CHAPTER 24

Graphs and Applications

Objectives

- To model real-world problems using graphs and explain the Seven Bridges of Königsberg problem (§24.1).
- To describe graph terminologies: vertices, edges, simple graphs, weighted/unweighted graphs, and directed/undirected graphs (§24.2).
- To represent vertices and edges using lists, adjacent matrices, and adjacent lists (§24.3).
- To model graphs using the **`Graph`** class (§24.4).
- To represent the traversal of a graph using the **`Tree`** class (§24.5).
- To design and implement depth-first search (§24.6).
- To design and implement breadth-first search (§24.7).
- To solve the nine-tail problem using breadth-first search (§24.8).

24.1 Introduction

Key Point: *Many real-world problems can be solved using graph algorithms.*

Graphs play an important role in modeling real-world problems. For example, the problem to find a shortest path between two cities can be modeled using a graph, where the vertices represent cities and the edges represent the roads and distances between two adjacent cities, as shown in Figure 24.1. The problem of finding a shortest path between two cities is reduced to finding a shortest path between two vertices in a graph.

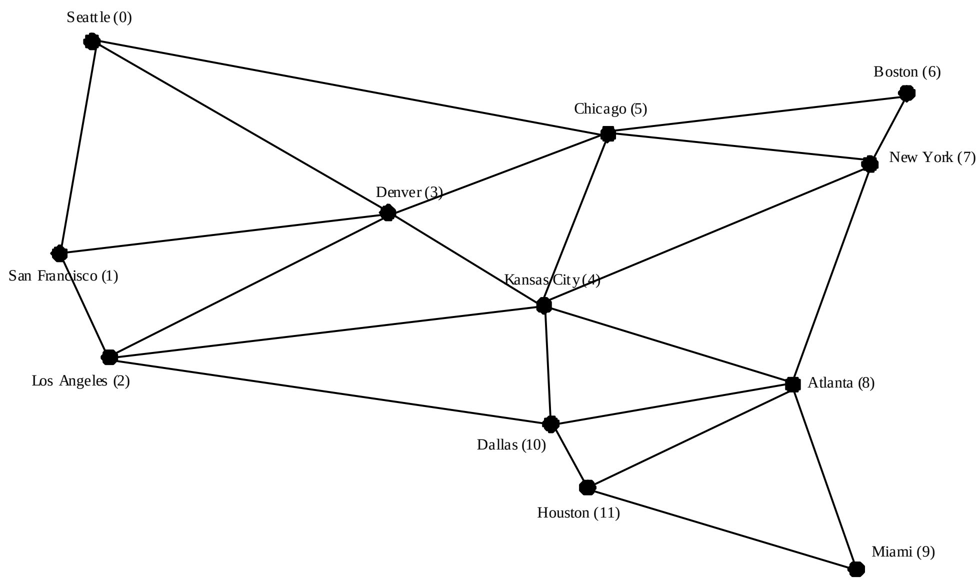
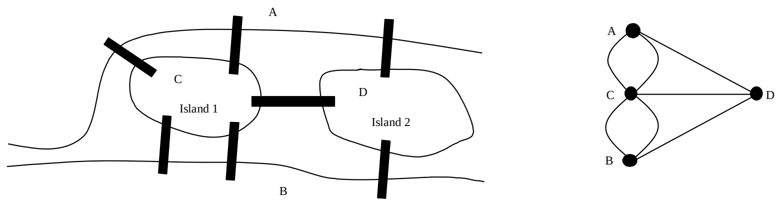


Figure 24.1

A graph can be used to model the distance between the cities.

The study of graph problems is known as *graph theory*. Graph theory was founded by Leonard Euler in 1736, when he introduced graph terminology to solve the famous *Seven Bridges of Königsberg* problem. The city of Königsberg, Prussia, (now Kaliningrad, Russia) was divided by the Pregel River. There were two islands on the river. The city and islands were connected by seven bridges, as shown in Figure 24.2a. The question is, can one take a walk, cross each bridge exactly once, and return to the starting point? Euler proved that it was not possible.



(a) Seven bridges sketch (b) Graph model

Figure 24.2

Seven bridges connecting islands and land.

To establish a proof, Euler first abstracted the Königsberg city map into the sketch shown in Figure 24.2a, by eliminating all streets. Second, he replaced each land mass with a dot, called a vertex or a node, and each bridge with a line, called an edge, as shown in Figure 24.2b. This structure with vertices and edges is called a graph.

Looking at the graph, we ask whether there is a path starting from any vertex, traversing all edges exactly once, and returning to the starting vertex. Euler proved that for such path to exist, each vertex must have an even number of edges. Therefore, the *Seven Bridges of Königsberg* problem has no solution.

Graph problems are often solved using algorithms. Graph algorithms have many applications in various areas, such as in computer science, mathematics, biology, engineering, economics, genetics, and social sciences. This chapter presents the algorithms for depth-first search and breadth-first search, and their applications. The next chapter presents the algorithms for finding a minimum spanning tree and shortest paths in weighted graphs, and their applications.

24.2 Basic Graph Terminologies

Key Point: *A graph consists of vertices, and edges that connect the vertices.*

This chapter does not assume that the reader has prior knowledge of graph theory or discrete mathematics. We use plain and simple terms to define graphs.

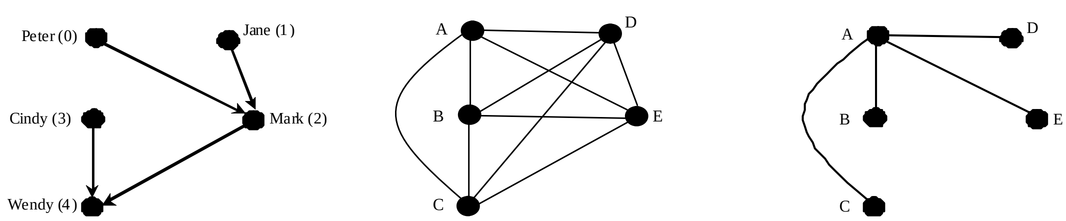
What is a graph? A *graph* is a mathematical structure that represents relationships among entities in the real world. For example, the graph is Figure 24.1 represents the roads and their distances among cities, and the graph in Figure 24.2b represents the bridges among land masses.

A graph consists of a nonempty set of vertices, nodes, or points, and a set of edges that connect the vertices. For convenience, we define a graph as $G = (V, E)$, where V represents a set of vertices and E a set of edges. For example, V and E for the graph in Figure 24.1 are as follows:

```
V = {"Seattle", "San Francisco", "Los Angeles",
  "Denver", "Kansas City", "Chicago", "Boston", "New York",
  "Atlanta", "Miami", "Dallas", "Houston"};

E = {{"Seattle", "San Francisco"}, {"Seattle", "Chicago"},
     {"Seattle", "Denver"}, {"San Francisco", "Denver"},
       ...
    };
```

A graph may be directed or undirected. In a directed graph, each edge has a direction, which indicates that you can move from one vertex to the other through the edge. You may model parent/child relationships using a directed graph, where an edge from vertex *A* to *B* indicates that *A* is a parent of *B*.

Figure 24.3a shows a directed graph. In an undirected graph, you can move in both directions between vertices. The graph in Figure 24.1 is undirected.



(a) A directed graph (b) A complete graph (c) A subgraph of the graph in (b)

Figure 24.3

Graphs may appear in many forms.

Edges may be weighted or unweighted. For example, each edge in the graph in Figure 24.1 has a weight that represents the distance between two cities.

Two vertices in a graph are said to be *adjacent* if they are connected by the same edge. Similarly two edges are said to be *adjacent* if they are connected to the same vertex. An edge in a graph that joins two vertices is said to be *incident* to both vertices. The *degree* of a vertex is the number of edges incident to it.

Two vertices are called *neighbors* if they are adjacent. Similarly two edges are called *neighbors* if they are incident.

A *loop* is an edge that links a vertex to itself. If two vertices are connected by two or more edges, these edges are called *parallel edges*. A simple graph is one that has no loops and parallel edges. A complete graph is one in which every two pairs of vertices are connected, as shown in Figure 24.3b.

A graph is *connected* if there exists a path between any two vertices in the graph. A *subgraph* of a graph *G* is a graph whose vertex set is a subset of that of *G* and whose edge set is a subset of that of *G*. For example, the graph in Figure 24.3c is a subgraph of the graph in Figure 24.3b.

Assume that the graph is connected and undirected. A *spanning tree* of a graph is a subgraph that is a tree and connects all vertices in the graph.

PEDAGOGICAL NOTE

Before we introduce graph algorithms and applications, it is helpful to get acquainted with graphs using the interactive tool at www.cs.armstrong.edu/liang/animation/GraphLearningTool.html, as shown in Figure 24.4. The tool allows you to add/remove/move vertices and draw edges using mouse gestures. You can also find depth-first search (DFS) trees and breadth-first search (BFS) trees, and the shortest path between two vertices.

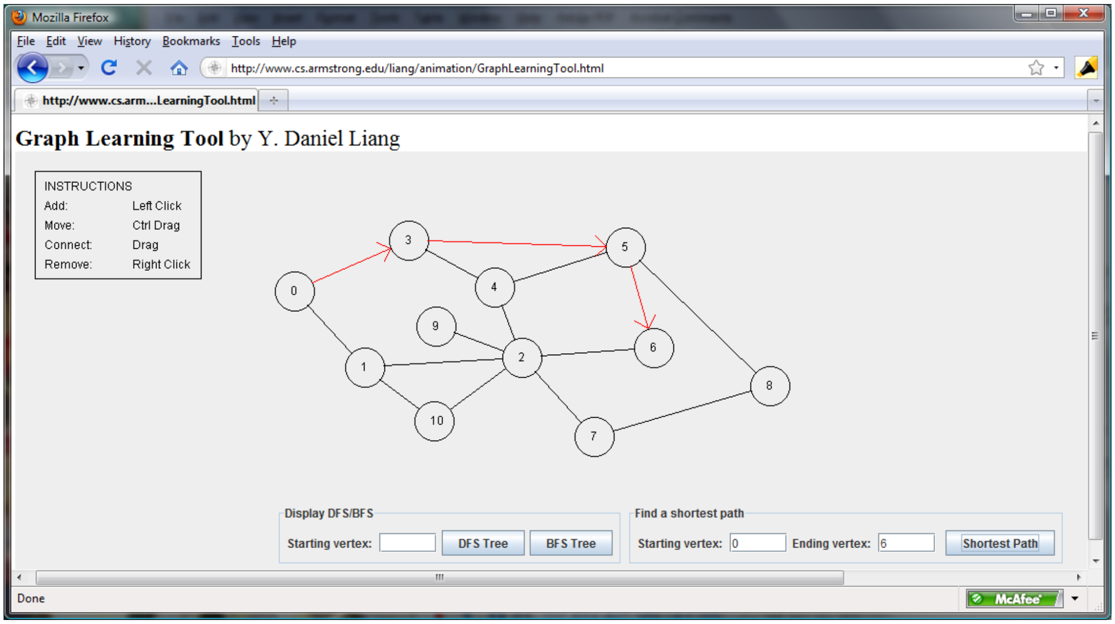


Figure 24.4

You can use the tool to create a graph with mouse gestures and show DFS/BFS trees and shortest paths.

Check point

24.1 What is the famous *Seven Bridges of Königsberg* problem?

24.2 What is a graph? Explain the following terms: undirected graph, directed graph, weighted graph, degree of a vertex, parallel edge, simple graph, complete graph, connected graph, cycle, subgraph, tree, and spanning tree.

24.3 How many edges are in a complete graph with 5 vertices? How many edges are in a tree of 5 vertices?

24.4 How many edges are in a complete graph with *n* vertices? How many edges are in a tree of *n* vertices?

24.3 Representing Graphs

Key Point: *Representing a graph is to store its vertices and edges in a program. The data structure for storing a graph is arrays or lists.*

To write a program that processes and manipulates graphs, you have to store or represent graphs in the computer.

24.3.1 Representing Vertices

The vertices can be stored in an array. For example, you can store all the city names in the graph in Figure 24.1 using the following array:

```
string vertices[] = {"Seattle", "San Francisco", "Los Angeles",
  "Denver", "Kansas City", "Chicago", "Boston", "New York",
  "Atlanta", "Miami", "Dallas", "Houston"};
```

NOTE:

The vertices can be objects of any type. For example, you may consider cities as objects that contain the information such as name, population, and mayor. So, you may define vertices as follows:

```
City city0("Seattle", 563374, "Greg Nickels");
...
City city11("Houston", 1000203, "Bill White");
City vertices[] = {city0, city1, ..., city11};

class City
{
public:
  City(string& cityName, int population, string& mayor)
  {
    this->cityName = cityName;
    this->population = population;
    this->mayor = mayor;
  }

  string getCityName() const
  {
    return cityName;
  }

  int getPopulation() const
  {
    return population;
  }

  string getMayor() const
  {
    return mayor;
  }

  void setMayor(string& mayor)
  {
    this->mayor = mayor;
  }
```

```
    void setPopulation(int population)
    {
      this->population = population;
    }

  private:
    string cityName;
    int population;
    string mayor;
  };
```

The vertices can be conveniently labeled using natural numbers `0`, `1`, `2`, ..., `n - 1`, for a graphs of *n* vertices. So, `vertices[0]` represents `"Seattle"`, `vertices[1]` represents `"San Francisco"`, and so on, as shown in Figure 24.4.

vertices[0]	Seattle
vertices[1]	San Francisco
vertices[2]	Los Angeles
vertices[3]	Denver
vertices[4]	Kansas City
vertices[5]	Chicago
vertices[6]	Boston
vertices[7]	New York
vertices[8]	Atlanta
vertices[9]	Miami
vertices[10]	Dallas
vertices[11]	Houston

Figure 24.4

An array stores the vertex names.

NOTE:

You can reference a vertex by its name or its index, whichever is convenient.

Obviously, it is easy to access a vertex via its index in a program.

24.3.2 Representing Edges (for input): Edge Array

The edges can be represented using a two-dimensional array. For example, you can store all the edges in the graph in Figure 24.1 using the following array:

```
int edges[][2] = {
  {0, 1}, {0, 3}, {0, 5},
  {1, 0}, {1, 2}, {1, 3},
  {2, 1}, {2, 3}, {2, 4}, {2, 10},
  {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
  {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
  {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
  {6, 5}, {6, 7},
  {7, 4}, {7, 5}, {7, 6}, {7, 8},
  {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
  {9, 8}, {9, 11},
  {10, 2}, {10, 4}, {10, 8}, {10, 11},
  {11, 8}, {11, 9}, {11, 10}
};
```

This is known as the *edge representation using arrays*.

24.3.3 Representing Edges (for input): **Edge** *Objects*

Another way to represent the edges is to define edges as objects and store them in a vector. The **Edge** class is defined in Listing 24.1:

Listing 24.1 Edge.h

```
#ifndef EDGE_H
#define EDGE_H

class Edge
{
public:
  int u;
  int v;

  Edge(int u, int v)
  {
    this->u = u;
    this->v = v;
  }
};
#endif
```

For example, you can store all the edges in the graph in Figure 24.1 using the following vector:

```
vector<Edge> edgeVector;
edgeVector.push_back(Edge(0, 1));
edgeVector.push_back(Edge(0, 3));
edgeVector.push_back(Edge(0, 5));
```

...

Storing **`Edge`** objects in a vector is useful if you don't know the edges in advance.

Representing edges using edge array or **`Edge`** objects in §24.3.2 and §24.3.3 is intuitive for input, but not efficient for internal processing. The next two sections introduce the representation of graphs using adjacency matrices and adjacency lists. These two data structures are efficient for processing graphs.

24.3.4 Representing Edges: Adjacency Matrices

Assume that the graph has *n* vertices. You can use a two-dimensional *n* × *n* matrix, say **`adjacencyMatrix`**, to represent edges. Each element in the array is **`0`** or **`1`**. **`adjacencyMatrix[i][j]`** is **`1`** if there is an edge from vertex *i* to vertex *j*; otherwise, **`adjacencyMatrix[i][j]`** is **`0`**. If the graph is undirected, the matrix is symmetric, because **`adjacencyMatrix[i][j]`** is the same as **`adjacencyMatrix[j][i]`**. For example, the edges in the graph in Figure 24.1 can be represented using an adjacency matrix as follows:

```
int adjacencyMatrix[12][12] = {
  {0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0}, // Seattle
  {1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0}, // San Francisco
  {0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0}, // Los Angeles
  {1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0}, // Denver
  {0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0}, // Kansas City
  {1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0}, // Chicago
  {0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0}, // Boston
  {0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0}, // New York
  {0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1}, // Atlanta
  {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1}, // Miami
  {0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1}, // Dallas
  {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0}  // Houston
};
```

The adjacency matrix for the directed graph in Figure 24.3a can be represented as follows:

```
int a[5][5] = {{0, 0, 1, 0, 0}, // Peter
               {0, 0, 1, 0, 0}, // Jane
               {0, 0, 0, 0, 1}, // Mark
               {0, 0, 0, 0, 1}, // Cindy
               {0, 0, 0, 0, 0}  // Wendy
              };
```

As discussed in §12.7, it is more flexible to represent arrays using vectors. When you pass an array to a function, you also have to pass its size, but when you pass a vector to a function, you don't have to pass its size, because a **`vector`** object contains the size information. The preceding adjacency matrix can be represented using a vector as follows:

```
vector<vector<int>> a(5);
a[0] = vector<int>(5); a[1] = vector<int>(5); a[2] = vector<int>(5);
a[3] = vector<int>(5); a[4] = vector<int>(5);

a[0][0] = 0; a[0][1] = 0; a[0][2] = 1; a[0][3] = 0; a[0][4] = 0;
a[1][0] = 0; a[1][1] = 0; a[1][2] = 1; a[1][3] = 0; a[1][4] = 0;
a[2][0] = 0; a[2][1] = 0; a[2][2] = 0; a[2][3] = 0; a[2][4] = 1;
a[3][0] = 0; a[3][1] = 0; a[3][2] = 0; a[3][3] = 0; a[3][4] = 1;
a[4][0] = 0; a[4][1] = 0; a[4][2] = 0; a[4][3] = 0; a[4][4] = 0;
```

24.3.5 Representing Edges: Adjacency Lists

You can represent edges using *adjacency vertex lists* or *adjacency edge lists*. An adjacency vertex list for a vertex *i* contains the vertices that are adjacent to *i* and an adjacency edge list for a vertex *i* contains the edges that are adjacent to *i*. You may define an array of lists. The array has *n* entries, and each entry is a list. The adjacency vertex list for vertex *i* contains all the vertices *j* such that there is an edge from vertex *i* to vertex *j*. For example, to represent the edges in the graph in Figure 24.1, you can create an array of lists as follows:

```
list<int> neighbors[12];
```

`neighbors[0]` contains all vertices adjacent to vertex **`0`** (i.e., Seattle), **`neighbors[1]`** contains all vertices adjacent to vertex **`1`** (i.e., San Francisco), and so on, as shown in Figure 24.5.

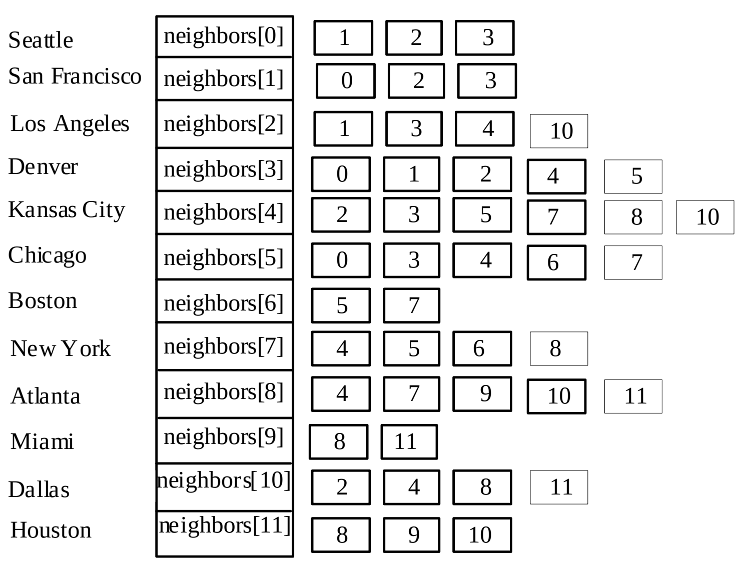


Figure 24.5

Edges in the graph in Figure 24.1 are represented using linked lists.

To represent the adjacency edge lists for the graph in Figure 24.1, you can create an array of lists as follows:

```
list<Edge> neighbors[12];
```

`neighbors[0]` contains all edges adjacent to vertex `0` (i.e., Seattle), `neighbors[1]` contains all edges adjacent to vertex `1` (i.e., San Francisco), and so on, as shown in Figure 24.6.

Seattle	neighbors[0]	Edge(0, 1)	Edge(0, 2)	Edge(0, 3)			
San Francisco	neighbors[1]	Edge(1, 0)	Edge(1, 2)	Edge(1, 3)			
Los Angeles	neighbors[2]	Edge(2, 1)	Edge(2, 3)	Edge(2, 4)	Edge(2, 10)		
Denver	neighbors[3]	Edge(3, 0)	Edge(3, 1)	Edge(3, 2)	Edge(3, 4)	Edge(3, 5)	
Kansas City	neighbors[4]	Edge(4, 2)	Edge(4, 3)	Edge(4, 5)	Edge(4, 7)	Edge(4, 8)	Edge(4, 10)
Chicago	neighbors[5]	Edge(5, 0)	Edge(5, 3)	Edge(5, 4)	Edge(5, 6)	Edge(5, 7)	
Boston	neighbors[6]	Edge(6, 5)	Edge(6, 7)				
New York	neighbors[7]	Edge(7, 4)	Edge(7, 5)	Edge(7, 6)	Edge(7, 8)		
Atlanta	neighbors[8]	Edge(8, 4)	Edge(8, 7)	Edge(8, 9)	Edge(8, 10)	Edge(8, 11)	
Miami	neighbors[9]	Edge(9, 8)	Edge(9, 11)				
Dallas	neighbors[10]	Edge(10, 2)	Edge(10, 4)	Edge(10, 8)	Edge(10, 11)		
Houston	neighbors[11]	Edge(11, 8)	Edge(11, 9)	Edge(11, 10)			

Figure 24.6

Edges in the graph in Figure 24.1 are represented using adjacency edge lists.

NOTE

You can represent a graph using an adjacency matrix or adjacency lists. Which one is better? If the graph is dense (i.e., there are a lot of edges), using an adjacency matrix is preferred. If the graph is very sparse (i.e., very few edges), using adjacency lists is better, because using an adjacency matrix would waste a lot of space.

Both adjacency matrices and adjacency lists may be used in a program to make algorithms more efficient. For example, it takes $O(1)$ constant time to check whether two vertices are connected using an adjacency matrix and it takes linear time $O(m)$ to print all edges in a graph using adjacency lists, where m is the number of edges.

NOTE

Adjacency vertex is simpler for representing unweighted graphs. However, adjacency edge lists are more flexible for a wide range of graph applications. It is easy to add additional constraints on edges using adjacency edge lists. For this reason, this book will use adjacency edge lists to represent graphs.

For flexibility and simplicity, we will use vectors to represent arrays. Also we will use vectors instead of lists, because our algorithms only requires search for adjacent vertices in the list. Using vectors can simplify coding. Using vectors, the adjacency list in Figure 24.5 can be built as follows:

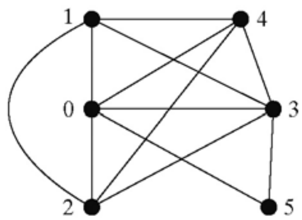
```
vector<vector<Edge*>> neighbors(12);
neighbors[0].push_back(new Edge(0, 1));
neighbors[0].push_back(new Edge(0, 3));
neighbors[0].push_back(new Edge(0, 5));
neighbors[1].push_back(new Edge(1, 0));
neighbors[1].push_back(new Edge(1, 2));
neighbors[1].push_back(new Edge(1, 3));
...
...
```

Note that the pointers of **Edge** objects are stored in **neighbors**. This enables **neighbors** to store any subtypes of **Edge** for using polymorphism. You will see its benefits in the next chapter when we add a weighted edge to **neighbors**.

Check point

24.5 How do you represent vertices in a graph? How do you represent edges using an edge array? How do you represent an edge using an edge object? How do you represent edges using an adjacency matrix? How do you represent edges using adjacency lists?

24.6 Represent the following graph using an edge array, a list of edge objects, an adjacency matrix, an adjacency vertex list, and an adjacency edge list, respectively.



24.4 The `Graph` Class

Key Point: *The **Graph** class defines the common operations for a graph.*

We now design a class to model graphs. What are the common operations for a graph? In general, you need to get the number of vertices in a graph, get all vertices in a graph, get the vertex object with a specified index, get the index of the vertex with a specified name, get the neighbors for a vertex, get the adjacency matrix, get the degree for a vertex, perform a depth-first search, and perform a breadth-first search. Depth-first search and breadth-first search will be introduced in the next section. Figure 24.7 illustrates these functions in the UML diagram.

Graph<V>	
#vertices: vector<V>	Vertices in the graph.
#neighbors: vector<vector<int>>	neighbors[i] stores all vertices adjacent to vertex with index i.
+Graph()	Constructs an empty graph.
+Graph(vertices: vector<V>&, edges[][2]: int, numberOfEdges: int)	Constructs a graph with the specified vertices in a vector and edges in a 2-D array.
+Graph(numberOfVertices: int, edges[][2]: int, numberOfEdges: int)	Constructs a graph whose vertices are 0, 1, …, *n*-1and edges are specified in a 2-D array.
+Graph(vertices: vector<V>&, edges: vector<Edge>&)	Constructs a graph with the vertices in a vector and edges in a vector of Edge objects.
+Graph(numberOfVertices: int, edges: vector<Edge>&)	Constructs a graph whose vertices are 0, 1, …, *n*–1 and edges in a vector of Edge objects.
+getSize(): int const	Returns the number of vertices in the graph.
+getDegree(v: int): int const	Returns the degree for a specified vertex index.
+getVertex(index: int): V const	Returns the vertex for the specified vertex index.
+getIndex(v: V): int const	Returns the index for the specified vertex.
+getVertices(): vector<V> const	Returns the vertices in the graph in a vector.
+getNeighbors(v: int): vector<int> const	Returns the neighbors of vertex with index v.
+printEdges(): void const	Prints the edges of the graph to the console.
+printAdjacencyMatrix(): void const	Prints the adjacency matrix of the graph to the console.
+clear(): void	Clears the graph.
+addVertex(v: V): bool	Returns true if v is added to the graph. Returns false if v is already in the graph.
+addEdge(u: int, v: int): bool	Adds an edge from u to v to the graph throws invalid_argument if u or v is invalid. Returns true if the edge is added and false if (u, v) is already in the graph.
#createEdge(e: Edge*): bool	This protected function is called by addEdge.
+dfs(v: int): Tree const	Obtains a depth-first search tree.
+bfs(v: int): Tree const	Obtains a breadth-first search tree.

Figure 24.7

*The **`Graph`** class defines the common operations for graphs.*

`vertices`, a vector, is defined in the **`Graph`** class to represent vertices. The vertices may be of any type: **`int`**, string, and so on. So, we use a generic type **`T`** to define it. **`neighbors`**, a vector of vectors, is defined to represent edges. With these two data fields, it is sufficient to implement all the functions defined in the **`Graph`** class.

A no-arg constructor is provided for convenience. With a no-arg constructor, it is easy to use the class as a base class and as a data type for data fields in a class. You can create a **`Graph`** object using one of the four other constructors, whichever is convenient. If you have an edge array, use the first two constructors in the

UML class diagram. If you have a vector of **Edge** objects, use the last two constructors in the UML class diagram.

The generic type **T** indicates the type of vertices—integer, string, and so on. You can create a graph with vertices of any type. If you create a graph without specifying the vertices, the vertices are integers **0**, **1**, ..., **n - 1**, where **n** is the number of vertices. Each vertex is associated with an index, which is the same as the index of the vertex in the array for vertices.

Assume the class is available in Graph.h. Listing 24.2 gives a test program that creates a graph for the one in Figure 24.1 and another graph for the one in Figure 24.3a.

Listing 24.2 TestGraph.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include "Graph.h" // Defined in Listing 24.2
#include "Edge.h" // Defined in Listing 24.1
using namespace std;

int main()
{
  // Vertices for graph in Figure 24.1
  string vertices[] = {"Seattle", "San Francisco", "Los Angeles",
    "Denver", "Kansas City", "Chicago", "Boston", "New York",
    "Atlanta", "Miami", "Dallas", "Houston"};

  // Edge array for graph in Figure 24.1
  int edges[][2] = {
    {0, 1}, {0, 3}, {0, 5},
    {1, 0}, {1, 2}, {1, 3},
    {2, 1}, {2, 3}, {2, 4}, {2, 10},
    {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
    {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
    {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
    {6, 5}, {6, 7},
    {7, 4}, {7, 5}, {7, 6}, {7, 8},
    {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
    {9, 8}, {9, 11},
    {10, 2}, {10, 4}, {10, 8}, {10, 11},
    {11, 8}, {11, 9}, {11, 10}
  };
  const int NUMBER_OF_EDGES = 46; // 46 edges in Figure 24.1

  // Create a vector for vertices
```

```
  vector<string> vectorOfVertices(12);
  copy(vertices, vertices + 12, vectorOfVertices.begin());

  Graph<string> graph1(vectorOfVertices, edges, NUMBER_OF_EDGES);
  cout << "The number of vertices in graph1: " << graph1.getSize();
  cout << "\nThe vertex with index 1 is " << graph1.getVertex(1);
  cout << "\nThe index for Miami is " << graph1.getIndex("Miami");

  cout << "\nedges for graph1: " << endl;
  graph1.printEdges();

  cout << "\nAdjacency matrix for graph1: " << endl;
  graph1.printAdjacencyMatrix();

  // vector of Edge objects for graph in Figure 24.3a
  vector<Edge> edgeVector;
  edgeVector.push_back(Edge(0, 2));
  edgeVector.push_back(Edge(1, 2));
  edgeVector.push_back(Edge(2, 4));
  edgeVector.push_back(Edge(3, 4));
  // Create a graph with 5 vertices
  Graph<int> graph2(5, edgeVector);

  cout << "The number of vertices in graph2: " << graph2.getSize();
  cout << "\nedges for graph2: " << endl;
  graph2.printEdges();

  cout << "\nAdjacency matrix for graph2: " << endl;
  graph2.printAdjacencyMatrix();

  return 0;
}
```

Sample output

```
The number of vertices in graph1: 12
The vertex with index 1 is San Francisco
The index for Miami is 9
edges for graph1:
Vertex Seattle(0): (Seattle, San Francisco)
  (Seattle, Denver) (Seattle, Chicago)
Vertex San Francisco(1): (San Francisco, Seattle)
  (San Francisco, Los Angeles) (San Francisco, Denver)
Vertex Los Angeles(2): (Los Angeles, San Francisco)
  (Los Angeles, Denver) (Los Angeles, Kansas City)
  (Los Angeles, Dallas)
Vertex Denver(3): (Denver, Seattle) (Denver, San Francisco)
  (Denver, Los Angeles) (Denver, Kansas City)
  (Denver, Chicago)
Vertex Kansas City(4): (Kansas City, Los Angeles)
  (Kansas City, Denver) (Kansas City, Chicago)
  (Kansas City, New York) (Kansas City, Atlanta)
  (Kansas City, Dallas)
Vertex Chicago(5): (Chicago, Seattle) (Chicago, Denver)
  (Chicago, Kansas City) (Chicago, Boston)
  (Chicago, New York)
Vertex Boston(6): (Boston, Chicago) (Boston, New York)
Vertex New York(7): (New York, Kansas City)
  (New York, Chicago) (New York, Boston) (New York, Atlanta)
Vertex Atlanta(8): (Atlanta, Kansas City)
  (Atlanta, New York) (Atlanta, Miami) (Atlanta, Dallas)
```

```
  (Atlanta, Houston)
Vertex Miami(9): (Miami, Atlanta) (Miami, Houston)
Vertex Dallas(10): (Dallas, Los Angeles)
  (Dallas, Kansas City) (Dallas, Atlanta)
  (Dallas, Houston)
Vertex Houston(11): (Houston, Atlanta) (Houston, Miami)
  (Houston, Dallas)

Adjacency matrix for graph1:
0 1 0 1 0 1 0 0 0 0 0 0
1 0 1 1 0 0 0 0 0 0 0 0
0 1 0 1 1 0 0 0 0 0 1 0
1 1 1 0 1 1 0 0 0 0 0 0
0 0 1 1 0 1 0 1 1 0 1 0
1 0 0 1 1 0 1 1 0 0 0 0
0 0 0 0 0 1 0 1 0 0 0 0
0 0 0 0 1 1 1 0 1 0 0 0
0 0 0 0 1 0 0 1 0 1 1 1
0 0 0 0 0 0 0 0 1 0 0 1
0 0 1 0 1 0 0 0 1 0 0 1
0 0 0 0 0 0 0 0 1 1 1 0
The number of vertices in graph2: 5
edges for graph2:
Vertex 0(0): (0, 2)
Vertex 1(1): (1, 2)
Vertex 2(2): (2, 4)
Vertex 3(3): (3, 4)
Vertex 4(4):

Adjacency matrix for graph2:
0 0 1 0 0
0 0 1 0 0
0 0 0 0 1
0 0 0 0 1
0 0 0 0 0
```

The program creates **graph1** for the graph in Figure 24.1 in lines 11–36. The vertices for **graph1** are defined in lines 11–13. The edges for **graph1** are defined in lines 16–29. The edges are represented using a two-dimensional array. For each row **i** in the array, **edges[i][0]** and **edges[i][1]** indicate that there is an edge from vertex **edges[i][0]** to vertex **edges[i][1]**. For example, the first row {**0**, **1**} represents the edge from vertex **0** (**edges[0][0]**) to vertex **1** (**edges[0][1]**). The row {**0**, **5**} represents the edge from vertex **0** (**edges[2][0]**) to vertex **5** (**edges[2][1]**). The graph is created in line 36. Line 42 invokes the **printEdges()** function on **graph1** to display all edges in **graph1**. Line 45 invokes the **printAdjacencyMatrix()** function on **graph1** to display the adjacency matrix for **graph1**.

The program creates **graph2** for the graph in Figure 24.3a in lines 48–54. The edges for **graph2** are defined in lines 48–52. **graph2** is created using a vector of **Edge** objects in line 54. Line 58 invokes the **printEdges()** function on **graph2** to display all edges in **graph2**. Line 61 invokes the **printAdjacencyMatrix()** function on **graph2** to display the adjacency matrix for **graph1**.

Note that **graph1** contains the vertices of strings and **graph2** contains the vertices with integers **0**, **1**, ..., **n-1**, where **n** is the number of vertices. In **graph1**, the vertices are associated with indices **0**, **1**, ..., **n-1**. The index is the location of the vertex in **vertices**. For example, the index of vertex **Miami** is **9**.

Now we turn our attention to implementing the **Graph** class, as shown in Listing 24.3.

Listing 24.3 Graph.h

```
#ifndef GRAPH_H
#define GRAPH_H

#include "Edge.h" // Defined in Listing 24.1
#include "Tree.h" // Defined in Listing 24.4
#include <vector>
#include <queue> // For implementing BFS
#include <stdexcept>
#include <sstream> // For converting a number to a string

using namespace std;

template<typename V>
class Graph
{
public:
  // Construct an empty graph
  Graph();

  // Construct a graph from vertices in a vector and
  //  edges in 2-D array
  Graph(vector<V>& vertices, int edges[][2], int numberOfEdges);

  // Construct a graph with vertices 0, 1, ..., n-1 and
  // edges in 2-D array
  Graph(int numberOfVertices, int edges[][2], int numberOfEdges);

  // Construct a graph from vertices and edges objects
  Graph(vector<V>& vertices, vector<Edge>& edges);

  // Construct a graph with vertices 0, 1, ..., n-1 and
  // edges in a vector
  Graph(int numberOfVertices, vector<Edge>& edges);
```

```

  // Return the number of vertices in the graph
  int getSize() const;

  // Return the degree for a specified vertex
  int getDegree(int v) const;

  // Return the vertex for the specified index
  V getVertex(int index) const;

  // Return the index for the specified vertex
  int getIndex(V v) const;

  // Return the vertices in the graph
  vector<V> getVertices() const;

  // Return the neighbors of vertex v
  vector<int> getNeighbors(int v) const;

  // Print the edges
  void printEdges() const;

  // Print the adjacency matrix
  void printAdjacencyMatrix() const;

  // Clear the graph
  void clear();

  // Adds a vertex to the graph
  virtual bool addVertex(V v);

  // Adds an edge from u to v to the graph
  bool addEdge(int u, int v);

  // Obtain a depth-first search tree
  // To be discussed in Section 24.6
  Tree dfs(int v) const;

  // Starting bfs search from vertex v
  // To be discussed in Section 24.7
  Tree bfs(int v) const;

protected:
  vector<V> vertices; // Store vertices
  vector<vector<Edge*>> neighbors; // Adjacency edge lists
  bool createEdge(Edge* e); // Add an edge

private:
  // Create adjacency lists for each vertex from an edge array
  void createAdjacencyLists(int numberOfVertices, int edges[][2],
    int numberOfEdges);

  // Create adjacency lists for each vertex from an Edge vector
  void createAdjacencyLists(int numberOfVertices,
    vector<Edge>& edges);

  // Recursive function for DFS search
  void dfs(int v, vector<int>& parent,
    vector<int>& searchOrders, vector<bool>& isVisited) const;
};
```

```
template<typename V>
Graph<V>::Graph()
{
}

template<typename V>
Graph<V>::Graph(vector<V>& vertices, int edges[][2],
  int numberOfEdges)
{
  for (unsigned i = 0; i < vertices.size(); i++)
    addVertex(vertices[i]);

  createAdjacencyLists(vertices.size(), edges, numberOfEdges);
}

template<typename V>
Graph<V>::Graph(int numberOfVertices, int edges[][2],
  int numberOfEdges)
{
  for (int i = 0; i < numberOfVertices; i++)
    addVertex(i); // vertices is {0, 1, 2, ..., n-1}

  createAdjacencyLists(numberOfVertices, edges, numberOfEdges);
}

template<typename V>
Graph<V>::Graph(vector<V>& vertices, vector<Edge>& edges)
{
  for (unsigned i = 0; i < vertices.size(); i++)
    addVertex(vertices[i]);

  createAdjacencyLists(vertices.size(), edges);
}

template<typename V>
Graph<V>::Graph(int numberOfVertices, vector<Edge>& edges)
{
  for (int i = 0; i < numberOfVertices; i++)
    addVertex(i); // vertices is {0, 1, 2, ..., n-1}

  createAdjacencyLists(numberOfVertices, edges);
}

template<typename V>
void Graph<V>::createAdjacencyLists(int numberOfVertices,
  int edges[][2],  int numberOfEdges)
{
  for (int i = 0; i < numberOfEdges; i++)
  {
    int u = edges[i][0];
    int v = edges[i][1];
    addEdge(u, v);
  }
}

template<typename V>
void Graph<V>::createAdjacencyLists(int numberOfVertices,
  vector<Edge>& edges)
{
```

```
  for (unsigned i = 0; i < edges.size(); i++)
  {
    int u = edges[i].u;
    int v = edges[i].v;
    addEdge(u, v);
  }
}

template<typename V>
int Graph<V>::getSize() const
{
  return vertices.size();
}

template<typename V>
int Graph<V>::getDegree(int v) const
{
  return neighbors[v].size();
}

template<typename V>
V Graph<V>::getVertex(int index) const
{
  return vertices[index];
}

template<typename V>
int Graph<V>::getIndex(V v) const
{
  for (unsigned i = 0; i < vertices.size(); i++)
  {
    if (vertices[i] == v)
      return i;
  }

  return -1; // If vertex is not in the graph
}

template<typename V>
vector<V> Graph<V>::getVertices() const
{
  return vertices;
}

template<typename V>
vector<int> Graph<V>::getNeighbors(int u) const
{
  vector<int> result;
  for (Edge* e: neighbors[u])
    result.push_back(e->v);
  return result;
}

template<typename V>
void Graph<V>::printEdges() const
{
  for (unsigned u = 0; u < neighbors.size(); u++)
  {
    cout << "Vertex " << getVertex(u) << "(" << u << "): ";
```

```
    for (Edge* e: neighbors[u])
    {
      cout << "(" << getVertex(e->u) << ", " << getVertex(e->v) << ") ";
    }
    cout << endl;
  }
}

template<typename V>
void Graph<V>::printAdjacencyMatrix() const
{
  // Use vector for 2-D array
  vector<vector<int>> adjacencyMatrix(getSize());

  // Initialize 2-D array for adjacency matrix
  for (int i = 0; i < getSize(); i++)
  {
    adjacencyMatrix[i] = vector<int>(getSize());
  }

  for (unsigned i = 0; i < neighbors.size(); i++)
  {
    for (Edge* e: neighbors[i])
    {
      adjacencyMatrix[i][e->v] = 1;
    }
  }

  for (unsigned i = 0; i < adjacencyMatrix.size(); i++)
  {
    for (unsigned j = 0; j < adjacencyMatrix[i].size(); j++)
    {
      cout << adjacencyMatrix[i][j] << " ";
    }

    cout << endl;
  }
}

template<typename V>
void Graph<V>::clear()
{
  vertices.clear();
  for (int i = 0; i < getSize(); i++)
    for (Edge* e: neighbors[i])
      delete e;
  neighbors.clear();
}

template<typename V>
bool Graph<V>::addVertex(V v)
{
  if (find(vertices.begin(), vertices.end(), v) == vertices.end())
  {
    vertices.push_back(v);
    neighbors.push_back(vector<Edge*>(0));
    return true;
  }
  else
```

```
  {
    return false;
  }
}

template<typename V>
bool Graph<V>::createEdge(Edge* e)
{
  if (e->u < 0 || e->u > getSize() - 1)
  {
    stringstream ss;
    ss << e->u;
    throw invalid_argument("No such edge: " + ss.str());
  }

  if (e->v < 0 || e->v > getSize() - 1)
  {
    stringstream ss;
    ss << e->v;
    throw invalid_argument("No such edge: " + ss.str());
  }

  vector<int> listOfNeighbors = getNeighbors(e->u);
  if (find(listOfNeighbors.begin(), listOfNeighbors.end(), e->v)
    == listOfNeighbors.end())
  {
    neighbors[e->u].push_back(e);
    return true;
  }
  else
  {
    return false;
  }
}

template<typename V>
bool Graph<V>::addEdge(int u, int v)
{
  return createEdge(new Edge(u, v));
}

template<typename V>
Tree Graph<V>::dfs(int u) const
{
  vector<int> searchOrders;
  vector<int> parent(vertices.size());
  for (unsigned i = 0; i < vertices.size(); i++)
    parent[i] = -1; // Initialize parent[i] to -1

  // Mark visited vertices
  vector<bool> isVisited(vertices.size());
  for (unsigned i = 0; i < vertices.size(); i++)
  {
    isVisited[i] = false;
  }

  // Recursively search
  dfs(u, parent, searchOrders, isVisited);

  // Return a search tree
```

```
332   return Tree(u, parent, searchOrders);
333 }
334
335 template<typename V>
336 void Graph<V>::dfs(int u, vector<int>& parent,
337   vector<int>& searchOrders, vector<bool>& isVisited) const
338 {
339   // Store the visited vertex
340   searchOrders.push_back(u);
341   isVisited[u] = true; // Vertex v visited
342
343   for (Edge* e: neighbors[u])
344   {
345     if (!isVisited[e->v])
346     {
347       parent[e->v] = u; // The parent of vertex i is v
348       dfs(e->v, parent, searchOrders, isVisited); // Recursive
search
349     }
350   }
351 }
352
353 template<typename V>
354 Tree Graph<V>::bfs(int v) const
355 {
356   vector<int> searchOrders;
357   vector<int> parent(vertices.size());
358   for (int i = 0; i < getSize(); i++)
359     parent[i] = -1; // Initialize parent[i] to -1
360
361   queue<int> queue; // Stores vertices to be visited
362   vector<bool> isVisited(getSize());
363   queue.push(v); // Enqueue v
364   isVisited[v] = true; // Mark it visited
365
366   while (!queue.empty())
367   {
368     int u = queue.front(); // Get from the front of the queue
369     queue.pop(); // remove the front element
370     searchOrders.push_back(u); // u searched
371     for (Edge* e: neighbors[u])
372     {
373       int w = e->v;
374       if (!isVisited[w])
375       {
376         queue.push(w); // Enqueue w
377         parent[w] = u; // The parent of w is u
378         isVisited[w] = true; // Mark it visited
379       }
380     }
381   }
382
383   return Tree(v, parent, searchOrders);
384 }
385
386 #endif
```

To construct a graph, you need to create vertices and edges. The vertices are stored in a **vector<T>** and the edges in a **vector<vector<Edge>>** (line 78), which is the adjacency edge list described in

§24.3.4. The constructors (lines 95–136) create vertices and edges. The edges may be created from an edge array (discussed in §24.3.2) or a vector of **`Edge`** objects (discussed in §24.3.3). The private function **`createAdjacencyList`** (lines 138–148) is used to create the adjacency list from an edge array and the overloaded **`createAdjacencyList`** function (lines 150–160) is used to create the adjacency list from a vector of **`Edge`** objects. The **`Edge`** class is defined in Listing 24.1, which simply defines two vertices having an edge.

`vertices` and **`neighbors`** are declared protected so that they can be accessed from derived classes of **`Graph`** for future extension.

The function **`getSize`** returns the number of the vertices in the graph (lines 162–166). The function **`getDegree(int v)`** returns the degree of a vertex with index **`v`** (lines 168–172). The function **`getVertex(int index)`** returns the vertex with the specified index (lines 174–178). The function **`getIndex(T v)`** returns the index of the specified vertex (lines 180–190). The function **`getVertices()`** returns the vector for the vertex (lines 192–196). The function **`getNeighbors(u)`** returns a list of vertices adjacent to u (lines 198–205). The function **`printEdges()`** (lines 207–219) displays all vertices and edges adjacent to each vertex. The function **`printAdjacencyMatrix()`** (lines 221–250) displays the adjacency matrix.

The function **`clear`** removes all edges and vertices from the graph (lines 252-260). The function **`addVertex(v)`** adds a new vertex to the graph and returns true if the vertex is not in the graph (lines 262-275). If the vertex is already in the graph, the function returns false (line 273).

The function **`createEdge`** adds a new edge to the graph (lines 277-305). It throws an **`invalid_argument`** exception if the edges are invalid (lines 280-292). It returns false if the edge is already in the graph (line 303). Note that this function is a protected function and may be used by a derived class to add a different type of edge to the graph. In this class, it is called by the **`addEdge(u, v)`** function to add an unweighted edge to the graph (line 310).

The code in lines 313–384 gives the functions for finding a depth-first search tree and a breadth-first search tree, which will be introduced in subsequent sections.

Check point

24.7 Describe the relationships among **Graph**, **Edge**, and **Tree**.

24.8 For the code in Listing 24.1, TestGraph.cpp, what is **graph1.getIndex("Seattle")**? What is **graph1.getDegree(5)**? What is **graph1.getVertex(4)**?

24.5 Graph Traversals

Key Point: *Depth-first and breadth-first are two common ways to traverse a graph.*

Graph traversal is the process of visiting each vertex in the graph exactly once. There are two popular ways to traverse a graph: *depth-first traversal* (or *depth-first search*) and *breadth-first traversal* (or *breadth-first search*). Both traversals result in a spanning tree, which can be modeled using a class, as shown in Figure 24.8. The **`Tree`** class describes the parent-child relationship of the nodes in the tree, as shown in Listing 24.4.

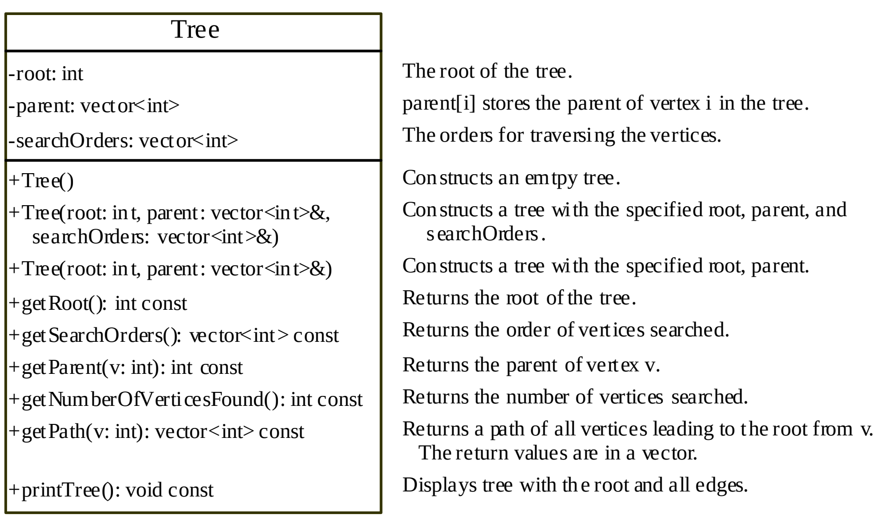


Figure 24.8

*The **`Tree`** class describes parent-child relationship of the nodes in a tree.*

Listing 24.4 Tree.h

```
#ifndef TREE_H
#define TREE_H

#include <vector>
using namespace std;

class Tree
{
public:
  // Construct an empty tree
  Tree()
  {
  }

  // Construct a tree with root, parent, and searchOrder
  Tree(int root, vector<int>& parent, vector<int>& searchOrders)
  {
    this->root = root;
    this->parent = parent;
    this->searchOrders = searchOrders;
  }

  // Return the root of the tree
  int getRoot() const
  {
    return root;
  }

  // Return the parent of vertex v
  int getParent(int v) const
  {
    return parent[v];
  }

  // Return search order
  vector<int> getSearchOrders() const
  {
    return searchOrders;
  }

  // Return number of vertices found
  int getNumberOfVerticesFound() const
  {
    return searchOrders.size();
  }

  // Return the path of vertices from v to the root in a vector
  vector<int> getPath(int v) const
  {
    vector<int> path;

    do
    {
      path.push_back(v);
      v = parent[v];
    }
    while (v != -1);
```

```
58
59       return path;
60     }
61
62     // Print the whole tree
63     void printTree() const
64     {
65       cout << "Root is: " << root << endl;
66       cout << "Edges: ";
67       for (unsigned i = 0; i < searchOrders.size(); i++)
68       {
69         if (parent[searchOrders[i]] != -1)
70         {
71           // Display an edge
72           cout << "(" << parent[searchOrders[i]] << ", " <<
73             searchOrders[i] << ") ";
74         }
75       }
76       cout << endl;
77     }
78
79   private:
80     int root; // The root of the tree
81     vector<int> parent; // Store the parent of each vertex
82     vector<int> searchOrders; // Store the search order
83   };
84   #endif
```

The **`Tree`** class has two constructors. The no-arg constructor constructs an empty tree. The other constructor constructs a tree with a search order (lines 16–21).

The **`Tree`** class defines seven functions. The **`getRoot()`** function returns the root of the tree (lines 24–27). You can invoke **`getParent(v)`** to find the parent of vertex **`v`** in the search (lines 30–33). You can get the order of the vertices searched by invoking the **`getSearchOrders()`** function (lines 36–39). Invoking **`getNumberOfVerticesFound()`** returns the number of vertices searched (lines 42–45). The **`getPath(v)`** function returns a path from the **`v`** to root (lines 48–60). You can display all edges in the tree using the **`printTree()`** function (lines 63–75).

Sections 24.6 and 24.7 will introduce depth-first search and breadth-first search, respectively. Both searches will result in an instance of the **`Tree`** class.

Check point

24.9 What function do you use to find the parent of a vertex in the tree?

24.6 Depth-First Search

Key Point: *The depth-first search of a graph starts from a vertex in the graph and visits all vertices in the graph as far as possible before backtracking.*

The depth-first search of a graph is like the depth-first search of a tree discussed in §21.2.5, “Tree Traversal.” In the case of a tree, the search starts from the root. In a graph, the search can start from any vertex.

A depth-first search of a tree first visits the root, then recursively visits the subtrees of the root. Similarly, the depth-first search of a graph first visits a vertex, then recursively visits all vertices adjacent to that vertex. The difference is that the graph may contain cycles, which may lead to an infinite recursion. To avoid this problem, you need to track the vertices that have already been visited and avoid visiting them again.

The search is called depth-first, because it searches “deeper” in the graph as much as possible. The search starts from some vertex *v*. After visiting *v*, visit an unvisited neighbor of *v*. If *v* has no unvisited neighbor, backtrack to the vertex from which we reached *v*.

24.6.1 Depth-First Search Algorithm

The algorithm for the depth-first search can be described in Listing 24.5.

Listing 24.5 Depth-first Search Algorithm

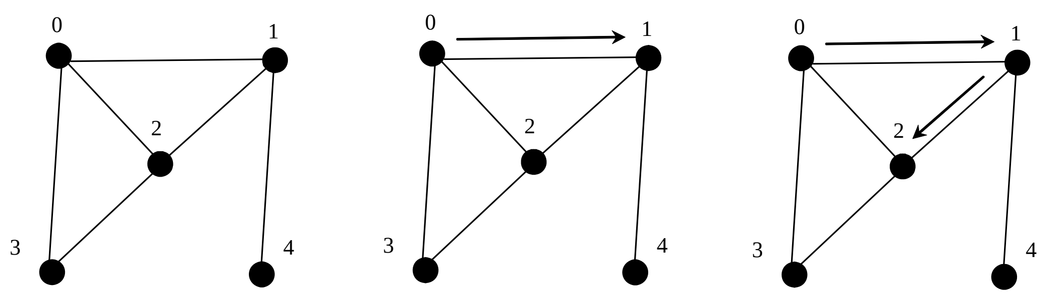
```
Input: G = (V, E) and a starting vertex v
Output: a DFS tree rooted at v

Tree dfs(vertex v)
{
  visit v;
  for each neighbor w of v
    if (w has not been visited)
    {
      parent[w] = v;
      dfs(w);
    }
}
```

You may use a vector named **isVisited** to denote whether a vertex has been visited. Initially, **isVisited[i]** is **false** for each vertex *i*. Once a vertex, say *v*, is visited, **isVisited[v]** is set to **true**.

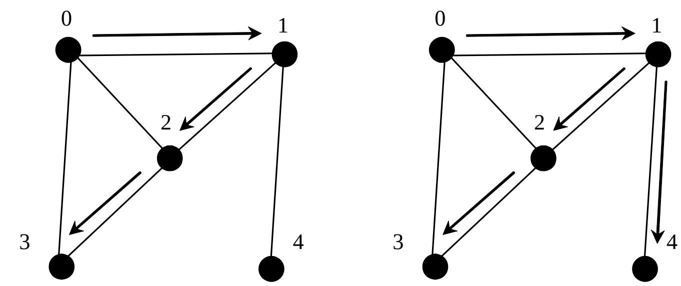
Consider the graph in Figure 24.9a. Suppose you start the depth-first search from vertex 0. First visit 0, then any of its neighbors, say 1. Now 1 is visited, as shown in Figure 24.9b. Vertex 1 has three neighbors: 0, 2, and 4. Since 0 has already been visited, you will visit either 2 or 4. Let us pick 2. Now 2 is visited, as shown in Figure 24.9c. 2 has three neighbors: 0, 1, and 3. Since 0 and 1 have already been visited, pick 3. 3 is now visited, as shown in Figure 24.9d. At this point, the vertices have been visited in this order:

0, 1, 2, 3

Since all the neighbors of 3 have been visited, backtrack to 2. Since all the vertices of 2 have been visited, backtrack to 1. 4 is adjacent to 1, but 4 has not been visited. So, visit 4, as shown in Figure 24.9e. Since all the neighbors of 4 have been visited, backtrack to 1. Since all the neighbors of 1 have been visited, backtrack to 0. Since all the neighbors of 0 have been visited, the search ends.



(a) **(b)** **(c)**



(d) **(e)**

Figure 24.9

Depth-first search visits a node and its neighbors recursively.

Since each edge and each vertex is visited only once, the time complexity of the `dfs` function is `O(|E| + |V|)`, where `|E|` denotes the number of edges and `|V|` the number of vertices.

24.6.2 Implementation of Depth-First Search

The algorithm is described in Listing 24.5, using recursion. It is natural to use recursion to implement it. Alternatively, you can use a stack (see Programming Exercise 24.4).

The `dfs(int v)` function is implemented in lines 245–265 in Listing 24.3. It returns an instance of the `Tree` class with vertex `v` as the root. The function stores the vertices searched in a list `searchOrders` (line 248), the parent of each vertex in an array `parent` (line 249), and uses the `isVisited` array to indicate whether a vertex has been visited (line 254). It invokes the helper function `dfs(v, parent, searchOrders, isVisited)` to perform a depth-first search (line 261).

In the recursive helper function, the search starts from vertex `v`. `v` is added to `searchOrders` (line 272) and is marked visited (line 273). For each unvisited neighbor of `v`, the function is recursively invoked to perform a depth-first search. When a vertex `i` is visited, the parent of `i` is stored in `parent[i]` (line

280). The function returns when all vertices are visited for a connected graph, or in a connected component.

Listing 24.6 gives a test program that displays a DFS for the graph in Figure 24.1, starting from **Chicago**.

Listing 24.6 TestDFS.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include "Graph.h" // Defined in Listing 24.2
#include "Edge.h" // Defined in Listing 24.1
#include "Tree.h" // Defined in Listing 24.4
using namespace std;

int main()
{
  // Vertices for graph in Figure 24.1
  string vertices[] = {"Seattle", "San Francisco", "Los Angeles",
    "Denver", "Kansas City", "Chicago", "Boston", "New York",
    "Atlanta", "Miami", "Dallas", "Houston"};

  // Edge array for graph in Figure 24.1
  int edges[][2] = {
    {0, 1}, {0, 3}, {0, 5},
    {1, 0}, {1, 2}, {1, 3},
    {2, 1}, {2, 3}, {2, 4}, {2, 10},
    {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
    {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
    {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
    {6, 5}, {6, 7},
    {7, 4}, {7, 5}, {7, 6}, {7, 8},
    {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
    {9, 8}, {9, 11},
    {10, 2}, {10, 4}, {10, 8}, {10, 11},
    {11, 8}, {11, 9}, {11, 10}
  };
  const int NUMBER_OF_EDGES = 46; // 46 edges in Figure 24.1

  // Create a vector for vertices
  vector<string> vectorOfVertices(12);
  copy(vertices, vertices + 12, vectorOfVertices.begin());

  Graph<string> graph(vectorOfVertices, edges, NUMBER_OF_EDGES);
  Tree dfs = graph.dfs(5); // Vertex 5 is Chicago

  vector<int> searchOrders = dfs.getSearchOrders();
  cout << dfs.getNumberOfVerticesFound() <<
    " vertices are searched in this DFS order:" << endl;
  for (unsigned i = 0; i < searchOrders.size(); i++)
    cout << graph.getVertex(searchOrders[i]) << " ";
  cout << endl << endl;

```

```
47    for (unsigned i = 0; i < searchOrders.size(); i++)
48      if (dfs.getParent(i) != -1)
49        cout << "parent of " << graph.getVertex(i) <<
50          " is " << graph.getVertex(dfs.getParent(i)) << endl;
51
52    return 0;
53  }
```

Sample output

```
      12 vertices are searched in this DFS order:
        Chicago Seattle San Francisco Los Angeles Denver Kansas City
        New York Boston Atlanta Miami Houston Dallas

      parent of Seattle is Chicago
      parent of San Francisco is Seattle
      parent of Los Angeles is San Francisco
      parent of Denver is Los Angeles
      parent of Kansas City is Denver
      parent of Boston is New York
      parent of New York is Kansas City
      parent of Atlanta is New York
      parent of Miami is Atlanta
      parent of Dallas is Houston
      parent of Houston is Miami
```

The program creates a graph for Figure 24.1 in line 37 and obtains a DFS tree starting from vertex **`Chicago`** in line 38. The search order is obtained in line 40. The graphical illustration of the DFS starting from **`Chicago`** is shown in Figure 24.10.

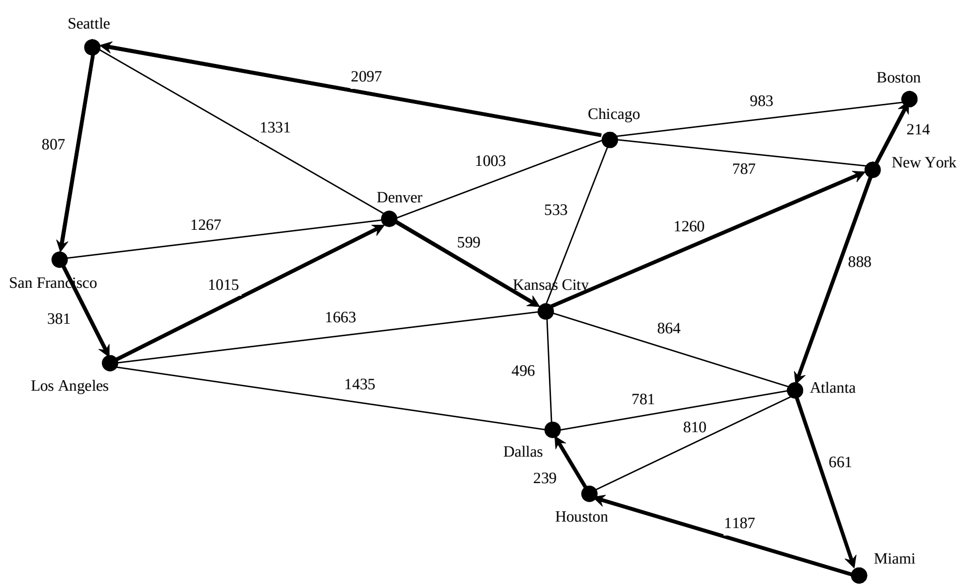


Figure 24.10

DFS search starts from Chicago.

Note it is not necessary to include Tree.h and Edge.h, because these two header files are already included in Graph.h.

24.6.3 Applications of the DFS

The depth-first search can be used to solve many problems, such as the following:

- Detecting whether a graph is connected. Search the graph starting from any vertex. If the number of vertices searched is the same as the number of vertices in the graph, the graph is connected. Otherwise, the graph is not connected.
- Detecting whether there is a path between two vertices.
- Finding a path between two vertices.
- Finding all connected components. A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path.
- Detecting whether there is a cycle in the graph.
- Finding a cycle in the graph.

The first four problems can be easily solved using the **`dfs`** function in Listing 24.3. To detect or find a cycle in the graph, you have to slightly modify the **`dfs`** function.

Check point

24.10 What is depth-first search?

24.11 Draw a DFS tree for the graph in Figure 24.3b starting from node `A`.

24.12 Draw a DFS tree for the graph in Figure 24.1 starting from vertex `Atlanta`.

24.13 What is the return type from invoking `dfs(v)`?

24.14 The depth-first search algorithm described in Listing 24.8 uses recursion. Alternatively, you can use a stack to implement it, as shown below. Point out the error in this algorithm and give a correct algorithm.

```
// Wrong version
```

```
Tree dfs(vertex v)
{
  push v into the stack;
  mark v visited;

  while (the stack is not empty)
  {
    pop a vertex, say u, from the stack
    visit u;
    for each neighbor w of u
      if (w has not been visited)
        push w into the stack;
  }
}
```

24.7 Breadth-First Search

Key Point: *The breadth-first search of a graph visits the vertices level by level. The first level consists of the starting vertex. Each next level consists of the vertices adjacent to the vertices in the preceding level.*

The breadth-first traversal of a graph is like the breadth-first traversal of a tree discussed in §21.2.5, "Tree Traversal." With breadth-first traversal of a tree, the nodes are visited level by level. First the root is visited, then all the children of the root, then the grandchildren of the root from left to right, and so on. Similarly, the breadth-first search of a graph first visits a vertex, then all its adjacent vertices, then all the vertices adjacent to those vertices, and so on. To ensure that each vertex is visited only once, skip a vertex if it has already been visited.

24.7.1 Breadth-First Search Algorithm

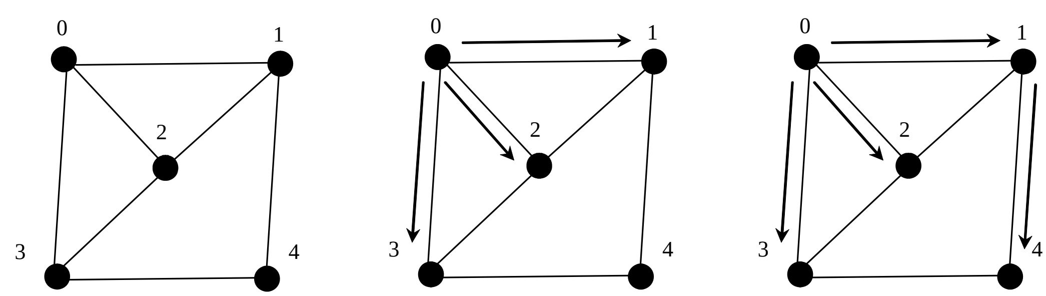
The algorithm for the breadth-first search starting from vertex *v* in a graph is described in Listing 24.7.

Listing 24.7 Breadth-first Search Algorithm

```
Input: G = (V, E) and a starting vertex v
```

```
Output: a BFS tree rooted at v

Tree bfs(vertex v)
{
  create an empty queue for storing vertices to be visited;
  add v into the queue;
  mark v visited;

  while the queue is not empty
  {
    dequeue a vertex, say u, from the queue
    visit u;
    for each neighbor w of u
    if w has not been visited
    {
      add w into the queue;
      mark w visited;
      parent[w] = v;
    }
  }
}
```

Consider the graph in Figure 24.11a. Suppose you start the breadth-first search from vertex 0. First visit 0, then all its visited neighbors, 1, 2, and 3, as shown in 24.11b. Vertex 1 has three neighbors, 0, 2, and 4. Since 0 and 2 have already been visited, you will now visit just 4, as shown in Figure 24.11c. Vertex 2 has three neighbors, 0, 1, and 3, which have all been visited. Vertex 3 has three neighbors, 0, 2, and 4, which have all been visited. Vertex 4 has two neighbors, 1 and 3, which have all been visited. So, the search ends.



(a) **(b)** **(c)**

Figure 24.11

Breadth-first search visits a node, then its neighbors, and then its neighbors' neighbors, and so on.

Since each edge and vertex is visited only once, the time complexity of the **`bfs`** function is **`O(|E| + |V|)`**, where **`|E|`** denotes the number of edges and **`|V|`** the number of vertices.

24.7.2 Implementation of Breadth-First Search

The **`bfs(int v)`** function is defined in Listing 24.3 (lines 286–317). It returns an instance of the **`Tree`** class with vertex **`v`** as the root. The function stores the vertices searched in a list **`searchOrders`** (line 289), the parent of each vertex in an array **`parent`** (line 290), stores the vertices to be visited in a queue (line 294), and uses the **`isVisited`** array to indicate whether a vertex has been visited (line 295). The search starts from vertex **`v`**. **`v`** is added to the queue (line 296) and is marked visited (line 297). The function now examines each vertex **`u`** in the queue (line 299) and adds it to **`searchOrders`** (line 303). The function adds each unvisited neighbor **`w`** of **`u`** to the queue (line 309), set its parent to **`u`** (line 310), and mark it visited (line 311).

Listing 24.8 gives a test program that displays a BFS for the graph in Figure 24.1, starting from **`Chicago`**.

Listing 24.8 TestBFS.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include "Graph.h" // Defined in Listing 24.2
using namespace std;

int main()
{
  // Vertices for graph in Figure 24.1
  string vertices[] = {"Seattle", "San Francisco", "Los Angeles",
    "Denver", "Kansas City", "Chicago", "Boston", "New York",
    "Atlanta", "Miami", "Dallas", "Houston"};

  // Edge array for graph in Figure 24.1
  int edges[][2] = {
    {0, 1}, {0, 3}, {0, 5},
    {1, 0}, {1, 2}, {1, 3},
    {2, 1}, {2, 3}, {2, 4}, {2, 10},
    {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
    {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
    {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
    {6, 5}, {6, 7},
    {7, 4}, {7, 5}, {7, 6}, {7, 8},
    {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
    {9, 8}, {9, 11},
    {10, 2}, {10, 4}, {10, 8}, {10, 11},
    {11, 8}, {11, 9}, {11, 10}
  };
  const int NUMBER_OF_EDGES = 46; // 46 edges in Figure 24.1
```

```
30
31    // Create a vector for vertices
32    vector<string> vectorOfVertices(12);
33    copy(vertices, vertices + 12, vectorOfVertices.begin());
34
35    Graph<string> graph(vectorOfVertices, edges, NUMBER_OF_EDGES);
36    Tree dfs = graph.bfs(5); // Vertex 5 is Chicago
37
38    vector<int> searchOrders = dfs.getSearchOrders();
39    cout << dfs.getNumberOfVerticesFound() <<
40      " vertices are searched in this BFS order:" << endl;
41    for (unsigned i = 0; i < searchOrders.size(); i++)
42      cout << graph.getVertex(searchOrders[i]) << " ";
43    cout << endl << endl;
44
45    for (unsigned i = 0; i < searchOrders.size(); i++)
46      if (dfs.getParent(i) != -1)
47        cout << "parent of " << graph.getVertex(i) <<
48          " is " << graph.getVertex(dfs.getParent(i)) << endl;
49
50    return 0;
51  }
```

Sample output

```
12 vertices are searched in this BFS order:
  Chicago Seattle Denver Kansas City Boston New York
  San Francisco Los Angeles Atlanta Dallas Miami Houston

parent of Seattle is Chicago
parent of San Francisco is Seattle
parent of Los Angeles is Denver
parent of Denver is Chicago
parent of Kansas City is Chicago
parent of Boston is Chicago
parent of New York is Chicago
parent of Atlanta is Kansas City
parent of Miami is Atlanta
parent of Dallas is Kansas City
parent of Houston is Atlanta
```

The program creates a graph for Figure 24.1 in line 35 and obtains a DFS tree starting from vertex Chicago in line 36. The search order is obtained in line 38. The graphical illustration of the BFS starting from Chicago is shown in Figure 24.12.

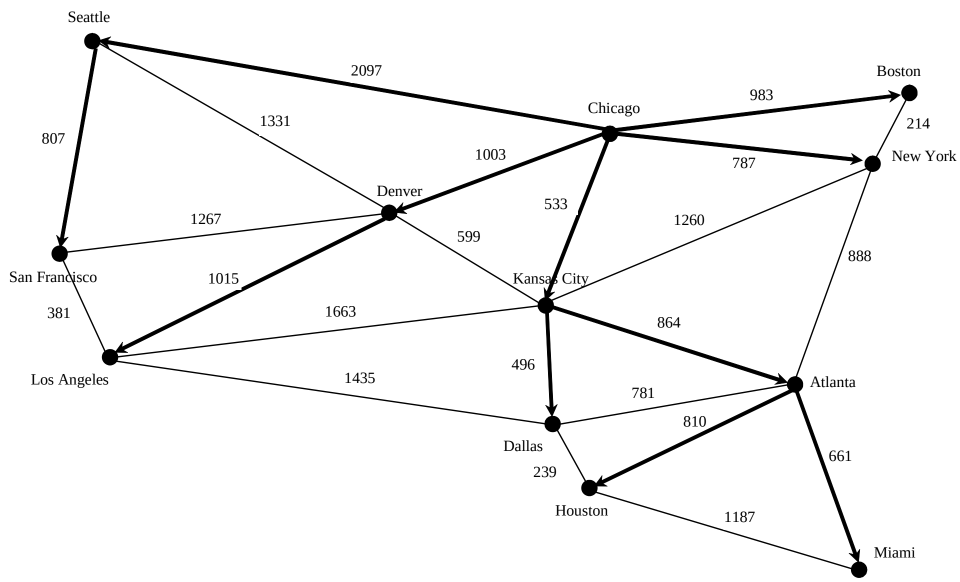


Figure 24.12

BFS search starts from Chicago.

24.7.3 Applications of the BFS

Many of the problems solved by the DFS can also be solved using the breadth-first search. Specifically, the BFS can be used to solve the following problems:

- Detecting whether a graph is connected. A graph is connected if there is a path between any two vertices in the graph.

- Detecting whether there is a path between two vertices.

- Finding a shortest path between two vertices. You can prove that the path between the root and any node in the BFS tree is the shortest path between the root and the node (see Review Question 24.10).
- Finding all connected components. A connected component is a maximal connected subgraph in which every pair of vertices are connected by a path.

- Detecting whether there is a cycle in the graph.
- Finding a cycle in the graph.
- Testing whether a graph is bipartite. A graph is bipartite if its vertices can be divided into two disjoint sets such that no edges exist between vertices in the same set.

Check point

24.15 What is the return type from invoking **bfs(v)**?

24.16 What is breadth-first search?

24.17 Draw a BFS tree for the graph in Figure 24.3b starting from node **A**.

24.18 Draw a BFS tree for the graph in Figure 24.1 starting from vertex **Atlanta**.

24.19 Prove that the path between the root and any node in the BFS tree is the shortest path between the root and the node.

24.8 Case Study: The Nine Tail Problem

Key Point: *The nine tails problem can be reduced to the shortest path problem.*

The DFS and BFS algorithms have many applications. This section applies the BFS to solve the nine tail problem.

The problem is stated as follows. Nine coins are placed in a three-by-three matrix with some face up and some face down. A legal move is to take any coin that is face up and reverse it, together with the coins adjacent to it (this does not include coins that are diagonally adjacent). Your task is to find the minimum number of moves that lead to all coins being face down. For example, you start with the nine coins as shown in Figure 24.13a. After flipping the second coin in the last row, the nine coins are now as shown in Figure 24.13b. After flipping the second coin in the first row, the nine coins are all face down, as shown in 24.13c.

H	H	H
T	T	T
H	H	H

(a)

H	H	H
T	H	T
T	T	T

(b)

T	T	T
T	T	T
T	T	T

(c)

Figure 24.13

The problem is solved when all coins are face down.

We will write a C++ program that prompts the user to enter an initial state of the nine coins and displays the solution, as shown in the following sample run.

Sample output

```
Enter an initial nine coin H and T's:
HHH
TTT
HHH

The steps to flip the coins are
HHH
TTT
HHH

HHH
THT
TTT

TTT
TTT
TTT
```

Each state of the nine coins represents a node in the graph. For example, the three states in Figure 24.13 correspond to three nodes in the graph. Intuitively, you can use a 3 × 3 matrix to represent all nodes and use **0** for head and **1** for tail. Since there are nine cells and each cell is either **0** or **1**, there are a total of 2^9 (512) nodes, labeled **0**, **1**, ..., and **511**, as shown in Figure 24.14.

0	0	0
0	0	0
0	0	0

0

0	0	0
0	0	0
0	0	1

1

0	0	0
0	0	0
0	1	0

2

0	0	0
0	0	0
0	1	1

3

…..

1	1	1
1	1	1
1	1	1

512

Figure 24.14

*There are total of **512** nodes, labeled in this order as **0**, **1**, **2**, ..., and **511**.*

We assign an edge from node **`u`** to **`v`** if there is a legal move from **`v`** to **`u`**. Figure 24.15 shows the directed edges to node **`56`**.

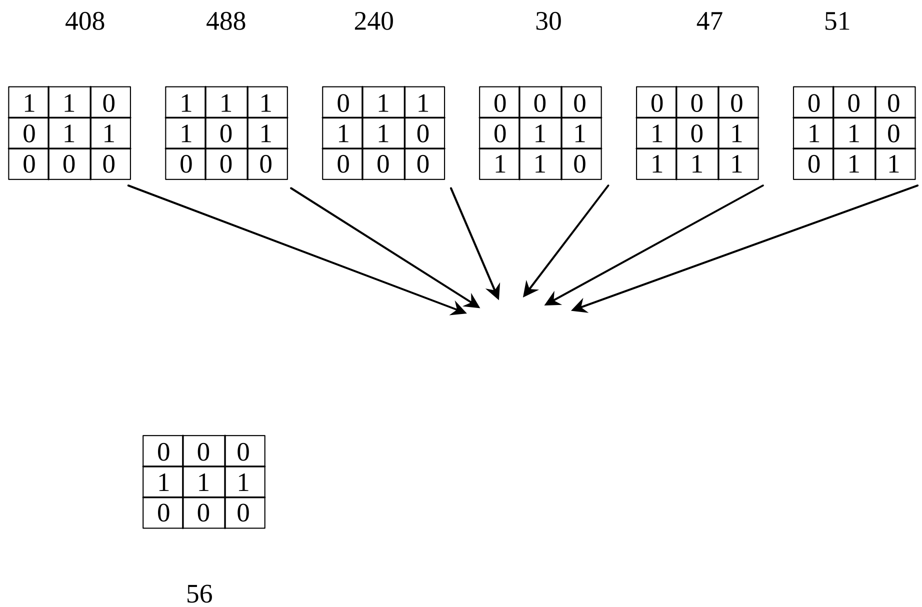


Figure 24.15

*If node v becomes node u after flipping cells, assign an edge from **`u`** to **`v`**.*

The last node in Figure 24.14 represents the state of nine face-down coins. For convenience, we call this last node the *target node*. So, the target node is labeled **`511`**. Suppose the initial state of the nine tail problem corresponds to the node **`s`**. The problem is reduced to finding a shortest path from the target node to **`s`** in a BFS tree rooted at the target node.

Now the task is to build a graph that consists of **`512`** nodes labeled **`0`**, **`1`**, **`2`**, ..., **`511`**, and edges among the nodes. Once the graph is created, obtain a BFS tree rooted at node **`511`**. From the BFS tree, you can find the shortest path from the root to any vertex. We will create a class named **`NineTailModel`**, which contains the function to get the shortest path from the target node to any other node. The class UML diagram is shown in Figure 24.16.

NineTailModel	
#tree: Tree*	A tree rooted at node 511.
+NineTailModel()	Constructs a model for the nine tail problem and obtains the tree.
+getShortestPath(nodeIndex: int): vector<int>	Returns a path from the specified node to the root. The path returned consists of the node labels in a vector.
+getNode(index: int): vector<char>	Returns a node consisting of nine characters of H's and T's.
+getIndex(node: vector<char>&): int	Returns the index of the specified node.
+printNode(node: vector<char>&): void	Displays the node to the console.
#getEdges(): vector<Edge> const	Returns a vector of Edge objects for the graph.
#getFlippedNode(node: vector<char>&, position: int): int const	Flips the node at the specified position and returns the index of the flipped node.
#flipACell(node: vector<char>&, row: int, column: int): void	Flips the node at the specified row and column.

Figure 24.16

The **`NineTailModel`** *class models the Nine Tail problem using a graph.*

Visually, a node is represented in a 3 × 3 matrix with letters H and T. In a C++ program, you can use a vector of nine characters to represent a node. The **`getNode(index)`** function returns the node for the specified index. For example, **`getNode(0)`** returns the node that contains nine H's. **`getNode(511)`** returns the node that contains nine T's. The **`getIndex(node)`** function returns the index of the node. The **`printNode(node)`** function displays the node visually on the console.

Note that the data field **`tree`** and functions **`getEdges()`**, **`getFlippedNode(node, position)`**, and **`flipACell(&node, row, column)`** are defined protected so that they can be accessed from child classes in the next chapter.

The **`getEdges()`** function returns a vector of **`Edge`** objects.

The **`getFlippedNode(node, position)`** function flips the node at the specified position and returns the index of the new node. For example, for node **`56`** in Figure 24.16, flip it at position **`0`**, and you will get node **`51`**. If you flip node 56 at position **`1`**, you will get node **`47`**.

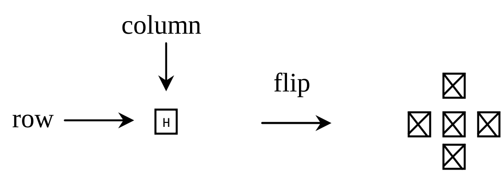
The **flipACell(&node, row, column)** function flips a node at the specified row and column. For example, if you flip node **56** at row **0** and column **0**, the new node is **408**. If you flip node **56** at row **2** and column **0**, the new node is **30**.

Listing 24.9 shows the source code for NineTailModel.h.

Listing 24.9 NineTailModel.h

```
#ifndef NINETAILMODEL_H
#define NINETAILMODEL_H

#include <iostream>
#include "Graph.h" // Defined in Listing 24.2

using namespace std;

const int NUMBER_OF_NODES = 512;

class NineTailModel
{
public:
  // Construct a model for the Nine Tail problem
  NineTailModel();

  // Return the index of the node
  int getIndex(vector<char>& node) const;

  // Return the node for the index
  vector<char> getNode(int index) const;

  // Return the shortest path of vertices from the specified
  // node to the root
  vector<int> getShortestPath(int nodeIndex) const;

  // Print a node to the console
  void printNode(vector<char>& node) const;

protected:
  Tree* tree;

  // Return a vector of Edge objects for the graph
  // Create edges among nodes
  vector<Edge> getEdges() const;

  // Return the index of the node that is the result of flipping
  // the node at the specified position
  int getFlippedNode(vector<char>& node, int position) const;

  // Flip a cell at the specified row and column
  void flipACell(vector<char>& node, int row, int column) const;
};

NineTailModel::NineTailModel()
{
  // Create edges
  vector<Edge> edges = getEdges();
```

```
  // Build a graph
  Graph<int> graph(NUMBER_OF_NODES, edges);

  // Build a BFS tree rooted at the target node
  tree = new Tree(graph.bfs(511));
}

vector<Edge> NineTailModel::getEdges() const
{
  vector<Edge> edges;

  for (int u = 0; u < NUMBER_OF_NODES; u++)
  {
    for (int k = 0; k < 9; k++)
    {
      vector<char> node = getNode(u);
      if (node[k] == 'H')
      {
        int v = getFlippedNode(node, k);
        // Add edge (v, u) for a legal move from node u to node v
        edges.push_back(Edge(v, u));
      }
    }
  }

  return edges;
}

int NineTailModel::getFlippedNode(vector<char>& node, int position)
  const
{
  int row = position / 3;
  int column = position % 3;

  flipACell(node, row, column);
  flipACell(node, row - 1, column);
  flipACell(node, row + 1, column);
  flipACell(node, row, column - 1);
  flipACell(node, row, column + 1);

  return getIndex(node);
}

void NineTailModel::flipACell(vector<char>& node,
  int row, int column) const
{
  if (row >= 0 && row <= 2 && column >= 0 && column <= 2)
  { // Within boundary
    if (node[row * 3 + column] == 'H')
      node[row * 3 + column] = 'T'; // Flip from H to T
    else
      node[row * 3 + column] = 'H'; // Flip from T to H
  }
}

int NineTailModel::getIndex(vector<char>& node) const
{
  int result = 0;

```



For example:
node: THHHHHHTT
index: 259

T	H	H
H	H	H
H	T	T

```
109    for (int i = 0; i < 9; i++)
110      if (node[i] == 'T')
111        result = result * 2 + 1;
112      else
113        result = result * 2 + 0;
114
115    return result;
116  }
117
118  vector<char> NineTailModel::getNode(int index) const
119  {
120    vector<char> result(9);
121
122    for (int i = 0; i < 9; i++)
123    {
124      int digit = index % 2;
125      if (digit == 0)
126        result[8 - i] = 'H';
127      else
128        result[8 - i] = 'T';
129      index = index / 2;
130    }
131
132    return result;
133  }
134
135  vector<int> NineTailModel::getShortestPath(int nodeIndex) const
136  {
137    return tree->getPath(nodeIndex);
138  }
139
140  void NineTailModel::printNode(vector<char>& node) const
141  {
142    for (int i = 0; i < 9; i++)
143      if (i % 3 != 2)
144        cout << node[i];
145      else
146        cout << node[i] << endl;
147
148    cout << endl;
149  }
150
151  #endif
```

For example:
node: THHHHHHTT
index: 259

```
T H H
H H H
H T T
```

For example:
node: THHHHHHTT
output:

```
T H H
H H H
H T T
```

The constructor (lines 45–55) creates a graph with 512 nodes, and each edge corresponds to the move from one node to the other (line 48). From the graph, a BFS tree rooted at the target node **511** is obtained and assigned to a pointer variable **tree** (line 54). We use a pointer for **tree** to enable tree to reference any type of **Tree**. Later in the next chapter, we will assign a **ShortestPathTree** to tree and **ShortestPathTree** is a subtype of **Tree**.

To create edges, the **getEdges** function (lines 57–76) checks each node **u** to see if it can be flipped to another node **v**. If so, add (**v**, **u**) to the **Edge** vector (line 70). The **getFlippedNode(node, position)** function finds a flipped node by flipping an **H** cell and its neighbors in a node (lines 78–91). The **flipACell(node, row, column)** function actually flips an **H** cell and its neighbors in a node (lines 93–103).

Note that the argument **node** is passed by value in **getFlippedNode(node, position)**. What would happen if it is mistakenly passed by reference? After invoking **getFlippedNode(node, k)** line 68, the contents of **node** will be changed and **node** no longer corresponds to vertex **u**.

The **getIndex(node)** function is implemented in the same way as converting a binary number to a decimal (lines 105–116). The **getNode(index)** function returns a node consisting of letters **H** and **T**'s (lines 118–133).

The **getShortestpath(nodeIndex)** function invokes the **getPath(nodeIndex)** function to get the vertices in the shortest path from the specified node to the target node (lines 135–138).

Listing 24.10 gives a program that prompts the user to enter an initial node and displays the steps to reach the target node.

Listing 24.10 NineTail.cpp

```
#include <iostream>
#include <vector>
#include "NineTailModel.h"
using namespace std;

int main()
{
  // Prompt the user to enter nine coins H and T's
  cout << "Enter an initial nine coin H’s and T's: ";
  vector<char> initialNode(9);

  for (int i = 0; i < 9; i++)
    cin >> initialNode[i];

```

```
15    cout << "The steps to flip the coins are " << endl;
16    NineTailModel model;
17    vector<int> path =
18      model.getShortestPath(model.getIndex(initialNode));
19
20    for (unsigned i = 0; i < path.size(); i++)
21      model.printNode(model.getNode(path[i]));
22
23    return 0;
24  }
```

The program prompts the user to enter an initial node with nine letters **H's** and **T**'s in lines 9–13, creates a model to create a graph and get the BFS tree (line 16), obtains a shortest path from the initial node to the target node (lines 16–17), and displays the nodes in the path (lines 17–18).

Check point

24.20 How are the nodes created for the graph in `NineTailModel`?

24.21 How are the edges created for the graph in `NineTailModel`?

24.22 What is returned after invoking `getIndex("HTHTTTHHH".toCharArray())` in Listing 24.13? What is returned after invoking `getNode(46)` in Listing 24.13?

24.23 The statement in line 66 in Listing 24.13 NineTailModel.cpp is

```
vector<char> node = getNode(u);
```

Will the program work if it is moved to before line 64? Explain the reason.

24.25 If the `flipACell` function in NineTailModel.h is redefined as follows:

```
void flipACell(vector<char> node, int row, int column);
```

what is wrong?

Key Terms

- **adjacency list**
- **adjacent vertices**
- **adjacency matrix**
- **breadth-first search**
- **complete graph**
- **degree**

- **depth-first search**
- **directed graph**
- **graph**
- **incident edges**
- **parallel edge**
- **Seven Bridges of Königsberg**
- **simple graph**
- **spanning tree**
- **weighted graph**
- **undirected graph**
- **unweighted graph**

Chapter Summary

1. A graph is a useful mathematical structure that represents relationships among entities in the real world.
2. A graph may be directed or undirected. In a directed graph, each edge has a direction, which indicates that you can move from one vertex to the other through the edge.
3. Edges may be weighted or unweighted. A weighted graph has weighted edges.
4. You can model graphs using classes and interfaces.
5. You can represent vertices and edges using arrays and linked lists.
6. Graph traversal is the process of visiting each vertex in the graph exactly once. Two popular ways of traversing a graph are depth-first search and breadth-first search.
7. The depth-first search of a graph first visits a vertex, then recursively visits all unvisited vertices adjacent to that vertex.
8. The breadth-first search of a graph first visits a vertex, then all its adjacent unvisited vertices, then all the unvisited vertices adjacent to those vertices, and so on.

9. DFS and BFS can be used to solve many problems, such as detecting whether a graph is connected, detecting whether there is a cycle in the graph, and finding a shortest path between two vertices.

Quiz

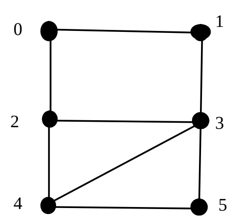
Do the quiz for this chapter online at www.cs.armstrong.edu/liang/cpp3e/quiz.html.

Programming Exercises

Sections 24.6-24.7

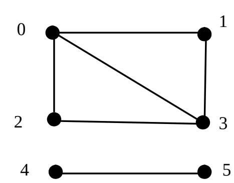
24.1* (*Test whether a graph is connected*) Write a program that reads a graph from a file and determines whether the graph is connected. The first line in the file contains a number that indicates the number of vertices (**`n`**). The vertices are labeled as **`0`**, **`1`**, ..., **`n-1`**. Each subsequent line, with the format **`u: v1, v2, ...`**, describes edges (**`u`**, **`v1`**), (**`u`**, **`v2`**), etc. Figure 24.17 gives the examples of two files for their corresponding graphs.

```
File
6
0: 1, 2
1: 0, 3
2: 0, 3, 4
3: 1, 2, 4, 5
4: 2, 3, 5
5: 3, 4
```



(a)

```
File
6
0: 1, 2, 3
1: 0, 3
2: 0, 3
3: 0, 1, 2
4: 5
5: 4
```



(b)

Figure 24.17

The vertices and edges of a graph can be stored in a file.

Your program should prompt the user to enter the name of the file, reads data from a file, creates an instance **`g`** of **`Graph`**, invokes **`g.printEdges()`** to display all edges, and invokes **`dfs(0)`** to obtain an instance **`tree`** of **`Tree`**. If **`tree.getNumberOfVerticeFound()`** is the same as the number of vertices in the graph, the graph is connected. Here is a sample run of the program:

Sample output

```
Enter a file name: c:\exercise\Exercise24_1a.txt
The number of vertices is 6
Vertex 0: (0, 1) (0, 2)
Vertex 1: (1, 0) (1, 3)
Vertex 2: (2, 0) (2, 3) (2, 4)
Vertex 3: (3, 1) (3, 2) (3, 4) (3, 5)
Vertex 4: (4, 2) (4, 3) (4, 5)
Vertex 5: (5, 3) (5, 4)
The graph is connected
```

(Hint: Use **`Graph(numberOfVertices, vectorOfEdges)`** to create a graph, where **`vectorOfEdges`** contains a vector of **`Edge`** objects. Use **`Edge(u, v)`** to create an edge. Read the first line to get the number of vertices. Read each subsequent line to extract the vertices from the string and creates edges from the vertices.)

24.2* (*Create a file for graph*) Modify Listing 24.2, TestGraph.cpp, to create a file for representing **`graph1`**. The file format is described in Exercise 24.1. Create the file from the array defined in lines 16-29 in Listing 24.2. The number of vertices for the graph is **`12`**, which will be stored in the first line of the file. The contents of the file should be as follows:

12

0: 1, 3, 5

1: 0, 2, 3

2: 1, 3, 4, 10

3: 0, 1, 2, 4, 5

4: 2, 3, 5, 7, 8, 10

5: 0, 3, 4, 6, 7

6: 5, 7

7: 4, 5, 6, 8

8: 4, 7, 9, 10, 11

9: 8, 11

10: 2, 4, 8, 11

11: 8, 9, 10

24.3* (*Find a shortest path*) Write a program that reads a connected graph from a file. The graph is stored in a file using the same format specified in Exercise 24.1. Your program should prompt the user to enter the name of the file, then two vertices, and displays the shortest path between the two vertices. For example, for the graph in Figure 24.17a, a shortest path between **`0`** and **`5`** may be displayed as **`0 1 3 5`**.

Here is a sample run of the program:

Sample output

```
Enter a file name: c:\exercise\Exercise24_3a.txt

Enter two vertices (integer indexes): 0 5
The number of vertices is 6
Vertex 0: (0, 1) (0, 2)
Vertex 1: (1, 0) (1, 3)
Vertex 2: (2, 0) (2, 3) (2, 4)
Vertex 3: (3, 1) (3, 2) (3, 4) (3, 5)
Vertex 4: (4, 2) (4, 3) (4, 5)
Vertex 5: (5, 3) (5, 4)
The path is 0 1 3 5
```

24.4* (*Implementing DFS using a stack*) The depth-first search algorithm described in Listing 24.5 uses recursion. Implement it without using recursion.

24.5* (*Find connected components*) Add a new function in the **`Graph`** class to find all connected components in a graph with the following header:

```
vector<vector<int>> getConnectedComponents();
```

The function returns a vector. Each element in the vector is another vector that contains all the vertices in a connected component. For example, if the graph has three connected components, the function returns a vector with three elements, each of which contains the vertices in a connected component.

24.6* (*Find paths*) Add a new function in **Graph** to find a path between two vertices with the following header:

```
vector<int> getPath(int u, int v);
```

The function returns a vector that contains all the vertices in a path from **u** to **v** in this order. Using the BFS approach, you can obtain a shortest path from **u** to **v**. If there is no path from **u** to **v**, the function returns an empty vector.

24.7* (*Detect cycles*) Add a new function in **Graph** to determine whether there is a cycle in the graph with the following header:

```
bool containsCyclic();
```

24.8* (*Find a cycle*) Add a new function in **Graph** to find a cycle in the graph with the following header:

```
vector<int> getACycle();
```

The function returns a vector that contains all the vertices in a cycle from **u** to **v** in this order. If the graph has no cycles, the function returns an empty vector.

24.9** (*Test bipartite*) Recall that a graph is bipartite if its vertices can be divided into two disjoint sets such that no edges exist between vertices in the same set. Add a new function in **Graph** to detect whether the graph is bipartite:

```
bool isBipartite();
```

24.10** (*Get bipartite sets*) Add a new function in **Graph** to return two bipartite sets if the graph is bipartite:

```
vector<vector<int>> getBipartiteSets();
```

The function returns a vector. Each element in the vector is another vector that contains a set of vertices.

24.11** (*Variation of the nine tail problem*) In the nine tail problem, when you flip a head, the horizontal and vertical neighboring cells are also flipped. Rewrite the program, assuming that all neighboring cells including the diagonal neighbors are also flipped.

24.12** (*4 × 4 16 tail model*) The nine tail problem in the text uses a 3 × 3 matrix. Assume that you have 16 coins placed in a 4 × 4 matrix. Create a new model class named **`TailModel16`**. Create an instance of the model and save the object into a file named Exercise24_12.dat.

24.13** (*Induced subgraph*) Given an undirected graph G = (*V*, *E*) and an integer *k*, find an induced subgraph H of G of maximum size such that all vertices of H have degree >= *k*, or conclude that no such induced subgraph exists. Implement the function with the following header:

```
Graph<V> maxInducedSubgraph(Graph<V> g, int k)
```

The function returns an empty graph if such subgraph does not exist.

(*Hint*: An intuitive approach is to remove vertices whose degree is less than *k*. As vertices are removed with their adjacent edges, the degrees of other vertices may be reduced. Continue the process until no vertices can be removed, or all the vertices are removed.)