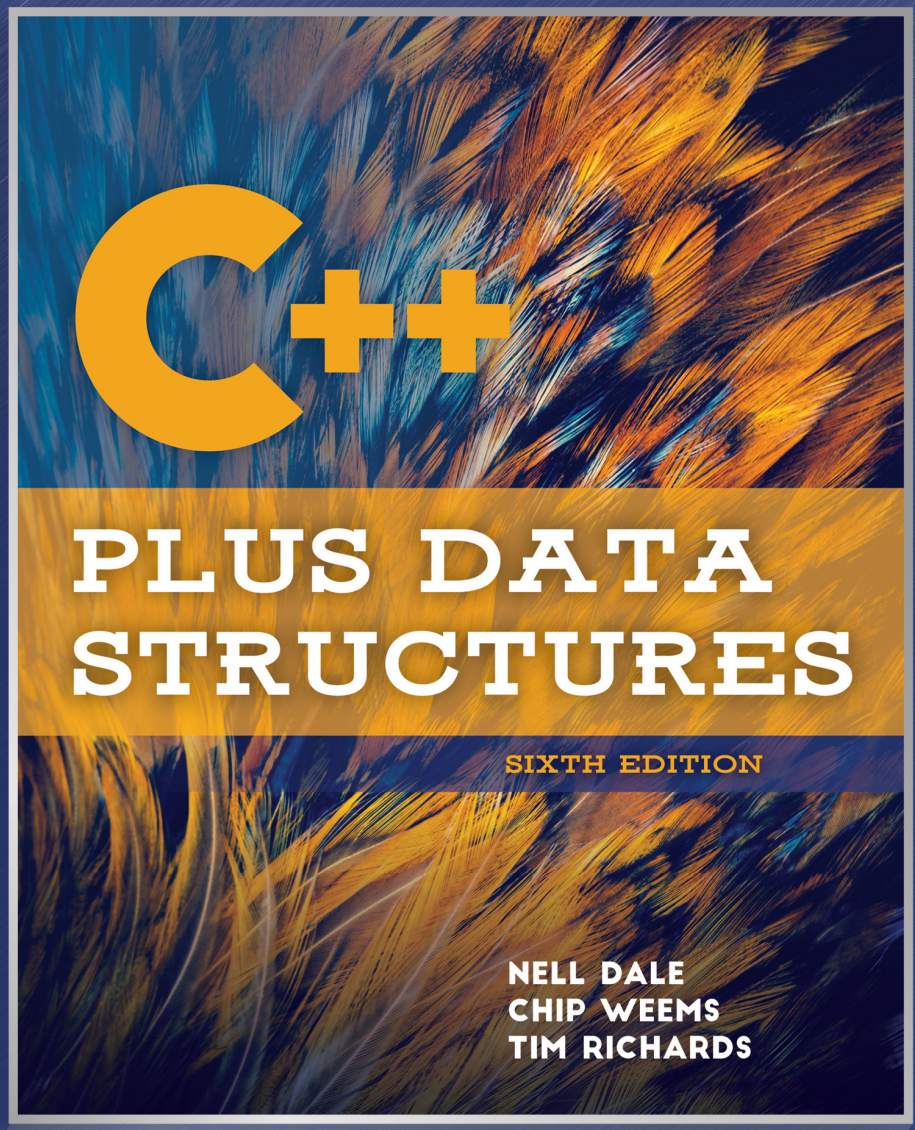
Chapter 13
Graphs



Graphs: Logical Level

- A **graph** is a set of **vertices** (nodes) and a set of **edges** that relate the vertices to each other
- Each vertex is a “thing” or data point, such as a person or a city
- Edges are represented as pairs of vertices
- Graphs are somewhat similar to binary trees, but have no restrictions on the connections between nodes

Edge Direction

- **Directed Graph:** Each edge is directed from one vertex to the other vertex
  - For example, on a graph of cities, a directed edge from NYC to Albany could represent a flight
- **Undirected Graph:** Edges have no direction
  - For example, on a graph of cities, an undirected edge from NYC to Albany could represent a road

Formal Definition

- Graphs are based on the mathematical concept
- A graph *G* is defined as *G = (V,E)*
  - *V* is a finite, nonempty set of vertices
  - *E* is a set of edges (written as pairs of vertices)

Graph Examples

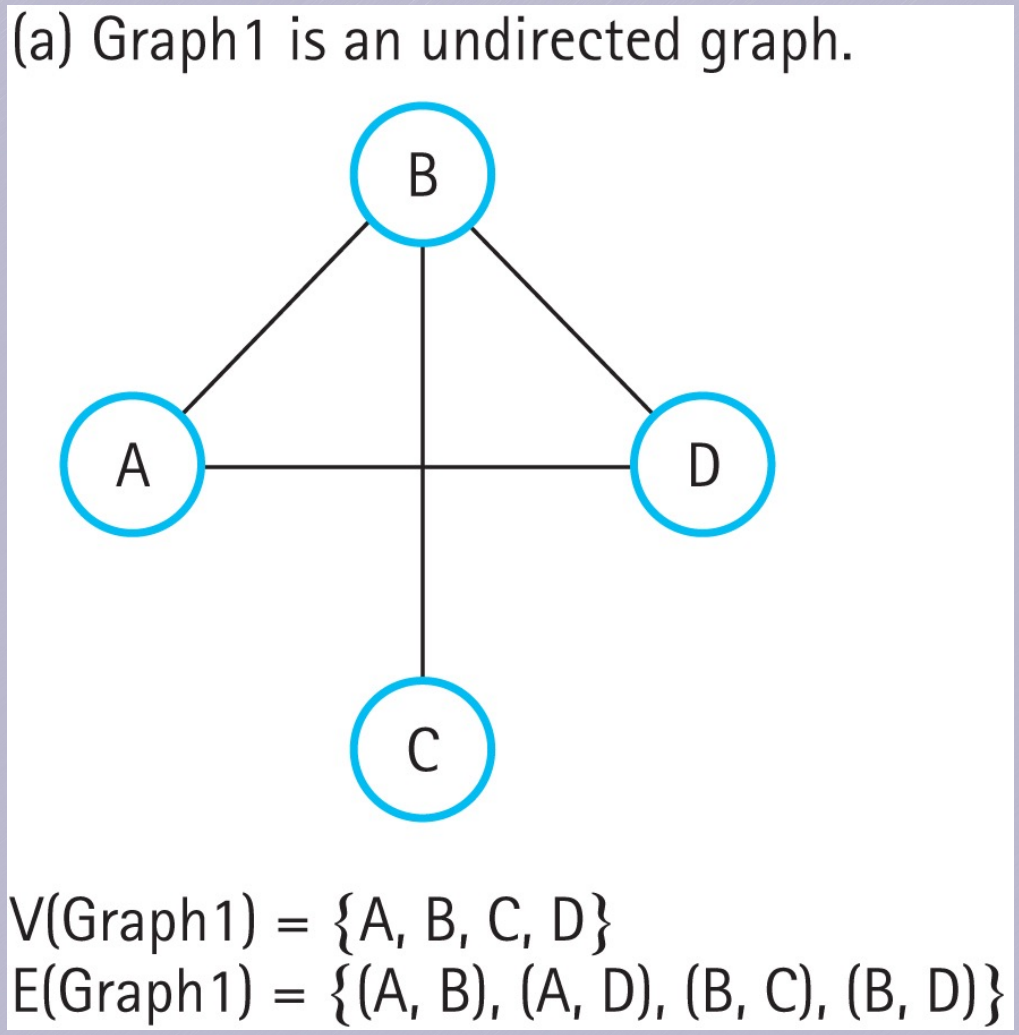


Figure 13.1a Some examples of graphs (a) Graph1 is an undirected graph (b) Graph2 is a directed graph (c) Graph3 is a directed graph

Graph Examples (cont.)

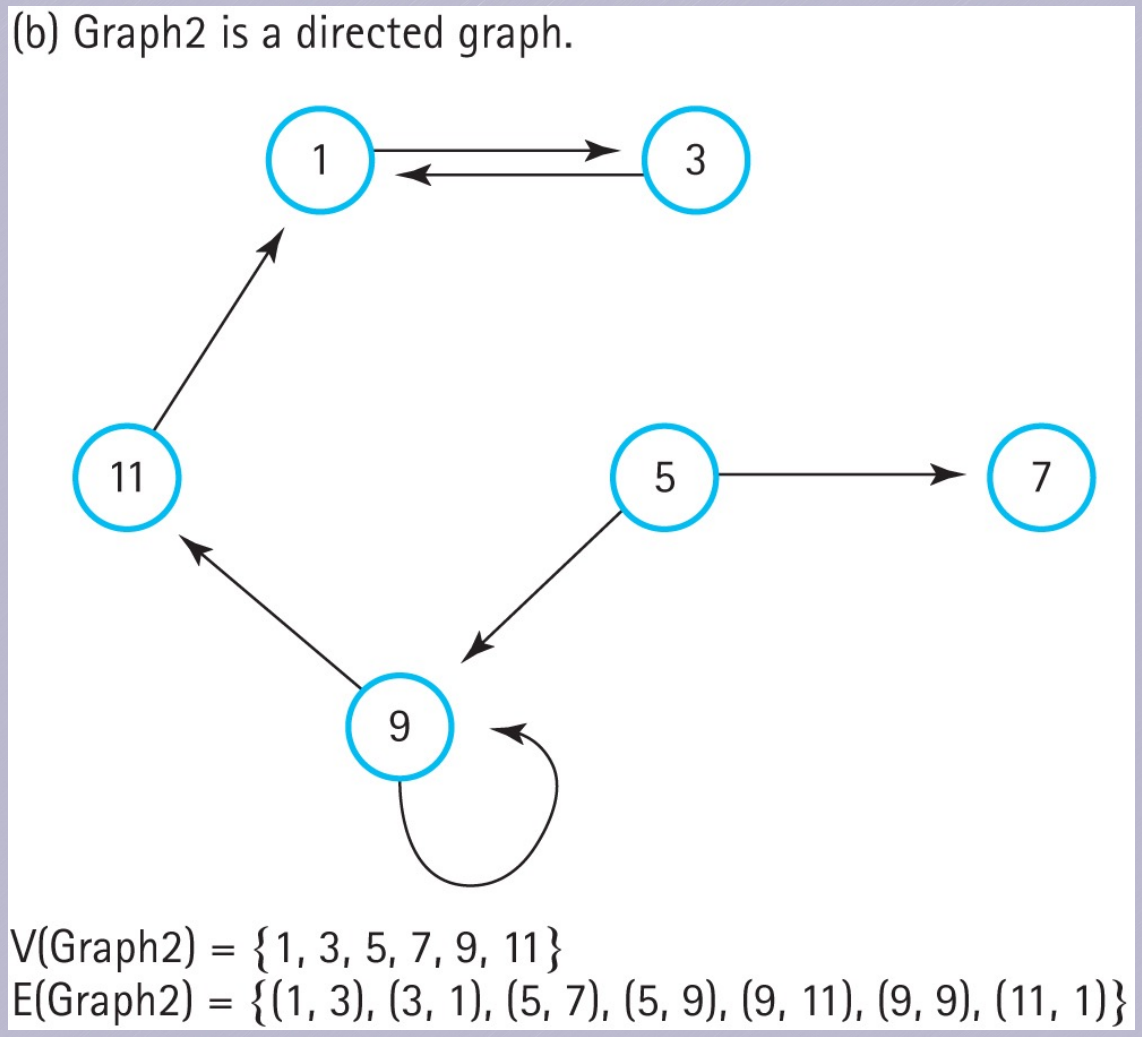
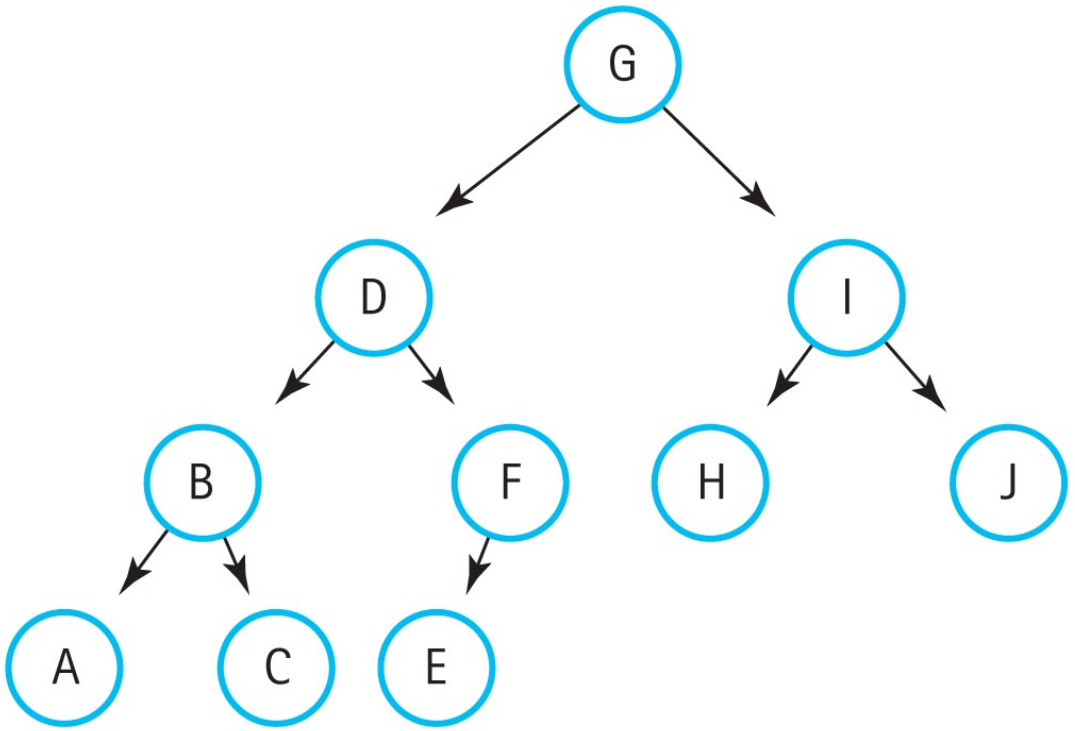


Figure 13.1b Some examples of graphs (a) Graph1 is an undirected graph (b) Graph2 is a directed graph (c) Graph3 is a directed graph

Graph Examples (cont.)

(c) Graph3 is a directed graph.



V(Graph3) = {A, B, C, D, E, F, G, H, I, J}
E(Graph3) = {(G, D), (G, I), (D, B), (D, F), (I, H), (I, J), (B, A), (B, C), (F, E)}

This graph is a tree, which is a special case of a directed graph

Figure 13.1c Some examples of graphs (a) Graph1 is an undirected graph (b) Graph2 is a directed graph (c) Graph3 is a directed graph

Adjacency

- Two vertices are said to be adjacent if they are connected by an edge
- If they're connected by a directed edge:
  - The first vertex is **adjacent to** the second vertex
  - The second vertex is **adjacent from** the first vertex
- A **path** is a sequence of vertices that connects two vertices in the graph

Completeness

- In a **complete graph**, every node is directly connected to every other node
- A complete directed graph with *N* nodes has
  - *N**(*N*-1) edges
  - A complete undirected graph with *N* nodes has *N**(*N*-1)/2 edges

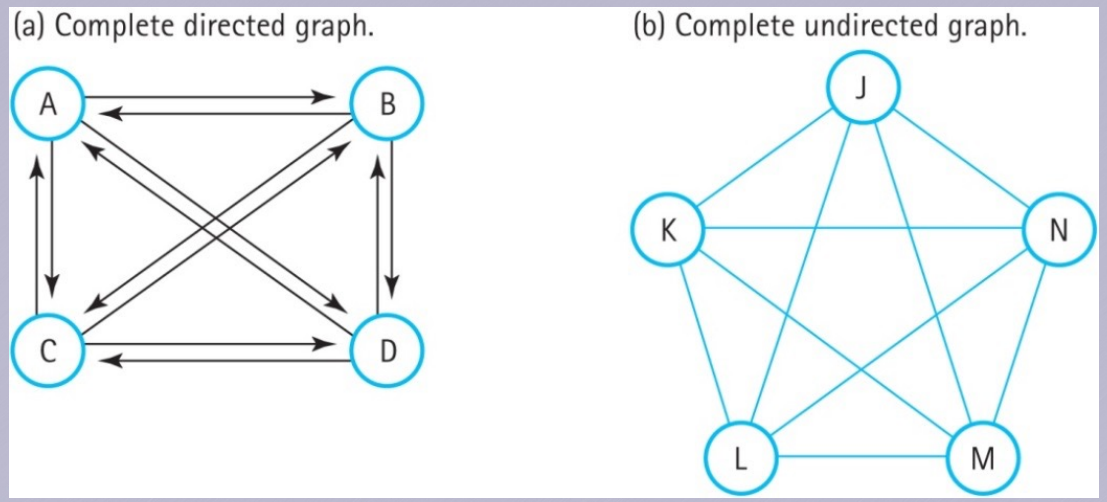


Figure 13.2 Two complete graphs (a) Complete directed graph (b) Complete undirected graph

Weighted Graph

- In a **weighted graph**, each edge has an associated value
- A weight typically represents the *cost* of moving between the two vertices
- For instance, the weights on a graph of cities would be the distance between the cities
- The total distance between two cities is the sum of weights on the path between them

Weighted City Graph

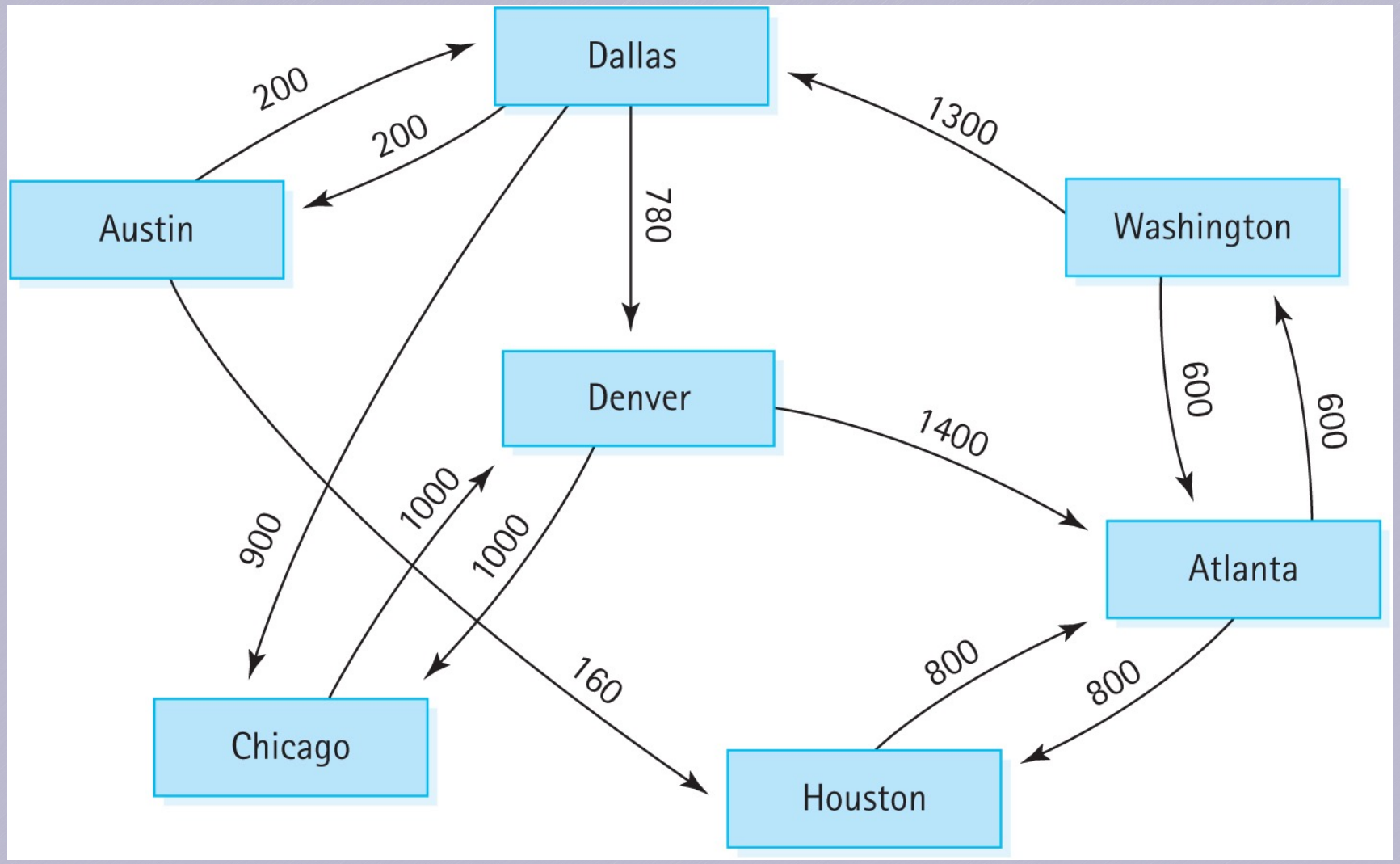


Figure 13.3 A weighted graph

Graph Operations

- Transformers: AddVertex, AddEdge, and MakeEmpty
- Observers: IsFull, IsEmpty, GetWeight, GetToVertices (returns a queue of adjacent vertices)
- Many other graph operations are possible, but these ones are all we need for now

Graphs: Application Level

- Graph traversal is a commonly used operation
- Like trees, there are multiple ways to do it
- Because traversing the graph is independent of the graph itself, the graph traversal methods are separate from the graph operations

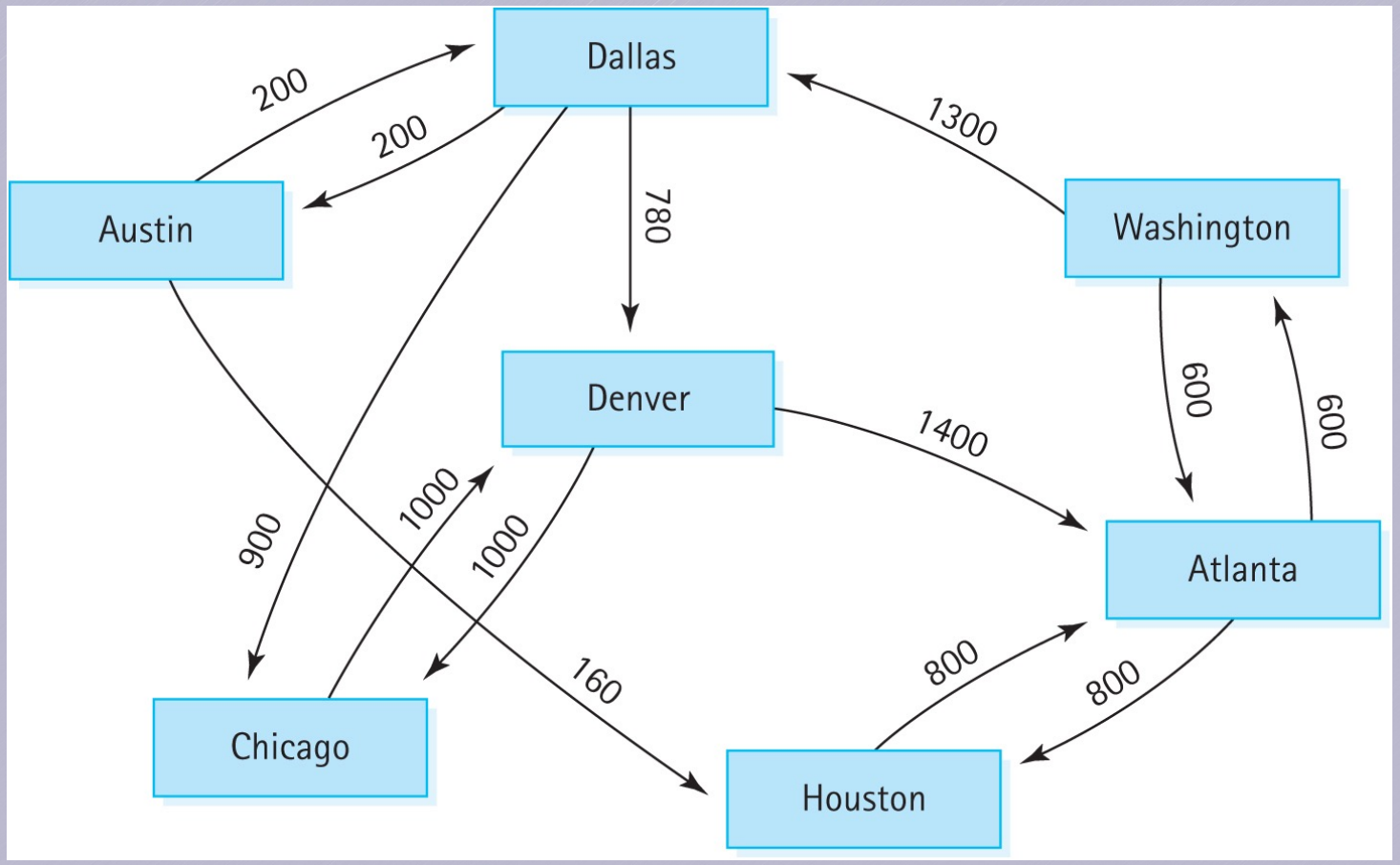
Depth-First Search

- **Depth-first search** follows a path as far as possible before backtracking
  - Similar to postorder traversal of binary trees
- It is useful in graphs for checking if a path exists between two nodes

DFS Algorithm

- Add the nodes adjacent to the start node to the stack
- Pop a node off the stack and examine it
- Add all of the nodes adjacent to the popped node to the stack
- Continue until the target node is found or the stack is empty (no more nodes to check)

DFS Example



We'll use the airport map for an example

Figure 13.3 A weighted graph

DFS Example (cont.)

- Flying from Austin to Washington
  - Push Austin onto the stack
  - Pop Austin, push Dallas and Houston onto the stack
  - Pop Houston, push Atlanta
  - Pop Atlanta, push Washington and Houston

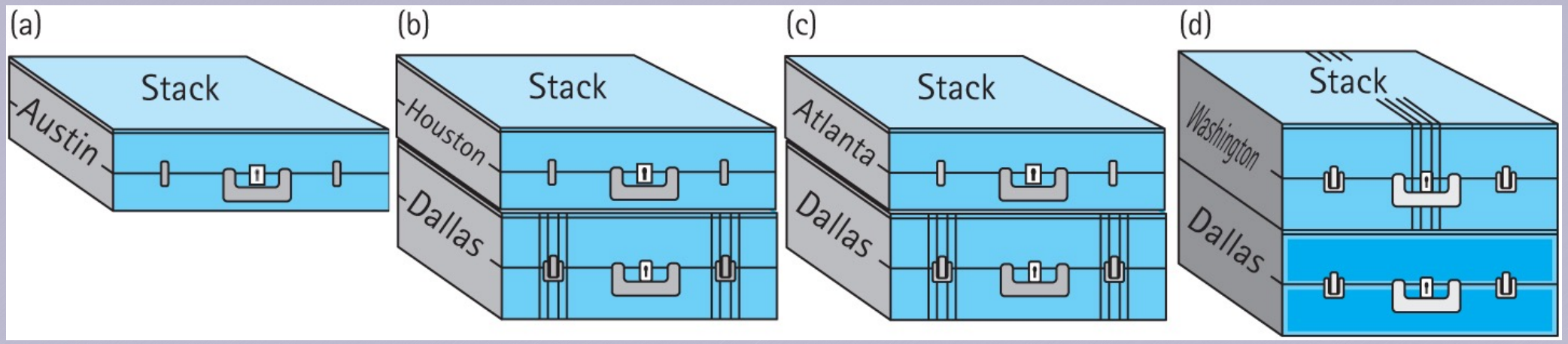


Figure 13.4 Using a stack to store the routes

Marking

- Atlanta leads to both Houston and Washington, but we already visited Houston
- By marking nodes, DFS can avoid revisiting nodes, which would lead to cycles
- Three new operations: MarkVertex transformer, IsMarked observer, and ClearMarks (clears marks from all nodes)
- Marked nodes are not added to the stack

DFS Example

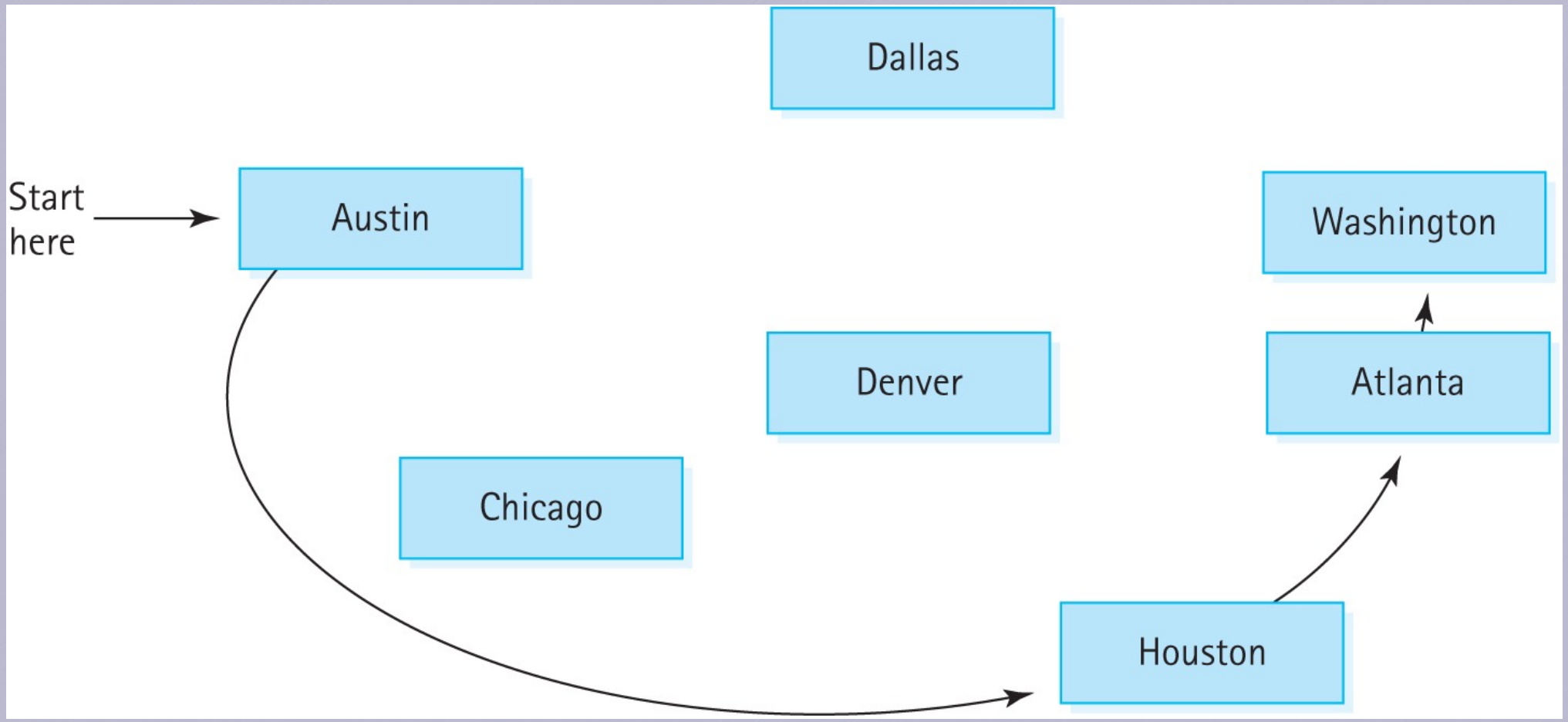


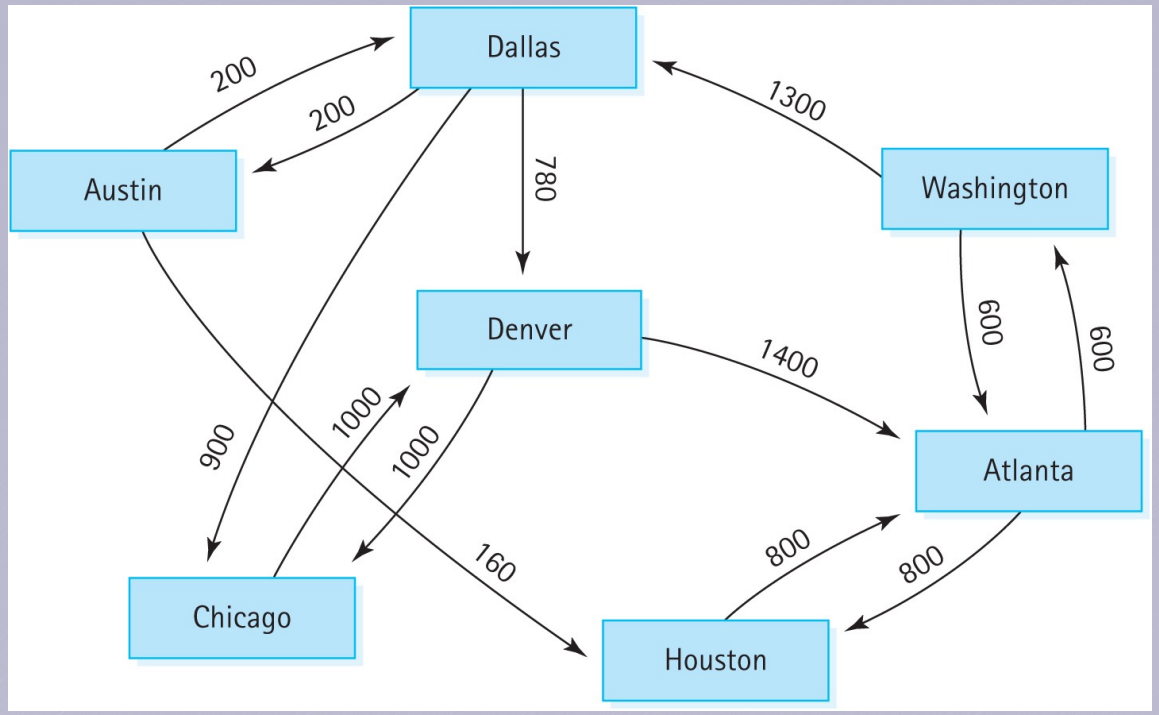
Figure 13.5 The depth-first search

Breadth-First Search

- **Breadth-first search** examines all possible paths of the same length before going further
- DFS backtracks as little as possible, while BFS backtracks as far as possible
- In a binary tree, BFS would explore all the nodes at a particular level before exploring any nodes on the next level
- BFS uses a queue to keep track of nodes

BFS Example

- Enqueue Dallas and Houston
- Dequeue Dallas, enqueue Chicago and Denver
- Dequeue Houston, enqueue Atlanta
- Continue until Washington is dequeued
- BFS explored all of the one-flight paths before checking the two-flight paths



BFS Example (cont.)

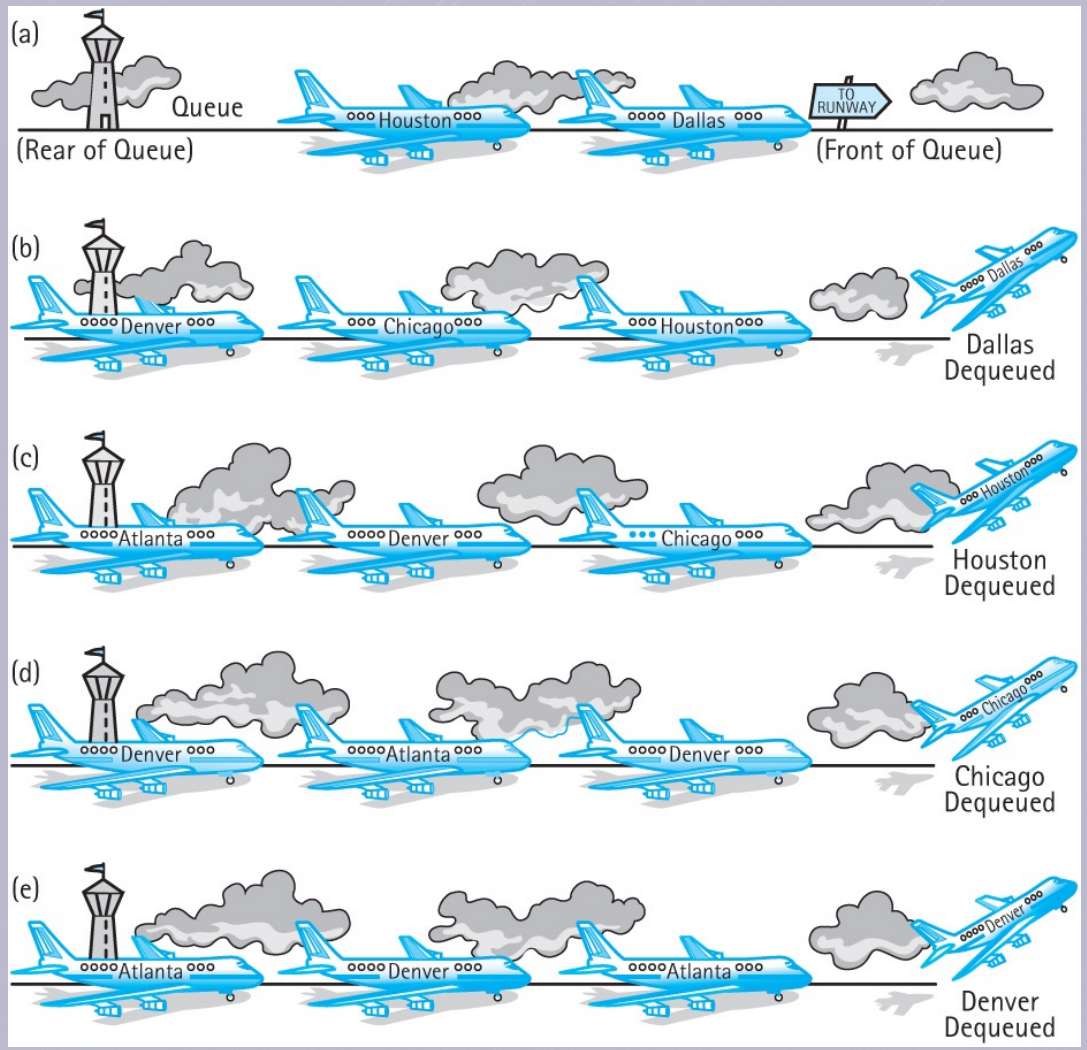


Figure 13.6 Using a queue to store the routes

BFS Example (cont.)

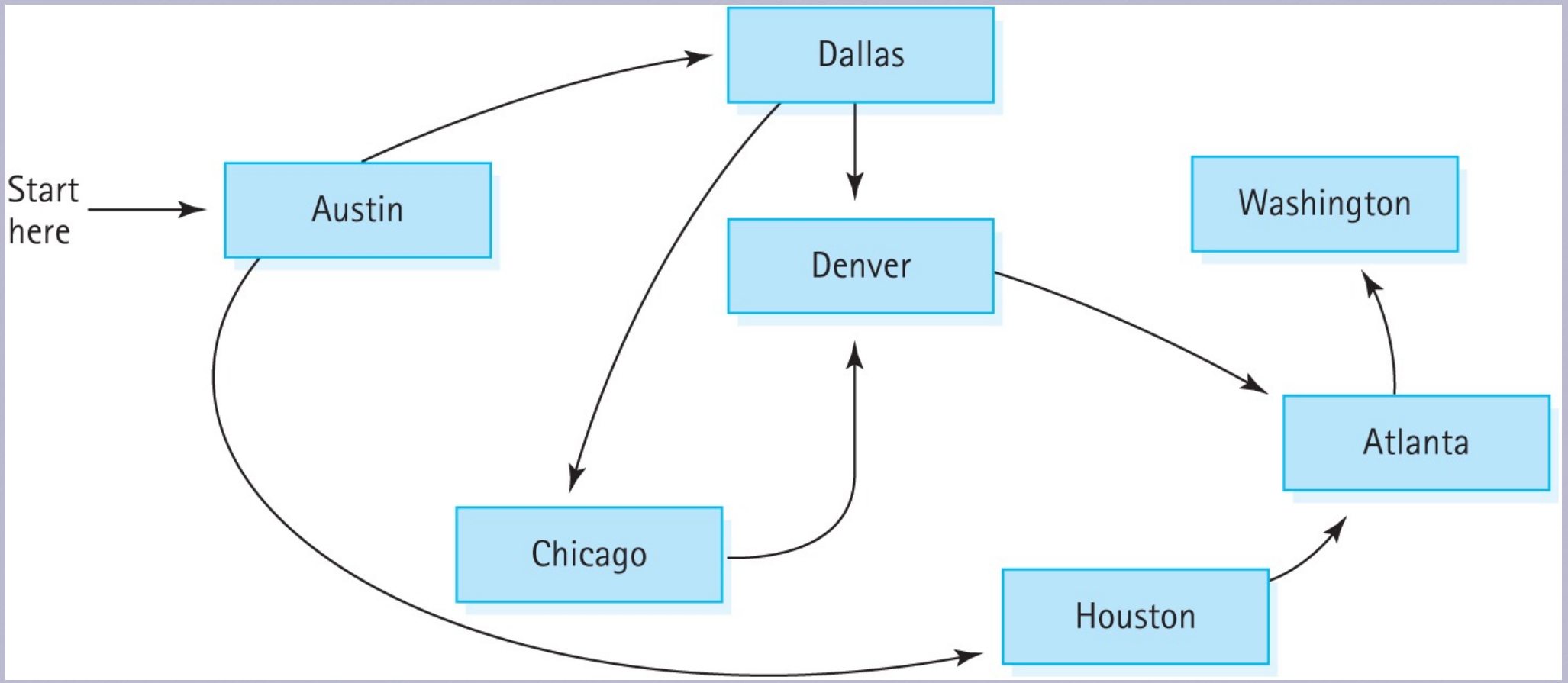


Figure 13.7 The breadth-first search

Marking and BFS

- A node adjacent to multiple nodes may be added to the queue twice
  - Denver is enqueued by Dallas and Chicago
- If nodes are marked when they're dequeued, a node may be enqueued multiple times
- Marking a node when it is enqueued prevents multiple copies in the queue
- Either approach can be useful

The Single-Source Shortest-Path Problem

- Find the shortest path from the starting city to every other city on the map
- Fewer flights does not mean shorter; instead, use the edge weights to find the shortest path
- That is, from the current node, find the node with the shortest distance

Shortest Path Algorithm

- Basically the same as DFS and BFS
- Stops when there are no unmarked cities
- Which auxiliary structure to use?
  - DFS uses stack, BFS uses queue
  - But shortest path needs the node with the next shortest distance
  - A minimum priority queue handles this, with the distance used as the priority

Graphs: Implementation Level

- Two main approaches: array-based and linked list-based
- The largest design question is how to handle edges
- Each approach handles edges differently

Array-Based Implementation

- Vertices are stored in an array
- Edges are represented in an **adjacency matrix**, an *N*x*N* table that shows the existence and weight of each edge in the graph
- Row[0] of the adjacency matrix contains the edges that start at Vertex[0]
- The matrix could store Boolean values, or numbers for weighted graphs

Array-Based Implementation (cont.)

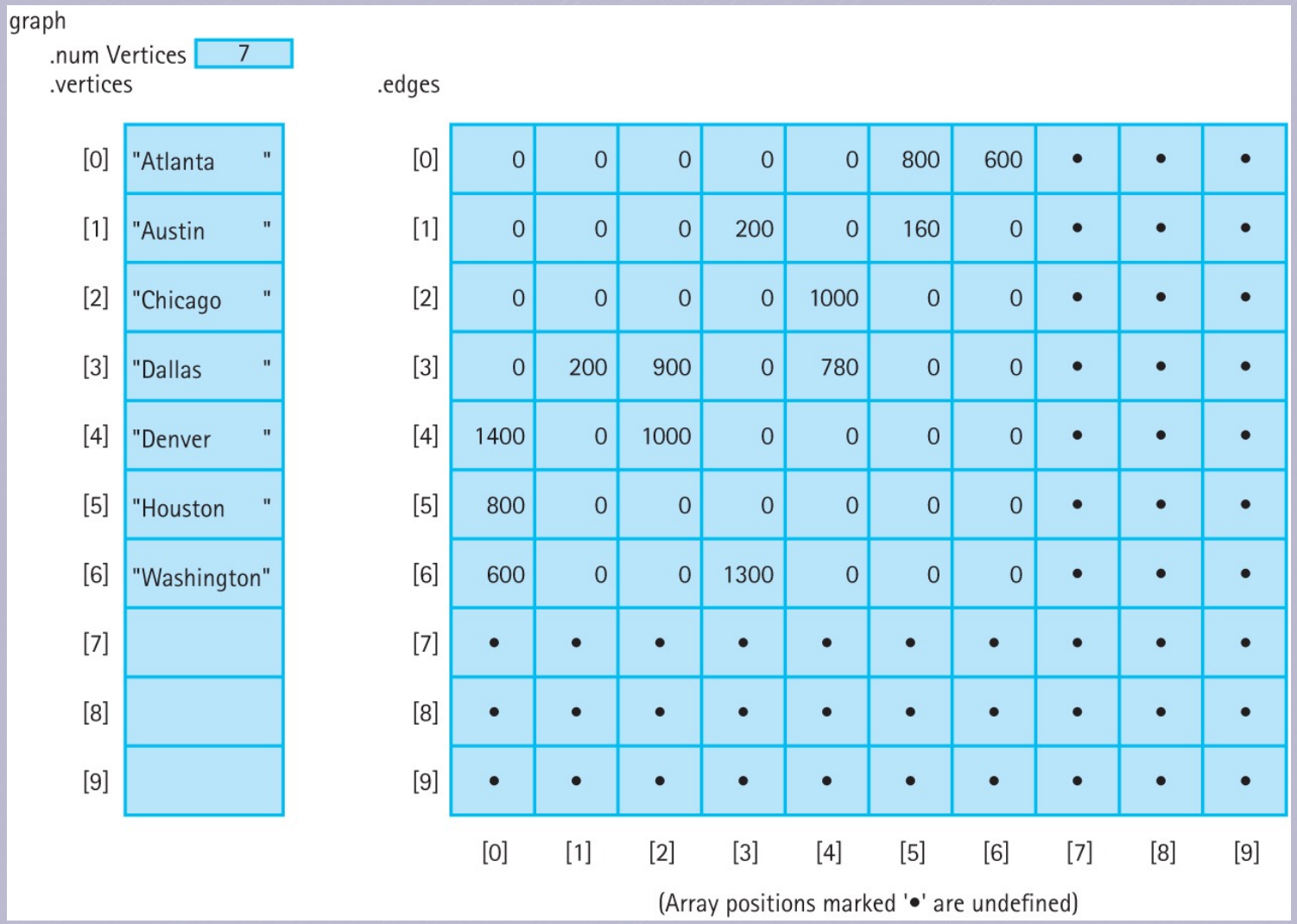


Figure 13.8 Graph of flight connections between cities

Class Constructors

- Dynamically allocating two-dimensional arrays can be somewhat complex
- The textbook's implementation fixes the adjacency matrix at 50x50 entries
- The vertices and marks arrays are dynamically allocated

Operations

- AddVertex: Inserts the vertex at the end of the array and sets all of its adjacency entries to 0
- AddEdge: Takes two vertices, but needs the indices of the vertices in the vertex array; simply searching the vertex array is sufficient
- GetToVertices: Enqueues each vertex that the source node has an edge to
- GetWeight: Looks up the weight in the matrix

Linked-List Implementation

- **Adjacency List:** A linked list that identifies all vertices to which a vertex is connected
- Each vertex has its own adjacency list
- Two implementations, using array indices or pointers to vertices in the adjacency list

Array of Vertices

- The adjacency list is an array of vertices
- Each entry contains the vertex data and a pointer to that vertex's adjacency list
- Each entry in the adjacency list contains the *index* of the adjacent vertex, the weight of the edge, and a pointer to the next node in the list

Array of Vertices (cont.)

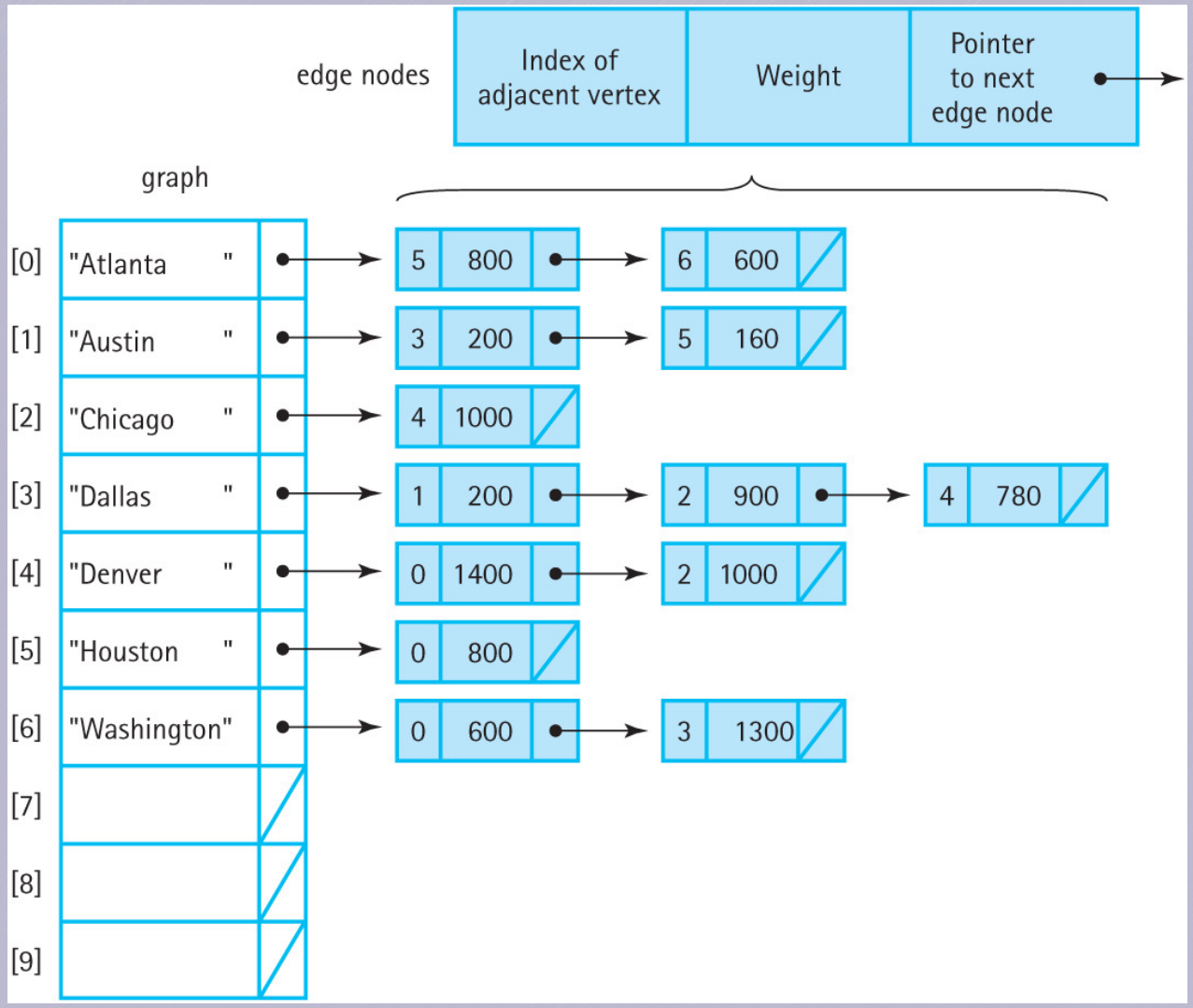


Figure 13.9 Adjacency list representation of graphs

Linked List of Vertices

- The vertices are stored in a linked list
- Each entry points to the next node in the vertex list and to vertex's adjacency list
- Each entry in the adjacency list contains a pointer to a node in the vertex list, the weight of the edge, and a pointer to the next node in the adjacency list

Linked List of Vertices (cont.)

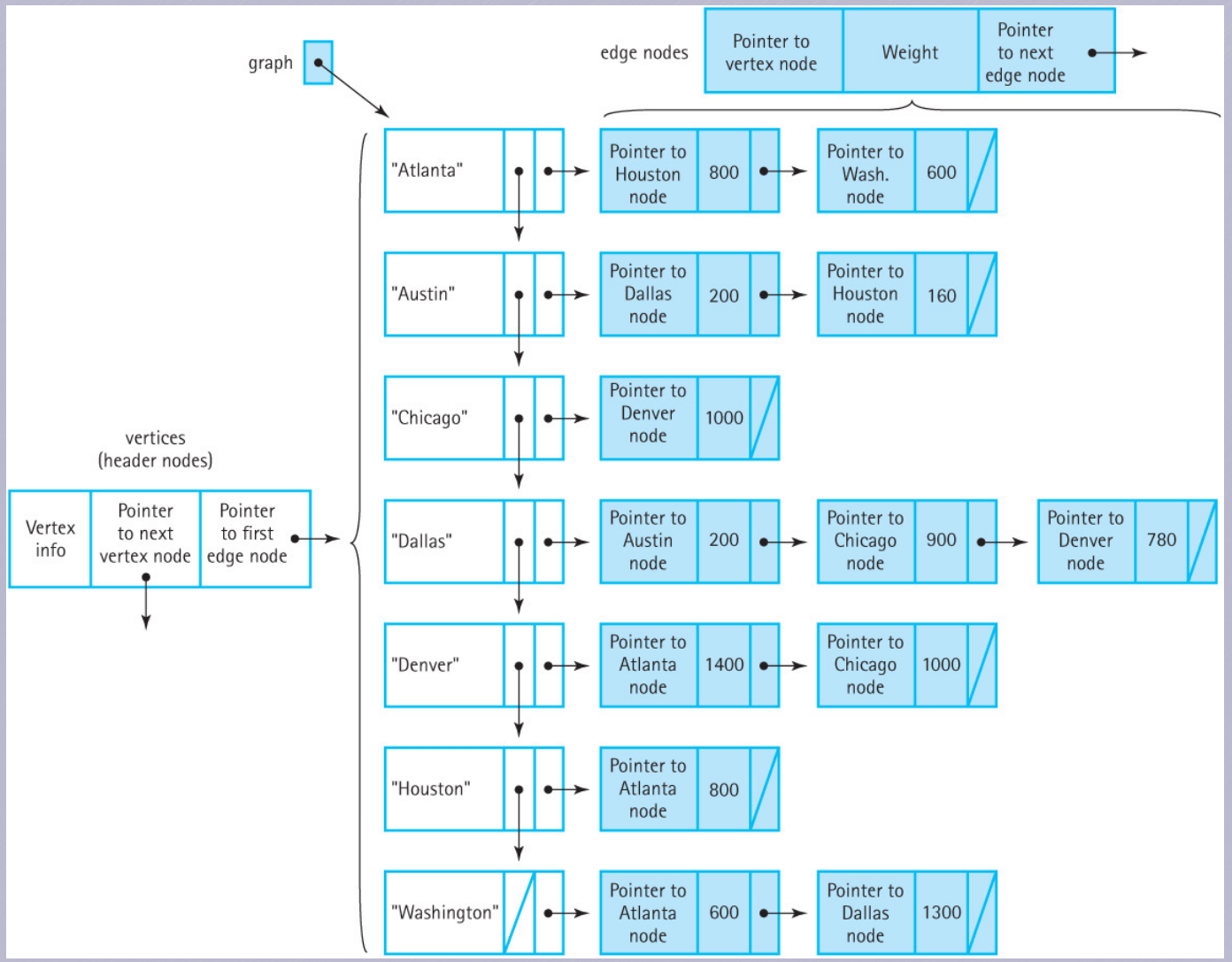


Figure 13.10 Another adjacency list representation of graphs

Which Version to Use?

- The array-based implementation is fast, with O(1) for adding and querying edges
- But it uses O(N^2) space for the matrix, which may be wasted if there are few edges
- The linked list-based implementation uses O(N) memory for sparse graphs, but can use O(N^2) if there are a lot of edges
- But looking up edges requires walking the lists