```
/*Alan Stoloff
 * Dr. Benjamin
 * Data Structures and Algorithms
 *
 *     WtGraph − implements the Graph interface
 *               for a weighted graph using a vertex list
 *               and an adjancency matrix.
 */

public class WtGraph implements Graph
{
    public static final int DEF_MAX_GRAPH_SIZE = 100;
    public static final int INFINITE_EDGE_WT = Integer.MAX_VALUE;

    private int size;             // The number of vertices in the graph.
    private Vertex[] vertexList;  // An array containing the grahs vertices.
    private int [][] adjMatrix;   // The adjacency matrix storing the edge weights
                                  //     between the vertices.

    /*
     *    default WtGraph() constructor.
     *       − uses the default maximum graph size.
     */

    public WtGraph()
    {
        setup( DEF_MAX_GRAPH_SIZE );  // Set up vertex list and adjacency matrix.
    }

    /*
     *    WtGraph() constructor.
     *       − accepts an argument to specify the maximum graph size.
     */

    public WtGraph(int maxNumber)
    {
        setup(maxNumber); // Set up the vertex list and adjacency matrix.
    }

    /*
     *    setup() − constructs the initial vertex list and adjacency matrix.
     */

    private void setup(int maxNumber)
    {
        vertexList = new Vertex[ maxNumber ];
        adjMatrix = new int[maxNumber][maxNumber];
        size = 0;

        for (int i =  0; i < maxNumber; i++)         // Set all the matrix entries to
            for (int j = 0; j < maxNumber; j++)      // INFINITE_EDGE_WT to indicate
                adjMatrix[i][j] = INFINITE_EDGE_WT;  // the absence of edges.
    }

```

```
55
56      /*
57       *    insertVertex() − add a new vertex to the graph.
58       */
59
60      public void insertVertex(Vertex newVertex)
61      {
62          vertexList[size] = newVertex;
63          size++;
64      }
65
```

```
65
66
67
68      /*
69       *    insertEdge() − insert a new edge with weight wt between
70       *                   verices with labels v1 and v2.
71       */
72
73      public void insertEdge(String v1, String v2, int wt)
74      {
75          int pos1=index(v1);
76          int pos2=index(v2);
77          for(int i=0;i<size;i++){
78              for(int j=0;j<size;j++){
79                  if(i==pos1&&j==pos2){
80                      adjMatrix[i][j]=wt;
81                  }
82                  if(i==pos2&&j==pos1){
83                      adjMatrix[i][j]=wt;
84                  }
85              }
86          }
87      }
88
89      /*
90       *  retrieveVertex()  − return a pointer to the Vertex with
91       *                      the given label v.  If no such vertex
92       *                      exists, return null.
93       */
94      public Vertex retrieveVertex(String v)
95      {
96          Vertex ptr;  // Return pointer to vertex
97          for(int i=0;i<size;i++){
98              if(vertexList[i].getLabel().equals(v)){
99                  ptr=vertexList[i];
100                 return ptr;
101             }
102         }
103         return null;
104     }
105
```

```
105
106     /*
107      *     edgeWeight() − return the weight of the edge between the vertices
108      *                    with labels v1 and v2.
109      */
110
111     public int edgeWeight(String v1, String v2)
112     {
113         int pos1=index(v1);
114         int pos2=index(v2);
115         return adjMatrix[pos1][pos2];
116     }
117
118     /*
119      *    removeVertex() − remove from the graph the vertex with label v.
120      *                     This involves removing the label from the vertex list
121      *                     as well as removing the edges the vertex was part of.
122      */
123     public void removeVertex(String v)
124     {
125         int ind = index(v);   // Get the vertex's array index.
126
127         if (ind == −1)   // Vertex not in graph.
128             return;
129
130         // Move vertices over to fill the gap in the vertex list.
131
132         for (int i = ind+1; i < size; i++)
133             vertexList[i−1] = vertexList[i];
134
135         // Move entries in the adjacency matrix to fill the gap.
136         for(int i=ind+1;i<size;i++){
137             for(int j=0;j<ind;j++){
138                 adjMatrix[i−1][j]=adjMatrix[i][j];
139             }
140         }
141         for(int i=0;i<ind;i++){
142             for(int j=ind+1;j<size;j++){
143                 adjMatrix[i][j−1]=adjMatrix[i][j];
144             }
145         }
146         for(int i=ind+1;i<size;i++){
147             for(int j=ind+1;j<size;j++){
148                 adjMatrix[i−1][j−1]=adjMatrix[i][j];
149             }
150         }
151               // 1st move the lower left entries up.
152
153               // 2nd move the upper right entries left.
154
155               // 3rd move the lower right entries up and to the left
156
157               // Finally remove unneeded entries from the vertex list and matrix.
158
159         for (int i = 0 ; i < size; i++) {
160             adjMatrix[i][size−1] = INFINITE_EDGE_WT;
161             adjMatrix[size−1][i] = INFINITE_EDGE_WT;
162         }
163
164         vertexList[size−1] = null;
165
166         size = size − 1;  // We have one fewer vertices.
167     }
168
```

```
168
169     /*
170      *   getSize() − return the number of vertices in the graph.
171      */
172
173     public int getSize()
174     {
175         return size;
176     }
177
178
179     /*
180      *   removeEdge() − remove the edge between the
181      *   vertices with labels v1 and v2.
182      */
183
184     public void removeEdge(String v1, String v2)
185     {
186         int index1=index(v1);
187         int index2=index(v2);//do i just replace the spot in adj matrix with −??
188         adjMatrix[index1][index2]=INFINITE_EDGE_WT;
189         adjMatrix[index2][index1]=INFINITE_EDGE_WT;
190     }
191
192     /*
193      *    getVertextList() − return a reference to the array of vertices.
194      */
195
196     public Vertex[] getVertexList()
197     {
198         return vertexList;
199     }
200
201     /*
202      *    clear() − remove all verices and edges from the graph.
203      */
204     public void clear()
205     {
206         for (int i = 0; i < vertexList.length; i++) {
207             vertexList[i] = null;
208             for (int j = 0; j < vertexList.length; j++)
209                 adjMatrix[i][j] = INFINITE_EDGE_WT;
210         }
211     }
212
```

```
213
214     /*
215      *   reset() − set all vertices in the graph to be not visited.
216      */
217
218     public void reset()
219     {
220         for(int i=0;i<size;i++){
221             vertexList[i].reset();
222         }
223     }
224
225     /*
226      * Check for empty or full graphs.
227      */
228
229     public boolean isEmpty() { return (size == 0); }
230     public boolean isFull() { return (size == vertexList.length);}
231
232
233     /*
234      *  showStructure() − display the data structures that implement the
235      *                    graph − this includes the vertexList and the
236      *                    adjacency matrix.
237      */
238
239     public void showStructure()
240     {
241         System.out.println("vertexList");
242         System.out.println("----------");
243         for (int i = 0 ; i < vertexList.length; i++)
244             if (vertexList[i] != null)
245                 System.out.printf("%3d  %s\n", i, vertexList[i].getLabel());
246
247         System.out.printf("\nAdjacency Matrix\n----------------\n");
248         System.out.printf("   |");
249         for (int i = 0; i < size; i++)
250             System.out.printf("%3d |",i);
251         System.out.println();
252
253         System.out.printf("---+");
254         for (int i = 0; i < size; i++)
255             System.out.printf("----+",i);
256         System.out.println();
257
258         for (int i = 0; i < size; i++) {
259             System.out.printf("%3d|",i);
260             for (int j = 0; j < size; j++)
261                 if (adjMatrix[i][j] == INFINITE_EDGE_WT)
262                     System.out.printf("  − |");
263                 else
264                     System.out.printf("%3d |", adjMatrix[i][j]);
265             System.out.println();
266         }
267     }
268
```

```
268
269     /*
270      *  neighbors() − return an array of vertices that
271      *                are neighbors of vertex V.
272      */
273
274     public Vertex[] neighbors(Vertex v)
275     {
276         int count = 0;  // Counts the number of v's neighbors
277         int pos=index(v.getLabel());
278         for(int i=0;i<size;i++){
279             if(adjMatrix[pos][i]!=INFINITE_EDGE_WT){
280                 count+=1;
281             }
282         }
283
284         // Count the number of v's neighbors.
285
286         // Construct an array to store the neighbors.
287         //
288         Vertex [] neighbor = new Vertex[count];
289         int neighborlistcount=0;
290         for(int i=0;i<size;i++){
291             if(adjMatrix[pos][i]!=INFINITE_EDGE_WT){
292                 neighbor[neighborlistcount]=vertexList[i];
293                 neighborlistcount+=1;
294             }
295         }
296         // Populate the array with the neighbors
297
298         return neighbor;
299     }
300
301
302
303     /*
304      *   depthFirst − print out the order in which the vertices
305      *                of a graph will be traversed in a depthFirst
306      *                traversal from a given starting node.
307      */
308
309     public static void depthFirst(String startV, WtGraph g)
310     {
311         Stack<Vertex> S = new Stack<Vertex>();
312         g.reset();
313         Vertex [] vList=g.getVertexList();
314         S.push(vList[g.index(startV)]);
315         while(!S.isEmpty()){
316             Vertex u=S.pop();
317             if(u.visit()==false){
318                 u.visit();
319                 System.out.println(u.getLabel());
320                 Vertex [] neighbors=g.neighbors(u);
321                 for(int i=0;i<neighbors.length;i++){
322                     S.push(neighbors[i]);
323                 }
324             }
325         }
326     }
327
328
329     /*
330      *   breadthFirst − print out the order in which the vertices
331      *                  of a graph will be traversed in a breadthFirst
332      *                  traversal from a given starting node.
333      */
334
335     public static void breadthFirst(String startV, WtGraph g)
336     {
```

```
337         Queue<Vertex> Q = new Queue<Vertex>();
338         g.reset();
339         Vertex [] vList=g.getVertexList();
340         Q.enqueue(vList[g.index(startV)]);
341         while(!Q.isEmpty()){
342             Vertex u=Q.dequeue();
343             if(u.visit()==false){
344                 u.visit();
345                 System.out.println(u.getLabel());
346                 Vertex [] neighbors=g.neighbors(u);
347                 for(int i=0;i<neighbors.length;i++){
348                     Q.enqueue(neighbors[i]);
349                 }
350             }
351         }
352     }
353 
354     /*
355      *    index() − return the vertexList index of the vertex with
356      *              label v.  If the vertex doesn't exist return −1.
357      */
358 
359     public int index(String v)
360     {
361         int ind;
362         for(int i=0;i<size;i++){
363             if(vertexList[i].getLabel().equals(v)){
364                 return i;
365             }
366         }
367         return −1; //−1 if the label is not found
368     }
369 
370     /*
371      *   getEdge() − get the edge weight of the edge between the vertices
372      *               with array indices row and col.
373      */
374 
375     public int getEdge(int row, int col)
376     {
377         return adjMatrix[row][col];
378     }
379 
380     /*
381      *   setEdge() − set the weight of the edge between vertices with
382      *               array indices row and col to wt.
383      */
384 
385     private void setEdge(int row, int col, int wt)
386     {
387         adjMatrix[row][col]=wt;
388     }
389 
390 
391 }
```