Chapter 13-1, Lecture Notes

Graphs

13.1 Terminology

1. You are undoubtedly familiar with graphs: Line graphs, bar graphs, and pie charts are in common use.
   1.1 The simple line graph in Figure 13-1 is an example of the type of graph that this chapter considers: a set of points that are joined by lines.
      1.1.1 Graphs provide a way to illustrate data.
      1.1.2 But graphs also represent the relationships among data items, and it is this feature of graphs that is important here.

   1.2 A **graph** G consists of two sets: a set V of vertices, or nodes, and a set E of edges that connect the vertices.
      1.2.1 For example, the campus map in Figure 13-2 is a graph whose vertices represent buildings and whose edges represent the sidewalks between the buildings.
      1.2.2 This definition of a graph is more genera! than the definition of a line graph.
      1.2.3 In fact, a line graph, with its points and lines, is a special case of the general definition of a graph.

   1.3 A **subgraph** consists of a subset of a graph's vertices and a subset of its edges.
      1.3.1 Figure 13-2b shows a sub graph of the graph in Figure 13-2a.
      1.3.2 Two vertices of a graph are adjacent if they are joined by an edge.
      1.3.3 In Figure 13-2b, the Library and the Student Union are adjacent.
         1.3.3.1 A path between two vertices is a sequence of edges that begins at one vertex and ends at another vertex.
         1.3.3.2 For example, there is a path in Figure 13-2a that begins at the Dormitory, leads first to the Library, then to the Student Union, and finally back to the Library.
         1.3.3.3 Though a path may pass through the same vertex more than once, as the path just described does, a **simple path** may not.
         1.3.3.4 The path Dormitory-Library-Student Union is a simple path.
         1.3.3.5 A **cycle** is a path that begins and ends at the same vertex; a **simple cycle** is a cycle that does not pass through other vertices more than once.
         1.3.3.6 The path Library-Student Union-Gymnasium-Dormitory-Library is a simple cycle in the graph in Figure 13-2a.
         1.3.3.7 A graph is connected if each pair of distinct vertices has a path between them.
         1.3.3.8 That is, in a connected graph you can get from any vertex to any other vertex by following a path.

1.3.3.9 Figure 13-3a shows a connected graph.
1.3.3.10 Notice that a connected graph does not necessarily have an edge between every pair of vertices.
1.3.3.11 Figure 13-3b shows a **disconnected graph**.

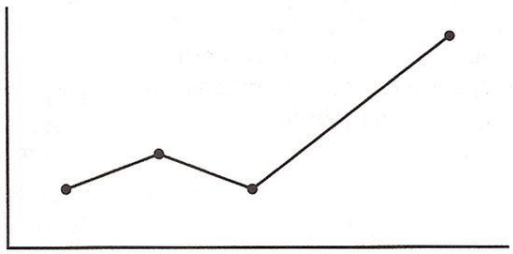

FIGURE 13-1
An ordinary line graph

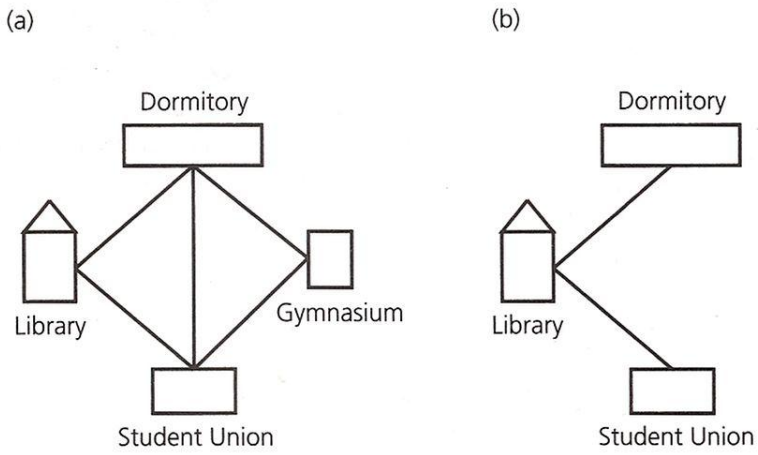

FIGURE 13-2
(a) A campus map as a graph; (b) a subgraph

1.4 In a **complete graph**, each pair of distinct vertices has an edge between them.
1.4.1 The graph in Figure 13-3c is complete.
1.4.2 Clearly, a complete graph is also connected, but the converse is not true; notice that the graph in Figure 13-3a is connected but is not complete.

1.5 Because a graph has a set of edges, a graph cannot have duplicate edges between vertices. However, a **multigraph**, as illustrated in Figure 13-4a, does allow multiple edges.
1.5.1 A graph's edges cannot begin and end at the same vertex.
1.5.2 Figure 13-4b shows such an edge, which is called a **self** edge, or loop.

1.6 You can label the edges of a graph.
1.6.1 When these labels represent numeric values, the graph is called a weighted graph.

1.6.2 The graph in Figure 13-5a is a weighted graph whose edges are labeled with the distances between cities.

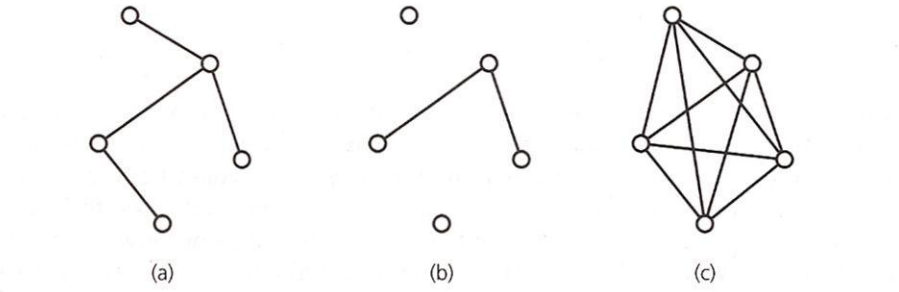

FIGURE 13-3

Graphs that are (a) connected; (b) disconnected; and (c) complete

1.7 All of the previous graphs are examples of undirected graphs, because the edges do not indicate a direction.
- 1.7.1 That is, you can travel in either direction along the edges between the vertices of an undirected graph.
- 1.7.2 In contrast, each edge in a directed graph, or digraph, has a direction and is called a directed edge.
- 1.7.3 Although each distinct pair of vertices in an undirected graph has only one edge between them, a directed graph can have two edges between a parr of vertices, one in each direction.
- 1.7.4 For example, the airline flight map in Figure 13-5b is a directed graph.
- 1.7.5 There are flights in both directions between Providence and New York, but, although there is a flight from San Francisco Albuquerque, there is no flight from Albuquerque to San Francisco.
- 1.7.6 You can convert an undirected graph to a directed graph by replacing each edge with two edges that point in opposite directions.

1.8 The definitions just given for undirected graphs apply also to directed graphs, with changes that account for direction.
- 1.8.1 For example, a directed path is a sequence of directed edges between two vertices, such as the directed path in Figure 13-5b that begins in Providence, goes to New York, and ends in San Francisco.
- 1.8.2 But the definition of adjacent vertices is not quite as obvious for a digraph.
- 1.8.3 If there is a directed edge from vertex x to vertex y, then y is adjacent to x.
- 1.8.4 (Alternatively, y is a successor of x, and x is a predecessor of y.)
- 1.8.5 It does not necessarily follow, however, that x is adjacent to y.
- 1.8.6 Thus, in Figure 13-5b, Albuquerque is adjacent to San Francisco, but San Francisco is not adjacent to Albuquerque.

13.2 Graphs as ADTs

2 You can treat graphs as abstract data types.

2.1 Insertion and deletion operations are somewhat different for graphs than for other ADTs that you have studied, in that they apply to either vertices or edges.

2.1.1 You can define the ADT graph so that its vertices either do or do not contain values.
2.1.2 A graph whose vertices do not contain values represents only the relationships among vertices.
2.1.3 Such graphs are not unusual, because many problems have no need for vertex values.
2.1.4 But the following ADT graph operations do assume that the graph's vertices contain values .

ADT Graph Operations

1. Create an empty graph.

2. Destroy a graph.

3. Determine whether a graph is empty.

4. Determine the number of vertices in a graph.

5. Determine the number of edges in a graph.

6. Determine whether an edge exists between two given vertices.

7. Insert a vertex in a graph whose vertices have distinct search keys that differ from the new vertex's search key.

8. Insert an edge between two given vertices in a graph.

9. Delete a particular vertex from a graph and any edges between the vertex and other vertices.

10. Delete the edge between two given vertices in a graph.

11. Retrieve from a graph the vertex that contains a given search key.

2.2 Several variations of this ADT are possible.
2.2.1 For example, if the graph is directed, you can replace occurrences of "edges" in the previous operations with "directed edges."
2.2.2 You can also add traversal operations to the ADT. Graph-traversal algorithms are discussed in the section "Graph Traversals."

Implementing Graphs

2.3 The two most common implementations of a graph are the adjacency matrix and the adjacency list.

2.3.1 An **adjacency matrix** for a graph with n vertices numbered 0,1, ... , n - 1 is an n by n array matrix such that matrix[i] [j] is 1 (true) if there is an edge from vertex i to vertex j, and 0 (false) otherwise.
2.3.2 Figure 13-6 shows a directed graph and its adjacency matrix.
2.3.3 Notice that the diagonal entries matrix[i] [i] are 0, although sometimes it can be useful to set these entries to 1.
2.3.4 You should choose the value that is most convenient for your application.

2.4 When the graph is weighted, you can let matrix[i] [j] be the weight that labels the edge from vertex i to vertex j, instead of simply 1, and let rnatrix[i] [j] equal ∞ instead of 0 when there is no edge from vertex i to vertex J.
2.4.1 For example, Figure 13-7 shows a weighted undirected graph and its adjacency matrix.
2.4.2 Notice that the adjacency matrix for an undirected graph symmetrical; that is, matrix[i] [j] equals matrix[j] [i].

2.5 Our definition of an adjacency matrix does not mention the value, if any. in a vertex.
2.5.1 If you need to associate values with vertices, you can use a second array, values, to represent the n vertex values.
2.5.2 The array values is one dimensional, and values [i] is the value in vertex i.

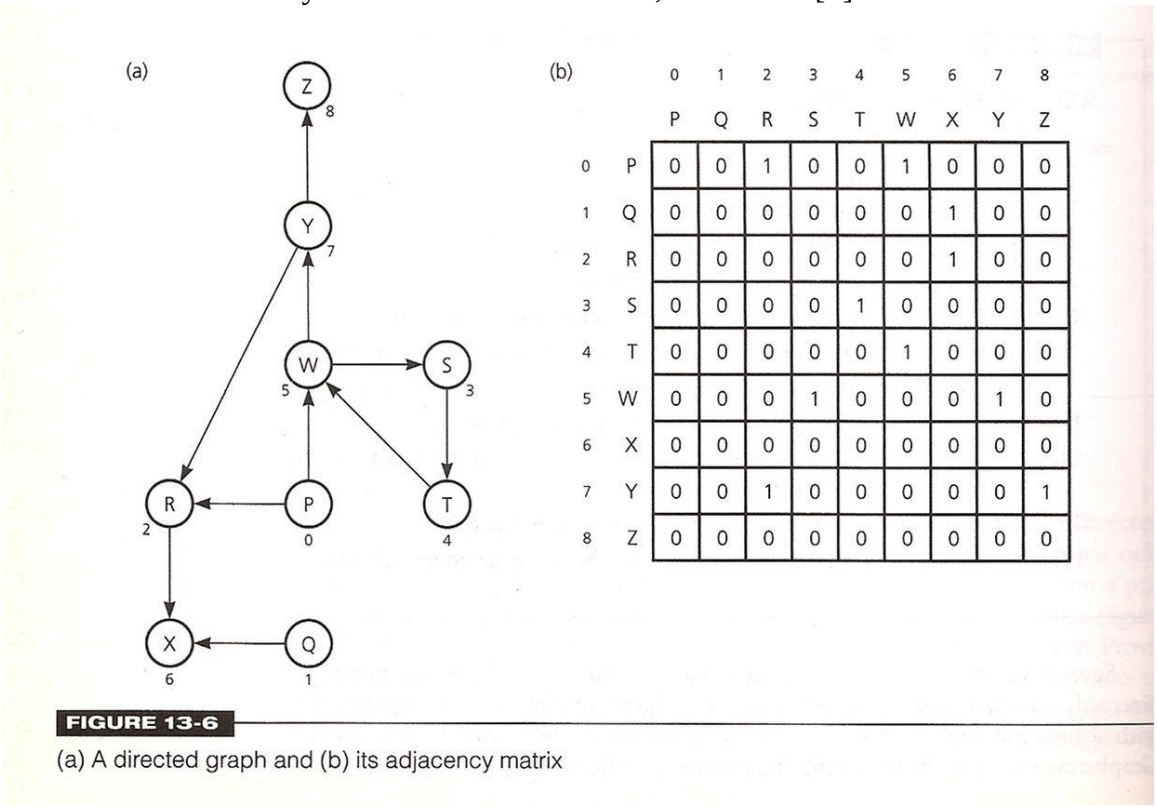

FIGURE 13-6
(a) A directed graph and (b) its adjacency matrix

2.6 An adjacency list for a graph with n vertices numbered 0, 1, ... , n - 1 consists of n linked lists.
2.6.1 The i^{th} linked list has a node for vertex j if and only if the graph contains an edge from vertex i to vertex i

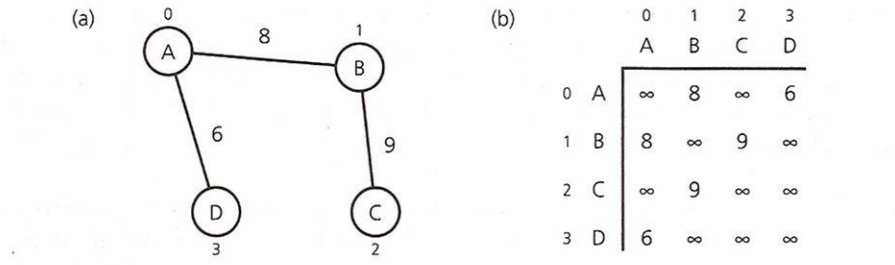

FIGURE 13-7

(a) A weighted undirected graph and (b) its adjacency matrix

2.6.2 This node can contain the vertex j's value, if any.

2.6.3 If the vertex has no value, the node needs to contain some indication of the vertex's identity.

2.6.4 Figure 13-8 shows a directed graph and its adjacency list. You can see, for example, that vertex 0 (P) has edges to vertex 2 (R) and vertex 5 (W).

2.6.5 Thus, the first linked list in the adjacency list contains nodes for Rand W.

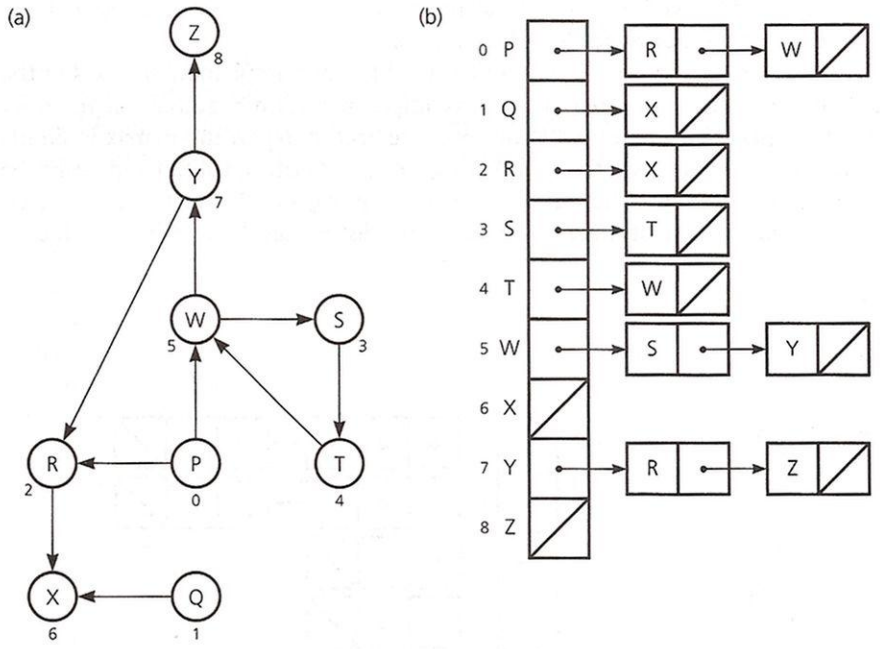

FIGURE 13-8

(a) A directed graph and (b) its adjacency list

2.7 Figure 13-9 shows an undirected graph and its adjacency list.

2.7.1 The adjacency list for an undirected graph treats each edge as if it were two directed edges in opposite directions.

2.7.2 Thus, the edge between A and B in Figure 13-9a appears as edges from A to B and from B to A in Figure 13-9b.

2.7.3 The graph in 13-9a happens to be weighted; you can include the edge weights in the nodes of the adjacency list, given in Figure 13-9b.

2.7.4 Which of these two implementations of a graph-the adjacency matrix or the adjacency list-is better?

2.7.4.1 The answer depends on how your particular application uses the graph.

2.7.4.2 For example, the two most commonly performed graph operations are

1. Determine whether there is an edge from vertex i to vertex j

2. Find all vertices adjacent to a given vertex i

2.7.4.3 The adjacency matrix supports the first operation somewhat more efficiently than does the adjacency list.

2.7.4.3.1 To determine whether there is an edge from i to j by using an adjacency matrix, you need only examine the value of matrix[i] [j].

2.7.4.3.2 If you use an adjacency list, however, you must traverse the ith linked list to determine whether a vertex corresponding to vertex j is present.

2.7.4.4 The second operation, on the other hand, is supported more efficiently by on 2 the adjacency list.

2.7.4.4.1 To determine all vertices adjacent to a given vertex i, giver the adjacency matrix, you must traverse the i^{th} row of the array; however given the adjacency list, you need only traverse the i^{th} linked list.

2.7.4.4.2 For a graph with n vertices, the i^{th} row of the adjacency matrix always has n entrie5 whereas the i^{th} linked list has only as many nodes as there are vertices adjacent to vertex i, a number typically far less than n.

2.7.5 Consider now the space requirements of the two implementations.

2.7.5.1 On the surface it might appear that the matrix implementation requires less memory than the linked list implementation, because each entry in the matrix is simply an integer, whereas each linked list node contains both a value to identify the vertex and a pointer.

2.7.5.1.1 The adjacency matrix, however, always has n 2 entries, whereas the number of nodes in an adjacency list equals the number of edges a directed graph or twice that number for an undirected graph.

2.7.5.1.2 Even though the adjacency list also has n head pointers, it often requires less storage than an adjacency matrix.

2.7.5.2 Thus, when choosing a graph implementation for a particular application, you must consider such factors as what operations you

will perform most frequently on the graph and the number of edges that the graph is likely to contain.

2.7.5.2.1 For example, Chapter 6 presented the HP Air problem, which was to determine whether an airline provided a sequence of flights from an origin city to a destination city.

2.7.5.2.2 The flight map for that problem is in fact a directed graph and appeared earlier in this chapter in Figure 13-8a.

2.7.5.2.3 Figures 13-6b and 13-8b show, respectively, the adjacency matrix and adjacency list for this graph.

2.7.5.2.4 Because the most frequent operation was to find all cities (vertices) adjacent to a given city (vertex), the adjacency list would be the more efficient implementation of the flight map.

2.7.5.2.5 The adjacency list also requires less storage than the adjacency matrix, which you can demonstrate as an exercise.

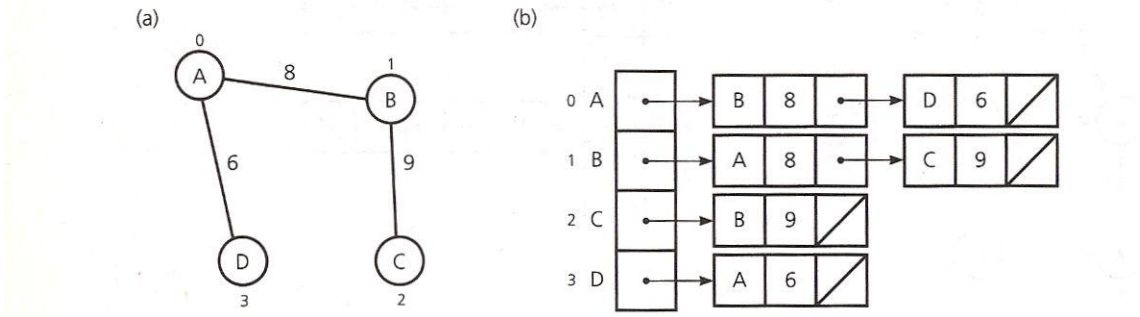

FIGURE 13-9

(a) A weighted undirected graph and (b) its adjacency list

Implementing a Graph Class Using the STL

2.8 Some C++ STL implementations provide a graph class; however, a graph class is not included as part of the STL.

2.8.1 Graph classes and their accompanying algorithms can be implemented in many different ways.

2.8.2 The Graph class in this section is an undirected, weighted graph. It is implemented with an adjacency list, which consists of a vector of maps.

2.8.3 The vector elements represent the vertices of a graph.

2.8.4 The map for each vertex contains element pairs, which consist of an adjacent vertex and an edge weight.

2.8.5 The number of vertices in the graph is determined by an integer argument passed to the constructor.

2.8.6 An Edge class holds both vertices of an edge, as well as the edge weight. The client application adds an edge to the graph by passing an Edge object to the add method.

2.8.7 The following files contain an implementation for a graph.

```
// @file Edge.h

// An Edge class for graph implementations.
class Edge
{

public:

   int v, w, weight;

   Edge(int firstVertex, int secondVertex, int edgeWeight)
   {
      v = firstVertex;
      w = secondVertex; weight = edgeWeight;
    } // end constructor
}; // end Edge
// End of header file

/** @file Graph.h */

#include <vector>
#include <list>
#include <map>
#include "Edge.h"
using namespace std;

/** An adjacency list representation of an undirected,

   * weighted graph. */

class Graph
{
public:

   /** Number of vertices in the graph. */
   int numVertices;
   /** Number of edges in the graph. */
   int numEdges;

   /** Adjacency list representation of the graph;
      * the map pair consists of the second vertex (key)
      * and the edge weight (value). */

   vector<map<int, int> > adjList;
```

```
    /** Constructor.
      * @pre The graph is empty.
      * @post The graph is initialized to hold n vertices. */
    Graph(int n);

  /** Determines the number of vertices in the graph.
    * @pre None.
    * @post None.
    * @return The number of vertices in the graph. */
  int getNumVertices() const;

  /** Determines the number of edges in the graph.
    * @pre None.
    * @post None.
    * @return The number of edges in the graph. */
  int getNumEdges() const;

  /** Determines the weight of an edge.
    * @pre The edge exists in the graph.
    * @post None.
    * @return The weight of the edge parameter. */
  int getWeight(Edge e) const;

  /** Creates an edge in the graph.
    * @pre The vertices exist in the graph.
    * @post Adds to both v and w's list. */
  void add(Edge e);

  /** Removes an edge from the graph.
    * @pre The vertices exist in the graph.
    * @post Removes edges from both v and w's list. */
  void remove(Edge e);

  /** Finds the edge connecting v and w.
    * @pre The edge exists.
    * @post None.
    * @return An iterator to map key w in vector[vl. */
  map<int, int>::iterator findEdge(int v, int w);
}; // end Graph
// End of header file


/** @file Graph.cpp
  * An adjacency list representation of an undirected,
  * weighted graph. */
```

```
#include "Graph.h"

Graph::Graph(int n)
{
   map<int, int> element;
   adjList.assign(n, element);
   numVertices = n;
} // end constructor

int Graph::getNumVertices() const
{
   return numVertices;
} // end getNumVertices

int Graph::getNumEdges() const
{
   return numEdges;
} // end getNumEdges

int Graph::getWeight(Edge e) const
{
   return e.weight;
} // end getWeight

void Graph::add(Edge e)
{
   int v = e.v,
       w = e.w,
   weight = e.weight;
   adjList[v].insert(make_pair(w, weight));
   adjList[w].insert(make_pair(v, weight));
   numEdges++;
} // end add

void Graph::remove(Edge e)
{
   int v = e.v,
       w = e.w,
       weight = e.weight;

   adjList[e.v].erase(w);
   adjList[e.w].erase(v);
   numEdges--;
} // end remove
```

```
map<int, int>::iterator Graph::findEdge(int v, int w)
{
   map<int, int> m = adjList[v];
   map<int, int>::iterator iter = m.find(w);

   return iter;
} // end findEdge
```

2.9 The programming problems at the end of the chapter ask you to add exception handling to the Graph class, modify the class to represent a directed graph, and to rewrite the Graph class as a template.