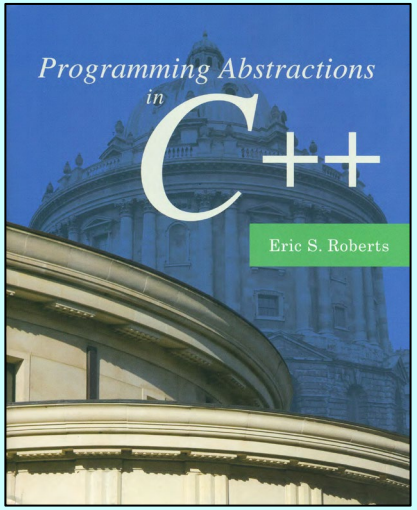
CHAPTER 18

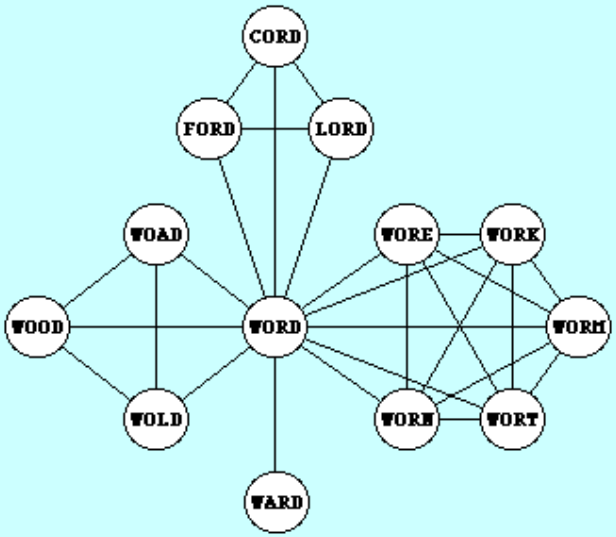
Graphs

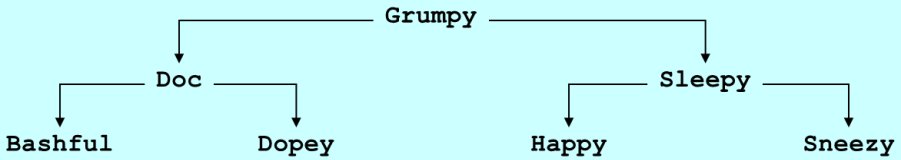
So I draw the world together link by link:
Yea, from Delos up to Limerick and back!

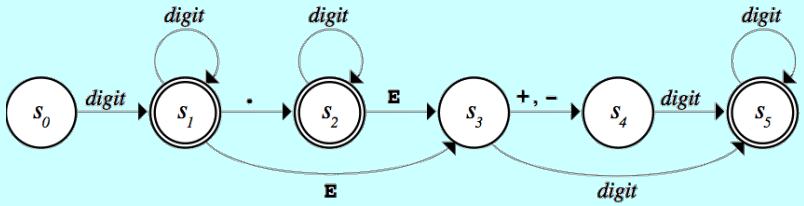
—Rudyard Kipling, "The Song of the Banjo," 1894

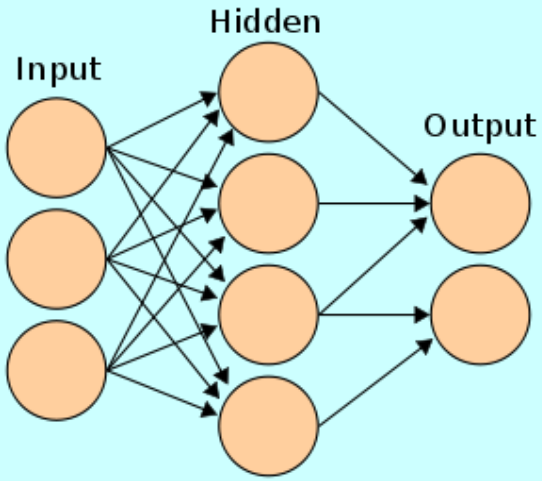


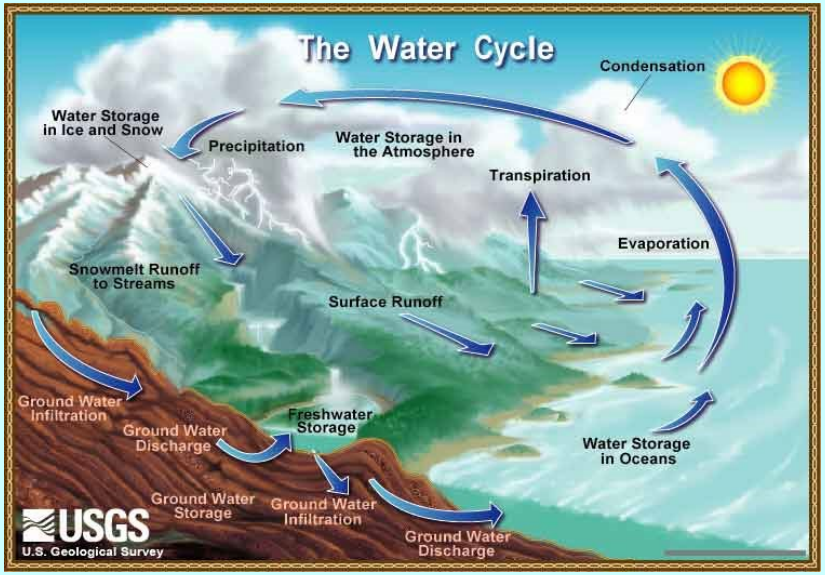
Examples of Graphs







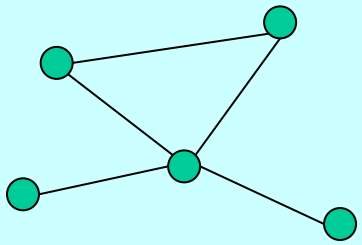




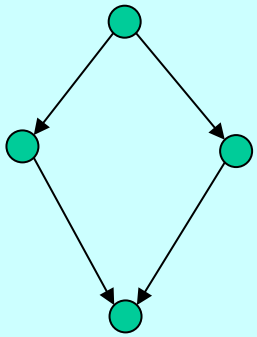
Example of Graphs

The Definition of a Graph

- A ***graph*** consists of a set of ***nodes*** together with a set of ***arcs***. The nodes correspond to the dots or circles in a graph diagram (which might be cities, words, neurons, or who knows what) and the arcs correspond to the links that connect two nodes.
- In some graphs, arcs are shown with an arrow that indicates the direction in which two nodes are linked. Such graphs are called ***directed graphs***.
- Other graphs, arcs are simple connections indicating that one can move in either direction between the two nodes. These graphs are ***undirected graphs***.



Undirected Graph



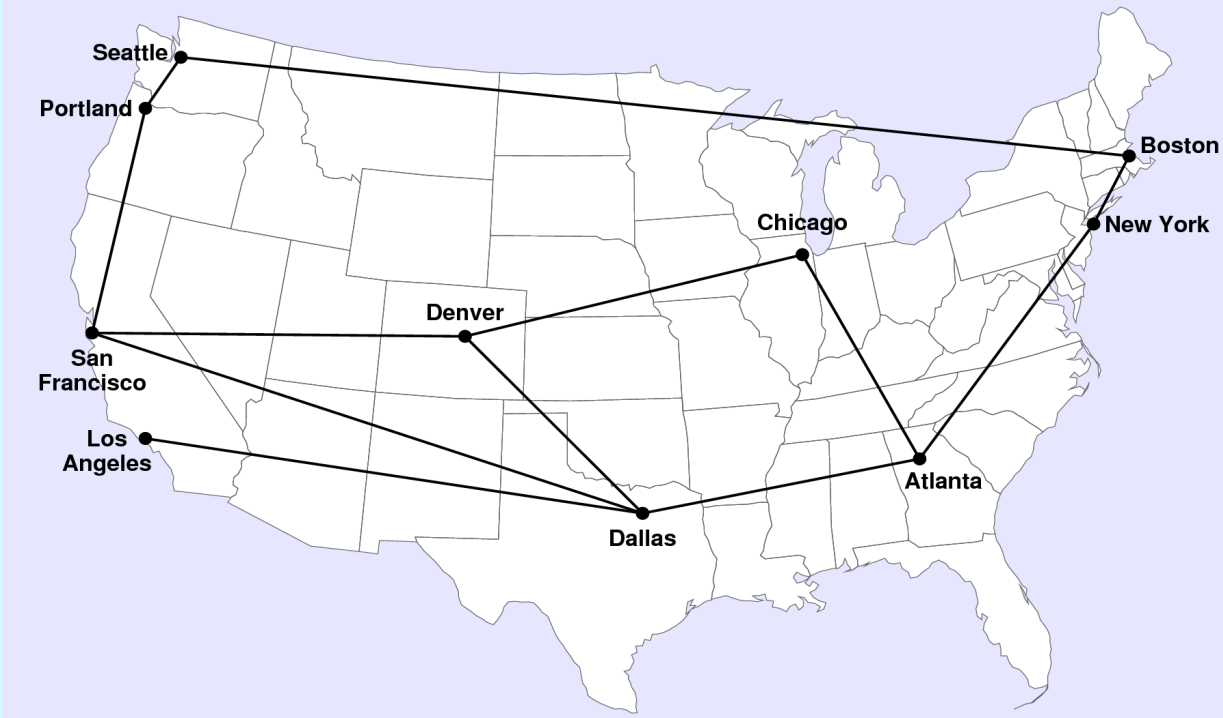
Directed Graph

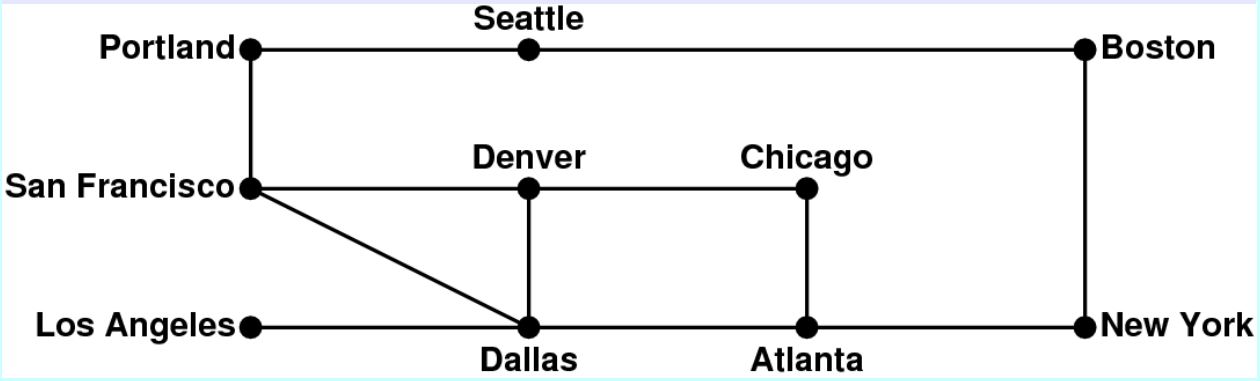
Graph Terminology

- Mathematicians often use the words ***vertex*** and ***edge*** for the structures that computer scientists call ***node*** and ***arc***. We'll stick with the computer science names, but you might see their mathematical counterparts in algorithmic descriptions.
- The nodes to which a particular node is connected are called its ***neighbors***.
- In an undirected graph, the number of connections from a node is called its ***degree***. In a directed graph, the number of arcs leading outward is called the ***out-degree***, and the number of arcs leading inward is called the ***in-degree***.
- A series of arcs that connect two nodes is called a ***path***. A path that returns to its starting point is called a ***cycle***. A ***simple path/cycle*** is a path/cycle without duplicated nodes.
- A graph in which there is a path between every pair of nodes is said to be ***strongly connected***. A directed graph is only ***weakly connected*** if its undirected counterpart is connected.

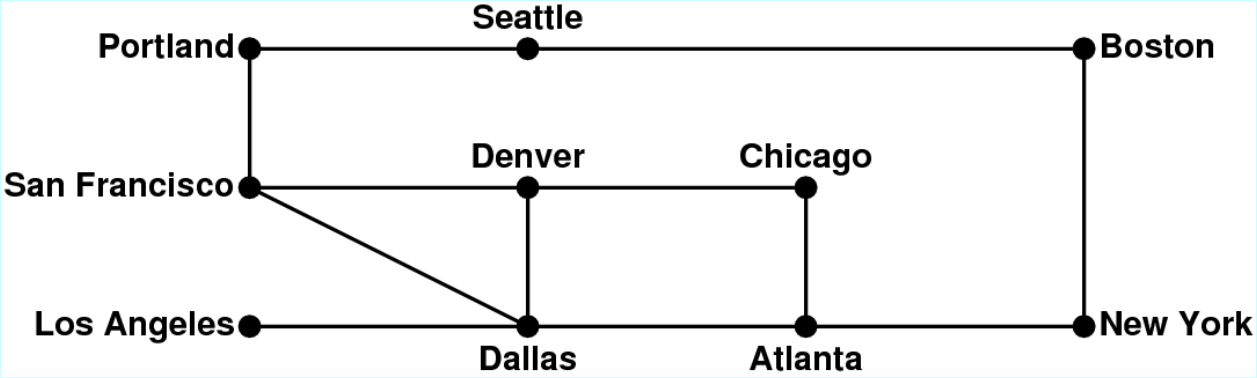
Graph Abstraction

FIGURE 18-1 Route map for a small airline serving ten cities





Graph Representations



$V =$ { Atlanta, Boston, Chicago, Dallas, Denver, Los Angeles, New York, Portland, San Francisco, Seattle }

$E =$ { Atlanta↔Chicago, Atlanta↔Dallas, Atlanta↔New York, Boston↔New York, Boston↔Seattle, Chicago↔Denver, Dallas↔Denver, Dallas↔Los Angeles, Dallas↔San Francisco, Denver↔San Francisco, Portland↔San Francisco, Portland↔Seattle }

Graph Representations

- Like most abstract structures, graphs can be implemented in different ways. The primary feature that differentiates these implementations is the strategy used to represent connections between nodes. In practice, the most common strategies are:
  - Storing the connections for each node in an ***adjacency list***.
  - Storing the connections for the entire graph in an ***adjacency matrix***.
  - Storing the connections for each node as a ***set of arcs***.

Graph Representations

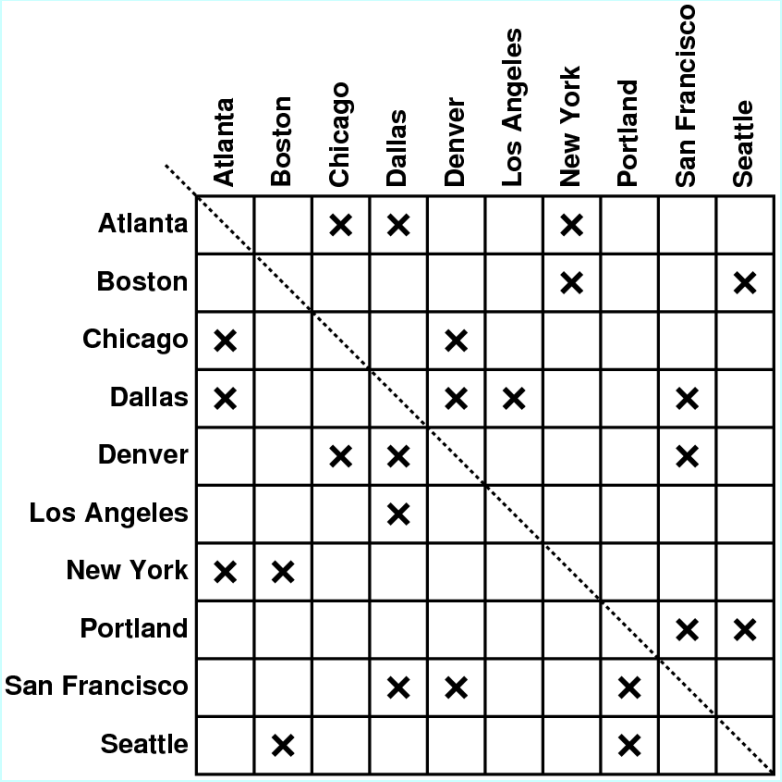
- Storing the connections for each node in an ***adjacency list***.

Atlanta → (Chicago, Dallas, New York)
Boston → (New York, Seattle)
Chicago → (Atlanta, Denver)
Dallas → (Atlanta, Denver, Los Angeles)
Denver → (Chicago, Dallas, San Francisco)
Los Angeles → (Dallas)
New York → (Atlanta, Boston)
Portland → (San Francisco, Seattle)
San Francisco → (Dallas, Denver, Portland)
Seattle → (Boston, Portland)

- Although simple and convenient, this representation can be inefficient when searching through the list of arcs associated with a node. This cost, $O(D)$, where D represents the degree of the originating node, becomes more significant in a ***dense*** graph (i.e., $D \rightarrow N$).

Graph Representations

- Storing the connections for the entire graph in an ***adjacency matrix***.



	Atlanta	Boston	Chicago	Dallas	Denver	Los Angeles	New York	Portland	San Francisco	Seattle
Atlanta			×	×			×			
Boston							×			×
Chicago	×				×					
Dallas	×				×	×			×	
Denver			×	×					×	
Los Angeles				×						
New York	×	×								
Portland									×	×
San Francisco				×	×			×		
Seattle		×						×		

- Similar to the idea of lookup table, this representation reduces the cost of checking for connections to $O(1)$. But the $O(N^2)$ storage space is inefficient for a ***sparse*** graph (i.e., $D << N$).

Graph Representations

- Storing the connections for each node as a ***set of arcs***. Start with a pair of sets:

```
struct StringBasedGraph {
   Set<string> nodes; /* Atlanta, … */
   Set<string> arcs; /* Atlanta_Chicago, … */
};
```

- This representation is conceptually simple and mirrors precisely the mathematical definition.
- Finding neighbors for any node requires going through every arc, which needs a collection class that supports iteration.
- Associating additional data with nodes and arcs requires using structures/classes to represent the nodes and arcs. E.g., many graph algorithms assign a numeric value to each of the arcs that indicates the cost of traversing that arc.

Three Strategies for Graph Abstraction

1. ***Use low-level structures***. This design uses the structure types **`Node`** and **`Arc`** to represent the components of a graph. This model gives clients complete freedom to extend these structures but offers no support for graph operations.
2. ***Adopt a hybrid strategy***. This design defines a **`Graph`** class as a parameterized class using templates so that it can use any structures or objects as the node and arc types. This strategy retains the flexibility of the low-level model and avoids the complexity associated with the class-based approach.
3. ***Define classes for each of the component types***. This design uses the **`Node`** and **`Arc`** classes to define the structure. In this model, clients define subclasses of the supplied types to particularize the graph data structure to their own application. More will be discussed later when we learn inheritance.

The Low-Level `graphtypes.h` Interface

```
/*
 * File: graphtypes.h
 * ------------------
 * This file defines low-level data structures that represent graphs.
 */

#ifndef _graphtypes_h
#define _graphtypes_h

#include <string>
#include "map.h"
#include "set.h"

struct Node;      /* Forward references to these two types so  */
struct Arc;       /* that the C++ compiler can recognize them. */

/*
 * Type: SimpleGraph
 * -----------------
 * This type represents a graph and consists of a set of nodes, a set of
 * arcs, and a map that creates an association between names and nodes.
 */

struct SimpleGraph {
   Set<Node *> nodes;
   Set<Arc *> arcs;
   Map<std::string,Node *> nodeMap;
};
```

The Low-Level `graphtypes.h` Interface

```
/*
 * Type: Node
 * ----------
 * This type represents an individual node and consists of the
 * name of the node and the set of arcs from this node.
 */

struct Node {
   std::string name;
   Set<Arc *> arcs;
};

/*
 * Type: Arc
 * ---------
 * This type represents an individual arc and consists of pointers
 * to the endpoints, along with the cost of traversing the arc.
 */

struct Arc {
   Node *start;
   Node *finish;
   double cost;
};

#endif
```

FIGURE 18-4 Program to create the airline graph

```
/*
 * File: AirlineGraph.cpp
 * ----------------------
 * This program initializes the graph for the airline example and then
 * prints the adjacency lists for each of the cities.
 */

#include <iostream>
#include <string>
#include "graphtypes.h"
#include "set.h"
using namespace std;

/* Function prototypes */

void printAdjacencyLists(SimpleGraph & g);
void initAirlineGraph(SimpleGraph & airline);
void addFlight(SimpleGraph & airline, string c1, string c2, int miles);
void addNode(SimpleGraph & g, string name);
void addArc(SimpleGraph & g, Node *n1, Node *n2, double cost);
```

```
/* Main program */

int main() {
   SimpleGraph airline;
   initAirlineGraph(airline);
   printAdjacencyLists(airline);
   return 0;
}

/*
 * Function: printAdjacencyLists
 * Usage: printAdjacencyLists(g);
 * ------------------------------
 * Prints out the adjacency list for each city in the graph.
 */

void printAdjacencyLists(SimpleGraph & g) {
   for (Node *node : g.nodes) {
      cout << node->name << " -> ";
      bool first = true;
      for (Arc *arc : node->arcs) {
         if (!first) cout << ", ";
         cout << arc->finish->name;
         first = false;
      }
      cout << endl;
   }
}
```

FIGURE 18-4 Program to create the airline graph (continued)

```
/*
 * Function: initAirlineGraph
 * Usage: initAirlineGraph(airline);
 * ---------------------------------
 * Initializes the airline graph to hold the flight data from Figure 19-2.
 * In a real application, the program would almost certainly read this
 * information from a data file.
 */

void initAirlineGraph(SimpleGraph & airline) {
   addNode(airline, "Atlanta");
   addNode(airline, "Boston");
   addNode(airline, "Chicago");
   addNode(airline, "Dallas");
   addNode(airline, "Denver");
   addNode(airline, "Los Angeles");
   addNode(airline, "New York");
   addNode(airline, "Portland");
   addNode(airline, "San Francisco");
   addNode(airline, "Seattle");
   addFlight(airline, "Atlanta", "Chicago", 599);
   addFlight(airline, "Atlanta", "Dallas", 725);
   addFlight(airline, "Atlanta", "New York", 756);
   addFlight(airline, "Boston", "New York", 191);
   addFlight(airline, "Boston", "Seattle", 2489);
   addFlight(airline, "Chicago", "Denver", 907);
   addFlight(airline, "Dallas", "Denver", 650);
   addFlight(airline, "Dallas", "Los Angeles", 1240);
   addFlight(airline, "Dallas", "San Francisco", 1468);
   addFlight(airline, "Denver", "San Francisco", 954);
   addFlight(airline, "Portland", "San Francisco", 550);
   addFlight(airline, "Portland", "Seattle", 130);
}
```

```
/*
 * Function: addFlight
 * Usage: addFlight(airline, c1, c2, miles);
 * ----------------------------------------
 * Adds an arc in each direction between the cities c1 and c2.
 */

void addFlight(SimpleGraph & airline, string c1, string c2, int miles) {
   Node *n1 = airline.nodeMap[c1];
   Node *n2 = airline.nodeMap[c2];
   addArc(airline, n1, n2, miles);
   addArc(airline, n2, n1, miles);
}
```

FIGURE 18-4 Program to create the airline graph (continued)

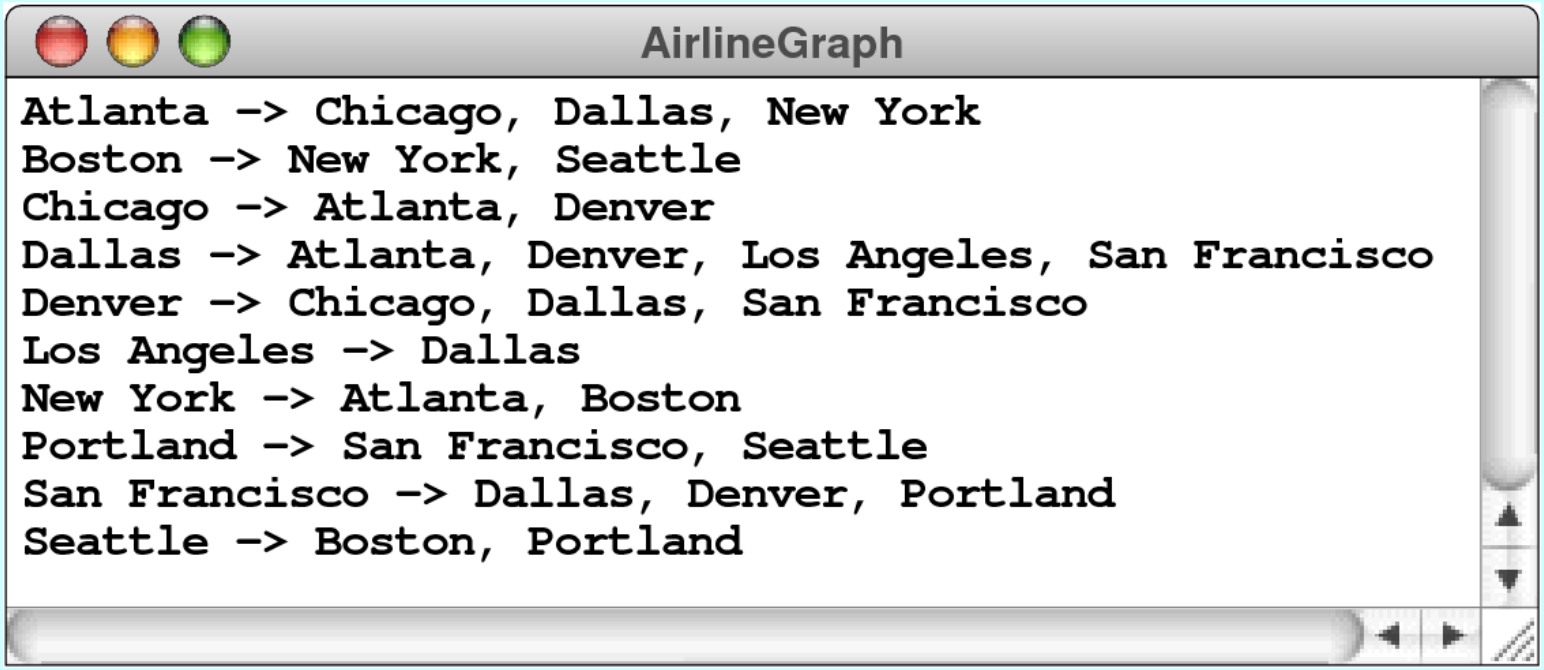
```
/*
 * Function: addNode
 * Usage: addNode(g, name);
 * ------------------------
 * Adds a new node with the specified name to the graph.
 */

void addNode(SimpleGraph & g, string name) {
   Node *node = new Node;
   node->name = name;
   g.nodes.add(node);
   g.nodeMap[name] = node;
}

/*
 * Function: addArc
 * Usage: addArc(g, n1, n2, cost);
 * -------------------------------
 * Adds a directed arc to the graph connecting n1 to n2.
 */

void addArc(SimpleGraph & g, Node *n1, Node *n2, double cost) {
   Arc *arc = new Arc;
   arc->start = n1;
   arc->finish = n2;
   arc->cost = cost;
   g.arcs.add(arc);
   n1->arcs.add(arc);
}
```

Using the `graphtypes.h` Interface



Graph Traversals

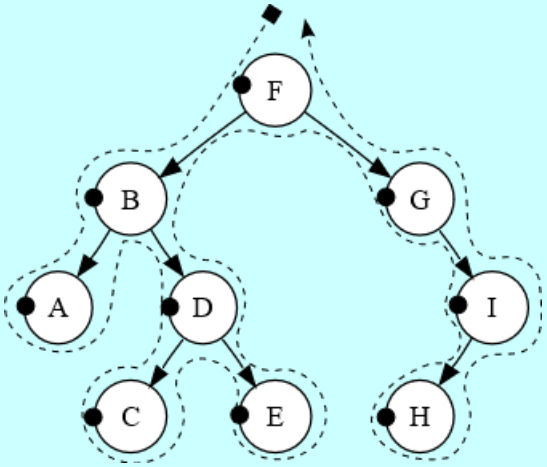
- The process of visiting each node in a graph by moving along its arcs is called ***traversing*** the graph. At each node, a function **`visit`** can be called to perform some operations, e.g., simply printing the name of the node being visited.

```
void visit(Node *node) {
   cout << node->name << endl;
}
```

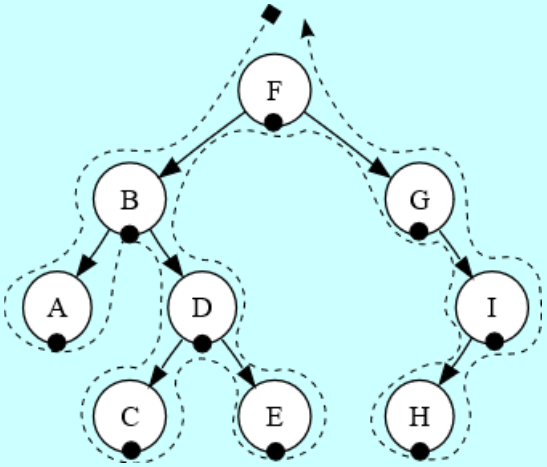
- The goal of a traversal is to call **`visit`** once, and only once, on every node in an order determined by the connections, which requires some structure to keep track of nodes that have already been visited. Common strategies are to include a **`visited`** flag in each node or to pass a set of visited nodes, as shown in the following code:

```
void depthFirstSearch(Node *node) {
   Set<Node *> visited;
   visitUsingDFS(node, visited);
}
```

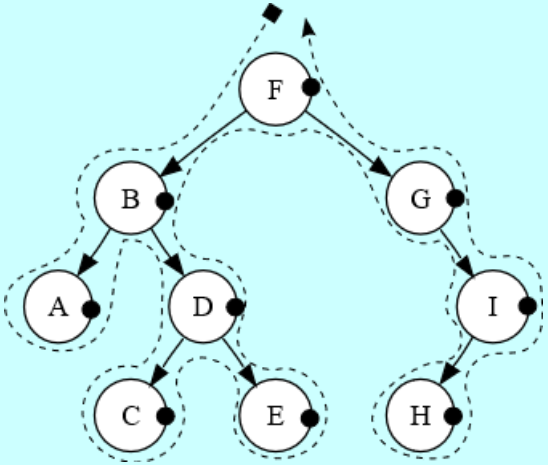
Example: DFS/BFS Tree Traversal



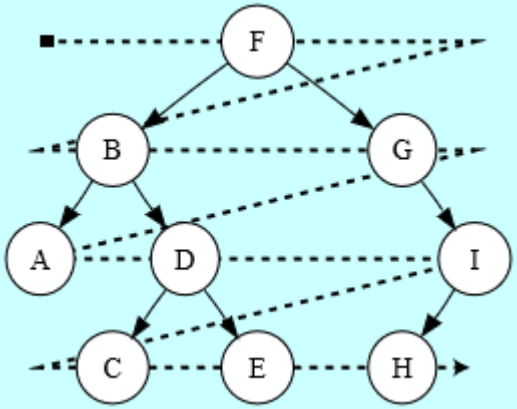
Pre-order: F, B, A, D, C, E, G, I, H.



In-order: A, B, C, D, E, F, G, H, I.



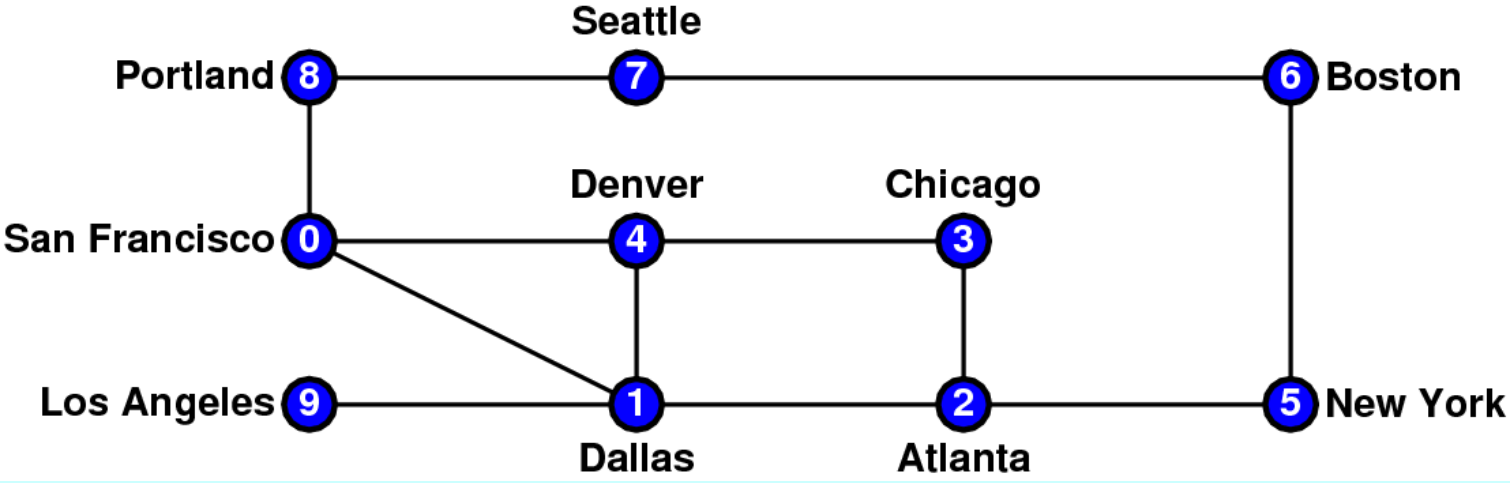
Post-order: A, C, E, D, B, H, I, G, F.



Level-order: F, B, G, A, D, I, C, E, H.

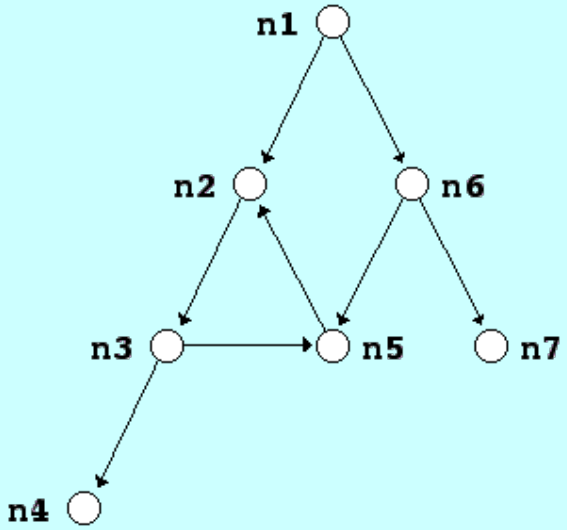
Depth-First Search

- The traversal strategy of ***depth-first search*** (or ***DFS*** for short) recursively processes the graph, following each branch, assuming that the neighboring nodes are stored in alphabetical order, visiting nodes as it goes, until every node is visited.



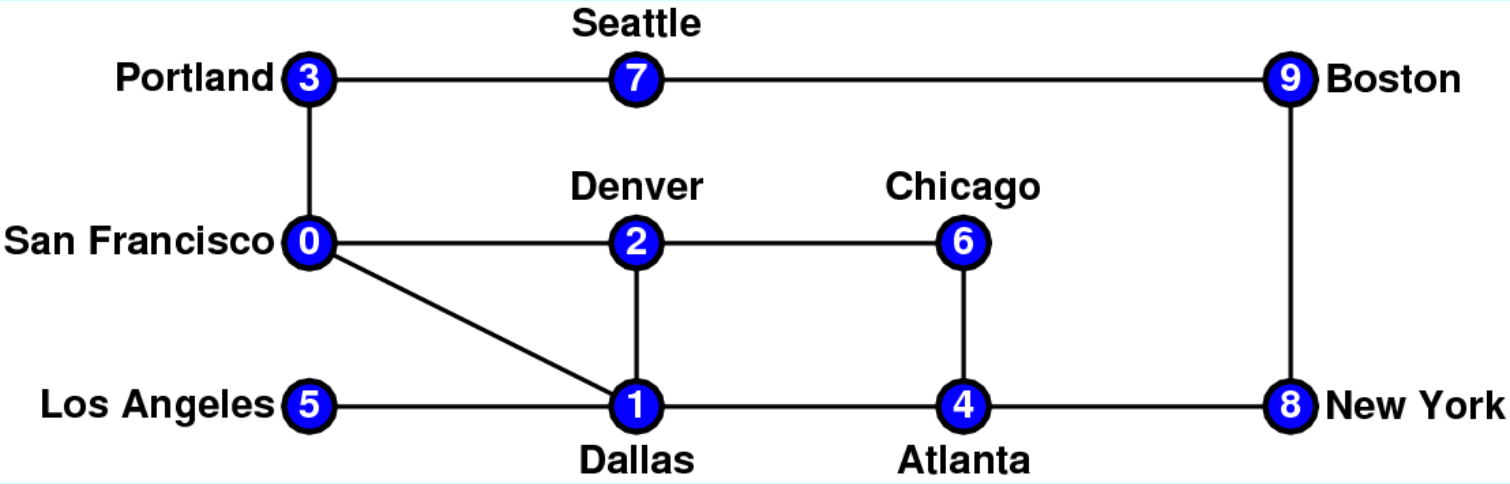
Breadth-First Search

- The traversal strategy of ***breadth-first search*** (or ***BFS*** for short) proceeds outward from the starting node, visiting the start node, then all nodes one hop away, and so on.
- For example, consider the graph:



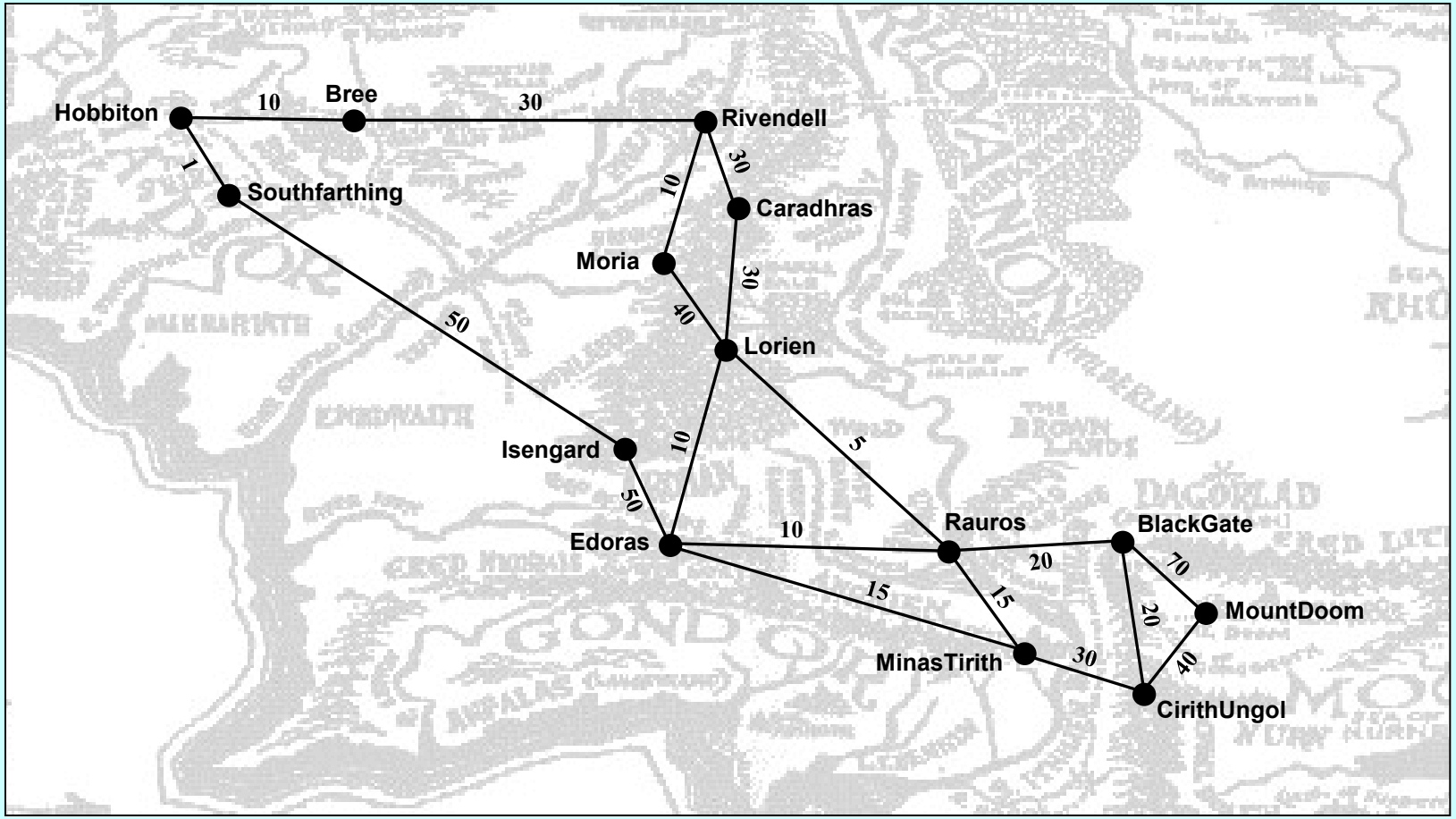
- Breadth-first search begins at the start node (`n1`), then does the one-hops (`n2` and `n6`), then the two hops (`n3`, `n5`, and `n7`) and finally the three hops (`n4`).

Exercise: Breadth-First Search



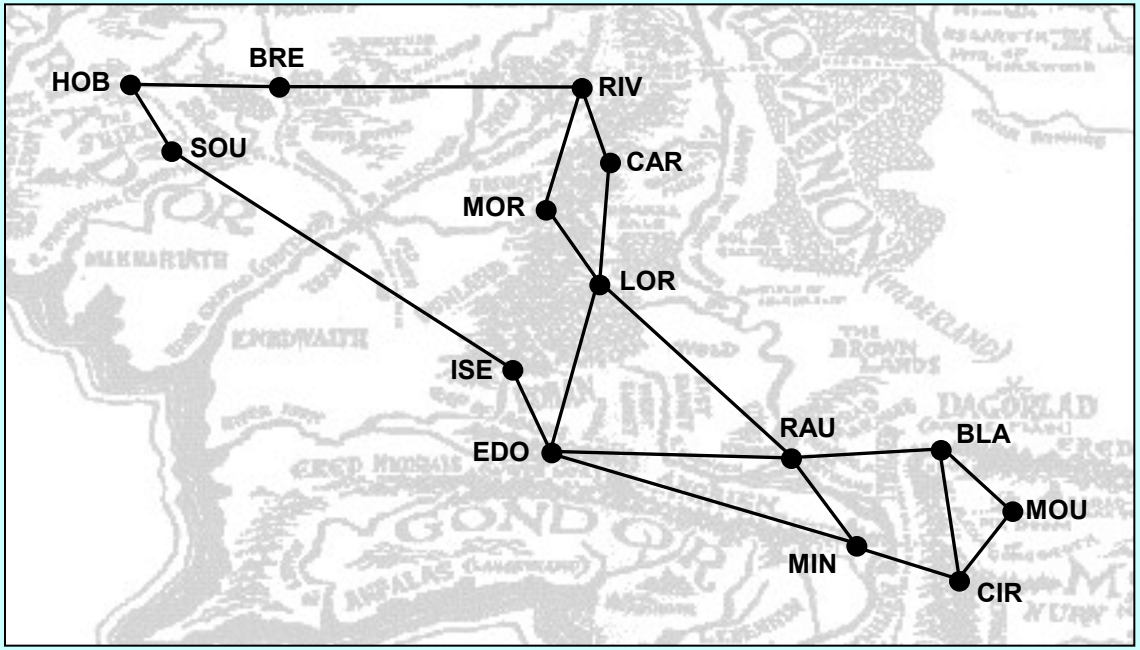
Example: Frodo's Journey

The Middle Earth Graph



Exercise: Depth-First Search

Construct a depth-first search starting from Hobbiton (**HOB**):



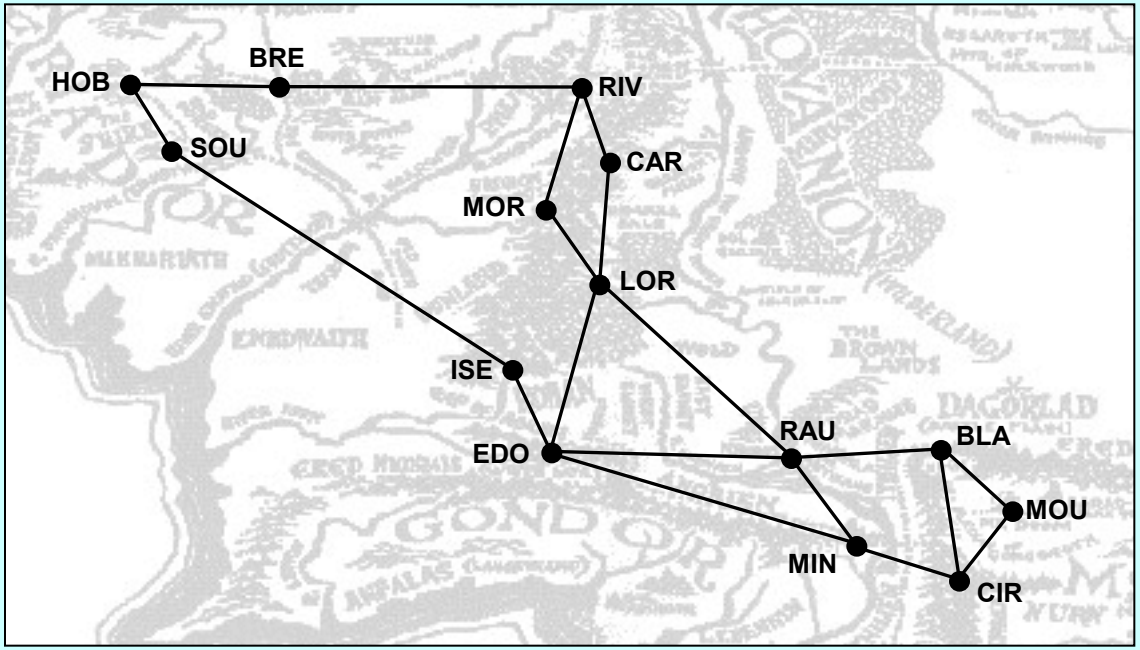
Visiting node **HOB**
Visiting node **BRE**
Visiting node **RIV**
Visiting node **CAR**
Visiting node **LOR**
Visiting node **EDO**
Visiting node **ISE**
Visiting node **SOU**
Visiting node **MIN**
Visiting node **CIR**
Visiting node **BLA**
Visiting node **MOU**
Visiting node **RAU**
Visiting node **MOR**

What data structure should we use?

Stack. Or even better, take advantage of the automatic stack frame by using recursion.

Exercise: Breadth-first search

Construct a breadth-first search starting from Isengard (**ISE**):



What data structure should we use?

Visiting node **ISE**
Visiting node **EDO**
Visiting node **SOU**
Visiting node **LOR**
Visiting node **MIN**
Visiting node **RAU**
Visiting node **HOB**
Visiting node **CAR**
Visiting node **MOR**
Visiting node **CIR**
Visiting node **BLA**
Visiting node **BRE**
Visiting node **RIV**
Visiting node **MOU**

Queue: ~~ISE~~ ~~EDO~~ ~~SOU~~ ~~LOR~~ ~~MIN~~ ~~RAU~~ ~~HOB~~ ~~CAR~~ ~~MOR~~ ~~CIR~~ ~~BLA~~ ~~BRE~~ ~~RIV~~ ~~MOU~~

Depth-First Search Using Recursion

```
void depthFirstSearch(Node *node) {
   Set<Node *> visited;
   visitUsingDFS(node, visited);
}

void visitUsingDFS(Node *node, Set<Node *> & visited) {
   if (visited.contains(node)) return;
   visit(node);
   visited.add(node);
   for (Arc *arc : node->arcs) {
      visitUsingDFS(arc->finish, visited);
   }
}
```

Exercise 1: Implement depth-first search using a stack explicitly.

Exercise 2: Implement breadth-first search using a queue.

Exercise 3 (*out of scope): The above program is pre-order traversal. Implement post-order traversal. How about in-order?

Designing a Graph Interface

- To take maximum advantage of the intuition that people have with graphs, it makes sense to design a graph package in C++ so that it adheres as closely as possible to the way in which graphs are defined in mathematics. In particular, the idea that a graph is a set of nodes together with a set of arcs should be clear from the definition.
- However, graphs are different from the more familiar collection classes we have studied, which contain values of some client-defined type that plays no essential role in the implementation of the collection class itself, essentially becoming the template parameter for the class definition.

Designing a Graph Interface

- For graphs, the elements are nodes and arcs. Both nodes and arcs have some properties that are closely tied to the graph in which they belong. Each node, for example, must keep track of the arcs that lead to its neighbors. In much the same way, each arc needs to know what nodes it connects.
- At the same time, both nodes and arcs are likely to contain other information that depends on the application, such as the name of a node or the cost of traversing an arc.
- Thus, nodes and arcs are hybrid structures that contain data required by the client along with data required by the implementation of the graph. They are an integral part of the graph and contain information required to maintain the overall data structure.

Three Strategies for Graph Abstraction

1. ***Use low-level structures***. This design uses the structure types `Node` and `Arc` to represent the components of a graph. This model gives clients complete freedom to extend these structures but offers no support for graph operations.
2. ***Adopt a hybrid strategy***. This design defines a `Graph` class as a parameterized class using templates so that it can use any structures or objects as the node and arc types. This strategy retains the flexibility of the low-level model and avoids the complexity associated with the class-based approach.
3. ***Define classes for each of the component types***. This design uses the `Node` and `Arc` classes to define the structure. In this model, clients define subclasses of the supplied types to particularize the graph data structure to their own application. More will be discussed later when we learn inheritance.

`Graph` as a parameterized class

- The basic idea behind this design is to export a parameterized `Graph` class that lets the client choose types for the `nodes` and `arcs`. Those types must adhere to a few basic rules that allow them to function correctly in the context of a graph. These required fields are essentially the ones that appear in the low-level structures.
- The type the client chooses to represent a node must contain
  - A string field called name that specifies the `name` of the node
  - A field called `arcs` that specifies the set of arcs that begin at this node
- The type chosen to represent an arc must contain
  - Fields called `start` and `finish` that indicate the endpoints of the arc (you might also need a `cost` for some algorithms)
- Beyond the required fields, the types used to represent nodes and arcs may contain additional information required for the client's application.

The Private Section of `graph.h`

```
template <typename NodeType,typename ArcType>
class Graph {

/* Private section */

private:

/*
 * Implementation notes: Data structure
 * ------------------------------------
 * The Graph class is defined -- as it traditionally is in
 * mathematics -- as a set of nodes and a set of arcs.  These
 * structures, in turn, are implemented using the Set class.
 * The element type for each set is a pointer to a structure
 * chosen by the client, which is specified as one of the
 * parameters to the class template.
 */

/* Instance variables */

   Set<NodeType *> nodes;                 /* The set of nodes in the graph */
   Set<ArcType *> arcs;                   /* The set of arcs in the graph  */
   Map<std::string,NodeType *> nodeMap; /* A map from names to nodes     */

/* Private methods */

   void deepCopy(const Graph & src);
   NodeType *getExistingNode(std::string name) const;
```

The class-based `graph.h` Interface

```
/*
 * File: graph.h
 * -------------
 * This interface exports a parameterized Graph class used to represent
 * graphs, which consist of a set of nodes and a set of arcs.
 */

#ifndef _graph_h
#define _graph_h

#include <string>
#include "error.h"
#include "map.h"
#include "set.h"

/*
 * Functions: nodeCompare, arcCompare
 * ----------------------------------
 * Standard comparison functions for nodes and arcs.
 */

template <typename NodeType>
int nodeCompare(NodeType *n1, NodeType *n2);

template <typename NodeType,typename ArcType>
int arcCompare(ArcType *a1, ArcType *a2);
```

The class-based `graph.h` Interface

```
/*
 * Class: Graph<NodeType,ArcType>
 * ------------------------------
 * This class represents a graph with the specified node and arc types.
 * The NodeType and ArcType parameters indicate the structure type or class
 * used for nodes and arcs, respectively.  These types can contain any
 * fields or methods required by the client, but must contain the following
 * public fields required by the Graph package itself:
 *
 * The NodeType definition must include:
 *   - A string field called name
 *   - A Set<ArcType *> field called arcs
 *
 * The ArcType definition must include:
 *   - A NodeType * field called start
 *   - A NodeType * field called finish
 */

template <typename NodeType,typename ArcType>
class Graph {

public:
```

The class-based `graph.h` Interface

```
/*
 * Constructor: Graph
 * Usage: Graph<NodeType,ArcType> g;
 * ---------------------------------
 * Creates an empty Graph object.
 */

   Graph();

/*
 * Destructor: ~Graph
 * Usage: (usually implicit)
 * -------------------------
 * Frees the internal storage allocated to represent the graph.
 */

   ~Graph();
```

The class-based `graph.h` Interface

```
/*
 * Method: size
 * Usage: int size = g.size();
 * ---------------------------
 * Returns the number of nodes in the graph.
 */

   int size();

/*
 * Method: isEmpty
 * Usage: if (g.isEmpty()) . . .
 * -----------------------------
 * Returns true if the graph is empty.
 */

   bool isEmpty();

/*
 * Method: clear
 * Usage: g.clear();
 * -----------------
 * Reinitializes the graph to be empty, freeing any heap storage.
 */

   void clear();
```

The class-based `graph.h` Interface

```
/*
 * Method: addNode
 * Usage: NodeType *node = g.addNode(name);
 *        NodeType *node = g.addNode(node);
 * ---------------------------------------
 * Adds a node to the graph.  The first version of this method creates a
 * new node of the appropriate type and initializes its fields; the second
 * assumes that the client has already created the node and simply adds it
 * to the graph.  Both versions of this method return a pointer to the node.
 */

   NodeType *addNode(std::string name);
   NodeType *addNode(NodeType *node);

/*
 * Method: getNode
 * Usage: NodeType *node = g.getNode(name);
 * ---------------------------------------
 * Looks up a node in the name table attached to the graph and returns a
 * pointer to that node.  If no node with the specified name exists,
 * getNode signals an error.
 */

   NodeType *getNode(std::string name);
```

The class-based `graph.h` Interface

```
/*
 * Method: addArc
 * Usage: g.addArc(s1, s2);
 *        g.addArc(n1, n2);
 *        g.addArc(arc);
 * ----------------------
 * Adds an arc to the graph.  The endpoints of the arc can be specified
 * either as strings indicating the names of the nodes or as pointers to
 * the node structures.  Alternatively, the client can create the arc
 * structure explicitly and pass that pointer to the addArc method.  All
 * three of these versions return a pointer to the arc in case the client
 * needs to capture this value.
 */

   ArcType *addArc(std::string s1, std::string s2);
   ArcType *addArc(NodeType *n1, NodeType *n2);
   ArcType *addArc(ArcType *arc);

/*
 * And all the remove methods…
 */
   void removeNode(std::string name);
   void removeNode(NodeType *node);

   void removeArc(std::string s1, std::string s2);
   void removeArc(NodeType *n1, NodeType *n2);
   void removeArc(ArcType *arc);
```

The class-based `graph.h` Interface

```
/*
 * Method: isConnected
 * Usage: if (g.isConnected(n1, n2)) . . .
 *        if (g.isConnected(s1, s2)) . . .
 * -----------------------------------
 * Returns true if the graph contains an arc from n1 to n2.  As in the
 * addArc method, nodes can be specified either as node pointers or by
 * name.
 */

   bool isConnected(NodeType *n1, NodeType *n2);
   bool isConnected(std::string s1, std::string s2);

/*
 * Method: getNeighbors
 * Usage: foreach (NodeType *node in g.getNeighbors(node)) . . .
 *        foreach (NodeType *node in g.getNeighbors(name)) . . .
 * ---------------------------------------------------------
 * Returns the set of nodes that are neighbors of the specified node, which
 * can be indicated either as a pointer or by name.
 */

   Set<NodeType *> getNeighbors(NodeType *node);
   Set<NodeType *> getNeighbors(std::string node);
```

The class-based `graph.h` Interface

```
/*
 * Method: getNodeSet
 * Usage: foreach (NodeType *node in g.getNodeSet()) . . .
 * ------------------------------------------------------
 * Returns the set of all nodes in the graph.
 */

   Set<NodeType *> & getNodeSet();

/*
 * Method: getArcSet
 * Usage: foreach (ArcType *arc in g.getArcSet()) . . .
 *        foreach (ArcType *arc in g.getArcSet(node)) . . .
 *        foreach (ArcType *arc in g.getArcSet(name)) . . .
 * ------------------------------------------------------
 * Returns the set of all arcs in the graph or, in the second and third
 * forms, the arcs that start at the specified node, which can be indicated
 * either as a pointer or by name.
 */

   Set<ArcType *> & getArcSet();
   Set<ArcType *> & getArcSet(NodeType *node);
   Set<ArcType *> & getArcSet(std::string name);
```

The private and implementation sections go here.

The Implementation Section

```
/* Implementation notes: Graph class
 * -----------------------------
 * As is typical for layered abstractions built on top of other classes,
 * the implementations of the methods in the graph class tend to be very
 * short, because they can hand all the hard work off to the underlying
 * class.
 *
 * Implementation notes: addNode
 * -----------------------------
 * The addNode method adds the node to the set of nodes for the graph and
 * to the map from names to nodes.
 */
template <typename NodeType,typename ArcType>
NodeType *Graph<NodeType,ArcType>::addNode(std::string name) {
   if (nodeMap.containsKey(name)) {
      error("addNode: Node " + name + " already exists");
   }
   NodeType *node = new NodeType();
   node->name = name;
   return addNode(node);
}
template <typename NodeType,typename ArcType>
NodeType *Graph<NodeType,ArcType>::addNode(NodeType *node) {
   nodes.add(node);
   nodeMap[node->name] = node;
   return node;
}
```

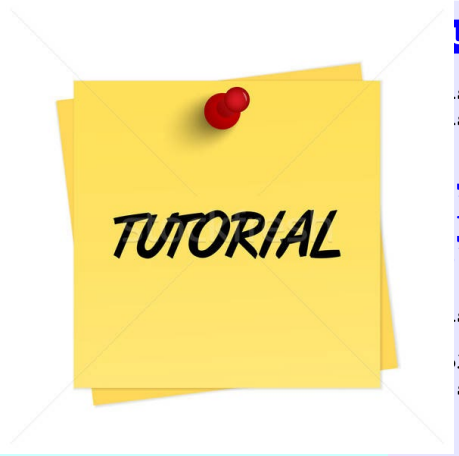
The Implementation Section

```
/*
 * Implementation notes: addArc
 * ----------------------------
 * The addArc method appears in three forms, as described in the interface.
 */

template <typename NodeType,typename ArcType>
ArcType *Graph<NodeType,ArcType>::addArc(std::string s1, std::string s2) {
   return addArc(getExistingNode(s1), getExistingNode(s2));
}

template <typename NodeType,typename ArcType>
ArcType *Graph<NodeType,ArcType>::addArc(NodeType *n1, NodeType *n2) {
   ArcType *arc = new ArcType();
   arc->start = n1;
   arc->finish = n2;
   return addArc(arc);
}

template <typename NodeType,typename ArcType>
ArcType *Graph<NodeType,ArcType>::addArc(ArcType *arc) {
   arc->start->arcs.add(arc);
   arcs.add(arc);
   return arc;
}
```



URE 18-7 Redefinition of the airline graph using classes

```
ss City;        /* Forward references to these two types so  */
ss Flight;      /* that the C++ compiler can recognize them. */


Type: City
----------
This class defines the node type for the airport graph.


ss City {

lic:
string getName() {
   return name;
}

private:
   string name;
   Set<Flight *> arcs;
   string airportCode;
   friend class Graph<City,Flight>;
};

/*
 * Type: Flight
 * ------------
 * This class defines the arc type for the airport graph.
 */

class Flight {

public:
   City *getStart() {
      return start;
   }

   City *getFinish() {
      return finish;
   }

   int getDistance() {
      return distance;
   }

   void setDistance(int miles) {
      distance = miles;
   }

private:
   City *start;
   City *finish;
   int distance;
   friend class Graph<City,Flight>;
};
```

SimpleGraph vs. Graph

```
/* graphtypes.h - header for
 * SimpleGraph structure
 */
struct Node {
   string name;
   Set<Arc *> arcs;
};
struct Arc {
   Node *start;
   Node *finish;
   double cost;
};
struct SimpleGraph {
   Set<Node *> nodes;
   Set<Arc *> arcs;
   Map<string,Node *> nodeMap;
};
```

```
/* AirlineGraph.cpp -
 * implementation of both
 * graph algorithms and
 * user applications
 */
```

```
/* graph.h - header for Graph class
 */
template <typename NodeType,typename ArcType>
class Graph {
private:
   Set<NodeType *> nodes;
   Set<ArcType *> arcs;
   Map<std::string,NodeType *> nodeMap;
/* The NodeType definition must include:
 *   - A string field called name
 *   - A Set<ArcType *> field called arcs
 *
 * The ArcType definition must include:
 *   - A NodeType * field called start
 *   - A NodeType * field called finish
 */
...
}
/* using name, arcs, start, finish to
 * implement the graph algorithms
 */
name...
arcs...
start...
finish...
```

SimpleGraph vs. Graph

```
/* graphtypes.h - header for
 * SimpleGraph structure
 */
struct Node {
   string name;
   Set<Arc *> arcs;
};
struct Arc {
   Node *start;
   Node *finish;
   double cost;
};
struct SimpleGraph {
   Set<Node *> nodes;
   Set<Arc *> arcs;
   Map<string,Node *> nodeMap;
};
```

```
/* AirlineGraph.cpp -
 * implementation of both
 * graph algorithms and
 * user applications
 */
```

```
/* graph.h - header for Graph class
 */
template <typename NodeType,typename ArcType>
class Graph {
private:
   Set<NodeType *> nodes;
   Set<ArcType *> arcs;
   Map<std::string,NodeType *> nodeMap; ... }
```

```
/* AirlineGraph.cpp - implementation of
 * user applications only
 */
class City;
class Flight;
class City {
private:
   string name;
   Set<Flight *> arcs;
   friend class Graph<City,Flight>;
};
class Flight {
private:
   City *start;
   City *finish;
   friend class Graph<City,Flight>;
};
```

Three Strategies for Graph Abstraction

1. ***Use low-level structures***. This design uses the structure types `Node` and `Arc` to represent the components of a graph. This model gives clients complete freedom to extend these structures but offers no support for graph operations.
2. ***Adopt a hybrid strategy***. This design defines a `Graph` class as a parameterized class using templates so that it can use any structures or objects as the node and arc types. This strategy retains the flexibility of the low-level model and avoids the complexity associated with the class-based approach.
3. ***Define classes for each of the component types***. This design uses the `Node` and `Arc` classes to define the structure. In this model, clients define subclasses of the supplied types to particularize the graph data structure to their own application. More will be discussed later when we learn inheritance.

Three Strategies for Graph Abstraction

- In most object-oriented languages, the usual strategy to use in situations of this kind is ***subclassing***.
- Under this model, the graph package itself would define base classes for nodes and arcs that contain the information necessary to represent the graph structure. Clients needing to include additional data would do so by extending these base classes to create new subclasses with additional fields and methods.
- For reasons that you will have a chance to explore more fully in the extended discussion of inheritance in Chapter 19, this approach is not ideal for C++.
- The fundamental problem is that dynamic memory allocation and inheritance don't always work together in C++ as seamlessly as they do in other languages, to the point that it is worth adopting a different approach.

The End