Data Structures and Problem Solving Using Java™

(Third Edition)
Resouce Manual

Mark Allen Weiss
3rd ed IRM revised by Tim Herman

PEARSON
Addison Wesley

Boston San Francisco New York
London Toronto Sydney Tokyo Singapore Madrid
Mexico City Munich Paris Cape Town Hong Kong Montreal

CHAPTER

1 Primitive Java

1.2 Solutions To Exercises

In Short

1.1 Java source files end in `.java` . Compiled files (containing j-code or byte-codes) end in `.class` .

1.2 `//`, which extends to the end of the line and `/*` and `/**`, both of which extend to a `*/`. Comments do not nest.

1.3 `boolean` , `byte` , `short` , `char` , `int` , `long` , `float` , and `double` .

1.4 `*` multiplies two primitive values, returning the result and not changing its two arguments. `*=` changes the left-hand argument to the product of the left-hand argument and the right-hand argument. The right-hand argument is unchanged.

1.5 Both the prefix and postfix increment operators add one to the target variable. The prefix operator uses the new value of the variable in a larger expression; the postfix operator uses the prior value.

1.6 The `while` loop is the most general loop and performs a test at the top of the loop. The body is executed zero or more times. The `do` loop is similar, but the test is performed at the bottom of the loop; thus the body is executed at least once. The `for` loop is used primarily for counting-like iteration and consists of an initialization, test, and update along with the body.

1.7 `break` is used to exit a loop. A labelled `break` exits the loop that is marked with a label. `break` is also used to exit a `switch` statement, rather than stepping through to the next case.

1.8 The `continue` statement is used to advance to the next iteration of the loop.

1.9 Method overloading allows the reuse of a method name in the same scope as long as the signatures (parameter list types) of the methods differ.

1.10 In call-by-value, the actual arguments are copied into the method's formal parameters. Thus, changes to the values of the formal parameters do not affect the values of the actual arguments.

In Theory

1.11 After line 1, `b` is 6, `c` is 9, and `a` is 13. After line 2, `b` is 7, `c` is 10, and `a` is 16. After line 3, `b` is 8, `c` is 11, and `d` is 18. After line 4, `b` is 9, `c` is 12, and `d` is 21.

1.12 The result is `true`. Note that the precedence rules imply that the expression is evaluated as `(true&&false) || true`.

1.13 The behavior is different if `statements` contains a `continue` statement.

1.14 Because of call-by-value, `x` must be 0 after the call to method `f`. Thus the only possible output is `0`.

In Practice

1.15 An equivalent statement is:

```
while( true )
    statement
```

1.16 This question is harder than it looks because I/O facilities are limited, making it difficult to align columns.

```
public class MultiplicationTable
{
    public static void main( String [ ] args )
    {
        for( int i = 1; i < 10; i++ )
        {
            for( int j = 1; j < 10; j++ )
            {
                if( i * j < 10 )
                    System.out.print( " " );
                System.out.print( i * j + "   " );
            }
            System.out.println( );
        }
    }
}
```

1.17 The methods are shown below (without a supporting class); we assume that this is placed in a class that already provides the `max` method for two parameters.

```
public static int max( int a, int b, int c )
{
    return max( max( a, b ), c );
}
```

```
public static int max( int a, int b, int c, int d )
{
   return max( max( a, b ), max( c, d ) );
}
```

1.18 The method is below:

```
public static boolean isLeap( int year )
{
   return year % 4 == 0 &&
      ( year % 100 != 0 || year % 400 == 0 );
}
```

CHAPTER

2 References

2.2 Solutions To Exercises

In Short

2.1 Reference values (logically) store the address where an object resides; a primitive value stores the value of a primitive variable. As a result, operations such as == have seemingly different meanings for reference types and primitive types.

2.2 The basic operations the can be applied to a reference type are assignment via `=`, comparison via == and `!=`, the dot operator, type conversion, and `instanceof` .

2.3 An array has its size associated with it. An `ArrayList` has a capacity in addition to size. Adding an element to an `ArrayList` will automatically expand the capacity of the array if needed.

2.4 Exceptions are thrown by a method. The exception immediately propagates back through the calling sequence until it is handled by a matching `catch` clause, at which point the exception is considered handled.

2.5 Basic string operations include `equals` and `compareTo` to compare, `=` to copy, `+` and `+=` to perform concatenation, `length` , `charAt` , and `substring` .

In Theory

2.6 The second statement outputs `5 7`, as expected. The first outputs `44`, because it used the ASCII value of `' '`, which is 32.

In Practice

2.9 A method to do this is below:

```
public static boolean isPrefix( String str1, String str2 )
{
    if( str1.length( ) > str2.length( ) )
        return false;
    for( int i = 0; i < str1.length( ); i++ )
        if( str1.charAt( i ) != str2.charAt( i ) )
            return false;
    return true;
}
```

2.10
```
public static int getTotalStrLength(String [] theStrings )
{
        int total = 0;
        for( String s : theStrings )
             total += s.length( );

    return total;
 }
```

2.11 The elements in the original array are not copied before it is reinitialized.

2.12

```
public static String [ ] split ( String str, String tokens )
{
     //use an ArrayList to hold the strings
     ArrayList<String> strList = new ArrayList<String>( );

     //go through string, looking for next token each time
   int start = 0;
   int end = str.indexOf( tokens, start );
   String s;

   while (end > -1){
        s= str.substring( start, end );
        if ( s.length( ) > 0 )
                strList.add( s );
        start = end + tokens.length( );
       .end  = str.indexOf( tokens, start );
   }

   //add the last token
   s = str.substring( start );
   if ( s.length( ) > 0 )
   ......strList.add( s );

   //convert the ArrayList to a String array and return it
   return  ( String[ ] ) strList.toArray( new String[ 0 ] );
}
```

CHAPTER

3 Objects and Classes

3.2 Solutions To Exercises

In Short

3.1 *Information hiding* makes implementation details, including components of an object, inaccessible. *Encapsulation* is the grouping of data and the operations that apply to them to form an aggregate while hiding the implementation of the aggregate. Encapsulation and information hiding are achieved in Java through the use of the class.

3.2 Members in the public section of a class are visible to non-class routines and can be accessed via the dot member operator. Private members are not visible outside of the class.

3.3 The constructor is called when an object is created by a call to `new`.

3.4 The default constructor is a member-by-member application of a default constructor, meaning that primitive members are initialized to zero and reference members are initialized to `null`.

3.5 `this` is a reference to the current object. This reference to the current object can be used to compare it with other objects or to pass the current object to some other unit.

3.6 Packages are used to organize similar classes. Members with no visibility modifier are package visible: that is, they are visible to other classes within the same package (it is not visible outside the package).

3.7 Output is typically performed by providing a `toString` method that generates a `String`. This `String` can be passed to `println` .

3.8 The two import directives are below:

```
import weiss.nonstandard.*;
import weiss.nonstandard.Exiting;
```

3.9 A design pattern describes a commonly occurring problem in many contexts and a generic solution that can be applied in a wide variety of these contexts.

3.10 (a) Line 17 is illegal because `p` is a non-static field and therefore needs an object reference in order to be reference in the `main` method (which is a static method). Line 18 is legal as `q` is created within the `main` method. (b) Line 20 and 23 are legal as `NO_SSN` is a public static field which can be accessed via an object reference or a class name. Line 22 is legal because by default, `name` is visible. Line 21 and 24 are illegal because `SSN` is private. Also, `Person.SSN` is illegal as `SSN` is nonstatic.

In Theory

3.11 By defining a single private constructor for a class A, we can disallow any other object to create an instance of A. For example, we may use A to hold only static data.

3.12 (a) Yes; `main` works anywhere. (b) Yes; if `main` was part of class `Int-Cell` , then `storedValue` would no longer be considered private to `main`.

3.13

```
public class Combolock {

private int num1;
private int num2;
private int num3;

public Combolock ( int a, int b, int c){
       num1 = a;
       num2 = b;
       num3 = c;
}

//returns true if the proper combination is given
public boolean open ( int a, int b, int c){
      return ( ( a == num1 ) && ( b == num2 ) && (c == num3 ) );
}
public boolean changeCombo( int a, int b, int c, int newA, int newB, int
newC ){
       if ( open( a, b, c) ){
...           num1 = newA;
...           num2 = newB;
...           num3 = newC;
...           return true;
       }
       return false;
}
```

In Practice

3.14 The suprising result in part (d) is that the compiler will find the bytecodes for the local List class, even though you have recommented it. And thus, no ambiguity will be reported. Even with the import directive, the local list class will win out over `java.awt.List` .

CHAPTER

4 Inheritance

4.2 Solutions To Exercises

In Short

4.1 Assuming public inheritance, only public and protected members of the base class are visible in the derived class unless the derived class is in the same package of the base class, in which case package-friendly members are also visible. Among base class members, only public members of the base class are visible to users of the derived class.

4.2 Private inheritance is used to indicate a *HAS-A* relationship while avoiding the overhead of a layer of function calls. Composition is a better alternative for this. In composition, a class is composed of objects of other classes.

4.3 Polymorphism is the ability of a reference type to reference objects of several different types. When operations are applied to the polymorphic type, the operation that is appropriate to the actual referenced object is automatically selected.

4.4 Autoboxing and unboxing are new features introduced in Java 1.5. The perform automatic conversion between primitive types and their object counterparts (e.g. int and java.lang.Integer).

4.5 A final method is a method that cannot be redefined in a derive class.

4.6 (a) Only `b.bPublic` and `d.dPublic` are legal. (b) if `main` is in `Base`, then only `d.dPrivate` is illegal. (c) If `main` is in `Derived`, then `b.bPrivate` is illegal. (d) Assuming `Tester` is in the same package, then `b.bProtect` is visible in all cases. (e) place the statements below in the public section of their respective classes; (f) and (g) `bPrivate` is the only member not accessible in Derived.

```
Base( int pub, int pri, int pro )
{
   bPublic = pub; bPrivate = pri; bProtect = pro;
}

Derived( int bpub, int bpri, int bpro, int dpub, int dpri )
{
   super( bpub, bpri, bpro );
   dPublic = dpub; dPrivate = dpri;
}
```

4.8 An abstract method has no meaningful definition. As a result, each derived class must redefine it to provide a definition. A base class that contains an abstract method is abstract and objects of the class cannot be defined.

4.9 An interface is a class that contains a protocol but no implementation. As such it consists exclusively of public abstract methods and public static final fields. This is different from abstract classes, which may have partial implementations.

4.10 The Java I/O library classes can be divided into two groups, input and output. The I/O is done via input and output streams. Byte oriented reading and writing is provided via `InputStream` and `OutputStream` classes whereas character-oriented I/O is provided via `Reader` and `Writer` classes. Most other classes for I/O (for files, pipes, sockets etc) are derived from these four classes.

```
Object
   Reader
      InputstreamReader
      BufferedReader
      FileReader
Writer
      Printwriter
InputStream
      FileInputStream
      SocketInputStream
      DataInputStream
      ObjectInputStream
OutputStream
      FileOutputStream
      DataOutputStream
      ObjectOutputStream
      GZIPOutputStream
      SocketOutputStream
```

4.11 Generic algorithms are implemented by using `Object` or some other interface (such as `Comparable`) as parameters and return types. In other words, generic algorithms are implemented by using inheritance. To store primitive types in a generic container, wrapper classes must be used.

4.12 A wrapper pattern is used to store an existing entity (e.g., a class) and add operations that the original type does not support. An adapter pattern is used to change the interface of an existing class to conform to another interface. Unlike the wrapper pattern, the aim of the adapter pattern is not to change or add functionality.

4.13 One way to implement an adaptor is via composition wherein a new class is defined that creates an instance of the original class and redefines the interface. The other method is via inheritance. Inheritance adds new methods and may leave the original methods intact whereas via composition, we create a class that implements just the new interface. A function object is implemented as a class with no data object and one method.

4.14 A local class is a class that is placed inside a method. The local class is visible only inside the method.An anonymous class is a class with no name. It is often used to implement function objects (the syntax allows writing `new Interface()` with the implementation of `Interface` immediately following it).

4.15 Type erasure is the process of converting a generic class to a non-generic implementation. Due to type erasure, generic types cannot be primitive types. Simirlarly, instanceof tests cannot be used with generic types, nor can a generic type be instantiated or referenced in a static context. Arrays of generic types cannot be created.

In Theory

4.17 Consider the case in which a method `foo()` contains a local class and returns an object of type `LocalClass`. In this case, if the local class accesses `x`, then it continues to be available after foo terminates.

```
Typetest foo ()
{
   final int x = 1;

   class LocalClass() implements Typetest
   {
      public int foo()
         { return x; }
   }
   return new localClass();
}
```

4.18 The output is 512 and the signature is `String getX()`. After changing as given in (c), the output is 17 and the signature is `int getX()`. If `Class1` is changed as in (d), we get an error as `Class2` is expecting a method `getX` returning an integer. If code inlining was allowed, the result would have been 17.

4.19 a. The last line throws a ClassCastException as the element in the array must be specifically cast, not the array itself. b. The last line contains an unnecessary cast. c. No errors. d. No errors.

4.20

„
```
public static <AnyType> void copy( AnyType [ ]  arr1, AnyType [ ] arr2 ){
     for ( AnyType val : arr1 )
           arr2[2] = val;
}
```

In Practice

4.21 The class below includes the generic method min and a test routine. max is similar.

```
public class Ex0420
{
   public static Comparable min( Comparable a, Comparable b )
   {
      return a.lessThan( b ) ? a : b;
   }

   public static void main( String [ ] args )
   {
      int x = 5, y = 7;

      System.out.println( min( new MyInteger( x ), new MyInteger( y ) ) );
   }
}
```

4.22

```
public static Comparable min( Comparable [] a )
{
   int minIndex = 0;
   for ( int i = 1; i < a.length; i++ )
   if ( a[minIndex].compareTo(a[i]) > 0 )
      minIndex = i;
   return a[minIndex];
}
```

4.23

```
class MaxTwo
{
   public static Comparable [] max2( Comparable [] a )
   {
      Comparable [] obj1 = new Comparable [2];
      int maxIndex0 = 0;
      int maxIndex1 = 0;

      if( a[0].compareTo(a[1]) > 0)   {  maxIndex0 = 1; }
      else                            { maxIndex1 = 1; }

      for( int i = 1; i < a.length; i++ )
      {
         if( a[maxIndex0].compareTo(a[i]) > 0 )
         {
            maxIndex1 = maxIndex0;
            maxIndex0 = i;
         }
      }
      obj1[0] = a[maxIndex0];
      obj1[1] = a[maxIndex1];
      return obj1;
   }
```

```
    public static void main( String [] args )
    {
        String [] st1 = { “B”, “C”, “A”, “F” };
        Comparable [] st2 = new Comparable[2];
        st2 = max2(st1);
        System.out.println(st2[0] + “ “ + st2[1]);
    }
}
```

4.24

```
class SortWithMin
{
    public static int findmin( Comparable [] a, int start )
    {
        int minIndex = start;

        for( int i = start; i < a.length; i++ )
            if( a[minIndex].compareTo(a[i]) > 0 )
                minIndex = i;
        return minIndex;
    }
    public static void sort (Comparable [] a)
    {
        int tempIndex;
        Comparable temp;

        for (int i = 0; i < a.length; i++)
        {
            tempIndex = findmin(a,i);
            temp = a[i];
            a[i] = a[tempIndex];
            a[tempIndex] = temp;
            }
        }

    public static void main (String [] args)
    {
        String [] st1 = { “B”, “C”, “A”, “F” };
        sort(st1);

        for( int i = 0; i < st1.length; i++ )
            System.out.println(st1[i]);
        Shape [] a = new Shape[ ]
        {  new Circle( 2.0 ), new Circle( 1.0), new Circle( 3.0 )
        };
        sort(a);
        for( int i = 0; i < a.length; i++ )
        System.out.println(a[i].area());
    }
}
```

This assumes that `Shape` implements `Comparable` (Figure 4.16).

4.25

```
public class Circle extends Shape
{
   public Circle( double rad )
   {
      if( rad >= 0 )
         radius = rad;
      else
         throw new InvalidArgumentException();
   }

   private double radius;
}

public class Square extends Rectangle
{
   public Square( double side )
   {
      super( side, side );
   }

   public String toString( )
   {
      return "Square: " + getLength( );
   }
}

public class Rectangle extends Shape
{
   public Rectangle( double len, double wid )
   {
      if( (len >= 0) && (wid >= 0) )
      {
         length = len; width = wid;
      }
      else
         throw new InvalidArgumentException();
   }

   private double length;
   private double width;
}

public class  InvalidArgumentException
               extends RuntimeException
{
}
```

4.26

```
class Person implements Comparable
{
    public int compareTo(Object rhs)
    {
        Person p2 = (Person) rhs;
        return (name.compareTo(p2.name));
    }

    // Class is identical, with changes in bold.
}
```

4.27

```
public class SingleBuffer <AnyType> {

        private AnyType theItem;
        private boolean isEmpty;

        public SingleBuffer( ){
                theItem = null;
                isEmpty = true;
        }

        public SingleBuffer( AnyType item ){
                theItem = item;
                if ( theItem != null ){
                        isEmpty = false;
                } else {
                        isEmpty = true;
                }
        }
        public AnyType get( ) throws SingleBufferException {
                if ( isEmpty ){
                        throw new SingleBufferException( "Buffer is empty!" );
                }
                isEmpty = true;
                AnyType item = theItem;
                theItem = null;
                return item;
        }
        public void put( AnyType item ) throws SingleBufferException {
                if ( !isEmpty ){
                        throw new SingleBufferException( "Buffer must be emptied by
        insertion!" );
                }
                theItem = item;
                isEmpty = false;
        }
}

public class SingleBufferException extends Exception {

        public SingleBufferException( String msg ){
                super( msg );
        }
}
```

4.28 Here is an even fancier version that uses iterators, inner classes, nested classes, and comparators. Of course it has way too many forward references for use at this point in the course, but you can remove complications as needed.

```
import java.util.*;
/**
 * A SortedArrayList stores objects in sorted order.
 * The SortedArrayList supports the add operation.
 * size is also provided, and so is a method
 * that returns an Iterator.
 * Of course a get routine could be provided too
 * as could numerous other routines..
 *
 * This example illustrates both instance inner classes
 * and static inner classes.
 * An instance inner class is used in the typical iterator pattern.
 * A static inner class is used to define a default comparator.
 */
class SortedArrayList
{
    private ArrayList data = new ArrayList( );       // The list, in sorted order
    private Comparator cmp;                          // The comparator object

    /**
     * Construct the SortedArrayList with specified Comparator.
     * @param compare The Comparator object.
     */
    public SortedArrayList( Comparator compare )
    {
        cmp = compare;
    }

    /**
     * Construct the SortedArrayList using natural ordering
     * If objects are not Comparable, an exception will be
     * thrown during an add operation.
     */
    public SortedArrayList( )
    {
        this( new DefaultComparator( ) );
    }

    private static class DefaultComparator implements Comparator
    {
        public int compare( Object obj1, Object obj2 )
        {
            return ((Comparable) obj1).compareTo( obj2 );
        }
    }
```

```
   /**
    * Add a new value to this SortedArrayList, maintaining sorted order.
    * @param x The Object to add.
    */
   public void add( Object x )
   {
      data.add( x );   // add at the end for now
      int i = data.size( ) - 1;
         // Slide x over to correct position
      for( ; i > 0 && cmp.compare( data.get( i - 1 ), x ) > 0; i– )
         data.set( i, data.get( i - 1 ) );
      data.set( i, x );
   }

   /**
    * Return the number of items in this SortedArrayList.
    * @return the number of items in this SortedArrayList.
    */
   public int size( )
   {
      return data.size( );
   }

   /**
    * Return an Iterator that can be used to traverse
    * this SortedArrayList. The remove operation is unimplemented.
    * @return An Iterator that can be used to traverse this SortedArrayList.
    */
   public Iterator iterator( )
   {
      return new SortedIterator( );
   }

   private class SortedIterator implements Iterator
   {
      private int current = 0;

      public boolean hasNext( )
      {
         return current < size( );
      }

      public Object next( )
      {
         return data.get( current++ );
      }

      public void remove( )
      {
         throw new UnsupportedOperationException( );
      }
   }
}
```

```
class TestSortedArrayList
{
   public static String listToString( SortedArrayList list )
   {
      Iterator itr = list.iterator( );
      StringBuffer sb = new StringBuffer( );

      for( int i = 0; itr.hasNext( ); i++ )
         sb.append( "[" + i + "]" + itr.next( ) + " " );
      return new String( sb );
   }
      // Test by inserting 20 Strings
   public static void main( String[] args )
   {
      SortedArrayList list = new SortedArrayList( );

      for( int i = 0; i < 20; i++ )
         list.add( "" + i );
      System.out.println( "Using iterator" );
      System.out.println( listToString( list ) );
   }
}
```

4.29

```
public interface Matchable
{
   boolean matches(int a);
}

class EqualsZero implements EqZeroFn
{
   public boolean matches(int a)
   {
      return a == 0;
   }
 }

class P428
{
   public static int countMatches( int [] a, Matchable func )
   {
      int num = 0;
      for( int i = 0; i < a.length; i++ )              if( func.matches(a[i])
   )
            num++;

      return num;
   }

   public static void main (String [] args)
   {
      int [] a = { 4, 0, 5, 0, 0};
      System.out.println(countMatches(a, new EqualsZero()));
   }
}
```

4.30

```
class EqualsK implements Matchable
{
   private int k;

   public EqualsK( int initialk )
      { k = initialk; }

   public EqualsK()
      { this( 0 ); }

   public boolean matches(int a)
      { return a == k; }
}

class P429
{
   public static void main (String [] args)
   {
      int [] a = { 4, 0, 5, 0, 0};
      System.out.println( P428.countMatches( a, new EqualsK(5) ) );
   }
```

CHAPTER

5 Algorithm Analysis

5.2 Solutions To Exercises

In Short

5.1 (a) 5, 5, 6; (b) 5, 6, 6; (c) 3, 3, 3; (d) 5, 5, 6.

5.2 Theorem 5.2 does not eliminate enough subsequences. If all the numbers are positive, for example, the theorem is not used at all by the algorithm.

5.3 (a) is true; (b) is true; (c) is false (example: $T_1(N) = N$ and $T_2(N) = 1$); (d) is false (same example).

5.4 All except *x* belong to the same group.

5.5 (a) Program *A*'s guarantee is ten times better than *B*'s. (b) Program *B*'s guarantee is ten times better than *A*'s. (c) the stated properties do not tell us anything about the average-case performance. (d) It is possible that program *B* will run faster on all inputs; the guarantee for *B* may be too pessimistic.

5.6 (a) The search could fail because the loop is exited even when a sub-array of one exists but has not been tested. (In fact, half the successful searches will be reported as unsuccessful); (b) `mid` will not be in the center, and may not even be in the range `low` to `high`; (c) Subarrays of size two which reach line 16 will not be reduced to a smaller subarray, resulting in an infinite loop; (d) Same problem as in part (c).

In Theory

5.7 (a) $O(N)$; (b) $O(N^2)$; (c) $O(N^2)$.

5.8 The running time is $O(N)$, assuming that each multiplication is unit cost. This assumption is somewhat unjustified since it is clear that the numbers are getting bigger at each step. But at this level, deeper calculations are not needed. If we take the size of `x` as a constant, then the *i*th multiplication costs $O(i)$, and the total running time is quadratic.

5.9 We need to calculate: $\sum_{i=1}^{N}\sum_{j=i}^{N}\sum_{k=i}^{j} 1$. This is

$$= \sum_{i=1}^{N}\sum_{j=i}^{N}(j-i+1)$$

$$= \sum_{i=1}^{N}\frac{(N-i+1)(N-i+2)}{2}$$

$$= \frac{1}{2}\sum_{i=1}^{N} i^2 - \left(N+\frac{3}{2}\right)\sum_{i=1}^{N} i + \frac{1}{2}(N^2+3N+2)\sum_{i=1}^{N} 1$$

$$= \frac{1}{2}\frac{N(N+1)(2N+1)}{6} - \left(N+\frac{3}{2}\right)\frac{N(N+1)}{2} + \frac{1}{2}(N^2+3N+2)N$$

$$= \frac{N^3+3N^2+2N}{6}$$

5.10 $N(N+1)/2$ times. The calculation is similar to, but simpler than, that in the previous exercise.

5.11 About (a) 2.5 milliseconds (5 times longer); (b) 3.5 milliseconds (about 7 times longer 5 * log(500)/log(100)); (c) 12.5 milliseconds (25 times longer); (d) 62.5 milliseconds (125 times longer).

5.12 In this problem, we are given 120,000 times as much time.
(a) For a linear-time algorithm, we can solve a problem 120,000 times as large, or 12,000,000 (assuming sufficient resources);
(b) For an $N\log N$ algorithm it is a little less (about 4,000,000 or so);
(c) For a quadratic algorithm, we can solve a problem $\sqrt{120000}$ = 346 times as large, so we can solve a problem of size 34,600;
(d) For a cubic algorithm, we can solve a problem $\sqrt[3]{120000}$ = 49 times as large, so we can solve an instance of size 4,900.

5.13 The cubic algorithm would take about 38 minutes for $N = 10{,}000$, 26 days for $N = 100{,}000$, and 72 centuries for $N = 10{,}000{,}000$. The quadratic algorithm would take 1.23 seconds for $N = 10{,}000$, about 2.25 minutes for $N = 100{,}000$ and about 16 days for $N = 10{,}000{,}000$. The $N \log N$ algorithm would use about 42 seconds for $N = 10{,}000{,}000$. These calculations assume a machine with enough memory to hold the array. The linear algorithm solves a problem of size 10,000,000 in about 0.32 seconds.

5.14 $2/N$, 37, $\sqrt{N}$, $N \log \log N$, $N \log N$, $N \log N$, $N \log(N^2)$, $N \log^2 N$, $N^{1.5}$, N^2, $N^2 \log N$, N^3, $2^{N/2}$, 2^N. $N \log N$ and $N \log(N^2)$ grow at the same rate.

5.15 The analysis below will agree with the simulation in all cases. Fragment 1: The running time is $O(N)$. Fragment 2: The running time is $O(N^2)$ because of two nested loops of size N each. Fragment 3: The loops are consecutive, so the running time is $O(N)$. Fragment 4: The inner loop is of size N^2 so the running time is $O(N^3)$. Fragment 5: The running time is $O(N^2)$. Fragment 6: Here we have three nested loops of size N, N^2, and N^2, respectively, so the total running time is $O(N^5)$.

5.16 Here the `if` statement is executed at most N^3 times, but it is only true $O(N^2)$ times (because it is true exactly i times for each i). Thus the innermost loop is only executed $O(N^2)$ times. Each time through it takes $O(N^2)$ time, for a total of $O(N^4)$.

5.17 (a) $2^{2^{N-1}}$; (b) $1 + \lceil \log \log D \rceil$.

5.19 I suppose the simplest example is an array of one item. What I meant by this question was to construct an example where almost every element is examined asymptotically. A sequence of N items in which the first is 1, the others are consecutive integers ending in N^2 is an example of a bad case. For instance, with $N = 10$, the sequence 1, 92, 93, 94, 95, 96, 97, 98, 99, 100 with search for 91 is bad and results in half the array being searched (without the technical adjustment we get almost the entire array searched). By using larger numbers, we can have even more entries searched.

5.20 Suppose for simplicity that the number of elements N is one less than a power of 2, that is, $N = 2^k - 1$. Then we see that there are exactly $2^i - 1$ items whose successful search requires exactly i element accesses. Thus the k average cost is

$$(1/N)\sum_{i=1}^{k} i2^{i-1} = ((k-1)2^k + 1)/N.$$

This evaluates to $\log(N + 1) - 1 + (\log(N + 1))/N$, which tells us that the cost of an average successful search is only about 1 less than the worst case successful search.

In Practice

5.23 Note that we require integers; use a variation of binary search to get a logarithmic solution (assuming that the array is preread).

5.24 (a) This is trial division; see Chapter 9 for a sample routine. The running time is $O(\sqrt{N})$; (b) $B = O(\log N)$; (c) $O(2^{B/2})$; (d) If a twenty-bit number can be tested in time T, then a forty-bit number would require about T^2 time.

5.26 First, a *candidate* majority element is found (this is the harder part). This candidate is the only element that could possibly be the majority element. The second step determines if this candidate is actually the majority element and is just a sequential search through the array. First, copy the array to an array A. To find a candidate in the array A, form a second array B. Then compare $A[1]$ and $A[2]$. If they are equal, add one of these to B; otherwise do nothing. Then compare $A[3]$ and $A[4]$. Again, if they are equal, add one of these to B; otherwise do nothing. Continue in this fashion until the entire array is read. If A has an odd number of elements, the last element is included in B if and only if B would otherwise have an even number of elements. We then move B back to A and repeat the process until eventually there is one element which is the candidate. If no candidate emerges, there is no majority element. This algorithm is linear.

Programming Projects

5.29 It is very difficult to see the $O(N \log \log N)$ behavior.

5.30 Note to the instructor: See Exercise 8.20 for an alternative algorithm.

CHAPTER

6 Collections API

6.2 Solutions To Exercises

In Short

6.1 (a) The removed items are 6 and then 1; (b) the removed items are 4 and then 8; (c) the removed items are 1 and then 4.

6.2 (a).

```
public static int count ( Collection<Collection<String>> c, String str )
{

      int count = 0;

      // go through each String object
      // in each Collection object in Collection c
      for( Collection strC : c ){
              for( Object o :  strC ){
                     String s = (String) o;
                   if ( s.equals( str ) )
```

```
                        count++;
                }
        }

        return count;
    }
```

(b). O (N^2)

(c). 18 milliseconds

In Theory

6.3 Yes. The simplest way is with a binary search tree (appropriately maintained to ensure worst-case performance).

6.4 The search tree and priority queue can be used to implement subquadratic sorts.

6.5 Let e be the extended stack (that supports the `findMin`). We will implement `e` with two stacks. One stack, which we will call `s`, is used to keep track of the `push` and `pop` operations, and the other, m, keeps track of the minimum. To implement `e.push(x)` , we perform `s.push(x)` . If `x` is smaller than or equal to the top element in stack `m`, then we also perform `m.push(x)` . To implement `e.pop()` , we perform `s.pop()` . If the former top item is equal to the top element in stack m, then we also `m.pop(x)` . `e.findMin()` is performed by examining the top element in `m`. All these operations are clearly constant time.

6.6 A double-ended queue is easily implemented in constant time by extending the basic queue operations.

In Practice

6.7

```
public static void printReverse( Collection c )
{
    java.util.Stack s = new java.util.Stack( );
    Iterator itr = c.iterator();

        while( itr.hasNext() )
            s.push( itr.next());

        while( !s.isEmpty() )
            System.out.println( s.pop() );
}
```

Programming Projects

6.10 `getFront` returns the item in position 0. This is a constant time operation. `enqueue` places the inserted item in the array position indicated by the current size and then increases the current size. This is also a constant time operation. dequeue slides all the items one array position lower and decreases the current size. Relevant error checks are performed in all cases. `dequeue` takes $O(N)$ time.

6.11 As usual, error checks must be performed; these are omitted from the description. `insert` performs a binary search to determine the insertion point; elements after the insertion point are slid one array position higher and the current size is incremented. `remove` performs a binary search to find the deleted item, and if found, items in higher array positions are slid one position lower and the current size is decremented. Both operations take linear time.

6.12 `findMin` is a constant time algorithm; array position zero stores the minimum. `deleteMin` is a linear-time algorithm; items in the array are slid one position lower. `insert` into a sorted list was done in Exercise 6.11.

6.13 `findMin` and `deleteMin` are linear-time scans; `deleteMin` can be finished by moving the last item into the deleted position. `insert` is implemented in constant time by placing the new element in the next available location.

6.14 `insert` adds to the next available location and then a comparison is performed to decide if it is the new minimum. If so, the extra member is updated. This is clearly constant time. `findMin` just examines the extra member. `deleteMin` is implemented by swapping the last item into the deleted position. However, the extra member that stores the position of the minimum element must then be updated and this requires a linear scan.

6.15 The smallest element is at the end, so `findMin` takes constant time. `deleteMin` can remove this element and update the current size; since no other data movement is needed, this is also constant time. `insert` is implemented by a binary search to find the insertion point and then linear data movement, and an update of the current size.

6.16 If we use a sorted array, (smallest in position 0), all operations are linear except for the following which are constant time: `findMin` , `findMax` , `deleteMax` . If we use an unsorted array, `insert` is constant time, all others are linear. If we use an unsorted array with two variables that store the position of the minimum and maximum, then all operations are constant time except `deleteMin` and `deleteMax` .

6.17 If a sorted array is used, then `findKth` is implemented in constant time and the other operations in linear time. The algorithms are similar to the previous exercises.

6.19
```
// Fill all positions in List l with Object value
public static void fill( List l, Object value )
{
      for ( int i = 0; i < l.size( ); i++){
            l.set( i, value );
      }
}
```

6.20
```
public static void reverse( List l )
{
      for ( int i = 0; i < ( l.size( )/2 ); i++ ){
            int swapIndex = ( l.size( )-1 ) - i;
            Object a = l.get( i );
            Object b = l.get( swapIndex );
            l.set( i, b );
            l.set( swapIndex, a );
      }
}
```

6.21
```
//removes every other element in the list
public static void removeOdd( List l )
{
     boolean odd = false;
     for ( int i = 0; i < l.size(); i++ ){
           if ( odd ){
                 l.remove(i);
                 i--;
           }
           odd = !odd;
     }
}
```

6.22
```
public static Map<String, String> swapPairs( Map<String, String> m )
{
      HashMap<String, String> newMap = new HashMap<String, String>( );
      Set keys = m.keySet();
      for ( Object o : keys ){
            String k = (String) o;
            String v = m.get( k );
            if ( newMap.containsKey( v ) ){
                  throw new IllegalArgumentException( "Original Map cannot con-
 tain duplicate values!" );
            } else {
                  newMap.put( v, k );
            }
      }
      return newMap;
}
```

CHAPTER

7 Recursion

7.2 Solutions To Exercises

In Short

7.1 1. Base cases: Always have at least one case that can be solved without using recursion; 2. Make progress: Every recursive call must make progress toward a base case; 3. Always assume the recursive call works; 4. Compound interest rule: Never duplicate work by solving the same instance of a problem in separate recursive calls.

7.3 In the first case, the %p is missing (see what happens when N is a power of 2). Even if the %p is placed at the end of the line, it is no good because the intermediate results become too large. In the second and third cases, when N is 2, we do not make progress. In the fourth case, we violate the compound interest rule and the result is $O(N)$ multiplications.

7.4 $2^{63}\text{mod } 37 = (2\ 4^{31}\ 37\text{mod }))\text{mod } 37$

$4^{63}\text{mod } 37 = (4\ 16^{15}\text{mod } 37))\text{mod } 37$

$16^{63}\text{mod}37 = (\ 16\ (256^{7}\text{mod } 37\ ))\text{mod } 37$

$= (16\ (34^{7}\text{mod } 37))\text{mod } 37$

$34^{7}\text{mod } 37 = (34\ (1156^{3}\text{mod } 37))\text{mod } 37$

$= (34\ (9^{3}\text{mod } 37))\text{ mod } 37$

$9^{3}\text{mod } 37 = (9\ (81^{1}\text{mod } 37))\text{mod } 37$

$= 26$

$34^{7}\text{mod } 37 = (34 \cdot 26)\text{mod } 37 = 33$

$16^{63}\text{mod } 37 = (16 \cdot 33)\text{mod } 37 = 10$

$4^{63}\text{mod } 37 = (4 \cdot 10\ )37\text{ mod} = 3$

$2^{63}\text{mod } 37 = (2 \cdot 3)37\text{ mod} = 6$

7.5 $gcd(1995,1492) = gcd(1492,503) = gcd(503,486) = gcd(486,17) = gcd(17,10) = gcd(10,7) = gcd(7,3) = gcd(3,1) = 1$.

7.6 $N = 1517$, $N' = 1440$. Suppose $e = 11$. Then $d = 131$.

7.7 If nickels are not part of United States Currency, then 30 cents is changed with a quarter and five pennies using the greedy algorithm, while 3 dimes would be optimal.

In Theory

7.8 The basis ($N = 0$ and $N = 1$) is easily verified to be true. So assume that the theorem is true for all $0 \le i < N$ and we will establish for N. Let $\phi_1 = (1 + \sqrt{5})/2$ and $\phi_2 = (1 - \sqrt{5})/2$. Observe that both ϕ_1 and ϕ_2 satisfy $\phi^N = \phi^{N-1} + \phi^{N-2}$ (this is verified by factoring and then solving a quadratic equation). Since $F_N = F_{N-1} + F_{N-2}$, by the inductive hypothesis we have

$$F_N = \frac{1}{\sqrt{5}}((\phi_1)^{N-1} - (\phi_2)^{N-1} + (\phi_1)^{N-2} - (\phi_2)^{N-2})$$
$$= \frac{1}{\sqrt{5}}((\phi_1)^N - (\phi_2)^N)$$

7.9 All of these identities are easily proved by induction.

7.10 (a) True because N clearly divides $(A + C) - (B + C) = A - B$; (b) True because N clearly divides $AD - BD = (A - B)D$; (c) True because $A^P - B^P$ has $A - B$ as a factor, so N must divide it.

7.11 If $B \le A / 2$, then the remainder must be less than $A / 2$ (by definition). Otherwise, the quotient is 1 and the remainder is $B - A < A / 2$. This implies that after two iterations, A has been at least halved, so the running time is logarithmic by the halving principle.

7.12 The proof is straightforward.

7.13 The proof is straightforward.

7.14 The proof is straightforward.

7.15

```
public static long gcd( long a, long b )
{
    if( a < b )
        return gcd( b, a );
    // a >= b
    if( b == 0 )
        return a;

    boolean aIsOdd = a % 2 == 1;
    boolean bIsOdd = b % 2 == 1;

    if( !aIsOdd && !bIsOdd )
        return 2 * gcd( a/2, b/2 );
    if( !aIsOdd &&  bIsOdd )
        return gcd( a/2, b );
    if(  aIsOdd && !bIsOdd )
        return gcd( a, b/2 );
    return gcd( (a+b)/2, (a-b)/2 );
}
```

7.16 The solution is

$$T(N) = \begin{cases} O(N^{\log_B A}) & \text{if } A > B^k \\ O(N^k \log^{P+1} N) & \text{if } A = B^k \\ O(N^k \log^P N) & \text{if } A < B^k \end{cases}$$

We prove the second case; the first and third are left to the reader. Observe $\log^P N = \log^P(B_M) = M^P \log^P B$. Working this through as in the text proof, we obtain

$$T(N) = T(B^M) = A^M \sum_{i=0}^{M} (B^k/A)^i i^P \log^P B$$

If $A = B^k$, then

$$T(N) = A^M \log^P B \sum_{i=0}^{M} i^P = O(A^M M^{P+1} \log^P B)$$

Since $M = (\log N) / (\log B)$ and $A^M = N^k$ and B is a constant, we obtain $T(N) = O(N^k \log^{P+1} N)$.

7.17 The recurrence is $T(N) = 7T(N / 2) + O(N^2)$. The solution is thus $T(N) = O(N^{\log_2 7}) = O(N^{2.81})$.

In Practice

7.18 The problem is that the absolute value of the most negative `int` is larger than the most positive `int`. Thus a special case must be added for it.

7.19 The method below assumes that N is positive.

```
public static int numOnes( long n )
{
    if( n <= 1 )
        return n;
    else
        return numOnes( n / 2 ) + n % 2;
}
```

7.20 A driver routine calls `binarySearch` with `0` and `n-1` as the last two parameters.

```
public static int binarySearch( Comparable [ ] a,
    Comparable x, int low, int high ) throws ItemNotFound
{
    if( low == high )
        if( a[ low ].compares( x ) == 0 )
            return low;
        else
            throw new ItemNotFound( );

    int mid = ( low + high ) / 2;

    if( a[ mid ].compares( x ) < 0 )
        return binarySearch( a, x, mid + 1, high );
    else
        return binarySearch( a, x, low, mid );
}
```

7.22 Let *Ways*(x, i) be the number of ways to make x cents in change without using the first i coin types. If there are N types of coins, then *Ways*(x, N) = 0 if $x \neq 0$ and *Ways*(0, i) = 1. Then *Ways*(x, i–1) is equal to the sum of *Ways*($x - pc_i$, i), for integer values of p no larger than x/c_i (but including 0).

7.23 Let *Sum*(x, i) be true if x can be represented using the first i integers in A. We need to compute *Sum*(N, K). This is easily done using a dynamic programming algorithm.

7.24 The following method returns `true` if there is a subset of `a` that sums to exactly `K`. Because there are two recursive calls of size N–1, the running time is $O(2^N)$.

```
public static boolean sum( int [ ] a, int n, int k )
{
    // Base Case
    if( n == 1 )
        return a[ 0 ] == k;
    // If sum exists without last item, return true
    if( sum( a, n - 1, k ) == 1 )
        return true;
    // Otherwise, use last item,
    // see if remaining sum exists
    return sum( a, n -1, k - a[ n - 1 ] );
}
```

7.25 The nonrecursive routine is straightforward. The recursive routine is shown below.

```
private void permute( char [ ] str, int low, int high )
{
    if( low < high )
        println( str );
    for( int i = low; i <= high; i++ )
    {
        char [ ] tmp = str.clone( ); // tmp will be str
        tmp[ i ]   = str[ low ]; // with i and low
        tmp[ low ] = str[ i ]; // swapped
        permute( tmp, low + 1, high );
    }
}
```

7.26 The squares will appear to move toward the viewer.

Programming Projects

7.31 This is the classic *blob problem*. The size of a group is one plus the size of all its neighbors and can be computed recursively. We must mark each neighbor as it is processed to avoid overcounting (and infinite recursion). The process can be viewed as a depth-first search.

CHAPTER 8 Sorting

8.2 Solutions To Exercises

In Short

8.1 (a) After *P*=2, the sequence is 1, 8, 4, 1, 5, 9, 2, 6, 5. After *P*=3, the sequence is 1, 4, 8, 1, 5, 9, 2, 6, 5. After *P*=4, the sequence is 1, 1, 4, 8, 5, 9, 2, 6, 5. After *P*=5, the sequence is 1, 1, 4, 5, 8, 9, 2, 6, 5. After *P*=6, the sequence is unchanged. After *P*=7, the sequence is 1, 1, 2, 4, 5, 8, 9, 6, 5. After *P*=8, the sequence is 1, 1, 2, 4, 5, 6, 8, 9, 5. After *P*=9, the sequence is sorted as 1, 1, 2, 4, 5, 5, 6, 8, 9. (b) After the 5-sort, the sequence is unchanged because it is already 5-sorted.After the 3-sort, the sequence is 1, 1, 4, 2, 5, 9, 8, 6, 5. The 1-sort completes the sort. (c) First the sequence 8, 1, 4, 1 is recursively sorted as 1, 1, 4, 8. Then the sequence 5, 9, 2, 6, 5 is recursively sorted as 2, 5, 5, 6, 9. The result is merged into a final sorted sequence. (d) is omitted because (e) is more informative: (e) After sorting the first, middle, and last elements, we have 5, 1, 4, 1, 5, 9, 2, 6, 8. Thus the pivot is 5. Hiding it gives 5, 1, 4, 1, 6, 9, 2, 5, 8. The first swap is between 6 and 2. The next swap crosses. After 5 is swapped back, we obtain 5, 1, 4, 1, 2, 5, 6, 9, 8. The first five elements are recursively sorted and so are the last three elements.

8.2 Insertion sort and mergesort are stable as long as the comparisons for equality do not destroy the order. The code in the text is stable. Neither quicksort nor shellsort are stable. It is easy for duplicate items to be disordered when one of them is compared against a third item.

8.3 First, selection via median-of-three gives a better than average pivot. Second, it avoids the need for a test to make sure that `j` does not run past the start of the array. Third, random number generation is expensive and occasionally the random number generators are faulty.

In Theory

8.4 (a) $O(N)$; (b) $O(N \log N)$, assuming the increment sequences described in the text (a logarithmic number of increments); (c) $O(N \log N)$; (d) $O(N \log N)$ for the implementation in the text in which both `i` and `j` stop on equality.

8.5 These answers are identical to the previous exercise: (a) $O(N)$; (b) $O(N \log N)$, assuming the increment sequences described in the text (that is, a logarithmic number of increments); (c) $O(N \log N)$; (d) $O(N \log N)$ for the implementation in the text in which both `i` and `j` stop on equality.

8.6 (a) $O(N^2)$; (b) $O(N \log N)$, assuming the increment sequences described in the text (the exact requirement is that each increment is within some multiplicative constant of the previous); (c) $O(N \log N)$; (d)$O(N \log N)$ for the implementation in the text in which both `i` and `j` stop on equality.

8.7 The inversion that existed between `a[i]` and `a[i+k]` is removed. This shows that at least one inversion is removed. For each of the k–1 elements `a[i+1]` , `a[i+2]` , ... `a[i+k-1]` , at most two inversions can be removed by the exchange. This gives a maximum of $2k - 1$.

8.8 (b) For 20 elements, here is a bad permutation for the median-of-three quicksort: 20, 3, 5, 7, 9, 11, 13, 15, 17, 19, 4, 10, 2, 12, 6, 14, 1, 16, 8, 18. To extend to larger amounts of data for even N: The first element is N, the middle is N–1, and the last is N–2. Odd numbers (except 1) are written in starting to the left of center in decreasing order. Even numbers are written in decreasing order by starting at the rightmost spot, and always skipping one available empty spot, and wrapping around when the center is reached. This method is suitable for a hand calculation, but takes $O(N \log N)$ time to generate a permutation. By inverting the actions of quicksort, it is possible to generate the permutation in linear time.

8.9 The recurrence is $T(N) = N - 1 + (1/N)\sum_{i=0}^{N-1} T(i)$ with an initial condition $T(1)= 0$ (this describes the number of comparisons performed on average). The solution is $T(N) = 2N + O(\log N)$ and is obtained in the same way as the bound for quicksort (the math is slightly simpler).

8.10 $T(N) \cong N \log N - N\log e$ (where $e = 2.71828...$).

8.11 Because $\lceil \log (4!) \rceil = 5$, at least 5 comparisons are required by any algorithm that sorts 4 elements. Let the four elements be A, B, C, and D. Compare and exchange, if necessary, $A$and B, and then C and D, so that $A < B$ and $C < D$. Then compare and exchange $A$and C, and then B and D so that $A < C$and $B < D$. At this point, A is the smallest and D the largest of the four elements. A fifth comparison establishes the order between B and C.

8.12 (a) In this algorithm, all elements up to `left` are equal to `pivot` and all elements after `right` are also equal to `pivot`. While partitioning, when `a[i]` is compared with `a[pivot]` , if they are equal, swap `a[i]` with `a[left]` . Do the same when `a[j]` is compared with `a[pivot]` . As a result, we will get an array in which all elements up to (but not including) `left` are equal to `pivot`, all elements between `left` and up to `i` as less than pivot, all elements from `j` onwards to right as greater than `pivot` and all elements from `right` onwards as equal to `pivot`. Now swap elements in the range `a[low..left-1]` with `a[i +low -left...i-1]`. Do a similar swap for elements equal to `pivot` on the right. (b) Each partition groups elements equal to the pivot at one place.If there are only d different values, then after d iterations of the partitioning algorithm, we will have all elements with same values grouped together in the sorted order. Since each partitioning takes $O(N)$ time, the total running time will be $O(dN)$.

8.13 The algorithm maintains two pointers, pointA and pointB, in arrays A and B respectively with an invariant (number of elements <= A[pointA] in A+ number of elements <= B[pointB] in B) = N). Initially, both point to the middle of the respective arrays. In each iteration, we first check if any one of A[pointA] or B[pointB] is the median. If not, then if A[pointA] is greater than B[pointB], then pointA is decreased and pointB is increased. Else, the opposite is done. The amount of increment to pointA or point B is $N/(2i+1)$ in iteration i.

In Practice

8.14 No; when `i` and `j` are both at items equal to the pivot, the result is an infinite loop.

8.15 Maintain an array indexed from 0 to 65,535 and initialized with all zeros. For each item read, increment the appropriate array entry. Then scan through the array and output the items in sorted order. Note that 65,535 is a constant, so anything that depends on it is a constant.

8.16

```
private static void quicksort( Comparable [ ] a, int low, int high )
{
   while  ( low + CUTOFF <= high )
   {
         // Sort low, middle, high
      int middle = ( low + high ) / 2;
      if( a[ middle ].compareTo( a[ low ] ) < 0 )
         swapReferences( a, low, middle );
      if( a[ high ].compareTo( a[ low ] ) < 0 )
         swapReferences( a, low, high );
      if( a[ high ].compareTo( a[ middle ] ) < 0 )
         swapReferences( a, middle, high );

         // Place pivot at position high - 1
      swapReferences( a, middle, high - 1 );
      Comparable pivot = a[ high - 1 ];
         // Begin partitioning
      int i, j;
      for( i = low, j = high -1; ; )
      {
         while( a[ ++i ].compareTo( pivot ) < 0 );
         while( pivot.compareTo( a[ —j ] ) < 0 );
         if( i >= j )
            break;
         swapReferences( a, i, j );
      }
         // Restore pivot
      swapReferences( a, i, high - 1 );
      quicksort( a, low, i - 1 );
      low = i + 1;
   }
   insertionSort( a, low, high );
}
```

8.17 The number of recursive calls is logarithmic because the smaller partition will always be less or equal to half the original size.

```
private static void quicksort( Comparable [ ] a, int low, int high )
{
   while ( low + CUTOFF <= high )
   {
         // Sort low, middle, high
      int middle = ( low + high ) / 2;
```

8.21 The algorithm can be succinctly stated as follows: for each `p`, see if there are two numbers that sum to exactly `-a[p]` . The initial sort costs $O(N \log N)$, the remaining work is N iterations of an$O(N)$ loop, for a total of $O(N^2)$. Note that in this algorithm, numbers may be repeated. It is possible to alter the algorithm to work if repeated numbers are not allowed.

8.22 This algorithm would require$O(N^3)$ space and a sort of these items would use $N^3 \log N$ time.

CHAPTER 9 Randomization

9.2 Solutions To Exercises

In Short

9.1 48271, 182605794, 1291394886, 1914720637, 2078669041, 407355683, 1105902161, 854716505, 564586691, 1596680831.

9.2 If $A = 2$, then although $2^{560} \equiv 1 \pmod{561}$, $2^{280} \equiv 1 \pmod{561}$ proves that 561 is not prime. If $A = 3$ then $3^{560} \equiv 375 \pmod{561}$, which proves that 561 is not prime. $A = 4$ does not fool the algorithm; as we saw above, $4^{140} \equiv 1 \pmod{561}$. However $A = 5$ is a false witness: $5^1 \equiv 5 \pmod{561}$, $5^2 \equiv 25 \pmod{561}$, $5^4 \equiv 64 \pmod{561}$, $5^8 \equiv 169 \pmod{561}$, $5^{16} \equiv 511 \pmod{561}$,

$5^{17} \equiv 311 \pmod{561}$, $5^{34} \equiv 229 \pmod{561}$, $5^{35} \equiv 23 \pmod{561}$, $5^{70} \equiv 529 \pmod{561}$, $5^{140} \equiv 463 \pmod{561}$, $5^{280} \equiv 67 \pmod{561}$, $5^{560} \equiv 1 \pmod{561}$.

9.3 The expected number of winners is 3 and satisfies a Poisson distribution. The probability of exactly 0 winners is $e^{-3} = 0.0498$. The probability of exactly 1 winner is $3e^{-3} = 0.149$.

9.4 If `seed` is 0, then all future values of `seed` will also be 0.

In Theory

9.5 The proof of correctness can be found in the paper by Park and Miller or in my textbook *Data Structures and Algorithm Analysis in Java*.

9.6 We only have to show that there is a sequence of swaps in the routine that corresponds to the generation of any permutation. This is easily seen inductively: if N is in position i, then the last swap was between position i and N–1. We work this logic backwards from N to 1, and conclude inductively that every permutation can be generated.

9.7 Flip the coins twice, number the outcomes. If both coins are identical, reflip both coins until they are not. If the latter flip is heads generate a 0, otherwise generate a 1.

Programming Projects

9.11 Th e expe cted numbe r o f r andom numb ers to fill the ith item is $N/(N - i + 1)$. Summing this gives a total of $O(N \log N)$

9.12 (a) Immediate from the description of the algorithm. (b) This can be shown by induction.

CHAPTER 10 Fun and Games

10.2 Solutions To Exercises

In Short

10.1 No check is performed to see if a line is too long. No check is made to make sure that the number of rows is sufficiently small. No checks are performed to make sure the characters are letters.

10.2 (a) H_{2C} and H_{2D} are refutations because a draw has already been established by C_1. (b) The position is a draw.

In Theory

10.3 Let `a[i]` be the smallest entry in `a` that stores a prefix of `x`. Since all larger positions store values greater than `x`, `a[mid]>x` for `mid>=i` . Thus the largest value `low` can assume is `i`; furthermore all smaller positions are greater than `x`. Thus, when the search is narrowed to one position, it must be narrowed to `low=mid` , at which point a prefix test can be performed.

10.4 (a) The running time doubles. (b) The running time quadruples.

In Practice

10.5 As the analysis in the text suggests, for large dictionaries, the sequential search will take significantly longer than the binary search.

10.6 The absence of the prefix test will affect performance, but not as significantly as if a sequential search was used instead of a binary search.

10.7 The difference in the performance is not very significant since the number of entries stored are roughly 300–400.

Programming Projects

10.14 The final score is 20-16 in favor of black.

CHAPTER

11 Stacks and Compilers

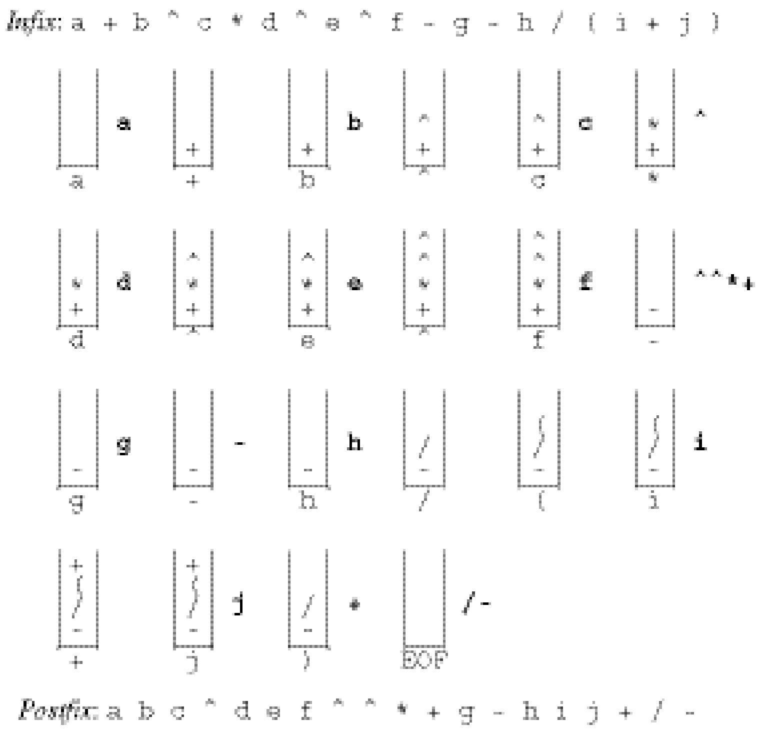
11.2 Solutions To Exercises

In Short

11.1 (a) An extraneous `}` is detected. (b) The `}` is detected to not match `(`. (c) Three messages are printed to indicate that `[` is unmatched. (d) The `)` is extraneous and the `(` is unmatched. (e) The `)` does not match the `[` and the `]` is extraneous.

11.2 (a) `1 2 + 3 4 ^ -`; (b) `1 2 ^ 3 4 * -`; (c) `1 2 3 * + 4 5 ^ - 6 +`; (d) `1 2 + 3 * 4 5 6 - ^ -`

11.3 (a) See Figure 11.1 (b) See Figure 11.2; t1 through t9 are temporaries. (c) See Figure 11.3.



Infix: a + b ^ c * d ^ e ^ f - g - h / (i + j)
Postfix: a b c ^ d e f ^ ^ * + g - h i j + / -

Figure 11.1 Infix-to-postfix for Exercise 11.3

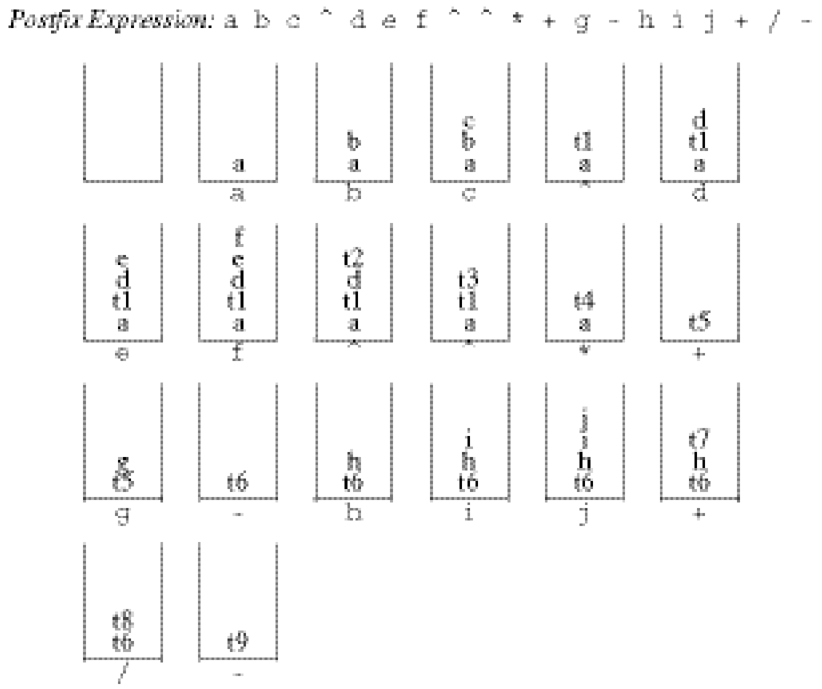


Figure 11.2 Postfix machine steps for Exercise 11.3

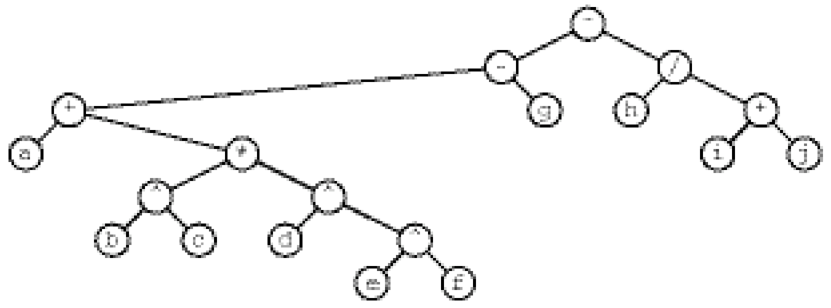


Figure 11.3 Expression tree for Exercise 11.3

In Theory

11.5 Unary minus and binary minus are considered different symbols. The unary minus operator pops only one symbol from the postfix stack instead of two and has precedence higher than the binary minus operator. It is right to left associative. To recognize a unary minus operator, we must remember if the last token matched was an operator; if so, we have a unary minus operator; otherwise we have a binary minus operator.

In Practice

11.6 The main difficulty is that the scanning routine must be modified to recognize the `**` token. This involves using lookahead when a `*` is seen, and pushing back a character if the next character is not the second `*`.

11.7 (a) 7; (b) Don't allow an operand to be accepted unless there has been an operator prior to it.

CHAPTER 12 Utilities

12.2 Solutions To Exercises

In Short

12.1 A Huffman tree is shown in Figure 12.1.

12.2 On UNIX, using `compress` , compression seems worthwhile for moderately large text files such as 50K or more. For large postscript files or Frame files, the savings can be 75% or more.

12.3 Because only leaves imply a code, Huffman's algorithm tends to notice missing bits quickly, and also tends to resynchronize quickly. Thus a few characters may be lost, but the rest of the transmission is likely to be intelligible.

In Theory

12.4 (a) Follows from the text discussion; (b) otherwise, we could swap a deep and more shallow node and lower the cost of the tree; (c) true, since the cost of the tree depends on the depth and frequency of a node.

12.5 (a) A 2-bit code is generated if a symbol is involved in the last merge. For this to occur, it must have frequency greater than 1/3 (so that when three symbols remain, it is not one of the smallest two trees). However, it is still possible that even with frequency less than 1/2, a node will have more than a two bit code. (b) Let F_N be the Nth Fibonacci number. If the character distribution is 1, 1, 1, 2, 3, 5, 8, 13, ..., then the first character will have a long code. Specifically, we can construct a Huffman tree of length N if the total character frequency is F_N. This is seen to be a worst-case scenario, so for a 20-bit code to occur, the frequency of a character must be at most $1/F_{20}$ (or about .0001), and probably somewhat less.

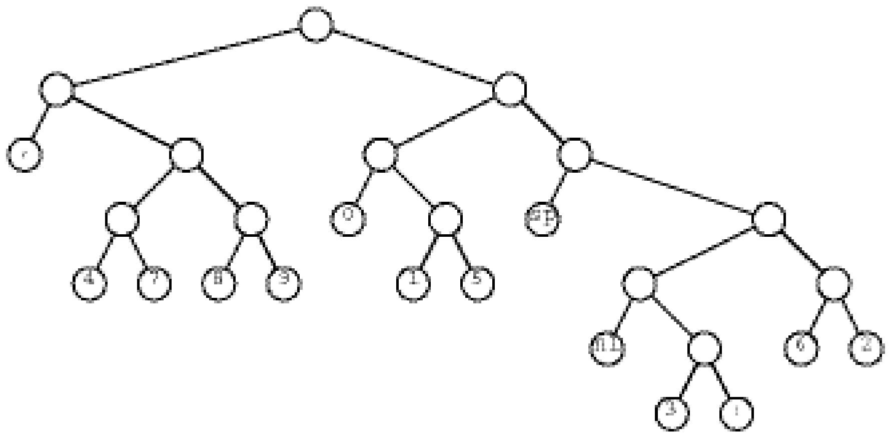


Figure 12.1 Huffman tree for Exercise 12.1

12.6 Maintain two queues $Q1$ and $Q2$. $Q1$ will store single-node trees in sorted order, and $Q2$ will store multi-node trees in sorted order. Place the initial sin-gle-node tree into $Q1$, enqueueing the smallest weight tree first. Initially $Q2$ is empty. Examine the first two entries of each of $Q1$ and $Q2$, and dequeue the two smallest. (This requires an easily implemented extension of the queue class). Merge the tree and place the result at the end of $Q2$. Continue this step until $Q1$ is empty and there is only one tree left in $Q2$.

12.7 In any coding scheme, the output file must contain the encoding table and then the compressed file. If the original file contains all symbols with the same frequency, there will not be any savings in terms of storage. The encoding table will be an extra overhead.

CHAPTER 13 Simulation

13.2 Solutions To Exercises

In Short

13.1 Player N wins.

13.2 Figure 13.1 shows that the order of elimination is 4, 1, 6, 5, 7, 3.

13.3 7 and 27 are two such values. They were obtained by writing a program to solve the Josephus problem for N= 30 and all M between 0 and 30.

13.4 (This is done in the text).

In Theory

13.5 After the first $N/2$ iterations, we are back at player 1, and half the players have been eliminated. In the next $N/4$ iterations, we eliminate another quarter of the players, and return back to player 1. Continuing in this way, we see that player 1 never gets eliminated and is thus the winner.

13.6 (a) Suppose $N$is even. After $N/2$ iterations, we are back at player 1 and only odd-numbered players remain. Renumber the players so that original player i becomes new player $(i + 1)/2$. The winner of the remaining game is new player J (N/2), which corresponds to old player 2J (N/2) – 1. (b) and (c) use the same logic.

13.7 A simple recursive algorithm will solve the problem in$O$(log N) time.

13.8 The solution uses the same ideas as Exercise 13.6 but is more complicated. The subproblems are of size 2N / 3.

13.9 10, 5, 2, 1, 3, 4, 7, 6, 8, 9, 15, 12, 11, 13, 14, 18, 16, 17, 19, 20. In other words, the insertion order is the same as a preorder traversal of the final tree.

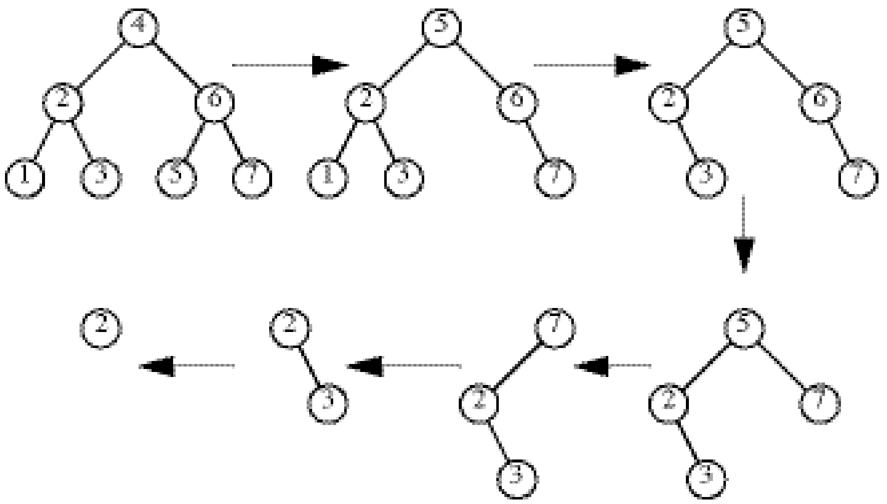


Figure 13.1 Stages of `BinarySearchTreeWithRank` when N = 7 and M = 3

In Practice

13.10 The running time of the algorithm would be O(N log N). The changes will require implementation of `findKth` element in a tree.

CHAPTER

14 Graphs and Paths

14.2 Solutions To Exercises

In Short

14.1 V_3->V_4, V_3->V_6, V_3->V_5, V_3->V_2, V_3->V_2->V_0, V_3->V_2->V_0->V_1.

14.2 V_3->V_4, V_3->V_2, V_3->V_6, V_3->V_6->V_5, V_3->V_2->V_0, V_3->V_2->V_0->V_1.

14.3 Dijkstra's algorithm can be used to compute shortest paths from a single source to all other nodes.

14.4 Changes to Figure 14.5: The adjacency lists will change as follows: `E` will have no outgoing edges, the first entry in the list for `D` will change to 43 with a pointer to `E` and the list for `C` will contain an additional entry of 10 with pointer to `D`. The first entry of `E` will be deleted. The shortest paths will change as follows: `A` to B is 12 with prev as `A`, `A` to `C` is infinity, `A` to `D` is 87 with prev as `A` and `A` to `E` is 23 with prev as `B`. The result of running the topological sort algorithm will be `C A D B E`.

14.5 The adjacency lists will change as follows: The list for `C` will have one more entry of 11 with pointer to `B`, and the list for `B` will have one more entry of 10 with pointer to `F`. The list for `F` will be empty. The shortest paths will be as follows: A to `B` is 12 with prev as `A`, `A` to `C` is 76 with prev as `D`, `A` to F is 22 with prev as `B`, `A` to `E` is 23 with prev as B, and `A` to `D` is 66 with prev as `E`.

In Theory

14.6 Keep an extra stack and add to each adjacency matrix entry a member which we call `whereOnStack`. Suppose `a[i][j]` is inserted. Push the pair `i,j` onto the stack, and then have the `whereOnStack` member point at the top of the stack. When we access an adjacency matrix entry, we check that `whereOnStack` references a valid part of the stack and that the pair stored there corresponds to the matrix entry that we are trying to access.

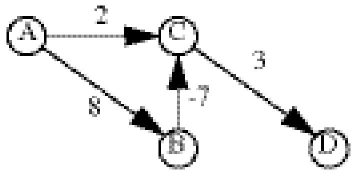
14.7 A queue will no longer work. We store the weighted cost of the path. Then, instead of testing if *w* is at distance infinity, also allow that *w* is at distance d_v+1 but with a total weight that would be smaller. Note that a queue no longer works; we need a priority queue again, and the weights must be non-negative. If there was an additional field called `weight`, this would be implemented as:

```
if( table[ w ].dist == Infinity ||
        table[ w ].dist == table[ v ].dist + 1 &&
        table[ w ].weight > table[ v ].weight + cvw )
{
    table[ w ].dist   = table[ v ].dist + 1;
    table[ w ].weight = table[ v ].weight + cvw;
    ...
}
```

14.8 Use an array `count` such that for any vertex `u`, `count[u]` is the number of distinct paths from `s` to `u` known so far. When a vertex `v` is marked as known, its adjacency list is traversed. Let `w` be a vertex on the adjacency list. If $D_v+c_{v,w}=D_w$, then increment `count[w]` by `count[v]` , because all shortest paths from `s` to `v` with last edge `(v,w)` give a shortest path to `w`. If $D_v+c_{v,w}<D_w$, then D_w and the previous vertex info get updated. All previously known shortest paths to `w` are now invalid, but all shortest paths to `v` now lead to shortest paths to `w`, so set `count[w]` to equal `count[v]` . Note: zero cost edges mess up this algorithm.

14.9 Use an array `numEdges` such that for any vertex `u`, `numEdges[u]` is the shortest number of edges on a path of distance D_u from `s` to `u` known so far. When a vertex `v` is marked as known, its adjacency list is traversed. Let `w` be a vertex on the adjacency list. If $D_v+c_{v,w}=D_w$, then change the previous vertex to `v` info an d se t `numEdges[w]` to `numEdges[v]+1` if `numEdges[v]+1<numEdges[w]` . If $D_v+c_{v,w}<D_w$, then `D` and the previous vertex info get updated and we set `numEdges[w]` to `numEdges[v]+1` .

14.10 In the graph below, Dijkstra's algorithm gives a shortest path from A to D of 5, even though the shortest path is 4.



14.11 This transformation adds more weight to paths that have many edges than it does to paths that have few edges; thus the resulting shortest paths do not necessarily correspond to the original.

14.12 We define a pass of the algorithm as follows: Pass 0 consists of marking the start vertex as known and placing its adjacent vertices on the queue. For $j > 0$, pass j consists of marking as known all vertices on the queue at the end of pass $j-1$. Each pass requires $O(|E|)$ time, since during a pass, a vertex is placed on the queue at most once. It is easy to show by induction that if there is a shortest path from s to v containing k edges, then d_v will equal the length of this path by the beginning of pass k (because the shortest path to its predecessor has length $k-1$, and has already been found, by the inductive hypothesis). Thus there are at most $|V|$ passes, giving an $O(|E| |V|)$time bound.

14.13 The phrasing in the text is a bit misleading—the question is still a single source problem. The longest path problem can be solved recursively: for each w adjacent to v compute the longest path that begins at w. Then the computation for v becomes simple. This works in acyclic graphs because the recursion is guaranteed to terminate (there are no cycles) and each edge is visited only once.

14.15 The maximum bottleneck problem can be solved by modifying the shortest path algorithm as follows: We define the cost of a path as the weight of the shortest edge in the path. In the algorithm of Figure 14.27, modify lines 35..38 as follows:

```
if( w.dist < min(v.dist,cvw) )
    w.dist = min(v.dist, cvw)
    w.prev = v;
```

14.16 (a) True. In an acyclic graph, there is either a path from $u$to v or from $v$to $u$or no path. If there is no path, then an edge can be added in either direction without creating a cycle. If there exists a path from *u*to *v*, then we can add an edge from *u*to *v*. Otherwise, an edge can be added from *v*to *u*. (b) True. If adding (*u*,*v*) creates a cycle, then there exists a path from *v*to *u*in *G*. If adding (*v*,*u*) creates a cycle, then there exists a path from *u*to *v*in *G*. Thus, there is a cycle in *G*.

Programming Practice

14.19 Each team is a vertex; if *X* has defeated *Y*, draw an edge from *X* to *Y*.

14.20 Each word is a vertex; if *X* and *Y* differ in one character, draw edges from *X* to *Y*and *Y* to *X*.

14.21 Each currency is a vertex; draw an edge of (–log *C*) between vertices to represent a currency exchange rate, *C*. A negative cycle represents an arbitrage play.

14.22 Each course is a vertex; if *X* is a prerequisite for *Y*, draw an edge from *X* to *Y*. Find the longest path in the acyclic graph.

14.23 Each actor is a vertex; an edge connects two vertices if the actors have a shared movie role. The graph is not drawn explicitly, but edges can be deduced as needed by examining cast lists.

CHAPTER 15 Inner Classes and Implementation of ArrayList

15.2 Solutions To Exercises

In Short

15.1 When an inner class object is constructed, it has an implicit reference to the outer class whereas this is not the case for a nested class. Inner class is like a non-static nested class.

15.2 Private members of an inner class are public to the outer class.

15.3 The declaration of `a` is legal and `b` is illegal because `Inner1` is non-static (inner) class whereas `Inner2` is static.

15.4 An object of type `Inner1` can be created using a

```
Inner1 in1 = new Inner1();
```

statement in any method of class `Outer`. An object of type `Inner2` can be created using a statement

```
Inner2 in2 = new Inner2(this);
```

A constructor needs to be added to class `Inner2`.

In Theory

15.6 To refer to `I`, we will need to specify `O.I`. However, this may be problem if `O` does not exist. Hence, we may need syntax such as `implements o.I`, where `o` is a class `O` object.

15.7 The running time of `clear` is $O(1)$. The running time of the inherited version would be $O(N)$, where N is the size of the list.

In Practice

15.8

```
public class MyContainer
{
    private Object [ ] items = new Object[ 5 ];
    private int size = 0;

    public Object get( int idx )
    {
        if( idx < 0 || idx >= size )
            throw new ArrayIndexOutOfBoundsException( );
        return items[ idx ];
    }

    public boolean add( Object x )
    {
        if( items.length == size )
        {
            Object [] old = items;
            items = new Object[ items.length * 2 + 1 ];
            for( int i = 0; i < size; i++ )
                items[ i ] = old[ i ];
        }

        items[ size++ ] = x;
        return true;
    }

    public Iterator iterator( )
    {
        return new LocalIterator( );
    }

    private class LocalIterator implements Iterator
    {
        private int current = 0;

        public boolean hasNext( )
        {
            return current < size;
        }

        public Object next( )
        {
            return items[ current++ ];
        }
        public boolean hasPrevious()
        {
            return current > 0;
        }

        public Object previous()
        {
            return items[current—];
        }
    }
}
```

15.9 This is an example of an adaptor pattern.

```
import java.util.*;
public class BetterIterator
{
    Iterator itr;

    public BetterIterator()
    {
        Iterator itr = new Iterator();
    }

    public boolean isValid()
    {
        return itr.hasNext();
    }

    public void advance()
    {
        itr.next();
    }

    public Object retrieve()
    {
        return itr.next();
    }
}
```

15.10 The first one will throw a `ConcurrentModificationExceptio(n`).

CHAPTER 16 Stacks and Queues

16.2 Solutions To Exercises

In Short

16.1 Figure 16.1 shows the states of the data structures. The first line illustrates the state of the array-based stack. Next is the array-based queue. The boldfaced item is the rear, while the italicized item is the front. The third picture shows the state of the stack when implemented as a list. The final picture shows the queue implemented as a list.

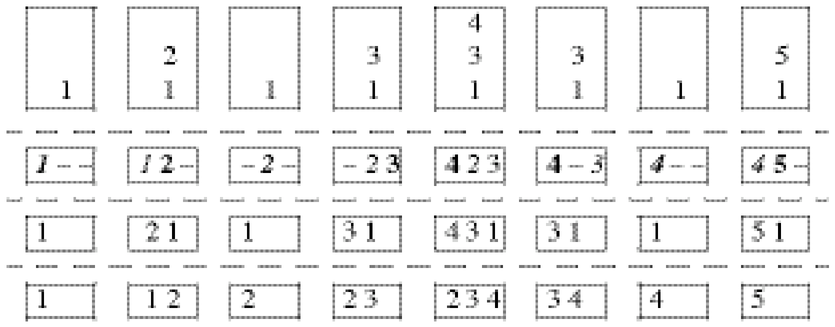


Figure 16.1 Stages of stacks and queues for Exercise 16.1

In Practice

16.4

```
public static void main( String [ ] args )
{
   Stack intStack = new StackAr( );
   Stack doubleStack = new StackAr( );

   intStack.push( new Integer( 5 ) );
   doubleStack.push( new Double( 7.0 ) );
   // etc...
}
```

16.5 For this inheritance to work, the previous private section of the queue class must be protected and we need to use the array implementation (because the node prior to the back is not easily accessible). Assuming that this is done,

```
public void addFront( Object x )
{
   if( currentSize == array.length )
      doubleQueue( );
   front = decrement( front );
   array[ front ] = x;
   currentSize++;
}

public void removeBack( ) throws Underflow
{
   if( isEmpty( ) )
      throw new Underflow( “Empty Deque” );
   currentSize—;
   decrement( back );
}

public Object getBack( ) throws Underflow
{
   if( isEmpty( ) )
      throw new Underflow( “Empty Deque” );
   return array[ back ];
}
```

16.6 `push(x)` is implemented as `add(x)`, `pop()` is implemented as `remove(size() - 1)` and `top()` as `get(size() -1)`. The `ArrayList` has an automatic array doubling feature.

16.7 `ArrayList` is not a good choice to implement to a queue because when the array size is doubled the, `ArrayList` implementation copies elements at the same location. In the case of a queue (see array based implementation of queue), in the new doubled array, the elements are copied so that the front of the queue is at location 0. This allows the new element to be added at the end of the array. The `ArrayList` implementation will require extra work in this case.

16.8 This is done in the Java library. Because of wraparound, after the array has been doubled, either the elements from the start of the array until position `rear` must be relocated or the elements from position `front` to the position that is representative of the end of the original array must be relocated to the end of the array. A simple calculation chooses the alternative that involves the fewer number of elements.

CHAPTER 17 *Linked Lists*

17.2 Solutions To Exercises

In Short

17.1 See Figure 17.5 on page 540 of the text.

17.2 See Figure 17.16 on page 550 of the text.

```
public void reverseList( )
{
    ListNode previousPosition;
    ListNode currentPosition;
    ListNode nextPosition;

    currentPosition = header.next;
    nextPosition = currentPosition.next;
    previousPosition = null;

    while( nextPosition != null )
    {
        currentPosition.next = previousPosition;
        previousPosition = currentPosition;
        currentPosition = nextPosition;
        nextPosition = nextPosition.next;
    }
    currentPosition.next = previousPosition;
    header.next = currentPosition;
}
```

In Theory

17.3 Reversal of a linked list in constant extra space can be done by maintaining 2 references: `previousPosition` and `currentPosition`. At any time, the list from the start to `previousPosition` is already reversed, while the rest of the list, from `currentPosition` to the end is normal. Each iteration of a loop extends the reversal by one position while maintaining these invariants. The code is shown above.

17.4 (a) Scan the list and mark each node as you visit it. If you reach an already visited node, there is a cycle. The marking will take $O(N)$ extra space. (b) Have two iterators `itr1` and `itr2`, where `itr1` moves twice as fast as `itr2`. In each iteration, `itr1` is moved twice and `itr1` is moved once. At each node visited by `itr1` (`itr2`), we check whether `itr2` (resp `itr1`) is pointing to the same node. If yes, then there exists a cycle.

17.5 Maintaining an iterator that corresponds to the last item will provide constant time access to the last item as well as the first item. Hence, we can perform queue operations in constant time.

17.6 Copy the value of the next node in the current node and then delete the next node.

17.7 (a) Add a new node after position `p` and copy the element in position `p` to the new node. Then, change data of position `p` to the new item. (b) Copy the next element to position `p` and delete the next node.

In Practice

17.9 `remove` is shown below.

```
public void remove( ) throws ItemNotFound
{
    boolean removedOK = false;
    for( ListNode p = theList.header; p.next != null; )
        if( !p.next.element.equals( x ) )
            p = p.next;
        else
        {
            removedOK = true;
            p.next = p.next.next;
        }

    if( removedOK )
        current = theList.header;
    else
        throw new ItemNotFound( “Remove fails” );
}
```

17.11 Since it is not part of the iterator class, it cannot access private iterator members.

17.14 To implement retreat, we save the original position and then traverse the list until we are right in front of the original position.

```
public void retreat( )
{
    ListNode original = current;
    current = theList.header;
    while( current.next != original )
        current = current.next;
}
```

CHAPTER 18 Trees

18.2 Solutions To Exercises

In Short

18.1 (a) The root is *A*; (b) Leaves are *G*, *H*, *I*, *L*, *M*, and *K*; (c) The depth is 4; (d) preorder: *A*, *B*, *D*, *G*, *H*, *E*, *I*, *J*, *L*, *M*, *C*, *F*, *K*; postorder: *G*, *H*, *D*, *I*, *L*, *M*, *J*, *E*, *B*, *K*, *F*, *C*, *A*; inorder: *G*, *D*, *H*, *B*, *I*, *E*, *L*, *J*, *M*, *A*, *C*, *F*, *K*; level order: *A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*, *I*, *J*, *K*, *L*, *M*.

18.2 See Figure 18.1.

18.3 abbdddbaceeecca (with appropriate newlines).

18.4 The stack states are shown in Figures 18.2 and 18.3.

In Theory

18.5 Proof is by induction. The theorem is trivially true for $H = 0$. Assume that it is true for $H = 0, 1, 2, ...,$

k ; we show that this implies it is true for $H = 1 + k$. This follows because a tree of height $k + 1$ has two subtrees whose height is at most k. By the induction hypothesis, these subtrees can have at most $2^{k+1} - 1$ nodes each. These $2^{k+2} - 2$ nodes plus the root prove the theorem for $k + 1$ and hence for all heights.

	Parent	Children	Siblings	Height	Depth	Size
A		B, C		4	0	13
B	A	D, E	C	3	1	9
C	A	F	B	2	1	3
D	B	G, H	E	1	2	3
E	B	I, J	D	2	2	5
F	C	K		1	2	2
G	D		H	0	3	1
H	D		G	0	3	1
I	E		J	0	3	1
J	E	L, M	I	1	3	3
K	F			0	3	1
L	J		M	0	4	1
M	J		L	0	4	1

Figure 18.1 Solution to Exercise 18.2

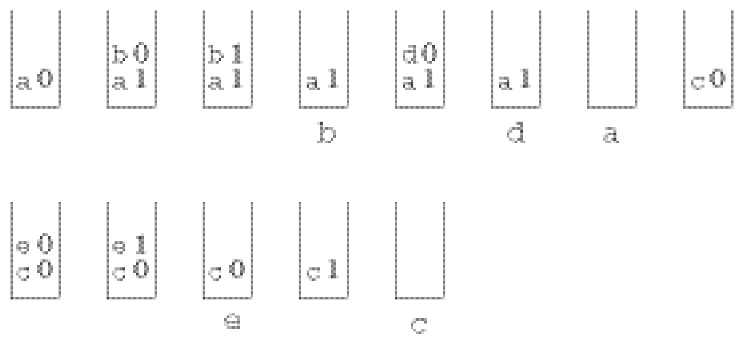


Figure 18.2 Stack states during inorder traversal

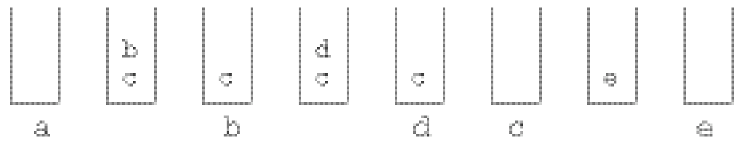


Figure 18.3 Stack states during preorder traversal

18.6 Let N = the number of nodes, F = the number of full nodes, L = the number of leaves, and H = the number of nodes with one child. Clearly, $N = F + L + H$. Further, the number of non-`null` references in the tree is exactly $N - 1$ since each such pointer connects a node to its parent and every node except the root has a parent. Thus $N - 1 = 2F + H$. Subtracting yields $1 = L - F$ so $F + 1 = L$.

18.7 As mentioned in the previous exercise, there are $N - 1$ non-`null` references in any tree. Since a binary tree has $2N$ references, this leaves $N + 1$ `null` references. An M-ary tree has MN references, so $(M - 1) N + 1$ of them are `null`.

18.8 This can be shown by induction. In a tree with no nodes, the sum is zero, and in a one node tree, the root is a leaf at depth zero, and so the claim is true. Suppose the theorem is true for all trees with at most k nodes. Consider any tree with $k + 1$ nodes. Such a tree consists of an i node left subtree and a $k - i$ node right subtree for some i. By the inductive hypothesis, the sum for the left subtree leaves is at most one with respect to the left subtree root. Because all leaves are one deeper with respect to the original tree root than with respect to the subtree, the sum is at most 1 / 2 with respect to the root. Similar logic implies that the sum for the leaves in the right subtree is also at most 1 / 2, proving the theorem. The equality is true if and only if there are n nodes with one child. If there is a node with one child, the equality cannot be true, because adding a second child would increase the sum to higher than 1. If no nodes have one child, then we can find and remove two sibling leaves, creating a new tree. It is easy to see that this new tree has the same sum as the old. Applying the step repeatedly, we arrive at a single node, whose sum is 1. Thus the original tree had sum 1.

In Practice

18.9

```
public static int numLeaves( BinaryNode t )
{
    if( t == null )
        return 0;
    int lf = ( t.left==null && t.right==null ) ? 1 : 0;
    return numLeaves(t.left) + numLeaves(t.right) + lf;
}

public static int numOne( BinaryNode t )
{
    if( t == null )
        return 0;
    int on = ( t.left==null != t.right==null ) ? 1 : 0;
    return numOne( t.left ) + numOne( t.right ) + on;
}

public static int numFull( BinaryNode t )
{
    if( t == null )
        return 0;
    int fl = ( t.left!=null && t.right!=null ) ? 1 : 0;
    return numFull( t.left ) + numFull( t.right ) + fl;
}
```

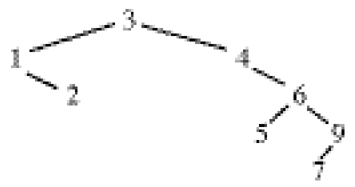
CHAPTER

19 Binary Search Trees

19.2 Solutions To Exercises

In Short

19.1 To conserve space, I list the trees by supplying a preorder traversal. Since an inorder traversal is given, this can be used to construct the tree. For this example, the resulting tree (shown below) would be 31246597; after the deletion it is 4126597 if the smallest node in the right subtree is used to replace the root.



19.2 In parentheses I list the number of possible insertion sequences. 1234 (1), 1243 (1), 1324 (2), 1423 (1), 1432 (1), 2143 (3), 2134 (3), 3214 (3), 3124 (3), 4321 (1), 4312 (1), 4213 (2), 4123 (1), 4132 (1).

19.3 Only one tree is possible: 213.

19.4 Only four trees are possible, and each are equally likely: 2134 (6), 2143 (6), 3124 (6), and 3214 (6).

19.5 For the AVL tree, 42136597. For the red-black tree, 21543769. The red nodes are 5, 3, 6, 9.

19.6 For three nodes, only one tree is possible: 213. However, in four instances nodes 1 and 3 are red, while in two instances they are both black. In all six cases, the root is black. The four node case is left to the reader.

In Theory

19.7 When there are zero nodes, the internal and external path lengths are equal. Observe that when a node is added to a binary search tree at depth d, the internal path length increases by d. The external path length increases by $2(d + 1)$ because of two new `null` references, but decreases by d (because a `null` reference is converted to the new node), for a new increase of $d + 2$. Thus each insertion into the tree increases $EPL(T) - IPL(T)$ by 2. After N inserts, $EPL(T) - IPL(T) = 2N$.

19.8 The tree that results is perfectly balanced (there is only one such tree for $N = 2^k - 1$ nodes). This can be proved by induction. It is easy to verify the claim for $1 \le k \le 3$. Suppose it is true for $k = 1, 2, 3, ..., h$. Then after the first $2^h - 1$ insertions, we know by the induction hypothesis that 2^{h-1} is at the root, and the right subtree is a balanced tree containing $2^{h-1} - 1$ through $2^h - 1$. Each of the next 2^{h-1} insertions, namely 2^h through $2^h + 2^{h-1} - 1$ insert a new maximum and get placed in the right subtree, eventually forming a perfectly balanced right subtree of height $h - 1$. This follows by the induction hypothesis because the right subtree may be viewed as being formed from the successive insertion of $2^{h-1} + 1$ through $2^h + 2^{h-1} - 1$. The next insertion forces an imbalance at the root, and creates a perfectly balanced left subtree of height $h - 1$. The new key is attached to the perfectly balanced right subtree of height $h - 2$ as the last node in the right path. Thus the right subtree is exactly as if the nodes $2^h + 1$ through $2^h + 2^{h-1}$ were inserted in order. By the inductive hypothesis, the subsequent successive insertions of $2^h + 2^{h-1} + 1$ through $2^{h+1} - 1$ will create a perfectly balanced right subtree of height $h - 1$. Thus after the last insertion, both the left and the right subtrees are perfectly balanced, and of the same height, so the entire tree of $2^{h+1} - 1$ nodes is perfectly balanced (and has height h).

19.9 A deletion algorithm is provided in Knuth, Volume III.

19.10 Let B equal the number of black nodes on the path to the bottom. It is easy to see by induction that there must be at least$2^B - 1$ black nodes in the tree. Let H be the height of the tree. Since consecutive red nodes are not allowed, $B \geq \lceil H / 2 \rceil$. Thus $2^{\lceil H/2 \rceil} - 1 \leq 2^B - 1 \leq N$. Consequently, $H \leq 2\log (N + 1)$. As an example of a bad case which we call $BAD (B)$, let $MIN (B)$ be the red–black tree of fewest nodes with B nodes on the path to the bottom. This is a perfect tree of $2^B - 1$. Construct a bad tree as follows: the root is red, the left child is a $MIN (B)$ tree and the right child is black. For this black child, its left child is a $MIN (B - 1)$ tree and its right child is a $BAD (B - 1)$ tree. A base case is a complete 7 node tree with black nodes on the middle level. Let $T(H)$ be the number of nodes in a bad tree of (odd) height h. $T(3) = 7$ and $T (H) = T (H - 2)$ and $2^{\lfloor H/2 \rfloor} + 2^{\lfloor H/2 \rfloor - 1}$. The solution of this recurrence is $T (H) = 3 \cdot 2^{\lceil H/2 \rceil} + 1$, so $H = 2 (\log(N - 1) - \log 3) + 1$.

19.11 Color a nonroot node red if its height is even and its parent's height is odd; otherwise color a nonroot node black. This coloring satisfies all the red-black properties. Not all red-black trees are AVL trees, since the deepest red-black tree is deeper than the maximum height proven for an AVL tree.

19.13 Since 8 bits can store heights up to 127 (unless tricks are used to reuse the negative numbers), and an AA-tree has the same height properties as a red–black tree, trees of roughly 2^{63} nodes can be stored.

In Practice

19.17 It avoids some additions in the searching routine.

19.18 The method below performs an inorder traversal but tests to make sure that a recursive call is not made in vain (implementation of a public driver is left to the reader). As a result, when these tests fail and a recursive call is not made, the effect of the routine is the same as if the portion of the subtree that would be visited did not exist. As a result, the nodes that are actually visited are the K nodes that are output plus the nodes on the path to the root of the range that is output. This additional path has average length of $O(\log N)$, giving an average running time of $O (K + \log N)$.

```
public void printInRange( BinaryNode t Comparable low, Comparable high )
{
    if( t != null )
    {
        if( t.element.compares( low ) >= 0 )
            printInRange( t.left, low, high );
        System.out.println( t.element );
        if( t.element.compares( high ) <= 0 )
            printInRange( t.right, low, high );
    }
}
```

19.19 The public driver is assumed already written.

```
BinaryNode buildTree( int low, int high )
{
    if( low > high )
        return null;

    int middle = ( low + high ) / 2;
    BinaryNode newNode = new BinaryNode( new MyInteger( middle ) );
    newNode.left = buildTree( low, middle - 1 );
    newNode.right = buildTree( middle + 1, high );
    newNode.size = high - low + 1;
    return newNode;
}
```

19.20

```
BinaryNode doubleRotateWithLeftChild( BinaryNode k3 )
{
   BinaryNode k1 = k3.left;
   BinaryNode k2 = k1.right;

   k1.right = k2.left;
   k3.left  = k2.right;
   k2.left  = k1;
   k2.right = k3;
   return k2;
}
```

Programming Projects

19.23 The basic problem is that the minimum element in the tree may be marked deleted, as may many of the next smallest elements. A recursive implementation of `findMin` is easiest.

```
BinaryNode findMin( BinaryNode t )
{
   if( t == null )
      return null;

   BinaryNode lt = findMin( t.left );
   if( lt != null )
      return lt;

   if( t.count != 0 ) // try current item
      return t;
   return findMin( t.right );
}
```

CHAPTER 20 Hash Tables

20.2 Solutions To Exercises

In Short

20.1 0 through 10.

20.2 23, because the load factor should be less than 0.5 and the table size should be prime.

20.3 Lazy deletion must be used: The items are marked deleted.

20.4 The cost of an unsuccessful search when the load factor is 0.25 is 25/18. The cost of a successful search is 7/6.

20.5 The hash tables are shown in Figure 20.1.

20.6 When rehashing, we choose a table size that is roughly twice as large, and prime. In our case, an appropriate new table size is 19, with hash function *hash*(*x*) = *x* mod 19.

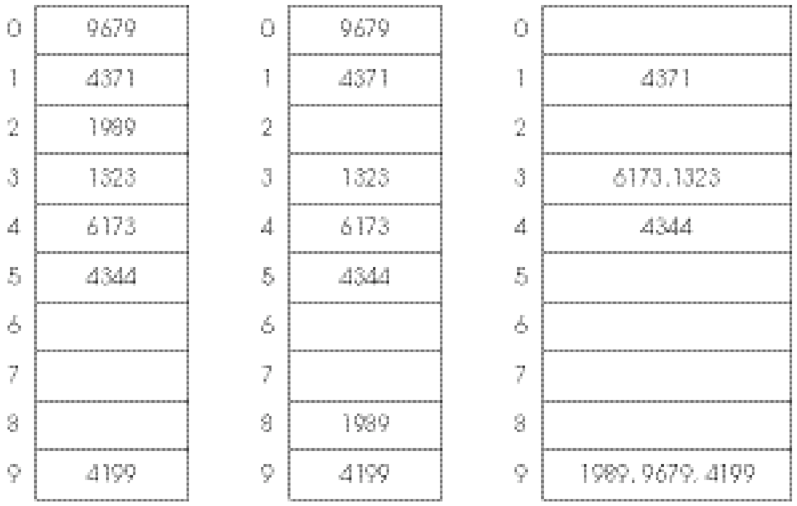


Figure 20.1 (a) linear probing; (b) quadratic probing; (c) separate chaining

a. The new locations are 9679 in bucket 8, 4371 in bucket 1, 1989 in bucket 13, 1323 in bucket 12, 6173 in bucket 17, 4344 in bucket 14 because both 12 and 13 are already occupied, and 4199 in bucket 0.
b. The new locations are 9679 in bucket 8, 4371 in bucket 1, 1989 in bucket 13, 1323 in bucket 12, 6173 in bucket 17, 4344 in bucket 16 because both 12 and 13 are already occupied, and 4199 in bucket 0.
c. The new locations are: 4371 in list 1, 6173 in list 17, 1323 in list 12, 4344 in list 12, 1989 in list 13, 9679 in list 8, and 4199 in list 0.

In Theory

20.8 It is true. However, that does not mean that the expected cost of the last insertion is obtained by plugging in 0.375 for the load factor. Instead, we must average the insertion cost over all load factors from 0.25 to 0.5 as

$$\frac{1}{0.50-0.25}\int_{0.25}^{0.50} I(\lambda)d\lambda .$$

20.9 As in the previous example, we want to compute the average cost of inserting an element into the new table. This is the same as the cost of an average successful search in the new table with load factor 0.25, namely 1.167. Thus the expected number of probes in the insertion sequence is 1.167*N*.

20.10 (a) The expected cost of an unsuccessful search is the same as an insertion;

(b) $$\begin{aligned} S(\lambda) &= \frac{1}{\lambda}\int_{x=0}^{\lambda} I(x)dx \\ &= \frac{1}{\lambda}\int_{x=0}^{\lambda}\left(\frac{1}{1-x} - x - \ln(1-x)\right)dx \\ &= \frac{1}{\lambda}\left(-\ln(1-x) - \frac{x^2}{2} + ((1-x)\ln(1-x) - (1-x))\right)\Bigg|_{x=0}^{\lambda} \\ &= \frac{1}{\lambda}\left(-\frac{x^2}{2} - x\ln(1-x) + x - 1\right)\Bigg|_{x=0}^{\lambda} \\ &= (1 - \ln(1-\lambda) - \lambda/2) \end{aligned}$$

20.11 (a) The hash table size is a prime near 25,013. (b) The memory usage is the memory for 10,000 `String` objects (at 8 bytes plus additional storage to maintain members of the `String` class (including the length, etc., and depends on the specific implementation)). (c) The memory usage for the hash table is one reference and a `boolean` per array item. Likely, this is 8 bytes for each array item, or approximately 200,000 bytes. (d) Add all this up for the total memory requirements. (e) The space overhead is the number in part (d) minus the number in part (b) and is probably considerable.

In Practice

20.12 Use the following version of `findPos` :

```
private int findPos( Object x )
{
    int currentPos = ( x == null ) ?
            0 : Math.abs( x.hashCode( ) % array.length );
    while( array[ currentPos ] != null )
    {
        if( x == null )
        {
            if( array[ currentPos ].element == null )
                break;
        }
        else if( x.equals( array[currentPos].element ) )
            break;
        if( ++currentPos == array.length )
            currentPos = 0;
    }
    return currentPos;
}
```

CHAPTER

21 A Priority Queue: The Binary Heap

21.2 Solutions To Exercises

In Short

21.1 The structure property is that a heap is a complete binary tree — all nodes are present as we go from top to bottom, left to right. The ordering property is that the value stored in a node's parent is no larger than the value stored in a node.

21.2 The parent is in location $\lfloor i / 2 \rfloor$, the left child is in location $2i$, and the right child is in location $2i + 1$.

21.3 The resulting heaps are shown in Figure 21.1.

21.4 In the `percolateDown(3)` step, the percolation could have gone one level deeper.

21.5 Reverse the heap-order property, change the direction of the element comparisons, and place infinity as the sentinel.

21.6 See Figure 21.2.

21.7 Heapsort is not stable.

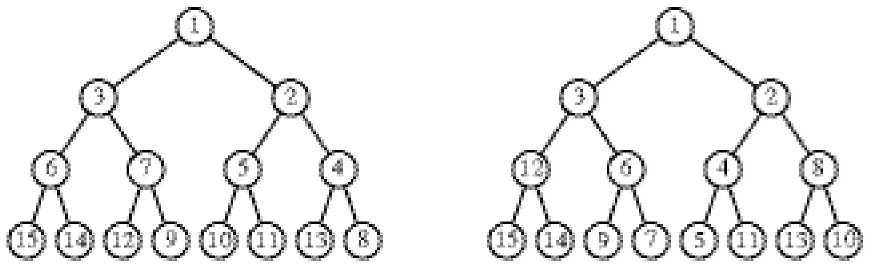


Figure 21.1 Sequential insertion (left); after `buildHeap` (right) Solutions To Exercises

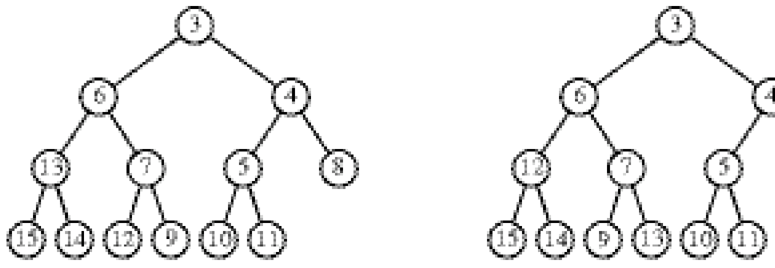


Figure 21.2 Exercise 21.6 after two `deleteMin` s: Starting from sequential insertion (left); starting from `buildHeap` (right)

In Theory

21.8 (a) 4 times as large; (b) The array must then have size $O(N^2)$; (c) $O(N^{4.1})$; (d) $O(2N)$.

21.9 (a) Any nonleaf node is a parent of a leaf and cannot be larger than the leaf. (b) Follows from earlier results on binary trees. (c) Let all leaf nodes except the unexamined leaf have positive value $3x$, and let all the leaf parents have the value x. If the unexamined leaf is not declared the maximum, the answer is wrong if it has value $4x$. If the unexamined leaf is declared the maximum, the answer is wrong if it has value $2x$. Thus no matter what answer is output, there is an input for which the algorithm will fail.

21.10 (Note that this question applies for complete trees). The summation is solved by letting S be the sum $\sum_{i=0}^{H} 2^i(H-i)$, letting $2S$ be twice the sum, and subtracting. As a result, many terms cancel and what is left is a geometric sum that is evaluated by a standard formula.

21.11

21.12 Observe that for $N = 0$ and $N = 1$, the claim is true. Assume that it is true for values of k up to and including $N-1$. Suppose the left and right sub-trees have L and R nodes respectively. Since the root has height $\lfloor \log N \rfloor$, we have

$$\begin{aligned} H(N) &= \lfloor \log N \rfloor + H(L) + H(R) \\ &= \lfloor \log N \rfloor + L - v(L) + R - v(R) \\ &= N - 1 + (\lfloor \log N \rfloor - v(L) - v(R)) \end{aligned}$$

The second line follows from the inductive hypothesis, the third follows because $L + R = N - 1$. Now the last node in the tree is either in the left sub + tree or the right subtree. If it is in the left subtree, then the right subtree is a perfect tree, and $v (R) = \lfloor \log N \rfloor - 1$. Further, the binary representation of N and L are identical with the exception that the leading 10 in N becomes 1 in L. (For instance if N = 37 = 100101, L = 10101.) It is clear that the second digit of N must be zero if the last node is in the left subtree. Thus in this case, $v (L) = v (N)$, and $H (N) = N - v (N)$.

If the last node is in the right subtree, then $v (L) = \lfloor \log N \rfloor$. The binary representation of R is identical to N except that the leading 1 is not present. (For in stance if N = 27 = 101011, L = 01011). Thus in this case, $v (L) = v (N) - 1$, and again, $H (N) = N - v (N)$

21.13 The leading term is $2N \log N$ because the number of comparisons in a `deleteMin` is $2\log N$.

21.14 See the paper in the references by Schaffer and Sedgewick.

21.15 The left child is at $r + 2(i - r) + 1$, right child is at $r + 2(i - r) + 2$ and the parent is at $r + (i - r) / 2 - 1$ if i is even and $r + (i - r - 1) / 2$ if i is odd.

21.17 (a) Create a new node with the root of the combined heap. Remove the rightmost element of the `rhs` and put at the root. Then, percolate the root to its correct position. (b) Merge the left subtree of `lhs` with `rhs` as described in (a). Then, percolate the root to its correct position. (c) If $l > r$, follow the left links $l - r$ times from the root to reach a subtree T of size $2r - 1$. Merge – this subtree with `rhs`. Then percolate each of the nodes in the path from the root to the root of the merged subtree.

21.18 `insert` takes $O (\log_d N)$.`deleteMin` takes $O (d \log_d N)$.

In Practice

21.23

```
private static void percDown( Comparable [ ] a, int i, int N )
{
    int child;
    Comparable tmp = a[ i ];

    for( ; i * 2 + 1 <= n; i = child )
    {
        child = i * 2 + 1;
        if( child != n && a[ child ].lessThan( a[ child + 1 ] ) )
            child++;
        if( tmp.lessThan( a[ child ] ) )
            a[ i ] = a[ child ];
        else
            break;
    }
    a[ i ] = tmp;
}
```

CHAPTER 22 Splay Trees

22.2 Solutions To Exercises

In Short

22.1 For the bottom-up splay, the preorder traversal yields 8, 6, 5, 2, 1, 4, 3, 9 (as in Chapter 18, this defines a unique binary search tree). For the top-down splay, the result is 8, 6, 4, 2, 1, 3, 5, 9.

22.2 For the bottom-up splay tree, deletion of the 3 requires that it be splayed to the root and deleted. The left subtree's maximum is splayed to the root (simple in this case) and then the right subtree is attached to the left subtree's new root as its right subtree. The result is 2, 1, 8, 5, 4, 6, 9. For the top-down splay tree, the result is 2, 1, 6, 4, 5, 8, 9.

In Theory

22.3 This result is shown in the Sleator and Tarjan paper.

22.4 This is easily shown by induction.

22.5 No. Consider a worst-case chain of right children. If the splaying access is on the root, and the non-splaying access on the deep node, then the cost of the two accesses is $O(N)$, and this can be repeated forever.

22.6 (a) 523776; (b) 262166, 133114, 68216.

22.7 This result is shown in the Sleator and Tarjan paper.

In Practice

22.8 `deleteMin` is implemented by searching for negative infinity (bringing the minimum to the root) and then deleting the item at the root. Note that `findMin` is not a constant member function.

22.9 Add a field to store the size of a tree. Note that the size changes during a rotation and must be maintained there in addition to during the insertion and deletion.

CHAPTER

23 Merging Priority Queues

23.2 Solutions To Exercises

In Short

23.1 The resulting skew heaps are shown in Figure 23.1.

23.2 (a) the root is 1 and all other nodes are children in the following order: 7, 6, 5, 4, 3, 2. (b) The root is 1 and all other nodes are children in the following order: 2, 7, 6, 3, 5, 4.

23.3 See Figure 23.2 and Figure 23.3.

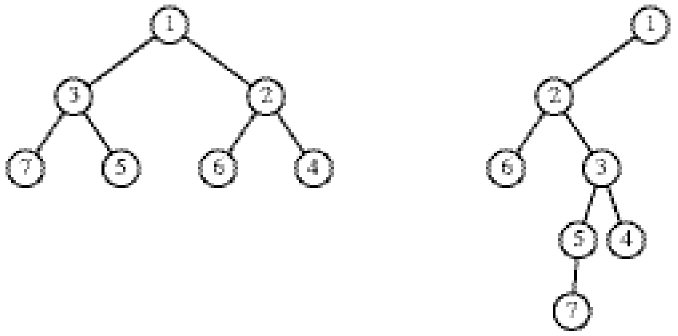


Figure 23.1 After skew heap insertions 1, 2, 3, 4, 5, 6, 7 (left); after skew heap insertions 4, 3, 5, 2, 6, 7, 1 (right)

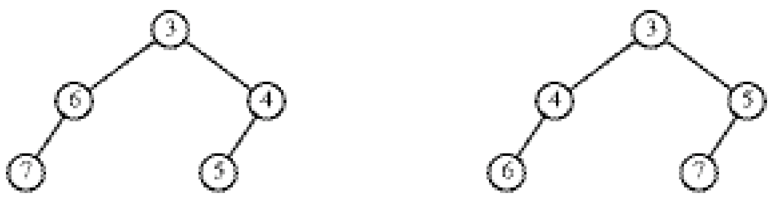
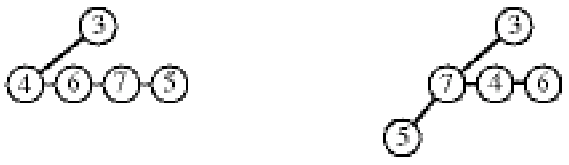


Figure 23.2 Skew heaps in Figure 23.1 after two `deleteMin` s each



CHAPTER

24 The Disjoint Set Class

24.2 Solutions To Exercises

In Short

24.1 We assume that parameters to the `union` s are roots. Also, in case of ties, the second tree is made a child of the first. Arbitrary union and union-by-height give the same answer, shown as the top tree in Figure 24.1. Union-by-size gives the bottom tree.

24.2 In both cases, change the parent of nodes 16 and 17 to the tree root.

24.3 Edges of cost 1, 2, 5, 6, 7, and 9 comprise the minimum spanning tree. The total cost is 30.

In Theory

24.5 Initially, each room is a node in a graph. Each connected component of the graph represents a set. Knocking a wall is equivalent to merging two connected components by add an edge. In this algorithm, a wall is knocked down only if it merges two connected component. Hence, it can be seen that the corresponding graph will be acyclic. Therefore, there is a unique path from start to ending point.

24.6 Modify the algorithm in Section 24.2.1 so that the component containing start and ending rooms are not merged. Thus, in the end, only two components remain, one containing the starting point and the other containing the ending point. One can then find a wall that can be knocked down to create a unique path

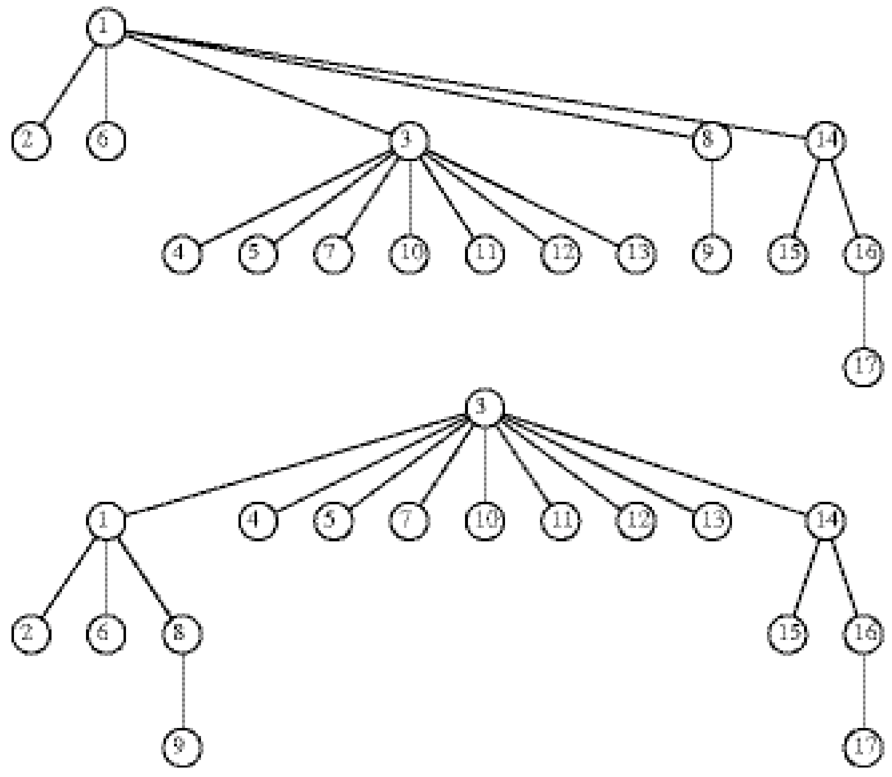


Figure 24.1 Result of unions for Exercise 24.1

24.7 The proof does not require the use of nonnegative edges. Here is an informal proof: if *T* is a spanning tree, then the addition of any edge *e* creates a cycle. Removal of any edge on the cycle reinstates the spanning tree property. The cost of the spanning tree is lowered if *e* has lower cost than the edge that was removed. Let *K* be the tree generated by Kruskal's algorithm. Suppose it is incorrect to use an edge that is accepted by Kruskal's algorithm. Let *T* be the spanning tree created by another algorithm and assume that it has lower cost. Add the smallest cost edge suggested by Kruskal's algorithm that is not found in *T*. This creates a cycle. Break the cycle by removing an edge that Kruskal's algorithm did not suggest (some edge must exist); if this edge has higher cost, then the replacement lowers the cost of *T*, contradicting its optimality. If it is equal, then continue with the replacement process. Eventually, either the contradiction will be reached, or we will see that *T* has the same cost as *K*, showing the Kruskal's algorithm is correct.

24.8 This is easily proved by induction because the depth can increase only when a tree is merged with a tree of equal height. So if we look at the smallest tree of height *H*, it is the result of merging the two smallest trees of height $H - 1$. Thus a tree of height *H* can be shown to have at least 2^H nodes.

24.9 Suppose there are *u* `union` and *f* `find` operations. Each `union` costs constant time, for a total of *u*. A `find` costs one unit per vertex visited. Charge under rule (A) if the vertex is a root or child of the root and under rule (B) otherwise. In other words, all vertices are in one rank group, so the total (A) charges are $2f$. Notice that a vertex can be charged at most once under rule (B), so the total (B) charges are at most *u*. All the charges thus add up to $2f + 2u$, which is linear.

24.10 We assume that the tree is implemented with node references instead of a simple array. Thus `find` will return a reference instead of an actual set name. We will keep an array to map set numbers to their tree nodes. `union` an d `find` are implemented in the standard manner. To perform `remove(x)` , first perform a `find(x)` with path compression. Then mark the node containing `x` as vacant. Create a new one node tree with `x` and have it referenced by the appropriate array entry. The time to perform a `remove` is the same as the time to perform a `find`, except that there could potentially be a large number of vacant nodes. To take care of this, after *N* `remove` s are performed, perform a `find` on every node, with path compression. If a `find(x)` returns a vacant root, then place `x` in the root node, and make the old node containing `x` vacant. The results of Exercise 24.9 guarantee that this will take linear time, which can be charged to the N `remove` s. At this point, all vacant nodes (indeed all nonroot nodes) are children of the root, and vacant nodes can be disposed. This all guarantees that there are never more than $2N$ nodes in the forest and preserves the time bound.

24.11 For each vertex v let the pseudorank R_v be defined as $\lfloor \log S_v \rfloor$, where S_v is the number of descendents (including itself) of v in the final tree, after all `union` s are performed, ignoring path compression. Although the pseudorank is not maintained by the algorithm, it is not hard to show that the pseudo-rank satisfies the same properties as the ranks do in union-by-rank. Clearly a vertex with pseudo-rank R_v has at least 2^{R_v} descendents (by its definition), and the number of vertices of pseudo-rank R is at most $N / 2^R$. The union-by-size rule ensures that the parent of a node has twice as many descendents as the node, so the pseudo-ranks increase monotonically on the path towards the root if there is no path compression. Path compression does not destroy this property.

24.12 This is most conveniently implemented without recursion, and is faster because even if full path compression is implemented non-recursively, it requires two passes up the tree. Path halving requires only one. We leave the coding details to the reader. The worst-case running time remains unchanged because the properties of the ranks are unchanged. Instead of charging one unit to each vertex on the path to the root, we can charge two units to alternating vertices (namely those vertices whose parents are altered by path halving). These vertices get parents of higher rank, as before, and the same kind of analysis bounds the total charges.

In Practice

24.14 (a) If path compression is implemented, the strategy does not work because path compression moves elements out of subtrees. For instance, the sequence `union(1,2)` ,`union(3,4)` ,