CSC416. Class Notes (DRAFT)

Massimo Di Pierro

School of Computer Science, Telecommunications and Information Systems

DePaul University, 243 S. Wabash Av, Chicago, IL 60604, USA

September 2, 2003

Abstract

These notes provide a coincise review to some of the material covered in the csc416 course. These notes are not intended as a substitution for the textbook. Attention: these notes are in a draft stage and may contain errors. Please report errors to mdipierro@cs.depaul.edu

Contents

1 Syllabus

1.1 Approximate weekly schedule

1. Java and Math review. Algorithms Analysis. (book chapters 1 and 2)
2. Arrays, Lists, Stacks and Queues (book chapters 3)
3. Sorting (book chapter 7)
4. Hashing (book chapter 5)
5. Prority Queues and Trees (book chapter 6 and 4)
6. Trees, AVL Trees and Red-Black Trees (book chapter 4)
7. Graphs (book chapter 9)
8. Graph Algorithms (book chapters 8 and 9)
9. Finite Automata and Formal Grammars
10. Finite Automata and Formal Grammars

1.2 Grading Policy

$$FG = 0.50 \cdot HA + 0.25 \cdot ME + 0.25 \cdot FE$$

where FG is the final grade, HA is the average of the homework assignments excluding the lowest grade, ME is the midterm exam, FE is the final exam. The grade is converted to a letter according to the following table:

92-100	A
89-91	A-
86-88	B+
82-85	B
79-81	B-
76-78	C+
73-75	C
70-72	C-
67-69	D+
60-66	D
0-59	F

2 Java review

2.1 Primitive types

```
boolean
char
byte
short
int
long
float
double
```

Note that `boolean` is either `true` or `false`, not equivalent to int as in C++

2.2 Packages

2.2.1 Required packages

```
import java.io.*;     // for input/output
import java.util.*;   // various stuff
import java.awt.*;    // for graphics
import java.applet.*; // for applets
```

2.3 Input/Output

2.3.1 To read from the console

```
BufferedReader input=new BufferedReader(
                     new InputStreamReader(System.in));
String  s=input.readLine(); // returns null is no input
Integer s=Integer.valueOf(input.readLine());
Double  s=Double.valueOf(input.readLine());
int     s=Integer.valueOf(input.readLine()).intValue();
double  s=Double.valueOf(input.readLine()).doubleValue();
```

2.3.2 To write to the console

```
System.out.print(""+whatever);   // do not go to newline
System.out.println(""+whatever); // do to newline
```

2.3.3 To read from a file

```
BufferedReader input=new BufferedReader(
                     new FileReader("filename"));
String  s=input.readLine(); // s==null at EOF
```

2.3.4 To write to a file

```
PrintWrter output=new PrintWriter(
                  new BufferedWriter(
                  new FileWriter("filename")));
output.print(""+whatever);
output.println(""+whatever);
```

2.3.5 How to break a string into tokens

```
s="1.34, 2.45";
StringTokenizer st=new StringTokenizer(s);
for(i=0; i<st.countTokens(); i++)
   x=Double.parseDouble(st.nextToken());
```

2.4 int, float, double vs Integer, Float, Double

`Integer` is an object wrapper of the primitive `int` type. `Float` is an object wrapper to the primitive `float` type. `Double` is an object wrapper to the primitive `Double` type.

By using objects instead of primitive types we can:

2.4.1 Store an int into an Integer

```
int i=5;
Integer j=new Integer(i);
```

2.4.2 Store an Integer into an int

```
Ingeter j=new integer(45);
int i=j.intValue();
```

2.4.3 What is the diffenerece then?

An Integer is a class therefore it has methods. For example:

```
String s=j.toString();
```

(convert to String)

```
int i=j.intValue();
```

(convert to int)

```
float a=j.floatValue();
```

(convert to float)

```
double x=j.doubleValue();
```

(convert to double)

```
j.parseInt("34");
```

(read from a String)

```
i=j.compareTo(new Integer(45));
```

(compare with another Integer)

```
int h=j.hashCode();
```

(hash it!)

What is more imporant we can cast an `Integer` into an `Object` and vice versa:

```
Object obj=new Integer(45);
Integer j=(Integer) obj;
```

Since the same is true for any object we can, for example, build an array of `Object`(s) and store `Integer`(s), `Float`(s), `Double`(s) and `String`(s) into it:

```
Object array[4];
array[0]=new Integer(3):
array[1]=new Float(3.14);
array[2]=new Double(3.14);
array[3]=new String("Hello World");
System.out.println("array[0]="+(Integer) array[0]);
System.out.println("array[1]="+(Float) array[1]);
System.out.println("array[2]="+(Double) array[2]);
System.out.println("array[3]="+(String) array[3]);
```

We will use this trick to build generic containers.

2.4.4 Comparable interface

Note that all numeric classes (`Integer`, `Float` and `Double`) and the `String` class implement the Comparable iterface. This means they have a method (`compareTo`) that can be used to compare an object with another object. For example we can define our own integer class:

```
public class MyInteger implements Comparable {
   private int value;
   public MyInteger(int x) { value=x; }
   public int intValue() { return value; }
   public int compareTo(Objects other) {
      if(value<((MyInteger) other).value) return -1;
      else if(value>((MyInteger) other).value) return +1;
      else return 0;
   }
}
```

So that we can do the following (as with Integer):

```
MyInteger x=new MyInteger(3);
MyInteger y=new MyInteger(5);
boolean comp=x.compareTo(y);
```

(`comp` is -1 in this case because x<y).

We will often assume our objects implement comparable.

2.5 More on Objects

In Java any variable is either a primitive type or a reference type (a pointer a a memory location where an object is stored. Consider the following example:

```
Integer a=new Integer(3);
Integer b=new Integer(3);
if(a==b) System.out.println("yes");
else System.out.println("no");
```

This program prints “no" since, even if a stores 3 and b stores 3, a and b refer to different instances of Integer(3), meaning they refer to different copies of the object stored in different memory locations. The condition a==b compares the memory locations of the references, not the values stored in the objects.

The following programs would, instead, print “yes".

```
Integer a=new Integer(3);
Integer b=a;
if(a==b) System.out.println("yes");
else System.out.println("no");

int a=3;
int b=3;
if(a==b) System.out.println("yes");
else System.out.println("no");
```

2.6 Exceptions

2.6.1 try ... catch ...

Many built-in methods may throw exceptions in case of failure. We should enclose these methods in try ... catch ... . For example:

```
Integer i;
String s="Hello World";
try {
   i=Integer.parseInt(s);
} catch(Exception e) {
   System.out.println("s does not contain an integer");
}
```

Our methods should also throw exception when there is a possibility of failure of the method itself. We throw the exception to inform the caller of the method that the method is unable to complete its task.

Note the following program:

```
public class MyException extends Exception {};
public class Test01 {
   public static void f(String[] args) {
      try {
         if(args.length==0) throw new MyException();
         return 1;
      } catch(Excetion e) {
         System.out.println("error in f: "+e);
         return 0;
      }
   }
   public static main(String[] args) {
      try {
         int i=f(args);
      } catch(Excetion e) {
         System.out.println("error in main: "+e);
      }
   }
}
```

Note: exception is caugth in `f`, not in `main`; `args.lentgh` retruns `int` and not `Integer`. It possible to throw Throwable rather than Exception but the former cannot be caught and will caouse the program to abort.

2.7 Conditionals

2.7.1 if ... else

```
boolean condition=true;
if(condition) {
    System.out.println("do this");
} else {
    System.out.println("do that");
}
```

Condition has to be boolean type, not int, Integer or else. Conditions can be combined uwing logical operators:

```
cond1 && cond2 (AND)
cond1 || cond2 (OR)
!cond1         (NOT)
```

2.7.2 swicth { case ... break ... otherwise }

```
int i=2;
switch(i) {
  case 0: System.out.println("i=0); break;
  case 1: System.out.println("i=1); break;
  case 2: System.out.println("i=2); break;
  default: System.out.println("i>2);
}
```

Note that switch works well with `int` and `String` but not with other types.

2.8 Loops

2.8.1 for(...)

```
int i;
for(i=0; i<100; i++) {
   System.out.println("i="+i);
}
```

In principle we could loop with Integer instead of int but it would be ugly:

```
Integer j;
for(j=new Integer(0);
    j.compareTo(new Integer(100))<0;
    j=new Integer(j.intValue()+1)) {
   System.out.println("i="+i);
}
```

In fact if j is Integer we cannot do j=j+1 or j++. This is why we do not do math with objects but only with primitive types. Java does not support operator overloading and this is one of its limitations.

2.8.2 while(...)

```
boolead cond=true;
while(cond) {
   System.out.println("cond="+cond);
   cond=false;
}
```

3 Math review

3.1 Symbols

∞	infinity
$\wedge$	and
$\vee$	or
$\cap$	intersection
$\cup$	union
$\in$	element or In
$\forall$	for each
$\exists$	exists
$\Rightarrow$	implies
$:$	such that
iff	if and only if

(1)

3.2 Set Theory

3.2.1 Important Sets

0	empty set
$\mathbb{N}$	natural numbers {0,1,2,3,...}
$\mathbb{N}^+$	positive natural numbers {1,2,3,...}
$\mathbb{Z}$	all integers {...,-3,-2,-1,0,1,2,3,...}
$\mathbb{R}$	all real numbers
$\mathbb{R}^+$	positive real numbers (not including 0)
$\mathbb{R}^*$	positive numbers including 0

(2)

3.2.2 Set operations

$\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ are some generic sets.

- **Intersection**
$$\mathcal{A} \cap \mathcal{B} \overset{def}{=} \{x : x \in \mathcal{A} \text{ and } x \in \mathcal{B}\} \tag{3}$$
- **Union**
$$\mathcal{A} \cup \mathcal{B} \overset{def}{=} \{x : x \in \mathcal{A} \text{ or } x \in \mathcal{B}\} \tag{4}$$
- **Difference**
$$\mathcal{A} - \mathcal{B} \overset{def}{=} \{x : x \in \mathcal{A} \text{ and } x \notin \mathcal{B}\} \tag{5}$$

3.3 Finite sums

3.3.1 Definition

$$\sum_{i=0}^{i \le n} f(i) \overset{def}{=} f(0) + f(1) + ... + f(n-1) + f(n) \tag{6}$$

Corresponding Python functions:

```
public Interface Summable {
   public static double f(int i);
}
public class MyFunction implements Summable {
   public static double f(int i) {
      return i*i; // example
   }
}
public class Adder {
   public static double sum(Summable a, int imin, int imax) {
      int i;
      for(i=imin; i<=imax; i++) {
         sum=sum+a.f(i);
      }
      return sum;
   }
   public static test() {
      double y=sum(new MyFunction(),0,10);
      System.out.println("sum="+y);
   }
}
```

3.3.2 Properties

- **Linearity**

$$\sum_{i=0}^{i\leq n} af(i)+bg(i) = a\left(\sum_{i=0}^{i\leq n} f(i)\right)+b\left(\sum_{i=0}^{i\leq n} g(i)\right) \tag{7}$$

Proof:

$$\begin{aligned}\sum_{i=0}^{i\leq n} af(i)+bg(i) &= (af(0)+bg(0))+(af(1)+bg(1))+...+(af(n)+bg(n)) \\ &= af(0)+af(1)+...+af(n)+bg(0)+bg(1)+...bg(n) \\ &= a\,(f(0)+f(1)+...+f(n))+b\,(g(0)+g(1)+...+g(n)) \\ &= a\left(\sum_{i=0}^{i\leq n} f(i)\right)+b\left(\sum_{i=0}^{i\leq n} g(i)\right) \end{aligned} \tag{8}$$

Examples:

$$\sum_{i=0}^{i\leq n} i = \frac{1}{2}n(n+1)$$

$$\sum_{i=0}^{i\le n} i^2 = \frac{1}{6}n(n+1)(2n+1)$$
$$\sum_{i=0}^{i\le n} i^3 = \frac{1}{4}n^2(n+1)^2$$
$$\sum_{i=0}^{i\le n} x^i = \frac{x^{n+1}-1}{x-1} \text{ (geometric sum)}$$

3.4 Order of growth of functions

Let $f(x)$ be the running time of an algorithm P1 as function of the input size x. Let $g(x)$ be the running time of another algorithm P2 as function of the input size x. The limit

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0 \tag{9}$$

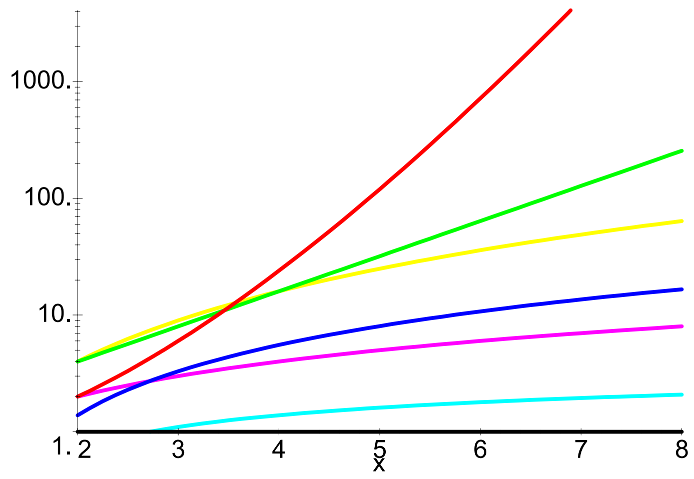
means that $g(x)$ grows more than $f(x)$ when $x \to \infty$. This induces a relation between $f(x)$ and $g(x)$:

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0 \Longrightarrow f(x) \text{ “grows less than" } g(x) \tag{10}$$

We can then sort functions according with their behavior at infinity for example:

$$1 \prec \log x \prec x \prec x\log x \prec x^2 \prec 2^x \prec x! \prec x^x \tag{11}$$

where the symbol $\prec$ in this context reads “grow less than".



(12)

3.4.1 Definitions

$$
\begin{aligned}
&O(g(x)) \overset{def}{=} \{f(x) : \exists x_0, c_0,\ \forall x > x_0,\ 0 \leq f(x) < c_0 g(x)\}\\
&\Omega(g(x)) \overset{def}{=} \{f(x) : \exists x_0, c_0,\ \forall x > x_0,\ 0 \leq c_0 g(x) < f(x)\}\\
&\Theta(g(x)) \overset{def}{=} O(g(x)) \cap \Omega(g(x))\\
&o(g(x)) \overset{def}{=} O(g(x)) - \Omega(g(x))\\
&\omega(g(x)) \overset{def}{=} \Omega(g(x)) - O(g(x))
\end{aligned}
$$

3.4.2 Intuitive meaning

- $O(g(x)) =$ Functions that grow no faster than $g(x)$ when $x \to \infty$
- $\Omega(g(x)) =$ Functions that grow no slower than $g(x)$ when $x \to \infty$
- $\Theta(g(x)) =$ Functions that grow at the same rate as $g(x)$ when $x \to \infty$
- $o(g(x)) =$ Functions that grow slower than $g(x)$ when $x \to \infty$
- $\omega(g(x)) =$ Functions that grow faster than $g(x)$ when $x \to \infty$

3.4.3 Practical rules

1. Compute the limit

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = a \tag{13}$$

2. Then look it up on the table

$$
\begin{array}{lcl}
a \text{ is positive or zero} & \Longrightarrow & f(x) \in O(g(x)) \Leftrightarrow f \preceq g\\
a \text{ is positive or infinity} & \Longrightarrow & f(x) \in \Omega(g(x)) \Leftrightarrow f \succeq g\\
a \text{ is positive} & \Longrightarrow & f(x) \in \Theta(g(x)) \Leftrightarrow f \sim g\\
a \text{ is zero} & \Longrightarrow & f(x) \in o(g(x)) \Leftrightarrow f \prec g\\
a \text{ is infinity} & \Longrightarrow & f(x) \in \omega(g(x)) \Leftrightarrow f \succeq g
\end{array} \tag{14}
$$

The rules assume the limits exist. The inverse is not true. For example:

$$f(x) \in \Theta(g(x)) \not\Rightarrow \lim_{x\to\infty} \frac{f(x)}{g(x)} \text{ is positive} \tag{15}$$

Theorem 1 *Any polynomial $P_m(n)$ of degree m has $T(n) \in \Theta(n^m), \in O(n^m)$*

Theorem 2 $T_1(n) \in O(f(n))$ *and* $T_2 \in O(g(n)) \Rightarrow T_1(n){+}T_2(n) \in O(\max(f(n), g(n)))$

Theorem 3 $T_1(n) \in O(f(n))$ *and* $T_2 \in O(g(n)) \Rightarrow T_1(n)T_2(n) \in O(f(n)g(n))$

3.4.4 Example: loop0

```
public static void loop0(int n) {
   int i;
   for(i=0; i<n; i++) {
      System.out.println("i="+i);
   }
}
```

$$T(n) = \sum_{i=0}^{i\leq n} 1 = n \in \Theta(n) \Rightarrow \texttt{loop0} \in \Theta(n)$$

3.4.5 Example: loop1

```
public static void loop1(int n) {
   int i;
   for(i=0; i<n*n; i++) {
      System.out.println("i="+i);
   }
}
```

$$T(n) = \sum_{i=0}^{i\leq n^2} 1 = n^2 \in \Theta(n) \Rightarrow \texttt{loop1} \in \Theta(n^2)$$

3.4.6 Example: loop2

```
public static void loop2(int n) {
   int i,j;
   for(i=0; i<n; i++) {
      System.out.println("i="+i);
      for(j=0; j<n; j++) {
         System.out.println("j="+j);
      }
   }
}
```

$$T(n) = \sum_{i=0}^{i\leq n}\sum_{j=0}^{j\leq n} 1 = \sum_{i=0}^{i\leq n} n = n^2 \in \Theta(n^2) \Rightarrow \texttt{loop2} \in \Theta(n^2)$$

3.4.7 Example: loop3

```
public static void loop3(int n) {
   int i,j;
   for(i=0; i<n; i++) {
      System.out.println("i="+i);
      for(j=0; j<i; j++) {
         System.out.println("j="+j);
      }
   }
}
```

$$T(n) = \sum_{i=0}^{i\le n}\sum_{j=0}^{j\le i} 1 = \sum_{i=0}^{i\le n} i = \frac{1}{2}n(n+1) \in \Theta(n^2) \Rightarrow \texttt{loop3} \in \Theta(n^2)$$

3.4.8 Example: loop4

```
public static void loop4(int n) {
   int i,j;
   for(i=0; i<n; i++) {
      System.out.println("i="+i);
      for(j=0; j<i*i; j++) {
         System.out.println("j="+j);
      }
   }
}
```

$$T(n) = \sum_{i=0}^{i\le n}\sum_{j=0}^{j\le i^2} 1 = \sum_{i=0}^{i\le n} i^2 = \frac{1}{6}n(n+1)(2n+1) \in \Theta(n^3) \Rightarrow \texttt{loop4} \in \Theta(n^3)$$

3.4.9 Example: factorial0

```
public static int factorial0(int n) {
   int i, prod=1;
   for(i=1; i<=n; i++) prod=prod*n;
   return n;
}
```

$$T(n) = \sum_{i=0}^{i\le n} 1 = n \in \Theta(n) \Rightarrow \texttt{factorial0} \in \Theta(n)$$

3.4.10 Example: concatenate0

```
public static void concatenate0(int n) {
   for(int i=0; i<n*n; i++) System.out.println(""+i);
   for(int i=0; i<n*n*n; i++) System.out.println(""+i);
}
```

$$T(n) = \Theta(\max(n^2, n^3)) \Rightarrow \texttt{concatenate0} \in \Theta(n^3)$$

3.4.11 Example: concatenate1

```
public static void concatenate1(int n, int a) {
   if(a<0)
      for(int i=0; i<n*n; i++) System.out.println(""+i);
   else
      for(int i=0; i<n*n*n; i++) System.out.println(""+i);
}
```

$$T(n) = \Theta(\max(n^2, n^3)) \Rightarrow \texttt{concatenate1} \in \Theta(n^3)$$

3.5 Recursive alorithms and recurrence relations

The running time $T(n)$ of recursive algorithms is determined by:

1. Writing the recurrence relation in $T(n)$
2. Solving the recurrence relation or, at least, put a bound on $T(n)$.

In these notes we assume $f(n)$ is a positive monotonic increasing function, $a \geq 1$ and $b \geq 2$.

For example:

```
public static int factorial1(int n) {
   if(n==0) return 1;
   else return n*factorial1(n-1);
}
```

$$T(n) = \left\{ \begin{array}{ll} c & \text{if } n = 1 \\ T(n-1) + 1 & \text{if } n > 1 \end{array} \right\} \Rightarrow T(n) =?, T(n) \in ?$$

Theorem 4 *If $P_m(n)$ is a polynomial of degree m*

$$T(n) = T(n-1) + P_m(n) \Rightarrow T(n) \in \Theta(n^{m+1})$$

Theorem 5 *If $P_m(n)$ is a polynomial of degree m, $a > 1$*

$$T(n) = aT(n-1) + P_m(n) \Rightarrow T(n) \in \Theta(a^n)$$

Theorem 6 *If $P_m(n)$ is a polynomial of degree m and $a < 1$*

$$T(n) = aT(n-1) + P_m(n) \Rightarrow T(n) \in \Theta(n^m)$$

Theorem 7 *For $m \geq 0$, $p \geq 0$ and $q > 1$, from the Master Theorem (in its most general form) we conclude that:*

$$\begin{aligned} T(n) &= aT(n/b) + \Theta(n^m) \text{ and } a < b^m \Rightarrow T(n) \in \Theta(n^m) \\ T(n) &= aT(n/b) + \Theta(n^m) \text{ and } a = b^m \Rightarrow T(n) \in \Theta(n^m \log n) \end{aligned}$$

$$
\begin{aligned}
T(n) &= aT(n/b) + \Theta(n^m) \textit{ and } a > b^m \Rightarrow T(n) \in \Theta(n^{\log_m a}) \\
T(n) &= aT(n/b) + \Theta(n^m \log^p n) \textit{ and } a < b^m \Rightarrow T(n) \in \Theta(n^m \log^p n) \\
T(n) &= aT(n/b) + \Theta(n^m \log^p n) \textit{ and } a = b^m \Rightarrow T(n) \in \Theta(n^m \log^{p+1} n) \\
T(n) &= aT(n/b) + \Theta(n^m \log^p n) \textit{ and } a > b^m \Rightarrow T(n) \in \Theta(n^{\log_b a}) \\
T(n) &= aT(n/b) + \Theta(q^n) \Rightarrow T(n) \in \Theta(q^n)
\end{aligned}
$$

Theorem 8 *For any constant $a > 0, b > 0$ and $c > 0$*

$$T(n) = T(a) + T(n - a - b) + c \Rightarrow T(n) \in \Theta(n)$$

3.5.1 Example: factorial1

```
public static int factorial1(int n) {
   if(n==0) return 1;
   else return n*factorial1(n-1);
}
```

$$T(n) = T(n-1) + 1 \Rightarrow T(n) \in \Theta(n) \Rightarrow \texttt{factorial1} \in \Theta(n)$$

3.5.2 Example: recursive0

```
public static int recursive0(int n) {
   if(n==0) return 1;
   else {
      loop3(n)
      return n*n*recursive0(n-1);
   }
}
```

$$T(n) = T(n-1) + P_3(n) \Rightarrow T(n) \in \Theta(n^4) \Rightarrow \texttt{recursive0} \in \Theta(n^4)$$

3.5.3 Example: recursive1

```
public static int recursive1(int n) {
   if(n==0) return 1;
   else {
      loop3(n)
      return n*recursive1(n-1)*recursive1(n-1);
   }
}
```

$$T(n) = 2T(n-1) + P_3(n) \Rightarrow T(n) \in \Theta(2^n) \Rightarrow \texttt{recursive1} \in \Theta(2^n)$$

3.5.4 Example: recursive2

```
public static int recursive2(int n) {
   if(n==0) return 1;
   else {
      int a;
      a=factorial0(n)
      return a*recursive2(n/2)*recursive2(n/2);
   }
}
```

$$T(n) = 2T(n/2) + P_1(n) \Rightarrow T(n) \in \Theta(n \log n) \Rightarrow \texttt{recursive2} \in \Theta(n \log n)$$

3.5.5 Example: binarySearch

```
public static int binarySearch(Comparable[] a, Comparable x) {
   int low=0, high=a.length-1;
   while(high>=low) {
      int mid=(high-low)/2;
      if(a[mid].compareTo(x)<0) low=mid+1;
      else if(a[mid].compareTo(x)>0) high=mid-1;
      else return mid;
   }
   return -1;
}
```

$$T(n) = T(n/2) + 1 \Rightarrow \texttt{binarySearch} \in O(\log n)$$

3.5.6 Example: Euclid Algorithm for the greatest common divisor

```
public static long gcd(long m, long n) {
   while(n>0) {
      long tmp=m % n;
      m=n;
      n=tmp;
   }
   return m;
}
```

$$\texttt{gcd} \in O(\log n)$$

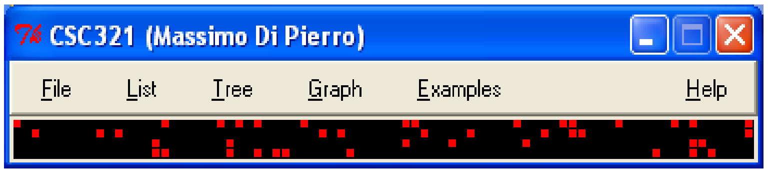
Note that this algorithm is the first known algorihtm and was invented by Euclid in 400b.c. This algorithm plays a crucial role in the RSA encryption scheme used in public key cryptography (secure internet transations).

4 Algorithms Animator and Data Structures

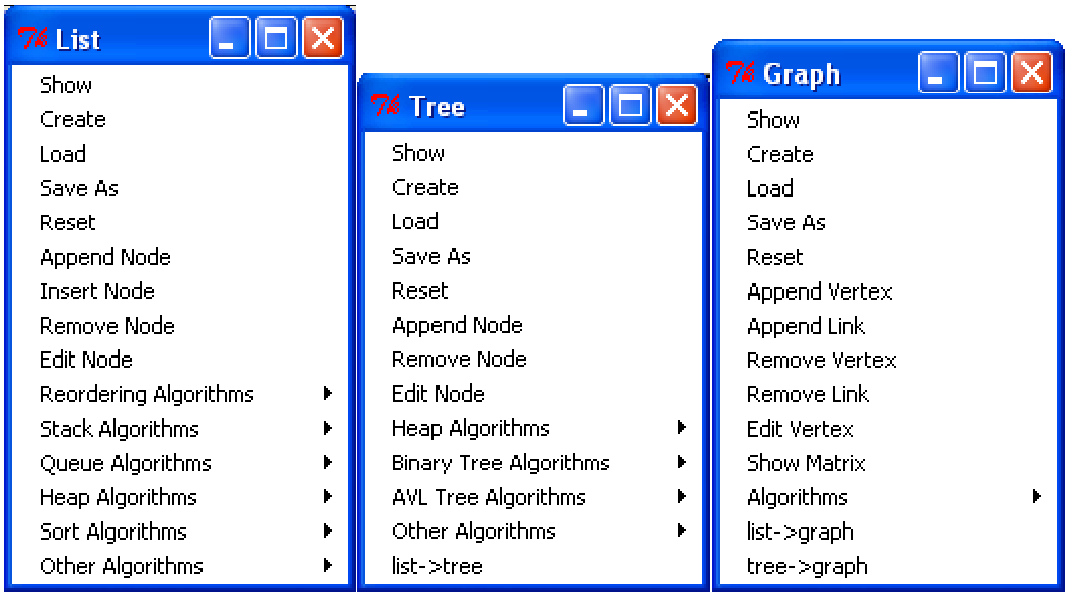
The Algorithms Animator program can be downloaded from:

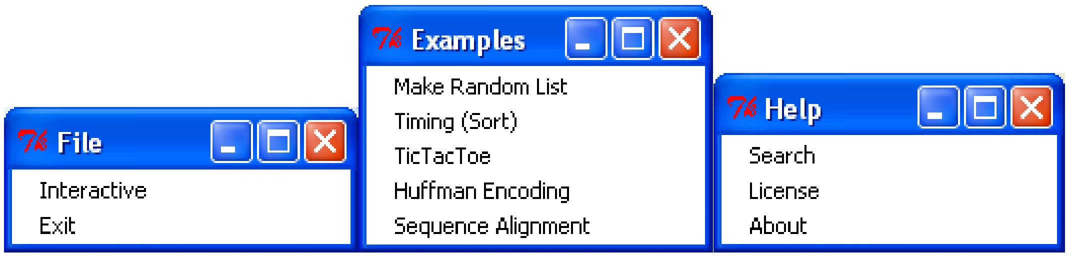
http://www.phoenixcollective.org/mdp/csc321.exe

The Algorithms Animator program is implemented in Python and based on the Tkinter graphic library and the Python Mega Widgets (Pmw). Python is an interpreted language invented by Guido van Rossum. This language was chosen because its syntax closely resembles the syntax commonly used to write pseudo-codes. Moreover Python includes, as basic types, structures such as tuples, lists and dictionaries. The version of the program distributed with these notes is packaged for Windows. Versions for other platforms are available upon request.



The main menus are





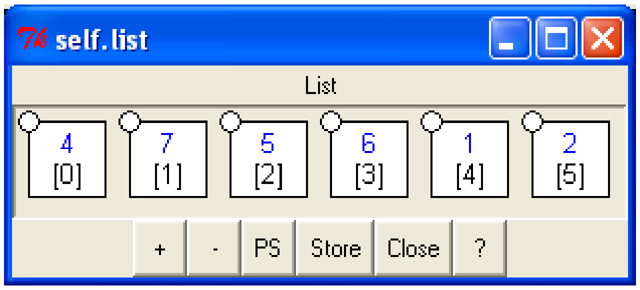
(16)

4.1 Lists

To start creating a list choose [**List**][**Create**] and type a Python list, for example:

$$[4, 7, 5, 6, 1, 2]$$

which is displayed as



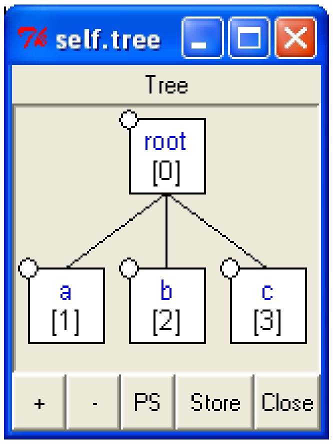
Note that each Node (in a list or a tree or a graph) can have different attributes: a **value**, a **name**, a **color**, etc. By default, when a list, a tree or a graph is created, only the value of the Nodes is specified ('root', 'a', 'b', 'c', 3, 4, 5). One should not confuse the value of a Node (for example 4 in the example) with its **location** ([0] in the example). The location is an index number which is automatically generated and is unique for each Node in a list, tree or graph. While the value of a node is not necessarily unique, the location index is unique.

4.2 Trees

In CSC321 a tree is implemented as a list of lists. For example, create the following list:

$$['root',' a',' b',' c']$$

which is displayed as:

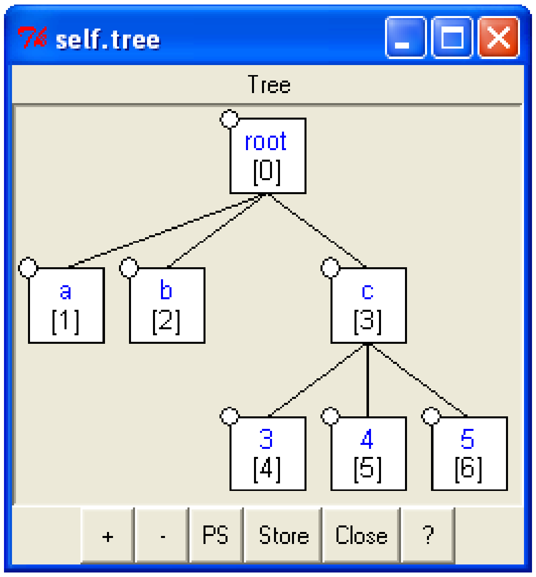


(17)

To add children Nodes to the Node c, one may create a new list where Node c itself is replaced by a subtree. For example, create the following list:

$$['root','a','b',['c',3,4,5]]$$

which is displayed as:



(18)

In general each Node in a tree can have an arbitrary number of children. It is possible to specialize a tree by imposing some constraints on it. In this course we examine three types of special trees:

- **Heaps**
- **Binary Trees**

- **AVL Tree**

4.3 Graphs

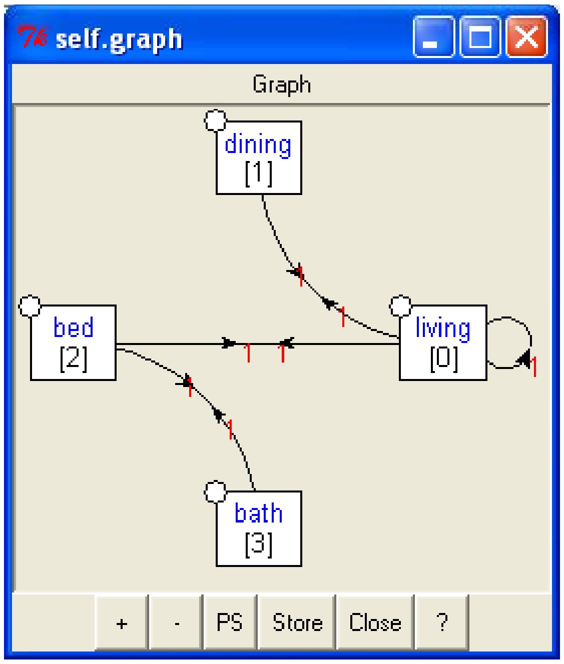
A graph is a collection of Nodes (with the same attributes as List Nodes and Tree Nodes) plus a collection of links between Nodes. In general links may have directions (for example Node [0] may be linked to Node [1] but not vice versa). Moreover each Node may be linked to itself (Node [0] may be linked to Node[0]).

In the Algorithms Animator a graph is implemented as a List of Lists where the element 0 of each sub-list is the value of the corresponding Node. The other elements of each sub-list are the location indices of the Nodes connected to the Node itself (**Adjacency List Representation**).

To start creating a graph choose [**Graph**][**Create**] and type a graph, for example:

$$[['living','dining','bed','bath],[[0,0],[0,1],[1,0],[0,2],[2,0],[2,3],[3,2]]]$$

that represents the topology of a one bedroom apartment. This graph is displayed as:



(19)

A graphs is represented as two lists: a list of vertices/nodes (for example: $V = ['living','dining','bed','bath'])$ and a list of links.

- The living room (Node [0]) is connected to itself [0,0] (has an island kitchen?) to the dining room [0,1] and to the bedroom [0,2].

- The dining room (Node [1]) is connected only to the living room [1,0].
- The bedroom (Node [2]) is connected to the living room [2,0] and the bathroom [2,3].
- The bathroom (Node [3]) is connected to the bedroom only [3,2].

In the representation of the graph the number 1 attached to each link represents the length of the link (this is one by default).

5 Algorithms

5.1 Defintions

Definition 9 *The **Divide-and-Conquer** is a method of designing algorithms that (informally) proceeds as follows: Given an instance of the problem to be solved, split this into several, smaller, sub-instances (of the same problem) independently solve each of the sub-instances and then combine the sub-instance solutions so as to yield a solution for the original instance. This description raises the question: By what methods are the sub-instances to be independently solved? The answer to this question is central to the concept of the Divide-and-Conquer algorithm and is a key factor in gauging their efficiency. The solution depends on the problem.*

Definition 10 ***Dynamic Programming** is a paradigm that is most often applied in the construction of algorithms to solve a certain class of optimization problems. That is problems which require the minimization or maximization of some measure. One disadvantage of using Divide-and-Conquer is that the process of recursively solving separate sub-instances can result in the same computations being performed repeatedly since identical sub-instances may arise. The idea behind dynamic programming is to avoid this pathology by obviating the requirement to calculate the same quantity twice. The method usually accomplishes this by maintaining a table of sub-instance results. We say that Dynamic Programming is a Bottom-Up technique in which the smallest sub-instances are explicitly solved first and the results of these used to construct solutions to progressively larger sub-instances. In contrast, we say that the Divide-and-Conquer is a Top-Down technique. One case of Top-Down approach, called **memoization,** that is normally included under the umbrella of Dynamic Programming. The reason is that the memoization approch uses tables to store intermediate results.*

Definition 11 ***Greedy algorithms** work in phases. In each phase, a decision is made that appears to be good, without regard for future consequences. Generally, this means that some local optimum is chosen. This 'take what you can get now' strategy is the source of the name for this class of algorithms. When the algorithm terminates, we hope that the local optimum is equal to the global optimum. If this is the case, then the algorithm is correct; otherwise, the algorithm has produced a suboptimal solution. If the best answer is not required, then simple greedy algorithms are sometimes used to generate approximate answers, rather than using the more complicated algorithms generally required to generate an exact answer. Even for problems which can be solved exactly by a greedy algorithm, establishing the correctness of the method may be a non-trivial process.*

There are other types of algorithms that do not follow in any of the above categories. One is, for example, backtracking. Backtracking is not covered in this course.

6 Arrays and Lists

6.1 Arrays

```
int size=10;
Object a[]=new Object[size];
for(int i=0; i<a.length; i++)
    a[i]=new Integer(i);
for(int i=0; i<a.length; i++)
    System.out.println("a["+i+"]="+((Integer) a[i]).intValue());
```

Arrays cannot be resized. We cannot append, remove, insert and element.

6.2 Vectors/Lists using arrays

```
public class MyVector  {
    Object data[];
    int datasize;
    int size;
    public MyVector() {
        datasize=10;
        size=0;
        data=new Object[datasize];
    }
    public MyVector(int n) {
        datasize=size=n;
        data=new Object[datasize];
    }
    public int length() {
        return size;
    }
    public void clear() {
        size=0;
    }
    public Object get(int i) throws Exception {
        if(i<0 || i>=size) throw new Exception();
        return data[i];
    }
    public void replace(int i, Object x) throws Exception {
        if(i<0 || i>=size) throw new Exception();
        data[i]=x;
    }
    public void remove(int i) throws Exception {
        if(i<0 || i>=size) throw new Exception();
        size--;
        int j;
        if(size>datasize/2) {
```

```
        for(j=i;j<size;j++) data[j]=data[j+1];
    } else {
        datasize=datasize/2;
        Object tmpdata[]=new Object[datasize];
        for(j=0;j<i;j++) tmpdata[j]=data[j];
        for(j=i;j<size;j++) tmpdata[j]=data[j+1];
        data=tmpdata;
    }
}
public void insert(int i, Object x) throws Exception {
    if(i<0 || i>=size) throw new Exception();
    size++;
    int j;
    if(size<=datasize) {
        for(j=i+1;j<size;j++) data[j]=data[j-1];
        data[i]=x;
    } else {
        datasize=datasize*2;
        Object tmpdata[]=new Object[datasize];
        for(j=0;j<i;j++) tmpdata[j]=data[j];
        tmpdata[i]=x;
        for(j=i+1;j<size;j++) tmpdata[j]=data[j-1];
        data=tmpdata;
    }
}
public void append(Object x) {
    size=size+1;
    if(size<=datasize) {
        data[size-1]=x;
    } else {
        int j;
        datasize=datasize*2;
        Object tmpdata[]=new Object[datasize];
        for(j=0;j<size-1;j++) tmpdata[j]=data[j];
        tmpdata[size-1]=x;
        data=tmpdata;
    }
}
public static void main(String args[]) {
    try {
        MyVector v=new MyVector();
        int i;
        for(i=0; i<20;i++)
            v.append(new Integer(i));
        for(i=0; i<v.length(); i++)
            System.out.println("v["+i+"]="+
                ((Integer) v.get(i)).intValue());
        } catch(Exception e) {
            System.out.println("Exception:"+e);
        }
    }
}
```

This implementation of a Vector uses a trick. It uses a buffer (called data) to store the elements. The size of the buffer (datasize) is bigger then the logical size of the vector (size). When size<datasize/2 the buffer is copied into a new buffer of half datasize. When size exceeds datasize the bufer is copied into a new buffer of double datasize.

The class MyVector allows fast sequential access to its elemnets (in fact get() and replace() are $O(1)$ since contain no loops) while the other methods are slow (in fact remove(), insert() and append() are $O(n)$ because contain one loop).

6.3 Lists

```
public class MyList {
    private class MyListNode {
        Object value;
        MyListNode next;
        MyListNode() {
            value=null;
            next=null;
        }
        MyListNode(Object x, MyListNode n) {
            value=x;
            next=n;
        }
    }
    MyListNode head,tail;
    int size;
    /** Creates a new instance of MyList */
    public MyList() {
        clear();
    }
    public int length() {
        return size;
    }
    public void clear() {
        head=tail=null;
        size=0;
    }
    public Object get(int i) throws Exception {
        if(i<0 || i>size) throw new Exception();
    MyListNode p=head;
    for(;i>0;i--) p=p.next;
    return p.value;
}

public void replace(int i, Object x) throws Exception {
    if(i<0 || i>=size) throw new Exception();
    MyListNode p=head;
    for(;i>0;i--) p=p.next;
    p.value=x;
}
public void remove(int i) throws Exception {
    if(i<0 || i>=size) throw new Exception();
    size--;
    if(i==0) {
        head=head.next;
        if(i==size) tail=head;
    } else {
        MyListNode p=head;
        for(;i>1;i--) p=p.next;
        p.next=p.next.next;
        if(i==size) tail=p;
    }
}
public void insert(int i, Object x) throws Exception {
    if(i<0 || i>=size) throw new Exception();
    if(i==0) {
        head=new MyListNode(x,head);
    } else {
        MyListNode p=head;
        for(;i>1;i--) p=p.next;
        p.next=new MyListNode(x,p.next);
    }
    size++;
}
public void append(Object x) {
    if(size==0) {
        head=new MyListNode(x,null);
        tail=head;
    } else {
        tail.next=new MyListNode(x,null);
        tail=tail.next;
    }
    size++;
}
public static void main(String args[]) {
        try {
            MyList list=new MyList();       // []
            list.append(new Integer(3));    // [3]
            list.insert(0,new Integer(4));  // [4,3]
            list.replace(1,new Integer(2)); // [4,2]
            list.append(new Integer(1));    // [4,2,1]
            list.remove(0);                 // [2,1]
            for(int i=0; i<list.length(); i++)
                System.out.println("list["+i+"]="+
                    ((Integer) list.get(i)).intValue());
        } catch (Exception e) {
            System.out.println("Exception:"+e);
        }
    }
}
```

This implemetation of lists allows to append in $O(1)$ but insert, remove, replace and get are $O(n)$.

7 Book Programs

7.1 AANode.java

```
package DataStructures;

// Basic node stored in AVL trees
// Note that this class is not accessible outside
// of package DataStructures

class AANode
{
        // Constructors
    AANode( Comparable theElement )
    {
        this( theElement, null, null );
    }

    AANode( Comparable theElement, AANode lt, AANode rt )
    {
        element  = theElement;
        left     = lt;
        right    = rt;
        level    = 1;
    }

        // Friendly data; accessible by other package routines
    Comparable element;      // The data in the node
    AANode     left;         // Left child
    AANode     right;        // Right child
    int        level;        // Level
}
```

7.2 AATree.java

```
package DataStructures;

// AATree class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x
// Comparable find( x )   --> Return item that matches x
// Comparable findMin( )  --> Return smallest item
// Comparable findMax( )  --> Return largest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// void printTree( )      --> Print tree in sorted order

/**
 * Implements an AA-tree.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 */
public class AATree
{
    /**
     * Construct the tree.
     */
    public AATree( )
    {
        root = nullNode;
    }

    /**
     * Insert into the tree. Does nothing if x is already present.
     * @param x the item to insert.
     */
    public void insert( Comparable x )
    {
        root = insert( x, root );
    }

    /**
     * Remove from the tree. Does nothing if x is not found.
     * @param x the item to remove.
     */
    public void remove( Comparable x )
    {
        deletedNode = nullNode;
        root = remove( x, root );
    }

    /**
     * Find the smallest item in the tree.
     * @return the smallest item or null if empty.
     */
    public Comparable findMin( )
    {
```

```
    if( isEmpty( ) )
        return null;

    AANode ptr = root;

    while( ptr.left != nullNode )
        ptr = ptr.left;

    return ptr.element;
}

/**
 * Find the largest item in the tree.
 * @return the largest item or null if empty.
 */
public Comparable findMax( )
{
    if( isEmpty( ) )
        return null;

    AANode ptr = root;

    while( ptr.right != nullNode )
        ptr = ptr.right;

    return ptr.element;
}

/**
 * Find an item in the tree.
 * @param x the item to search for.
 * @return the matching item of null if not found.
 */
public Comparable find( Comparable x )
{
    AANode current = root;
    nullNode.element = x;

    for( ; ; )
    {
        if( x.compareTo( current.element ) < 0 )
            current = current.left;
        else if( x.compareTo( current.element ) > 0 )
            current = current.right;
        else if( current != nullNode )
            return current.element;
        else
            return null;
    }
}

/**
 * Make the tree logically empty.
 */
public void makeEmpty( )
{
    root = nullNode;
```

```
}

/**
 * Test if the tree is logically empty.
 * @return true if empty, false otherwise.
 */
public boolean isEmpty( )
{
    return root == nullNode;
}

/**
 * Print the tree contents in sorted order.
 */
public void printTree( )
{
    if( isEmpty( ) )
        System.out.println( "Empty tree" );
    else
        printTree( root );
}

/**
 * Internal method to insert into a subtree.
 * @param x the item to insert.
 * @param t the node that roots the tree.
 * @return the new root.
 */
private AANode insert( Comparable x, AANode t )
{
    if( t == nullNode )
        t = new AANode( x, nullNode, nullNode );
    else if( x.compareTo( t.element ) < 0 )
        t.left = insert( x, t.left );
    else if( x.compareTo( t.element ) > 0 )
        t.right = insert( x, t.right );
    else
        return t;

    t = skew( t );
    t = split( t );
    return t;
}

/**
 * Internal method to remove from a subtree.
 * @param x the item to remove.
 * @param t the node that roots the tree.
 * @return the new root.
 */
private AANode remove( Comparable x, AANode t )
{
    if( t != nullNode )
    {
        // Step 1: Search down the tree and set lastNode and deletedNode
        lastNode = t;
        if( x.compareTo( t.element ) < 0 )
```

```
                t.left = remove( x, t.left );
            else
            {
                deletedNode = t;
                t.right = remove( x, t.right );
            }

            // Step 2: If at the bottom of the tree and
            //         x is present, we remove it
            if( t == lastNode )
            {
                if( deletedNode == nullNode || x.compareTo( deletedNode.element ) != 0 )
                    return t;   // Item not found; do nothing
                deletedNode.element = t.element;
                t = t.right;
            }

            // Step 3: Otherwise, we are not at the bottom; rebalance
            else
                if( t.left.level < t.level - 1 || t.right.level < t.level - 1 )
                {
                    if( t.right.level > --t.level )
                        t.right.level = t.level;
                    t = skew( t );
                    t.right = skew( t.right );
                    t.right.right = skew( t.right.right );
                    t = split( t );
                    t.right = split( t.right );
                }
        }
        return t;
    }

    /**
     * Internal method to print a subtree in sorted order.
     * @param t the node that roots the tree.
     */
    private void printTree( AANode t )
    {
        if( t != t.left )
        {
            printTree( t.left );
            System.out.println( t.element.toString( ) );
            printTree( t.right );
        }
    }

    /**
     * Skew primitive for AA-trees.
     * @param t the node that roots the tree.
     * @return the new root after the rotation.
     */
    private AANode skew( AANode t )
    {
        if( t.left.level == t.level )
            t = rotateWithLeftChild( t );
        return t;
```

```
}

/**
 * Split primitive for AA-trees.
 * @param t the node that roots the tree.
 * @return the new root after the rotation.
 */
private AANode split( AANode t )
{
    if( t.right.right.level == t.level )
    {
        t = rotateWithRightChild( t );
        t.level++;
    }
    return t;
}

/**
 * Rotate binary tree node with left child.
 */
static AANode rotateWithLeftChild( AANode k2 )
{
    AANode k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    return k1;
}

/**
 * Rotate binary tree node with right child.
 */
static AANode rotateWithRightChild( AANode k1 )
{
    AANode k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    return k2;
}

private AANode root;
private static AANode nullNode;
    static         // static initializer for nullNode
    {
        nullNode = new AANode( null );
        nullNode.left = nullNode.right = nullNode;
        nullNode.level = 0;
    }

private static AANode deletedNode;
private static AANode lastNode;

    // Test program; should print min and max and nothing else
public static void main( String [ ] args )
{
    AATree t = new AATree( );
    final int NUMS = 40000;
    final int GAP  =   307;
```

```
        System.out.println( "Checking... (no bad output means success)" );

        for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
            t.insert( new MyInteger( i ) );
        System.out.println( "Inserts complete" );

        for( int i = 1; i < NUMS; i+= 2 )
            t.remove( new MyInteger( i ) );
        System.out.println( "Removes complete" );

        if( NUMS < 40 )
            t.printTree( );
        if( ((MyInteger)(t.findMin( ))).intValue( ) != 2 ||
            ((MyInteger)(t.findMax( ))).intValue( ) != NUMS - 2 )
            System.out.println( "FindMin or FindMax error!" );

        for( int i = 2; i < NUMS; i+=2 )
            if( ((MyInteger)t.find( new MyInteger( i ) )).intValue( ) != i )
                System.out.println( "Error: find fails for " + i );

        for( int i = 1; i < NUMS; i+=2 )
            if( t.find( new MyInteger( i ) )  != null )
                System.out.println( "Error: Found deleted item " + i );
    }
}
```

7.3 AvlNode.java

```
package DataStructures;

// Basic node stored in AVL trees
// Note that this class is not accessible outside
// of package DataStructures

class AvlNode
{
        // Constructors
    AvlNode( Comparable theElement )
    {
        this( theElement, null, null );
    }

    AvlNode( Comparable theElement, AvlNode lt, AvlNode rt )
    {
        element  = theElement;
        left     = lt;
        right    = rt;
        height   = 0;
    }

        // Friendly data; accessible by other package routines
    Comparable element;      // The data in the node
    AvlNode    left;         // Left child
    AvlNode    right;        // Right child
    int        height;       // Height
}
```

7.4 AvlTree.java

```
package DataStructures;

// BinarySearchTree class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x (unimplemented)
// Comparable find( x )   --> Return item that matches x
// Comparable findMin( )  --> Return smallest item
// Comparable findMax( )  --> Return largest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// void printTree( )      --> Print tree in sorted order

/**
 * Implements an AVL tree.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 */
public class AvlTree
{
    /**
     * Construct the tree.
     */
    public AvlTree( )
    {
        root = null;
    }

    /**
     * Insert into the tree; duplicates are ignored.
     * @param x the item to insert.
     */
    public void insert( Comparable x )
    {
        root = insert( x, root );
    }

    /**
     * Remove from the tree. Nothing is done if x is not found.
     * @param x the item to remove.
     */
    public void remove( Comparable x )
    {
        System.out.println( "Sorry, remove unimplemented" );
    }

    /**
     * Find the smallest item in the tree.
     * @return smallest item or null if empty.
     */
    public Comparable findMin( )
    {
        return elementAt( findMin( root ) );
```

```
}

/**
 * Find the largest item in the tree.
 * @return the largest item of null if empty.
 */
public Comparable findMax( )
{
    return elementAt( findMax( root ) );
}

/**
 * Find an item in the tree.
 * @param x the item to search for.
 * @return the matching item or null if not found.
 */
public Comparable find( Comparable x )
{
    return elementAt( find( x, root ) );
}

/**
 * Make the tree logically empty.
 */
public void makeEmpty( )
{
    root = null;
}

/**
 * Test if the tree is logically empty.
 * @return true if empty, false otherwise.
 */
public boolean isEmpty( )
{
    return root == null;
}

/**
 * Print the tree contents in sorted order.
 */
public void printTree( )
{
    if( isEmpty( ) )
        System.out.println( "Empty tree" );
    else
        printTree( root );
}

/**
 * Internal method to get element field.
 * @param t the node.
 * @return the element field or null if t is null.
 */
private Comparable elementAt( AvlNode t )
{
    return t == null ? null : t.element;
```

```
}

/**
 * Internal method to insert into a subtree.
 * @param x the item to insert.
 * @param t the node that roots the tree.
 * @return the new root.
 */
private AvlNode insert( Comparable x, AvlNode t )
{
    if( t == null )
        t = new AvlNode( x, null, null );
    else if( x.compareTo( t.element ) < 0 )
    {
        t.left = insert( x, t.left );
        if( height( t.left ) - height( t.right ) == 2 )
            if( x.compareTo( t.left.element ) < 0 )
                t = rotateWithLeftChild( t );
            else
                t = doubleWithLeftChild( t );
    }
    else if( x.compareTo( t.element ) > 0 )
    {
        t.right = insert( x, t.right );
        if( height( t.right ) - height( t.left ) == 2 )
            if( x.compareTo( t.right.element ) > 0 )
                t = rotateWithRightChild( t );
            else
                t = doubleWithRightChild( t );
    }
    else
        ;  // Duplicate; do nothing
    t.height = max( height( t.left ), height( t.right ) ) + 1;
    return t;
}

/**
 * Internal method to find the smallest item in a subtree.
 * @param t the node that roots the tree.
 * @return node containing the smallest item.
 */
private AvlNode findMin( AvlNode t )
{
    if( t == null )
        return t;

    while( t.left != null )
        t = t.left;
    return t;
}

/**
 * Internal method to find the largest item in a subtree.
 * @param t the node that roots the tree.
 * @return node containing the largest item.
 */
private AvlNode findMax( AvlNode t )
```

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

    while( t.right != null )
        t = t.right;
    return t;
}

/**
 * Internal method to find an item in a subtree.
 * @param x is item to search for.
 * @param t the node that roots the tree.
 * @return node containing the matched item.
 */
private AvlNode find( Comparable x, AvlNode t )
{
    while( t != null )
        if( x.compareTo( t.element ) < 0 )
            t = t.left;
        else if( x.compareTo( t.element ) > 0 )
            t = t.right;
        else
            return t;    // Match

    return null;   // No match
}

/**
 * Internal method to print a subtree in sorted order.
 * @param t the node that roots the tree.
 */
private void printTree( AvlNode t )
{
    if( t != null )
    {
        printTree( t.left );
        System.out.println( t.element );
        printTree( t.right );
    }
}

/**
 * Return the height of node t, or -1, if null.
 */
private static int height( AvlNode t )
{
    return t == null ? -1 : t.height;
}

/**
 * Return maximum of lhs and rhs.
 */
private static int max( int lhs, int rhs )
{
    return lhs > rhs ? lhs : rhs;
}
```

```
/**
 * Rotate binary tree node with left child.
 * For AVL trees, this is a single rotation for case 1.
 * Update heights, then return new root.
 */
private static AvlNode rotateWithLeftChild( AvlNode k2 )
{
    AvlNode k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    k2.height = max( height( k2.left ), height( k2.right ) ) + 1;
    k1.height = max( height( k1.left ), k2.height ) + 1;
    return k1;
}

/**
 * Rotate binary tree node with right child.
 * For AVL trees, this is a single rotation for case 4.
 * Update heights, then return new root.
 */
private static AvlNode rotateWithRightChild( AvlNode k1 )
{
    AvlNode k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    k1.height = max( height( k1.left ), height( k1.right ) ) + 1;
    k2.height = max( height( k2.right ), k1.height ) + 1;
    return k2;
}

/**
 * Double rotate binary tree node: first left child
 * with its right child; then node k3 with new left child.
 * For AVL trees, this is a double rotation for case 2.
 * Update heights, then return new root.
 */
private static AvlNode doubleWithLeftChild( AvlNode k3 )
{
    k3.left = rotateWithRightChild( k3.left );
    return rotateWithLeftChild( k3 );
}

/**
 * Double rotate binary tree node: first right child
 * with its left child; then node k1 with new right child.
 * For AVL trees, this is a double rotation for case 3.
 * Update heights, then return new root.
 */
private static AvlNode doubleWithRightChild( AvlNode k1 )
{
    k1.right = rotateWithLeftChild( k1.right );
    return rotateWithRightChild( k1 );
}

  /** The tree root. */
private AvlNode root;
```

```
        // Test program
    public static void main( String [ ] args )
    {
        AvlTree t = new AvlTree( );
        final int NUMS = 4000;
        final int GAP  =   37;

        System.out.println( "Checking... (no more output means success)" );

        for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
            t.insert( new MyInteger( i ) );

        if( NUMS < 40 )
            t.printTree( );
        if( ((MyInteger)(t.findMin( ))).intValue( ) != 1 ||
            ((MyInteger)(t.findMax( ))).intValue( ) != NUMS - 1 )
            System.out.println( "FindMin or FindMax error!" );

        for( int i = 1; i < NUMS; i++ )
             if( ((MyInteger)(t.find( new MyInteger( i ) ))).intValue( ) != i )
                 System.out.println( "Find error1!" );
    }
}
```

7.5 BinaryHeap.java

```
package DataStructures;

// BinaryHeap class
//
// CONSTRUCTION: with optional capacity (that defaults to 100)
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// Comparable deleteMin( )--> Return and remove smallest item
// Comparable findMin( )  --> Return smallest item
// boolean isEmpty( )     --> Return true if empty; else false
// boolean isFull( )      --> Return true if full; else false
// void makeEmpty( )      --> Remove all items
// ******************ERRORS********************************
// Throws Overflow if capacity exceeded

/**
 * Implements a binary heap.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 */
public class BinaryHeap
{
    /**
     * Construct the binary heap.
     */
    public BinaryHeap( )
    {
        this( DEFAULT_CAPACITY );
    }

    /**
     * Construct the binary heap.
     * @param capacity the capacity of the binary heap.
     */
    public BinaryHeap( int capacity )
    {
        currentSize = 0;
        array = new Comparable[ capacity + 1 ];
    }

    /**
     * Insert into the priority queue, maintaining heap order.
     * Duplicates are allowed.
     * @param x the item to insert.
     * @exception Overflow if container is full.
     */
    public void insert( Comparable x ) throws Overflow
    {
        if( isFull( ) )
            throw new Overflow( );

            // Percolate up
        int hole = ++currentSize;
        for( ; hole > 1 && x.compareTo( array[ hole / 2 ] ) < 0; hole /= 2 )
            array[ hole ] = array[ hole / 2 ];
    array[ hole ] = x;
}

/**
 * Find the smallest item in the priority queue.
 * @return the smallest item, or null, if empty.
 */
public Comparable findMin( )
{
    if( isEmpty( ) )
        return null;
    return array[ 1 ];
}

/**
 * Remove the smallest item from the priority queue.
 * @return the smallest item, or null, if empty.
 */
public Comparable deleteMin( )
{
    if( isEmpty( ) )
        return null;

    Comparable minItem = findMin( );
    array[ 1 ] = array[ currentSize-- ];
    percolateDown( 1 );

    return minItem;
}

/**
 * Establish heap order property from an arbitrary
 * arrangement of items. Runs in linear time.
 */
private void buildHeap( )
{
    for( int i = currentSize / 2; i > 0; i-- )
        percolateDown( i );
}

/**
 * Test if the priority queue is logically empty.
 * @return true if empty, false otherwise.
 */
public boolean isEmpty( )
{
    return currentSize == 0;
}

/**
 * Test if the priority queue is logically full.
 * @return true if full, false otherwise.
 */
public boolean isFull( )
{
    return currentSize == array.length - 1;
}
```

```
        /**
         * Make the priority queue logically empty.
         */
        public void makeEmpty( )
        {
            currentSize = 0;
        }

        private static final int DEFAULT_CAPACITY = 100;

        private int currentSize;      // Number of elements in heap
        private Comparable [ ] array; // The heap array

        /**
         * Internal method to percolate down in the heap.
         * @param hole the index at which the percolate begins.
         */
        private void percolateDown( int hole )
        {
/* 1*/      int child;
/* 2*/      Comparable tmp = array[ hole ];

/* 3*/      for( ; hole * 2 <= currentSize; hole = child )
            {
/* 4*/          child = hole * 2;
/* 5*/          if( child != currentSize &&
/* 6*/                  array[ child + 1 ].compareTo( array[ child ] ) < 0 )
/* 7*/              child++;
/* 8*/          if( array[ child ].compareTo( tmp ) < 0 )
/* 9*/              array[ hole ] = array[ child ];
                else
/*10*/              break;
            }
/*11*/      array[ hole ] = tmp;
        }

            // Test program
        public static void main( String [ ] args )
        {
            int numItems = 10000;
            BinaryHeap h = new BinaryHeap( numItems );
            int i = 37;

            try
            {
                for( i = 37; i != 0; i = ( i + 37 ) % numItems )
                    h.insert( new MyInteger( i ) );
                for( i = 1; i < numItems; i++ )
                    if( ((MyInteger)( h.deleteMin( ) )).intValue( ) != i )
                        System.out.println( "Oops! " + i );

                for( i = 37; i != 0; i = ( i + 37 ) % numItems )
                    h.insert( new MyInteger( i ) );
                h.insert( new MyInteger( 0 ) );
                i = 9999999;
                h.insert( new MyInteger( i ) );
            for( i = 1; i <= numItems; i++ )
                if( ((MyInteger)( h.deleteMin( ) )).intValue( ) != i )
                    System.out.println( "Oops! " + i + " " );
        }
        catch( Overflow e )
          { System.out.println( "Overflow (expected)! " + i  ); }
    }
}
```

7.6 BinaryNode.java

```
package DataStructures;

// Basic node stored in unbalanced binary search trees
// Note that this class is not accessible outside
// of package DataStructures

class BinaryNode
{
        // Constructors
    BinaryNode( Comparable theElement )
    {
        this( theElement, null, null );
    }

    BinaryNode( Comparable theElement, BinaryNode lt, BinaryNode rt )
    {
        element  = theElement;
        left     = lt;
        right    = rt;
    }

        // Friendly data; accessible by other package routines
    Comparable element;      // The data in the node
    BinaryNode left;         // Left child
    BinaryNode right;        // Right child
}
```

7.7 BinarySearchTree.java

```
package DataStructures;

// BinarySearchTree class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x
// Comparable find( x )   --> Return item that matches x
// Comparable findMin( )  --> Return smallest item
// Comparable findMax( )  --> Return largest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// void printTree( )      --> Print tree in sorted order

/**
 * Implements an unbalanced binary search tree.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 */
public class BinarySearchTree
{
    /**
     * Construct the tree.
     */
    public BinarySearchTree( )
    {
        root = null;
    }

    /**
     * Insert into the tree; duplicates are ignored.
     * @param x the item to insert.
     */
    public void insert( Comparable x )
    {
        root = insert( x, root );
    }

    /**
     * Remove from the tree. Nothing is done if x is not found.
     * @param x the item to remove.
     */
    public void remove( Comparable x )
    {
        root = remove( x, root );
    }

    /**
     * Find the smallest item in the tree.
     * @return smallest item or null if empty.
     */
    public Comparable findMin( )
    {
        return elementAt( findMin( root ) );
```

```
}

/**
 * Find the largest item in the tree.
 * @return the largest item of null if empty.
 */
public Comparable findMax( )
{
    return elementAt( findMax( root ) );
}

/**
 * Find an item in the tree.
 * @param x the item to search for.
 * @return the matching item or null if not found.
 */
public Comparable find( Comparable x )
{
    return elementAt( find( x, root ) );
}

/**
 * Make the tree logically empty.
 */
public void makeEmpty( )
{
    root = null;
}

/**
 * Test if the tree is logically empty.
 * @return true if empty, false otherwise.
 */
public boolean isEmpty( )
{
    return root == null;
}

/**
 * Print the tree contents in sorted order.
 */
public void printTree( )
{
    if( isEmpty( ) )
        System.out.println( "Empty tree" );
    else
        printTree( root );
}

/**
 * Internal method to get element field.
 * @param t the node.
 * @return the element field or null if t is null.
 */
private Comparable elementAt( BinaryNode t )
{
    return t == null ? null : t.element;
```

```
}

/**
 * Internal method to insert into a subtree.
 * @param x the item to insert.
 * @param t the node that roots the tree.
 * @return the new root.
 */
private BinaryNode insert( Comparable x, BinaryNode t )
{
    if( t == null )
        t = new BinaryNode( x, null, null );
    else if( x.compareTo( t.element ) < 0 )
        t.left = insert( x, t.left );
    else if( x.compareTo( t.element ) > 0 )
        t.right = insert( x, t.right );
    else
        ;  // Duplicate; do nothing
    return t;
}

/**
 * Internal method to remove from a subtree.
 * @param x the item to remove.
 * @param t the node that roots the tree.
 * @return the new root.
 */
private BinaryNode remove( Comparable x, BinaryNode t )
{
    if( t == null )
        return t;   // Item not found; do nothing
    if( x.compareTo( t.element ) < 0 )
        t.left = remove( x, t.left );
    else if( x.compareTo( t.element ) > 0 )
        t.right = remove( x, t.right );
    else if( t.left != null && t.right != null ) // Two children
    {
        t.element = findMin( t.right ).element;
        t.right = remove( t.element, t.right );
    }
    else
        t = ( t.left != null ) ? t.left : t.right;
    return t;
}

/**
 * Internal method to find the smallest item in a subtree.
 * @param t the node that roots the tree.
 * @return node containing the smallest item.
 */
private BinaryNode findMin( BinaryNode t )
{
    if( t == null )
        return null;
    else if( t.left == null )
        return t;
    return findMin( t.left );
```

```
}

/**
 * Internal method to find the largest item in a subtree.
 * @param t the node that roots the tree.
 * @return node containing the largest item.
 */
private BinaryNode findMax( BinaryNode t )
{
    if( t != null )
        while( t.right != null )
            t = t.right;

    return t;
}

/**
 * Internal method to find an item in a subtree.
 * @param x is item to search for.
 * @param t the node that roots the tree.
 * @return node containing the matched item.
 */
private BinaryNode find( Comparable x, BinaryNode t )
{
    if( t == null )
        return null;
    if( x.compareTo( t.element ) < 0 )
        return find( x, t.left );
    else if( x.compareTo( t.element ) > 0 )
        return find( x, t.right );
    else
        return t;    // Match
}

/**
 * Internal method to print a subtree in sorted order.
 * @param t the node that roots the tree.
 */
private void printTree( BinaryNode t )
{
    if( t != null )
    {
        printTree( t.left );
        System.out.println( t.element );
        printTree( t.right );
    }
}

  /** The tree root. */
private BinaryNode root;


    // Test program
public static void main( String [ ] args )
{
    BinarySearchTree t = new BinarySearchTree( );
    final int NUMS = 4000;
```

```
        final int GAP  =   37;

        System.out.println( "Checking... (no more output means success)" );

        for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
            t.insert( new MyInteger( i ) );

        for( int i = 1; i < NUMS; i+= 2 )
            t.remove( new MyInteger( i ) );

        if( NUMS < 40 )
            t.printTree( );
        if( ((MyInteger)(t.findMin( ))).intValue( ) != 2 ||
            ((MyInteger)(t.findMax( ))).intValue( ) != NUMS - 2 )
            System.out.println( "FindMin or FindMax error!" );

        for( int i = 2; i < NUMS; i+=2 )
             if( ((MyInteger)(t.find( new MyInteger( i ) ))).intValue( ) != i )
                 System.out.println( "Find error1!" );

        for( int i = 1; i < NUMS; i+=2 )
        {
            if( t.find( new MyInteger( i ) ) != null )
                System.out.println( "Find error2!" );
        }
    }
}
```

7.8 BinomialNode.java

```
package DataStructures;

// Basic node stored in binomial queues
// Note that this class is not accessible outside
// of package DataStructures

class BinomialNode
{
        // Constructors
    BinomialNode( Comparable theElement )
    {
        this( theElement, null, null );
    }

    BinomialNode( Comparable theElement, BinomialNode lt, BinomialNode nt )
    {
        element     = theElement;
        leftChild   = lt;
        nextSibling = nt;
    }

        // Friendly data; accessible by other package routines
    Comparable   element;     // The data in the node
    BinomialNode leftChild;   // Left child
    BinomialNode nextSibling; // Right child
}
```

7.9 BinomialQueue.java

```
package DataStructures;

// BinomialQueue class
//
// CONSTRUCTION: with a negative infinity sentinel
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// Comparable deleteMin( )--> Return and remove smallest item
// Comparable findMin( )  --> Return smallest item
// boolean isEmpty( )     --> Return true if empty; else false
// boolean isFull( )      --> Return true if full; else false
// void makeEmpty( )      --> Remove all items
// vod merge( rhs )       --> Absord rhs into this heap
// ******************ERRORS********************************
// Overflow if CAPACITY is exceeded

/**
 * Implements a binomial queue.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 */
public class BinomialQueue
{
    /**
     * Construct the binomial queue.
     */
    public BinomialQueue( )
    {
        theTrees = new BinomialNode[ MAX_TREES ];
        makeEmpty( );
    }

    /**
     * Merge rhs into the priority queue.
     * rhs becomes empty. rhs must be different from this.
     * @param rhs the other binomial queue.
     * @exception Overflow if result exceeds capacity.
     */
    public void merge( BinomialQueue rhs ) throws Overflow
    {
        if( this == rhs )    // Avoid aliasing problems
            return;

        if( currentSize + rhs.currentSize > capacity( ) )
            throw new Overflow( );

        currentSize += rhs.currentSize;

        BinomialNode carry = null;
        for( int i = 0, j = 1; j <= currentSize; i++, j *= 2 )
        {
            BinomialNode t1 = theTrees[ i ];
            BinomialNode t2 = rhs.theTrees[ i ];

            int whichCase = t1 == null ? 0 : 1;
            whichCase += t2 == null ? 0 : 2;
            whichCase += carry == null ? 0 : 4;

            switch( whichCase )
            {
              case 0: /* No trees */
              case 1: /* Only this */
                break;
              case 2: /* Only rhs */
                theTrees[ i ] = t2;
                rhs.theTrees[ i ] = null;
                break;
              case 4: /* Only carry */
                theTrees[ i ] = carry;
                carry = null;
                break;
              case 3: /* this and rhs */
                carry = combineTrees( t1, t2 );
                theTrees[ i ] = rhs.theTrees[ i ] = null;
                break;
              case 5: /* this and carry */
                carry = combineTrees( t1, carry );
                theTrees[ i ] = null;
                break;
              case 6: /* rhs and carry */
                carry = combineTrees( t2, carry );
                rhs.theTrees[ i ] = null;
                break;
              case 7: /* All three */
                theTrees[ i ] = carry;
                carry = combineTrees( t1, t2 );
                rhs.theTrees[ i ] = null;
                break;
            }
        }

        for( int k = 0; k < rhs.theTrees.length; k++ )
            rhs.theTrees[ k ] = null;
        rhs.currentSize = 0;
    }

    /**
     * Return the result of merging equal-sized t1 and t2.
     */
    private static BinomialNode combineTrees( BinomialNode t1,
                                              BinomialNode t2 )
    {
        if( t1.element.compareTo( t2.element ) > 0 )
            return combineTrees( t2, t1 );
        t2.nextSibling = t1.leftChild;
        t1.leftChild = t2;
        return t1;
    }

    /**
     * Insert into the priority queue, maintaining heap order.
     * This implementation is not optimized for O(1) performance.
```

```
     * @param x the item to insert.
     * @exception Overflow if capacity exceeded.
     */
    public void insert( Comparable x ) throws Overflow
    {
        BinomialQueue oneItem = new BinomialQueue( );
        oneItem.currentSize = 1;
        oneItem.theTrees[ 0 ] = new BinomialNode( x );

        merge( oneItem );
    }

    /**
     * Find the smallest item in the priority queue.
     * @return the smallest item, or null, if empty.
     */
    public Comparable findMin( )
    {
        if( isEmpty( ) )
            return null;

        return theTrees[ findMinIndex( ) ].element;
    }


    /**
     * Find index of tree containing the smallest item in the priority queue.
     * The priority queue must not be empty.
     * @return the index of tree containing the smallest item.
     */
    private int findMinIndex( )
    {
        int i;
        int minIndex;

        for( i = 0; theTrees[ i ] == null; i++ )
            ;

        for( minIndex = i; i < theTrees.length; i++ )
            if( theTrees[ i ] != null &&
                theTrees[ i ].element.compareTo( theTrees[ minIndex ].element ) < 0 )
                minIndex = i;

        return minIndex;
    }

    /**
     * Remove the smallest item from the priority queue.
     * @return the smallest item, or null, if empty.
     */
    public Comparable deleteMin( )
    {
        if( isEmpty( ) )
            return null;

        int minIndex = findMinIndex( );
        Comparable minItem = theTrees[ minIndex ].element;
        BinomialNode deletedTree = theTrees[ minIndex ].leftChild;

        BinomialQueue deletedQueue = new BinomialQueue( );
        deletedQueue.currentSize = ( 1 << minIndex ) - 1;
        for( int j = minIndex - 1; j >= 0; j-- )
        {
            deletedQueue.theTrees[ j ] = deletedTree;
            deletedTree = deletedTree.nextSibling;
            deletedQueue.theTrees[ j ].nextSibling = null;
        }

        theTrees[ minIndex ] = null;
        currentSize -= deletedQueue.currentSize + 1;

        try
          { merge( deletedQueue ); }
        catch( Overflow e ) { }
        return minItem;
    }

    /**
     * Test if the priority queue is logically empty.
     * @return true if empty, false otherwise.
     */
    public boolean isEmpty( )
    {
        return currentSize == 0;
    }

    /**
     * Test if the priority queue is logically full.
     * @return true if full, false otherwise.
     */
    public boolean isFull( )
    {
        return currentSize == capacity( );
    }

    /**
     * Make the priority queue logically empty.
     */
    public void makeEmpty( )
    {
        currentSize = 0;
        for( int i = 0; i < theTrees.length; i++ )
            theTrees[ i ] = null;
    }


    private static final int MAX_TREES = 14;

    private int currentSize;            // # items in priority queue
    private BinomialNode [ ] theTrees;  // An array of tree roots


    /**
```

```
     * Return the capacity.
     */
    private int capacity( )
    {
        return ( 1 << theTrees.length ) - 1;
    }

    public static void main( String [ ] args )
    {
        int numItems = 10000;
        BinomialQueue h  = new BinomialQueue( );
        BinomialQueue h1 = new BinomialQueue( );
        int i = 37;

        System.out.println( "Starting check." );
        try
        {
            for( i = 37; i != 0; i = ( i + 37 ) % numItems )
                if( i % 2 == 0 )
                    h1.insert( new MyInteger( i ) );
                else
                    h.insert( new MyInteger( i ) );

            h.merge( h1 );
            for( i = 1; i < numItems; i++ )
                if( ((MyInteger)( h.deleteMin( ) )).intValue( ) != i )
                    System.out.println( "Oops! " + i );
        }
        catch( Overflow e ) { System.out.println( "Unexpected overflow" ); }
        System.out.println( "Check done." );
    }
}
```

7.10 Comparable.java

```
package DataStructures;

/**
 * Protocol for Comparable objects.
 * In Java 1.2, you can remove this file.
 * @author Mark Allen Weiss
 */
public interface Comparable
{
    /**
     * Compare this object with rhs.
     * @param rhs the second Comparable.
     * @return 0 if two objects are equal;
     *     less than zero if this object is smaller;
     *     greater than zero if this object is larger.
     */
    int     compareTo( Comparable rhs );
}
```

7.11 CursorList.java

```
package DataStructures;

// CursorList class
//
// CONSTRUCTION: with no initializer
// Access is via CursorListItr class
//
// ******************PUBLIC OPERATIONS*********************
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// CursorListItr zeroth( )--> Return position to prior to first
// CursorListItr first( ) --> Return first position
// void insert( x, p )    --> Insert x after current iterator position p
// void remove( x )       --> Remove x
// CursorListItr find( x )
//                        --> Return position that views x
// CursorListItr findPrevious( x )
//                        --> Return position prior to x
// ******************ERRORS********************************
// No special errors

/**
 * Linked list implementation of the list
 *    using a header node; cursor version.
 * Access to the list is via CursorListItr.
 * @author Mark Allen Weiss
 * @see CursorListItr
 */
public class CursorList
{
    private static int alloc( )
    {
        int p = cursorSpace[ 0 ].next;
        cursorSpace[ 0 ].next = cursorSpace[ p ].next;
        if( p == 0 )
            throw new OutOfMemoryError( );
        return p;
    }

    private static void free( int p )
    {
        cursorSpace[ p ].element = null;
        cursorSpace[ p ].next = cursorSpace[ 0 ].next;
        cursorSpace[ 0 ].next = p;
    }

    /**
     * Construct the list.
     */
    public CursorList( )
    {
        header = alloc( );
        cursorSpace[ header ].next = 0;
    }

    /**
```

```
     * Test if the list is logically empty.
     * @return true if empty, false otherwise.
     */
    public boolean isEmpty( )
    {
        return cursorSpace[ header ].next == 0;
    }

    /**
     * Make the list logically empty.
     */
    public void makeEmpty( )
    {
        while( !isEmpty( ) )
            remove( first( ).retrieve( ) );
    }


    /**
     * Return an iterator representing the header node.
     */
    public CursorListItr zeroth( )
    {
        return new CursorListItr( header );
    }

    /**
     * Return an iterator representing the first node in the list.
     * This operation is valid for empty lists.
     */
    public CursorListItr first( )
    {
        return new CursorListItr( cursorSpace[ header ].next );
    }

    /**
     * Insert after p.
     * @param x the item to insert.
     * @param p the position prior to the newly inserted item.
     */
    public void insert( Object x, CursorListItr p )
    {
        if( p != null && p.current != 0 )
        {
            int pos = p.current;
            int tmp = alloc( );

            cursorSpace[ tmp ].element = x;
            cursorSpace[ tmp ].next = cursorSpace[ pos ].next;
            cursorSpace[ pos ].next = tmp;
        }
    }

    /**
     * Return iterator corresponding to the first node containing an item.
     * @param x the item to search for.
     * @return an iterator; iterator isPastEnd if item is not found.
```

```
         */
        public CursorListItr find( Object x )
        {
/* 1*/      int itr = cursorSpace[ header ].next;

/* 2*/      while( itr != 0 && !cursorSpace[ itr ].element.equals( x ) )
/* 3*/          itr = cursorSpace[ itr ].next;

/* 4*/      return new CursorListItr( itr );
        }

        /**
         * Return iterator prior to the first node containing an item.
         * @param x the item to search for.
         * @return appropriate iterator if the item is found. Otherwise, the
         * iterator corresponding to the last element in the list is returned.
         */
        public CursorListItr findPrevious( Object x )
        {
/* 1*/      int itr = header;

/* 2*/      while( cursorSpace[ itr ].next != 0 &&
                  !cursorSpace[ cursorSpace[ itr ].next ].element.equals( x ) )
/* 3*/          itr = cursorSpace[ itr ].next;

/* 4*/      return new CursorListItr( itr );
        }

        /**
         * Remove the first occurrence of an item.
         * @param x the item to remove.
         */
        public void remove( Object x )
        {
            CursorListItr p = findPrevious( x );
            int pos = p.current;

            if( cursorSpace[ pos ].next != 0 )
            {
                int tmp = cursorSpace[ pos ].next;
                cursorSpace[ pos ].next = cursorSpace[ tmp ].next;
                free( tmp );
            }
        }

        // Simple print method
        static public void printList( CursorList theList )
        {
            if( theList.isEmpty( ) )
                System.out.print( "Empty list" );
            else
            {
                CursorListItr itr = theList.first( );
                for( ; !itr.isPastEnd( ); itr.advance( ) )
                    System.out.print( itr.retrieve( ) + " " );
            }
        System.out.println( );
    }

    private int header;
    static CursorNode[ ] cursorSpace;

    private static final int SPACE_SIZE = 100;

    static
    {
        cursorSpace = new CursorNode[ SPACE_SIZE ];
        for( int i = 0; i < SPACE_SIZE; i++ )
            cursorSpace[ i ] = new CursorNode( null, i + 1 );
        cursorSpace[ SPACE_SIZE - 1 ].next = 0;
    }

    public static void main( String [ ] args )
    {
        CursorList    theList = new CursorList( );
        CursorListItr theItr;
        int i;

        theItr = theList.zeroth( );
        printList( theList );

        for( i = 0; i < 10; i++ )
        {
            theList.insert( new MyInteger( i ), theItr );
            printList( theList );
            theItr.advance( );
        }

        for( i = 0; i < 10; i += 2 )
            theList.remove( new MyInteger( i ) );

        for( i = 0; i < 10; i++ )
            if( ( i % 2 == 0 ) != ( theList.find( new MyInteger( i ) ).isPastEnd( ) ) )
                System.out.println( "Find fails!" );

        System.out.println( "Finished deletions" );
        printList( theList );
    }

}
```

7.12 CursorListItr.java

```
package DataStructures;

// CursorListItr class; maintains "current position"
//
// CONSTRUCTION: Package friendly only, with a CursorNode
//
// ******************PUBLIC OPERATIONS*********************
// void advance( )        --> Advance
// boolean isPastEnd( )   --> True if at valid position in list
// Object retrieve        --> Return item in current position

/**
 * Linked list implementation of the list iterator
 *    using a header node; cursor version.
 * @author Mark Allen Weiss
 * @see CursorList
 */
public class CursorListItr
{
    /**
     * Construct the list iterator
     * @param theNode any node in the linked list.
     */
    CursorListItr( int theNode )
    {
        current = theNode;
    }

    /**
     * Test if the current position is past the end of the list.
     * @return true if the current position is null-equivalent.
     */
    public boolean isPastEnd( )
    {
        return current == 0;
    }

    /**
     * Return the item stored in the current position.
     * @return the stored item or null if the current position
     * is not in the list.
     */
    public Object retrieve( )
    {
        return isPastEnd( ) ? null : CursorList.cursorSpace[ current ].element;
    }

    /**
     * Advance the current position to the next node in the list.
     * If the current position is null, then do nothing.
     */
    public void advance( )
    {
        if( !isPastEnd( ) )
            current = CursorList.cursorSpace[ current ].next;
    }
    int current;    // Current position
}
```

7.13 CursorNode.java

```
package DataStructures;

// Basic node stored in a linked list -- cursor version
// Note that this class is not accessible outside
// of package DataStructures

class CursorNode
{
        // Constructors
    CursorNode( Object theElement )
    {
        this( theElement, 0 );
    }

    CursorNode( Object theElement, int n )
    {
        element = theElement;
        next    = n;
    }

        // Friendly data; accessible by other package routines
    Object   element;
    int      next;
}
```

7.14 DisjSets.java

```java
package DataStructures;

// DisjSets class
//
// CONSTRUCTION: with int representing initial number of sets
//
// ******************PUBLIC OPERATIONS*********************
// void union( root1, root2 ) --> Merge two sets
// int find( x )              --> Return set containing x
// ******************ERRORS********************************
// No error checking is performed

/**
 * Disjoint set class. (Package friendly so not used accidentally)
 * Does not use union heuristics or path compression.
 * Elements in the set are numbered starting at 0.
 * @author Mark Allen Weiss
 * @see DisjSetsFast
 */
class DisjSets
{
    /**
     * Construct the disjoint sets object.
     * @param numElements the initial number of disjoint sets.
     */
    public DisjSets( int numElements )
    {
        s = new int [ numElements ];
        for( int i = 0; i < s.length; i++ )
            s[ i ] = -1;
    }

    /**
     * Union two disjoint sets.
     * For simplicity, we assume root1 and root2 are distinct
     * and represent set names.
     * @param root1 the root of set 1.
     * @param root2 the root of set 2.
     */
    public void union( int root1, int root2 )
    {
        s[ root2 ] = root1;
    }

    /**
     * Perform a find.
     * Error checks omitted again for simplicity.
     * @param x the element being searched for.
     * @return the set containing x.
     */
    public int find( int x )
    {
        if( s[ x ] < 0 )
            return x;
        else
            return find( s[ x ] );
    }

    private int [ ] s;


    // Test main; all finds on same output line should be identical
    public static void main( String [ ] args )
    {
        int numElements = 128;
        int numInSameSet = 16;

        DisjSets ds = new DisjSets( numElements );
        int set1, set2;

        for( int k = 1; k < numInSameSet; k *= 2 )
        {
            for( int j = 0; j + k < numElements; j += 2 * k )
            {
                set1 = ds.find( j );
                set2 = ds.find( j + k );
                ds.union( set1, set2 );
            }
        }

        for( int i = 0; i < numElements; i++ )
        {
            System.out.print( ds.find( i )+ "*" );
            if( i % numInSameSet == numInSameSet - 1 )
                System.out.println( );
        }
        System.out.println( );
    }
}
```

7.15 DisjSetsFast.java

```
package DataStructures;

// DisjSetsFast class
//
// CONSTRUCTION: with int representing initial number of sets
//
// ******************PUBLIC OPERATIONS*********************
// void union( root1, root2 ) --> Merge two sets
// int find( x )              --> Return set containing x
// ******************ERRORS********************************
// No error checking is performed

/**
 * Disjoint set class, using union by rank
 * and path compression.
 * Elements in the set are numbered starting at 0.
 * @author Mark Allen Weiss
 */
public class DisjSetsFast
{
    /**
     * Construct the disjoint sets object.
     * @param numElements the initial number of disjoint sets.
     */
    public DisjSetsFast( int numElements )
    {
        s = new int [ numElements ];
        for( int i = 0; i < s.length; i++ )
            s[ i ] = -1;
    }

    /**
     * Union two disjoint sets using the height heuristic.
     * For simplicity, we assume root1 and root2 are distinct
     * and represent set names.
     * @param root1 the root of set 1.
     * @param root2 the root of set 2.
     */
    public void union( int root1, int root2 )
    {
        if( s[ root2 ] < s[ root1 ] )  // root2 is deeper
            s[ root1 ] = root2;        // Make root2 new root
        else
        {
            if( s[ root1 ] == s[ root2 ] )
                s[ root1 ]--;          // Update height if same
            s[ root2 ] = root1;        // Make root1 new root
        }
    }

    /**
     * Perform a find with path compression.
     * Error checks omitted again for simplicity.
     * @param x the element being searched for.
     * @return the set containing x.
     */
    public int find( int x )
    {
        if( s[ x ] < 0 )
            return x;
        else
            return s[ x ] = find( s[ x ] );
    }

    private int [ ] s;


    // Test main; all finds on same output line should be identical
    public static void main( String [ ] args )
    {
        int NumElements = 128;
        int NumInSameSet = 16;

        DisjSetsFast ds = new DisjSetsFast( NumElements );
        int set1, set2;

        for( int k = 1; k < NumInSameSet; k *= 2 )
        {
            for( int j = 0; j + k < NumElements; j += 2 * k )
            {
                set1 = ds.find( j );
                set2 = ds.find( j + k );
                ds.union( set1, set2 );
            }
        }

        for( int i = 0; i < NumElements; i++ )
        {
            System.out.print( ds.find( i )+ "*" );
            if( i % NumInSameSet == NumInSameSet - 1 )
                System.out.println( );
        }
        System.out.println( );
    }
}
```

7.16 DSL.java

```java
package DataStructures;

// BinarySearchTree class
//
// CONSTRUCTION: with a value at least as large as all others
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x (unimplemented)
// Comparable find( x )   --> Return item that matches x
// Comparable findMin( )  --> Return smallest item
// Comparable findMax( )  --> Return largest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items

/**
 * Implements a deterministic skip list.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 */
public class DSL
{
    /**
     * Construct the DSL.
     * @param inf the largest Comparable.
     */
    public DSL( Comparable inf )
    {
        infinity = inf;
        bottom = new SkipNode( null );
        bottom.right = bottom.down = bottom;
        tail   = new SkipNode( infinity );
        tail.right = tail;
        header = new SkipNode( infinity, tail, bottom );
    }

    /**
     * Insert into the DSL.
     * @param x the item to insert.
     */
    public void insert( Comparable x )
    {
        SkipNode current = header;

        bottom.element = x;
        while( current != bottom )
        {
            while( current.element.compareTo( x ) < 0 )
                current = current.right;

            // If gap size is 3 or at bottom level and
            // must insert, then promote middle element
            if( current.down.right.right.element.compareTo( current.element ) < 0 )
            {
                current.right = new SkipNode( current.element, current.right,
                                              current.down.right.right );
                current.element = current.down.right.element;
            }
            else
                current = current.down;
        }

        // Raise height of DSL if necessary
        if( header.right != tail )
            header = new SkipNode( infinity, tail, header );
    }

    /**
     * Remove from the DSL. Unimplemented.
     * @param x the item to remove.
     */
    public void remove( Comparable x )
    {
        System.out.println( "Sorry, remove unimplemented" );
    }

    /**
     * Find the smallest item in the DSL.
     * @return smallest item, or null if empty.
     */
    public Comparable findMin( )
    {
        if( isEmpty( ) )
            return null;

        SkipNode current = header;
        while( current.down != bottom )
            current = current.down;

        return elementAt( current );
    }

    /**
     * Find the largest item in the DSL.
     * @return the largest item, or null if empty.
     */
    public Comparable findMax( )
    {
        if( isEmpty( ) )
            return null;

        SkipNode current = header;
        for( ; ; )
             if( current.right.right != tail )
                 current = current.right;
             else if( current.down != bottom )
                 current = current.down;
             else
                 return elementAt( current );
    }

    /**
     * Find an item in the DSL.
```

```
 * @param x the item to search for.
 * @return the matching item, or null if not found.
 */
public Comparable find( Comparable x )
{
    SkipNode current = header;

    bottom.element = x;
    for( ; ; )
        if( x.compareTo( current.element ) < 0 )
            current = current.down;
        else if( x.compareTo( current.element ) > 0 )
            current = current.right;
        else
            return elementAt( current );
}

/**
 * Make the DSL logically empty.
 */
public void makeEmpty( )
{
    header.right = tail;
    header.down = bottom;
}

/**
 * Test if the DSL is logically empty.
 * @return true if empty, false otherwise.
 */
public boolean isEmpty( )
{
    return header.right == tail && header.down == bottom;
}

/**
 * Internal method to get element field.
 * @param t the node.
 * @return the element field, or null if t is null.
 */
private Comparable elementAt( SkipNode t )
{
    return t == bottom ? null : t.element;
}

/**
 * Print the DSL.
 */
private void printList( )
{
    SkipNode current = header;

    while( current.down != bottom )
         ;

    while( current.right != tail )
    {
```

```
                System.out.println( current.element );
                current = current.right;
            }
        }

          /** The DSL header. */
        private SkipNode header;
        private Comparable infinity;
        private SkipNode bottom = null;
        private SkipNode tail   = null;


            // Test program
        public static void main( String [ ] args )
        {
            DSL t = new DSL( new MyInteger( 100000000 ) );
            final int NUMS = 4000;
            final int GAP  =   37;

            System.out.println( "Checking... (no more output means success)" );

            for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
                t.insert( new MyInteger( i ) );

            if( NUMS < 40 )
                t.printList( );
            if( ((MyInteger)(t.findMin( ))).intValue( ) != 1 ||
                ((MyInteger)(t.findMax( ))).intValue( ) != NUMS - 1 )
                System.out.println( "FindMin or FindMax error!" );

            for( int i = 1; i < NUMS; i++ )
                 if( ((MyInteger)(t.find( new MyInteger( i ) ))).intValue( ) != i )
                     System.out.println( "Find error1!" );
            if( t.find( new MyInteger( 0 ) ) != null )
                System.out.println( "Find error2!" );
            if( t.find( new MyInteger( NUMS + 10 ) ) != null )
                System.out.println( "Find error2!" );
    }
}
```

7.17 Hashable.java

```
package DataStructures;

/**
 * Protocol for Hashable objects.
 * @author Mark Allen Weiss
 */
public interface Hashable
{
    /**
     * Compute a hash function for this object.
     * @param tableSize the hash table size.
     * @return (deterministically) a number between
     *     0 and tableSize-1, distributed equitably.
     */
    int hash( int tableSize );
}
```

7.18 HashEntry.java

```
package DataStructures;

// The basic entry stored in ProbingHashTable

class HashEntry
{
    Hashable element;   // the element
    boolean  isActive;  // false is deleted

    public HashEntry( Hashable e )
    {
        this( e, true );
    }

    public HashEntry( Hashable e, boolean i )
    {
        element   = e;
        isActive  = i;
    }
}
```

7.19 LeftHeapNode.java

```
package DataStructures;

// Basic node stored in leftist heaps
// Note that this class is not accessible outside
// of package DataStructures

class LeftHeapNode
{
        // Constructors
    LeftHeapNode( Comparable theElement )
    {
        this( theElement, null, null );
    }

    LeftHeapNode( Comparable theElement, LeftHeapNode lt, LeftHeapNode rt )
    {
        element = theElement;
        left    = lt;
        right   = rt;
        npl     = 0;
    }

        // Friendly data; accessible by other package routines
    Comparable   element;      // The data in the node
    LeftHeapNode left;         // Left child
    LeftHeapNode right;        // Right child
    int          npl;          // null path length
}
```

7.20 LeftistHeap.java

```
package DataStructures;

// LeftistHeap class
//
// CONSTRUCTION: with a negative infinity sentinel
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// Comparable deleteMin( )--> Return and remove smallest item
// Comparable findMin( )  --> Return smallest item
// boolean isEmpty( )     --> Return true if empty; else false
// boolean isFull( )      --> Return false in this implementation
// void makeEmpty( )      --> Remove all items
// void merge( rhs )      --> Absorb rhs into this heap

/**
 * Implements a leftist heap.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 */
public class LeftistHeap
{
    /**
     * Construct the leftist heap.
     */
    public LeftistHeap( )
    {
        root = null;
    }

    /**
     * Merge rhs into the priority queue.
     * rhs becomes empty. rhs must be different from this.
     * @param rhs the other leftist heap.
     */
    public void merge( LeftistHeap rhs )
    {
        if( this == rhs )    // Avoid aliasing problems
            return;

        root = merge( root, rhs.root );
        rhs.root = null;
    }

    /**
     * Internal static method to merge two roots.
     * Deals with deviant cases and calls recursive merge1.
     */
    private static LeftHeapNode merge( LeftHeapNode h1, LeftHeapNode h2 )
    {
        if( h1 == null )
            return h2;
        if( h2 == null )
            return h1;
        if( h1.element.compareTo( h2.element ) < 0 )
            return merge1( h1, h2 );
        else
            return merge1( h2, h1 );
    }

    /**
     * Internal static method to merge two roots.
     * Assumes trees are not empty, and h1's root contains smallest item.
     */
    private static LeftHeapNode merge1( LeftHeapNode h1, LeftHeapNode h2 )
    {
        if( h1.left == null )   // Single node
           h1.left = h2;       // Other fields in h1 already accurate
        else
        {
            h1.right = merge( h1.right, h2 );
            if( h1.left.npl < h1.right.npl )
                swapChildren( h1 );
            h1.npl = h1.right.npl + 1;
        }
        return h1;
    }

    /**
     * Swaps t's two children.
     */
    private static void swapChildren( LeftHeapNode t )
    {
        LeftHeapNode tmp = t.left;
        t.left = t.right;
        t.right = tmp;
    }

    /**
     * Insert into the priority queue, maintaining heap order.
     * @param x the item to insert.
     */
    public void insert( Comparable x )
    {
        root = merge( new LeftHeapNode( x ), root );
    }

    /**
     * Find the smallest item in the priority queue.
     * @return the smallest item, or null, if empty.
     */
    public Comparable findMin( )
    {
        if( isEmpty( ) )
            return null;
        return root.element;
    }

    /**
     * Remove the smallest item from the priority queue.
     * @return the smallest item, or null, if empty.
     */
    public Comparable deleteMin( )
```

```
    {
        if( isEmpty( ) )
            return null;

        Comparable minItem = root.element;
        root = merge( root.left, root.right );

        return minItem;
    }

    /**
     * Test if the priority queue is logically empty.
     * @return true if empty, false otherwise.
     */
    public boolean isEmpty( )
    {
        return root == null;
    }

    /**
     * Test if the priority queue is logically full.
     * @return false in this implementation.
     */
    public boolean isFull( )
    {
        return false;
    }

    /**
     * Make the priority queue logically empty.
     */
    public void makeEmpty( )
    {
        root = null;
    }

    private LeftHeapNode root;    // root

    public static void main( String [ ] args )
    {
        int numItems = 100;
        LeftistHeap h  = new LeftistHeap( );
        LeftistHeap h1 = new LeftistHeap( );
        int i = 37;

            for( i = 37; i != 0; i = ( i + 37 ) % numItems )
                if( i % 2 == 0 )
                    h1.insert( new MyInteger( i ) );
                else
                    h.insert( new MyInteger( i ) );

            h.merge( h1 );
            for( i = 1; i < numItems; i++ )
                if( ((MyInteger)( h.deleteMin( ) )).intValue( ) != i )
                    System.out.println( "Oops! " + i );
    }
}
```

7.21 LinkedList.java

```
package DataStructures;

// LinkedList class
//
// CONSTRUCTION: with no initializer
// Access is via LinkedListItr class
//
// ******************PUBLIC OPERATIONS*********************
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// LinkedListItr zeroth( )--> Return position to prior to first
// LinkedListItr first( ) --> Return first position
// void insert( x, p )    --> Insert x after current iterator position p
// void remove( x )       --> Remove x
// LinkedListItr find( x )
//                        --> Return position that views x
// LinkedListItr findPrevious( x )
//                        --> Return position prior to x
// ******************ERRORS********************************
// No special errors

/**
 * Linked list implementation of the list
 *    using a header node.
 * Access to the list is via LinkedListItr.
 * @author Mark Allen Weiss
 * @see LinkedListItr
 */
public class LinkedList
{
    /**
     * Construct the list
     */
    public LinkedList( )
    {
        header = new ListNode( null );
    }

    /**
     * Test if the list is logically empty.
     * @return true if empty, false otherwise.
     */
    public boolean isEmpty( )
    {
        return header.next == null;
    }

    /**
     * Make the list logically empty.
     */
    public void makeEmpty( )
    {
        header.next = null;
    }
```

```
        /**
         * Return an iterator representing the header node.
         */
        public LinkedListItr zeroth( )
        {
            return new LinkedListItr( header );
        }

        /**
         * Return an iterator representing the first node in the list.
         * This operation is valid for empty lists.
         */
        public LinkedListItr first( )
        {
            return new LinkedListItr( header.next );
        }

        /**
         * Insert after p.
         * @param x the item to insert.
         * @param p the position prior to the newly inserted item.
         */
        public void insert( Object x, LinkedListItr p )
        {
            if( p != null && p.current != null )
                p.current.next = new ListNode( x, p.current.next );
        }

        /**
         * Return iterator corresponding to the first node containing an item.
         * @param x the item to search for.
         * @return an iterator; iterator isPastEnd if item is not found.
         */
        public LinkedListItr find( Object x )
        {
/* 1*/      ListNode itr = header.next;

/* 2*/      while( itr != null && !itr.element.equals( x ) )
/* 3*/          itr = itr.next;

/* 4*/      return new LinkedListItr( itr );
        }

        /**
         * Return iterator prior to the first node containing an item.
         * @param x the item to search for.
         * @return appropriate iterator if the item is found. Otherwise, the
         * iterator corresponding to the last element in the list is returned.
         */
        public LinkedListItr findPrevious( Object x )
        {
/* 1*/      ListNode itr = header;

/* 2*/      while( itr.next != null && !itr.next.element.equals( x ) )
/* 3*/          itr = itr.next;

/* 4*/      return new LinkedListItr( itr );
```

```
}

/**
 * Remove the first occurrence of an item.
 * @param x the item to remove.
 */
public void remove( Object x )
{
    LinkedListItr p = findPrevious( x );

    if( p.current.next != null )
        p.current.next = p.current.next.next;  // Bypass deleted node
}

// Simple print method
public static void printList( LinkedList theList )
{
    if( theList.isEmpty( ) )
        System.out.print( "Empty list" );
    else
    {
        LinkedListItr itr = theList.first( );
        for( ; !itr.isPastEnd( ); itr.advance( ) )
            System.out.print( itr.retrieve( ) + " " );
    }

    System.out.println( );
}

private ListNode header;


public static void main( String [ ] args )
{
    LinkedList    theList = new LinkedList( );
    LinkedListItr theItr;
    int i;

    theItr = theList.zeroth( );
    printList( theList );

    for( i = 0; i < 10; i++ )
    {
        theList.insert( new MyInteger( i ), theItr );
        printList( theList );
        theItr.advance( );
    }

    for( i = 0; i < 10; i += 2 )
        theList.remove( new MyInteger( i ) );

    for( i = 0; i < 10; i++ )
        if( ( i % 2 == 0 ) != ( theList.find( new MyInteger( i ) ).isPastEnd( ) ) )
            System.out.println( "Find fails!" );

    System.out.println( "Finished deletions" );
    printList( theList );
    }
}
```

7.22 LinkedListItr.java

```
package DataStructures;

// LinkedListItr class; maintains "current position"
//
// CONSTRUCTION: Package friendly only, with a ListNode
//
// ******************PUBLIC OPERATIONS*********************
// void advance( )        --> Advance
// boolean isPastEnd( )   --> True if at "null" position in list
// Object retrieve        --> Return item in current position

/**
 * Linked list implementation of the list iterator
 *    using a header node.
 * @author Mark Allen Weiss
 * @see LinkedList
 */
public class LinkedListItr
{
    /**
     * Construct the list iterator
     * @param theNode any node in the linked list.
     */
    LinkedListItr( ListNode theNode )
    {
        current = theNode;
    }

    /**
     * Test if the current position is past the end of the list.
     * @return true if the current position is null.
     */
    public boolean isPastEnd( )
    {
        return current == null;
    }

    /**
     * Return the item stored in the current position.
     * @return the stored item or null if the current position
     * is not in the list.
     */
    public Object retrieve( )
    {
        return isPastEnd( ) ? null : current.element;
    }

    /**
     * Advance the current position to the next node in the list.
     * If the current position is null, then do nothing.
     */
    public void advance( )
    {
        if( !isPastEnd( ) )
            current = current.next;
    }
    ListNode current;    // Current position
}
```

7.23 ListNode.java

```
package DataStructures;

// Basic node stored in a linked list
// Note that this class is not accessible outside
// of package DataStructures

class ListNode
{
        // Constructors
    ListNode( Object theElement )
    {
        this( theElement, null );
    }

    ListNode( Object theElement, ListNode n )
    {
        element = theElement;
        next    = n;
    }

        // Friendly data; accessible by other package routines
    Object   element;
    ListNode next;
}
```

7.24 MyInteger.java

```
package DataStructures;

/**
 * Wrapper class for use with generic data structures.
 * Mimics Integer.
 * In Java 1.2, you can use Integer if Comparable is needed.
 * @author Mark Allen Weiss
 */
public final class MyInteger implements Comparable, Hashable
{
    /**
     * Construct the MyInteger object with initial value 0.
     */
    public MyInteger( )
    {
        this( 0 );
    }

    /**
     * Construct the MyInteger object.
     * @param x the initial value.
     */
    public MyInteger( int x )
    {
        value = x;
    }

    /**
     * Gets the stored int value.
     * @return the stored value.
     */
    public int intValue( )
    {
        return value;
    }

    /**
     * Implements the toString method.
     * @return the String representation.
     */
    public String toString( )
    {
        return Integer.toString( value );
    }

    /**
     * Implements the compareTo method.
     * @param rhs the other MyInteger object.
     * @return 0 if two objects are equal;
     *     less than zero if this object is smaller;
     *     greater than zero if this object is larger.
     * @exception ClassCastException if rhs is not
     *     a MyInteger.
     */
    public int compareTo( Comparable rhs )
    {
        return value < ((MyInteger)rhs).value ? -1 :
               value == ((MyInteger)rhs).value ? 0 : 1;
    }

    /**
     * Implements the equals method.
     * @param rhs the second MyInteger.
     * @return true if the objects are equal, false otherwise.
     * @exception ClassCastException if rhs is not
     *     a MyInteger.
     */
    public boolean equals( Object rhs )
    {
        return rhs != null && value == ((MyInteger)rhs).value;
    }

    /**
     * Implements the hash method.
     * @param tableSize the hash table size.
     * @return a number between 0 and tableSize-1.
     */
    public int hash( int tableSize )
    {
        if( value < 0 )
            return -value % tableSize;
        else
            return value % tableSize;
    }

    private int value;
}
```

7.25 Overflow.java

```java
package DataStructures;

/**
 * Exception class for access in full containers
 * such as stacks, queues, and priority queues.
 * @author Mark Allen Weiss
 */
public class Overflow extends Exception
{
}
```

7.26 PairHeap.java

```
package DataStructures;

// PairHeap class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// PairNode insert( x )   --> Insert x, return position
// Comparable deleteMin( )--> Return and remove smallest item
// Comparable findMin( )  --> Return smallest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// void decreaseKey( PairNode p, newVal )
//                        --> Decrease value in node p

/**
 * Implements a pairing heap.
 * Supports a decreaseKey operation.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 * @see PairNode
 */
public class PairHeap
{
    /**
     * Construct the pairing heap.
     */
    public PairHeap( )
    {
        root = null;
    }

    /**
     * Insert into the priority queue, and return a PairNode
     * that can be used by decreaseKey.
     * Duplicates are allowed.
     * @param x the item to insert.
     * @return the node containing the newly inserted item.
     */
    public PairNode insert( Comparable x )
    {
        PairNode newNode = new PairNode( x );

        if( root == null )
            root = newNode;
        else
            root = compareAndLink( root, newNode );
        return newNode;
    }

    /**
     * Find the smallest item in the priority queue.
     * @return the smallest item, or null if empty.
     */
    public Comparable findMin( )
    {
```

```
    if( isEmpty( ) )
        return null;
    return root.element;
}

/**
 * Remove the smallest item from the priority queue.
 * @return the smallest item, or null if empty.
 */
public Comparable deleteMin( )
{
    if( isEmpty( ) )
        return null;

    Comparable x = findMin( );
    if( root.leftChild == null )
        root = null;
    else
        root = combineSiblings( root.leftChild );

    return x;
}

/**
 * Change the value of the item stored in the pairing heap.
 * Does nothing if newVal is larger than the currently stored value.
 * @param p any node returned by addItem.
 * @param newVal the new value, which must be smaller
 *    than the currently stored value.
 */
public void decreaseKey( PairNode p, Comparable newVal )
{
    if( p.element.compareTo( newVal ) < 0 )
        return;    // newVal cannot be bigger
    p.element = newVal;
    if( p != root )
    {
        if( p.nextSibling != null )
            p.nextSibling.prev = p.prev;
        if( p.prev.leftChild == p )
            p.prev.leftChild = p.nextSibling;
        else
            p.prev.nextSibling = p.nextSibling;

        p.nextSibling = null;
        root = compareAndLink( root, p );
    }
}

/**
 * Test if the priority queue is logically empty.
 * @return true if empty, false otherwise.
 */
public boolean isEmpty( )
{
    return root == null;
}
```

```
/**
 * Make the priority queue logically empty.
 */
public void makeEmpty( )
{
    root = null;
}

private PairNode root;

/**
 * Internal method that is the basic operation to maintain order.
 * Links first and second together to satisfy heap order.
 * @param first root of tree 1, which may not be null.
 *    first.nextSibling MUST be null on entry.
 * @param second root of tree 2, which may be null.
 * @return result of the tree merge.
 */
private PairNode compareAndLink( PairNode first, PairNode second )
{
    if( second == null )
        return first;

    if( second.element.compareTo( first.element ) < 0 )
    {
        // Attach first as leftmost child of second
        second.prev = first.prev;
        first.prev = second;
        first.nextSibling = second.leftChild;
        if( first.nextSibling != null )
            first.nextSibling.prev = first;
        second.leftChild = first;
        return second;
    }
    else
    {
        // Attach second as leftmost child of first
        second.prev = first;
        first.nextSibling = second.nextSibling;
        if( first.nextSibling != null )
            first.nextSibling.prev = first;
        second.nextSibling = first.leftChild;
        if( second.nextSibling != null )
            second.nextSibling.prev = second;
        first.leftChild = second;
        return first;
    }
}

private PairNode [ ] doubleIfFull( PairNode [ ] array, int index )
{
    if( index == array.length )
    {
        PairNode [ ] oldArray = array;

        array = new PairNode[ index * 2 ];
        for( int i = 0; i < index; i++ )
            array[ i ] = oldArray[ i ];
    }
    return array;
}

    // The tree array for combineSiblings
private PairNode [ ] treeArray = new PairNode[ 5 ];

/**
 * Internal method that implements two-pass merging.
 * @param firstSibling the root of the conglomerate;
 *     assumed not null.
 */
private PairNode combineSiblings( PairNode firstSibling )
{
    if( firstSibling.nextSibling == null )
        return firstSibling;

        // Store the subtrees in an array
    int numSiblings = 0;
    for( ; firstSibling != null; numSiblings++ )
    {
        treeArray = doubleIfFull( treeArray, numSiblings );
        treeArray[ numSiblings ] = firstSibling;
        firstSibling.prev.nextSibling = null;  // break links
        firstSibling = firstSibling.nextSibling;
    }
    treeArray = doubleIfFull( treeArray, numSiblings );
    treeArray[ numSiblings ] = null;

        // Combine subtrees two at a time, going left to right
    int i = 0;
    for( ; i + 1 < numSiblings; i += 2 )
        treeArray[ i ] = compareAndLink( treeArray[ i ], treeArray[ i + 1 ] );

    int j = i - 2;

        // j has the result of last compareAndLink.
        // If an odd number of trees, get the last one.
    if( j == numSiblings - 3 )
        treeArray[ j ] = compareAndLink( treeArray[ j ], treeArray[ j + 2 ] );

        // Now go right to left, merging last tree with
        // next to last. The result becomes the new last.
    for( ; j >= 2; j -= 2 )
        treeArray[ j - 2 ] = compareAndLink( treeArray[ j - 2 ], treeArray[ j ] );

    return treeArray[ 0 ];
}

    // Test program
public static void main( String [ ] args )
{
    PairHeap h = new PairHeap( );
    int numItems = 10000;
    int i = 37;
```

```
        int j;

        System.out.println( "Checking; no bad output is good" );
        for( i = 37; i != 0; i = ( i + 37 ) % numItems )
           h.insert( new MyInteger( i ) );
        for( i = 1; i < numItems; i++ )
            if( ((MyInteger)( h.deleteMin( ) )).intValue( ) != i )
                System.out.println( "Oops! " + i );

        PairNode [ ] p = new PairNode[ numItems ];
        for( i = 0, j = numItems / 2; i < numItems; i++, j =(j+71)%numItems )
            p[ j ] = h.insert( new MyInteger( j + numItems ) );
        for( i = 0, j = numItems / 2; i < numItems; i++, j =(j+53)%numItems )
            h.decreaseKey( p[ j ], new MyInteger(
                     ((MyInteger)p[ j ].element).intValue( ) - numItems ) );
        i = -1;
        while( !h.isEmpty( ) )
            if( ((MyInteger)( h.deleteMin( ) )).intValue( ) != ++i )
                System.out.println( "Oops! " + i + " " );
        System.out.println( "Check completed" );
    }
}
```

7.27 PairNode.java

```java
package DataStructures;

/**
 * Public class for use with PairHeap. It is public
 * only to allow references to be sent to decreaseKey.
 * It has no public methods or members.
 * @author Mark Allen Weiss
 * @see PairHeap
 */
public class PairNode
{
    /**
     * Construct the PairNode.
     * @param theElement the value stored in the node.
     */
    PairNode( Comparable theElement )
    {
        element     = theElement;
        leftChild   = null;
        nextSibling = null;
        prev        = null;
    }

        // Friendly data; accessible by other package routines
    Comparable element;
    PairNode   leftChild;
    PairNode   nextSibling;
    PairNode   prev;
}
```

7.28 QuadraticProbingHashTable.java

```
package DataStructures;

// QuadraticProbingHashTable abstract class
//
// CONSTRUCTION: with an approximate initial size or a default.
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x
// Hashable find( x )     --> Return item that matches x
// void makeEmpty( )      --> Remove all items
// int hash( String str, int tableSize )
//                        --> Static method to hash strings

/**
 * Probing table implementation of hash tables.
 * Note that all "matching" is based on the equals method.
 * @author Mark Allen Weiss
 */
public class QuadraticProbingHashTable
{
    /**
     * Construct the hash table.
     */
    public QuadraticProbingHashTable( )
    {
        this( DEFAULT_TABLE_SIZE );
    }

    /**
     * Construct the hash table.
     * @param size the approximate initial size.
     */
    public QuadraticProbingHashTable( int size )
    {
        allocateArray( size );
        makeEmpty( );
    }

    /**
     * Insert into the hash table. If the item is
     * already present, do nothing.
     * @param x the item to insert.
     */
    public void insert( Hashable x )
    {
            // Insert x as active
        int currentPos = findPos( x );
        if( isActive( currentPos ) )
            return;

        array[ currentPos ] = new HashEntry( x, true );

            // Rehash; see Section 5.5
        if( ++currentSize > array.length / 2 )
            rehash( );
        }

        /**
         * Expand the hash table.
         */
        private void rehash( )
        {
            HashEntry [ ] oldArray = array;

                // Create a new double-sized, empty table
            allocateArray( nextPrime( 2 * oldArray.length ) );
            currentSize = 0;

                // Copy table over
            for( int i = 0; i < oldArray.length; i++ )
                if( oldArray[ i ] != null && oldArray[ i ].isActive )
                    insert( oldArray[ i ].element );

            return;
        }

        /**
         * Method that performs quadratic probing resolution.
         * @param x the item to search for.
         * @return the position where the search terminates.
         */
        private int findPos( Hashable x )
        {
/* 1*/      int collisionNum = 0;
/* 2*/      int currentPos = x.hash( array.length );

/* 3*/      while( array[ currentPos ] != null &&
                    !array[ currentPos ].element.equals( x ) )
            {
/* 4*/          currentPos += 2 * ++collisionNum - 1;  // Compute ith probe
/* 5*/          if( currentPos >= array.length )       // Implement the mod
/* 6*/              currentPos -= array.length;
            }

/* 7*/      return currentPos;
        }

        /**
         * Remove from the hash table.
         * @param x the item to remove.
         */
        public void remove( Hashable x )
        {
            int currentPos = findPos( x );
            if( isActive( currentPos ) )
                array[ currentPos ].isActive = false;
        }

        /**
         * Find an item in the hash table.
         * @param x the item to search for.
         * @return the matching item.
```

```
     */
    public Hashable find( Hashable x )
    {
        int currentPos = findPos( x );
return isActive( currentPos ) ? array[ currentPos ].element : null;
    }

    /**
     * Return true if currentPos exists and is active.
     * @param currentPos the result of a call to findPos.
     * @return true if currentPos is active.
     */
    private boolean isActive( int currentPos )
    {
        return array[ currentPos ] != null && array[ currentPos ].isActive;
    }

    /**
     * Make the hash table logically empty.
     */
    public void makeEmpty( )
    {
        currentSize = 0;
        for( int i = 0; i < array.length; i++ )
            array[ i ] = null;
    }

    /**
     * A hash routine for String objects.
     * @param key the String to hash.
     * @param tableSize the size of the hash table.
     * @return the hash value.
     */
    public static int hash( String key, int tableSize )
    {
        int hashVal = 0;

        for( int i = 0; i < key.length( ); i++ )
            hashVal = 37 * hashVal + key.charAt( i );

        hashVal %= tableSize;
        if( hashVal < 0 )
            hashVal += tableSize;

        return hashVal;
    }

    private static final int DEFAULT_TABLE_SIZE = 11;

        /** The array of elements. */
    private HashEntry [ ] array;   // The array of elements
    private int currentSize;       // The number of occupied cells

    /**
     * Internal method to allocate array.
     * @param arraySize the size of the array.
     */
```

```
private void allocateArray( int arraySize )
{
    array = new HashEntry[ arraySize ];
}

/**
 * Internal method to find a prime number at least as large as n.
 * @param n the starting number (must be positive).
 * @return a prime number larger than or equal to n.
 */
private static int nextPrime( int n )
{
    if( n % 2 == 0 )
        n++;

    for( ; !isPrime( n ); n += 2 )
        ;

    return n;
}

/**
 * Internal method to test if a number is prime.
 * Not an efficient algorithm.
 * @param n the number to test.
 * @return the result of the test.
 */
private static boolean isPrime( int n )
{
    if( n == 2 || n == 3 )
        return true;

    if( n == 1 || n % 2 == 0 )
        return false;

    for( int i = 3; i * i <= n; i += 2 )
        if( n % i == 0 )
            return false;

    return true;
}


// Simple main
public static void main( String [ ] args )
{
    QuadraticProbingHashTable H = new QuadraticProbingHashTable( );

    final int NUMS = 4000;
    final int GAP  =   37;

    System.out.println( "Checking... (no more output means success)" );


    for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
        H.insert( new MyInteger( i ) );
    for( int i = 1; i < NUMS; i+= 2 )
```

```
            H.remove( new MyInteger( i ) );

        for( int i = 2; i < NUMS; i+=2 )
            if( ((MyInteger)(H.find( new MyInteger( i ) ))).intValue( ) != i )
                System.out.println( "Find fails " + i );

        for( int i = 1; i < NUMS; i+=2 )
        {
            if( H.find( new MyInteger( i ) ) != null )
                System.out.println( "OOPS!!! " +  i  );
        }
    }

}
```

7.29 QueueAr.java

```
package DataStructures;

// QueueAr class
//

// CONSTRUCTION: with or without a capacity; default is 10
//
// ******************PUBLIC OPERATIONS*********************
// void enqueue( x )      --> Insert x
// Object getFront( )     --> Return least recently inserted item
// Object dequeue( )      --> Return and remove least recent item
// boolean isEmpty( )     --> Return true if empty; else false
// boolean isFull( )      --> Return true if capacity reached
// void makeEmpty( )      --> Remove all items
// ******************ERRORS********************************
// Overflow thrown for enqueue on full queue

/**
 * Array-based implementation of the queue.
 * @author Mark Allen Weiss
 */
public class QueueAr
{
    /**
     * Construct the queue.
     */
    public QueueAr( )
    {
        this( DEFAULT_CAPACITY );
    }

    /**
     * Construct the queue.
     */
    public QueueAr( int capacity )
    {
        theArray = new Object[ capacity ];
        makeEmpty( );
    }

    /**
     * Test if the queue is logically empty.
     * @return true if empty, false otherwise.
     */
    public boolean isEmpty( )
    {
        return currentSize == 0;
    }

    /**
     * Test if the queue is logically full.
     * @return true if full, false otherwise.
     */
    public boolean isFull( )
    {
        return currentSize == theArray.length;
}

/**
 * Make the queue logically empty.
 */
public void makeEmpty( )
{
    currentSize = 0;
    front = 0;
    back = -1;
}

/**
 * Get the least recently inserted item in the queue.
 * Does not alter the queue.
 * @return the least recently inserted item in the queue, or null, if empty.
 */
public Object getFront( )
{
    if( isEmpty( ) )
        return null;
    return theArray[ front ];
}

/**
 * Return and remove the least recently inserted item from the queue.
 * @return the least recently inserted item in the queue, or null, if empty.
 */
public Object dequeue( )
{
    if( isEmpty( ) )
        return null;
    currentSize--;

    Object frontItem = theArray[ front ];
    theArray[ front ] = null;
    front = increment( front );
    return frontItem;
}

/**
 * Insert a new item into the queue.
 * @param x the item to insert.
 * @exception Overflow if queue is full.
 */
public void enqueue( Object x ) throws Overflow
{
    if( isFull( ) )
        throw new Overflow( );
    back = increment( back );
    theArray[ back ] = x;
    currentSize++;
}

/**
 * Internal method to increment with wraparound.
 * @param x any index in theArray's range.
```

```
     * @return x+1, or 0, if x is at the end of theArray.
     */
    private int increment( int x )
    {
        if( ++x == theArray.length )
            x = 0;
        return x;
    }

    private Object [ ] theArray;
    private int        currentSize;
    private int        front;
    private int        back;

    static final int DEFAULT_CAPACITY = 10;


    public static void main( String [ ] args )
    {
        QueueAr q = new QueueAr( );

        try
        {
            for( int i = 0; i < 10; i++ )
                q.enqueue( new MyInteger( i ) );
        }
        catch( Overflow e ) { System.out.println( "Unexpected overflow" ); }

        while( !q.isEmpty( ) )
            System.out.println( q.dequeue( ) );
    }
}
```

7.30 Random.java

```
package DataStructures;

// Random class
//
// CONSTRUCTION: with (a) no initializer or (b) an integer
//     that specifies the initial state of the generator
//
// ******************PUBLIC OPERATIONS*********************
//     Return a random number according to some distribution:
// int randomInt( )                     --> Uniform, 1 to 2^31-1
// int random0_1( )                     --> Uniform, 0 to 1
// int randomInt( int low, int high )   --> Uniform low..high
// long randomLong( long low, long high ) --> Uniform low..high
// void permute( Object [ ] a )         --> Randomly permutate

/**
 * Random number class, using a 31-bit
 * linear congruential generator.
 * Note that java.util contains a class Random,
 * so watch out for name conflicts.
 * @author Mark Allen Weiss
 */
public class Random
{
    private static final int A = 48271;
    private static final int M = 2147483647;
    private static final int Q = M / A;
    private static final int R = M % A;

    /**
     * Construct this Random object with
     * initial state obtained from system clock.
     */
    public Random( )
    {
        this( (int) ( System.currentTimeMillis( ) % Integer.MAX_VALUE ) );
    }

    /**
     * Construct this Random object with
     * specified initial state.
     * @param initialValue the initial state.
     */
    public Random( int initialValue )
    {
        if( initialValue < 0 )
            initialValue += M;

        state = initialValue;
        if( state == 0 )
            state = 1;
    }

    /**
     * Return a pseudorandom int, and change the
     * internal state.
 * @return the pseudorandom int.
 */
public int randomInt( )
{
    int tmpState = A * ( state % Q ) - R * ( state / Q );
    if( tmpState >= 0 )
        state = tmpState;
    else
        state = tmpState + M;

    return state;
}

/**
 * Return a pseudorandom int, and change the
 * internal state. DOES NOT WORK.
 * @return the pseudorandom int.
 */
public int randomIntWRONG( )
{
    return state = ( A * state ) % M;
}

/**
 * Return a pseudorandom double in the open range 0..1
 * and change the internal state.
 * @return the pseudorandom double.
 */
public double random0_1( )
{
    return (double) randomInt( ) / M;
}

/**
 * Return an int in the closed range [low,high], and
 * change the internal state.
 * @param low the minimum value returned.
 * @param high the maximum value returned.
 * @return the pseudorandom int.
 */
public int randomInt( int low, int high )
{
    double partitionSize = (double) M / ( high - low + 1 );

    return (int) ( randomInt( ) / partitionSize ) + low;
}

/**
 * Return an long in the closed range [low,high], and
 * change the internal state.
 * @param low the minimum value returned.
 * @param high the maximum value returned.
 * @return the pseudorandom long.
 */
public long randomLong( long low, long high )
{
    long longVal =  ( (long) randomInt( ) << 31 ) + randomInt( );
```

```
        long longM =  ( (long) M << 31 ) + M;

        double partitionSize = (double) longM / ( high - low + 1 );
        return (long) ( longVal / partitionSize ) + low;
    }
    /**
     * Randomly rearrange an array.
     * The random numbers used depend on the time and day.
     * @param a the array.
     */
    public static final void permute( Object [ ] a )
    {
        Random r = new Random( );

        for( int j = 1; j < a.length; j++ )
            Sort.swapReferences( a, j, r.randomInt( 0, j ) );
    }

    private int state;

        // Test program
    public static void main( String [ ] args )
    {
        Random r = new Random( 1 );

        for( int i = 0; i < 20; i++ )
            System.out.println( r.randomInt( ) );
    }
}
```

7.31 RedBlackNode.java

```
package DataStructures;

// Basic node stored in red-black trees
// Note that this class is not accessible outside
// of package DataStructures

class RedBlackNode
{
        // Constructors
    RedBlackNode( Comparable theElement )
    {
        this( theElement, null, null );
    }

    RedBlackNode( Comparable theElement, RedBlackNode lt, RedBlackNode rt )
    {
        element  = theElement;
        left     = lt;
        right    = rt;
        color    = RedBlackTree.BLACK;
    }

        // Friendly data; accessible by other package routines
    Comparable   element;    // The data in the node
    RedBlackNode left;       // Left child
    RedBlackNode right;      // Right child
    int          color;      // Color
}
```

7.32 RedBlackTree.java

```
package DataStructures;

// RedBlackTree class
//
// CONSTRUCTION: with a negative infinity sentinel
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x (unimplemented)
// Comparable find( x )   --> Return item that matches x
// Comparable findMin( )  --> Return smallest item
// Comparable findMax( )  --> Return largest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// void printTree( )      --> Print tree in sorted order

/**
 * Implements a red-black tree.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 */
public class RedBlackTree
{
    /**
     * Construct the tree.
     * @param negInf a value less than or equal to all others.
     */
    public RedBlackTree( Comparable negInf )
    {
        header      = new RedBlackNode( negInf );
        header.left = header.right = nullNode;
    }

    /**
     * Insert into the tree. Does nothing if item already present.
     * @param item the item to insert.
     */
    public void insert( Comparable item )
    {
        current = parent = grand = header;
        nullNode.element = item;

        while( current.element.compareTo( item ) != 0 )
        {
            great = grand; grand = parent; parent = current;
            current = item.compareTo( current.element ) < 0 ?
                         current.left : current.right;

                // Check if two red children; fix if so
            if( current.left.color == RED && current.right.color == RED )
                 handleReorient( item );
        }

            // Insertion fails if already present
        if( current != nullNode )
            return;
        current = new RedBlackNode( item, nullNode, nullNode );

            // Attach to parent
        if( item.compareTo( parent.element ) < 0 )
            parent.left = current;
        else
            parent.right = current;
        handleReorient( item );
    }

    /**
     * Remove from the tree.
     * Not implemented in this version.
     * @param x the item to remove.
     */
    public void remove( Comparable x )
    {
        System.out.println( "Remove is not implemented" );
    }

    /**
     * Find the smallest item  the tree.
     * @return the smallest item or null if empty.
     */
    public Comparable findMin( )
    {
        if( isEmpty( ) )
            return null;

        RedBlackNode itr = header.right;

        while( itr.left != nullNode )
            itr = itr.left;

        return itr.element;
    }

    /**
     * Find the largest item in the tree.
     * @return the largest item or null if empty.
     */
    public Comparable findMax( )
    {
        if( isEmpty( ) )
            return null;

        RedBlackNode itr = header.right;

        while( itr.right != nullNode )
            itr = itr.right;

        return itr.element;
    }

    /**
     * Find an item in the tree.
     * @param x the item to search for.
     * @return the matching item or null if not found.
     */
    public Comparable find( Comparable x )
    {
        nullNode.element = x;
        current = header.right;

        for( ; ; )
        {
            if( x.compareTo( current.element ) < 0 )
                current = current.left;
            else if( x.compareTo( current.element ) > 0 )
                current = current.right;
            else if( current != nullNode )
                return current.element;
            else
                return null;
        }
    }

    /**
     * Make the tree logically empty.
     */
    public void makeEmpty( )
    {
        header.right = nullNode;
    }

    /**
     * Test if the tree is logically empty.
     * @return true if empty, false otherwise.
     */
    public boolean isEmpty( )
    {
        return header.right == nullNode;
    }

    /**
     * Print the tree contents in sorted order.
     */
    public void printTree( )
    {
        if( isEmpty( ) )
            System.out.println( "Empty tree" );
        else
            printTree( header.right );
    }

    /**
     * Internal method to print a subtree in sorted order.
     * @param t the node that roots the tree.
     */
    private void printTree( RedBlackNode t )
    {
        if( t != nullNode )
        {
            printTree( t.left );
            System.out.println( t.element );
            printTree( t.right );
        }
    }

    /**
     * Internal routine that is called during an insertion
     * if a node has two red children. Performs flip and rotations.
     * @param item the item being inserted.
     */
    private void handleReorient( Comparable item )
    {
            // Do the color flip
        current.color = RED;
        current.left.color = BLACK;
        current.right.color = BLACK;

        if( parent.color == RED )   // Have to rotate
        {
            grand.color = RED;
            if( ( item.compareTo( grand.element ) < 0 ) !=
                ( item.compareTo( parent.element ) < 0 ) )
                parent = rotate( item, grand );  // Start dbl rotate
            current = rotate( item, great );
            current.color = BLACK;
        }
        header.right.color = BLACK; // Make root black
    }

    /**
     * Internal routine that performs a single or double rotation.
     * Because the result is attached to the parent, there are four cases.
     * Called by handleReorient.
     * @param item the item in handleReorient.
     * @param parent the parent of the root of the rotated subtree.
     * @return the root of the rotated subtree.
     */
    private RedBlackNode rotate( Comparable item, RedBlackNode parent )
    {
        if( item.compareTo( parent.element ) < 0 )
            return parent.left = item.compareTo( parent.left.element ) < 0 ?
                rotateWithLeftChild( parent.left )  :  // LL
                rotateWithRightChild( parent.left ) ;  // LR
        else
            return parent.right = item.compareTo( parent.right.element ) < 0 ?
                rotateWithLeftChild( parent.right ) :  // RL
                rotateWithRightChild( parent.right );  // RR
    }

    /**
     * Rotate binary tree node with left child.
     */
    static RedBlackNode rotateWithLeftChild( RedBlackNode k2 )
    {
        RedBlackNode k1 = k2.left;
        k2.left = k1.right;
        k1.right = k2;
```

```
        return k1;
    }

    /**
     * Rotate binary tree node with right child.
     */
    static RedBlackNode rotateWithRightChild( RedBlackNode k1 )
    {
        RedBlackNode k2 = k1.right;
        k1.right = k2.left;
        k2.left = k1;
        return k2;
    }

    private RedBlackNode header;
    private static RedBlackNode nullNode;
        static         // Static initializer for nullNode
        {
            nullNode = new RedBlackNode( null );
            nullNode.left = nullNode.right = nullNode;
        }

    static final int BLACK = 1;    // Black must be 1
    static final int RED   = 0;

        // Used in insert routine and its helpers
    private static RedBlackNode current;
    private static RedBlackNode parent;
    private static RedBlackNode grand;
    private static RedBlackNode great;


        // Test program
    public static void main( String [ ] args )
    {
        RedBlackTree t = new RedBlackTree( new MyInteger( Integer.MIN_VALUE ) );
        final int NUMS = 40000;
        final int GAP  =   307;

        System.out.println( "Checking... (no more output means success)" );

        for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
            t.insert( new MyInteger( i ) );

        if( NUMS < 40 )
            t.printTree( );
        if( ((MyInteger)(t.findMin( ))).intValue( ) != 1 ||
            ((MyInteger)(t.findMax( ))).intValue( ) != NUMS - 1 )
            System.out.println( "FindMin or FindMax error!" );

        for( int i = 1; i < NUMS; i++ )
             if( ((MyInteger)(t.find( new MyInteger( i ) ))).intValue( ) != i )
                 System.out.println( "Find error1!" );
    }
}
```

7.33 Rotations.java

```
package DataStructures;

final class Rotations
{
    /**
     * Rotate binary tree node with left child.
     * For AVL trees, this is a single rotation for case 1.
     */
    static BinaryNode withLeftChild( BinaryNode k2 )
    {
        BinaryNode k1 = k2.left;
        k2.left = k1.right;
        k1.right = k2;
        return k1;
    }

    /**
     * Rotate binary tree node with right child.
     * For AVL trees, this is a single rotation for case 4.
     */
    static BinaryNode withRightChild( BinaryNode k1 )
    {
        BinaryNode k2 = k1.right;
        k1.right = k2.left;
        k2.left = k1;
        return k2;
    }

    /**
     * Double rotate binary tree node: first left child
     * with its right child; then node k3 with new left child.
     * For AVL trees, this is a double rotation for case 2.
     */
    static BinaryNode doubleWithLeftChild( BinaryNode k3 )
    {
        k3.left = withRightChild( k3.left );
        return withLeftChild( k3 );
    }

    /**
     * Double rotate binary tree node: first right child
     * with its left child; then node k1 with new right child.
     * For AVL trees, this is a double rotation for case 3.
     */
    static BinaryNode doubleWithRightChild( BinaryNode k1 )
    {
        k1.right = withLeftChild( k1.right );
        return withRightChild( k1 );
    }
}
```

7.34 SeparateChainingHashTable.java

```
package DataStructures;

// SeparateChainingHashTable class
//
// CONSTRUCTION: with an approximate initial size or default of 101
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x
// Hashable find( x )     --> Return item that matches x
// void makeEmpty( )      --> Remove all items
// int hash( String str, int tableSize )
//                        --> Static method to hash strings
// ******************ERRORS********************************
// insert overrides previous value if duplicate; not an error

/**
 * Separate chaining table implementation of hash tables.
 * Note that all "matching" is based on the equals method.
 * @author Mark Allen Weiss
 */
public class SeparateChainingHashTable
{
    /**
     * Construct the hash table.
     */
    public SeparateChainingHashTable( )
    {
        this( DEFAULT_TABLE_SIZE );
    }

    /**
     * Construct the hash table.
     * @param size approximate table size.
     */
    public SeparateChainingHashTable( int size )
    {
        theLists = new LinkedList[ nextPrime( size ) ];
        for( int i = 0; i < theLists.length; i++ )
            theLists[ i ] = new LinkedList( );
    }

    /**
     * Insert into the hash table. If the item is
     * already present, then do nothing.
     * @param x the item to insert.
     */
    public void insert( Hashable x )
    {
        LinkedList whichList = theLists[ x.hash( theLists.length ) ];
        LinkedListItr itr = whichList.find( x );

        if( itr.isPastEnd( ) )
            whichList.insert( x, whichList.zeroth( ) );
    }
```

```
/**
 * Remove from the hash table.
 * @param x the item to remove.
 */
public void remove( Hashable x )
{
   theLists[ x.hash( theLists.length ) ].remove( x );
}

/**
 * Find an item in the hash table.
 * @param x the item to search for.
 * @return the matching item, or null if not found.
 */
public Hashable find( Hashable x )
{
    return (Hashable)theLists[ x.hash( theLists.length ) ].find( x ).retrieve( );
}

/**
 * Make the hash table logically empty.
 */
public void makeEmpty( )
{
    for( int i = 0; i < theLists.length; i++ )
        theLists[ i ].makeEmpty( );
}

/**
 * A hash routine for String objects.
 * @param key the String to hash.
 * @param tableSize the size of the hash table.
 * @return the hash value.
 */
public static int hash( String key, int tableSize )
{
    int hashVal = 0;

    for( int i = 0; i < key.length( ); i++ )
        hashVal = 37 * hashVal + key.charAt( i );

    hashVal %= tableSize;
    if( hashVal < 0 )
        hashVal += tableSize;

    return hashVal;
}

private static final int DEFAULT_TABLE_SIZE = 101;

    /** The array of Lists. */
private LinkedList [ ] theLists;

/**
 * Internal method to find a prime number at least as large as n.
 * @param n the starting number (must be positive).
 * @return a prime number larger than or equal to n.
 */
private static int nextPrime( int n )
{
    if( n % 2 == 0 )
        n++;

    for( ; !isPrime( n ); n += 2 )
        ;

    return n;
}

/**
 * Internal method to test if a number is prime.
 * Not an efficient algorithm.
 * @param n the number to test.
 * @return the result of the test.
 */
private static boolean isPrime( int n )
{
    if( n == 2 || n == 3 )
        return true;

    if( n == 1 || n % 2 == 0 )
        return false;

    for( int i = 3; i * i <= n; i += 2 )
        if( n % i == 0 )
            return false;

    return true;
}


    // Simple main
public static void main( String [ ] args )
{
    SeparateChainingHashTable H = new SeparateChainingHashTable( );

    final int NUMS = 4000;
    final int GAP  =   37;

    System.out.println( "Checking... (no more output means success)" );

    for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
        H.insert( new MyInteger( i ) );
    for( int i = 1; i < NUMS; i+= 2 )
        H.remove( new MyInteger( i ) );

    for( int i = 2; i < NUMS; i+=2 )
        if( ((MyInteger)(H.find( new MyInteger( i ) ))).intValue( ) != i )
            System.out.println( "Find fails " + i );

    for( int i = 1; i < NUMS; i+=2 )
    {
        if( H.find( new MyInteger( i ) ) != null )
            System.out.println( "OOPS!!! " +  i  );
            }
        }

    }
```

7.35 SkipNode.java

```
package DataStructures;

// Basic node stored in skip lists
// Note that this class is not accessible outside
// of package DataStructures

class SkipNode
{
        // Constructors
    SkipNode( Comparable theElement )
    {
        this( theElement, null, null );
    }

    SkipNode( Comparable theElement, SkipNode rt, SkipNode dt )
    {
        element  = theElement;
        right    = rt;
        down     = dt;
    }

        // Friendly data; accessible by other package routines
    Comparable element;      // The data in the node
    SkipNode   right;        // Right link
    SkipNode   down;         // Down link
}
```

7.36 Sort.java

```
    package DataStructures;

    /**
     * A class that contains several sorting routines,
     * implemented as static methods.
     * Arrays are rearranged with smallest item first,
     * using compareTo.
     * @author Mark Allen Weiss
     */
    public final class Sort
    {
        /**
         * Simple insertion sort.
         * @param a an array of Comparable items.
         */
        public static void insertionSort( Comparable [ ] a )
        {
            int j;

/* 1*/      for( int p = 1; p < a.length; p++ )
            {
/* 2*/          Comparable tmp = a[ p ];
/* 3*/          for( j = p; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
/* 4*/              a[ j ] = a[ j - 1 ];
/* 5*/          a[ j ] = tmp;
            }
        }

        /**
         * Shellsort, using Shell's (poor) increments.
         * @param a an array of Comparable items.
         */
        public static void shellsort( Comparable [ ] a )
        {
            int j;

/* 1*/      for( int gap = a.length / 2; gap > 0; gap /= 2 )
/* 2*/          for( int i = gap; i < a.length; i++ )
                {
/* 3*/              Comparable tmp = a[ i ];
/* 4*/              for( j = i; j >= gap &&
                                tmp.compareTo( a[ j - gap ] ) < 0; j -= gap )
/* 5*/                  a[ j ] = a[ j - gap ];
/* 6*/              a[ j ] = tmp;
                }
        }


        /**
         * Standard heapsort.
         * @param a an array of Comparable items.
         */
        public static void heapsort( Comparable [ ] a )
        {
/* 1*/      for( int i = a.length / 2; i >= 0; i-- )  /* buildHeap */
/* 2*/          percDown( a, i, a.length );
/* 3*/      for( int i = a.length - 1; i > 0; i-- )
            {
/* 4*/          swapReferences( a, 0, i );            /* deleteMax */
/* 5*/          percDown( a, 0, i );
            }
        }

        /**
         * Internal method for heapsort.
         * @param i the index of an item in the heap.
         * @return the index of the left child.
         */
        private static int leftChild( int i )
        {
            return 2 * i + 1;
        }

        /**
         * Internal method for heapsort that is used in
         * deleteMax and buildHeap.
         * @param a an array of Comparable items.
         * @index i the position from which to percolate down.
         * @int n the logical size of the binary heap.
         */
        private static void percDown( Comparable [ ] a, int i, int n )
        {
            int child;
            Comparable tmp;

/* 1*/      for( tmp = a[ i ]; leftChild( i ) < n; i = child )
            {
/* 2*/          child = leftChild( i );
/* 3*/          if( child != n - 1 && a[ child ].compareTo( a[ child + 1 ] ) < 0 )
/* 4*/              child++;
/* 5*/          if( tmp.compareTo( a[ child ] ) < 0 )
/* 6*/              a[ i ] = a[ child ];
                else
/* 7*/              break;
            }
/* 8*/      a[ i ] = tmp;
        }

        /**
         * Mergesort algorithm.
         * @param a an array of Comparable items.
         */
        public static void mergeSort( Comparable [ ] a )
        {
            Comparable [ ] tmpArray = new Comparable[ a.length ];

            mergeSort( a, tmpArray, 0, a.length - 1 );
        }

        /**
         * Internal method that makes recursive calls.
         * @param a an array of Comparable items.
         * @param tmpArray an array to place the merged result.
```

```
 * @param left the left-most index of the subarray.
 * @param right the right-most index of the subarray.
 */
private static void mergeSort( Comparable [ ] a, Comparable [ ] tmpArray,
           int left, int right )
{
    if( left < right )
    {
        int center = ( left + right ) / 2;
        mergeSort( a, tmpArray, left, center );
        mergeSort( a, tmpArray, center + 1, right );
        merge( a, tmpArray, left, center + 1, right );
    }
}

/**
 * Internal method that merges two sorted halves of a subarray.
 * @param a an array of Comparable items.
 * @param tmpArray an array to place the merged result.
 * @param leftPos the left-most index of the subarray.
 * @param rightPos the index of the start of the second half.
 * @param rightEnd the right-most index of the subarray.
 */
private static void merge( Comparable [ ] a, Comparable [ ] tmpArray,
       int leftPos, int rightPos, int rightEnd )
{
    int leftEnd = rightPos - 1;
    int tmpPos = leftPos;
    int numElements = rightEnd - leftPos + 1;

    // Main loop
    while( leftPos <= leftEnd && rightPos <= rightEnd )
        if( a[ leftPos ].compareTo( a[ rightPos ] ) <= 0 )
            tmpArray[ tmpPos++ ] = a[ leftPos++ ];
        else
            tmpArray[ tmpPos++ ] = a[ rightPos++ ];

    while( leftPos <= leftEnd )    // Copy rest of first half
        tmpArray[ tmpPos++ ] = a[ leftPos++ ];

    while( rightPos <= rightEnd )  // Copy rest of right half
        tmpArray[ tmpPos++ ] = a[ rightPos++ ];

    // Copy tmpArray back
    for( int i = 0; i < numElements; i++, rightEnd-- )
        a[ rightEnd ] = tmpArray[ rightEnd ];
}

/**
 * Quicksort algorithm.
 * @param a an array of Comparable items.
 */
public static void quicksort( Comparable [ ] a )
{
    quicksort( a, 0, a.length - 1 );
}
```

```
        private static final int CUTOFF = 3;

        /**
         * Method to swap to elements in an array.
         * @param a an array of objects.
         * @param index1 the index of the first object.
         * @param index2 the index of the second object.
         */
        public static final void swapReferences( Object [ ] a, int index1, int index2 )
        {
            Object tmp = a[ index1 ];
            a[ index1 ] = a[ index2 ];
            a[ index2 ] = tmp;
        }

        /**
         * Return median of left, center, and right.
         * Order these and hide the pivot.
         */
        private static Comparable median3( Comparable [ ] a, int left, int right )
        {
            int center = ( left + right ) / 2;
            if( a[ center ].compareTo( a[ left ] ) < 0 )
                swapReferences( a, left, center );
            if( a[ right ].compareTo( a[ left ] ) < 0 )
                swapReferences( a, left, right );
            if( a[ right ].compareTo( a[ center ] ) < 0 )
                swapReferences( a, center, right );

                // Place pivot at position right - 1
            swapReferences( a, center, right - 1 );
            return a[ right - 1 ];
        }

        /**
         * Internal quicksort method that makes recursive calls.
         * Uses median-of-three partitioning and a cutoff of 10.
         * @param a an array of Comparable items.
         * @param left the left-most index of the subarray.
         * @param right the right-most index of the subarray.
         */
        private static void quicksort( Comparable [ ] a, int left, int right )
        {
/* 1*/          if( left + CUTOFF <= right )
            {
/* 2*/              Comparable pivot = median3( a, left, right );

                    // Begin partitioning
/* 3*/              int i = left, j = right - 1;
/* 4*/              for( ; ; )
                {
/* 5*/                  while( a[ ++i ].compareTo( pivot ) < 0 ) { }
/* 6*/                  while( a[ --j ].compareTo( pivot ) > 0 ) { }
/* 7*/                  if( i < j )
/* 8*/                      swapReferences( a, i, j );
                    else
/* 9*/                      break;
```

```
                }

/*10*/          swapReferences( a, i, right - 1 );   // Restore pivot

/*11*/          quicksort( a, left, i - 1 );    // Sort small elements
/*12*/          quicksort( a, i + 1, right );   // Sort large elements
            }
            else  // Do an insertion sort on the subarray
/*13*/          insertionSort( a, left, right );
        }

        /**
         * Internal insertion sort routine for subarrays
         * that is used by quicksort.
         * @param a an array of Comparable items.
         * @param left the left-most index of the subarray.
         * @param right the right-most index of the subarray.
         */
        private static void insertionSort( Comparable [ ] a, int left, int right )
        {
            for( int p = left + 1; p <= right; p++ )
            {
                Comparable tmp = a[ p ];
                int j;

                for( j = p; j > left && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
                    a[ j ] = a[ j - 1 ];
                a[ j ] = tmp;
            }
        }

        /**
         * Quick selection algorithm.
         * Places the kth smallest item in a[k-1].
         * @param a an array of Comparable items.
         * @param k the desired rank (1 is minimum) in the entire array.
         */
        public static void quickSelect( Comparable [ ] a, int k )
        {
            quickSelect( a, 0, a.length - 1, k );
        }

        /**
         * Internal selection method that makes recursive calls.
         * Uses median-of-three partitioning and a cutoff of 10.
         * Places the kth smallest item in a[k-1].
         * @param a an array of Comparable items.
         * @param left the left-most index of the subarray.
         * @param right the right-most index of the subarray.
         * @param k the desired index (1 is minimum) in the entire array.
         */
        private static void quickSelect( Comparable [ ] a, int left,
                                                int right, int k )
        {
/* 1*/      if( left + CUTOFF <= right )
            {
/* 2*/          Comparable pivot = median3( a, left, right );
```

```
                // Begin partitioning
/* 3*/          int i = left, j = right - 1;
/* 4*/          for( ; ; )
                {
/* 5*/              while( a[ ++i ].compareTo( pivot ) < 0 ) { }
/* 6*/              while( a[ --j ].compareTo( pivot ) > 0 ) { }
/* 7*/              if( i < j )
/* 8*/                  swapReferences( a, i, j );
                    else
/* 9*/                  break;
                }

/*10*/          swapReferences( a, i, right - 1 );   // Restore pivot

/*11*/          if( k <= i )
/*12*/              quickSelect( a, left, i - 1, k );
/*13*/          else if( k > i + 1 )
/*14*/              quickSelect( a, i + 1, right, k );
            }
            else  // Do an insertion sort on the subarray
/*15*/          insertionSort( a, left, right );
        }


        private static final int NUM_ITEMS = 1000;
        private static int theSeed = 1;

        private static void checkSort( MyInteger [ ] a )
        {
            for( int i = 0; i < a.length; i++ )
                if( a[ i ].intValue( ) != i )
                    System.out.println( "Error at " + i );
            System.out.println( "Finished checksort" );
        }


        public static void main( String [ ] args )
        {
            MyInteger [ ] a = new MyInteger[ NUM_ITEMS ];
            for( int i = 0; i < a.length; i++ )
                a[ i ] = new MyInteger( i );

            for( theSeed = 0; theSeed < 20; theSeed++ )
            {
                Random.permute( a );
                Sort.insertionSort( a );
                checkSort( a );

                Random.permute( a );
                Sort.heapsort( a );
                checkSort( a );

                Random.permute( a );
                Sort.shellsort( a );
                checkSort( a );
                Random.permute( a );
                Sort.mergeSort( a );
                checkSort( a );

                Random.permute( a );
                Sort.quicksort( a );
                checkSort( a );

                Random.permute( a );
                Sort.quickSelect( a, NUM_ITEMS / 2 );
                System.out.println( a[ NUM_ITEMS / 2 - 1 ].intValue( ) + " " +
                                  NUM_ITEMS / 2 );
            }
        }
    }
```

7.37 SplayTree.java

```
package DataStructures;

// SplayTree class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x
// Comparable find( x )   --> Return item that matches x
// Comparable findMin( )  --> Return smallest item
// Comparable findMax( )  --> Return largest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// void printTree( )      --> Print tree in sorted order

/**
 * Implements a top-down splay tree.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 */
public class SplayTree
{
    /**
     * Construct the tree.
     */
    public SplayTree( )
    {
        root = nullNode;
    }

    /**
     * Insert into the tree.
     * @param x the item to insert.
     */
    public void insert( Comparable x )
    {
        if( newNode == null )
            newNode = new BinaryNode( null );
        newNode.element = x;

        if( root == nullNode )
        {
            newNode.left = newNode.right = nullNode;
            root = newNode;
        }
        else
        {
            root = splay( x, root );
            if( x.compareTo( root.element ) < 0 )
            {
                newNode.left = root.left;
                newNode.right = root;
                root.left = nullNode;
                root = newNode;
            }
        else
        if( x.compareTo( root.element ) > 0 )
        {
            newNode.right = root.right;
            newNode.left = root;
            root.right = nullNode;
            root = newNode;
        }
        else
            return;
    }
    newNode = null;   // So next insert will call new
}

/**
 * Remove from the tree.
 * @param x the item to remove.
 */
public void remove( Comparable x )
{
    BinaryNode newTree;

        // If x is found, it will be at the root
    root = splay( x, root );
    if( root.element.compareTo( x ) != 0 )
        return;   // Item not found; do nothing

    if( root.left == nullNode )
        newTree = root.right;
    else
    {
        // Find the maximum in the left subtree
        // Splay it to the root; and then attach right child
        newTree = root.left;
        newTree = splay( x, newTree );
        newTree.right = root.right;
    }
    root = newTree;
}

/**
 * Find the smallest item in the tree.
 * Not the most efficient implementation (uses two passes), but has correct
 *     amortized behavior.
 * A good alternative is to first call Find with parameter
 *     smaller than any item in the tree, then call findMin.
 * @return the smallest item or null if empty.
 */
public Comparable findMin( )
{
    if( isEmpty( ) )
        return null;

    BinaryNode ptr = root;

    while( ptr.left != nullNode )
        ptr = ptr.left;
    root = splay( ptr.element, root );
    return ptr.element;
}

/**
 * Find the largest item in the tree.
 * Not the most efficient implementation (uses two passes), but has correct
 *     amortized behavior.
 * A good alternative is to first call Find with parameter
 *     larger than any item in the tree, then call findMax.
 * @return the largest item or null if empty.
 */
public Comparable findMax( )
{
    if( isEmpty( ) )
        return null;

    BinaryNode ptr = root;

    while( ptr.right != nullNode )
        ptr = ptr.right;

    root = splay( ptr.element, root );
    return ptr.element;
}

/**
 * Find an item in the tree.
 * @param x the item to search for.
 * @return the matching item or null if not found.
 */
public Comparable find( Comparable x )
{
    root = splay( x, root );

    if( root.element.compareTo( x ) != 0 )
        return null;

    return root.element;
}

/**
 * Make the tree logically empty.
 */
public void makeEmpty( )
{
    root = nullNode;
}

/**
 * Test if the tree is logically empty.
 * @return true if empty, false otherwise.
 */
public boolean isEmpty( )
{
    return root == nullNode;
```

```
}

/**
 * Print the tree contents in sorted order.
 */
public void printTree( )
{
    if( isEmpty( ) )
        System.out.println( "Empty tree" );
    else
        printTree( root );
}

/**
 * Internal method to perform a top-down splay.
 * The last accessed node becomes the new root.
 * @param x the target item to splay around.
 * @param t the root of the subtree to splay.
 * @return the subtree after the splay.
 */
private BinaryNode splay( Comparable x, BinaryNode t )
{
    BinaryNode leftTreeMax, rightTreeMin;

    header.left = header.right = nullNode;
    leftTreeMax = rightTreeMin = header;

    nullNode.element = x;   // Guarantee a match

    for( ; ; )
        if( x.compareTo( t.element ) < 0 )
        {
            if( x.compareTo( t.left.element ) < 0 )
                t = rotateWithLeftChild( t );
            if( t.left == nullNode )
                break;
            // Link Right
            rightTreeMin.left = t;
            rightTreeMin = t;
            t = t.left;
        }
        else if( x.compareTo( t.element ) > 0 )
        {
            if( x.compareTo( t.right.element ) > 0 )
                t = rotateWithRightChild( t );
            if( t.right == nullNode )
                break;
            // Link Left
            leftTreeMax.right = t;
            leftTreeMax = t;
            t = t.right;
        }
        else
            break;

    leftTreeMax.right = t.left;
    rightTreeMin.left = t.right;
    t.left = header.right;
    t.right = header.left;
    return t;
}

/**
 * Rotate binary tree node with left child.
 */
static BinaryNode rotateWithLeftChild( BinaryNode k2 )
{
    BinaryNode k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    return k1;
}

/**
 * Rotate binary tree node with right child.
 */
static BinaryNode rotateWithRightChild( BinaryNode k1 )
{
    BinaryNode k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    return k2;
}

/**
 * Internal method to print a subtree in sorted order.
 * WARNING: This is prone to running out of stack space.
 * @param t the node that roots the tree.
 */
private void printTree( BinaryNode t )
{
    if( t != t.left )
    {
        printTree( t.left );
        System.out.println( t.element.toString( ) );
        printTree( t.right );
    }
}


private BinaryNode root;
private static BinaryNode nullNode;
    static         // Static initializer for nullNode
    {
        nullNode = new BinaryNode( null );
        nullNode.left = nullNode.right = nullNode;
    }

private static BinaryNode newNode = null;  // Used between different inserts
private static BinaryNode header = new BinaryNode( null ); // For splay


    // Test program; should print min and max and nothing else
public static void main( String [ ] args )
```

```
    {
        SplayTree t = new SplayTree( );
        final int NUMS = 40000;
        final int GAP  =   307;

        System.out.println( "Checking... (no bad output means success)" );

        for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
            t.insert( new MyInteger( i ) );
        System.out.println( "Inserts complete" );

        for( int i = 1; i < NUMS; i+= 2 )
            t.remove( new MyInteger( i ) );
        System.out.println( "Removes complete" );

        if( NUMS < 40 )
            t.printTree( );
        if( ((MyInteger)(t.findMin( ))).intValue( ) != 2 ||
            ((MyInteger)(t.findMax( ))).intValue( ) != NUMS - 2 )
            System.out.println( "FindMin or FindMax error!" );

        for( int i = 2; i < NUMS; i+=2 )
            if( ((MyInteger)t.find( new MyInteger( i ) )).intValue( ) != i )
                System.out.println( "Error: find fails for " + i );

        for( int i = 1; i < NUMS; i+=2 )
            if( t.find( new MyInteger( i ) )  != null )
                System.out.println( "Error: Found deleted item " + i );
    }
}
```

7.38 StackAr.java

```
package DataStructures;

// StackAr class
//
// CONSTRUCTION: with or without a capacity; default is 10
//
// ******************PUBLIC OPERATIONS*********************
// void push( x )         --> Insert x
// void pop( )            --> Remove most recently inserted item
// Object top( )          --> Return most recently inserted item
// Object topAndPop( )    --> Return and remove most recently inserted item
// boolean isEmpty( )     --> Return true if empty; else false
// boolean isFull( )      --> Return true if full; else false
// void makeEmpty( )      --> Remove all items
// ******************ERRORS********************************
// Overflow and Underflow thrown as needed

/**
 * Array-based implementation of the stack.
 * @author Mark Allen Weiss
 */
public class StackAr
{
    /**
     * Construct the stack.
     */
    public StackAr( )
    {
        this( DEFAULT_CAPACITY );
    }

    /**
     * Construct the stack.
     * @param capacity the capacity.
     */
    public StackAr( int capacity )
    {
        theArray = new Object[ capacity ];
        topOfStack = -1;
    }

    /**
     * Test if the stack is logically empty.
     * @return true if empty, false otherwise.
     */
    public boolean isEmpty( )
    {
        return topOfStack == -1;
    }

    /**
     * Test if the stack is logically full.
     * @return true if full, false otherwise.
     */
    public boolean isFull( )
    {
```

```
    return topOfStack == theArray.length - 1;
}

/**
 * Make the stack logically empty.
 */
public void makeEmpty( )
{
    topOfStack = -1;
}

/**
 * Get the most recently inserted item in the stack.
 * Does not alter the stack.
 * @return the most recently inserted item in the stack, or null, if empty.
 */
public Object top( )
{
    if( isEmpty( ) )
        return null;
    return theArray[ topOfStack ];
}

/**
 * Remove the most recently inserted item from the stack.
 * @exception Underflow if stack is already empty.
 */
public void pop( ) throws Underflow
{
    if( isEmpty( ) )
        throw new Underflow( );
    theArray[ topOfStack-- ] = null;
}

/**
 * Insert a new item into the stack, if not already full.
 * @param x the item to insert.
 * @exception Overflow if stack is already full.
 */
public void push( Object x ) throws Overflow
{
    if( isFull( ) )
        throw new Overflow( );
    theArray[ ++topOfStack ] = x;
}

/**
 * Return and remove most recently inserted item from the stack.
 * @return most recently inserted item, or null, if stack is empty.
 */
public Object topAndPop( )
{
    if( isEmpty( ) )
        return null;
    Object topItem = top( );
    theArray[ topOfStack-- ] = null;
    return topItem;
    }

    private Object [ ] theArray;
    private int        topOfStack;

    static final int DEFAULT_CAPACITY = 10;

    public static void main( String [ ] args )
    {
        StackAr s = new StackAr( 12 );

        try
        {
            for( int i = 0; i < 10; i++ )
                s.push( new MyInteger( i ) );
        }
        catch( Overflow e ) { System.out.println( "Unexpected overflow" ); }

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

7.39 StackLi.java

```
package DataStructures;

// StackLi class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// void push( x )         --> Insert x
// void pop( )            --> Remove most recently inserted item
// Object top( )          --> Return most recently inserted item
// Object topAndPop( )    --> Return and remove most recent item
// boolean isEmpty( )     --> Return true if empty; else false
// boolean isFull( )      --> Always returns false
// void makeEmpty( )      --> Remove all items
// ******************ERRORS********************************
// pop on empty stack

/**
 * List-based implementation of the stack.
 * @author Mark Allen Weiss
 */
public class StackLi
{
    /**
     * Construct the stack.
     */
    public StackLi( )
    {
        topOfStack = null;
    }

    /**
     * Test if the stack is logically full.
     * @return false always, in this implementation.
     */
    public boolean isFull( )
    {
        return false;
    }

    /**
     * Test if the stack is logically empty.
     * @return true if empty, false otherwise.
     */
    public boolean isEmpty( )
    {
        return topOfStack == null;
    }

    /**
     * Make the stack logically empty.
     */
    public void makeEmpty( )
    {
        topOfStack = null;
    }
```

```
/**
 * Get the most recently inserted item in the stack.
 * Does not alter the stack.
 * @return the most recently inserted item in the stack, or null, if empty.
 */
public Object top( )
{
    if( isEmpty( ) )
        return null;
    return topOfStack.element;
}

/**
 * Remove the most recently inserted item from the stack.
 * @exception Underflow if the stack is empty.
 */
public void pop( ) throws Underflow
{
    if( isEmpty( ) )
        throw new Underflow( );
    topOfStack = topOfStack.next;
}

/**
 * Return and remove the most recently inserted item from the stack.
 * @return the most recently inserted item in the stack, or null, if empty.
 */
public Object topAndPop( )
{
    if( isEmpty( ) )
        return null;

    Object topItem = topOfStack.element;
    topOfStack = topOfStack.next;
    return topItem;
}

/**
 * Insert a new item into the stack.
 * @param x the item to insert.
 */
public void push( Object x )
{
    topOfStack = new ListNode( x, topOfStack );
}

private ListNode topOfStack;


public static void main( String [ ] args )
{
    StackLi s = new StackLi( );

    for( int i = 0; i < 10; i++ )
        s.push( new MyInteger( i ) );
        while( !s.isEmpty( ) )
            System.out.println( s.topAndPop( ) );
    }
}
```

7.40 Treap.java

```
package DataStructures;

// Treap class
//
// CONSTRUCTION: with no initializer
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// void remove( x )       --> Remove x
// Comparable find( x )   --> Return item that matches x
// Comparable findMin( )  --> Return smallest item
// Comparable findMax( )  --> Return largest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// void printTree( )      --> Print tree in sorted order

/**
 * Implements a treap.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 */
public class Treap
{
    /**
     * Construct the treap.
     */
    public Treap( )
    {
        root = nullNode;
    }

    /**
     * Insert into the tree. Does nothing if x is already present.
     * @param x the item to insert.
     */
    public void insert( Comparable x )
    {
        root = insert( x, root );
    }

    /**
     * Remove from the tree. Does nothing if x is not found.
     * @param x the item to remove.
     */
    public void remove( Comparable x )
    {
        root = remove( x, root );
    }

    /**
     * Find the smallest item in the tree.
     * @return the smallest item, or null if empty.
     */
    public Comparable findMin( )
    {
        if( isEmpty( ) )
        return null;

    TreapNode ptr = root;

    while( ptr.left != nullNode )
        ptr = ptr.left;

    return ptr.element;
}

/**
 * Find the largest item in the tree.
 * @return the largest item, or null if empty.
 */
public Comparable findMax( )
{
    if( isEmpty( ) )
        return null;

    TreapNode ptr = root;

    while( ptr.right != nullNode )
        ptr = ptr.right;

    return ptr.element;
}

/**
 * Find an item in the tree.
 * @param x the item to search for.
 * @return the matching item, or null if not found.
 */
public Comparable find( Comparable x )
{
    TreapNode current = root;
    nullNode.element = x;

    for( ; ; )
    {
        if( x.compareTo( current.element ) < 0 )
            current = current.left;
        else if( x.compareTo( current.element ) > 0 )
            current = current.right;
        else if( current != nullNode )
            return current.element;
        else
            return null;
    }
}

/**
 * Make the tree logically empty.
 */
public void makeEmpty( )
{
    root = nullNode;
}
```

```
/**
 * Test if the tree is logically empty.
 * @return true if empty, false otherwise.
 */
public boolean isEmpty( )
{
    return root == nullNode;
}

/**
 * Print the tree contents in sorted order.
 */
public void printTree( )
{
    if( isEmpty( ) )
        System.out.println( "Empty tree" );
    else
        printTree( root );
}

/**
 * Internal method to insert into a subtree.
 * @param x the item to insert.
 * @param t the node that roots the tree.
 * @return the new root.
 */
private TreapNode insert( Comparable x, TreapNode t )
{
    if( t == nullNode )
        t = new TreapNode( x, nullNode, nullNode );
    else if( x.compareTo( t.element ) < 0 )
    {
        t.left = insert( x, t.left );
        if( t.left.priority < t.priority )
            t = rotateWithLeftChild( t );
    }
    else if( x.compareTo( t.element ) > 0  )
    {
        t.right = insert( x, t.right );
        if( t.right.priority < t.priority )
            t = rotateWithRightChild( t );
    }
    // Otherwise, it's a duplicate; do nothing

    return t;
}

/**
 * Internal method to remove from a subtree.
 * @param x the item to remove.
 * @param t the node that roots the tree.
 * @return the new root.
 */
private TreapNode remove( Comparable x, TreapNode t )
{
    if( t != nullNode )
    {
        if( x.compareTo( t.element ) < 0 )
            t.left = remove( x, t.left );
        else if( x.compareTo( t.element ) > 0 )
            t.right = remove( x, t.right );
        else
        {
                // Match found
            if( t.left.priority < t.right.priority )
                t = rotateWithLeftChild( t );
            else
                t = rotateWithRightChild( t );

            if( t != nullNode )     // Continue on down
                t = remove( x, t );
            else
                t.left = nullNode;  // At a leaf
        }
    }
    return t;
}

/**
 * Internal method to print a subtree in sorted order.
 * @param t the node that roots the tree.
 */
private void printTree( TreapNode t )
{
    if( t != t.left )
    {
        printTree( t.left );
        System.out.println( t.element.toString( ) );
        printTree( t.right );
    }
}

/**
 * Rotate binary tree node with left child.
 */
static TreapNode rotateWithLeftChild( TreapNode k2 )
{
    TreapNode k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    return k1;
}

/**
 * Rotate binary tree node with right child.
 */
static TreapNode rotateWithRightChild( TreapNode k1 )
{
    TreapNode k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    return k2;
}
```

```
    private TreapNode root;
    private static TreapNode nullNode;
        static         // Static initializer for NullNode
        {
            nullNode = new TreapNode( null );
            nullNode.left = nullNode.right = nullNode;
            nullNode.priority = Integer.MAX_VALUE;
        }

        // Test program
    public static void main( String [ ] args )
    {
        Treap t = new Treap( );
        final int NUMS = 40000;
        final int GAP  =   307;

        System.out.println( "Checking... (no bad output means success)" );

        for( int i = GAP; i != 0; i = ( i + GAP ) % NUMS )
            t.insert( new MyInteger( i ) );
        System.out.println( "Inserts complete" );

        for( int i = 1; i < NUMS; i+= 2 )
            t.remove( new MyInteger( i ) );
        System.out.println( "Removes complete" );

        if( NUMS < 40 )
            t.printTree( );
        if( ((MyInteger)(t.findMin( ))).intValue( ) != 2 ||
            ((MyInteger)(t.findMax( ))).intValue( ) != NUMS - 2 )
            System.out.println( "FindMin or FindMax error!" );

        for( int i = 2; i < NUMS; i+=2 )
            if( ((MyInteger)t.find( new MyInteger( i ) )).intValue( ) != i )
                System.out.println( "Error: find fails for " + i );

        for( int i = 1; i < NUMS; i+=2 )
            if( t.find( new MyInteger( i ) )  != null )
                System.out.println( "Error: Found deleted item " + i );
    }
}
```

7.41 TreapNode.java

```
package DataStructures;

// Basic node stored in treaps
// Note that this class is not accessible outside
// of package DataStructures

class TreapNode
{
        // Constructors
    TreapNode( Comparable theElement )
    {
        this( theElement, null, null );
    }

    TreapNode( Comparable theElement, TreapNode lt, TreapNode rt )
    {
        element  = theElement;
        left     = lt;
        right    = rt;
        priority = randomObj.randomInt( );
    }

        // Friendly data; accessible by other package routines
    Comparable element;      // The data in the node
    TreapNode  left;         // Left child
    TreapNode  right;        // Right child
    int        priority;     // Priority

    private static Random randomObj = new Random( );
}
```

7.42 Underflow.java

```
package DataStructures;

/**
 * Exception class for access in empty containers
 * such as stacks, queues, and priority queues.
 * @author Mark Allen Weiss
 */
public class Underflow extends Exception
{
}
```

8 Finite Automata and Formal Grammars