THE STACK.JAVA PROGRAM

```
// Stack.java
// demonstrates stacks
// to run this program: C>java StackApp
import java.io.*; // for I/O
////////////////////////////////////////////////////////////////
class StackX
{
private int maxSize; // size of stack array
private double[] stackArray;
private int top; // top of stack
//--------------------------------------------------------------
public StackX(int s) // constructor
{
maxSize = s; // set array size
stackArray = new double[maxSize]; // create array
top = -1; // no items yet
}
//--------------------------------------------------------------
public void push(double j) // put item on top of stack
{
stackArray[++top] = j; // increment top, insert item
}
//--------------------------------------------------------------
public double pop() // take item from top of stack
{
return stackArray[top--]; // access item, decrement top
}
//--------------------------------------------------------------
public double peek() // peek at top of stack
{
return stackArray[top];
}
//--------------------------------------------------------------
public boolean isEmpty() // true if stack is empty
{return (top == -1);
}
//--------------------------------------------------------------
public boolean isFull() // true if stack is full
{
return (top == maxSize-1);
}
//--------------------------------------------------------------
} // end class StackX
////////////////////////////////////////////////////////////////
class StackApp
{
public static void main(String[] args)
{
StackX theStack = new StackX(10); // make new stack
theStack.push(20); // push items onto stack
theStack.push(40);
theStack.push(60);
theStack.push(80);
while( !theStack.isEmpty() ) // until it's empty,
{ // delete item from
stack
```

```
double value = theStack.pop();
System.out.print(value); // display it
System.out.print(" ");
} // end while
System.out.println("");
} // end main()
} // end class StackApp
```

THE BRACKETS.JAVA PROGRAM

```
// brackets.java
// stacks used to check matching brackets
// to run this program: C>java BracketsApp
import java.io.*; // for I/O
////////////////////////////////////////////////////////////////
class StackX
{
private int maxSize;
private char[] stackArray;
private int top;
//--------------------------------------------------------------
public StackX(int s) // constructor
{
maxSize = s;
stackArray = new char[maxSize];
top = -1;
}
//--------------------------------------------------------------
public void push(char j) // put item on top of stack
{
stackArray[++top] = j;
}
//--------------------------------------------------------------
public char pop() // take item from top of stack
{
return stackArray[top--];
}
//--------------------------------------------------------------
public char peek() // peek at top of stack
{
return stackArray[top];
}
//--------------------------------------------------------------
public boolean isEmpty() // true if stack is empty
{
return (top == -1);
}
//--------------------------------------------------------------
} // end class StackX
////////////////////////////////////////////////////////////////
class BracketChecker
{
private String input; // input string
//--------------------------------------------------------------
public BracketChecker(String in) // constructor
{ input = in; }
//--------------------------------------------------------------
public void check()
```

```
{
int stackSize = input.length(); // get max stack
size
StackX theStack = new StackX(stackSize); // make stack
for(int j=0; j<input.length(); j++) // get chars in turn
{
char ch = input.charAt(j); // get char
switch(ch)
{
case '{': // opening symbols
case '[':
case '(':
theStack.push(ch); // push them
break;
case '}': // closing symbols
case ']':
case ')':
if( !theStack.isEmpty() ) // if stack not
empty,
{
char chx = theStack.pop(); // pop and check
if( (ch=='}' && chx!='{') ||
(ch==']' && chx!='[') ||
(ch==')' && chx!='(') )
System.out.println("Error: "+ch+" at "+j);
}
else // prematurely empty
System.out.println("Error: "+ch+" at "+j);
break;
default: // no action on other characters
break;
} // end switch
} // end for
// at this point, all characters have been processed
if( !theStack.isEmpty() )
System.out.println("Error: missing right delimiter");
} // end check()
//------------------------------------------------------------
} // end class BracketChecker
//////////////////////////////////////////////////////////////
class BracketsApp
{
public static void main(String[] args) throws IOException
{
String input;
while(true)
{
System.out.print(
"Enter string containing delimiters: ");
System.out.flush();
input = getString(); // read a string from kbd
if( input.equals("") ) // quit if [Enter]
break;
// make a BracketChecker
BracketChecker theChecker = new BracketChecker(input);
theChecker.check(); // check brackets
} // end while
} // end main()
```

```
//-------------------------------------------------------------
public static String getString() throws IOException
{
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
String s = br.readLine();
return s;
}
//-------------------------------------------------------------
} // end class BracketsApp
```

THE QUEUE.JAVA PROGRAM

```
// Queue.java
// demonstrates queue
// to run this program: C>java QueueApp
import java.io.*; // for I/O
////////////////////////////////////////////////////////////////
class Queue
{
private int maxSize;
private int[] queArray;
private int front;
private int rear;
private int nItems;
//-------------------------------------------------------------
public Queue(int s) // constructor
{
maxSize = s;
queArray = new int[maxSize];
front = 0;
rear = -1;
nItems = 0;
}
//-------------------------------------------------------------
public void insert(int j) // put item at rear of queue
{
if(rear == maxSize-1) // deal with wraparound
rear = -1;
queArray[++rear] = j; // increment rear and
insert
nItems++; // one more item
}
//-------------------------------------------------------------
public int remove() // take item from front of queue
{
int temp = queArray[front++]; // get value and incr front
if(front == maxSize) // deal with wraparound
front = 0;
nItems--; // one less item
return temp;
}
//-------------------------------------------------------------
public int peekFront() // peek at front of queue
{
return queArray[front];
}
//-------------------------------------------------------------
```

4

```
public boolean isEmpty() // true if queue is empty
{
return (nItems==0);
}
//-------------------------------------------------------------
public boolean isFull() // true if queue is full
{return (nItems==maxSize);
}
//-------------------------------------------------------------
public int size() // number of items in queue
{
return nItems;
}
//-------------------------------------------------------------

} // end class Queue
////////////////////////////////////////////////////////////////
class QueueApp
{
public static void main(String[] args)
{
Queue theQueue = new Queue(5); // queue holds 5 items
theQueue.insert(10); // insert 4 items
theQueue.insert(20);
theQueue.insert(30);
theQueue.insert(40);
theQueue.remove(); // remove 3 items
theQueue.remove(); // (10, 20, 30)
theQueue.remove();
theQueue.insert(50); // insert 4 more items
theQueue.insert(60); // (wraps around)
theQueue.insert(70);
theQueue.insert(80);
while( !theQueue.isEmpty() ) // remove and display
{ // all items
int n = theQueue.remove(); // (40, 50, 60, 70, 80)
System.out.print(n);
System.out.print(" ");
}
System.out.println("");
} // end main()
} // end class QueueApp
```


THE QUEUE CLASS WITHOUT NITEMS

```
class Queue
{
private int maxSize;
private int[] queArray;
private int front;
private int rear;
//-------------------------------------------------------------
public Queue(int s) // constructor
{
maxSize = s+1; // array is 1 cell larger
queArray = new int[maxSize]; // than requested
front = 0;
rear = -1;
```

```
}
//-------------------------------------------------------------
public void insert(int j) // put item at rear of queue
{
if(rear == maxSize-1)
rear = -1;
queArray[++rear] = j;
}
//-------------------------------------------------------------
public int remove() // take item from front of queue
{
int temp = queArray[front++];
if(front == maxSize)
front = 0;
return temp;
}
//-------------------------------------------------------------
public int peek() // peek at front of queue
{
return queArray[front];
}
//-------------------------------------------------------------
public boolean isEmpty() // true if queue is empty
{
return ( rear+1==front || (front+maxSize-1==rear) );
}
//-------------------------------------------------------------
public boolean isFull() // true if queue is full
{
return ( rear+2==front || (front+maxSize-2==rear) );
}
//-------------------------------------------------------------
public int size() // (assumes queue not empty)
{
if(rear >= front) // contiguous sequence
return rear-front+1;
else // broken sequence
return (maxSize-front) + (rear+1);
}
//-------------------------------------------------------------
} // end class Queue
```

THE PRIORITYQ.JAVA PROGRAM

```
// priorityQ.java
// demonstrates priority queue
// to run this program: C>java PriorityQApp
import java.io.*; // for I/O
////////////////////////////////////////////////////////////////
class PriorityQ
{
// array in sorted order, from max at 0 to min at size-1
private int maxSize;
private double[] queArray;
private int nItems;
//-------------------------------------------------------------
public PriorityQ(int s) // constructor
{
```

```
maxSize = s;
queArray = new double[maxSize];
nItems = 0;
}
//-------------------------------------------------------------
public void insert(double item) // insert item
{
int j;
if(nItems==0) // if no items,
queArray[nItems++] = item; // insert at 0
else // if any items,
{
for(j=nItems-1; j>=0; j--) // start at end,
{
if( item > queArray[j] ) // if new item
larger,
queArray[j+1] = queArray[j]; // shift upward
else // if smaller,
break; // done shifting
} // end for
queArray[j+1] = item; // insert it
nItems++;
} // end else (nItems > 0)
} // end insert()
//-------------------------------------------------------------
public double remove() // remove minimum item
{ return queArray[--nItems]; }
//-------------------------------------------------------------
public double peekMin() // peek at minimum item
{ return queArray[nItems-1]; }
//-------------------------------------------------------------
public boolean isEmpty() // true if queue is empty
{ return (nItems==0); }
//-------------------------------------------------------------
public boolean isFull() // true if queue is full
{ return (nItems == maxSize); }
//-------------------------------------------------------------
} // end class PriorityQ
////////////////////////////////////////////////////////////////
class PriorityQApp
{
public static void main(String[] args) throws IOException
{
PriorityQ thePQ = new PriorityQ(5);
thePQ.insert(30);
thePQ.insert(50);
thePQ.insert(10);
thePQ.insert(40);
thePQ.insert(20);
while( !thePQ.isEmpty() )
{
double item = thePQ.remove();
System.out.print(item + " "); // 10, 20, 30, 40, 50
} // end while
System.out.println("");
} // end main()
//-------------------------------------------------------------
} // end class PriorityQApp
```

THE INFIX.JAVA PROGRAM

```
// infix.java
// converts infix arithmetic expressions to postfix
// to run this program: C>java InfixApp
import java.io.*; // for I/O
////////////////////////////////////////////////////////////////
class StackX
{
private int maxSize;
private char[] stackArray;
private int top;
//--------------------------------------------------------------
public StackX(int s) // constructor
{
maxSize = s;
stackArray = new char[maxSize];
top = -1;
}
//--------------------------------------------------------------
public void push(char j) // put item on top of stack
{ stackArray[++top] = j; }
//--------------------------------------------------------------
public char pop() // take item from top of stack
{ return stackArray[top--]; }
//--------------------------------------------------------------
public char peek() // peek at top of stack
{ return stackArray[top]; }
//--------------------------------------------------------------
public boolean isEmpty() // true if stack is empty
{ return (top == -1); }
//--------------------------------------------------------------
public int size() // return size
{ return top+1; }
//--------------------------------------------------------------
public char peekN(int n) // return item at index n
{ return stackArray[n]; }
//--------------------------------------------------------------
public void displayStack(String s)
{
System.out.print(s);
System.out.print("Stack (bottom-->top): ");
for(int j=0; j<size(); j++)
{
System.out.print( peekN(j) );
System.out.print(' ');
}
System.out.println("");
}
//--------------------------------------------------------------
} // end class StackX
////////////////////////////////////////////////////////////////
// infix to postfix conversion
{
private StackX theStack;
private String input;
private String output = "";
//--------------------------------------------------------------
```


BY
AMAR PANCHAL
9821601163
SUBJECTS: CP-1,CP-2,DSF,DSA,CNDD,MC,DC,MMS

```
public InToPost(String in) // constructor
{
input = in;
int stackSize = input.length();
theStack = new StackX(stackSize);
}
//--------------------------------------------------------------
public String doTrans() // do translation to postfix
{
for(int j=0; j<input.length(); j++)
{
char ch = input.charAt(j);
theStack.displayStack("For "+ch+" "); // *diagnostic*
switch(ch)
{
case '+': // it's + or -
case '-':
gotOper(ch, 1); // go pop operators
break; // (precedence 1)
case '*': // it's * or /
case '/':
gotOper(ch, 2); // go pop operators
break; // (precedence 2)
case '(': // it's a left paren
theStack.push(ch); // push it
break;
case ')': // it's a right paren
gotParen(ch); // go pop operators
break;
default: // must be an operand
output = output + ch; // write it to output
break;
} // end switch
} // end for
while( !theStack.isEmpty() ) // pop remaining opers
{
theStack.displayStack("While "); // *diagnostic*
output = output + theStack.pop(); // write to output
}
theStack.displayStack("End "); // *diagnostic*
return output; // return postfix
} // end doTrans()
//--------------------------------------------------------------
public void gotOper(char opThis, int prec1)
{ // got operator from
input
while( !theStack.isEmpty() )
{
char opTop = theStack.pop();
if( opTop == '(' ) // if it's a '('
{
theStack.push(opTop); // restore '('
break;
}
else // it's an operator
{
int prec2; // precedence of new op
if(opTop=='+' || opTop=='-') // find new op prec
```

```
prec2 = 1;
else
prec2 = 2;
if(prec2 < prec1) // if prec of new op
less
{ // than prec of old
theStack.push(opTop); // save newly-popped op
break;
}
else // prec of new not less
output = output + opTop; // than prec of old
} // end else (it's an operator)
} // end while
theStack.push(opThis); // push new operator
} // end gotOp()
//-------------------------------------------------------------
public void gotParen(char ch)
{ // got right paren from
input
while( !theStack.isEmpty() )
{
char chx = theStack.pop();
if( chx == '(' ) // if popped '('
break; // we're done
else // if popped operator
output = output + chx; // output it
} // end while
} // end popOps()
//-------------------------------------------------------------
} // end class InToPost
////////////////////////////////////////////////////////////////
class InfixApp
{
public static void main(String[] args) throws IOException
{
String input, output;
while(true)
{
System.out.print("Enter infix: ");
System.out.flush();
input = getString(); // read a string from kbd
if( input.equals("") ) // quit if [Enter]
break;
// make a translator
InToPost theTrans = new InToPost(input);
output = theTrans.doTrans(); // do the translation
System.out.println("Postfix is " + output + '\n');
} // end while
} // end main()
//-------------------------------------------------------------
public static String getString() throws IOException
{
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
String s = br.readLine();
return s;
}
//-------------------------------------------------------------
```

```
} // end class InfixApp
```

THE POSTFIX.JAVA PROGRAM

```
// postfix.java
// parses postfix arithmetic expressions
// to run this program: C>java PostfixApp
import java.io.*; // for I/O
////////////////////////////////////////////////////////////////
class StackX
{
private int maxSize;
private int[] stackArray;
private int top;
//--------------------------------------------------------------
public StackX(int size) // constructor
{
maxSize = size;
stackArray = new int[maxSize];
top = -1;
}
//--------------------------------------------------------------
public void push(int j) // put item on top of stack
{ stackArray[++top] = j; }
//--------------------------------------------------------------
public int pop() // take item from top of stack
{ return stackArray[top--]; }
//--------------------------------------------------------------
public int peek() // peek at top of stack
{ return stackArray[top]; }
//--------------------------------------------------------------
public boolean isEmpty() // true if stack is empty
{ return (top == -1); }
//--------------------------------------------------------------
public boolean isFull() // true if stack is full
{ return (top == maxSize-1); }
//--------------------------------------------------------------
public int size() // return size
{ return top+1; }
//--------------------------------------------------------------
public int peekN(int n) // peek at index n
{ return stackArray[n]; }
//--------------------------------------------------------------
public void displayStack(String s)
{
System.out.print(s);
System.out.print("Stack (bottom-->top): ");
for(int j=0; j<size(); j++)
{
System.out.print( peekN(j) );
System.out.print(' ');
}
System.out.println("");
}
//--------------------------------------------------------------
} // end class StackX
////////////////////////////////////////////////////////////////
class ParsePost
```

```
{
private StackX theStack;
private String input;
//-----------------------------------------------------------
public ParsePost(String s)
{ input = s; }
//-----------------------------------------------------------
public int doParse()
{
theStack = new StackX(20); // make new stack
char ch;
int j;
int num1, num2, interAns;
for(j=0; j<input.length(); j++) // for each char,
{
ch = input.charAt(j); // read from input
theStack.displayStack(""+ch+" "); // *diagnostic*
if(ch >= '0' && ch <= '9') // if it's a number
theStack.push( (int)(ch-'0') ); // push it
else // it's an operator
{
num2 = theStack.pop(); // pop operands
num1 = theStack.pop();
switch(ch) // do arithmetic
{
case '+':
interAns = num1 + num2;
break;
case '-':
interAns = num1 - num2;
break;
case '*':
interAns = num1 * num2;
break;
case '/':
interAns = num1 / num2;
break;
default:
interAns = 0;
} // end switch
theStack.push(interAns); // push result
} // end else
} // end for
interAns = theStack.pop(); // get answer
return interAns;
} // end doParse()
} // end class ParsePost
////////////////////////////////////////////////////////////////
class PostfixApp
{
public static void main(String[] args) throws IOException
{
String input;
int output;
while(true)
{
System.out.print("Enter postfix: ");
System.out.flush();
```

```
input = getString(); // read a string from kbd
if( input.equals("") ) // quit if [Enter]
break;
// make a parser
ParsePost aParser = new ParsePost(input);
output = aParser.doParse(); // do the evaluation
System.out.println("Evaluates to " + output);
} // end while
} // end main()
//--------------------------------------------------------------
public static String getString() throws IOException
{
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
String s = br.readLine();
return s;
}
//--------------------------------------------------------
} // end class PostfixApp
```

THE LINKLIST.JAVA PROGRAM

```
// linkList.java
// demonstrates linked list
// to run this program: C>java LinkListApp
////////////////////////////////////////////////////////////////
class Link
{
public int iData; // data item (key)
public double dData; // data item
public Link next; // next link in list
// -------------------------------------------------------------
public Link(int id, double dd) // constructor
{
iData = id; // initialize data
dData = dd; // ('next' is automatically
} // set to null)
// -------------------------------------------------------------
public void displayLink() // display ourself
{
System.out.print("{" + iData + ", " + dData + "} ");
}
} // end class Link
////////////////////////////////////////////////////////////////
class LinkList
{
private Link first; // ref to first link on list
// -------------------------------------------------------------
public LinkList() // constructor
{
first = null; // no items on list yet
}
// -------------------------------------------------------------
public boolean isEmpty() // true if list is empty
{
return (first==null);
}
// -------------------------------------------------------------
```

```
// insert at start of list
public void insertFirst(int id, double dd)
{ // make new link
Link newLink = new Link(id, dd);
newLink.next = first; // newLink --> old first
first = newLink; // first --> newLink
}
// -------------------------------------------------------------
public Link deleteFirst() // delete first item
{ // (assumes list not empty)
Link temp = first; // save reference to link
first = first.next; // delete it: first-->old
next
return temp; // return deleted link
}
// -------------------------------------------------------------
public void displayList()
{
System.out.print("List (first-->last): ");
Link current = first; // start at beginning of list
while(current != null) // until end of list,
{
current.displayLink(); // print data
current = current.next; // move to next link
}
System.out.println("");
}
// -------------------------------------------------------------
} // end class LinkList
////////////////////////////////////////////////////////////////
class LinkListApp
{
public static void main(String[] args)
{
LinkList theList = new LinkList(); // make new list
theList.insertFirst(22, 2.99); // insert four items
theList.insertFirst(44, 4.99);
theList.insertFirst(66, 6.99);
theList.insertFirst(88, 8.99);
theList.displayList(); // display list
while( !theList.isEmpty() ) // until it's empty,
{
Link aLink = theList.deleteFirst(); // delete link
System.out.print("Deleted "); // display it
aLink.displayLink();
System.out.println("");
}
theList.displayList(); // display list
} // end main()
} // end class LinkListApp
```

F INDING AND DELETING SPECIFIED LINKS

```
// linkList2.java
// demonstrates linked list
// to run this program: C>java LinkList2App
////////////////////////////////////////////////////////////////
class Link
```

```
{
public int iData; // data item (key)
public double dData; // data item
public Link next; // next link in list
// -------------------------------------------------------------
public Link(int id, double dd) // constructor
{
iData = id;
dData = dd;
}
// -------------------------------------------------------------
public void displayLink() // display ourself
{
System.out.print("{" + iData + ", " + dData + "} ");
}
} // end class Link
////////////////////////////////////////////////////////////////
class LinkList
{
private Link first; // ref to first link on list
public LinkList() // constructor
{
first = null; // no links on list yet
}
// -------------------------------------------------------------
public void insertFirst(int id, double dd)
{ // make new link
Link newLink = new Link(id, dd);
newLink.next = first; // it points to old first
link
first = newLink; // now first points to this
}
// -------------------------------------------------------------
public Link find(int key) // find link with given key
{ // (assumes non-empty list)
Link current = first; // start at 'first'
while(current.iData != key) // while no match,
{
if(current.next == null) // if end of list,
return null; // didn't find it
else // not end of list,
current = current.next; // go to next link
}
return current; // found it
}
// -------------------------------------------------------------
public Link delete(int key) // delete link with given key
{ // (assumes non-empty list)
Link current = first; // search for link
Link previous = first;
while(current.iData != key)
{
if(current.next == null)
return null; // didn't find it
else
{
previous = current; // go to next link
current = current.next;
```

```
}
} // found it
if(current == first) // if first link,
first = first.next; // change first
else // otherwise,
previous.next = current.next; // bypass it
return current;
}
// -------------------------------------------------------------
public void displayList() // display the list
{
System.out.print("List (first-->last): ");
Link current = first; // start at beginning of list
while(current != null) // until end of list,
{
current.displayLink(); // print data
current = current.next; // move to next link
}
System.out.println("");
}
// -------------------------------------------------------------
} // end class LinkList
////////////////////////////////////////////////////////////////
class LinkList2App
{
public static void main(String[] args)
{
LinkList theList = new LinkList(); // make list
theList.insertFirst(22, 2.99); // insert 4 items
theList.insertFirst(44, 4.99);
theList.insertFirst(66, 6.99);
theList.insertFirst(88, 8.99);
theList.displayList(); // display list
Link f = theList.find(44); // find item
if( f != null)
System.out.println("Found link with key " + f.iData);
else
System.out.println("Can't find link");
Link d = theList.delete(66); // delete item
if( d != null )
System.out.println("Deleted link with key " +
d.iData);
else
System.out.println("Can't delete link");
theList.displayList(); // display list
} // end main()
} // end class LinkList2App
```


DOUBLE-ENDED LISTS

```
// firstLastList.java
// demonstrates list with first and last references
// to run this program: C>java FirstLastApp
////////////////////////////////////////////////////////////////
class Link
{
public double dData; // data item
public Link next; // next link in list
```

```
// -------------------------------------------------------------
public Link(double d) // constructor
{ dData = d; }
// -------------------------------------------------------------
public void displayLink() // display this link
{ System.out.print(dData + " "); }
// -------------------------------------------------------------
} // end class Link
////////////////////////////////////////////////////////////////
class FirstLastList
{
private Link first; // ref to first link
private Link last; // ref to last link
// -------------------------------------------------------------
public FirstLastList() // constructor
{
first = null; // no links on list yet
last = null;
}
// -------------------------------------------------------------
public boolean isEmpty() // true if no links
{ return first==null; }
// -------------------------------------------------------------
public void insertFirst(double dd) // insert at front of
list
{
Link newLink = new Link(dd); // make new link
if( isEmpty() ) // if empty list,
last = newLink; // newLink <-- last
newLink.next = first; // newLink --> old first
first = newLink; // first --> newLink
}
// -------------------------------------------------------------
public void insertLast(double dd) // insert at end of list
{
Link newLink = new Link(dd); // make new link
if( isEmpty() ) // if empty list,
first = newLink; // first --> newLink
else
last.next = newLink; // old last --> newLink
last = newLink; // newLink <-- last
}
// -------------------------------------------------------------
public double deleteFirst() // delete first link
{ // (assumes non-emptylist)
double temp = first.dData; // save the data
if(first.next == null) // if only one item
last = null; // null <-- last
first = first.next; // first --> old next
return temp;
}
// -------------------------------------------------------------
public void displayList()
{
System.out.print("List (first-->last): ");
Link current = first; // start at beginning
while(current != null) // until end of list,
{
```

```
current.displayLink(); // print data
current = current.next; // move to next link
}
System.out.println("");
}
// -------------------------------------------------------------
} // end class FirstLastList
////////////////////////////////////////////////////////////////
class FirstLastApp
{
public static void main(String[] args)
{ // make a new list
FirstLastList theList = new FirstLastList();
theList.insertFirst(22); // insert at front
theList.insertFirst(44);
theList.insertFirst(66);
theList.insertLast(11); // insert at rear
theList.insertLast(33);
theList.insertLast(55);
theList.displayList(); // display the list
theList.deleteFirst(); // delete first two items
theList.deleteFirst();
theList.displayList(); // display again
} // end main()
} // end class FirstLastApp]
```

A Stack Implemented by a Linked List

```
// linkStack.java
// demonstrates a stack implemented as a list
// to run this program: C>java LinkStackApp
import java.io.*; // for I/O
////////////////////////////////////////////////////////////////
class Link
{
public double dData; // data item
public Link next; // next link in list
// -------------------------------------------------------------
public Link(double dd) // constructor
{ dData = dd; }
// -------------------------------------------------------------
public void displayLink() // display ourself
{ System.out.print(dData + " "); }
} // end class Link
////////////////////////////////////////////////////////////////
class LinkList
{
private Link first; // ref to first item on list
// -------------------------------------------------------------
public LinkList() // constructor
{ first = null; } // no items on list yet
// -------------------------------------------------------------
public boolean isEmpty() // true if list is empty
{ return (first==null); }
// -------------------------------------------------------------
public void insertFirst(double dd) // insert at start of list
{ // make new link
Link newLink = new Link(dd);
```

```
newLink.next = first; // newLink --> old first
first = newLink; // first --> newLink
}
// -------------------------------------------------------------
public double deleteFirst() // delete first item
{ // (assumes list not empty)
Link temp = first; // save reference to link
first = first.next; // delete it: first-->old
next
return temp.dData; // return deleted link
}
// -------------------------------------------------------------
public void displayList()
{
Link current = first; // start at beginning of list
while(current != null) // until end of list,
{
current.displayLink(); // print data
current = current.next; // move to next link
}
System.out.println("");
}
// -------------------------------------------------------------
} // end class LinkList
////////////////////////////////////////////////////////////////
class LinkStack
{
private LinkList theList;
//--------------------------------------------------------------
public LinkStack() // constructor
{
theList = new LinkList();
}
//--------------------------------------------------------------
public void push(double j) // put item on top of stack
{
theList.insertFirst(j);
}
//--------------------------------------------------------------
public double pop() // take item from top of stack
{
return theList.deleteFirst();
}
//--------------------------------------------------------------
public boolean isEmpty() // true if stack is empty
{
return ( theList.isEmpty() );
}
//--------------------------------------------------------------
public void displayStack()
{
System.out.print("Stack (top-->bottom): ");
theList.displayList();
}
//--------------------------------------------------------------
} // end class LinkStack
////////////////////////////////////////////////////////////////
class LinkStackApp
```

```
{
public static void main(String[] args) throws IOException
{
LinkStack theStack = new LinkStack(); // make stack
theStack.push(20); // push items
theStack.push(40);
theStack.displayStack(); // display stack
theStack.push(60); // push items
theStack.push(80);
theStack.displayStack(); // display stack
theStack.pop(); // pop items
theStack.pop();
theStack.displayStack(); // display stack
} // end main()
} // end class LinkStackApp
```

A Queue Implemented by a Linked List

```
// linkQueue.java
// demonstrates queue implemented as double-ended list
// to run this program: C>java LinkQueueApp
import java.io.*; // for I/O
////////////////////////////////////////////////////////////////
class Link
{
public double dData; // data item
public Link next; // next link in list
// -------------------------------------------------------------
public Link(double d) // constructor
{ dData = d; }
// -------------------------------------------------------------
public void displayLink() // display this link
{ System.out.print(dData + " "); }
// -------------------------------------------------------------
} // end class Link
////////////////////////////////////////////////////////////////
class FirstLastList
{
private Link first; // ref to first item
private Link last; // ref to last item
// -------------------------------------------------------------
public FirstLastList() // constructor
{
first = null; // no items on list yet
last = null;
}
// -------------------------------------------------------------
public boolean isEmpty() // true if no links
{ return first==null; }
// -------------------------------------------------------------
-
public void insertLast(double dd) // insert at end of list
{
Link newLink = new Link(dd); // make new link
if( isEmpty() ) // if empty list,
first = newLink; // first --> newLink
else
last.next = newLink; // old last --> newLink
```

```
last = newLink; // newLink <-- last
}
// -------------------------------------------------------------
public double deleteFirst() // delete first link
{ // (assumes non-empty
list)
double temp = first.dData;
if(first.next == null) // if only one item
last = null; // null <-- last
first = first.next; // first --> old next
return temp;
}
// -------------------------------------------------------------
public void displayList()
{
Link current = first; // start at beginning
while(current != null) // until end of list,
{
current.displayLink(); // print data
current = current.next; // move to next link
}
System.out.println("");
}
// -------------------------------------------------------------
} // end class FirstLastList
////////////////////////////////////////////////////////////////
class LinkQueue
{
private FirstLastList theList;
//--------------------------------------------------------------
public LinkQueue() // constructor
{
theList = new FirstLastList(); // make a 2-ended list
}
//--------------------------------------------------------------
public boolean isEmpty() // true if queue is empty
{
return theList.isEmpty();
}
//--------------------------------------------------------------
public void insert(double j) // insert, rear of queue
{
theList.insertLast(j);
}
//--------------------------------------------------------------
public double remove() // remove, front of queue
{
return theList.deleteFirst();
}
//--------------------------------------------------------------
public void displayQueue()
{
System.out.print("Queue (front-->rear): ");
theList.displayList();
}
//--------------------------------------------------------------
} // end class LinkQueue
////////////////////////////////////////////////////////////////
```

21

```
class LinkQueueApp
{
public static void main(String[] args) throws IOException
{
LinkQueue theQueue = new LinkQueue();
theQueue.insert(20); // insert items
theQueue.insert(40);
theQueue.displayQueue(); // display queue
theQueue.insert(60); // insert items
theQueue.insert(80);
theQueue.displayQueue(); // display queue
theQueue.remove(); // remove items
theQueue.remove();
theQueue.displayQueue(); // display queue
} // end main()
} // end class LinkQueueApp
```

THE SORTEDLIST.JAVA PROGRAM

```
// sortedList.java
// demonstrates sorted list
// to run this program: C>java SortedListApp
import java.io.*; // for I/O
////////////////////////////////////////////////////////////////
class Link
{
public double dData; // data item
public Link next; // next link in list
// -------------------------------------------------------------
public Link(double dd) // constructor
{ dData = dd; }
// -------------------------------------------------------------
public void displayLink() // display this link
{ System.out.print(dData + " "); }
} // end class Link
////////////////////////////////////////////////////////////////
class SortedList
{
private Link first; // ref to first item on
list
// -------------------------------------------------------------
public SortedList() // constructor
{ first = null; }
// -------------------------------------------------------------
public boolean isEmpty() // true if no links
{ return (first==null); }
// -------------------------------------------------------------
public void insert(double key) // insert in order
{
Link newLink = new Link(key); // make new link
Link previous = null; // start at first
Link current = first;
// until end of list,
while(current != null && key > current.dData)
{ // or key > current,
previous = current;
current = current.next; // go to next item
}
```

```
if(previous==null) // at beginning of list
first = newLink; // first --> newLink
else // not at beginning
previous.next = newLink; // old prev --> newLink
newLink.next = current; // newLink --> old currnt
} // end insert()
// -------------------------------------------------------------
public Link remove() // return & delete first link
{ // (assumes non-empty list)
Link temp = first; // save first
first = first.next; // delete first
return temp; // return value
}
// -------------------------------------------------------------
public void displayList()
{
System.out.print("List (first-->last): ");
Link current = first; // start at beginning of list
while(current != null) // until end of list,
{
current.displayLink(); // print data
current = current.next; // move to next link
}
System.out.println("");
}
} // end class SortedList
////////////////////////////////////////////////////////////////
class SortedListApp
{
public static void main(String[] args)
{ // create new list
SortedList theSortedList = new SortedList();
theSortedList.insert(20); // insert 2 items
theSortedList.insert(40);
theSortedList.displayList(); // display list
theSortedList.insert(10); // insert 3 more items
theSortedList.insert(30);
theSortedList.insert(50);
theSortedList.displayList(); // display list
theSortedList.remove(); // remove an item
theSortedList.displayList(); // display list
} // end main()
} // end class SortedListApp
```

List Insertion Sort

```
// listInsertionSort.java
// demonstrates sorted list used for sorting
// to run this program: C>java ListInsertionSortApp
import java.io.*; // for I/O
////////////////////////////////////////////////////////////////
class Link
{
public double dData; // data item
public Link next; // next link in list
// -------------------------------------------------------------
public Link(double dd) // constructor
{ dData = dd; }
```

```
// -------------------------------------------------------------
} // end class Link
////////////////////////////////////////////////////////////////
class SortedList
{
private Link first; // ref to first item on list
// -------------------------------------------------------------
public SortedList() // constructor (no args)
{ first = null; }
// -------------------------------------------------------------
public SortedList(Link[] linkArr) // constructor (array as
{ // argument)
first = null;; // initialize list
for(int j=0; j<linkArr.length; j++) // copy array
insert( linkArr[j] ); // to list
}
// -------------------------------------------------------------
public void insert(Link k) // insert, in order
{
Link previous = null; // start at first
Link current = first;
// until end of list,
while(current != null && k.dData > current.dData)
{ // or key > current,
previous = current;
current = current.next; // go to next item
}
if(previous==null) // at beginning of list
first = k; // first --> k
else // not at beginning
previous.next = k; // old prev --> k
k.next = current; // k --> old current
} // end insert()
// -------------------------------------------------------------
public Link remove() // return & delete first link
{ // (assumes non-empty list)
Link temp = first; // save first
first = first.next; // delete first
return temp; // return value
}
// -------------------------------------------------------------
} // end class SortedList
////////////////////////////////////////////////////////////////
class ListInsertionSortApp
{
public static void main(String[] args)
{
int size = 10;
// create array of links
Link[] linkArray = new Link[size];
for(int j=0; j<size; j++) // fill array with links
{ // random number
int n = (int)(java.lang.Math.random()*99);
Link newLink = new Link(n); // make link
linkArray[j] = newLink; // put in array
}
// display array contents
System.out.print("Unsorted array: ");
```

```
for(int j=0; j<size; j++)
System.out.print( linkArray[j].dData + " " );
System.out.println("");
// create new list,
// initialized with array
SortedList theSortedList = new SortedList(linkArray);
for(int j=0; j<size; j++) // links from list to array
linkArray[j] = theSortedList.remove();
// display array contents
System.out.print("Sorted Array: ");
for(int j=0; j<size; j++)
System.out.print(linkArray[j].dData + " ");
System.out.println("");
} // end main()
} // end class ListInsertionSortApp
```

THE DOUBLYLINKED.JAVA PROGRAM

```
// doublyLinked.java
// demonstrates a doubly-linked list
// to run this program: C>java DoublyLinkedApp
////////////////////////////////////////////////////////////////
class Link
{
public double dData; // data item
public Link next; // next link in list
public Link previous; // previous link in list
// -------------------------------------------------------------
public Link(double d) // constructor
{ dData = d; }
// -------------------------------------------------------------
public void displayLink() // display this link
{ System.out.print(dData + " "); }
// -------------------------------------------------------------
} // end class Link
////////////////////////////////////////////////////////////////
class DoublyLinkedList
{
private Link first; // ref to first item
private Link last; // ref to last item
// -------------------------------------------------------------
public DoublyLinkedList() // constructor
{
first = null; // no items on list yet
last = null;
}
// -------------------------------------------------------------
public boolean isEmpty() // true if no links
{ return first==null; }
// -------------------------------------------------------------
public void insertFirst(double dd) // insert at front of
list
{
Link newLink = new Link(dd); // make new link
if( isEmpty() ) // if empty list,
last = newLink; // newLink <-- last
else
first.previous = newLink; // newLink <-- old first
newLink.next = first; // newLink --> old first
```

```
first = newLink; // first --> newLink
}
// -------------------------------------------------------------
public void insertLast(double dd) // insert at end of list
{
Link newLink = new Link(dd); // make new link
if( isEmpty() ) // if empty list,
first = newLink; // first --> newLink
else
{
last.next = newLink; // old last --> newLink
newLink.previous = last; // old last <-- newLink
}
last = newLink; // newLink <-- last
}
// -------------------------------------------------------------
public Link deleteFirst() // delete first link
{ // (assumes non-empty
list)
Link temp = first;
if(first.next == null) // if only one item
last = null; // null <-- last
else
first.next.previous = null; // null <-- old next
first = first.next; // first --> old next
return temp;
}
// -------------------------------------------------------------
public Link deleteLast() // delete last link
{ // (assumes non-empty
list)
Link temp = last;
if(first.next == null) // if only one item
first = null; // first --> null
else
last.previous.next = null; // old previous --> null
last = last.previous; // old previous <-- last
return temp;
}
// -------------------------------------------------------------
// insert dd just after
key
public boolean insertAfter(double key, double dd)
{ // (assumes non-empty
list)
Link current = first; // start at beginning
while(current.dData != key) // until match is found,
{
current = current.next; // move to next link
if(current == null)
return false; // didn't find it
}
Link newLink = new Link(dd); // make new link
if(current==last) // if last link,
{
newLink.next = null; // newLink --> null
last = newLink; // newLink <-- last
}
```

```
else // not last link,
{
newLink.next = current.next; // newLink --> old next
// newLink <-- old next
current.next.previous = newLink;
}
newLink.previous = current; // old current <-- newLink
current.next = newLink; // old current --> newLink
return true; // found it, did insertion
}
// -------------------------------------------------------------
public Link deleteKey(double key) // delete item w/ given
key
{ // (assumes non-empty
list)
Link current = first; // start at beginning
while(current.dData != key) // until match is found,
{
current = current.next; // move to next link
if(current == null)
return null; // didn't find it
}
if(current==first) // found it; first item?
first = current.next; // first --> old next
else // not first
// old previous --> old
next
current.previous.next = current.next;
if(current==last) // last item?
last = current.previous; // old previous <-- last else // not last
// old previous <-- old
next
current.next.previous = current.previous;
return current; // return value
}
// -------------------------------------------------------------
public void displayForward()
{
System.out.print("List (first-->last): ");
Link current = first; // start at beginning
while(current != null) // until end of list,
{
current.displayLink(); // display data
current = current.next; // move to next link
}
System.out.println("");
}
// -------------------------------------------------------------
public void displayBackward()
{
System.out.print("List (last-->first): ");
Link current = last; // start at end
while(current != null) // until start of list,
{
current.displayLink(); // display data
current = current.previous; // move to previous link
}
System.out.println("");
```

```
}
// -------------------------------------------------------------
} // end class DoublyLinkedList
////////////////////////////////////////////////////////////////
class DoublyLinkedApp
{
public static void main(String[] args)
{ // make a new list
DoublyLinkedList theList = new DoublyLinkedList();
theList.insertFirst(22); // insert at front
theList.insertFirst(44);
theList.insertFirst(66);
theList.insertLast(11); // insert at rear
theList.insertLast(33);
theList.insertLast(55);
theList.displayForward(); // display list forward
theList.displayBackward(); // display list backward
theList.deleteFirst(); // delete first item
theList.deleteLast(); // delete last item
theList.deleteKey(11); // delete item with key 11
theList.displayForward(); // display list forward
theList.insertAfter(22, 77); // insert 77 after 22
theList.insertAfter(33, 88); // insert 88 after 33
theList.displayForward(); // display list forward
} // end main()
} // end class DoublyLinkedApp
```

THE TOWERS.JAVA PROGRAM

```
// towers.java
// evaluates triangular numbers
// to run this program: C>java TowersApp
import java.io.*; // for I/O
////////////////////////////////////////////////////////////////
class TowersApp
{
static int nDisks = 3;
public static void main(String[] args)
{
doTowers(nDisks, 'A', 'B', 'C');
}
//-----------------------------------------------------------
public static void doTowers(int topN,
char from, char inter, char to)
{
if(topN==1)
System.out.println("Disk 1 from " + from + " to "+
to);
else
{
doTowers(topN-1, from, to, inter); // from-->inter
System.out.println("Disk " + topN +
" from " + from + " to "+ to);
doTowers(topN-1, inter, from, to); // inter-->to
}
}
//-----------------------------------------------------------
} // end class TowersApp
```

MERGESORT

```
// mergeSort.java
// demonstrates recursive mergesort
// to run this program: C>java MergeSortApp
import java.io.*; // for I/O
////////////////////////////////////////////////////////////////
class DArray
{
private double[] theArray; // ref to array theArray
private int nElems; // number of data items
//-----------------------------------------------------------
public DArray(int max) // constructor
{
theArray = new double[max]; // create array
nElems = 0;
}
//-----------------------------------------------------------
public void insert(double value) // put element into array
{
theArray[nElems] = value; // insert it
nElems++; // increment size
}
//-----------------------------------------------------------
public void display() // displays array contents
{
for(int j=0; j<nElems; j++) // for each element,
System.out.print(theArray[j] + " "); // display it
System.out.println("");
}
//-----------------------------------------------------------
public void mergeSort() // called by main()
{ // provides workspace
double[] workSpace = new double[nElems];
recMergeSort(workSpace, 0, nElems-1);
}
//-----------------------------------------------------------
private void recMergeSort(double[] workSpace, int
lowerBound,
int upperBound)
{
if(lowerBound == upperBound) // if range is 1,
return; // no use sorting
else
{ // find midpoint
int mid = (lowerBound+upperBound) / 2;
// sort low half
recMergeSort(workSpace, lowerBound, mid);
// sort high half
recMergeSort(workSpace, mid+1, upperBound);
// merge them
merge(workSpace, lowerBound, mid+1, upperBound);
} // end else
} // end recMergeSort
//-----------------------------------------------------------
private void merge(double[] workSpace, int lowPtr,
int highPtr, int upperBound)
```

```
{
int j = 0; // workspace index
int lowerBound = lowPtr;
int mid = highPtr-1;
int n = upperBound-lowerBound+1; // # of items
while(lowPtr <= mid && highPtr <= upperBound)
if( theArray[lowPtr] < theArray[highPtr] )
workSpace[j++] = theArray[lowPtr++];
else
workSpace[j++] = theArray[highPtr++];
while(lowPtr <= mid)
workSpace[j++] = theArray[lowPtr++];
while(highPtr <= upperBound)
workSpace[j++] = theArray[highPtr++];
for(j=0; j<n; j++)
theArray[lowerBound+j] = workSpace[j];
} // end merge()
//-----------------------------------------------------------
} // end class DArray
////////////////////////////////////////////////////////////////
class MergeSortApp
{
public static void main(String[] args)
{
int maxSize = 100; // array size
DArray arr; // reference to array
arr = new DArray(maxSize); // create the array
arr.insert(64); // insert items
arr.insert(21);
arr.insert(33);
arr.insert(70);
arr.insert(12);
arr.insert(85);
arr.insert(44);
arr.insert(3);
arr.insert(99);
arr.insert(0);
arr.insert(108);
arr.insert(36);
arr.display(); // display items
arr.mergeSort(); // mergesort the array
arr.display(); // display items again
} // end main()
} // end class MergeSortApp
```


JAVA CODE FOR THE SHELLSORT

```
// shellSort.java
// demonstrates shell sort
// to run this program: C>java ShellSortApp
//-------------------------------------------------------------
class ArraySh
{
private double[] theArray; // ref to array theArray
private int nElems; // number of data items
//-------------------------------------------------------------
public ArraySh(int max) // constructor
{
```

```
theArray = new double[max]; // create the array
nElems = 0; // no items yet
}
//-------------------------------------------------------------
public void insert(double value) // put element into array
{
theArray[nElems] = value; // insert it
nElems++; // increment size
}
//-------------------------------------------------------------
public void display() // displays array contents
{
System.out.print("A=");
for(int j=0; j<nElems; j++) // for each element,
System.out.print(theArray[j] + " "); // display it
System.out.println("");
}
//-------------------------------------------------------------
public void shellSort()
{
int inner, outer;
double temp;
int h = 1; // find initial value of h
while(h <= nElems/3)
h = h*3 + 1; // (1, 4, 13, 40, 121,
...)
while(h>0) // decreasing h, until h=1
{
// h-sort the file
for(outer=h; outer<nElems; outer++)
{
temp = theArray[outer];
inner = outer;
// one subpass (eg 0, 4,8)
while(inner > h-1 && theArray[inner-h] >= temp)
{
theArray[inner] = theArray[inner-h];
inner -= h;
}
theArray[inner] = temp;
} // end for
h = (h-1) / 3; // decrease h
} // end while(h>0)
} // end shellSort()
//-------------------------------------------------------------
-
} // end class ArraySh
////////////////////////////////////////////////////////////////
class ShellSortApp
{
public static void main(String[] args)
{
int maxSize = 10; // array size
ArraySh arr;
arr = new ArraySh(maxSize); // create the array
for(int j=0; j<maxSize; j++) // fill array with
{ // random numbers
double n = (int)(java.lang.Math.random()*99);
```

```
arr.insert(n);
}
arr.display(); // display unsorted array
arr.shellSort(); // shell sort the array
arr.display(); // display sorted array
} // end main()
} // end class ShellSortApp
```

THE QUICKSORT1.JAVA PROGRAM

```
// quickSort1. // quickSort1.java
// demonstrates simple version of quick sort
// to run this program: C>java QuickSort1App
////////////////////////////////////////////////////////////////
class ArrayIns
{
private double[] theArray; // ref to array theArray
private int nElems; // number of data items
//-------------------------------------------------------------
-
public ArrayIns(int max) // constructor
{
theArray = new double[max]; // create the array
nElems = 0; // no items yet
}
//-------------------------------------------------------------
public void insert(double value) // put element into array
{
theArray[nElems] = value; // insert it
nElems++; // increment size
}
//-------------------------------------------------------------
public void display() // displays array contents
{
System.out.print("A=");
for(int j=0; j<nElems; j++) // for each element,
System.out.print(theArray[j] + " "); // display it
System.out.println("");
}
//-------------------------------------------------------------
public void quickSort()
{
recQuickSort(0, nElems-1);
}
//-------------------------------------------------------------
public void recQuickSort(int left, int right)
{
if(right-left <= 0) // if size <= 1,
return; // already sorted
else // size is 2 or larger
{
double pivot = theArray[right]; // rightmost item
// partition range
int partition = partitionIt(left, right, pivot);
recQuickSort(left, partition-1); // sort left side
recQuickSort(partition+1, right); // sort right side
}
} // end recQuickSort()
```

```
//-------------------------------------------------------------
public int partitionIt(int left, int right, double pivot)
{
int leftPtr = left-1; // left (after ++)
int rightPtr = right; // right-1 (after --)
while(true)
{ // find bigger item
while(theArray[++leftPtr] < pivot)
; // (nop)
// find smaller item
while(rightPtr > 0 && theArray[--rightPtr] > pivot)
; // (nop)
if(leftPtr >= rightPtr) // if pointers cross,
break; // partition done
else // not crossed, so
swap(leftPtr, rightPtr); // swap elements
} // end while(true)
swap(leftPtr, right); // restore pivot
return leftPtr; // return pivot location
} // end partitionIt()
//-------------------------------------------------------------
public void swap(int dex1, int dex2) // swap two elements
{
double temp = theArray[dex1]; // A into temp
theArray[dex1] = theArray[dex2]; // B into A
theArray[dex2] = temp; // temp into B
} // end swap(
//-------------------------------------------------------------
} // end class ArrayIns
////////////////////////////////////////////////////////////////
class QuickSort1App
{
public static void main(String[] args)
{
int maxSize = 16; // array size
ArrayIns arr;
arr = new ArrayIns(maxSize); // create array
for(int j=0; j<maxSize; j++) // fill array with
{ // random numbers
double n = (int)(java.lang.Math.random()*99);
arr.insert(n);
}
arr.display(); // display items
arr.quickSort(); // quicksort them
arr.display(); // display them again
} // end main()
} // end class QuickSort1App
```

THE NODE CLASS OF TREE

```
class Node
{
int iData; // data used as key value
float fData; // other data
node leftChild; // this node's left child
node rightChild; // this node's right child
public void displayNode()
{
```

```
// (see Listing for method body)
}
}
```

The Tree Class

```
class Tree
{
private Node root; // the only data field in Tree
public void find(int key)
{
}
public void insert(int id, double dd)
{
}
public void delete(int id)
{
}
// various other methods
} // end class Tree
```

The TreeApp Class

```
class TreeApp
{
public static void main(String[] args)
{
Tree theTree = new Tree; // make a tree
theTree.insert(50, 1.5); // insert 3 nodes
theTree.insert(25, 1.7);
theTree.insert(75, 1.9);
node found = theTree.find(25); // find node with key 25
if(found != null)
System.out.println("Found the node with key 25");
else
System.out.println("Could not find node with key 25");
} // end main()
} // end class TreeApp
```

Finding a Node

```
public Node find(int key) // find node with given key
{ // (assumes non-empty tree)
Node current = root; // start at root
while(current.iData != key) // while no match,
{
if(key < current.iData) // go left?
current = current.leftChild;
else
current = current.rightChild; // or go right?
if(current == null) // if no child,
return null; // didn't find it
}
return current; // found it
}
```

Inserting a Node

```
public void insert(int id, double dd)
{
Node newNode = new Node(); // make new node
newNode.iData = id; // insert data
newNode.dData = dd;
```

```
if(root==null) // no node in root
root = newNode;
else // root occupied
{
Node current = root; // start at root
Node parent;
while(true) // (exits internally)
{
parent = current;
if(id < current.iData) // go left?
{
current = current.leftChild;
if(current == null) // if end of the line,
{ // insert on left
parent.leftChild = newNode;
return;
}
} // end if go left
else // or go right?
{
current = current.rightChild;
if(current == null) // if end of the line
{ // insert on right
parent.rightChild = newNode;
return;
}
} // end else go right
} // end while
} // end else not root
} // end insert()
// -------------------------------------------------------------
```


Traversing the Tree

```
private void inOrder(node localRoot)
{
if(localRoot != null)
{
inOrder(localRoot.leftChild);
localRoot.displayNode();
inOrder(localRoot.rightChild);
}
}
```

Finding Maximum and Minimum Values

```
public Node minimum() // returns node with minimum key
value
{
Node current, last;
current = root; // start at root
while(current != null) // until the bottom,
{
last = current; // remember node
current = current.leftChild; // go to left child
}
return last;
}
```

JAVA CODE FOR A LINEAR PROBE HASH TABLE

```
public DataItem find(int key) // find item with key
// (assumes table not full)
{
int hashVal = hashFunc(key); // hash the key
while(hashArray[hashVal] != null) // until empty cell,
{ // found the key?
if(hashArray[hashVal].iData == key)
return hashArray[hashVal]; // yes, return item
++hashVal; // go to next cell
hashVal %= arraySize; // wrap around if necessary
}
return null; // can't find item
}
```

The insert() Method

```
public void insert(DataItem item) // insert a DataItem
// (assumes table not full)
{
int key = item.iData; // extract key
int hashVal = hashFunc(key); // hash the key
// until empty cell or -1,
while(hashArray[hashVal] != null &&
hashArray[hashVal].iData != -1)
{
++hashVal; // go to next cell
hashVal %= arraySize; // wrap around if necessary
}
hashArray[hashVal] = item; // insert item
} // end insert()
public DataItem delete(int key) // delete a DataItem
{
int hashVal = hashFunc(key); // hash the key
while(hashArray[hashVal] != null) // until empty cell,
{ // found the key?
if(hashArray[hashVal].iData == key)
{
DataItem temp = hashArray[hashVal]; // save item
hashArray[hashVal] = nonItem; // delete item
return temp; // return item
}
++hashVal; // go to next cell
hashVal %= arraySize; // wrap around if necessary
}
return null; // can't find item
} // end delete()
```

THE HASH.JAVA PROGRAM

```
// hash.java
// demonstrates hash table with linear probing
// to run this program: C:>java HashTableApp
import java.io.*; // for I/O
import java.util.*; // for Stack class
import java.lang.Integer; // for parseInt()
////////////////////////////////////////////////////////////////
class DataItem
{ // (could have more data)
```

```
public int iData; // data item (key)
//--------------------------------------------------------------
public DataItem(int ii) // constructor
{ iData = ii; }
//--------------------------------------------------------------
} // end class DataItem
////////////////////////////////////////////////////////////////
class HashTable
{
DataItem[] hashArray; // array holds hash table
int arraySize;
DataItem nonItem; // for deleted items
// -------------------------------------------------------------
public HashTable(int size) // constructor
{
arraySize = size;
hashArray = new DataItem[arraySize];
nonItem = new DataItem(-1); // deleted item key is -1
}
// -------------------------------------------------------------
public void displayTable()
{
System.out.print("Table: ");
for(int j=0; j<arraySize; j++)
{
if(hashArray[j] != null)
System.out.print(hashArray[j].iData+ " ");
else
System.out.print("** ");
}
System.out.println("");
}
// -------------------------------------------------------------
public int hashFunc(int key)
{
return key % arraySize; // hash function
}
// -------------------------------------------------------------
public void insert(DataItem item) // insert a DataItem
// (assumes table not full)
{
int key = item.iData; // extract key
int hashVal = hashFunc(key); // hash the key
// until empty cell or -1,
while(hashArray[hashVal] != null &&
hashArray[hashVal].iData != -
1)
{
++hashVal; // go to next cell
hashVal %= arraySize; // wraparound if necessary
}
hashArray[hashVal] = item; // insert item
} // end insert()
// -------------------------------------------------------------
public DataItem delete(int key) // delete a DataItem
{
int hashVal = hashFunc(key); // hash the key
while(hashArray[hashVal] != null) // until empty cell,
```

```
{ // found the key?
if(hashArray[hashVal].iData == key)
{
DataItem temp = hashArray[hashVal]; // save item
hashArray[hashVal] = nonItem; // delete item
return temp; // return item
}
++hashVal; // go to next cell
hashVal %= arraySize; // wraparound if necessary
}
return null; // can't find item
} // end delete()
// -------------------------------------------------------------
public DataItem find(int key) // find item with key
{
int hashVal = hashFunc(key); // hash the key
while(hashArray[hashVal] != null) // until empty cell,
{ // found the key?
if(hashArray[hashVal].iData == key)
return hashArray[hashVal]; // yes, return item
++hashVal; // go to next cell
hashVal %= arraySize; // wraparound if necessary
}
return null; // can't find item
}
// -------------------------------------------------------------
} // end class HashTable
////////////////////////////////////////////////////////////////
class HashTableApp
{
public static void main(String[] args) throws IOException
{
DataItem aDataItem;
int aKey, size, n, keysPerCell;
// get sizes
putText("Enter size of hash table: ");
size = getInt();
putText("Enter initial number of items: ");
n = getInt();
keysPerCell = 10;
// make table
HashTable theHashTable = new HashTable(size);
for(int j=0; j<n; j++) // insert data
{
aKey = (int)(java.lang.Math.random() *
keysPerCell * size);
aDataItem = new DataItem(aKey);
theHashTable.insert(aDataItem);
}
while(true) // interact with user
{
putText("Enter first letter of ");
putText("show, insert, delete, or find: ");
char choice = getChar();
switch(choice)
{
case 's':
theHashTable.displayTable();
```

```
break;
case 'i':
putText("Enter key value to insert: ");
aKey = getInt();
aDataItem = new DataItem(aKey);
theHashTable.insert(aDataItem);
break;
case 'd':
putText("Enter key value to delete: ");
aKey = getInt();
theHashTable.delete(aKey);
break;
case 'f':
putText("Enter key value to find: ");
aKey = getInt();
aDataItem = theHashTable.find(aKey);
if(aDataItem != null)
{
System.out.println("Found " + aKey);
}
else
System.out.println("Could not find " + aKey);
break;
default:
putText("Invalid entry\n");
} // end switch
} // end while
} // end main()
//-------------------------------------------------------------
public static void putText(String s)
{
System.out.print(s);
System.out.flush();
}
//-------------------------------------------------------------
public static String getString() throws IOException
{
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
String s = br.readLine();
return s;
}
//-------------------------------------------------------------
public static char getChar() throws IOException
{
String s = getString();
return s.charAt(0);
}
//-------------------------------------------------------------
public static int getInt() throws IOException
{
String s = getString();
return Integer.parseInt(s);
}
//-------------------------------------------------------------
} // end class HashTableApp
```

JAVA CODE FOR DOUBLE HASHING

```
// hashDouble.java
// demonstrates hash table with double hashing
// to run this program: C:>java HashDoubleApp
import java.io.*; // for I/O
import java.util.*; // for Stack class
import java.lang.Integer; // for parseInt()
////////////////////////////////////////////////////////////////
class DataItem
{ // (could have more items)
public int iData; // data item (key)
//--------------------------------------------------------------
public DataItem(int ii) // constructor
{ iData = ii; }
//--------------------------------------------------------------
} // end class DataItem
////////////////////////////////////////////////////////////////
class HashTable
{
DataItem[] hashArray; // array is the hash table
int arraySize;
DataItem nonItem; // for deleted items
// -------------------------------------------------------------
HashTable(int size) // constructor
{
arraySize = size;
hashArray = new DataItem[arraySize];
nonItem = new DataItem(-1);
}
// -------------------------------------------------------------
public void displayTable()
{
System.out.print("Table: ");
for(int j=0; j<arraySize; j++)
{
if(hashArray[j] != null)
System.out.print(hashArray[j].iData+ " ");
else
System.out.print("** ");
}
System.out.println("");
}
// -------------------------------------------------------------
public int hashFunc1(int key)
{
return key % arraySize;
}
// -------------------------------------------------------------
public int hashFunc2(int key)
{
// non-zero, less than array size, different from hF1
// array size must be relatively prime to 5, 4, 3, and 2
return 5 - key % 5;
}
// -------------------------------------------------------------
// insert a DataItem
public void insert(int key, DataItem item)
```

```
// (assumes table not full)
{
int hashVal = hashFunc1(key); // hash the key
int stepSize = hashFunc2(key); // get step size
// until empty cell or -1
while(hashArray[hashVal] != null &&
hashArray[hashVal].iData != -
1)
{
hashVal += stepSize; // add the step
hashVal %= arraySize; // for wraparound
}
hashArray[hashVal] = item; // insert item
} // end insert()
// -------------------------------------------------------------
public DataItem delete(int key) // delete a DataItem
{
int hashVal = hashFunc1(key); // hash the key
int stepSize = hashFunc2(key); // get step size
while(hashArray[hashVal] != null) // until empty cell,
{ // is correct hashVal?
if(hashArray[hashVal].iData == key)
{
DataItem temp = hashArray[hashVal]; // save item
hashArray[hashVal] = nonItem; // delete item
return temp; // return item
}
hashVal += stepSize; // add the step
hashVal %= arraySize; // for wraparound
}
return null; // can't find item
} // end delete()
// -------------------------------------------------------------
public DataItem find(int key) // find item with key
// (assumes table not full)
{
int hashVal = hashFunc1(key); // hash the key
int stepSize = hashFunc2(key); // get step size
while(hashArray[hashVal] != null) // until empty cell,
{ // is correct hashVal?
if(hashArray[hashVal].iData == key)
return hashArray[hashVal]; // yes, return item
hashVal += stepSize; // add the step
hashVal %= arraySize; // for wraparound
}
return null; // can't find item
}
// -------------------------------------------------------------
} // end class HashTable
////////////////////////////////////////////////////////////////
class HashDoubleApp
{
public static void main(String[] args) throws IOException
{
int aKey;
DataItem aDataItem;
int size, n;
// get sizes
```

```
putText("Enter size of hash table: ");
size = getInt();
putText("Enter initial number of items: ");
n = getInt();
// make table
HashTable theHashTable = new HashTable(size);
for(int j=0; j<n; j++) // insert data
{
aKey = (int)(java.lang.Math.random() * 2 * size);
aDataItem = new DataItem(aKey);
theHashTable.insert(aKey, aDataItem);
}
while(true) // interact with user
{
putText("Enter first letter of ");
putText("show, insert, delete, or find: ");
char choice = getChar();
switch(choice)
{
case 's':
theHashTable.displayTable();
break;
case 'i':
putText("Enter key value to insert: ");
aKey = getInt();
aDataItem = new DataItem(aKey);
theHashTable.insert(aKey, aDataItem);
break;
case 'd':
putText("Enter key value to delete: ");
aKey = getInt();
theHashTable.delete(aKey);
break;
case 'f':
putText("Enter key value to find: ");
aKey = getInt();
aDataItem = theHashTable.find(aKey);
if(aDataItem != null)
System.out.println("Found " + aKey);
else
System.out.println("Could not find " + aKey);
break;
default:
putText("Invalid entry\n");
} // end switch
} // end while
} // end main()
//-------------------------------------------------------------
public static void putText(String s)
{
System.out.print(s);
System.out.flush();
}
//-------------------------------------------------------------
public static String getString() throws IOException
{
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
```

```
String s = br.readLine();
return s;
}
//-------------------------------------------------------------
public static char getChar() throws IOException
{
String s = getString();
return s.charAt(0);
}
//-------------------------------------------------------------
public static int getInt() throws IOException
{
String s = getString();
return Integer.parseInt(s);
}
//-------------------------------------------------------------
} // end class HashDoubleApp
```

Java Code for Separate Chaining

```
// hashChain.java
// demonstrates hash table with separate chaining
// to run this program: C:>java HashChainApp
import java.io.*; // for I/O
import java.util.*; // for Stack class
import java.lang.Integer; // for parseInt()
////////////////////////////////////////////////////////////////
class Link
{ // (could be other
items)
public int iData; // data item
public Link next; // next link in list
// -------------------------------------------------------------
public Link(int it) // constructor
{ iData= it; }
// -------------------------------------------------------------
public void displayLink() // display this link
{ System.out.print(iData + " "); }
} // end class Link
////////////////////////////////////////////////////////////////
class SortedList
{
private Link first; // ref to first list item
// -------------------------------------------------------------
public void SortedList() // constructor
{ first = null; }
// -------------------------------------------------------------
public void insert(Link theLink) // insert link, in order
{
int key = theLink.iData;
Link previous = null; // start at first
Link current = first;
// until end of list,
while(current != null && key > current.iData)
{ // or current > key,
previous = current;
current = current.next; // go to next item
}
```

```
if(previous==null) // if beginning of list,
first = theLink; // first --> new link
else // not at beginning,
previous.next = theLink; // prev --> new link
theLink.next = current; // new link --> current
} // end insert()
// -------------------------------------------------------------
public void delete(int key) // delete link
{ // (assumes non-emptylist)
Link previous = null; // start at first
Link current = first;
// until end of list,
while(current != null && key != current.iData)
{ // or key == current,
previous = current;
current = current.next; // go to next link
}
// disconnect link
if(previous==null) // if beginning of list
first = first.next; // delete first link
else // not at beginning
previous.next = current.next; // delete current
link
} // end delete()
// -------------------------------------------------------------
public Link find(int key) // find link
{
Link current = first; // start at first
// until end of list,
while(current != null && current.iData <= key)
{ // or key too small,
if(current.iData == key) // is this the link?
return current; // found it, return link
current = current.next; // go to next item
}
return null; // didn't find it
} // end find()
// -------------------------------------------------------------
public void displayList()
{
System.out.print("List (first-->last): ");
Link current = first; // start at beginning of list
while(current != null) // until end of list,
{
current.displayLink(); // print data
current = current.next; // move to next link
}
System.out.println("");
}
} // end class SortedList
////////////////////////////////////////////////////////////////
class HashTable
{
private SortedList[] hashArray; // array of lists
private int arraySize;
// -------------------------------------------------------------
public HashTable(int size) // constructor
{
```

```
arraySize = size;
hashArray = new SortedList[arraySize]; // create array
for(int j=0; j<arraySize; j++) // fill array
hashArray[j] = new SortedList(); // with lists
}
// -------------------------------------------------------------
public void displayTable()
{
for(int j=0; j<arraySize; j++) // for each cell,
{
System.out.print(j + ". "); // display cell number
hashArray[j].displayList(); // display list
}
}
// -------------------------------------------------------------
public int hashFunc(int key) // hash function
{
return key % arraySize;
}
// -------------------------------------------------------------
public void insert(Link theLink) // insert a link
{
int key = theLink.iData;
int hashVal = hashFunc(key); // hash the key
hashArray[hashVal].insert(theLink); // insert at hashVal
} // end insert()
// -------------------------------------------------------------
public void delete(int key) // delete a link
{
int hashVal = hashFunc(key); // hash the key
hashArray[hashVal].delete(key); // delete link
} // end delete()
// -------------------------------------------------------------
public Link find(int key) // find link
{
int hashVal = hashFunc(key); // hash the key
Link theLink = hashArray[hashVal].find(key); // get link
return theLink; // return link
}
// -------------------------------------------------------------
} // end class HashTable
////////////////////////////////////////////////////////////////
class HashChainApp
{
public static void main(String[] args) throws IOException
{
int aKey;
Link aDataItem;
int size, n, keysPerCell = 100;
// get sizes
putText("Enter size of hash table: ");
size = getInt();
putText("Enter initial number of items: ");
n = getInt();
// make table
HashTable theHashTable = new HashTable(size);
for(int j=0; j<n; j++) // insert data
{
```

```
aKey = (int)(java.lang.Math.random() *
keysPerCell * size);
aDataItem = new Link(aKey);
theHashTable.insert(aDataItem);
}
while(true) // interact with user
{
putText("Enter first letter of ");
putText("show, insert, delete, or find: ");
char choice = getChar();
switch(choice)
{
case 's':
theHashTable.displayTable();
break;
case 'i':
putText("Enter key value to insert: ");
aKey = getInt();
aDataItem = new Link(aKey);
theHashTable.insert(aDataItem);
break;
case 'd':
putText("Enter key value to delete: ");
aKey = getInt();
theHashTable.delete(aKey);
break;
case 'f':
putText("Enter key value to find: ");
aKey = getInt();
aDataItem = theHashTable.find(aKey);
if(aDataItem != null)
System.out.println("Found " + aKey);
else
System.out.println("Could not find " + aKey);
break;
default:
putText("Invalid entry\n");
} // end switch
} // end while
} // end main()
//-------------------------------------------------------------
public static void putText(String s)
{
System.out.print(s);
System.out.flush();
}
//-------------------------------------------------------------
public static String getString() throws IOException
{
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);
String s = br.readLine();
return s;
}
//-------------------------------------------------------------
public static char getChar() throws IOException
{
String s = getString();
```

```
return s.charAt(0);
}
//-------------------------------------------------------------
public static int getInt() throws IOException
{
String s = getString();
return Integer.parseInt(s);
}
//-------------------------------------------------------------
} // end class HashChainApp
```

Representing a Graph in a Program

```
class Vertex
{
public char label; // label (e.g. 'A')
public boolean wasVisited;
public Vertex(char lab) // constructor
{
label = lab;
wasVisited = false;
}
} // end class Vertex
```

The Graph Class

```
class Graph
{
private final int MAX_VERTS = 20;
private Vertex vertexList[]; // array of vertices
private int adjMat[][]; // adjacency matrix
private int nVerts; // current number of vertices
// -----------------------------------------------------------
public Graph() // constructor
{
vertexList = new Vertex[MAX_VERTS];
// adjacency matrix
adjMat = new int[MAX_VERTS][MAX_VERTS];
nVerts = 0;
for(int j=0; j<MAX_VERTS; j++) // set adjacency
for(int k=0; k<MAX_VERTS; k++) // matrix to 0
adjMat[j][k] = 0;
} // end constructor
// -----------------------------------------------------------
public void addVertex(char lab) // argument is label
{
vertexList[nVerts++] = new Vertex(lab);
}
// -----------------------------------------------------------
public void addEdge(int start, int end)
{
adjMat[start][end] = 1;
adjMat[end][start] = 1;
}
// -----------------------------------------------------------
public void displayVertex(int v)
{
System.out.print(vertexList[v].label);
}
```

```
// ------------------------------------------------------------
} // end class Graph
```

THE DFS.JAVA PROGRAM

```
// dfs.java
// demonstrates depth-first search
// to run this program: C>java DFSApp
import java.awt.*;
////////////////////////////////////////////////////////////////
class StackX
{
private final int SIZE = 20;
private int[] st;
private int top;
public StackX() // constructor
{
st = new int[SIZE]; // make array
top = -1;
}
public void push(int j) // put item on stack
{ st[++top] = j; }
public int pop() // take item off stack
{ return st[top--]; }
public int peek() // peek at top of stack
{ return st[top]; }
public boolean isEmpty() // true if nothing on stack
{ return (top == -1); }
} // end class StackX
////////////////////////////////////////////////////////////////
class Vertex
{
public char label; // label (e.g. 'A')
public boolean wasVisited;
// ------------------
public Vertex(char lab) // constructor
{
label = lab;
wasVisited = false;
}
// ------------------
} // end class Vertex
////////////////////////////////////////////////////////////////
class Graph
{
private final int MAX_VERTS = 20;
private Vertex vertexList[]; // list of vertices
private int adjMat[][]; // adjacency matrix
private int nVerts; // current number of vertices
private StackX theStack;
// ------------------
public Graph() // constructor
{
vertexList = new Vertex[MAX_VERTS];
// adjacency matrix
adjMat = new int[MAX_VERTS][MAX_VERTS];
nVerts = 0;
for(int j=0; j<MAX_VERTS; j++) // set adjacency
```

48

```
for(int k=0; k<MAX_VERTS; k++) // matrix to 0
adjMat[j][k] = 0;
theStack = new StackX();
} // end constructor
// ------------------
public void addVertex(char lab)
{
vertexList[nVerts++] = new Vertex(lab);
}
// ------------------
public void addEdge(int start, int end)
{
adjMat[start][end] = 1;
adjMat[end][start] = 1;
}
// ------------------
public void displayVertex(int v)
{
System.out.print(vertexList[v].label);
}
// ------------------
public void dfs() // depth-first search
{ // begin at vertex 0
vertexList[0].wasVisited = true; // mark it
displayVertex(0); // display it
theStack.push(0); // push it
while( !theStack.isEmpty() ) // until stack empty,
{
// get an unvisited vertex adjacent to stack top
int v = getAdjUnvisitedVertex( theStack.peek() );
if(v == -1) // if no such vertex,
theStack.pop();
else // if it exists,
{
vertexList[v].wasVisited = true; // mark it
displayVertex(v); // display it
theStack.push(v); // push it
}
} // end while
// stack is empty, so we're done
for(int j=0; j<nVerts; j++) // reset flags
vertexList[j].wasVisited = false;
} // end dfs
// ------------------
// returns an unvisited vertex adj to v
public int getAdjUnvisitedVertex(int v)
{
for(int j=0; j<nVerts; j++)
if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
return j;
return -1;
} // end getAdjUnvisitedVert()
// ------------------
} // end class Graph
////////////////////////////////////////////////////////////////
class DFSApp
{
public static void main(String[] args)
```

```
{
Graph theGraph = new Graph();
theGraph.addVertex('A'); // 0 (start for dfs)
theGraph.addVertex('B'); // 1
theGraph.addVertex('C'); // 2
theGraph.addVertex('D'); // 3
theGraph.addVertex('E'); // 4
theGraph.addEdge(0, 1); // AB
theGraph.addEdge(1, 2); // BC
theGraph.addEdge(0, 3); // AD
theGraph.addEdge(3, 4); // DE
System.out.print("Visits: ");
theGraph.dfs(); // depth-first search
System.out.println();
} // end main()
} // end class DFSApp
////////////////////////////////////////////////////////////////
```

The bfs.java Program

```
// bfs.java
// demonstrates breadth-first search
// to run this program: C>java BFSApp
import java.awt.*;
////////////////////////////////////////////////////////////////
class Queue
{
private final int SIZE = 20;
private int[] queArray;
private int front;
private int rear;
public Queue() // constructor
{
queArray = new int[SIZE];
front = 0;
rear = -1;
}
public void insert(int j) // put item at rear of queue
{
if(rear == SIZE-1)
rear = -1;
queArray[++rear] = j;
}
public int remove() // take item from front of queue
{
int temp = queArray[front++];
if(front == SIZE)
front = 0;
return temp;
}
public boolean isEmpty() // true if queue is empty
{
return ( rear+1==front || (front+SIZE-1==rear) );
}
} // end class Queue
////////////////////////////////////////////////////////////////
class Vertex
{
public char label; // label (e.g. 'A')
```

```
public boolean wasVisited;
// -------------------------------------------------------------
public Vertex(char lab) // constructor
{
label = lab;
wasVisited = false;
}
// -------------------------------------------------------------
} // end class Vertex
////////////////////////////////////////////////////////////////
class Graph
{
private final int MAX_VERTS = 20;
private Vertex vertexList[]; // list of vertices
private int adjMat[][]; // adjacency matrix
private int nVerts; // current number of vertices
private Queue theQueue;
// ------------------
public Graph() // constructor
{
vertexList = new Vertex[MAX_VERTS];
// adjacency matrix
adjMat = new int[MAX_VERTS][MAX_VERTS];
nVerts = 0;
for(int j=0; j<MAX_VERTS; j++) // set adjacency
for(int k=0; k<MAX_VERTS; k++) // matrix to 0
adjMat[j][k] = 0;
theQueue = new Queue();
} // end constructor
// -------------------------------------------------------------
public void addVertex(char lab)
{
vertexList[nVerts++] = new Vertex(lab);
}
// -------------------------------------------------------------
public void addEdge(int start, int end)
{
adjMat[start][end] = 1;
adjMat[end][start] = 1;
}
// -------------------------------------------------------------
public void displayVertex(int v)
{
System.out.print(vertexList[v].label);
}
// -------------------------------------------------------------
public void bfs() // breadth-first search
{ // begin at vertex 0
vertexList[0].wasVisited = true; // mark it
displayVertex(0); // display it
theQueue.insert(0); // insert at tail
int v2;
while( !theQueue.isEmpty() ) // until queue empty,
{
int v1 = theQueue.remove(); // remove vertex at head
// until it has no unvisited neighbors
while( (v2=getAdjUnvisitedVertex(v1)) != -1 )
{ // get one,
```

```
vertexList[v2].wasVisited = true; // mark it
displayVertex(v2); // display it
theQueue.insert(v2); // insert it
} // end while
} // end while(queue not empty)
// queue is empty, so we're done
for(int j=0; j<nVerts; j++) // reset flags
vertexList[j].wasVisited = false;
} // end bfs()
// ------------------------------------------------------------
// returns an unvisited vertex adj to v
public int getAdjUnvisitedVertex(int v)
{
for(int j=0; j<nVerts; j++)
if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
return j;
return -1;
} // end getAdjUnvisitedVert()
// ------------------------------------------------------------
} // end class Graph
////////////////////////////////////////////////////////////////
class BFSApp
{
public static void main(String[] args)
{
Graph theGraph = new Graph();
theGraph.addVertex('A'); // 0 (start for dfs)
theGraph.addVertex('B'); // 1
theGraph.addVertex('C'); // 2
theGraph.addVertex('D'); // 3
theGraph.addVertex('E'); // 4
theGraph.addEdge(0, 1); // AB
theGraph.addEdge(1, 2); // BC
theGraph.addEdge(0, 3); // AD
theGraph.addEdge(3, 4); // DE
System.out.print("Visits: ");
theGraph.bfs(); // breadth-first search
System.out.println();
} // end main()
} // end class BFSApp
////////////////////////////////////////////////////////////////
```

Minimum Spanning Trees

```
// mst.java
// demonstrates minimum spanning tree
// to run this program: C>java MSTApp
import java.awt.*;
////////////////////////////////////////////////////////////////
class StackX
{
private final int SIZE = 20;
private int[] st;
private int top;
public StackX() // constructor
{
st = new int[SIZE]; // make array
top = -1;
```

```
}
public void push(int j) // put item on stack
{ st[++top] = j; }
public int pop() // take item off stack
{ return st[top--]; }
public int peek() // peek at top of stack
{ return st[top]; }
public boolean isEmpty() // true if nothing on stack
{ return (top == -1); }
}
////////////////////////////////////////////////////////////////
class Vertex
{
public char label; // label (e.g. 'A')
public boolean wasVisited;
// ------------------------------------------------------------
public Vertex(char lab) // constructor
{
label = lab;
wasVisited = false;
}
// ------------------------------------------------------------
} // end class Vertex
////////////////////////////////////////////////////////////////
class Graph
{
private final int MAX_VERTS = 20;
private Vertex vertexList[]; // list of vertices
private int adjMat[][]; // adjacency matrix
// current number of vertices
private StackX theStack;
// ------------------------------------------------------------
public Graph() // constructor
{
vertexList = new Vertex[MAX_VERTS];
// adjacency matrix
adjMat = new int[MAX_VERTS][MAX_VERTS];
nVerts = 0;
for(int j=0; j<MAX_VERTS; j++) // set adjacency
for(int k=0; k<MAX_VERTS; k++) // matrix to 0
adjMat[j][k] = 0;
theStack = new StackX();
} // end constructor
// ------------------------------------------------------------
public void addVertex(char lab)
{
vertexList[nVerts++] = new Vertex(lab);
}
// ------------------------------------------------------------
public void addEdge(int start, int end)
{
adjMat[start][end] = 1;
adjMat[end][start] = 1;
}
// ------------------------------------------------------------
public void displayVertex(int v)
{
System.out.print(vertexList[v].label);
```

```
}
// -------------------------------------------------------------
public void mst() // minimum spanning tree (depth first)
{ // start at 0
vertexList[0].wasVisited = true; // mark it
theStack.push(0); // push it
while( !theStack.isEmpty() ) // until stack empty
{ // get stack top
int currentVertex = theStack.peek();
// get next unvisited neighbor
int v = getAdjUnvisitedVertex(currentVertex);
if(v == -1) // if no more
neighbors
theStack.pop(); // pop it away
else // got a neighbor
{
vertexList[v].wasVisited = true; // mark it
theStack.push(v); // push it
// display edge
displayVertex(currentVertex); // from currentV
displayVertex(v); // to v
System.out.print(" ");
}
} // end while(stack not empty)
// stack is empty, so we're done
for(int j=0; j<nVerts; j++) // reset flags
vertexList[j].wasVisited = false;
} // end tree
// -------------------------------------------------------------
// returns an unvisited vertex adj to v
public int getAdjUnvisitedVertex(int v)
{
for(int j=0; j<nVerts; j++)
if(adjMat[v][j]==1 && vertexList[j].wasVisited==false)
return j;
return -1;
} // end getAdjUnvisitedVert()
// -------------------------------------------------------------
} // end class Graph
////////////////////////////////////////////////////////////////
class MSTApp
{
public static void main(String[] args)
{
Graph theGraph = new Graph();
theGraph.addVertex('A'); // 0 (start for mst)
theGraph.addVertex('B'); // 1
theGraph.addVertex('C'); // 2
theGraph.addVertex('D'); // 3
theGraph.addVertex('E'); // 4
theGraph.addEdge(0, 1); // AB
theGraph.addEdge(0, 2); // AC
theGraph.addEdge(0, 3); // AD
theGraph.addEdge(0, 4); // AE
theGraph.addEdge(1, 2); // BC
theGraph.addEdge(1, 3); // BD
theGraph.addEdge(1, 4); // BE
theGraph.addEdge(2, 3); // CD
```

```
theGraph.addEdge(2, 4); // CE
theGraph.addEdge(3, 4); // DE
System.out.print("Minimum spanning tree: ");
theGraph.mst(); // minimum spanning tree
System.out.println();
} // end main()
} // end class MSTApp
////////////////////////////////////////////////////////////////
```

The mstw.java Program

```
// mstw.java
// demonstrates minimum spanning tree with weighted graphs
// to run this program: C>java MSTWApp
import java.awt.*;
////////////////////////////////////////////////////////////////
class Edge
{
public int srcVert; // index of a vertex starting edge
public int destVert; // index of a vertex ending edge
public int distance; // distance from src to dest
public Edge(int sv, int dv, int d) // constructor
{
srcVert = sv;
destVert = dv;
distance = d;
}
} // end class Edge
////////////////////////////////////////////////////////////////
class PriorityQ
{
// array in sorted order, from max at 0 to min at size-1
private final int SIZE = 20;
private Edge[] queArray;
private int size;
public PriorityQ() // constructor
{
queArray = new Edge[SIZE];
size = 0;
}
public void insert(Edge item) // insert item in sorted
order
{
int j;
for(j=0; j<size; j++) // find place to insert
if( item.distance >= queArray[j].distance )
break;
for(int k=size-1; k>=j; k--) // move items up
queArray[k+1] = queArray[k];
queArray[j] = item; // insert item
size++;
}
public Edge removeMin() // remove minimum item
{ return queArray[--size]; }
public void removeN(int n) // remove item at n
{
for(int j=n; j<size-1; j++) // move items down
queArray[j] = queArray[j+1];
```

```
size--;
}
public Edge peekMin() // peek at minimum item
{ return queArray[size-1]; }
public int size() // return number of items
{ return size; }
public boolean isEmpty() // true if queue is empty
{ return (size==0); }
public Edge peekN(int n) // peek at item n
{ return queArray[n]; }
public int find(int findDex) // find item with specified
{ // destVert value
for(int j=0; j<size; j++)
if(queArray[j].destVert == findDex)
return j;
return -1;
}
} // end class PriorityQ
////////////////////////////////////////////////////////////////
class Vertex
{
public char label; // label (e.g. 'A')
public boolean isInTree;
// -------------------------------------------------------------
public Vertex(char lab) // constructor
{
label = lab;
isInTree = false;
}
// -------------------------------------------------------------
} // end class Vertex
////////////////////////////////////////////////////////////////
class Graph
{
private final int MAX_VERTS = 20;
private final int INFINITY = 1000000;
private Vertex vertexList[]; // list of vertices
private int adjMat[][]; // adjacency matrix
private int nVerts; // current number of vertices
private int currentVert;
private PriorityQ thePQ;
private int nTree; // number of verts in tree
// -------------------------------------------------------------
public Graph() // constructor
{
vertexList = new Vertex[MAX_VERTS];
// adjacency matrix
adjMat = new int[MAX_VERTS][MAX_VERTS];
nVerts = 0;
for(int j=0; j<MAX_VERTS; j++) // set adjacency
for(int k=0; k<MAX_VERTS; k++) // matrix to 0
adjMat[j][k] = INFINITY;
thePQ = new PriorityQ();
} // end constructor
// -------------------------------------------------------------
public void addVertex(char lab)
{
vertexList[nVerts++] = new Vertex(lab);
```

```
}
// -------------------------------------------------------------
public void addEdge(int start, int end, int weight)
{
adjMat[start][end] = weight;
adjMat[end][start] = weight;
}
// -------------------------------------------------------------
public void displayVertex(int v)
{
System.out.print(vertexList[v].label);
}
// -------------------------------------------------------------
public void mstw() // minimum spanning tree
{
currentVert = 0; // start at 0
while(nTree < nVerts-1) // while not all verts in tree
{ // put currentVert in tree
vertexList[currentVert].isInTree = true;
nTree++;
// insert edges adjacent to currentVert into PQ
for(int j=0; j<nVerts; j++) // for each vertex,
{
if(j==currentVert) // skip if it's us
continue;
if(vertexList[j].isInTree) // skip if in the tree
continue;
int distance = adjMat[currentVert][j];
if( distance == INFINITY) // skip if no edge
continue;
putInPQ(j, distance); // put it in PQ (maybe)
}
if(thePQ.size()==0) // no vertices in PQ?
{
System.out.println(" GRAPH NOT CONNECTED");
return;
}
// remove edge with minimum distance, from PQ
Edge theEdge = thePQ.removeMin();
int sourceVert = theEdge.srcVert;
currentVert = theEdge.destVert;
// display edge from source to current
System.out.print( vertexList[sourceVert].label );
System.out.print( vertexList[currentVert].label );
System.out.print(" ");
} // end while(not all verts in tree)
// mst is complete
for(int j=0; j<nVerts; j++) // unmark vertices
vertexList[j].isInTree = false;
} // end mstw
// -------------------------------------------------------------
public void putInPQ(int newVert, int newDist)
{
// is there another edge with the same destination
vertex?
int queueIndex = thePQ.find(newVert);
if(queueIndex != -1) // got edge's index
{
```

```
Edge tempEdge = thePQ.peekN(queueIndex); // get edge
int oldDist = tempEdge.distance;
if(oldDist > newDist) // if new edge shorter,
{
thePQ.removeN(queueIndex); // remove old edge
Edge theEdge =
new Edge(currentVert, newVert,
newDist);
thePQ.insert(theEdge); // insert new edge
}
// else no action; just leave the old vertex there
} // end if
else // no edge with same destination vertex
{ // so insert new one
Edge theEdge = new Edge(currentVert, newVert,
newDist);
thePQ.insert(theEdge);
}
} // end putInPQ()
// -------------------------------------------------------------
} // end class Graph
////////////////////////////////////////////////////////////////
class MSTWApp
{
public static void main(String[] args)
{
Graph theGraph = new Graph();
theGraph.addVertex('A'); // 0 (start for mst)
theGraph.addVertex('B'); // 1
theGraph.addVertex('C'); // 2
theGraph.addVertex('D'); // 3
theGraph.addVertex('E'); // 4
theGraph.addVertex('F'); // 5
theGraph.addEdge(0, 1, 6); // AB 6
theGraph.addEdge(0, 3, 4); // AD 4
theGraph.addEdge(1, 2, 10); // BC 10
theGraph.addEdge(1, 3, 7); // BD 7
theGraph.addEdge(1, 4, 7); // BE 7
theGraph.addEdge(2, 3, 8); // CD 8
theGraph.addEdge(2, 4, 5); // CE 5
theGraph.addEdge(2, 5, 6); // CF 6
theGraph.addEdge(3, 4, 12); // DE 12
theGraph.addEdge(4, 5, 7); // EF 7
System.out.print("Minimum spanning tree: ");
theGraph.mstw(); // minimum spanning tree
System.out.println();
} // end main()
} // end class MSTWApp
```

SHORTEST-PATH PROBLEM

```
// path.java
// demonstrates shortest path with weighted, directed graphs
// to run this program: C>java PathApp
import java.awt.*;
////////////////////////////////////////////////////////////////
class DistPar // distance and parent
{ // items stored in sPath array
public int distance; // distance from start to this vertex
```

```
public int parentVert; // current parent of this vertex
public DistPar(int pv, int d) // constructor
{
distance = d;
parentVert = pv;
}
} // end class DistPar
////////////////////////////////////////////////////////////////
class Vertex
{
public char label; // label (e.g. 'A')
public boolean isInTree;
// -------------------------------------------------------------
public Vertex(char lab) // constructor
{
label = lab;
isInTree = false;
}
// -------------------------------------------------------------
} // end class Vertex
////////////////////////////////////////////////////////////////
class Graph
{
private final int MAX_VERTS = 20;
private final int INFINITY = 1000000;
private Vertex vertexList[]; // list of vertices
private int adjMat[][]; // adjacency matrix
private int nVerts; // current number of vertices
private int nTree; // number of verts in tree
private DistPar sPath[]; // array for shortest-path data
private int currentVert; // current vertex
private int startToCurrent; // distance to currentVert
// -------------------------------------------------------------
public Graph() // constructor
{
vertexList = new Vertex[MAX_VERTS];
// adjacency matrix
adjMat = new int[MAX_VERTS][MAX_VERTS];
nVerts = 0;
nTree = 0;
for(int j=0; j<MAX_VERTS; j++) // set adjacency
for(int k=0; k<MAX_VERTS; k++) // matrix
adjMat[j][k] = INFINITY; // to infinity
sPath = new DistPar[MAX_VERTS]; // shortest paths
} // end constructor
// -------------------------------------------------------------
public void addVertex(char lab)
{
vertexList[nVerts++] = new Vertex(lab);
}
// -------------------------------------------------------------
public void addEdge(int start, int end, int weight)
{
adjMat[start][end] = weight; // (directed)
}
// -------------------------------------------------------------
public void path() // find all shortest paths
{
```

59

```
int startTree = 0; // start at vertex 0
vertexList[startTree].isInTree = true;
nTree = 1; // put it in tree
// transfer row of distances from adjMat to sPath
for(int j=0; j<nVerts; j++)
{
int tempDist = adjMat[startTree][j];
sPath[j] = new DistPar(startTree, tempDist);
}
// until all vertices are in the tree
while(nTree < nVerts)
{
int indexMin = getMin(); // get minimum from sPath
int minDist = sPath[indexMin].distance;
if(minDist == INFINITY) // if all infinite
{ // or in tree,
System.out.println("There are unreachable
vertices");
break; // sPath is complete
}
else
{ // reset currentVert
currentVert = indexMin; // to closest vert
startToCurrent = sPath[indexMin].distance;
// minimum distance from startTree is
// to currentVert, and is startToCurrent
}
// put current vertex in tree
vertexList[currentVert].isInTree = true;
nTree++;
adjust_sPath(); // update sPath[] array
} // end while(nTree<nVerts)
displayPaths(); // display sPath[]
contents
nTree = 0; // clear tree
for(int j=0; j<nVerts; j++)
vertexList[j].isInTree = false;
} // end path()
// -------------------------------------------------------------
public int getMin() // get entry from sPath
{ // with minimum
distance
int minDist = INFINITY; // assume minimum
int indexMin = 0;
for(int j=1; j<nVerts; j++) // for each vertex,
{ // if it's in tree and
if( !vertexList[j].isInTree && // smaller than old
one
sPath[j].distance < minDist )
{
minDist = sPath[j].distance;
indexMin = j; // update minimum
}
} // end for
return indexMin; // return index of minimum
} // end getMin()
// -------------------------------------------------------------
public void adjust_sPath()
```

```
{
// adjust values in shortest-path array sPath
int column = 1; // skip starting vertex
while(column < nVerts) // go across columns
{
// if this column's vertex already in tree, skip it
if( vertexList[column].isInTree )
{
column++;
continue;
}
// calculate distance for one sPath entry
// get edge from currentVert to column
int currentToFringe = adjMat[currentVert][column];
// add distance from start
int startToFringe = startToCurrent + currentToFringe;
// get distance of current sPath entry
int sPathDist = sPath[column].distance;
// compare distance from start with sPath entry
if(startToFringe < sPathDist) // if shorter,
{ // update sPath
sPath[column].parentVert = currentVert;
sPath[column].distance = startToFringe;
}
column++;
} // end while(column < nVerts)
} // end adjust_sPath()
// -------------------------------------------------------------
public void displayPaths()
{
for(int j=0; j<nVerts; j++) // display contents of
sPath[]
{
System.out.print(vertexList[j].label + "="); // B=
if(sPath[j].distance == INFINITY)
System.out.print("inf"); // inf
else
System.out.print(sPath[j].distance); // 50
char parent = vertexList[ sPath[j].parentVert ].label;
System.out.print("(" + parent + ") "); // (A)
}
System.out.println("");
}
// -------------------------------------------------------------
} // end class Graph
////////////////////////////////////////////////////////////////
class PathApp
{
public static void main(String[] args)
{
Graph theGraph = new Graph();
theGraph.addVertex('A'); // 0 (start)
theGraph.addVertex('C'); // 2
theGraph.addVertex('B'); // 1
theGraph.addVertex('D'); // 3
theGraph.addVertex('E'); // 4
theGraph.addEdge(0, 1, 50); // AB 50
theGraph.addEdge(0, 3, 80); // AD 80
```

```
theGraph.addEdge(1, 2, 60); // BC 60
theGraph.addEdge(1, 3, 90); // BD 90
theGraph.addEdge(2, 4, 40); // CE 40
theGraph.addEdge(3, 2, 20); // DC 20
theGraph.addEdge(3, 4, 70); // DE 70
theGraph.addEdge(4, 1, 50); // EB 50
System.out.println("Shortest paths");
theGraph.path(); // shortest paths
System.out.println();
} // end main()
} // end class PathApp
////////////////////////////////////////////////////////////////
```