Graphs

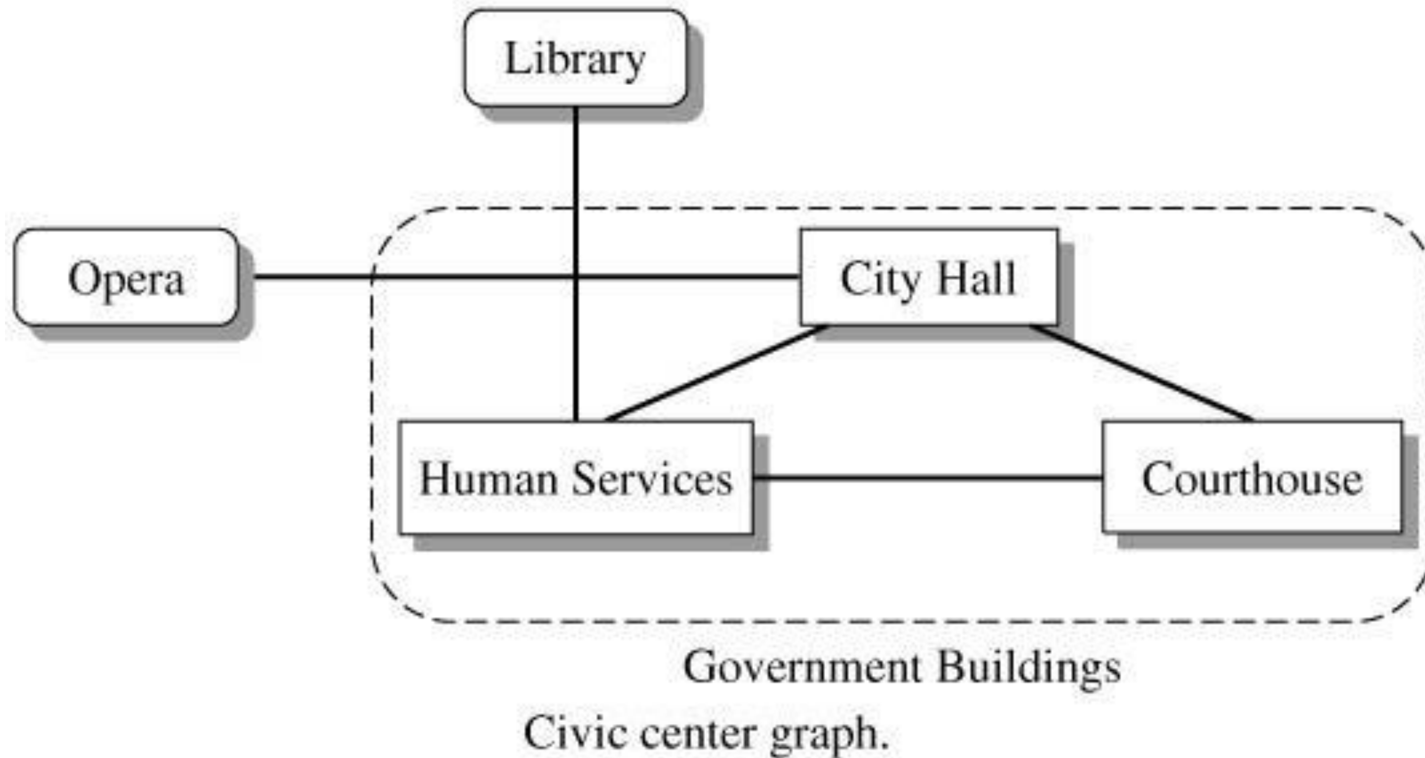
Graph Terminology

- A graph consists of a set of *vertices* V, along with a set of *edges* E that connect pairs of vertices.
  - An edge e = (vi,vj) connects vertices vi and vj.
  - A *self-loop* is an edge that connects a vertex to itself. We assume that none of our graphs have self-loops.

$$\text{Vertices} = \{v_1, v_2, v_3, \ldots, v_m\}$$
$$\text{Edges} = \{e_1, e_2, e_3, \ldots, {}_{en}\}$$

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Graph Terminology (continued)



Civic center graph.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

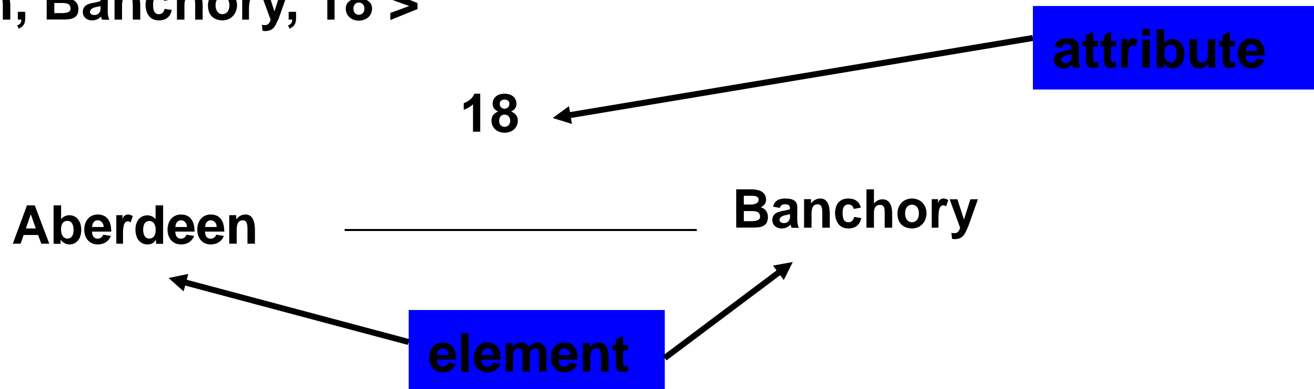
Data in Graphs

Both vertices and edges may have data values associated with them:

- ***element*** **: the data value at a node**
- ***attribute*** : **the data value associated with an edge**

Where there is a data value associated with an edge this is called a **weighted** edge**.**

< Aberdeen, Banchory, 18 >



Graph Terminology (continued)

- The *degree* of a vertex is the number of edges originating at the vertex.
- Two vertices in a graph are *adjacent* (*neighbors*) if there is an edge connecting the vertices.
- A path between vertices v and w is a series of edges leading from v to w. The path length is the number of edges in the path.

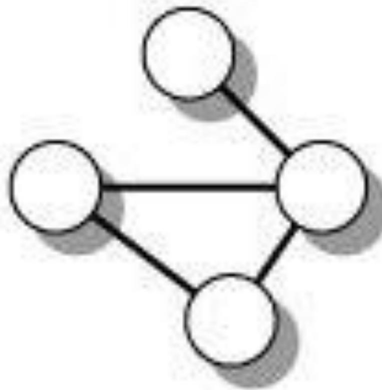
© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

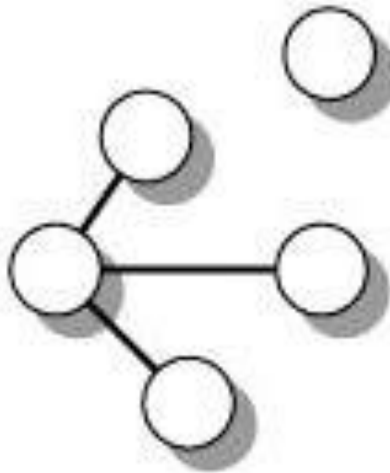
Graph Terminology (continued)

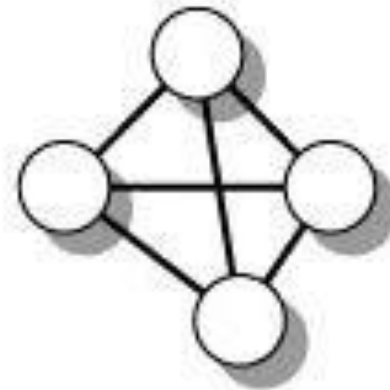
- A *path* is simple if all its edges are distinct. A cycle is a simple path that starts and ends on the same vertex.
- A graph is *connected* if there is a path between any pair of distinct vertices.
- A *complete graph* is a connected graph in which each pair of vertices is linked by an edge.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Graph Terminology (continued)


(a) Connected


(b) Disconnected


(c) Complete

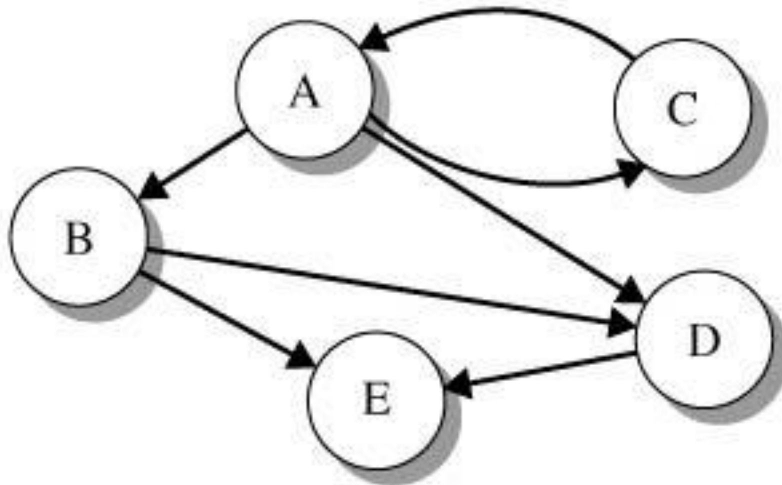
© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Graph Terminology (continued)

- A graph described until now is termed an *undirected graph*. Movement between vertices can occur in either direction.
- In a *digraph*, edges have a direction. There might be an edge from v to w but no edge from w to v.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Graph Terminology (continued)



Vertices V = {A, B, C, D, E}
Edges E = {(A, B), (A, C), (A, D), (B, D), (B, E), (C, A), (D, E)}

Sample digraph with five vertices and seven edges.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Graph Terminology (continued)

- In a digraph, a *directed path* (path) connecting vertices v_s and v_e is a sequence of directed edges that begin at v_s and end at v_e.
- The number of the edges that emanate from a vertex v is called the *out-degree* of the vertex.
- The number of the edges that terminate in vertex v is the *in-degree* of the vertex.

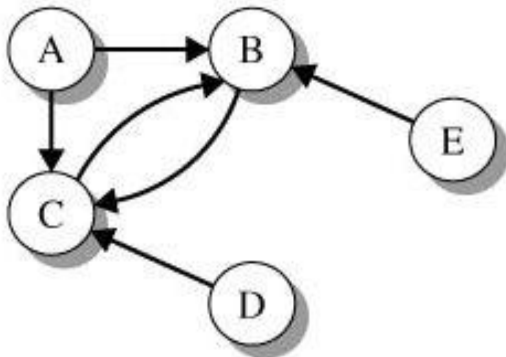
© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Graph Terminology (continued)

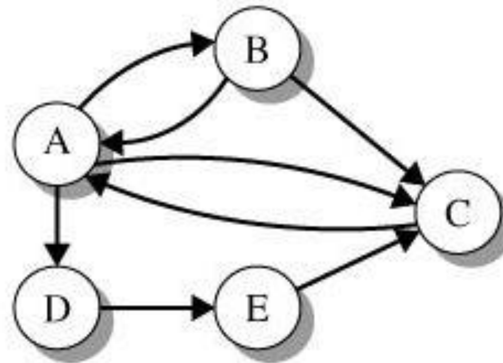
- A digraph is *strongly connected* if there is a path from any vertex to any other vertex.
- The digraph is *weakly connected* if, for each pair of vertices v_i and v_j, there is either a path $P(v_i, v_j)$ or a path $P(v_j, v_i)$.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

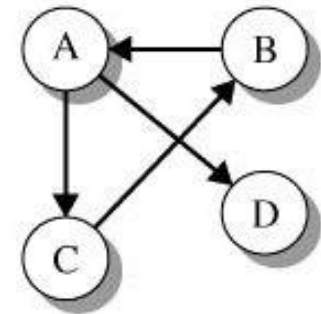
Graph Terminology (continued)



Not Strongly or Weakly Connected
(No path from E to D or from D to E)
(a)



Strongly Connected
(b)

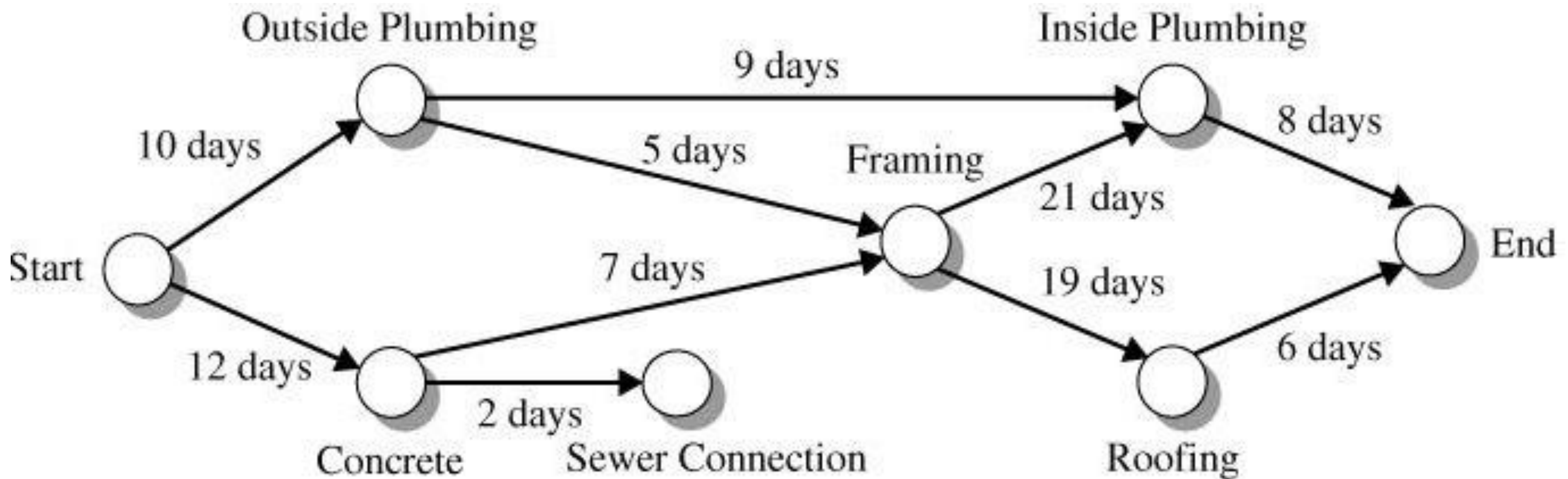


Weakly Connected
(No path from D to any other vertex)
(c)

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Graph Terminology (concluded)

- An *acyclic* graph has no cycles.
- Each edge in a *weighted digraph*, has a cost associated with traversing the edge.



© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Creating and Using Graphs

- The Graph interface specifies all basic graph operations including inserting and erasing vertices and edges.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Creating and Using Graphs (continued)

interface GRAPH<T>	**ds.util.Graph**
	Methods
boolean	**addEdge**(T v1, T v2, int w) If the edge (v1, v2) is not in the graph, adds the edge with weight w and returns true. Returns false if the edge is already in the graph. If v1 or v2 is not a vertex in the graph, throws IllegalArgumentException.
boolean	**addVertex**(T v) If v is not in the graph, adds it to the graph and returns true; otherwise, returns false.
void	**clear**() Removes all of the vertices and edges from the graph.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Creating and Using Graphs (continued)

interface GRAPH<T>	ds.util.Graph
	Methods (continued)
boolean	**containsEdge**(T v1, T v2) Returns true if there is an edge from v1 to v2 and returns false otherwise. If v1 or v2 is not a vertex in the graph, throws IllegalArgumentException.
boolean	**containsVertex**(Object v) Returns true if v is a vertex in the graph and false otherwise.
Set<T>	**getNeighbors**(T v) Returns the vertices that are adjacent to vertex v in a Set object. If v is not a graph vertex, throws IllegalArgumentException.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Creating and Using Graphs (continued)

interface GRAPH<T>	ds.util.Graph
	Methods (continued)
int	**getWeight**(T v1, T v2) Returns the weight of the edge connecting vertex v1 to v2. If the edge (v1,v2) does not exist, return -1. If v1 or v2 is not a vertex in the graph, throws IllegalArgumentException.
boolean	**isEmpty**() Returns true if the graph has no vertices or edges and false otherwise.
int	**numberOfEdges**() Returns the number of edges in the graph.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Creating and Using Graphs (continued)

interface GRAPH<T>	**ds.util.Graph**
	Methods (continued)
int	**numberOfVertices**() Returns the number of vertices in the graph.
boolean	**removeEdge**(T v1, T v2) If (v1,v2) is an edge, removes the edge and returns true; otherwise, returns false. If v1 or v2 is not a vertex in the graph, throws IllegalArgumentException.
boolean	**removeVertex**(Object v) If v is a vertex in the graph, removes it from the graph and returns true; otherwise, returns false.

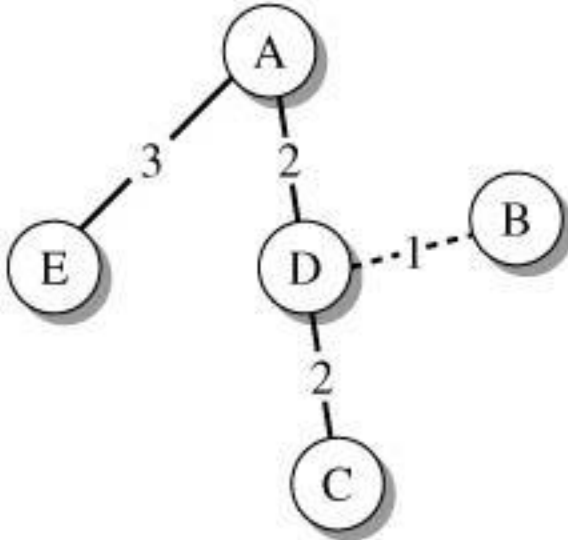
© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Creating and Using Graphs (continued)

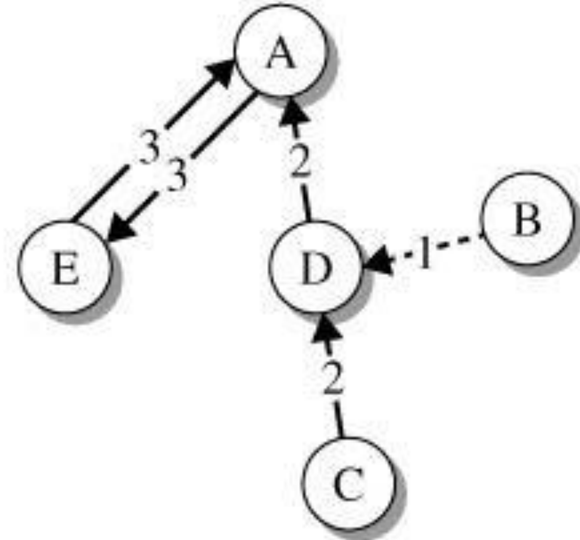
interface GRAPH<T>	**ds.util.Graph**
	Methods (continued)
int	**setWeight**(T v1, T v2, int w) If edge (v1, v2) is in the graph, update the weight of the edge and return the previous weight; otherwise, return -1. If v1 or v2 is not a vertex in the graph, throws IllegalArgumentException.
Set<T>	**vertexSet**() Returns a set-view of the vertices in the graph.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Creating and Using Graphs (continued)



(a)



(b)

Differences in implementation between a digraph and an undirected graph.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

The DiGraph Class

- The DiGraph class implements the Graph interface and adds other methods that are useful in applications.
  - A constructor creates an empty graph.
  - The methods inDegree() and outDegree() are special methods that access a properties that are unique to a digraph.
  - The static method readGraph() builds a graph whose vertices are strings.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

The DiGraph Class (continued)

- DiGraph method readGraph() inputs the vertex values and the edges from a textfile.
  - File format:

$$
\begin{array}{lll}
\texttt{(Number of Edges n)} & & \\
\texttt{Source}_1 & \texttt{Destination}_1 & \texttt{Weight}_1 \\
\texttt{Source}_2 & \texttt{Destination}_2 & \texttt{Weight}_2 \\
\texttt{. . .} & & \\
\texttt{Source}_n & \texttt{Destination}_n & \texttt{Weight}_n
\end{array}
$$

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

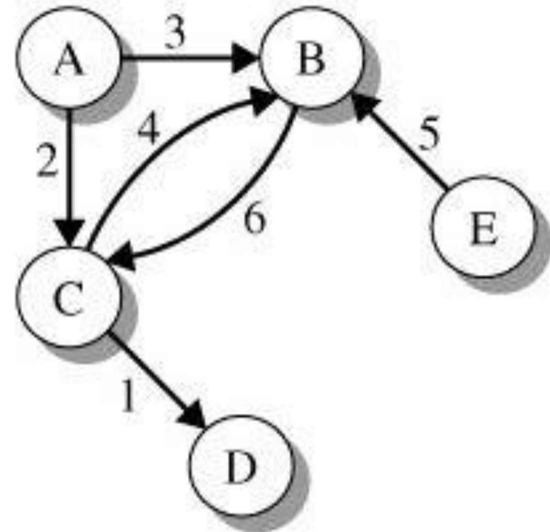
The DiGraph Class (continued)

- The method toString() provides a representation of a graph. For each vertex, the string gives the list of adjacent vertices along with the weight for the corresponding edge. The information for each vertex also includes its in-degree and out-degree.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

The DiGraph Class (continued)

```
File samplegraph.dat
5          // data for the vertices
A B C D E
6          // data for the edges
A B 3
A C 2
B C 6
C B 4
C D 1
E B 5
```




```
// input vertices, edges, and weights from samplegraph.dat
DiGraph g = DiGraph.readGraph("samplegraph.dat");

// display the graph
System.out.println(g)
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

The DiGraph Class (continued)

```
Output:
A:   in-degree 0  out-degree 2
     Edges: B(3)  C(2)
B:   in-degree 3  out-degree 1
     Edges: C(6)
C:   in-degree 2  out-degree 2
     Edges: B(4)  D(1)
D:   in-degree 1  out-degree 0
     Edges:
E:   in-degree 0  out-degree 1
     Edges: B(5)
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Program 24.1 (continued)

```
import java.io.FileNotFoundException;

import ds.util.Set;
import ds.util.Iterator;
import ds.util.DiGraph;

public class Program24_1
{
   public static void main(String[] args)
   throws FileNotFoundException
   {
      // construct graph with vertices of type
      // String by reading from the file "graphIO.dat"
      DiGraph<String> g =
            DiGraph.readGraph("graphIO.dat");
      String vtxName;
      // sets for vertexSet() and adjacent
      // vertices (neighbors)
      Set<String> vtxSet, neighborSet;
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Program 24.1 (continued)

```
// output number of vertices and edges
System.out.println("Number of vertices: " +
      g.numberOfVertices());
System.out.println("Number of edges: " +
      g.numberOfEdges());

// properties relative to vertex A
System.out.println("inDegree for A: " +
      g.inDegree("A"));
System.out.println("outDegree for A: " +
      g.outDegree("A"));
System.out.println("Weight e(A,B): " +
      g.getWeight("A","B"));

// delete edge with weight 2
g.removeEdge("B", "A");

// delete vertex "E" and edges (E,C),
// (C,E) and (D,E)
g.removeVertex("E");
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Program 24.1 (continued)

```
/* add and update attributes of the graph */
// increase weight from 4 to 8
g.setWeight("A","B",8);
// add vertex F
g.addVertex("F");
// add edge (F,D) with weight 3
g.addEdge("F","D",3);

// after all updates, output the graph
// and its properties
System.out.println("After all the graph updates");
System.out.println(g);

// get the vertices as a Set and
// create set iterator
vtxSet = g.vertexSet();
Iterator vtxIter = vtxSet.iterator();
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Program 24.1 (concluded)

```
      // scan the vertices and display
      // the set of neighbors
      while(vtxIter.hasNext())
      {
         vtxName = (String)vtxIter.next();
         neighborSet = g.getNeighbors(vtxName);
         System.out.println("   Neighbor set for " +
               "vertex " + vtxName + " is "
               + neighborSet);
      }
   }
}
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Program 24.1 (Run)

```
Number of vertices: 5
Number of edges: 8
inDegree for A: 1
outDegree for A: 3
Weight e(A,B): 4
After all the graph updates
A:  in-degree 0  out-degree 3
    Edges: B(8)  C(7)  D(6)
B:  in-degree 2  out-degree 0
    Edges:
C:  in-degree 1  out-degree 1
    Edges: B(3)
D:  in-degree 2  out-degree 0
    Edges:
F:  in-degree 0  out-degree 1
    Edges: D(3)
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Program 24.1 (Run, concluded)

```
Neighbor set for vertex D is []
Neighbor set for vertex F is [D]
Neighbor set for vertex A is [D, B, C]
Neighbor set for vertex B is []
Neighbor set for vertex C is [B]
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Graph Traversal Algorithms

- In general, graphs do not have a vertex, like a root, that initiates unique paths to each of the vertices. From any starting vertex in a graph, it might not be possible to search all of the vertices. In addition, a graph could have a cycle that results in multiple visits to a vertex.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Graph Traversal Algorithms (continued)

- The breadth-first search visits vertices in the order of their path length from a starting vertex. It may not visit every vertex of the graph
- The depth-first search traverses all the vertices of a graph by making a series of recursive calls that follow paths through the graph.

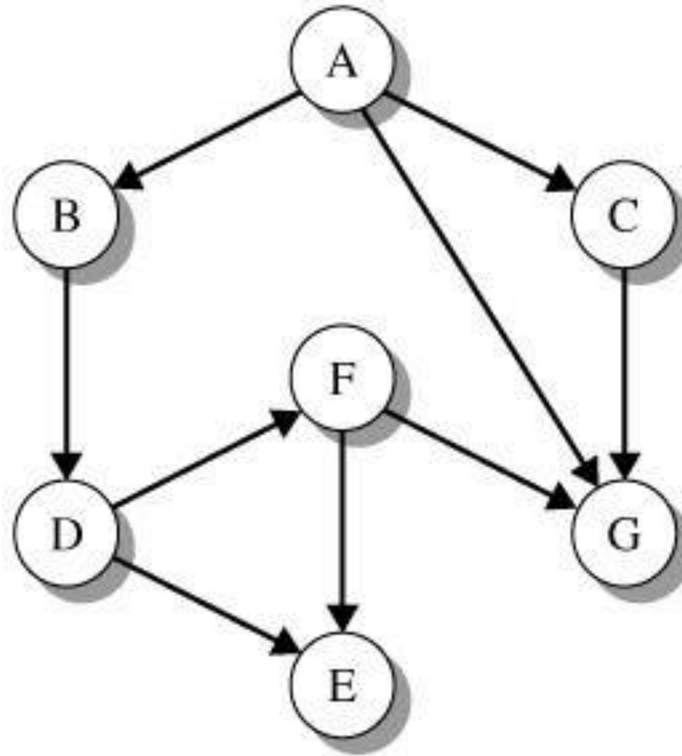
© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Graph Traversal Algorithms (continued)

- Graph algorithms discern the state of a vertex during the algorithm by using the colors WHITE, GRAY, and BLACK.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Breadth-First Search Algorithm

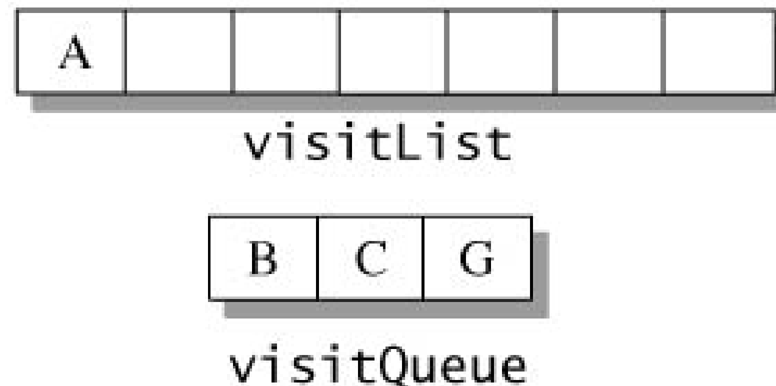


Demonstration graph
for the breadth-first search.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Breadth-First Search Algorithm (continued)

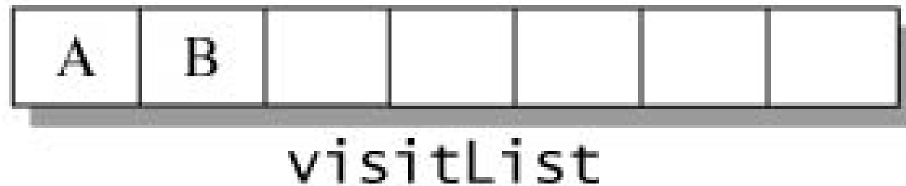
- Color all vertices of the sample graph WHITE and push the starting vertex (A) onto the queue visitQueue.
- Pop A from the queue, color it BLACK, and insert it into visitList, which is the list of visited vertices. Push all WHITE neighbors onto the queue.

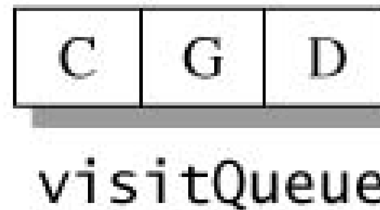


© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Breadth-First Search Algorithm (continued)

- Pop B from the queue and place it in visitList with color BLACK. The only adjacent vertex for B is D, which is still colored WHITE. Color D GRAY and add it to the queue

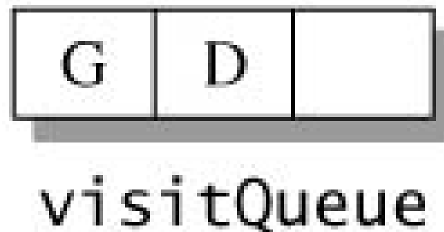




© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Breadth-First Search Algorithm (continued)

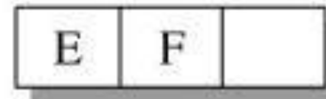
- Pop C and place it in visitList. The adjacent vertex G is GRAY. No new vertices enter the queue.



© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Breadth-First Search Algorithm (continued)

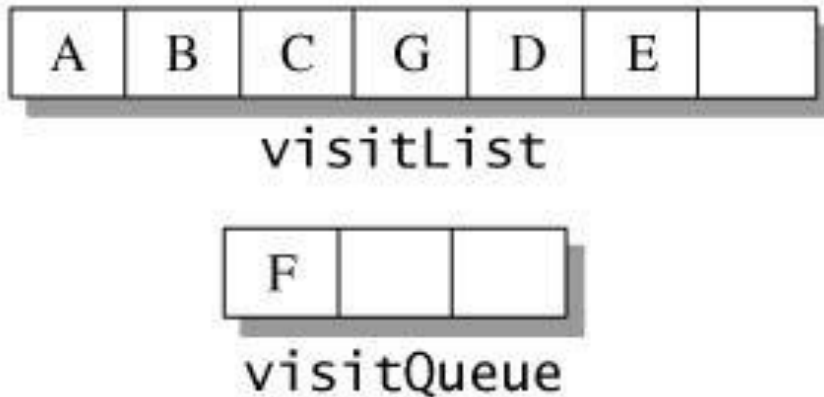
- Pop vertex G from the queue and place it in visitList. G has no adjacent vertices, so pop D from the queue. The neighbors, E and F, enter the queue.

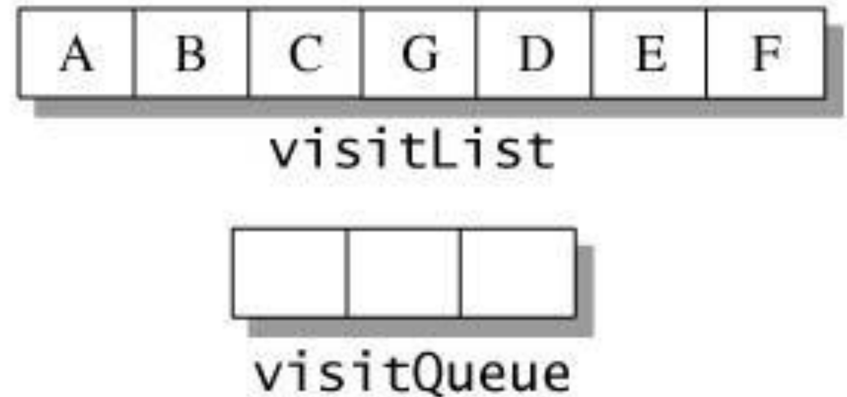


© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Breadth-First Search Algorithm (continued)

- Continue in this fashion until the queue is empty.





© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Implementing Breadth-First Search

- Define

```
public enum VertexColor
{
        WHITE, GRAY, BLACK
}
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Implementing Breadth-First Search (continued)

- The DiGraph class declares three methods that access and update the color attribute of a vertex.

class DiGraph<T> (color methods)	ds.util
void	**colorWhite**() Set the color in each vertex to WHITE.
VertexColor	**getColor**(T v) Returns the color of vertex v. If v is not a graph vertex, throws IllegalArgumentException.
VertexColor	**setColor**(T v, VertexColor c) Sets the color of vertex v and returns the previous color. If v is not a graph vertex, throws IllegalArgumentException.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Implementing Breadth-First Search (continued)

- The method bfs() returns a list of vertices visited during the breadth-first search from a starting vertex.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

bfs()

```
// perform the breadth-first traversal
// from sVertex and return the list
// of visited vertices
public static <T> LinkedList<T> bfs(
DiGraph<T> g, T sVertex)
{
   // queue stores adjacent vertices; list
   // stores visited vertices
   LinkedQueue<T> visitQueue = new LinkedQueue<T>();
   LinkedList<T> visitList = new LinkedList<T>();

   // set and iterator retrieve and scan
   // neighbors of a vertex
   Set<T> edgeSet;
   Iterator<T> edgeIter;

   T currVertex = null, neighborVertex = null;
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

bfs() (continued)

```
// check that starting vertex is valid
if (!g.containsVertex(sVertex))
   throw new IllegalArgumentException(
         "bfs(): starting vertex not in the graph");

// color all vertices WHITE
g.colorWhite();

// initialize queue with starting vertex
visitQueue.push(sVertex);

while (!visitQueue.isEmpty())
{
   // remove a vertex from the queue, color
   // it black, and add to the list of
   // visited vertices
   currVertex = visitQueue.pop();
   g.setColor(currVertex,VertexColor.BLACK);
   visitList.add(currVertex);
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

bfs() (continued)

```
// obtain the set of neighbors for current vertex
edgeSet = g.getNeighbors(currVertex);
// sequence through the neighbors and look
// for vertices that have not been visited
edgeIter = edgeSet.iterator();
while (edgeIter.hasNext())
{
   neighborVertex = edgeIter.next();
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

bfs() (concluded)

```
         if (g.getColor(neighborVertex) ==
               VertexColor.WHITE)
         {
            // color unvisited vertex GRAY and
            // push it onto queue
            g.setColor(neighborVertex,VertexColor.GRAY);
            visitQueue.push(neighborVertex);
         }
      }
   }

   return visitList;
}
```

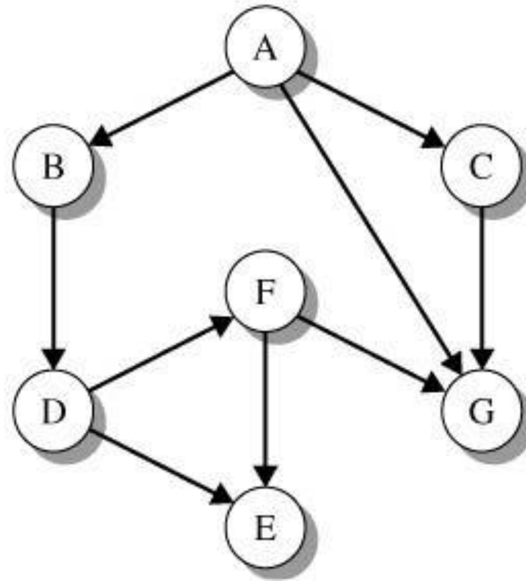
© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Running Time of Breadth-First Search

- The running time for the breadth-first search is $O(V + E)$, where V is the number of vertices and E is the number of edges.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Breadth-First Search Example




```
// create a graph g and declare startVertex and visitList
DiGraph<String> g = DiGraph.readGraph("bfsgraph.dat");
String startVertex;
List<String> visitList;
                 ...
// call bfs() with arguments g and startVertx
visitList = DiGraphs.bfs(g, startVertex);
// output the visitList
System.out.println("BFS visitList from " + startVertex +
                   ":   " + visitList);
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Breadth-First Search Example (concluded)

Output:

```
    Run 1: (startVertex = "A")
       BFS visitList from A:   [A, G, B, C, D, E, F]
    Run 2: (startVertex = "D")
       BFS visitList from D:   [D, E, F, G]
    Run 3: (startVertex = "E")
       BFS visitList from E:   [E]
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Depth-First Visit

- The depth-first visit algorithm is modeled after the recursive postorder scan of a binary tree. In the tree, a node is visited only after visits are made to all of the nodes in its subtree. In the graph, a node is visited only after visiting all of the nodes in paths that emanate from the node.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Depth-First Visit (continued)

- Color all vertices WHITE. A vertex is colored GRAY when it is first contacted in a recursive descent. Only when the vertex is actually visited does it become BLACK.
- Begin at a starting vertex and search down paths of neighbors until reaching a vertex that has no neighbors or only neighbors that are BLACK. At this point, a visit occurs at the "terminal" vertex and it becomes BLACK.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

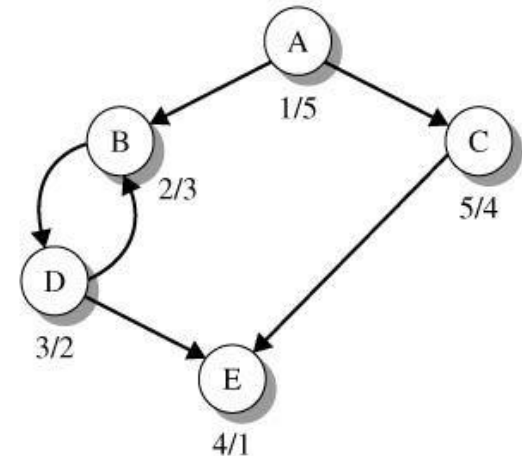
Depth-First Visit (continued)

- Backtrack to the previous recursive step and look for another adjacent vertex and launch a scan down its paths. There is no ordering among vertices in an adjacency list, so the paths and hence the order of visits to vertices can vary.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Depth-First Visit (continued)

- Discover A (color GRAY)
- Discover B (color GRAY)
- Discover D (color GRAY)
- Discover E (color GRAY, then BLACK)
- Backtrack D (color BLACK)
- Backtrack B (color BLACK)
- Backtrack A (A remains GRAY)
- Discover C (color BLACK)
- Backtrack A (color BLACK). Visit com



Graph illustrating depth-first visit.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

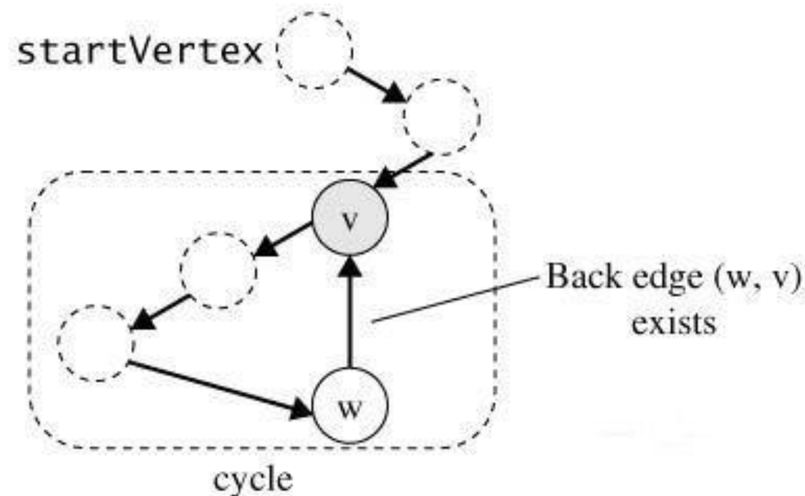
Depth-First Visit (continued)

- The depth-first visit is a recursive algorithm that distinguishes the discovery and finishing time (when a vertex becomes BLACK) of a vertex.
- The depth-first visit returns a list of the vertices found in the reverse order of their finishing times.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Depth-First Visit (continued)

- An edge that connects a vertex to a neighbor that has color GRAY is called a *back edge*.
  - A depth-first visit has a cycle if and only if it has a back edge.



Discovering a cycle assuming v is a GRAY neighbor of w (Edge (w, v) is a back edge).

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

dfsVisit()

```
// depth-first visit assuming a WHITE starting
// vertex; dfsList contains the visited vertices in
// reverse order of finishing time; when checkForCycle
// is true, throws IllegalPathStateException if it
// detects a cycle
public static <T> void dfsVisit(DiGraph<T> g, T sVertex,
LinkedList<T> dfsList, boolean checkForCycle)
{
   T neighborVertex;
   Set<T> edgeSet;
   // iterator to scan the adjacency set of a vertex
   Iterator<T> edgeIter;
   VertexColor color;

   if (!g.containsVertex(sVertex))
      throw new IllegalArgumentException(
            "dfsVisit(): vertex not in the graph");
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

dfsVisit() (continued)

```
// color vertex GRAY to note its discovery
g.setColor(sVertex, VertexColor.GRAY);

edgeSet = g.getNeighbors(sVertex);

// sequence through the adjacency set and look
// for vertices that are not yet discovered
// (colored WHITE); recursively call dfsVisit()
// for each such vertex; if a vertex in the adjacency
// list is GRAY, the vertex was discovered during a
// previous call and there is a cycle that begins and
// ends at the vertex; if checkForCycle is true,
// throw an exception
edgeIter = edgeSet.iterator();
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

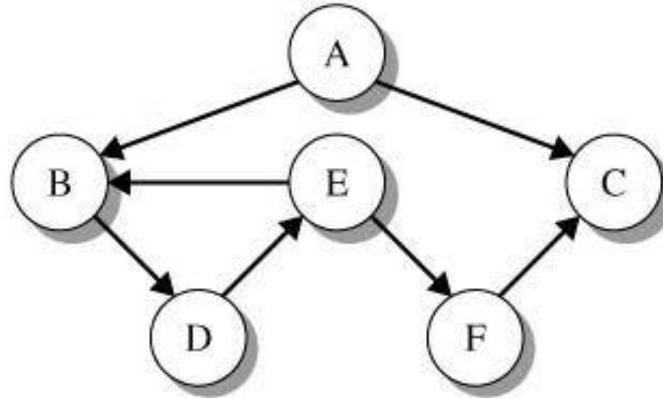
dfsVisit() (concluded)

```
   while (edgeIter.hasNext())
   {
      neighborVertex = edgeIter.next();
      color = g.getColor(neighborVertex);
      if (color == VertexColor.WHITE)
         dfsVisit(g, neighborVertex, dfsList,
               checkForCycle);
      else if (color == VertexColor.GRAY && checkForCycle)
         throw new IllegalPathStateException(
               "dfsVisit(): graph has a cycle");
   }

   // finished with vertex sVertex; make it BLACK
   // and add it to the front of dfsList
   g.setColor(sVertex, VertexColor.BLACK);
   dfsList.addFirst(sVertex);
}
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Depth-First Visit Example

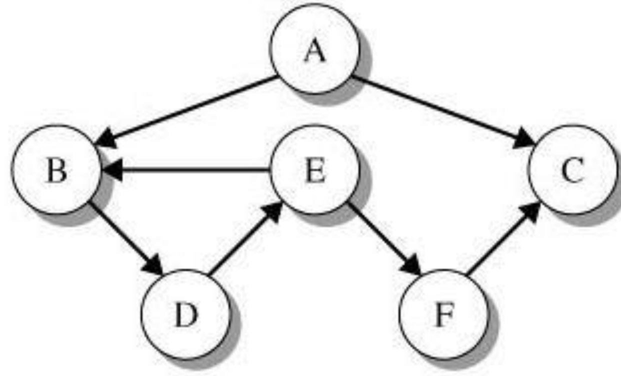



```
LinkedList<String> finishOrder = new LinkedList<String>();
g.colorWhite();
DiGraphs.dfsVisit(g, "B", finishOrder, false);
```

```
Output:  finishOrder: [B, D, E, F, C]
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Depth-First Visit Example (concluded)



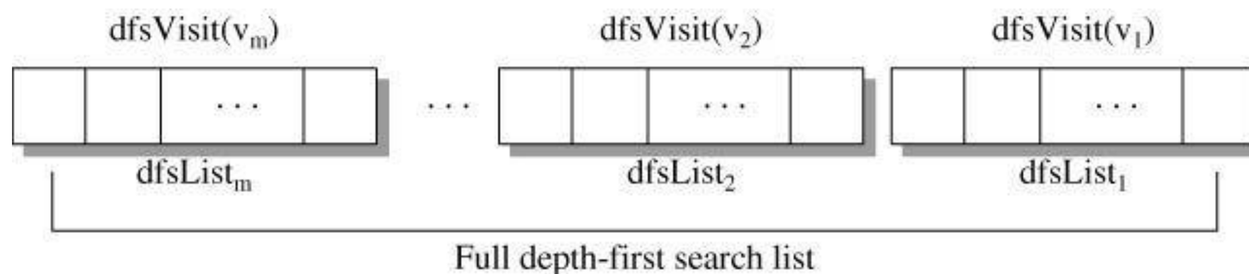

```
finishOrder = new LinkedList<String>();
g.colorWhite();
try
{
   DiGraphs.dfsVisit(g, "E", finishOrder, true);
   System.out.println("finishOrder: " + finishOrder);
}
catch (IllegalPathStateException ipse)
{
   System.out.println(ipse.getMessage());
}
```

```
Output: dfsVisit(): cycle involving vertices D and E
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Depth-First Search Algorithm

- Depth-first search begins with all WHITE vertices and performs depth-first visits until all vertices of the graph are BLACK.
- The algorithm returns a list of all vertices in the graph in the reverse order of their finishing times.



Combined list of visits for the depth-first search.

© 2005 Pears
Inc., Upper Saddle River, NJ. All rights reserved.

dfs()

```
// depth-first search; dfsList contains all
// the graph vertices in the reverse order
// of their finishing times
public static <T> void dfs(DiGraph<T> g,
LinkedList<T> dfsList)
{
   Iterator<T> graphIter;
   T vertex = null;

   // clear dfsList
   dfsList.clear();

   // initialize all vertices to WHITE
   g.colorWhite();
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

dfs() (concluded)

```
   // call dfsVisit() for each WHITE vertex
   graphIter = g.vertexSet().iterator();
   while (graphIter.hasNext())
   {
      vertex = graphIter.next();
      if (g.getColor(vertex) == VertexColor.WHITE)
         dfsVisit(g, vertex, dfsList, false);
   }
}
```

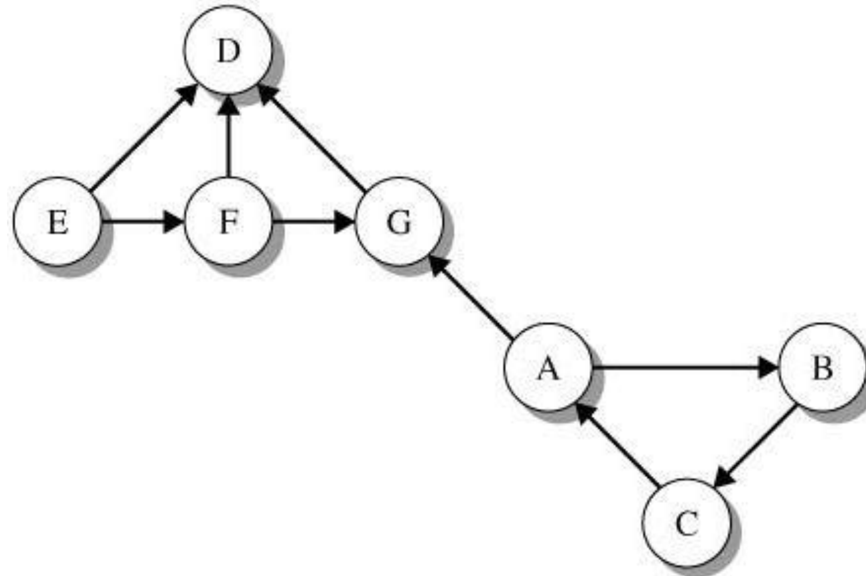
© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Running Time for Depth-First Search

- An argument similar to that for the breadth-first search shows that that the running time for dfs() is O(V+E), where V is the number of vertices in the graph and E is the number of edges.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Depth-First Search Example



dfsVisit() starting at E followed by dfsVisit() starting at A
dfsList: [A, B, C, E, F, G, D]

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Acyclic Graphs

- To check for a cycle anywhere in a graph, traverse all of the vertices by using multiple calls to dfsVisit().
  - This is essentially the dfs() algorithm where the focus is on identifying cycles and not obtaining a list of visited vertices. This approach relies on the fact that any cycle must be included within one of the calls to dfsVisit(). The method, with checkForCycle set to true, identifies the cycle.

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

acyclic()

```
// determine if the graph is acyclic
public static <T> boolean acyclic(DiGraph<T> g)
{
   // use for calls to dfsVisit()
   LinkedList<T> dfsList = new LinkedList<T>();
   Iterator<T> graphIter;
   T vertex = null;

   // initialize all vertices to WHITE
   g.colorWhite();

   // call dfsVisit() for each WHITE vertex; catch
   // an IllegalPathStateException in a call to
   // dfsVisit()
   try
   {
      // call dfsVisit() for each WHITE vertex
      graphIter = g.vertexSet().iterator();
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

acyclic() (concluded)

```
        while (graphIter.hasNext())
        {
            vertex = graphIter.next();
            if (g.getColor(vertex) == VertexColor.WHITE)
                dfsVisit(g,vertex, dfsList, true);
        }
    }

    catch (IllegalPathStateException iae)
    {
        return false;
    }

    return true;
}
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Program 24.2

```
import java.io.FileNotFoundException;

import ds.util.DiGraph;
import ds.util.DiGraphs;

public class Program24_2
{
   public static void main(String[] args)
   throws FileNotFoundException
   {
      DiGraph<String> g = DiGraph.readGraph("cycle.dat");

      // determine if the graph is acyclic
      if (DiGraphs.acyclic(g))
         System.out.println("Graph is acyclic");
      else
         System.out.println("Graph is not acyclic");
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.

Program 24.2 (concluded)

```
      // add edge (E,B) to create a cycle
      System.out.print("   Adding edge (E,B): ");
      g.addEdge("E", "B", 1);

      // retest the graph to see if it is acyclic
      if (DiGraphs.acyclic(g))
         System.out.println("New graph is acyclic");
      else
         System.out.println("New graph is not acyclic");
   }
}
```

```
Run:

Graph is acyclic
   Adding edge (E,B): New graph is not acyclic
```

© 2005 Pearson Education, Inc., Upper Saddle River, NJ. All rights reserved.