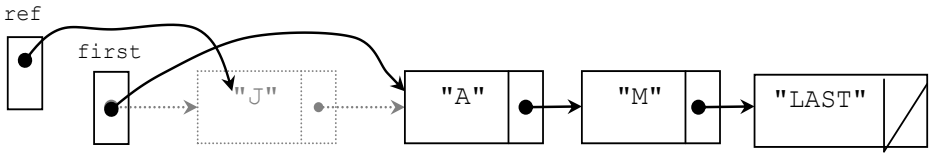
The trick to solving this problem is comparing the data that is one node ahead. Then you must make a reference to the node before the found element (assuming it exists in the list). The following code does just that. It removes the first occurrence of the `objectToRemove` found in the linked list. It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

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

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

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

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



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

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

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

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

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

Self-Check

16-8 What can happen if the subexpressions in the loop test above are reversed?

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

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

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

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

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

Self-Check

16-9 In the space provided, write the expected value that would make the assertions pass:

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

16-10 What must you take into consideration when executing the following code?

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

14.3 When to use Linked Structures

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

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

The linked list implementation also may be more time efficient during inserts and removes. With an array, removing the first element required n assignments. Removing from a linked list requires only one assignment. Removing an internal node may also run a bit faster for a linked implementation, since the worst case rarely occurs. With an array, the worst case always occurs—n operations are

needed no matter which element is being removed. With a linked list, it may be more like n/2 operations.

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

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

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

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

When not to use Linked Structures

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

A rather specific time to avoid linked structures (or any dynamic memory allocations) is when building software for control systems on aircraft. The United States Federal Aviation Association (FAA) does not allow it because it's not safe to have a airplane system run out of memory in flight. The code must work with fixed arrays. All airline control code is carefully reviewed to ensure that allocating memory at runtime is not present. With Java, this would mean there could never be any existence of `new`.

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

	`OurArrayList`	`OurLinkedList`
get and set	O(1)	O(n)
remove removeElementAt	O(n)	O(n)
find[5]	O(n)	O(n)
add(int index, Object el)	O(n)	O(n)
size[6]	O(1)	O(n)
addFirst	O(n)	O(1)
addLast[7]	O(1)	O(n)

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

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

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

Self-Check

16-11 Suppose you needed to organize a collection of student information for your school's administration. There are approximately 8,000 students in the university with no significant growth expected in the coming years. You expect several hundred lookups on the collection everyday. You have only two data structures to store the data, and array and a linked structure. Which would you use? Explain.

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

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

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

Answers to Self-Checks

16-1 first.data.equals("Bob")

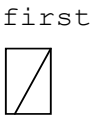
16-2 first.next.data ("Chris");

16-3 first.next.next.next.data.equals("Zorro");

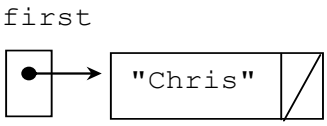
16-4 first.next.next.next refers to a Node with a reference to "Zorro" and null in its next field.

16-5 drawing of memory

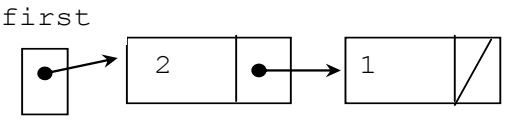
a.



b.



c.



16-6 c would fail ("JO" should be "Jo")

16-7 which would throw an exception
-a- IndexOutOfBoundsException
-c- the largest valid index is currently 4
-d- Okay since the largest index can be the size, which is 4 in this case

16-8 if switched, ref would move one Node too far and cause a NullPointerException

16-9 assertions - listed in correct order

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


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

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

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

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

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

16-10 Whether or not a node actually exists at current.next. It could be null.

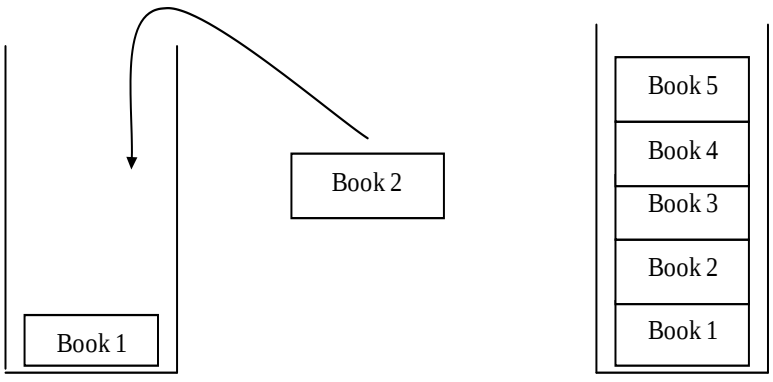
16-11 An array so the more efficient binary search could be used rather than the sequential search necessary with a linked structure.

Chapter 17
Stacks and Queues

17.1 Stacks

The stack abstract data type allows access to only one element—the one most recently added. This location is referred to as the top of the stack.

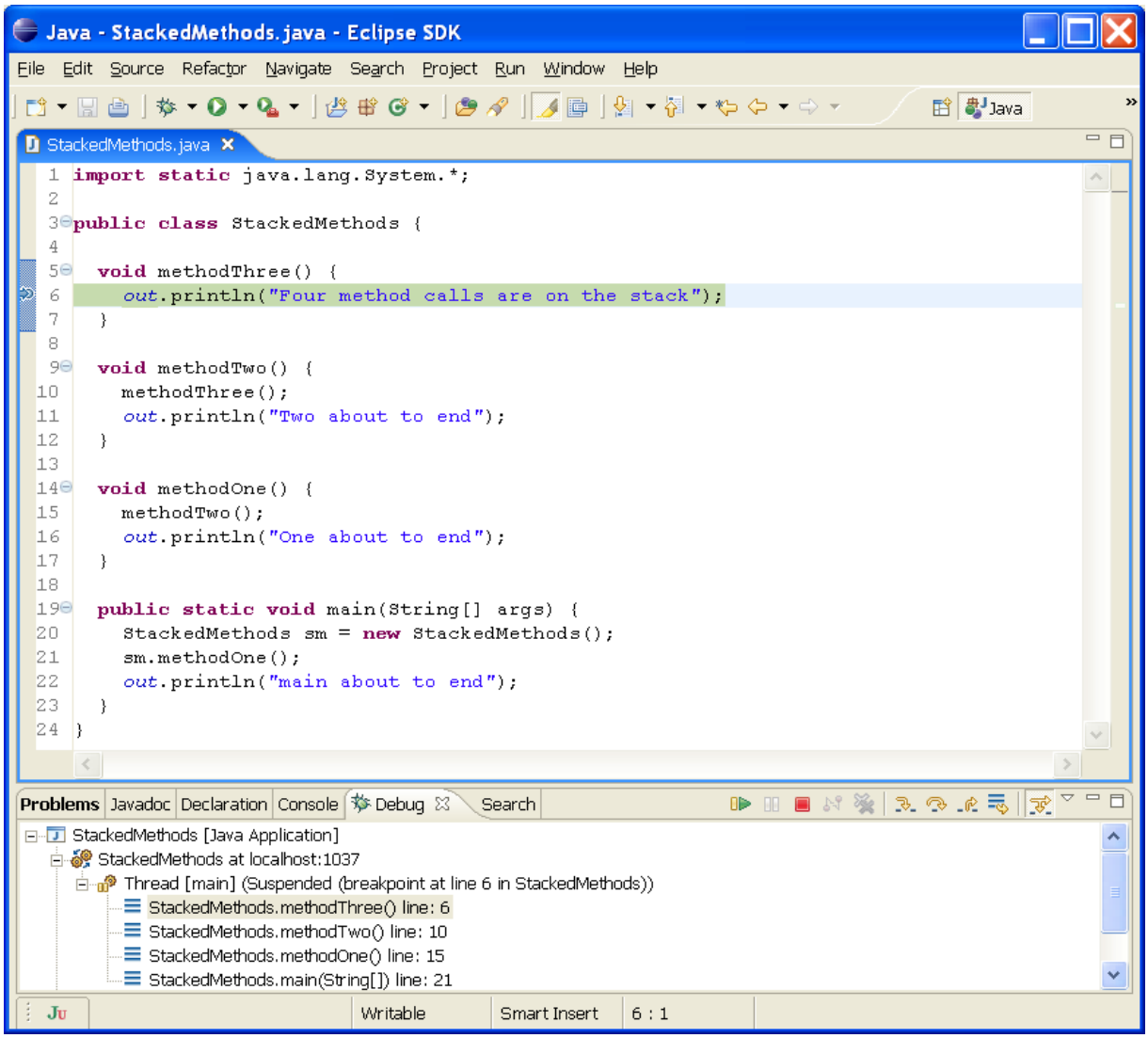
Consider how a stack of books might be placed into and removed from a cardboard box, assuming you can only move one book at a time. The most readily available book is at the top of the stack. For example, if you add two books—Book 1 and then Book 2—into the box, the most recently added book (Book 2) will be at the top of the stack. If you continue to stack books on top of one another until there are five, Book 5 will be at the top of the stack. To get to the least recently added book (Book 1), you first remove the topmost four books (Book 5, Book 4, Book 3, Book 2) — one at a time. Then the top of the stack would be Book 1 again.



Stack elements are added and removed in a last in first out (LIFO) manner. The most recent element added to the collection will be the first element to be removed from the collection. Sometimes, the

only data that is readily needed is the most recently accessed one. The other elements, if needed later, will be in the reverse order of when they were pushed. Many real world examples of a stack exist. In a physical sense, there are stacks of books, stacks of cafeteria trays, and stacks of paper in a printer's paper tray. The sheet of paper on the top is the one that will get used next by the printer.
For example, a stack maintains the order of method calls in a program. If `main` calls `function1`, that method calls `function2`, which in turn calls `function3`. Where does the program control go to when `function3` is finished? After `function3` completes, it is removed from the stack as the most recently added method. Control then returns to the method that is at the new top of the stack — `function2`.

Here is a view of the stack of function calls shown in a thread named `main`. This environment (Eclipse) shows the first method (`main`) at the bottom of the stack. `main` will also be the last method popped as the program finishes — the *first* method called is the *last* one to execute. At all other times, the method on the top of the stack is executing. When a method finishes, it can be removed and the method that called it will be the next one to be removed form the stack of method calls.



The program output indicates the last in first out (or first in last out) nature of stacks:

```
Four method calls are on the stack
Two about to end
One about to end
main about to end
```

Another computer-based example of a stack occurs when a compiler checks the syntax of a program. For example, some compilers first check to make sure that `[ ]`, `{ }`, and ( ) are balanced properly. Thus, in a Java `class`, the final `}` should match the opening `{`. Some compilers do this type of symbol balance checking first (before other syntax is checked) because incorrect matching could otherwise lead to numerous error messages that are not really errors. A stack is a natural data structure that allows the compiler to match up such opening and closing symbols (an algorithm will be discussed in detail later).

A Stack Interface to capture the ADT

Here are the operations usually associated with a stack. (As shown later, others may exist):

- `push` place a new element at the "top" of the stack
- `pop` remove the top element and return a reference to the top element
- `isEmpty` return `true` if there are no elements on the stack
- `peek` return a reference to the element at the top of the stack

Programmers will sometimes add operations and/or use different names. For example, in the past, Sun programmers working on Java collection classes have used the name `empty` rather than `isEmpty`. Also, some programmers write their stack class with a `pop` method that does not return a reference to the element at the top of the stack. Our `pop` method will modify and access the state of stack during the same message.

Again, a Java interface helps specify Stack as an abstract data type. For the discussion of how a stack behaves, consider that `LinkedStack` (a collection class) implements the `OurStack` interface, which is an ADT specification in Java.

```
import java.util.EmptyStackException;

public interface OurStack<E> {
/** Check if the stack is empty to help avoid popping an empty stack.
  * @returns true if there are zero elements in this stack.
  */
  public boolean isEmpty();

/** Put element on "top" of this Stack object.
  * @param The new element to be placed at the top of this stack.
  */
  public void push(E element);

/** Return reference to the element at the top of this stack.
  * @returns A reference to the top element.
  * @throws EmptyStackException if the stack is empty.
  */
  public E peek() throws EmptyStackException;

/** Remove element at top of stack and return a reference to it.
  * @returns A reference to the most recently pushed element.
```

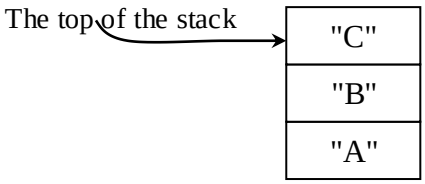
```
  * @throws EmptyStackException if the stack is empty.
  */
  public E pop() throws EmptyStackException;
}
```

You might need a stack of integers, or a stack of string values, or a stack of some new class of `Token` objects (pieces of source code). One solution would be to write and test a different stack class for each new class of object or primitive value that you want to store. This is a good reason for developing an alternate solution— a generic stack.

The interface to be implemented specifies the operations for a stack class. It represents the *abstract* specification. There is no particular data storage mentioned and there is no code in the methods. The type parameter `<E>` and return types `E` indicate that the objects of the implementing class will store any type of element. For example, `push` takes an `E` parameter while `peek` and `pop` return an `E` reference.

The following code demonstrates the behavior of the stack class assuming it is implemented by a class named LinkedStack.

```
// Construct an empty stack that can store any type of element
OurStack stackOfStrings<String> = new LinkedStack<String>();

// Add three string values to the stack
stackOfStrings.push("A");
stackOfStrings.push("B");
stackOfStrings.push("C");

// Show each element before each element is
while (! stackOfStrings.isEmpty()) {
  // Print the value of the element at the
  System.out.println(stackOfStrings.pop());
}
```



Output

```
C
B
A
```

Self-Check

17-1 Write the output generated by the following code:

```
OurStack<String> aStack = new LinkedStack<String> ();
aStack.push("x");
aStack.push("y");
aStack.push("z");
while (! aStack.isEmpty()) {
  out.println(aStack.pop());
}
```

17-2 Write the output generated by the following code:

```
OurStack<Character> opStack = new OurLinkedStack<Character>();
out.println(opStack.isEmpty());
opStack.push('>');
opStack.push('+');
```

```
opStack.push('<');
out.print(opStack.peek());
out.print(opStack.peek());  // careful
out.print(opStack.peek());
```

17-3 Write the output generated by the following code:

```
OurStack<Integer> aStack = new OurLinkedStack<Integer>();
aStack.push(3);
aStack.push(2);
aStack.push(1);
out.println(aStack.isEmpty());
out.println(aStack.peek());
aStack.pop();
out.println(aStack.peek());
aStack.pop();
out.println(aStack.peek());
aStack.pop();
out.println(aStack.isEmpty());
```

17.2 Stack Application: Balanced Symbols

Some compilers perform symbol balance checking before checking for other syntax errors. For example, consider the following code and the compile time error message generated by a particular Java compiler (your compiler may vary).

```
public class BalancingErrors
  public static void main(String[] args) {
    int x = p;
    int y = 4;
    in z = x + y;
    System.out.println("Value of z = " + z);
  }
}
```

```
BalancingErrors.java:1: '{' expected
public class BalancingErrors
                             ^
```

Notice that the compiler did not report other errors, one of which is on line 3. There should have been an error message indicating `p` is an unknown symbol. Another compile time error is on line 5 where `z` is incorrectly declared as an `in` not `int`. If you fix the first error by adding the left curly brace on a new line 1 you will see these other two errors.

```
public class BalancingErrors  {  // <- add an opening curly brace
  public static void main(String[] args) {
    int x = p;
    int y = 4;
    in z = x + y;
    System.out.println("Value of z = " + z);
  }
}

BalancingErrors.java:3: cannot resolve symbol
symbol  : variable p
location: class BalancingErrors
    int x = p;
            ^
BalancingErrors.java:5: cannot resolve symbol
symbol  : class in
location: class BalancingErrors
    in z = x + y;
```

```
    ^
2 errors
```

This behavior could be due to a compiler that first checks for balanced { and } symbols before looking for other syntax errors.

Now consider how a compiler might use a stack to check for balanced symbols such as ( ) , { }, and [ ]. As it reads the Java source code, it will only consider opening symbols: ({ [, and closing symbols:) }]. If an opening symbol is found in the input file, it is pushed onto a stack. When a closing symbol is read, it is compared to the opener on the top of the stack. If the symbols match, the stack gets popped. If they do not match, the compiler reports an error to the programmer. Now imagine processing these tokens, which represent only the openers and closers in a short Java program: {{([])}}. As the first four symbols are read — all openers — they get pushed onto the stack.

Java source code starts as: {{([])}}

```
[
(       push the first four opening symbols with [ at the top. Still need to read ])}}
{
{
```

The next symbol read is a closer: "]". The "[" would be popped from the top of the stack and compared to "]". Since the closer matches the opening symbol, no error would be reported. The stack would now look like this with no error reported:

```
(
{           pop [ which matches ]. There is no error. Still need to read )}}
{
```

The closing parenthesis ")" is read next. The stack gets popped again. Since the symbol at the top of the stack "(" matches the closer ")", no error needs to be reported. The stack would now have the two opening curly braces.

```
{               pop ( which matches). There is no error. Still need to read }}
{
```

The two remaining closing curly braces would cause the two matching openers to be popped with no errors. It is the last-in-first-out nature of stacks that allows the first pushed opener "{" to be associated with the last closing symbol "}" that is read.

Now consider Java source code with only the symbols (]. The opener "(" is pushed. But when the closer "]" is encountered, the popped symbol "(" does not match "]" and an error could be reported. Here are some other times when the use of a stack could be used to help detect unbalanced symbols:

9. If a closer is found and the stack is empty. For example, when the symbols are {}}. The opening { is pushed and the closer "}" is found to be correct. However when the second } is encountered, the stack is empty. There is an error when } is discovered to be an extra closer (or perhaps { is missing).

10. If all source code is read and the stack is not empty, an error should be reported. This would happen with Java source code of `{{([])}`. In this case, there is a missing right curly brace. Most symbols are processed without error. At the end, the stack *should* be empty. Since the stack is *not* empty, an error should be reported to indicate a missing closer.

One of the end-of-chapter programming projects provides a summary of this algorithm. It also includes some example input files and the expected error messages like this:

```
public class Test0 }
  public static void main(String[args {
    System.out.println();
  }
}
```

```
Check symbols in Test1.java
Test0.java:1 Unexpected }
Test0.java:2 expecting ]
Test0.java:2 expecting )
3 errors
```

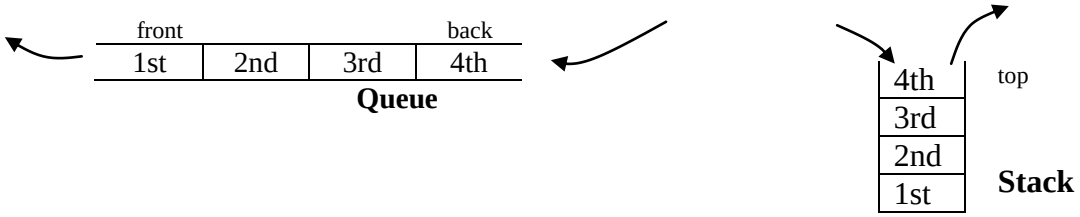
Self-Check

17-4 Write the errors generated when the algorithm above processes the following input file:

```
public class Test2 {
   public static void main(String[] args) {
     System.out.println();
  )) {
```

17.3 FIFO Queues

A first-in, first-out (FIFO) **queue** — pronounced "Q" — models a waiting line. Whereas stacks add and remove elements at one location — the `top` — `queues` add and remove elements at different locations. New elements are added at the back of the queue. Elements are removed from the front of the queue.

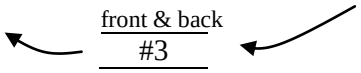


Whereas stacks mimic LIFO behavior, queues mimic a first in first out (FIFO) behavior. So, for example, the queue data structure models a waiting line such as people waiting for a ride at an amusement park. The person at the front of the line will be the first person to ride. The most recently added person must wait for all the people in front of them to get on the ride. With a FIFO queue, the person waiting longest in line is served before all the others who have waited less time[8].

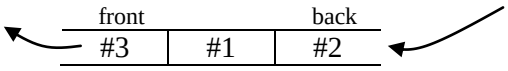
Another example of queue behavior can be found when several documents need to be printed at a shared printer. Consider three students, on the same network, trying to print one document

[8] Note: A *priority queue* has different behavior where elements with a higher priority would be removed first. For example, the emergency room patient with the most need is attended to next, not the patient who has been there the longest.

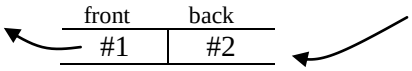
each. Who gets their document printed first? If a FIFO queue is being used to store incoming print requests, the student whose request reached the print queue first will get printed ahead of the others. Now assume that the printer is busy and the print queue gets a print request from student #3 while a ducument is printing. The print queue would look something like this:



In this case the queue's front element is also at the back end of the queue. The queue contains one element. Now add another request from student #1, followed by another request from student #2 for printing, and the print queue would look like this:



Student #1 and student #2 requests were added to the back of queue. The print requests are stored in the order in which they arrived. As the printer prints documents, the document will be removed from the front. Once the printer has printed the current document, the document for student #3 will then be removed. Then the state of the queue will now look like this:



A Queue Interface — Specifying the methods

There is no universally agreed upon set of operations; however the following is a reasonable set of operations for a FIFO Queue ADT.

- isEmpty Return true only when there are zero elements in the queue
- add Add an element at the back of the queue
- peek Return a reference to the element at the front of the queue
- remove Return a reference to the element at the front and remove the element

This leads to the following interface for a queue that can store any class of object.

```
public interface OurQueue<E> {

  /**
   * Find out if the queue is empty.
   * @returns true if there are zero elements in this queue.
   */
  public boolean isEmpty();

  /**
   * Add element to the "end" of this queue.
   * @param newEl element to be placed at the end of this queue.
   */
  public void add(E newEl);

  /**
   * Return a reference to the element at the front of this queue.
```

```
    * @returns A reference to the element at the front.
    * @throws NoSuchElementException if this queue is empty.
    */
  public E peek();

  /**
    * Return a reference to front element and remove it.
    * @returns A reference to the element at the front.
    * @throws NoSuchElementException if this queue is empty.
    */
  public E remove();
}
```

The following code demonstrates the behavior of the methods assuming OurLinkedQueue implements interface OurQueue:

```
OurQueue<Integer> q = new OurLinkedQueue<Integer>();
q.add(6);
q.add(2);
q.add(4);
while (!q.isEmpty()) {
  System.out.println(q.peek());
  q.remove();
}
```

Output

```
6 2 4
```

Self-Check

17-5 Write the output generated by the following code.

```
OurQueue<String> stringQueue = new OurLinkedQueue<String>();
stringQueue.add("J");
stringQueue.add("a");
stringQueue.add("v");
stringQueue.add("a");
while (!stringQueue.isEmpty()) {
  System.out.print(stringQueue.remove());
}
```

17-6 Write the output generated by the following code until you understand what is going on.

```
OurQueue<String> stringQueue = new OurLinkedQueue<String>();
stringQueue.add("first");
stringQueue.add("second");
while (! stringQueue.isEmpty()) {
  System.out.println(stringQueue.peek());
}
```

17-7 Write code that displays a message to indicate if each integer in a queue named `intQueue` is even or odd. The queue must remain intact after you are done. The queue is initialized with random integers in the range of 0 through 99.

```
OurQueue<Integer> intQueue = new OurLinkedQueue<Integer>();
Random generator = new Random();
for(int j = 1; j <= 7; j++) {
  intQueue.add(generator.nextInt(100));
}
// Your solution goes here
```

Sample Output (output varies since random integers are added)

```
28 is even
72 is even
4 is even
37 is odd
94 is even
98 is even
33 is odd
```

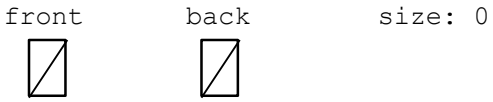
17.4 Queue with a Linked Structure

We will implement interface OurQueue with a class that uses a singly linked structure. There are several reasons to choose a linked structure over an array:

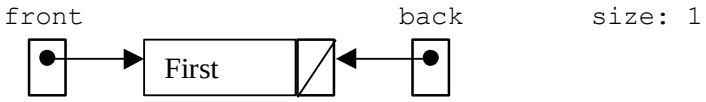
- It is easier to implement. (A programming project explains the trickier array-based implementation).
- The Big-O runtime of all algorithms is as efficient as if an array were used to store the elements. All algorithms can be O(1).

- An array-based queue would have `add` and `remove` methods, which will occasionally run O(n) rather than O(1). This occurs whenever the array capacity needs to be increased or decreased.
- It provides another good example of implementing a data structure using the linked structure introduced in the previous chapter.

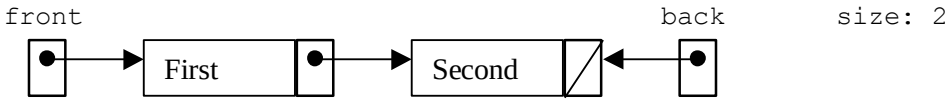
Elements are removed from the "front" of a queue. New elements are added at the back of the queue. Both "ends" of the queue are frequently accessed. Therefore, this implementation of `OurQueue` will use two external references. Only one external reference to the front is required. However, this would make for O(n) behavior during `add` messages, since a loop would need to sequence through all elements before reaching the end. With only a reference to the front, all elements must be visited to find the end of the list before one could be added. Therefore, an external reference named `back` will be maintained in addition to front. This will allow `add` to be O(1). An empty OurLinkedQueue will look like this:



After `q.add("First")`, a queue of size 1 will look like this:



After `q.add("Second")`, the queue of size 2 will look like this:



This test method shows the changing state of a queue that follows the above pictures of memory.

```
@Test
public void testAddAndPeek() {
  OurQueue<String> q = new OurLinkedQueue<String>();
  assertTrue(q.isEmpty()); // front == null
  q.add("first");
  assertEquals("first", q.peek());  // front.data is "first"
  assertFalse(q.isEmpty());

  q.add("second");  // Change back, not front
  // Front element should still be the same
  assertEquals("first", q.peek());
}
```

The first element is accessible as `front.data`. A new element is added by storing a reference to the new node into `back.next` and adjusting `back` to reference the new node at the end.

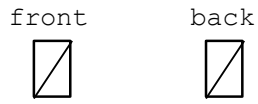
Here is the beginning of class OurLinkedQueue that once again uses a private inner `Node` class to store the data along with a link to the next element in the collection. There are two instance variables to maintain both ends of the queue.

```
public class OurLinkedQueue<E> implements OurQueue<E> {

  private class Node {
    private E data;
    private Node next;

    public Node() {
      data = null;
      next = null;
    }

    public Node(E elementReference, Node nextReference) {
      data = elementReference;
      next = nextReference;
    }
  } // end class Node

  // External references to maintain both ends of a Queue
  private Node front;
  private Node back;

  /**
   * Construct an empty queue (no elements) of size 0.
   */
  public OurLinkedQueue() {
    front = null;
    back = null;
  }

 /** Find out if the queue is empty.
  * @returns true if there are zero elements in this queue.
  */
  public boolean isEmpty()  {
    return front == null;
  }

  // More methods to be added . . .
}
```

This implementation recognizes an empty queue when `front` is `null`.

add

The `add` operation will first check for the special case of adding to an empty queue. The code to add to a non-empty queue is slightly different. If the queue is empty, the external references front and back are both `null`.



In the case of an empty queue, the single element added will be at front of the queue and also at the back of the queue. So, after building the new node, front and back should both refer to the same node. Here is a before and after picture made possible with the code shown.

```
  // Build a node to be added at the end. A queue can
  // grow as long as the computer has enough memory.
```

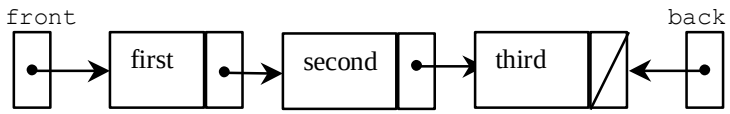
```
// With a linked structure, resizing is not necessary.
if (this.isEmpty()) {
   front = new Node(element, null);
   back = front;
 }
```

When an add messages is sent to a queue that is not empty, the last node in the queue must be made to refer to the node with the new element. Although `front` must remain the same during `add` messages, `back` must be changed to refer the new element at the end.

```
else {
  back.next = new Node(element);
  back = back.next;
}
```

There are several viable variations of how algorithms could be implemented when a linked structure is used to store the collection of elements. The linked structure used here always maintains two external references for the `front` and `back` of the linked structure. This was done so `add` is O(1) rather than O(n). In summary, the following code will generate the linked structure shown below.

```
OurQueue<String> q = new OurLinkedQueue<String>();
q.add("first");
q.add("second");
q.add("third");
```



Self-Check

17-8 Draw a picture of what the memory would look like after this code has executed

```
OurQueue<Double> q1 = new OurLinkedQueue<Double>();
q1.add(5.6));
q1.add(7.8));
```

17-9 Implement a `toString` method for `OurLinkedQueue` so this assertion would pass after the code in the previous self-check question:

```
assertEquals("[a, b]", q2.toString());
```

peek

The `peek` method throws a `NoSuchElementException` if the queue is empty. Otherwise, `peek` returns a reference to the element stored in front.data.

```
/**
  * Return a reference to the element at the front of this queue.
  * @returns A reference to the element at the front.
  * @throws NoSuchElementException if this queue is empty.
```

```
 */
public E peek() {
  if (this.isEmpty())
    throw new java.util.NoSuchElementException();
  else
    return front.data;
}
```

The next two test methods verify that `peek` returns the expected value and that it does not modify the queue.

```
@Test
public void testPeek() {
  OurQueue<String> q = new OurLinkedQueue<String>();
  q.add(new String("first"));
  assertEquals("first", q.peek());
  assertEquals("first", q.peek());

  OurQueue<Double> numbers = new OurLinkedQueue<Double>();
  numbers.add(1.2);
  assertEquals(1.2, numbers.peek(), 1e-14);
  assertEquals(1.2, numbers.peek(), 1e-14);
}

public void testIsEmptyAfterPeek() {
  OurQueue<String> q = new OurLinkedQueue<String>();
  q.add("first");
  assertFalse(q.isEmpty());
  assertEquals("first", q.peek());
}
```

An attempt to peek at the element at the front of an empty queue results in a `java.util.NoSuchElementException`, as verified by this test:

```
@Test(expected = NoSuchElementException.class)
public void testPeekOnEmptyList() {
  OurQueue<String> q = new OurLinkedQueue<String>();
  q.peek();
}
```

remove

The `remove` method will throw an exception if the queue is empty. Otherwise, `remove` returns a reference to the object at the front of the queue (the same element as `peek()` would). The `remove` method also removes the front element from the collection.

```
@Test
public void testRemove() {
  OurQueue<String> q = new OurLinkedQueue<String>();
  q.add("c");
  q.add("a");
  q.add("b");
  assertEquals("c", q.remove());
  assertEquals("a", q.remove());
  assertEquals("b", q.remove());
}

@Test(expected = NoSuchElementException.class)
```

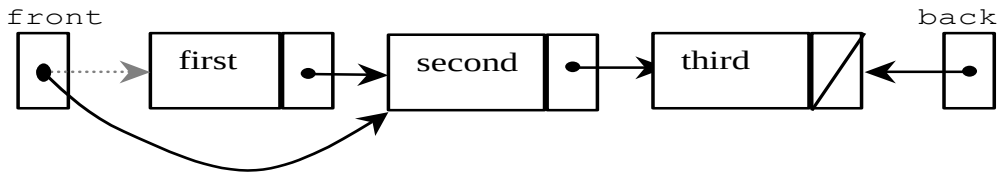
```
public void testRemoveThrowsAnException() {
  OurQueue<Integer> q = new OurLinkedQueue<Integer>();
  q.remove();
}
```

Before the front node element is removed, a reference to the front element must be stored so it can be returned after removing it.

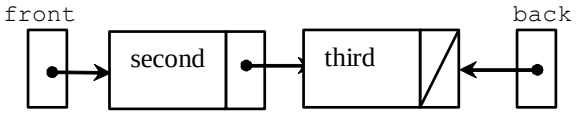
```
E frontElement = front.data;
```

`front`'s next field can be sent around the first element to eliminate it from the linked structure.

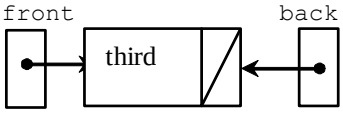
```
front = front.next;
```



Now the method can return a reference to `firstElement`. The linked structure would now look like this.



Another remove makes the list look like this.



Another remove message will return "third". The `remove` method should set front to null so isEmpty() will still work. This will leave the linked structure like this with back referring to a node that is no longer considered to be part of the queue. In this case, back will also be set to null.

Self-Check

17-10 Complete method `remove` so it return a reference to the element at the front of this queue while removing the front element. If the queue is empty, `throw new NoSuchElementException()`.

```
public E remove() {
```

Answers to Self-Checks

17-1
```
z
y
x
```

17-2
```
true
<<<
```

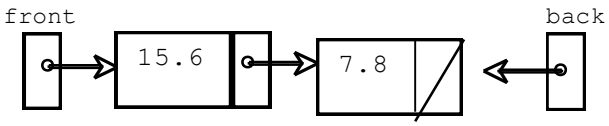
17-3
```
false
1
2
3
true
```

17-4
```
Check symbols in Test2.java
Abc.java:2 expecting ]
Abc.java:4 expecting }
Abc.java:4 expecting }
missing }
4 errors
```

17-5
```
Java
```

17-6
```
first
first
first
  ...
first until someone terminates the program or the power goes out
```

17-7
```
int size = intQueue.size();
for (int j = 1; j <= size; j++) {
  int nextInt = intQueue.peek();
  if (nextInt % 2 != 0)
     System.out.println(nextInt + " is odd");
  else
     System.out.println(nextInt + " is even");
  intQueue.remove();
  intQueue.add(nextInt);
}
```



17-9
```
public String toString() {
   String result = "[";

   // Concatenate all but the last one (if size > 0)
   Node ref = front;
   while (ref != back) {
     result += ref.data + ", ";
     ref = ref.next;
   }

   // Last element does not have ", " after it
   if (ref != null)
     result += ref.data;

   result += "]";

   return result;
 }
```

17-10
```
public E remove() throws NoSuchElementException {
  if (this.isEmpty())
    throw new NoSuchElementException();

  E frontElement = front.data;
  front = front.next;

  if (front == null)
    front = back = null;

  return frontElement;
}
```

Chapter 18

Recursion

18.1 Simple Recursion

One day, an instructor was having difficulties with a classroom's multimedia equipment. The bell rang, and still there was no way to project the lecture notes for the day. To keep her students occupied while waiting for the AV folks, she asked one of her favorite students, Kelly, to take attendance. Since the topic of the day was recursion, the instructor proposed a recursive solution: Instead of counting each student in the class, Kelly could count the number of students in her row and remember that count. The instructor asked Kelly to ask another student in the row behind her to do the same thing—count the number of students in their row, remember the count, and ask the same question of the next row.

By the time the question reached the last row of seats in the room, there was one person in each row who knew the number of students in that particular row. Andy, who had just counted eight students in the last row, asked his instructor what to do since there were no more rows of seats behind him. The teacher responded that all he had to do was return the number of students in his row to the person who asked him the question moments ago. So, Andy gave his answer of eight to the person in the row in front of him.

The student in the second to last row added the number of people in her row (12) to the number of students in the row behind her, which Andy had just told her (8). She returned the sum of 20 to the person in the row in front of her.

At that point the AV folks arrived and discovered a bent pin in a video jack. As they were fixing this, the students continued to return the number of students in their row plus the number of students behind them, until Kelly was informed that there were 50 students in all the rows behind her. At that point, the lecture notes, entitled "Recursion", were visible on the screen. Kelly told her teacher that there were 50 students behind her, plus 12 students in her first row, for a total of 62 students present.

The teacher adapted her lecture. She began by writing the algorithm for the head count problem. Every row got this same algorithm.

if you have rows behind you
  return the number of students in your row plus the number behind you
otherwise
  return the number of students in your row

Andy asked why Kelly couldn't have just counted the students one by one. The teacher responded, "That would be an *iterative* solution. Instead, you just solved a problem using a *recursive* solution. This is precisely how I intend to introduce recursion to you, by comparing recursive solutions to problems that could also be solved with iteration. I will suggest to you that some problems are better handled by a recursive solution."

Recursive solutions have a final situation when nothing more needs to be done—this is the base case—and situations when the same thing needs to be done again while bringing the problem closer to a base case. Recursion involves partitioning problems into simpler subproblems. It requires that each subproblem be identical in structure to the original problem.

Before looking at some recursive Java methods, consider a few more examples of recursive solutions and definitions. Recursive definitions define something by using that something as part of the definition.

Recursion Example 1

Look up a word in a dictionary:

find the word in the dictionary
if there is a word in the definition that you do not understand
  look up that word in the dictionary

Example: Look up the term **object**

Look up **object**, which is defined as "an instance of a **class**."

What is a class? *Look up* **class** to find "a collection of **method**s and data."

What is a method? *Look up* **method** to find "a **method heading** followed by a collection of programming statements."

Example: Look up the term **method heading**

What is a method heading? *Look up* **method heading** to find "the name of a method, its **return type**, followed by a **parameter list** in parentheses."

What is a parameter list? *Look up* **parameter list** to find "a **list** of **parameters**." *Look up* **list**, *look up* **parameters**, and *look up* **return type**, and you finally get a definition of all of the terms using the same method you used to *look up* the original term. And then, when all new terms are defined, you have a definition for **object**.

Recursion Example 2

A definition of a *queue*:

empty
or has someone at the front of the queue followed by a *queue*

Recursion Example 3

An arithmetic expression is defined as one of these:

a numeric constant such as 123 or –0.001
or a numeric variable that stores a numeric constant
or an *arithmetic expression* enclosed in parentheses
or an *arithmetic expression* followed by a binary operator (`+`, `-`, `/`, `%`, or `*`)
followed by an *arithmetic expression*

Characteristics of Recursion

A **recursive definition** is a definition that includes a simpler version of itself. One example of a recursive definition is given next: the power method that raises an integer (x) to an integer power (n).

This definition is recursive because x^{n-1} is part of the definition itself. For example,

$$4^3 = 4 \times 4^{(n-1)} = 4 \times 4^{(3-1)} = 4 \times 4^2$$

What is 4^2? Using the recursive definition above, 4^2 is defined as:

$$4^2 = 4 \times 4^{(n-1)} = 4 \times 4^{(2-1)} = 4 \times 4^1$$

and 4^1 is defined as

$$4^1 = 4 \times 4^{(n-1)} = 4 \times 4^{(1-1)} = 4 \times 4^0$$

and 4^0 is a base case defined as

$$4^0 = 1$$

The recursive definition of 4^3 includes 3 recursive definitions. The base case is `n==0`:
$x^n = 1$ if $n = 0$

To get the actual value of 4^3, work backward and let 1 replace 4^0, 4 * 1 replace 4^1, 4 * 4^1 replace 4^2, and 4 * 4^2 replace 4^3. Therefore, 4^3 is defined as 64.

To be recursive, an algorithm or method requires at least one recursive case and at least one base case. The recursive algorithm for power illustrates the characteristics of a recursive solution to a problem.

- The problem can be decomposed into a simpler version of itself in order to bring the problem closer to a base case.
- There is at least one base case that does not make a recursive call.
- The partial solutions are managed in such a way that all occurrences of the recursive and base cases can communicate their partial solutions to the proper locations (values are returned).

Comparing Iterative and Recursive Solutions

For many problems involving repetition, a recursive solution exists. For example, an iterative solution is shown below along with a recursive solution in the `TestPowFunctions` class, with the methods `powLoop` and `powRecurse`, respectively. First, a unit test shows calls to both methods, with the same arguments and same expected results.

```
// File RecursiveMethodsTest.java
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class RecursiveMethodsTest {

  @Test
  public void testPowLoop() {
    RecursiveMethods rf = new RecursiveMethods();
    assertEquals(1, rf.powLoop(4, 0));
    assertEquals(1, rf.powRecurse(4, 0));

    assertEquals(4, rf.powLoop(4, 1));
    assertEquals(4, rf.powRecurse(4, 1));

    assertEquals(16, rf.powLoop(2, 4));
    assertEquals(16, rf.powRecurse(2, 4));
  }
}

// File RecursiveMethods.java
public class RecursiveMethods {

  public int powLoop(int base, int power) {
    int result;
    if (power == 0)
      result = 1;
    else {
      result = base;
      for (int j = 2; j <= power; j++)
        result = result * base;
    }
    return result;
  }

  public int powRecurse(int base, int power) {
    if (power == 0)
      return 1;
    else
                    // Make a recursive call \\
      return base * powRecurse(base, power - 1);
  }
}
```

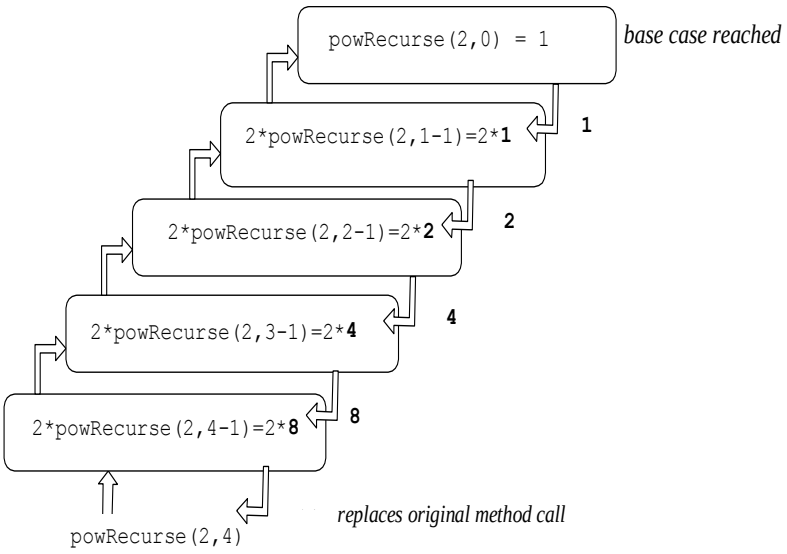
In `powRecurse`, if n is 0—the base case—the method call evaluates to 1. When n > 0—the recursive case—the method is invoked again with the argument reduced by one. For example, `powRecurse(4,1)` calls `powRecurse(4, 1-1)`, which immediately returns 1. For another example, the original call `powRecurse(2,4)` calls `powRecurse(2,3)`, which then calls `powRecurse(2,2)`, which then calls `powRecurse(2,1)`, which then calls `powRecurse(2,0)`, which returns 1. Then, `2*powRecurse(2,0)` evaluates to 2*1, or 2, so

`2*powRecurse(2,1)` evaluates to 4, `2*powRecurse(2,2)` evaluates to 8, `2*powRecurse(2,3)` evaluates to 16, and `2*powRecurse(2,4)` evaluates to 32.

Tracing recursive methods requires diligence, but it can help you understand what is going on. Consider tracing a call to the recursive power method to get 2^4.

```
assertEquals(16, rf.powRecurse(2, 4));
```

After the initial call to `powRecurse(2,4)`, `powRecurse` calls another instance of itself until the base case of `power==0` is reached. The following picture illustrates a method that calls instances of the same method. The arrows that go up indicate this. When an instance of the method can return something, it returns that value to the method that called it. The arrows that go down with the return values written to the right indicate this.



The final value of 16 is returned to the `main` method, where the arguments of 2 and 4 were passed to the first instance of `powRecurse`.

Self-Check

18-1 What is the value of `rf.powRecurse(3, 0)`?

18-2 What is the value of `rf.powRecurse(3, 1)`?

18-3 Fill in the blanks with a trace of the call `rf.powRecurse(3, 4)`.

Tracing Recursive Methods

In order to fully understand how recursive tracing is done, consider a series of method calls given the following method headers and the simple `main` method:

```
// A program to call some recursive methods
public class Call2RecursiveMethods {

  public static void main(String[] args) {
    Methods m = new Methods();
    System.out.println ("Hello");
    m.methodOne(3);
    m.methodTwo(6);
  }
}


// A class to help demonstrate recursive method calls
public class Methods {
  public void methodOne(int x) {
    System.out.println("In methodOne, argument is " + x);
  }

  public void methodTwo(int z) {
    System.out.println("In methodTwo, argument is " + z);
  }
}
```

Output

```
Hello
In methodOne, argument is 3
In methodTwo, argument is 6
```

This program begins by printing out `"Hello"`. Then a method call is made to `methodOne`. Program control transfers to `methodOne`, but not before remembering where to return to. After pausing execution of `main`, it begins to execute the body of the `methodOne` method. After `methodOne` has finished, the program flow of control goes back to the last place it was and starts executing where it left off—in this case, in the `main` method, just after the call to `methodOne` and just before the call to `methodTwo`. Similarly, the computer continues executing main and again transfers control to another method: `methodTwo`. After it completes execution of `methodTwo`, the program terminates.

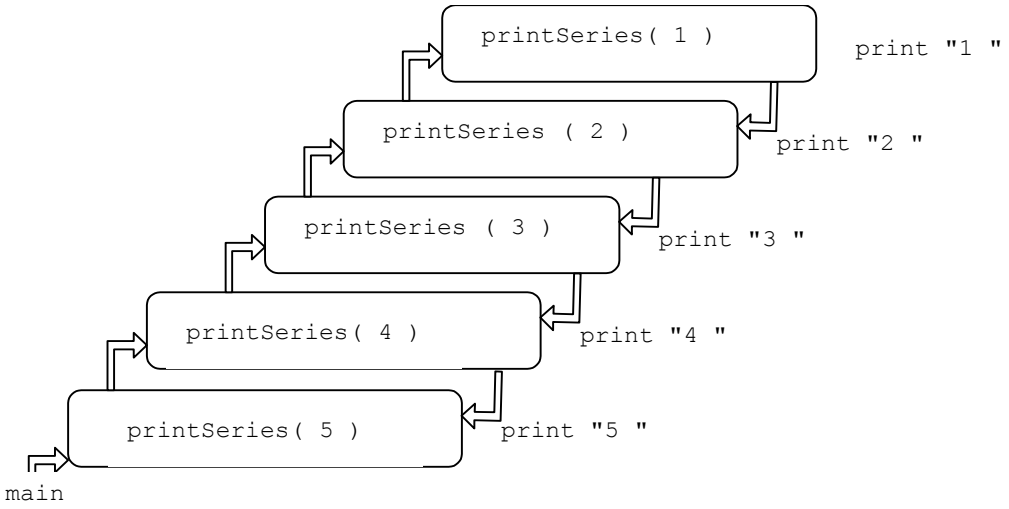
The above example shows that program control can go off to execute code in other methods and then know the place to come back to. It is relatively easy to follow control if no recursion is involved. However, it can be difficult to trace through code with recursive methods. Without recursion, you can follow the code from the beginning of the method to the end. In a recursive method, you must trace through the same method while trying to remember how many times the method was called, where to continue tracing, and the values of the local variables (such as the parameter values). Take for example the following code to print out a list of numbers from 1 to n, given n as an input parameter.

```
public void printSeries(int n) {
  if (n == 1)
    System.out.print(n + " ");
  else  {
    printSeries(n - 1);
    // after recursive call \\
    System.out.print(n + " ");
  }
}
```

A call to `printSeries(5)` generates this output:

```
1 2 3 4 5
```

Let's examine step by step how the result is printed. Each time the method is called, it is stacked with its argument. For each recursive case, the argument is an integer one less than the previous call. This brings the method one step closer to the base case. When the base case is reached (`n==1`) the value of `n` is printed. Then the previous method finishes up by returning to the last line of code below `/* after recursive call */`.



Recursive execution of printSeries(5)

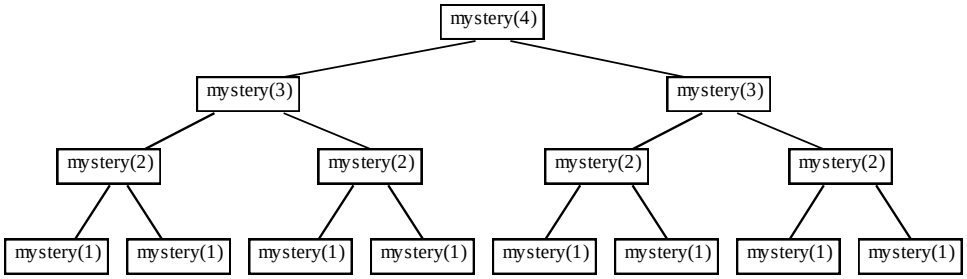
Notice that when a new recursive call is made, the current invocation of the method `printSeries(5)` starts a completely new invocation of the same method `printSeries(4)`. The system pushes the new method invocation on the top of a stack. Method invocations are pushed until the method finds a base case and finishes. Control returns back to previous invocation `printSeries(2)` by popping the stack, and the value of `n` (`2`) prints.

Now consider a method that has multiple recursive calls within the recursive case.

```
public void mystery(int n) {
  if (n == 1)
    System.out.print(" 1 ");
```

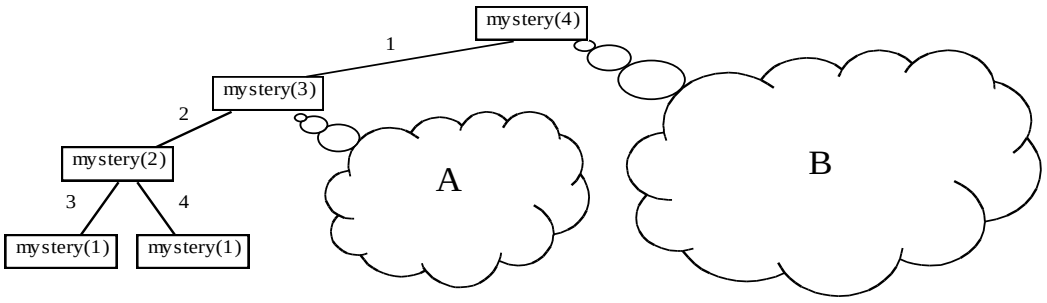
```
    else {
      mystery(n - 1);
      System.out.print("<" + n + ">");
      mystery(n - 1);
    }
  }
```

When the base case has not yet been reached, there is a recursive call, then a print statement, and then another recursive call. With two recursive calls, it proves more insightful to approach a trace from a graphical perspective. A method call tree for `mystery(4)` looks like this.



Recursive execution of mystery(4)

As you can see, when there are multiple recursive calls in the same method, the number of calls increases exponentially — there are eight calls to `mystery(1)`. The recursion reaches the base case when at the lowest level of the structure (at the many calls to `mystery(1)`). At that point `" 1 "` is printed out and control returns to the calling method. When the recursive call returns, `"<" + n + ">"` is printed and the next recursive call is called. First consider the left side of this tree. The branches that are numbered 1, 2, 3, and 4 represent the method calls after `mystery(4)` is called. After the call #3 to `mystery`, `n` is `1` and the first output `" 1 "` occurs.



The first part of mystery

Control returns to the previous call when `n` was 2. `<2>` is printed. The output so far:

```
1 <2>
```

Then a recursive call is a made as `mystery(2-1)` and `" 1 "` is printed again. The output so far:

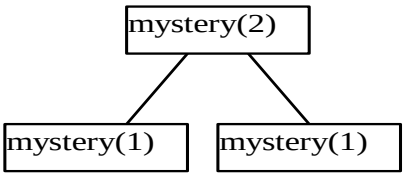
```
1 <2> 1
```

Control then returns to the first call to mystery(3) and <3> is printed. The output so far:

```
1 <2> 1 <3>
```

Then these method calls behind 

occur.



With n == 2, the base case is skipped and the recursive case is called once again. This means another call to `mystery(2-1)`, which is `" 1 "`, a printing of `<2>`, followed by another call to `mystery(2-1)`, which is yet another `" 1 "`. Add these three prints to the previous output and the output so far is:

```
1 <2> 1 <3> 1 <2> 1
```

This represents the output from `mystery(3)`. Control then returns to the original call `mystery(4)` when `<4>` is printed. Then the cloud behind B prints the same output as `mystery(3)`, the output shown immediately above. The final output is `mystery(3)`, followed by printing `n` when `n` is 4, followed by another mystery(3).

```
1 <2> 1 <3> 1 <2> 1 <4> 1 <2> 1 <3> 1 <2> 1
```

Self-Check

18-4 Describe the output that would be generated by the message `mystery(5);`.

18-5 What does `mystery2(4)` return?

```
public void mystery2(int n)  {
  if (n > 1)
    mystery2(n - 1);
  System.out.print(n + " ");
}
```

Infinite Recursion

Infinite recursion occurs when a method keeps calling other instances of the same method without making progress towards the base case or when the base case can never be met.

```
public int sum(int n) {
  if (n  ==  0)
```

```
    return 0;
  else
    return n + sum(n + 1);
}
```

In this example, no progress is made towards the base case when n is a positive integer because every time sum is called, it is called with a larger value, so the base condition will never be met.

Recursion and Method Calls

Recursion can be implemented on computer systems by allowing each method call to create a **stack frame** (also known as an activation record). This stack frame contains the information necessary for the proper execution of the many methods that are active while programs run. Stack frames contain information about the values of local variables, parameters, the return value (for non-void methods), and the return address where the program should continue executing after the method completes. This approach to handling recursive method calls applies to all methods. A recursive method does not call itself; instead, a recursive call creates an instance of a method that just happens to have the same name.

With or without recursion, there may be one too many stack frames on the stack. Each time a method is called, memory is allocated to create the stack frame. If there are many method calls, the computer may not have enough memory. Your program could throw a `StackOverflowError`. In fact you will get a `StackOverflowError` if your recursive case does not get you closer to a base case.

```
// This method causes a StackOverflowError.
// Recursive case does not bring the problem closer to the base case.
public int pow(int base, int power) {
  if (power == 0)
    return 1;
  else
    return base * pow(base, power + 1); // <- should be power - 1
}
```

Output

```
java.lang.StackOverflowError
```

The exception name hints at the fact that the method calls are being pushed onto a stack (as stack frames). At some point, the capacity of the stack used to store stack frames was exceeded. `pow`, as written above, will never stop on its own.

Many times your recursive solution can be easily made into an iterative solution, which will run faster and use less memory. A type of recursion called tail recursion lends itself very easily to this conversion.

Tail-Recursion

Tail recursion is where the recursive call is the last thing done in the method. There is no more computation or any additional code after the recursive call. For tail recursion to be present, there can be no preceding recursive call. The following correct implementation of `pow` is one example of tail recursion.

```
// Tail recursive -- could be written as a loop.
```

```
public int pow(int base, int power) {
  if (power == 0)
    return 1;
  else
    return base * pow(base, power - 1);
}
```

Any method that fits the tail recursive definition has an iterative solution. For example, the method above can be implemented with a loop.

```
public int pow(int base, int power) {
  int result = 1;

  for (int j = power; j > 0; j--)
    result = base * result;

  return result;
}
```

A tail recursive method can be changed into an iterative solution by replacing the `if` statement with a loop and incrementing or decrementing the loop control variable for each time that a recursive call would have been made.

Here is another example of tail recursion — computing n! (n factorial), which was discussed earlier. n! is really just n*(n-1)!, which is n*(n-1) (n-2)!, and so on until you reach 1. A recursive solution exists.

```
public int factorial(int n) {
    if (n == 0)
      return 1;
    else
      return n * factorial(n-1);
  }
```

Since the recursive call comes at the end of this method, and there is no other recursive call before it, the most common implementation of the factorial method uses a loop like this:

```
public int iterativeFactorial(int n) {
  int result = 1;
  for (int i = n; i >= 1; i--)
   result = result * i;
  return result;
}
```

The code above gives the same result as the recursive factorial method, but uses iteration to achieve its results. Basically, it starts the result at 1 (because we are multiplying), then multiplies n*(n-1)*(n-2)*… * 1.

There are other times when recursive solutions cannot be easily replaced with iteration, and the recursive solution is the preferred implementation—backtracking for example (see Escape the Obstacle course and the Word puzzle game later in this chapter). In Java, tail recursive methods may have little advantage—or even a disadvantage—over iterative solutions. However, some uses of recursion can be efficient. Because tail recursion is essentially iteration with high overhead, the iterative solution may be the way to go.

Self-Check

18-6 Write the return value of each.

a. ____ `mystery6(-5)`
b. ____ `mystery6(1)`
c. ____ `mystery6(2)`
d. ____ `mystery6(3)`
e. ____ `mystery6(4)`

```
public int mystery6(int n) {
  if (n < 1)
    return 0;
  else if (n == 1)
    return 1;
  else
    return 2 * mystery6(n - 1);
}
```

18-7 Write the return value of each.

a. ____ `mystery7(14, 7)`
b. ____ `mystery7(3, 6)`
c. ____ `mystery7(4, 8)`

```
public boolean mystery7(int a, int b) {
  if (a >= 10 || b <= 3)
    return false;
  if (a == b)
    return true;
  else
   return mystery7(a + 2, b - 2) || mystery7(a + 3, b - 4);
}
```

18-8 Given the following definition of the Fibonacci sequence, write a recursive method to compute the nth term in the sequence.

```
fibonacci(0) = 1;
fibonacci(1) = 1;
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2); when n >= 2
```

18-9 Write recursive method `howOften` as if it were n class RecursiveMethods that will compute how often a substring occurs in a string. Do not use a loop. Use recursion. These assertions must pass:

```
@Test
public void testSequentialSearchWhenNotHere() {
  RecursiveMethods rm = new RecursiveMethods();
  assertEquals(5, rm.howOften("AAAAA", "A")) ;
  assertEquals(0, rm.howOften("AAAAA", "B")) ;
  assertEquals(2, rm.howOften("catdogcat", "cat")) ;
  assertEquals(1, rm.howOften("catdogcat", "dog")) ;
  assertEquals(2, rm.howOften("AAAAA", "AA")) ;
}
```

18.2 Palindrome

Suppose that you had a word and you wanted the computer to check whether or not it was a palindrome. A **palindrome** is a word that is the same whether read forward or backward; radar, madam, and racecar, for example. To determine if a word is a palindrome, you could put one finger under the first letter, and one finger under the last letter. If those letters match, move your fingers one letter closer to each other, and check those letters. Repeat this until two letters do not match or your fingers touch because there are no more letters to consider.

The recursive solution is similar to this. To solve the problem using a simpler version of the problem, you can check the two letters on the end. If they match, ask whether the `String` with the end letters removed is a palindrome.

The base case occurs when the method finds a `String` of length two with the same two letters. A simpler case would be a `String` with only one letter, or a `String` with no letters. Checking for a `String` with 0 or 1 letters is easier than comparing the ends of a `String` with the same two letters. When thinking about a base case, ask yourself, "Is this the simplest case? Or can I get anything simpler?" Two base cases (the number of characters is 0 or 1) can be handled like this (assume `str` is the `String` object being checked).

```
if (str.length() <= 1)
  return true;
```

Another base case is the discovery that the two end letters are different when `str` has two or more letters.

```
else if (str.charAt(0) != str.charAt(str.length() - 1))
  return false; // The end characters do not match
```

So now the method can handle the base cases with `String`s such as `""`, `"A"`, and `"no"`. The first two are palindromes; `"no"` is not.

If a `String` is two or more characters in length and the end characters match, no decision can be made other than to keep trying. The same method can now be asked to solve a simpler version of the problem. Take off the end characters and check to see if the smaller string is a palindrome. `String`'s `substring` method will take the `substring` of a `String` like `"abba"` to get `"bb"`.

```
// This is a substring of the original string
// with both end characters removed.
return isPalindrome(str.substring(1, str.length() - 1));
```

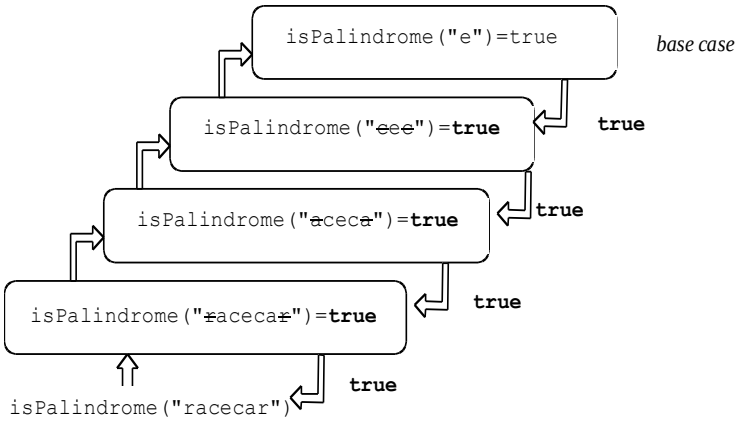
This message will not resolve on the next call. When `str` is `"bb"`, the next call is `isPalindrome("")`, which returns `true`. It has reached a base case—length is 0. Here is a complete recursive palindrome method.

```
// Return true if str is a palindrome or false if it is not
public boolean isPalindrome(String str) {
  if (str.length() <= 1) {
    // Base case when this method knows to return true.
    return true;
  }
  else if (str.charAt(0) != str.charAt(str.length() - 1)) {
    // Base case when this method knows to return false
    // because the first and last characters do not match.
    return false;
  }
  else {
    // The first and last characters are equal so check if the shorter
    // string--a simpler version of this problem--is a palindrome.
    return isPalindrome(str.substring(1, str.length() - 1));
  }
}
```

If the length of the string is greater than 1 and the end characters match, `isPalindrome` calls another instance of `isPalindrome` with smaller and smaller `String` arguments until one base case is reached. Either a `String` is found that has a length less than or equal to 1, or the

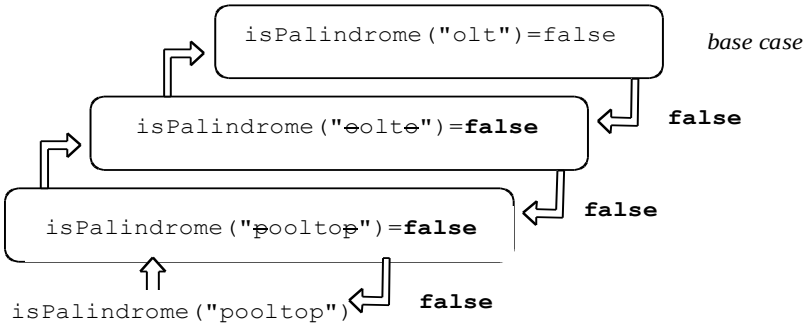
characters on the ends are not the same. The following trace of `isPalindrome("racecar")` visualizes the calls that are made in this way.

`isPalindrome` Recursive Calls (true result)



Since the fourth (topmost) call to `isPalindrome` is called with the `String` `"e"`, a base case is found—a `String` with length 1. This `true` value gets returned to its caller (argument was `"e"`), which in turn returns `true` back to its caller (the argument was `"cec"`), until `true` gets passed back to the first caller of `isPalindrome`, the method call with the original argument of `"racecar"`, which returns the value `true`. Now consider tracing the recursive calls for the `String` `"pooltop"`.

`isPalindrome` Recursive Calls (false result)



The base case is reached when the method compares the letters at the ends—`"o"` and `"t"` do not match. That particular method call returns `false` back to its caller (whose argument was `"oolto"`), which returns `false` to its caller. The original call to `isPalindrome("pooltop")` is replaced with `false` to the method that originally asked if `"pooltop"` was a palindrome.

Self-Check

18-10 What value is returned from `isPalindrome("yoy")`?

18-11 What value is returned from `isPalindrome("yoyo")`?

18-12 Write the return value of each method call

a. _____ huh("+abc+"));
b. _____ huh("-abc-"));
c. _____ huh("-a-b-c-"));
d. _____ huh("------abc-----"));

```
public String huh(String str) {
  if (str.charAt(0) == '-')
    return huh(str.substring(1, str.length()));
  else if (str.charAt(str.length() - 1) == '-')
    return huh(str.substring(0, str.length() - 1));
  else
    return str;
}
```

18.3 Recursion with Arrays

The *sequential search* algorithm uses an integer subscript that increases if an element is not found and the index is still in the range (meaning that there are more elements to compare). This test method demonstrates the desired behavior.

```
@Test
public void testSequentialSearchWhenHere() {
  RecursiveMethods rm = new RecursiveMethods();
  String[] array = { "Kelly", "Mike", "Jen", "Marty", "Grant" };
  int lastIndex = array.length - 1;

  assertTrue(rm.exists(array, lastIndex, "Kelly"));
  assertTrue(rm.exists(array, lastIndex, "Mike"));
  assertTrue(rm.exists(array, lastIndex, "Jen"));
  assertTrue(rm.exists(array, lastIndex, "Marty"));
  assertTrue(rm.exists(array, lastIndex, "Grant"));
}
```

The same algorithm can be implemented in a recursive fashion. The two base cases are:

1. If the element is found, return true.
2. If the index is out of range, terminate the search by returning false.

The recursive case looks in the portion of the array that has not been searched. With sequential search, it does not matter if the array is searched from the smallest index to the largest or the largest index to the smallest. The `exists` message compares the search element with the largest valid array index. If it does not match, the next call narrows the search. This happens when the recursive call simplifies the problem by decreasing the index. If the element does not exist in the array, eventually the index goes to -1 and the method returns `false` to the preceding call, which returns `false` to the preceding call, until the original method call to `exists` returns `false` to the point of the call.

```
// This is the only example of a parameterized method.
// The extra <T>s allow any type of arguments.
public <T> boolean exists(T[] array, int lastIndex, T target) {
  if (lastIndex < 0) {
    // Base case 1: Nothing left to search
    return false;
  } else if (array[lastIndex].equals(target)) { // Base case 2: Found it
```

```
      return true;
    } else { // Recursive case
      return exists(array, lastIndex - 1, target);
    }
  }
```

A test should also be made to ensure exists returns false when the target is not in the array.

```
@Test
public void testSequentialSearchWhenNotHere() {
  RecursiveMethods rm = new RecursiveMethods();
  Integer[] array = { 1, 2, 3, 4, 5 };
  int lastIndex = array.length - 1;
  assertFalse(rm.exists(array, lastIndex, -123));
  assertFalse(rm.exists(array, lastIndex, 999));
}
```

Self-Check

18-13 What would happen when `lastIndex` is not less than the array's capacity as in this assertion?

```
assertFalse(rm.exists(array, array.length + 1, "Kelly"));
```

18-14 What would happen when `lastIndex` is less than 0 as in this assertion?

```
assertFalse(rm.exists(array, -1, "Mike"));
```

18-15 Write a method `printForward` that prints all objects referenced by the array named `x` (that has `n` elements) from the first element to the last. Use recursion. Do not use any loops. Use this method heading:

```
public void printForward(Object[] array, int n)
```

18-16 Complete this `testReverse` method so a method named reverse in class RecursiveMethods will reverse the order of all elements in an array of Objects that has `n` elements. Use this heading:

```
public void printReverse(Object[] array, int leftIndex, int rightIndex)
```

```
@Test
public void testReverse() {
  RecursiveMethods rm = new RecursiveMethods();

  String[] array =  { "A", "B", "C" };
  rm.reverse(array, 0, 2);
  assertEquals(                    );
  assertEquals(                    );
  assertEquals(                    );
}
```

18-17 Write the recursive method reverse is if it were in class RecursiveMethods. Use recursion. Do not use any loops.

18.4 Recursion with a Linked Structure

This section considers a problem that you have previously resolved using a loop — searching for an object reference from within a linked structure. Consider the base cases first.

The simplest base case occurs with an empty list. In the code shown below, this occurs when there are no more elements to compare. A recursive find method returns `null` to indicate that the object being searched for did not equal any in the list. The other base case is when the object is found. A recursive method then returns a reference to the element in the node.

The recursive case is also relatively simple: If there is some portion of the list to search (not yet at the end), and the element is not yet found, search the remainder of the list. This is a simpler version of the same problem – search the list that does not have the element that was just compared. In summary, there are two base cases and one recursive case that will search the list beginning at the next node in the list:

Base cases:

If there is no list to search, return `null`.

If the current node equals the object, return the reference to the data.

Recursive case:

Search the remainder of the list – from the next node to the end of the list

The code for a recursive search method is shown next as part of `class SimpleLinkedList`. Notice that there are two `findRecursively` methods — one public and one private. (Two methods of the same name are allowed in one class if the number of parameters differs.) This allows users to search without knowing the internal implementation of the class. The public method requires the object being searched for, but not the private instance variable named `front`, or any knowledge of the `Node` class. The public method calls the private method with the element to search for along with the first node to compare — `front`.

```
public class SimpleLinkedList<E> {

  private class Node  {
    private E data;
    private Node next;

    public Node(E objectReference, Node nextReference) {
      data = objectReference;
      next = nextReference;
    }
  } // end class Node

  private Node front;

  public SimpleLinkedList() {
    front = null;
  }

  public void addFirst(E element) {
    front = new Node(element, front);
  }

   // Return a reference to the element in the list that "equals" target
   // Precondition: target's type overrides "equals" to compare state
  public E findRecursively(E target) {
    // This public method hides internal implementation details
    // such as the name of the reference to the first node to compare.
```

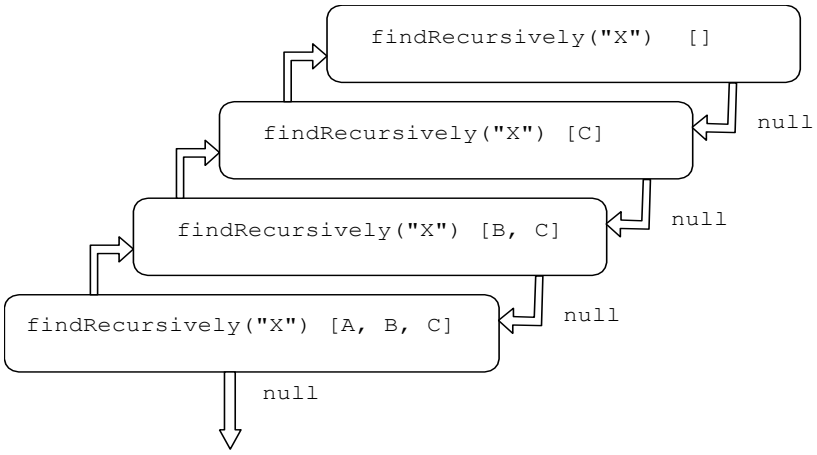
```
    //
    // The private recursive find, with two arguments, will do the work.
    // We don't want the programmer to reference first (it's private).
    // Begin the search at the front, even if front is null.
    return findRecursively(target, front);
  }

  private E findRecursively(E target, Node currentNode) {
    if (currentNode == null) // Base case--nothing to search for
      return null;
    else if (target.equals(currentNode.data)) // Base case -- element found
      return currentNode.data;
    else
      // Must be more nodes to search, and still haven't found it;
      // try to find from the next to the last. This could return null.
      return findRecursively(target, currentNode.next);
  }
}
```

Each time the `public findRecursively` method is called with the object to find, the `private findRecursively` method is called. This private method takes two arguments: the object being searched for and `front` — the reference to the first node in the linked structure. If `front` is `null`, the private `findRecursively` method returns `null` back to the public method `findRecursively` , which in turn returns `null` back to main (where it is printed).

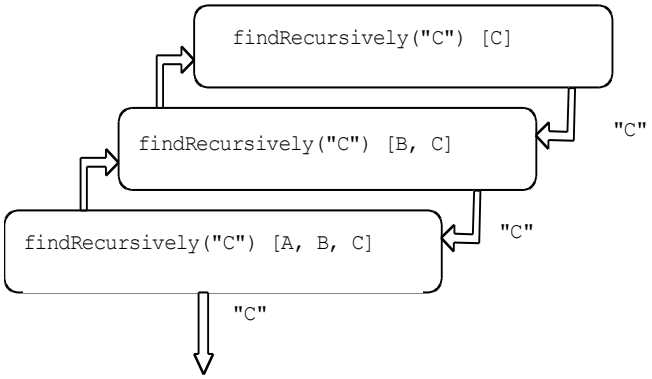
In a non-empty list, `currentNode` refers to a node containing the first element. If the first node's data does not equal the search target, a recursive call is made. The second argument would be a reference to the second node in the list. (The method still needs to pass the object being searched). This time, the problem is simpler because there is one less element in the list to search. Each recursive call to `findRecursively` has a list with one less node to consider. The code is effectively "shrinking" the search area.

The recursive method keeps making recursive calls until there is either nothing left of the list to search (return `null`), or the element being searched for is found in the smaller portion of the list. In this latter case, the method returns the reference back to the public method, which in turn returns the reference back to main (where it is printed).

At some point, the method will eventually reach a base case. There will be one method call on the stack for each method invocation. The worst case for `findRecursively` is the same for sequential search: O(n). This means that a value must be returned to the calling function, perhaps thousands of times. A trace of looking for an element that is not in the list could look like this. The portion of the list being searched is shown to the right of the call (`[]` is an empty list):



And here is a trace of a successful search for "C". If "C" were at the end of the list with size() == 975, there would have been 975 method calls on the stack.



Self-Check

18-18 Add a recursive method `toString` to the `SimpleLinkedList` class above that returns a string with the `toString` of all elements in the linked list separated by spaces. Use recursion, do not use a loop.

Answers to Self-Checks

18-1 powRecurse (3, 0) == 1

18-2 powRecurse (3, 1) == 3

18-3 filled in from top to bottom

3*(3, 0) = 1
3*(3, 1) = 3*1 = 3
3*(3, 2) = 3*3 = 9
3*(3, 3) = 3*9 = 27
3*(3, 4) = 3*27 = 81

18-4 result mystery(5)
1 <2> 1 <3> 1 <2> 1 <4> 1 <2> 1 <3> 1 <2> 1 <5> 1 <2> 1 <3> 1 <2> 1 <4> 1 <2> 1 <3> 1 <2> 1
fence post pattern - the brackets follow the numbers being recursed back into the method

18-5 mystery2(4) result: 1 2 3 4

18-6
a. __0__ `mystery6(-5)`
b. __1__ `mystery6(1)`
c. __2__ `mystery6(2)`
d. __4__ `mystery6(3)`
e. __8__ `mystery6(4)`

18-7 a. false b. false c. true

18-8

```
public int fibonacci(int n){
  if(n == 0)
    return 1;
  else if(n ==1)
    return 1;
  else if(n >= 2)
    return fibonacci(n-1) + fibonacci(n-2);
  else
   return -1;
}
```

18-9

```
public int howOften(String str, String sub) {
  int subsStart = str.indexOf(sub);
  if (subsStart < 0)
    return 0;
  else
    return 1 + howOften(str.substring(subsStart + sub.length()), sub);
}
```

18-10 isPalindrome ("yoy") == true

18-11 isPalindrome("yoyo") == false

18-12 return values for huh, in order

a. +abc+
b. abc
c. a-b-c
d. abc

18-13 - if "Kelly" is not found at the first index, it will throw an arrayIndexOutOfBounds exception

18-14 - it will immediately return false without searching

18-15

```
public void printForward(Object[] array, int n) {
  if (n > 0) {
    printForward(array, n - 1);
    System.out.println(array[n-1].toString());
  }
}
```

18-16

```
assertEquals("A", array[2]);
assertEquals("B", array[1]);
assertEquals("C", array[0]);
```

18-17

```
public void reverse(Object[] array, int leftIndex, int rightIndex) {
  if (leftIndex < rightIndex) {
    Object temp = array[leftIndex];
    array[leftIndex] = array[rightIndex];
    array[rightIndex] = temp;
    reverse(array, leftIndex + 1, rightIndex - 1);
  }
}
```

18-18

```
public String toString (){
  return toStringHelper(front);
}

private String toStringHelper(Node ref) {
  if(ref == null)
    return "";
  else
    return ref.data.toString() + " " + toStringHelper(ref.next);
}
```

Chapter 19

Binary Trees

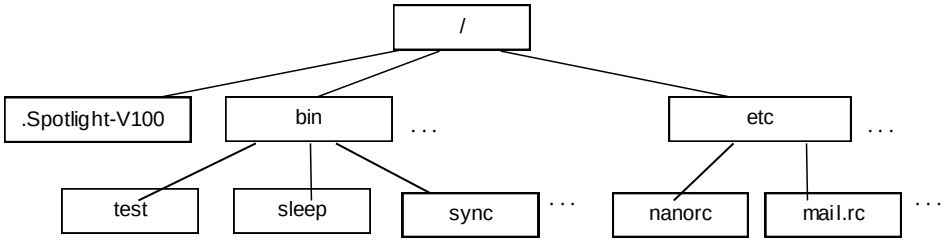
The data structures presented so far are predominantly linear. Every element has one unique predecessor and one unique successor (except the first and last elements). Arrays, and singly linked structures used to implement lists, stacks, and queues all have this linear characteristic. The **tree** structure presented in this chapter is a hierarchical in that nodes may have more than one successor.

Goals

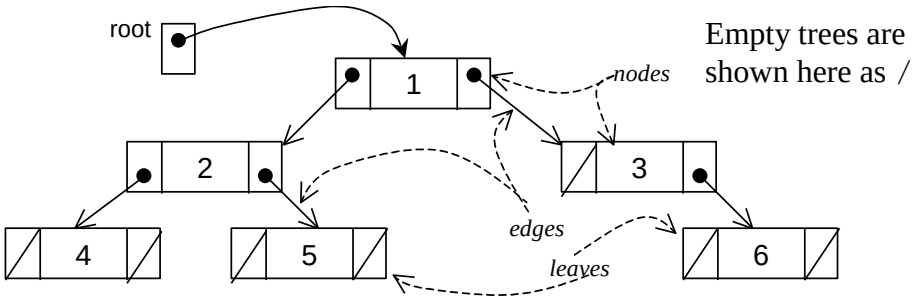
- Become familiar with tree terminology and some uses of trees
- Store data in a hierarchical data structure as a Java Collection class
- Implement binary tree algorithms
- Implement algorithms for a Binary Search Tree

19.1 Trees

Trees are often used to store large collections of data in a hierarchical manner where elements are arranged in successive levels. For example, file systems are implemented as a tree structure with the root directory at the highest level. The collection of files and directories are stored as a tree where a directory may have files and other directories. Trees are hierarchical in nature as illustrated in this view of a very small part of a file system (the root directory is signified as /).



Each node in a tree has exactly one parent except for the distinctive node known as the **root**. Whereas the root of a real tree is usually located in the ground and the leaves are above the root, computer scientists draw trees upside down. This convention allows us to grow trees down from the root since most people find it more natural to write from top to bottom. You are more likely to see the root at the 'top' with the leaves at the 'bottom' of trees. Trees implemented with a linked structure can also be pictured like this:

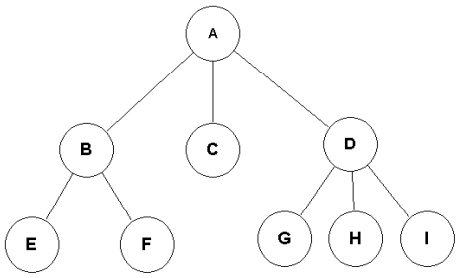


A nonempty tree is a collection of nodes with one node designated as the **root**. Each **node** contains a reference to an element and has **edges** connecting it to other nodes, which are also trees. These other nodes are called children. A tree can be empty — have no nodes. Trees may have nodes with two or more children.

A leaf is a node with no children. In the tree above, the nodes with 4, 5, and 6 are leafs. All nodes that are not leaves are called the internal nodes of a tree, which are 1, 2, and 3 above. A leaf node could later grow a nonempty tree as a child. That leaf node would then become an internal node. Also, an internal node might later have its children become empty trees. That internal node would become a leaf.

A tree with no nodes is called an empty tree. A single node by itself can be considered a tree. A structure formed by taking a node N and one or more separate trees and making N the parent of all roots of the trees is also a tree. This recursive definition enables us to construct trees from existing trees. After the construction, the new tree would contain the old trees as subtrees. A subtree is a tree by itself. By definition, the empty tree can also be considered a subtree of every tree.

All nodes with the same parent are called siblings. The level of a node is the number of edges it takes to reach that particular node from the root. For example, the node in the tree above containing J is at level 2. The height of a tree is the level of the node furthest away from its root. These definitions are summarized with a different tree where the letters A through I represent the elements.

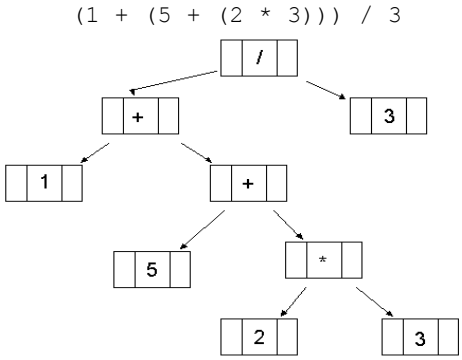


root	A
internal nodes	A B D
leaves	E F C G H I
height	3
level of root	0
level of node with F	2
nodes at level 1	3
parent of G, H and I	D
children of B	E F

A **binary tree** is a tree where each node has exactly two binary trees, commonly referred to as the left child and right child. Both the left or right trees are also binary trees. They could be empty trees. When both children are empty trees, the node is considered a leaf. Under good circumstances, binary trees have the property that you can reach any node in the tree within $\log_2 n$ steps, where n is the number of nodes in the tree.

Expression Tree

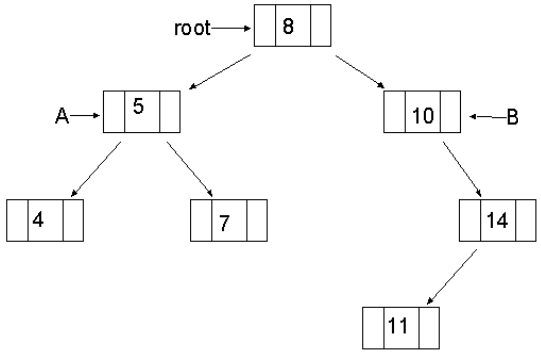
An **expression tree** is a binary tree that stores an arithmetic expression. The tree can then be traversed to evaluate the expression. The following expression is represented as a binary tree with operands as the leaves and operators as internal nodes.



Depending on how you want to traverse this tree — visit each node once — you could come up with different orderings of the same expression: infix, prefix, or postfix. These tree traversal algorithms are presented later in this chapter.

Binary Search Tree

Binary Search Trees are binary trees with the nodes arranged according to a specific ordering property. For example, consider a binary search tree that stores `Integer` elements. At each node, the value in the left child is less than the value of the parent. The right child has a value that is greater than the value of its parent. Also, since the left and right children of every node are binary search trees, the same ordering holds for all nodes. For example, all values in the left subtree will be less than the value in the parent. All values in the right subtree will be greater than the value of the parent.



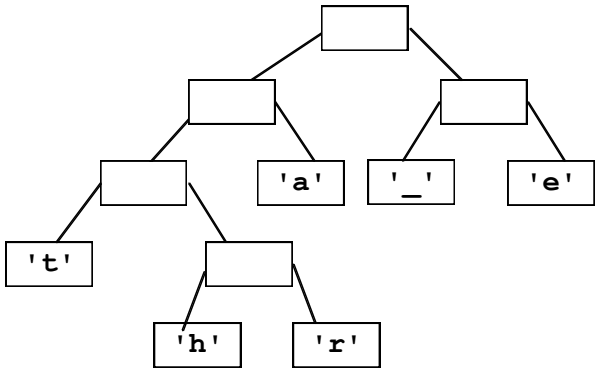
The left child of the root (referenced by **A**) has a value (5) that is less than the value of the root (8). Likewise, the value of the right child of the root has a value (10) that is greater than the root's value (8). Also, all the values in the subtree referenced by **A** (4, 5, 7), are less than the value in the root (8).

To find the node with the value 10 in a binary search tree, the search begins at the root. If the search value (10) is greater than the element in the root node, search the binary search tree to the right. Since the right tree has the value you are looking for, the search is successful. If the key is further down the tree, the search keeps going left or right until the key is found or the subtree is empty indicating the key was not in the BST. Searching a binary search tree can be O(log n) since half the nodes are removed from the search at each comparison. Binary search trees store large amounts of real world data because of their fast searching, insertions, and removal capabilities. The binary search tree will be explored later in this chapter.

Huffman Tree

David Huffman designed one of the first compression algorithms in 1952. In general, the more frequently occurring symbols have the shorter encodings. Huffman coding is an integral part of the standards for high definition television (HDTV). The same approach to have the most frequently occurring characters in a text file be represented by shorter codes, allows a file to be compressed to consume less disk space and to take less time to arrive over the Internet.

Part of the compression algorithm involves creation of a Huffman tree that stores all characters in the file as leaves in a tree. The most frequently occurring letters will have the shortest paths in the binary tree. The least occurring characters will have longer paths. For example, assuming a text file contains only the characters 'a', 'e', 'h', 'r', 't', and '_', the Huffman tree could look like this assuming that 'a', 'e', and '_' occur more frequently than 'h' and 'r'.



With the convention that 0 means go left and 1 right, the 6 letters have the following codes:

```
'a'  01
'_'  10
'e'  11
't'  000
'h'  0010
'r'  0011
```

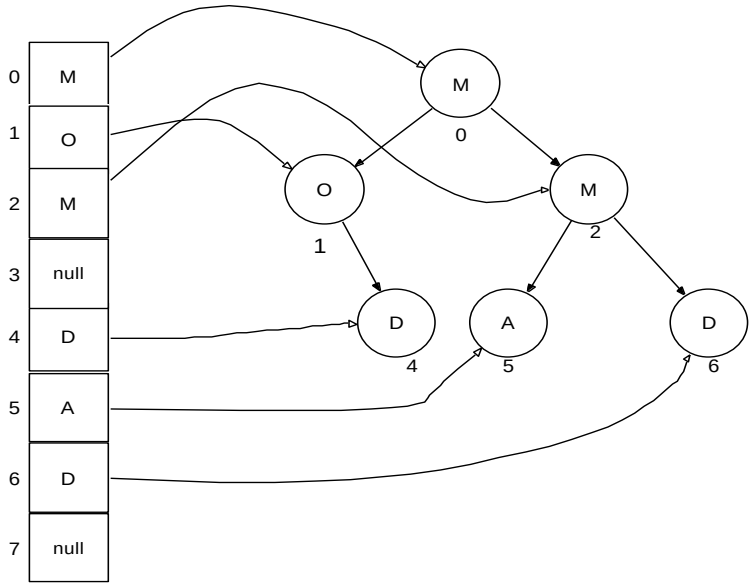
Instead of storing 8 bits for each character, the most frequently occurring letters in this example use only 2 or 3 bits. Some of the characters in a typical file would have codes for some characters that are much longer than 8 bits. These 31 bits represent a text file containing the text "tea_at_three".

```
000110110010001000010001111
  | | | | |  | |  |   |   | | |
  t e a _ a  t _  t   h   r e e
```

Assuming 8 bit ASCII characters, these 31 bits would require 12*8 or 96 bits.

19.2 Implementing Binary Trees

A binary tree can be represented in an array. With an array-based implementation, the root node will always be positioned at array index 0. The root's left child will be positioned at index 1, and the right child will be positioned at array index 2. This basic scheme can be carried out for each successive node counting up by one, and spanning the tree from left to right on a level-wise basis.



Notice that some nodes are not used. These unused array locations show the "holes" in the tree. For example, nodes at indexes 3 and 7 do not appear in the tree and thus have the `null` value in the array. In order to find any left or right child for a node, all that is needed is the node's index. For instance to find node 2's left and right children, use the following formula:

```
Left Child's Index   =  2 * Parent's Index + 1
Right Child's Index  =  2 * Parent's Index + 2
```

So in this case, node 2's left and right children have indexes of 5 and 6 respectively. Another benefit of using an array is that you can quickly find a node's parent with this formula:

```
Parent's Index = (Child's Index - 1) / 2
```

For example, (5-1)/2 and (6-1)/2 both have the same parent in index 2. This works, because with integer division, 4/2 equals 5/2.

Linked Implementation

Binary trees are often implemented as a linked structure. Whereas nodes in a singly linked structure had one reference field to refer to the successor element, a `TreeNode` will have two references — one to the left child and one to the right child. A tree is a collection of nodes with a particular node chosen as the root. Assume the `TreeNode` class will be an inner class with private instance variables that store these three fields

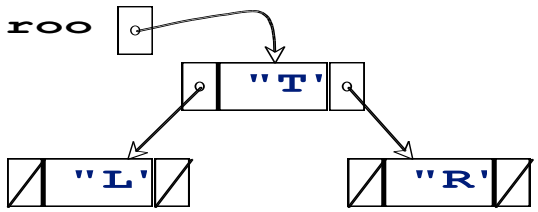
- a reference to the element
- a reference to a left tree (another `TreeNode`),
- a reference to a right tree (another `TreeNode`).

To keep things simple, the `TreeNode` class begins like this so it can store only strings. There are no generics (Chapter 12 will show a generic binary tree).

```
// A type to store an element and a reference to two other TreeNode objects
private class TreeNode {

  private String data;
  private TreeNode left;
  private TreeNode right;

  public TreeNode(String elementReference) {
    data = elementReference;
    left = null;
    right = null;
  }
}
```

The following three lines of code (if in the same class as this inner node class) will generate the binary tree structure shown:

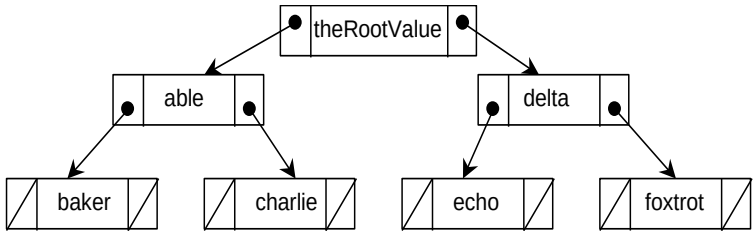
```
TreeNode root = new TreeNode("T");
root.left = new TreeNode("L");
root.right = new TreeNode("R");
```



Self-Check

11-1 Using the tree shown below, identify

a) the root
b) size
c) the leaves
d) the internal nodes
e) the children of delta
f) the number of nodes on level 4



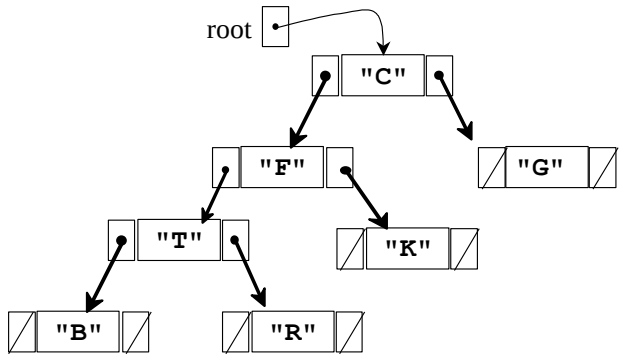
11-2 Using the `TreeNode` class above, write the code that generates the tree above.

Node as an Inner Class

Like the node classes of previous collections, this `TreeNode` class can also be placed inside another. However, instead of a collection class with an insert method, `hardCodeATree` will be used here to create a small binary tree. This will be the tree used to present several binary tree algorithms such as tree traversals in the section that follows.

```
// This simple class stores a collection of strings in a binary tree.
// There is no add or insert method. Instead a tree must be "hard coded" to
// demonstrate algorithms such as tree traversals, makeMirror, and height.
public class BinaryTreeOfStrings {

 private class TreeNode {
   private String data;
   private TreeNode left;
   private TreeNode right;
   public TreeNode(String elementReference) {
     data = elementReference;
      left = null;
      right = null;
    }
  }

  // The entry point into the tree
  private TreeNode root;

  // Construct and empty tree
  public BinaryTreeOfStrings() {
    root = null;
  }

  // Hard code a tree of size 6 on 4 levels
  public void hardCodeATree() {
    root = new TreeNode("C");
    root.left = new TreeNode("F");
    root.left.left = new TreeNode("T");
    root.left.left.left = new TreeNode("B");
    root.left.left.right = new TreeNode("R");
    root.left.right = new TreeNode("K");
    root.right = new TreeNode("G");
  }
}
```

The tree built in `hardCodeATree()`



19.3 Binary Tree Traversals

Code that traverses a linked list would likely visit the nodes in sequence, from the first element to the last. Thus, if the list were sorted in a natural ordering, nodes would be visited in from smallest to largest. With binary trees, the traversal is quite different. We need to stack trees of parents before visiting children. Common tree traversal algorithms include three of a possible six:

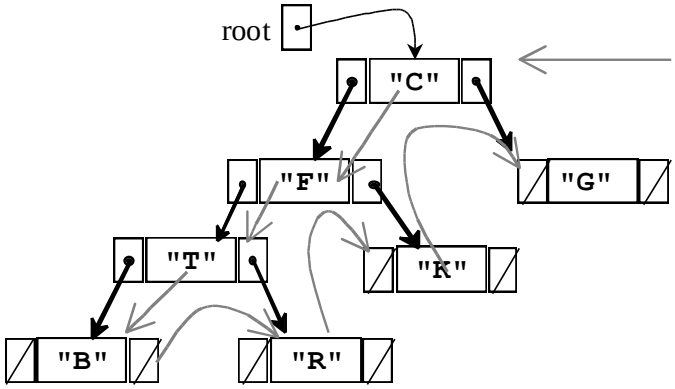
- Preorder traversal: Visit the root, preorder traverse the left tree, preorder traverse the right subtree
- Inorder traversal: Inorder traverse the left subtree, visit the root, inorder traverse the right subtree
- Postorder traversal: Postorder traverse the left subtree, postorder traverse the right subtree, visit the root

When a tree is traversed in a preorder fashion, the parent is processed *before* its children — the left and right subtrees.

Algorithm: Preorder Traversal of a Binary Tree

- Visit the root
- Visit the nodes in the left subtree in preorder
- Visit the nodes in the right subtree preorder

When a binary tree is traversed in a preorder fashion, the root of the tree is "visited" *before* its children — its left and right subtrees. For example, when `preorderPrint` is called with the argument `root`, the element **C** would first be visited. Then a call is made to do a preorder traversal beginning at the left subtree. After the left subtree has been traversed, the algorithm traverses the right subtree of the root node making the element **G** the last one visited during this preorder traversal.

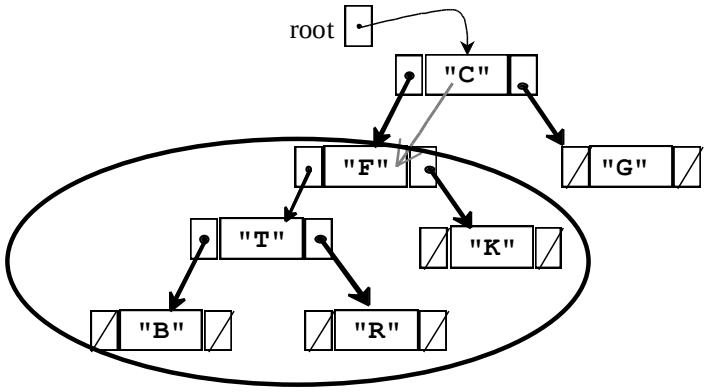


The following method performs a preorder traversal over the tree built in `hardCodeATree` (see above) that has the string "C" in the root node. Writing a solution to this method without recursion would require a stack and a loop. This algorithm is simpler to write with recursion.

```
public void preOrderPrint() {
```

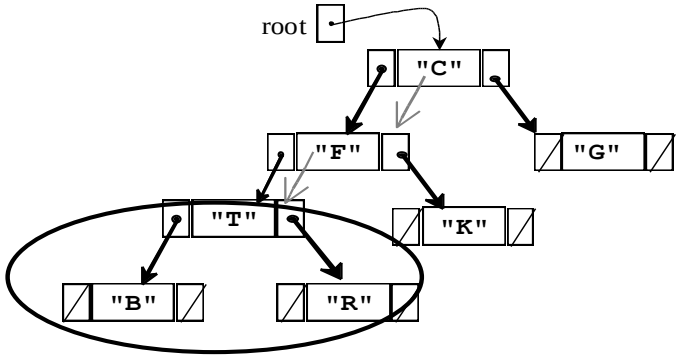
```
    preOrderPrint(root);
  }

  private void preOrderPrint(TreeNode tree) {
    if (tree != null) {
      // Visit the root
      System.out.print(tree.data + " ");
      // Traverse the left subtree
      preOrderPrint(tree.left);
      // Traverse the right subtree
      preOrderPrint(tree.right);
    }
  }
```

When the public method calls `preOrderPrint` passing the reference to the root of the tree, the node with **C** is first visited. Next, a recursive call passes a reference to the left subtree with **F** at the root. Since this TreeNode argument it is not null, **F** is visited next and is printed.



Preorder Traversal so far: **C F**

Next, a recursive call is made with a reference to the left subtree of **F** with **T** at the root, which is visited before the left and right subtrees.



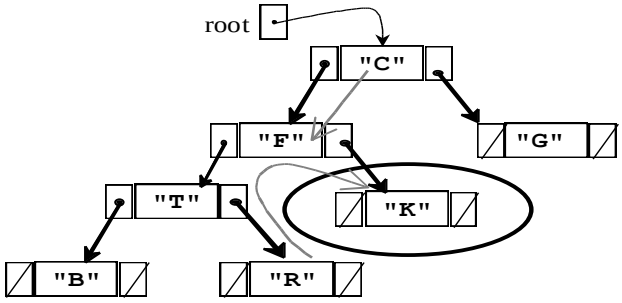
Preorder Traversal so far: C F T

After the root is visited, another recursive call is made with a reference to the left subtree **B** and it is printed. Recursive calls are made with both the left and right subtrees of B. Since they

are both null, the if statement is false and the block of three statement is skipped. Control returns to the method with **T** at the root where the right subtree is passed as the argument.

Preorder Traversal so far: C F T B R

The flow of control returns to visiting the right subtree of **F**, which is **K**. The recursive calls are then made for both of K's children (empty trees). Again, in both calls, the block of three statements is skipped since `t.left` and `t.right` are both `null`.



Preorder Traversal so far: C F T B R K

Finally, control returns to visit the right subtree in the first call with the root as the parameter to visit the right subtree in preorder fashion when **G** is printed.

Inorder Traversal

During an inorder traversal, each parent gets processed *between* the processing of its left and right children. The algorithm changes slightly.

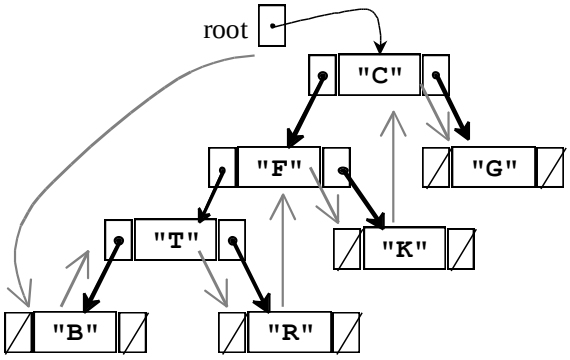
- Traverse the nodes in the left subtree inorder
- Process the root
- Traverse the nodes in the right subtree inorder

Inorder traversal visits the root of each tree only after its left subtree has been traversed inorder. The right subtree is traversed inorder after the root.

```
public void inOrderPrint() {
    inOrderPrint(root);
  }

  private void inOrderPrint(TreeNode t) {
    if (t != null) {
      inOrderPrint(t.left);
      System.out.print(t.data + " ");
      inOrderPrint(t.right);
    }
  }
```

Now a call to `inOrderPrint` would print out the values of the following tree as

B T R F K C G



The `inOrderPrint` method keeps calling `inOrderPrint` recursively with the left subtree. When the left subtree is finally empty, `t.left==null`, the block of three statements executed for **B**.

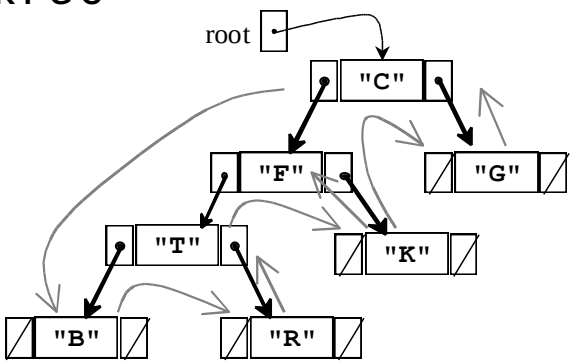
Postorder Traversal

In a postorder traversal, the root node is processed *after* the left and right subtrees. The algorithm shows the process step after the two recursive calls.

1. Traverse the nodes in the left subtree in a postorder manner
2. Traverse the nodes in the right subtree in a postorder manner
3. Process the root

A postorder order traversal would visit the nodes of the same tree in the following fashion:

B R T K F G C



The `toString` method of linear structures, such as lists, is straightforward. Create one big string from the first element to the last. A `toString` method of a tree could be implemented to return the elements concatenated in pre-, in-, or post-order fashion. A more insightful method would be to print the tree to show levels with the root at the leftmost (this only works on trees that are not too big). A tree can be printed sideways with a *reverse* inorder traversal. Visit the right, the root, and then the left.

```
    G
C
        K
    F
            R
        T
            B
```

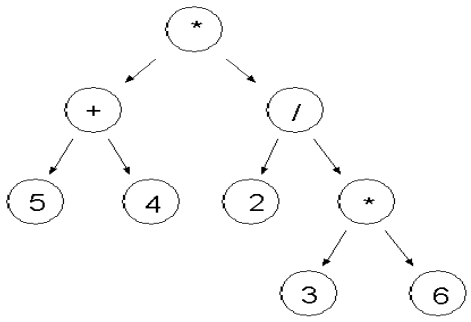
The `printSideways` method below does just this To show the different levels, the additional parameter `depth` begins at 0 to print a specific number of blank spaces depth times before each element is printed. When the root is to be printed depth is 0 and no blanks are printed.

```
public void printSideways() {
  printSideways(root, 0);
}

private void printSideways(TreeNode t, int depth) {
  if (t != null) {
    printSideways(t.right, depth + 1);
    for (int j = 1; j <= depth; j++)
      System.out.print("    ");
    System.out.println(t.data);
    printSideways(t.left, depth + 1);
  }
}
```

Self-Check

11-3 Write out the values of each node of this tree as the tree is traversed both

a. inOrder b. preorder c. postOrder



11-4 Implement the private helper method `postOrderPrint` that will print all elements separated by a space when this public method is called:

```
public void postOrderPrint() {
  postOrderPrint(root);
}
```

19.4 A few other methods

This section provides a few algorithms on binary trees, a few of which you may find useful.

height

The height of an empty tree is -1, the height of an tree with one node (the root node) is 0, and the height of a tree of size greater than 1 is the longest path found in the left tree from the root. The private `height` method first considers the base case to return -1 if the tree is empty.

```
// Return the longest path in this tree or -1 if this tree is empty.
public int height() {
  return height(root);
}

private int height(TreeNode t) {
  if (t == null)
    return -1;
  else
    return 1 + Math.max(height(t.left), height(t.right));
}
```

When there is one node, `height` returns 1 + the maximum height of the left or right trees. Since both are empty, `Math.max` returns -1 and the final result is (1 + -1) or 0. For larger trees, `height` returns the larger of the height of the left subtree or the height of the right subtree.

leafs

Traversal algorithms allow all nodes in a binary tree to be visited. So the same pattern can be used to search for elements, send messages to all elements, or count the number of nodes. In these situations, the entire binary tree will be traversed.

The following methods return the number of leafs in a tree When a leaf is found, the method returns 1 + all leafs to the left + all leafs to the right. If `t` references an internal node (not a leaf), the recursive calls to the left and right must still be made to search further down the tree for leafs.

```
public int leafs() {
  return leafs(root);
}

private int leafs(TreeNode t) {
  if (t == null)
    return 0;
  else {
    int result = 0;
    if (t.left == null && t.right == null)
      result = 1;
    return result + leafs(t.left) + leafs(t.right);
  }
}
```

findMin

The `findMin` method returns the string that precedes all others alphabetically. It uses a preorder traversal to visit the root nodes first (`findMin` could also use be a postorder or inorder traversal). This example show that it may be easier to understand or implement a binary tree algorithm that has an instance variable initialized in the public

method and adjusted in the private helper method.

```
// This instance variable is initialized it in the public method findMin.
private String min;

// Return a reference the String that alphabetically precedes all others
public String findMin() {
  if (root == null)
    return null;
  else {
    min = root.data;
    findMinHelper(root);
    return min;
  }
}

public void findMinHelper(TreeNode t) {
  // Only compare elements in nonempty nodes
  if (t != null) {
   // Use a preorder traversal to compare all elements in the tree.
   if (t.data.compareTo(min) < 0)
      min = t.data;
    findMinHelper(t.left);
    findMinHelper(t.right);
  }
}
```

Self-Check

11-5 To `BinaryTreeOfStrings`, add method `findMax` that returns the string that follows all others alphabetically.

11-6 To `BinaryTreeOfStrings`, add method `size` that returns the number of nodes in the tree.

11-7 To `BinaryTreeOfStrings`, add method `toString` that returns all elements inorder.

11-8 To `BinaryTreeOfStrings`, add method method `isFull` that returns true if the binary tree is full or false if it is not. A full tree is a binary tree in which each node has exactly zero or two children.

19.5 Binary Search Trees

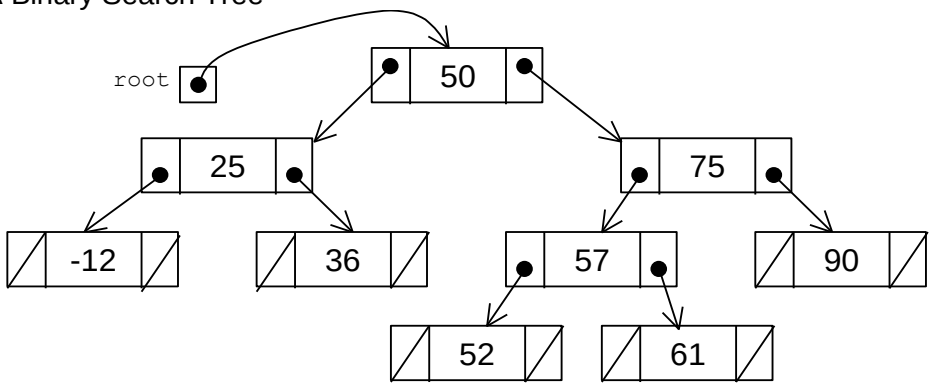
A Binary Search Tree is a binary tree with an ordering property that allows O(log n) retrieval, insertion, and removal of individual elements. Defined recursively, a binary search tree is

4. an empty tree, or
5. consists of a node called the root, and two children, left and right, each of which are themselves binary search trees. Each node contains data at the root that is greater than all

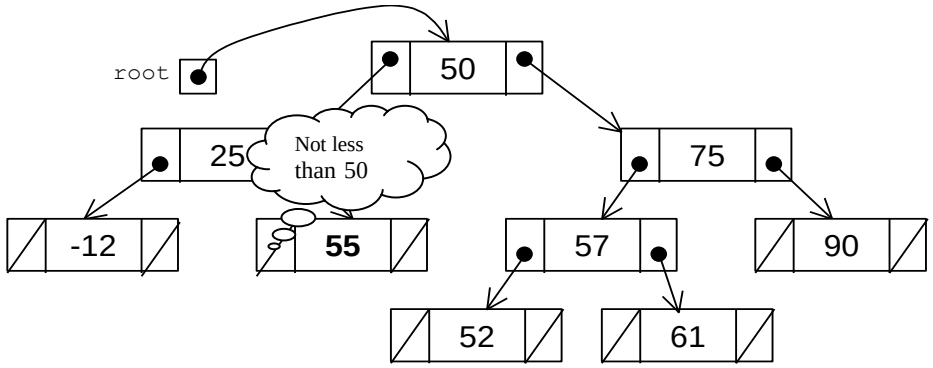
values in the left subtree while also being less than all values in the right subtree. No two nodes compare equally. This is called the binary search tree ordering property.

The following two trees represent a binary search tree and a binary tree respectively. Both have the same structure — every node has two children (which may be empty trees shown with `/)`. Only the first tree has the binary search ordering property. The second does not have the BST ordering property. The node containing 55 is found in the left subtree of 50 instead of the right subtree.

A Binary Search Tree



A Binary Tree that does not have the ordering property (55 in wrong place)



The BinarySearchTree class will add the following methods:

insert	Add an element to the binary search tree while maintaining the ordering property.
find	Return a reference to the element that "equals" the argument according to compareTo
remove	Remove the that "equals" while maintaining the ordering property (left as an exercise)

Java generics will make this collection class more type safe. It would be tempting to use this familiar class heading.

```
public class BinarySearchTree<E>
```

However, to maintain the ordering property, BinarySearchTree algorithms frequently need to compare two elements to see if one element is greater than, less than, or equal to another element. These comparisons can be made for types that have the compareTo method.

Java generics have a way to ensure that a type has the compareTo method. Rather than accepting any type with <E>, programmers can ensure that the type used to construct an instance does indeed implement the Comparable interface (or any interface that extends the Comparable interface) with this syntax:

```
public class BinarySearchTree <E extends Comparable<E>> {
```

This class heading uses a bounded parameter to restrict the types allowed in a `BinarySearchTree` to `Comparable`s only. This heading will also avoid the need to cast to `Comparable`. Using `<E extends Comparable <E>>` will also avoid cast exceptions errors at runtime. Instead, an attempt to compile a construction with a `NonComparable` — assuming `NonComparable` is a class that does not implement `Comparable` — results in a more preferable compile time error.

```
BinarySearchTree<String> strings = new BinarySearchTree<String>();
BinarySearchTree<Integer> integers = new BinarySearchTree<Integer>();
BinarySearchTree<NonComparable> no = new BinarySearchTree<NonComparable>();
                      ⇑
  Bound mismatch: The type NonComparable is not a valid substitute for the
  bounded parameter <E extends Comparable<E>>
```

So far, most elements have been String or Integer objects. This makes explanations shorter. For example, it is easier to write stringTree.insert("A"); than accountTree.insert(new BankAccount("Zeke Nathanielson", 150.00)); (and it is also easier for authors to fit short strings and integers in the boxes that represent elements of a tree).

However, collections of only strings or integers are not all that common outside of textbooks. You will more likely need to store real-world data. Then the find method seems more appropriate. For example, you could have a binary search tree that stores BankAccount objects assuming BankAccount implements Comparable. Then the return value from find could be used to update the object in the collection, by sending withdraw, deposit, or getBalance messages.

```
accountCollection.insert(new BankAccount("Mark", 50.00));
accountCollection.insert(new BankAccount("Jeff", 100.00));
accountCollection.insert(new BankAccount("Nathan", 150.00));

// Need to create a dummy object that will "equals" the account in the BST
BankAccount toBeMatched = new BankAccount("Jeff", -999);
```

```
BankAccount currentReference = accountCollection.find(toBeMatched);
assertNotNull(currentReference);
assertEquals("Jeff", currentReference.getID());

accountCollection.printSideways();
currentReference.deposit(123.45);

System.out.println("After a deposit for Jeff");
accountCollection.printSideways();
```

Output (Notice that the element with ID Jeff changes):

```
    Nathan $150.00
Mark $50.00
    Jeff $100.00
After a deposit for Jeff
    Nathan $150.00
Mark $50.00
    Jeff $223.45
```

Linked Implementation of a BST

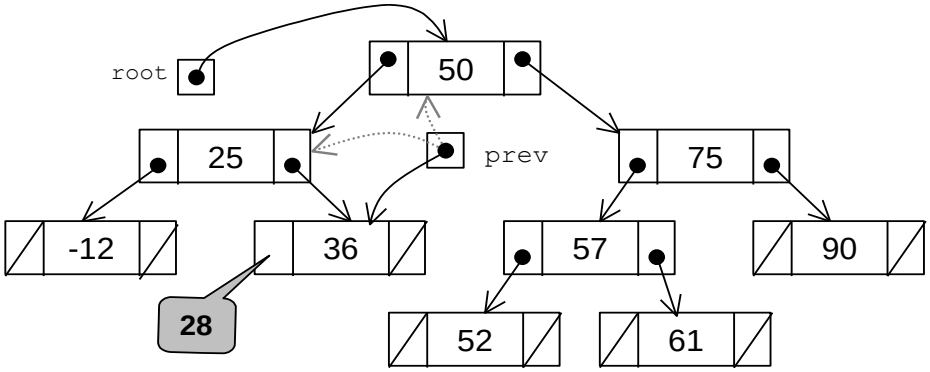
The linked implementation of a binary search tree presented here uses a private inner class `TreeNode` that stores the type `E` specified as the type parameter. This means the nodes can only store the type of element passed as the type argument at construction (which must implement `Comparable` or an interface that extends interface `Comparable`).

```
// This simple class stores a collection of strings in a binary tree.
// There is no add or insert method. Instead a tree must be "hard coded" to
// demonstrate algorithms such as tree traversals, makeMirror, and height.
public class BinarySearchTree<E extends Comparable<E>> {

  private class TreeNode {

    private E data;
    private TreeNode left;
    private TreeNode right;

    TreeNode(E theData) {
      data = theData;
      left = null;
      right = null;
    }
  }

  private TreeNode root;

  public BinarySearchTree() {
    root = null;
  }

  // The insert and find methods will be added here
}
```

insert

A new node will always be inserted as a leaf. The insert algorithm begins at the root and proceeds as if it were searching for that element. For example, to insert a new `Integer` object

with the value of 28 into the following binary search tree, 28 will first be compared to 50. Since 28 is less than the root value of 50, the search proceeds down the left subtree. Since 28 is greater than 25, the search proceeds to the right subtree. Since 28 is less than 36, the search attempts to proceed left, but stops. The tree to the left is empty. At this point, the new element should be added to the tree as the left child of the node with 36.



The search to find the insertion point ends under either of these two conditions:

1. A node matching the new value is found.
2. There is no further place to search. The node can then be added as a leaf.

In the first case, the insert method could simply quit without adding the new node (recall that binary search trees do not allow duplicate elements). If the search stopped due to finding an empty tree, then a new `TreeNode` with the integer 28 gets constructed and the reference to this new node replaces one of the empty trees (the `null` value) in the leaf last visited. In this case, the reference to the new node with 28 replaces the empty tree to the left of 36.

One problem to be resolved is that a reference variable (named `curr` in the code below) used to find the insertion point eventually becomes `null`. The algorithm must determine where it should store the reference to the new node. It will be in either the left link or the right link of the node last visited. In other words, after the insertion spot is found in the loop, the code must determine if the new element is greater than or less than its soon to be parent.

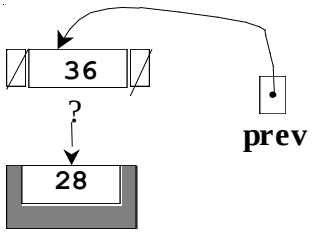
Therefore, two reference variables will be used to search through the binary search tree. The `TreeNode` reference named `prev` will keep track of the previous node visited. (Note: There are other ways to implement this).

The following method is one solution to insertion. It utilizes the Binary Search Tree ordering property. The algorithm checks that the element about to be inserted is either less than or greater than each node visited. This allows the appropriate path to be taken. It ensures that the new element will be inserted into a location that keeps the tree a binary search tree. If the new element to be inserted compares equally to the object in a node, the insert is abandoned with a `return` statement.

```
public boolean insert(E newElement) {
    // newElement will be added and this will still be a
    // BinarySearchTree. This tree will not insert newElement
    // if it will compareTo an existing element equally.
    if (root == null)
```

```
      root = new TreeNode(newElement);
    else {
      // find the proper leaf to attach to
      TreeNode curr = root;
      TreeNode prev = root;

      while (curr != null) {
        prev = curr;
        if (newElement.compareTo(curr.data) < 0)
          curr = curr.left;
        else if (newElement.compareTo(curr.data) > 0)
          curr = curr.right;
        else {
          System.out.println(newElement + " in this BST");
          return false;
        }
      }

      // Correct leaf has now been found. Determine whether to
      // link the new node came from prev.left or prev.right
      if (newElement.compareTo(prev.data) < 0)
        prev.left = new TreeNode(newElement);
      else
        prev.right = new TreeNode(newElement);
    }
    return true;
  } // end insert
```

When `curr` finally becomes `null`, it must be from either `prev`'s left or right.



This situation is handled by the code at the end of insert that compares `newElement` to `prev.data`.

find

This `BinarySearchTree` needed some way to insert elements before `find` could be tested so insert could be tested, a bit of illogicality. Both will be tested now with a unit test that begins by inserting a small set of integer elements. The `printSideways` message ensures the structure of the tree has the BST ordering property.

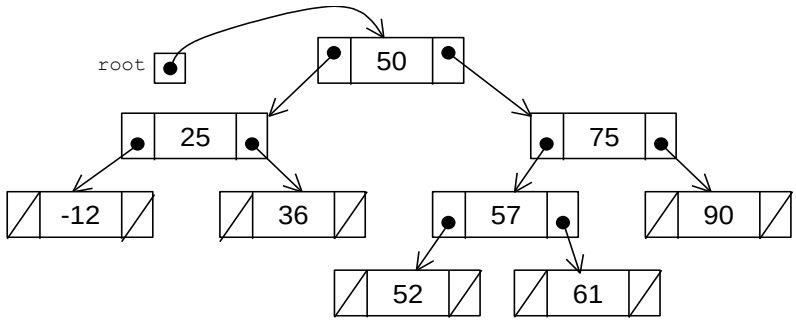
```
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.Before;

public class BinarySearchTreeTest {
```

```
  private BinarySearchTree<Integer> aBST;

  // Initialize aBST before each test method.
  // The test methods can always assume they begin with this BST.
  @Before
  public void setUpBST() {
    aBST = new BinarySearchTree<Integer>();
    aBST.insert(50);
    aBST.insert(25);
    aBST.insert(75);
    aBST.insert(-12);
    aBST.insert(36);
    aBST.insert(57);
    aBST.insert(90);
    aBST.insert(52);
    aBST.insert(61);
    aBST.printSideways();
  }

  // Any @Test in this unit test can use aBST with the same 9 integers
  // shown in @Before as seyUpBST will be called before each @Test
}
```

Output

```
        90
    75
            61
        57
            52
50
        36
    25
        -12
```

Picture the state of aBST



The first test method ensures that elements that can be added result in true and those that can't result in false. Programmers could use this to ensure the element was added or the element already existed.

```
@Test
  public void testInsertDoesNotAddExistingElements() {
    assertTrue(aBST.insert(789));
    assertTrue(aBST.insert(-789));
    assertFalse(aBST.insert(50));
    assertFalse(aBST.insert(61));
  }
```

This test method ensures that the integers are found and that the correct value is returned.

```
  @Test
  public void testFindWhenInserted() {
    assertEquals(50, aBST.find(50));
    assertEquals(25, aBST.find(25));
    assertEquals(75, aBST.find(75));
    assertEquals(-12,aBST.find(-12));
    assertEquals(36, aBST.find(36));
    assertEquals(57, aBST.find(57));
    assertEquals(90, aBST.find(90));
```

```
    assertEquals(52, aBST.find(52));
    assertEquals(61, aBST.find(61));
  }
```

And this test method ensures that a few integers not inserted are also not found.

```
  @Test
  public void testFindWhenElementsNotInserted() {
    assertNull(aBST.find(999));
    assertNull(aBST.find(0));
  }
```

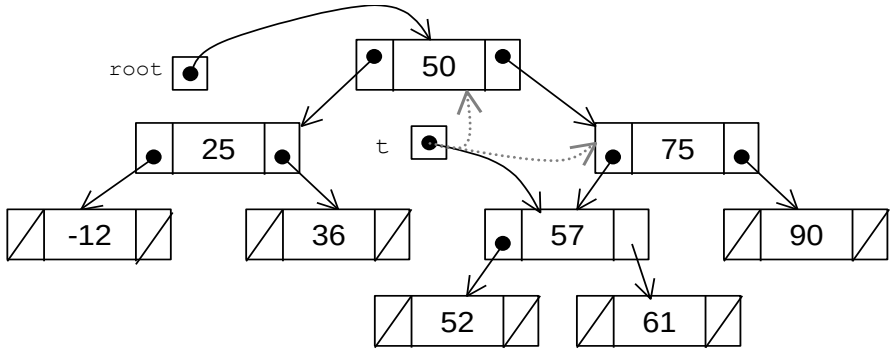
The search through the nodes of a aBST begins at the root of the tree. For example, to search for a node that will compareTo 57 equally, the method first compares 57 to the root element, which has the value of 50. Since 57 is greater than 50, the search proceeds down the *right* subtree (recall that nodes to the right are greater). Then 57 is compared to 75. Since 57 is less than 75, the search proceeds down the *left* subtree of 75. Then 57 is compared to the node with 57. Since these compare equally, a reference to the element is returned to the caller. The binary search continues until one of these two events occur:

1. The element is found
2. There is an attempt to search an empty tree (nowhere to go -- the node is not in the tree)

In the first case, the reference to the data in the node is returned to the sender. In the second case, the method returns null to indicate that the element was not in the tree. Here is an implementation of find method.

```
  // Return a reference to the object that will compareTo
  // searchElement equally. Otherwise, return null.
  public E find(E searchElement) {
    // Begin the search at the root
    TreeNode ref = root;
    // Search until found or null is reached
    while (ref != null) {
      if (searchElement.compareTo(ref.data) == 0)
        return ref.data; // found
      else if (searchElement.compareTo(ref.data) < 0)
        ref = ref.left; // go down the left subtree
      else
        ref = ref.right; // go down the right subtree
    }
    // Found an empty tree. SearchElement was not found
    return null;
  }
```

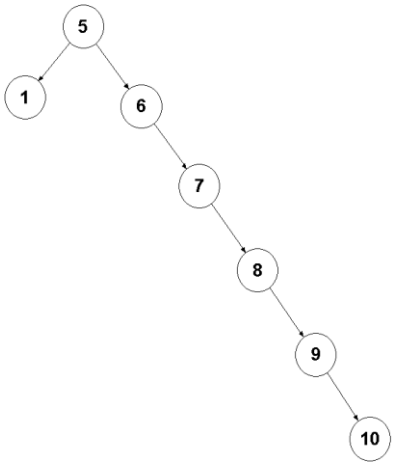
The following picture shows the changing values of the external reference t as it references the three different nodes in its search for 57:



One of the reasons that binary search trees are frequently used to store collections is the speed at which elements can be found. In a manner similar to a binary search algorithm, half of the elements can be eliminated from the search in a BST at each loop iteration. When you go left from one node, you ignore all the elements to the right, which is usually about half of the remaining nodes. Assuming the BinarySearchTree is fairly complete, searching in a binary search tree is O(log n). For example, in the previous search, t referred to only three nodes in a collection of size 9. A tree with 10 levels could have a maximum size of 1,024 nodes. It could take as few as 10 comparisons to find something on level 10.

9.6 Efficiency

Much of the motivation for the design of trees comes from the fact that the algorithms are more efficient than those with arrays or linked structures. It makes sense that the basic operations on a binary search tree should require O(log n) time where n is the number of elements of the tree. We know that the height of a balanced binary tree is $\log_2 n$ where *n* is the number elements in the tree. In this case, the cost to find the element should be on the order of O(log *n*). However, with a tree like the following one that is not balanced, runtime performance takes a hit.



If the element we were searching for was the right-most element in this tree (10), the search time would be O(n), the same as a singly linked structure.

Thus, it is very important that the tree remain balanced. If values are inserted randomly to a binary search tree, this condition may be met, and the tree will remain adequately balanced so that search and insertion time will be O(log *n*).

The study of trees has been very fruitful because of this problem. There are many variants of trees, e.g., red-black trees, AVL trees, B-trees, that solve this problem by re-balancing the tree after operations that unbalance it are performed on them. Re-balancing a binary tree is a very tedious task and is beyond the scope of this book. However, it should be noted that having to rebalance a binary tree every now and then adds overhead to the runtime of a program that requires a binary search tree. But if you are mostly searching, which is often the case, the balanced tree might be appropriate.

The table below compares a binary search tree's performance with a sorted array and singly linked structure (the remove method for BST is left as an exercise).

	Sorted Array	Singly Linked	Binary Search Tree
remove	O(n)	O(n)	O(log n)
find	O(log n)	O(n)	O(log n)
insert	O(n)	O(n)	O(log n)