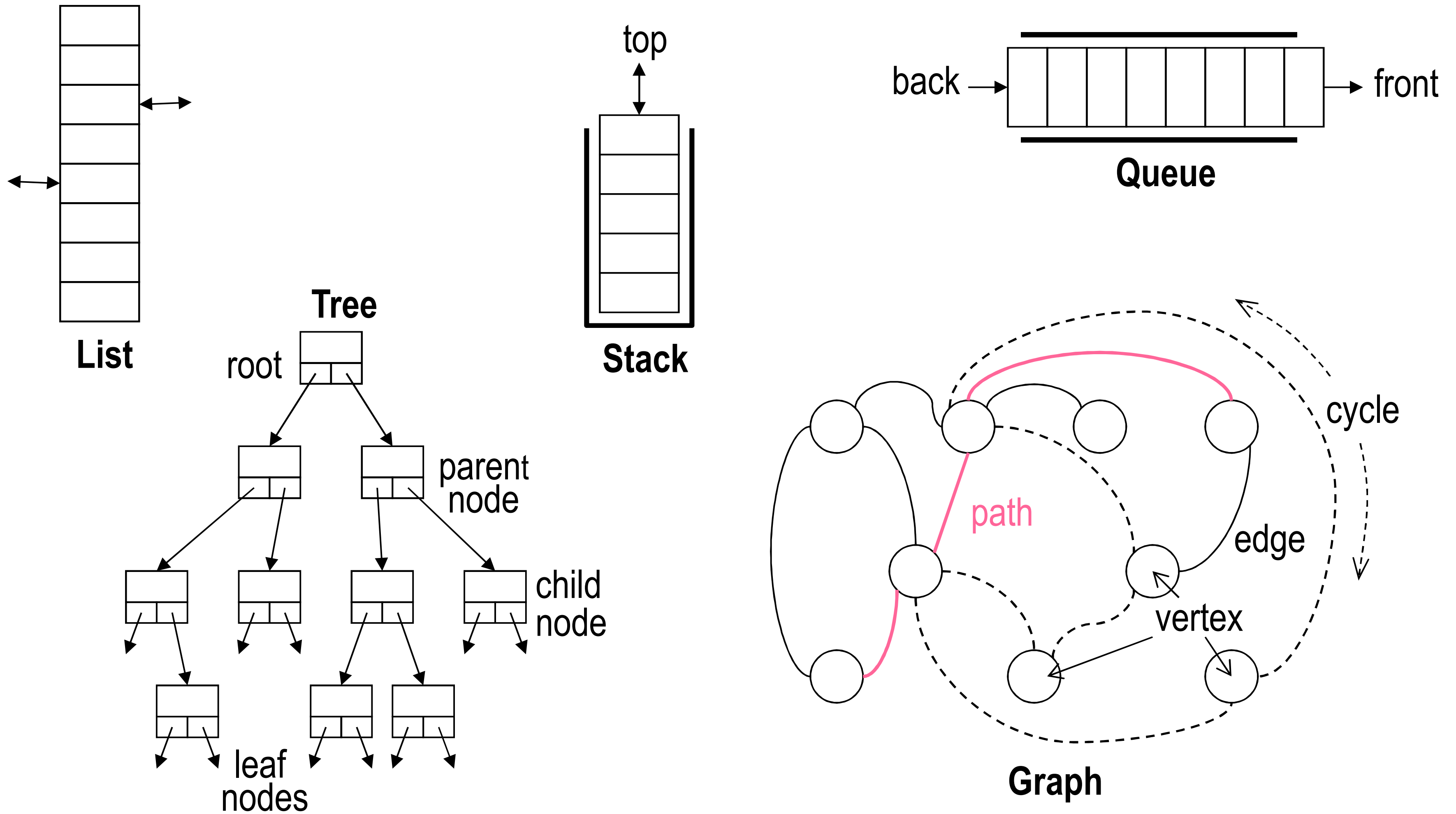
Lecture 9

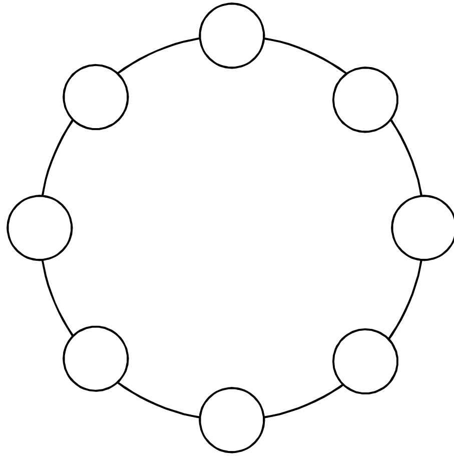
Graphs

This course is intended for 3rd and/or 4th year undergraduate majors in Computer Science.

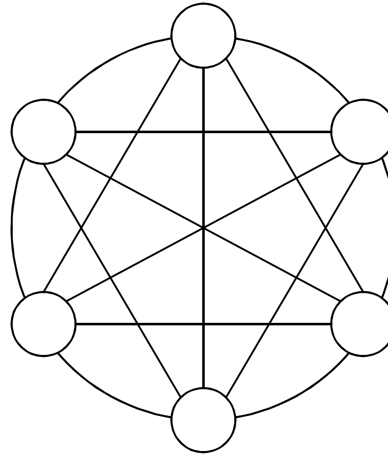
You need to be familiar with the design and use of basic data structures such as Lists, Stacks, Queues, Trees, and Graphs.



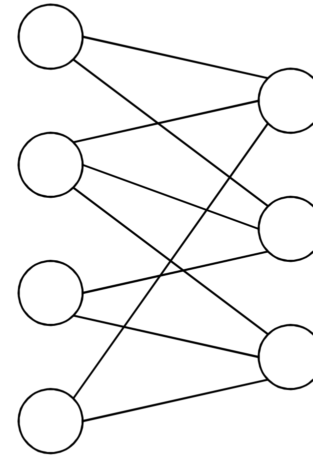
Types of Graphs



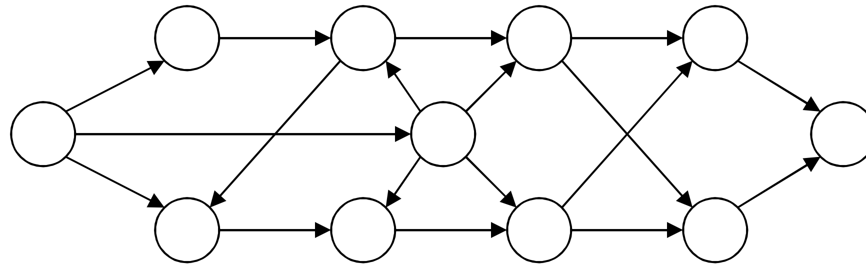
ring



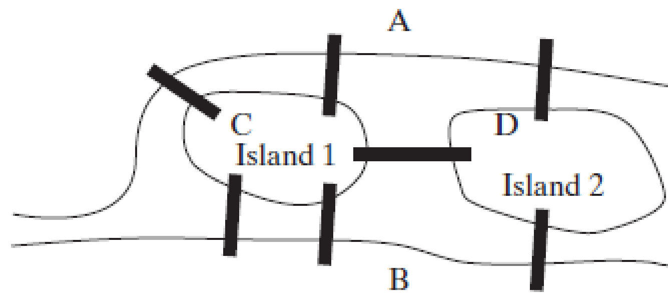
complete graph



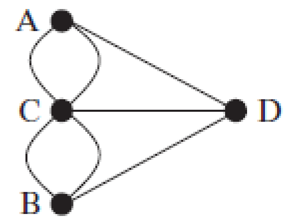
bipartite graph



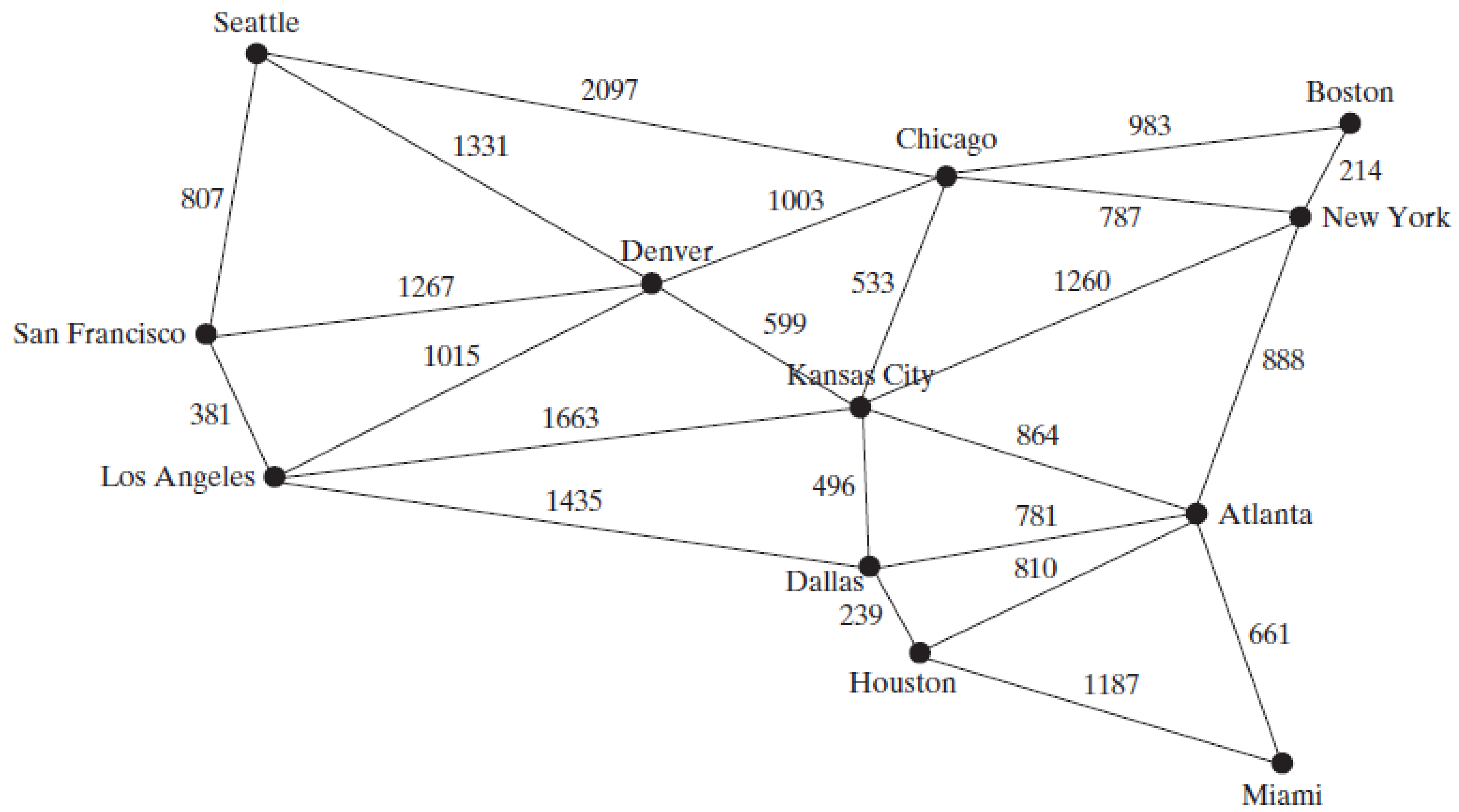
directed acyclic graph

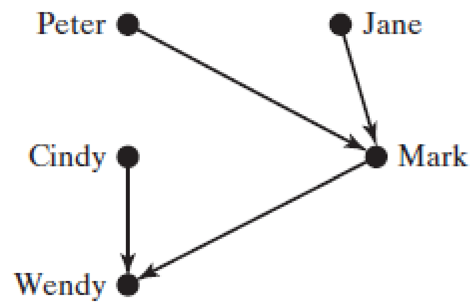


(a) Seven bridges sketch

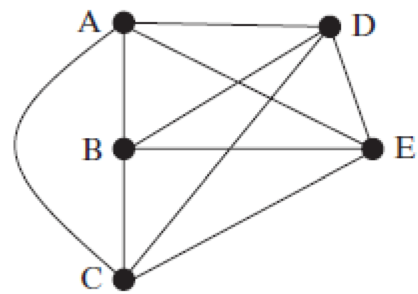


(b) Graph model


Seattle
2097
Boston
983
Chicago
1331
214
807
1003
787
New York
Denver
1267
533
1260
San Francisco
599
1015
888
Kansas City
381
1663
864
Los Angeles
496
1435
781
Atlanta
810
Dallas
239
661
Houston
1187
Miami

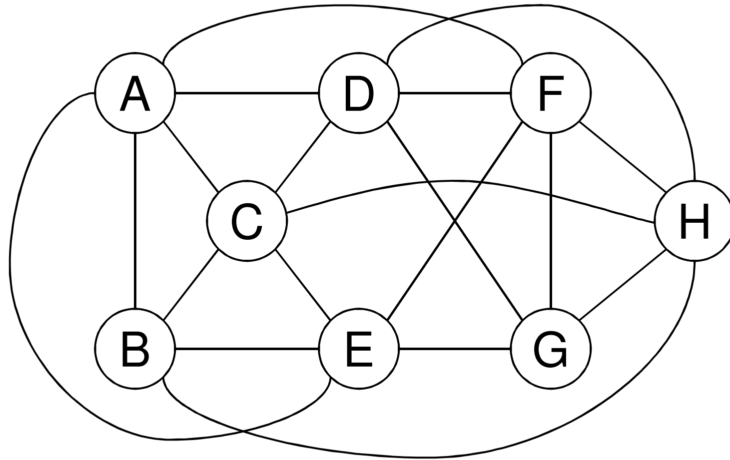


(a) A directed graph



(b) A complete graph

Graph Representations



node list - lists the nodes connected to each node

edge list - lists each of the edges as a pair of nodes
undirected edges may be listed twice XY and YX
in order to simplify algorithm implementation

adjacency matrix - for an n-node graph we build an nxn array with 1's indicating edges and 0's no edge the main diagonal of the matrix is unused unless a node has an edge connected to itself. If graph is weighted, 1's are replaced with edge weight values

node list

A - B C D E F
B - A C E H
C - A B D E H
D - A C F G H
E - A B C F G
F - A D E G H
G - D E F H
H - B C D F G

adjacency matrix

	A	B	C	D	E	F	G	H
A	–	1	1	1	1	1	0	0
B	1	–	1	0	1	0	0	1
C	1	1	–	1	1	0	0	1
D	1	0	1	–	0	1	1	1
E	1	1	1	0	–	1	1	0
F	1	0	0	1	1	–	1	1
G	0	0	0	1	1	1	–	1
H	0	1	1	1	0	1	1	–

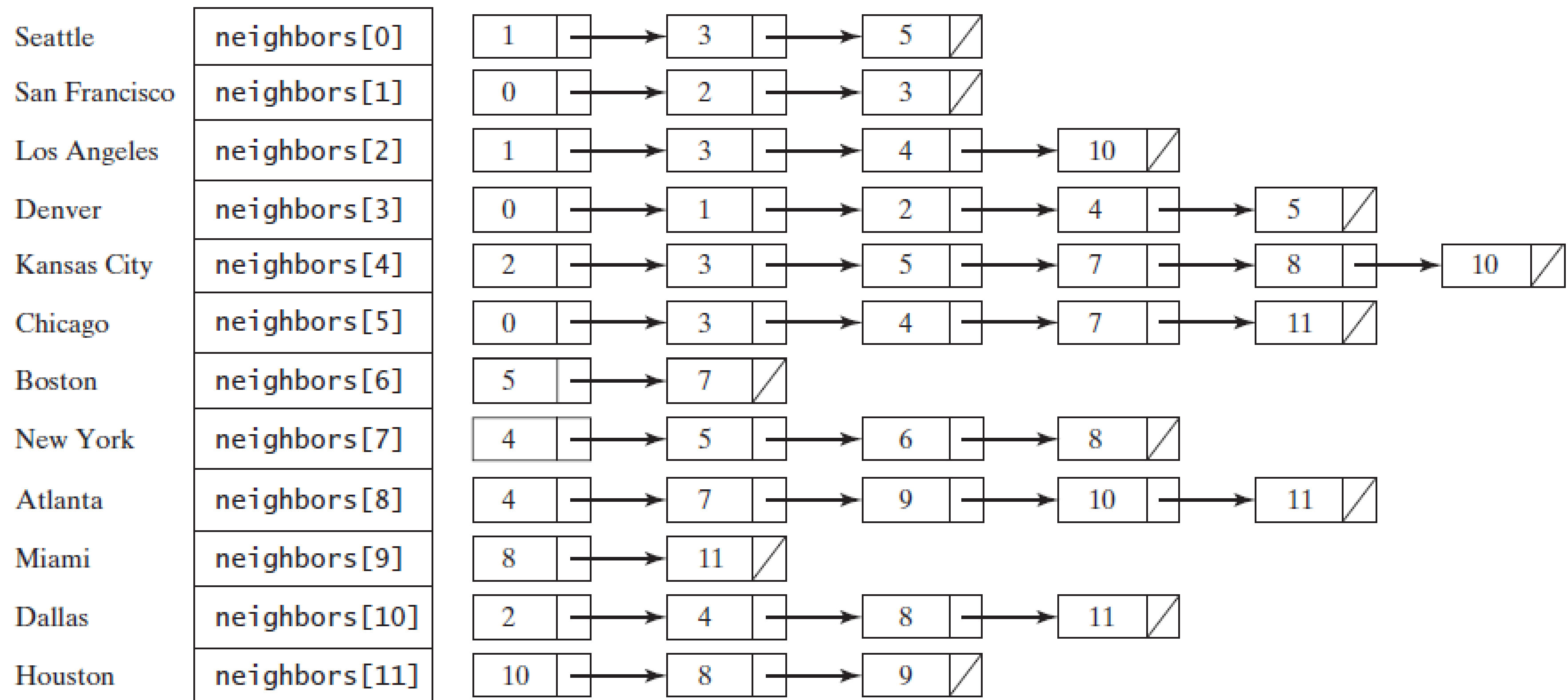
edge list

A B
A C
A D
A E
A F
B A
B C
B E
B H
C A
C B
C D
C E
C H
D A
D C
D F
D G
D H
E A
E B
E C
E F
E G
F A
F D
F E
F G
F H
G D
G E
G F
G H
H B
H C
H D
H F
H G

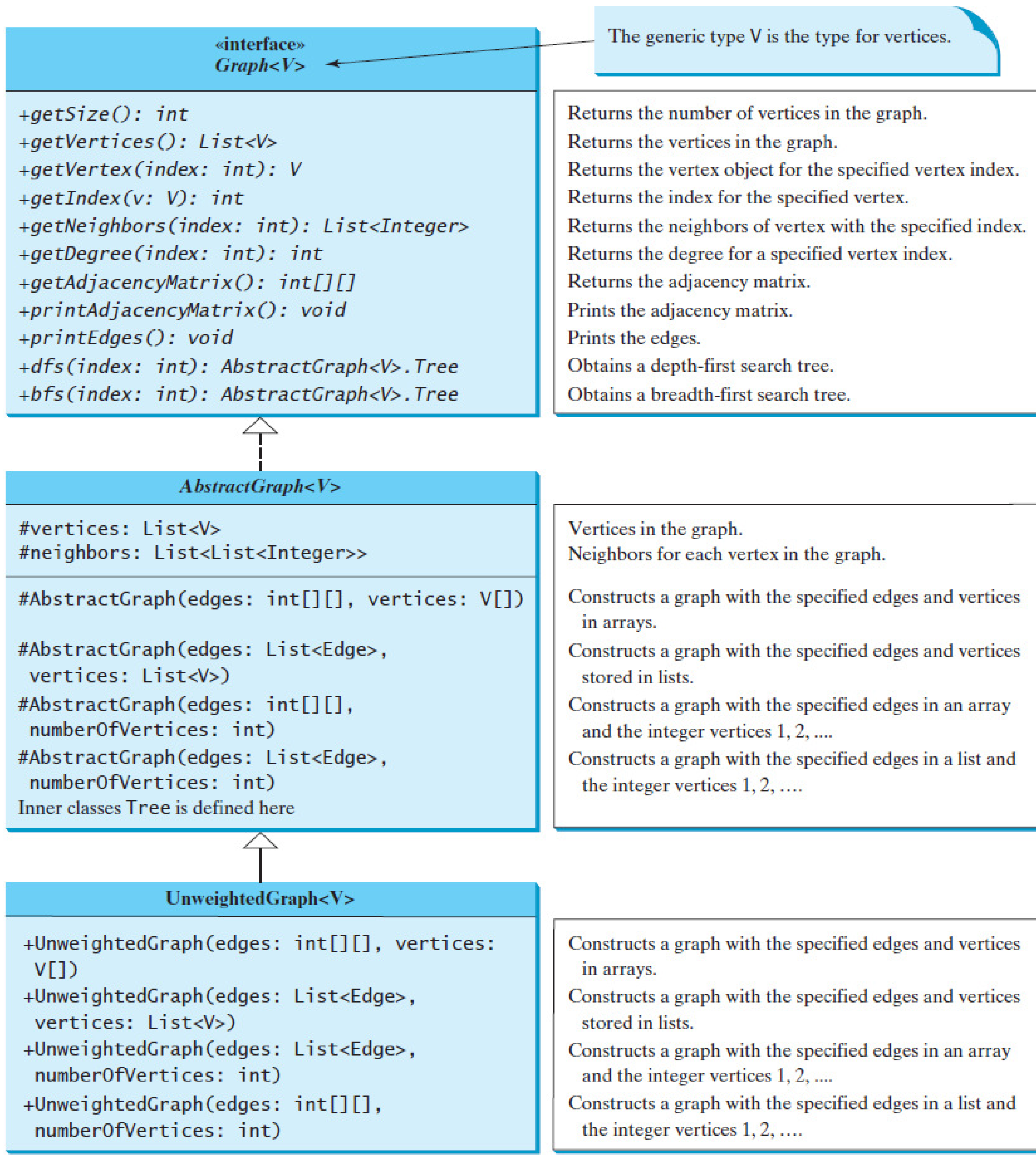
vertices[0]	Seattle
vertices[1]	San Francisco
vertices[2]	Los Angeles
vertices[3]	Denver
vertices[4]	Kansas City
vertices[5]	Chicago
vertices[6]	Boston
vertices[7]	New York
vertices[8]	Atlanta
vertices[9]	Miami
vertices[10]	Dallas
vertices[11]	Houston

```
int[][] edges = {
  {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}
};
```

```java
int[][] adjacencyMatrix = {
  {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
};
```



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



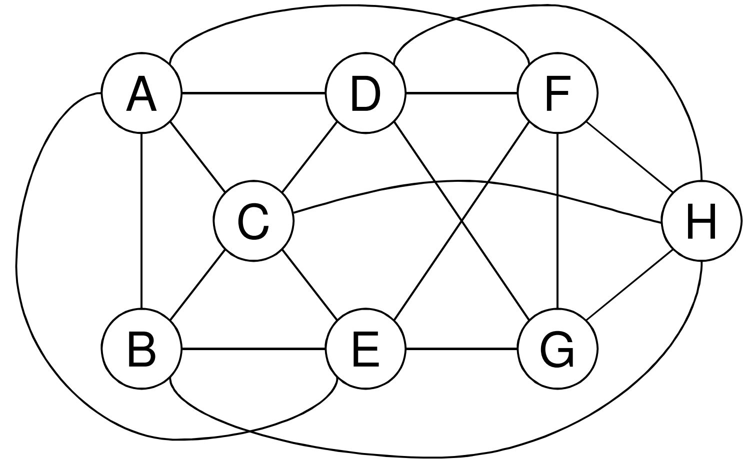
«interface»
Graph<V>
The generic type V is the type for vertices.
+getSize(): int
+getVertices(): List<V>
+getVertex(index: int): V
+getIndex(v: V): int
+getNeighbors(index: int): List<Integer>
+getDegree(index: int): int
+getAdjacencyMatrix(): int[][]
+printAdjacencyMatrix(): void
+printEdges(): void
+dfs(index: int): AbstractGraph<V>.Tree
+bfs(index: int): AbstractGraph<V>.Tree
Returns the number of vertices in the graph.
Returns the vertices in the graph.
Returns the vertex object for the specified vertex index.
Returns the index for the specified vertex.
Returns the neighbors of vertex with the specified index.
Returns the degree for a specified vertex index.
Returns the adjacency matrix.
Prints the adjacency matrix.
Prints the edges.
Obtains a depth-first search tree.
Obtains a breadth-first search tree.
AbstractGraph<V>
#vertices: List<V>
#neighbors: List<List<Integer>>
#AbstractGraph(edges: int[][], vertices: V[])
#AbstractGraph(edges: List<Edge>, vertices: List<V>)
#AbstractGraph(edges: int[][], numberOfVertices: int)
#AbstractGraph(edges: List<Edge>, numberOfVertices: int)
Inner classes Tree is defined here
Vertices in the graph.
Neighbors for each vertex in the graph.
Constructs a graph with the specified edges and vertices in arrays.
Constructs a graph with the specified edges and vertices stored in lists.
Constructs a graph with the specified edges in an array and the integer vertices 1, 2, ...
Constructs a graph with the specified edges in a list and the integer vertices 1, 2, ….
UnweightedGraph<V>
+UnweightedGraph(edges: int[][], vertices: V[])
+UnweightedGraph(edges: List<Edge>, vertices: List<V>)
+UnweightedGraph(edges: List<Edge>, numberOfVertices: int)
+UnweightedGraph(edges: int[][], numberOfVertices: int)
Constructs a graph with the specified edges and vertices in arrays.
Constructs a graph with the specified edges and vertices stored in lists.
Constructs a graph with the specified edges in an array and the integer vertices 1, 2, ...
Constructs a graph with the specified edges in a list and the integer vertices 1, 2, ….

```java
public class TestGraph {
  public static void main(String[] args) {
    String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
      "Denver", "Kansas City", "Chicago", "Boston", "New York",
      "Atlanta", "Miami", "Dallas", "Houston"};

    // Edge array for graph in Figure 27.1
    int[][] edges = {
      {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}
    };

    Graph<String> graph1 =
      new UnweightedGraph<String>(edges, vertices);
    System.out.println("The number of vertices in graph1: "
      + graph1.getSize());
    System.out.println("The vertex with index 1 is "
      + graph1.getVertex(1));
    System.out.println("The index for Miami is " +
      graph1.getIndex("Miami"));
    System.out.println("The edges for graph1:");
    graph1.printEdges();
    System.out.println("Adjacency matrix for graph1:");
    graph1.printAdjacencyMatrix();

    // List of Edge objects for graph in Figure 27.3(a)
    String[] names = {"Peter", "Jane", "Mark", "Cindy", "Wendy"};
    java.util.ArrayList<AbstractGraph.Edge> edgeList
      = new java.util.ArrayList<AbstractGraph.Edge>();
    edgeList.add(new AbstractGraph.Edge(0, 2));
    edgeList.add(new AbstractGraph.Edge(1, 2));
    edgeList.add(new AbstractGraph.Edge(2, 4));
    edgeList.add(new AbstractGraph.Edge(3, 4));
    // Create a graph with 5 vertices
    Graph<String> graph2 = new UnweightedGraph<String>
      (edgeList, java.util.Arrays.asList(names));
    System.out.println("The number of vertices in graph2: "
      + graph2.getSize());
    System.out.println("The edges for graph2:");
    graph2.printEdges();
    System.out.println("\nAdjacency matrix for graph2:");
    graph2.printAdjacencyMatrix();

    for (int i = 0; i < 5; i++)
      System.out.println("vertex " + i + ": " + graph2.getVertex(i));
  }
}
```

Graph Breadth-First Traversal

Given a graph *G(V,E)* and a starting vertex *s*, perform a breadth-first traversal (BFT) of *G*. such that each reachable vertex is entered exactly once.

If all vertices are reachable, the edges traversed and the set of vertices will represent a spanning tree embedded in the graph *G*.



1) BFT suggests an iterative process (rather than a recursive one)

2) BFT vertices order of traversal can be maintained using a Queue data structure

3) The preferred representation for the graph is an adjacency matrix

4) We will need a way to keep up with which vertices have been "used" (e.g. a Boolean list)

5) Process begins by placing the starting vertex in the Queue

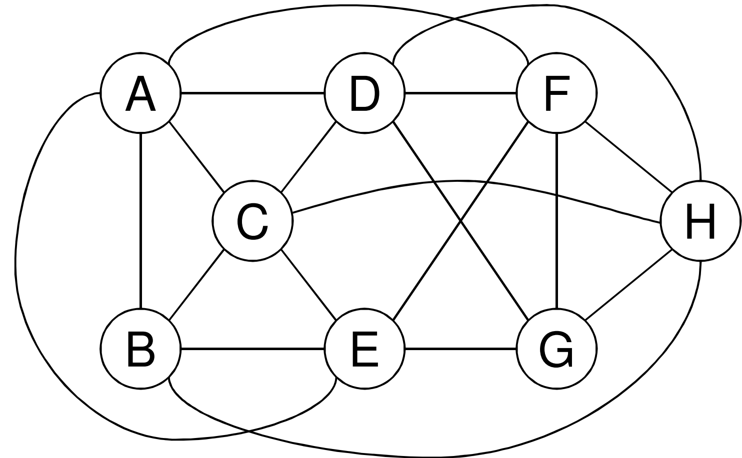
6) A vertex is taken from the Queue, every unused vertex adj to this vertex is added to the Queue
   This operation is repeated until the Queue is empty.

8) The output (answer) is returned in the form of a list of vertices in the order they entered the Queue

Graph Depth-First Traversal

Given a graph *G(V,E)* and a starting vertex *s*, perform a depth-first traversal (BFT) of *G*. such that each reachable vertex is entered exactly once.

If all vertices are reachable, the edges traversed and the set of vertices will represent a spanning tree embedded in the graph *G*.



1) DFT suggests a recursive process (rather than an iterative one)

2) DFT vertices order of traversal are maintained automatically by the recursion process (as a Stack)

3) The preferred representation for the graph is an adjacency matrix.

4) We will need a way to keep up with which vertices have been "used" (e.g. a Boolean list)

5) Process begins by passing the starting vertex to a recursive function DFT(s)

6) For the current vertex, s DFT(s) calls itself for each adjacent, unused vertex remaining.
   This operation is completed when all calls to DFT( ) are completed.

8) The output is returned as a of a list of vertices in the order they were passed to DFT( ).

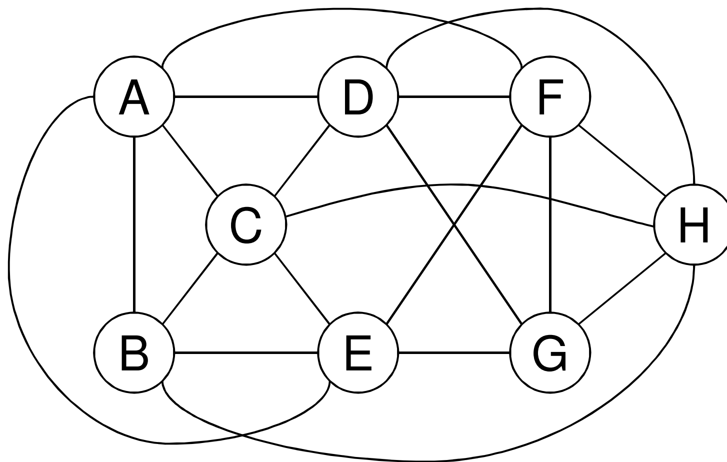
Graph Depth-First Traversal (DFT) Algorithm Implementation

text file representation

Given a graph G(V,E) and a starting node v_{start} perform a depth-first traversal. Provide a list of the nodes in the order they are traversed.

Graph is defined in a text file of the following format:

1st Line - # of nodes, N   # of edges, E
2nd Line - List of node labels (you may assume 1 char each)
3rd Line - through N+3 Line
an N x N adjacency matrix




```
8
A B C D E F G H
0 1 1 1 1 1 0 0
1 0 1 0 1 0 0 1
1 1 0 1 1 0 0 1
1 0 1 0 0 1 1 1
1 1 1 0 0 1 1 0
1 0 0 1 1 0 1 1
0 0 0 1 1 1 0 1
0 1 1 1 0 1 1 0
```

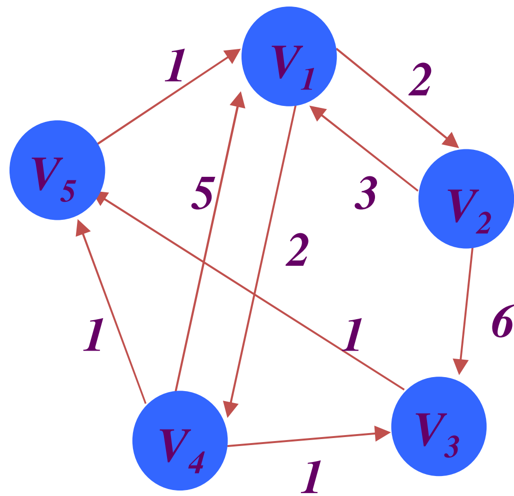
Graph Depth-First Traversal (DFT) Algorithm Implementation

pseudo-code

```
DFT( node vk )
{
    // deal with current node
    for each node, vi
    {
        if (node_avail(vi)) then
            DFT(vi)
    }
}
```

node_avail(v_i *)* is a Boolean function that returns *true* if node v_i is available for traversal, which means that v_i has not been traversed, and v_i is adjacent to v_k.

Floyd's Algorithm for Shortest Paths



	1	2	3	4	5
1	0	2	–	–	–
2	3	0	6	–	–
3	–	–	0	–	1
4	2	–	1	0	1
5	1	–	–	–	0

```
procedure floyd(W,D:matype) is
begin
  D:=W;
  for k in 1..n loop
    for i in 1..n loop
      for j in 1..n loop
        D(i,j):=min(D(i,j),D(i,k)+D(k,j));
      end loop;
    end loop;
  end loop;
end floyd;
```

Floyd's algorithm is very simple to implement. The fact that it works at all is not obvious. Be sure to work through the proof of algorithm correctness in the text.