# Ch 17. Graphs



SNU
IDB Lab.

# Bird's-Eye View (0)

- Chapter 5-8: Linear List
- Chapter 9-11: Stack & Queue
- Chapter 12-16: Tree
- Chapter 17: Graph

# Bird's eye review (1)

- Graphs
  - Used to model and solve many real-world problems

- In this chapter
  - Graph terminology
  - Different types of graphs
  - Common graph representations
  - Standard graph search methods
  - Algorithms to find a path in a graph
  - Specifying an abstract data type as an abstract class

# Table of Contents

- Definition and Application
- The ADT Graph
- Representation of Unweighted Graphs
- Representation of Weighted Graphs
- Class Implementations
- Graph Search Methods
- Application Revisited
  - Find a path in a Digraph
  - Connected Graph in a Graph
  - Component Labeling Problem in a Graph
  - Spanning Tree in a Graph

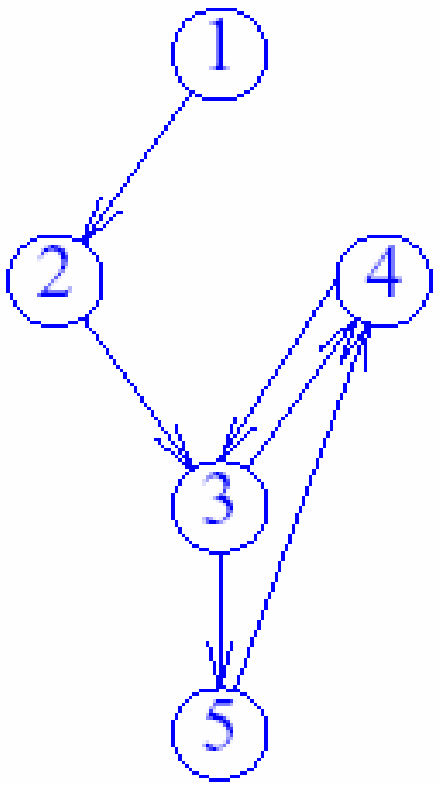
# Graph Definition (1)

- Graph G = (V, E)
  - Finite set V (=vertices, nodes, points)
  - Finite set E (=edges, arcs, lines)

- Directed edge: orientation
- Undirected edge: no orientation

- Vertices i and j are **adjacent** vertices iff (i,j) is an edge in the graph
- Edge(i,j) is **incident** on the vertices i and j

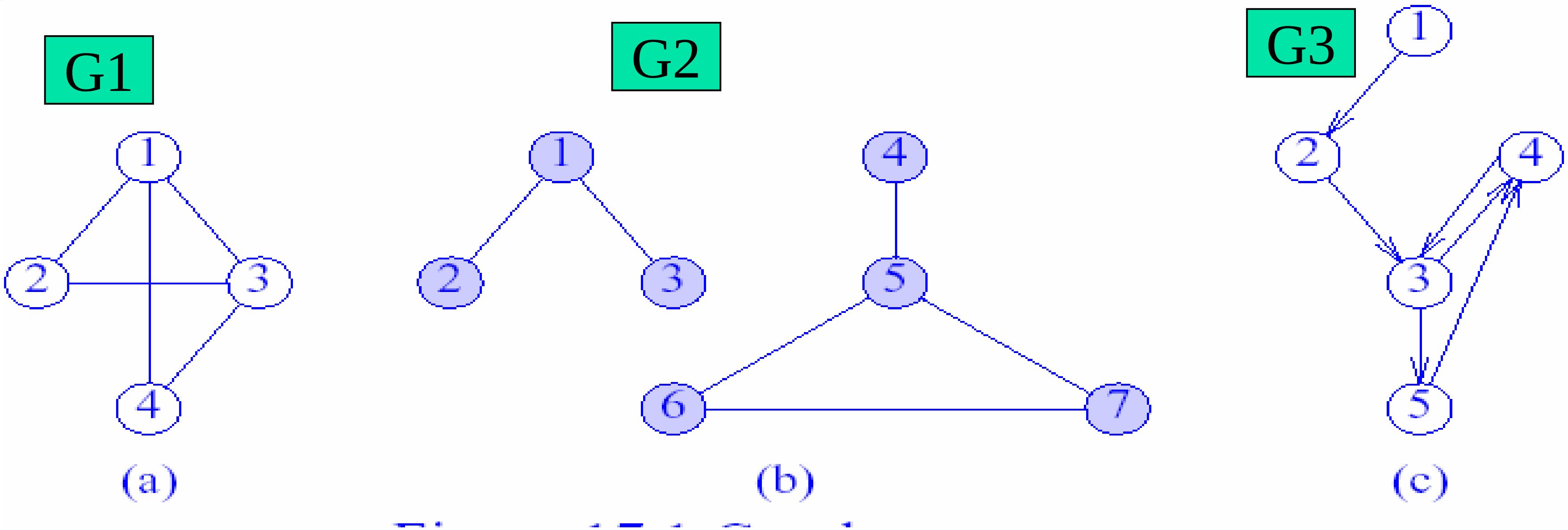
# Graph Definition (2)



(c)

- Vertex 2 is adjacent from vertex 1, while vertex 1 is adjacent to vertex 2
- Edge(1,2) is incident from vertex 1 and incident to vertex 2
- Vertex 4 is both adjacent to and from vertex 3
- Edge(3,4) is incident from vertex 3 and incident to vertex 4

# Graph Definition (3)




- $G_1 = (V_1, E_1)$:  $V_1 = \{1, 2, 3, 4\}$, $E_1 = \{(1,2), (1,3), (2,3), (1,4), (3,4)\}$
- $G_2 = (V_2, E_2)$:
  $V_2 = \{1, 2, 3, 4, 5, 6, 7\}$, $E_2 = \{(1,2), (1,3), (4,5), (5,6), (5,7),(6,7)\}$
- $G_3 = (V_3, E_3)$:
  $V_3 = \{1, 2, 3, 4, 5\}$,      $E_3 = \{(1,2), (2,3), (3,4), (4,3), (3,5), (5,4)\}$

# Graph Definition (4)

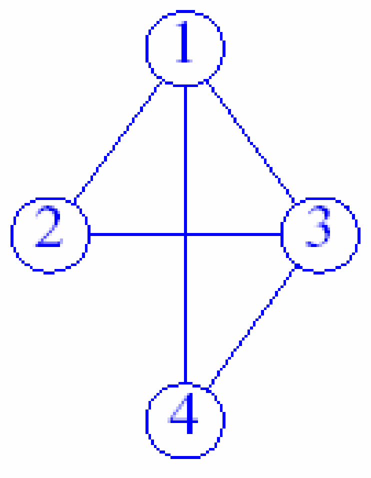
- Directed Graph
  - All the edge are directed (=digraph)
  - Self-edges (loop) are not allowed: (i, i), (j, j)
  - Directed Acyclic Graph (DAG): No cycle in digraph

- Weights can be assigned to edges
  - weighted undirected graph
  - weighted directed graph (digraph)

- Graph == Network (synonym)

# Graph Definition (5)

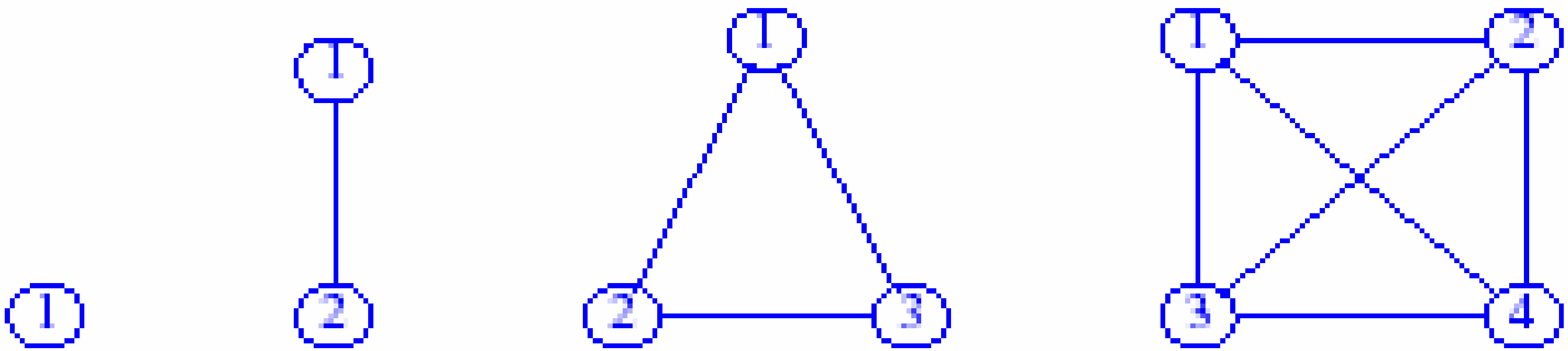
- Definition: Degree $d_i$ of vertex i of an undirected graph is the number of edges incident on vertex i
- Example:
  - $d_1 = 3$
  - $d_2 = 2$
  - $d_3 = 3$
  - $d_4 = 2$



(a)

# Edges in Undirected Graph

- Let G=(V,E) be an undirected graph and
  Let n=|V| and e=|E|
  A. $\sum_{i=1}^{n} d_i = 2e$
  B. $0 \leq e \leq n(n-1)/2$
- A complete undirected graph has n*(n-1) / 2 edges



(a) $K_1$ (b) $K_2$ (c) $K_3$ (d) $K_4$

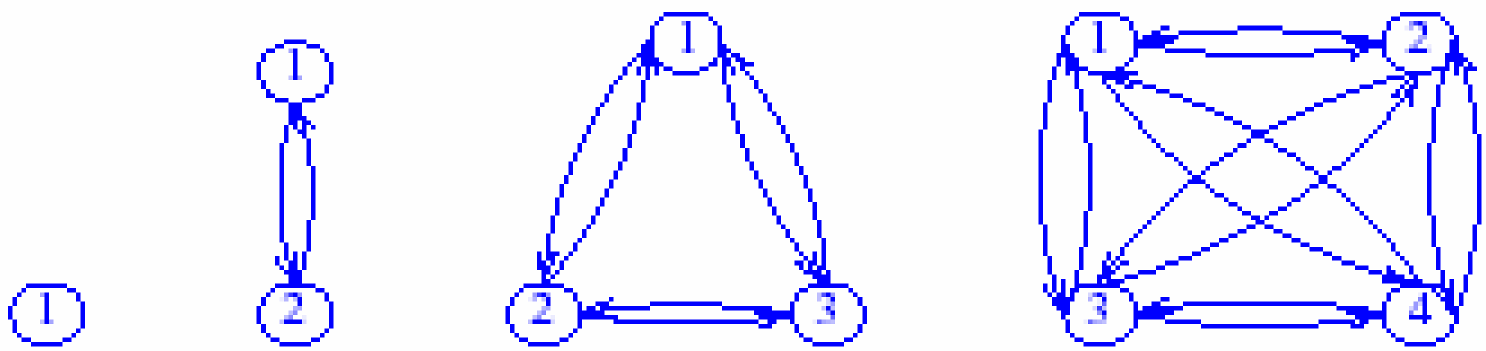
# Edges in Directed Graph

- Let G=(V,E) be a directed graph

  Let n = |V| and e = |E|

  A. $0 \le e \le n(n-1)$
  B. $\sum_{i=1}^{n} d_i^{in} = \sum_{i=1}^{n} d_i^{out} = e$

- A complete directed graph has n* (n-1) edges




(a) $K_1$ (b) $K_2$ (c) $K_3$ (d) $K_4$

# Graph Terms

- G=(V,E)
- A graph G is a connected graph IFF there is a path between every pair of vertices
- A graph H is a subgraph of another graph G IFF vertex and edge sets of H are subsets of those of graph G respectively
- Simple path is a path in which all vertices, except possibly the first and last, are different
  - A cycle is a simple path with same start and end vertex in a graph
- A spanning tree is a tree and a subgraph of G that contains all the vertices of G

# Graph Application: Path Problems

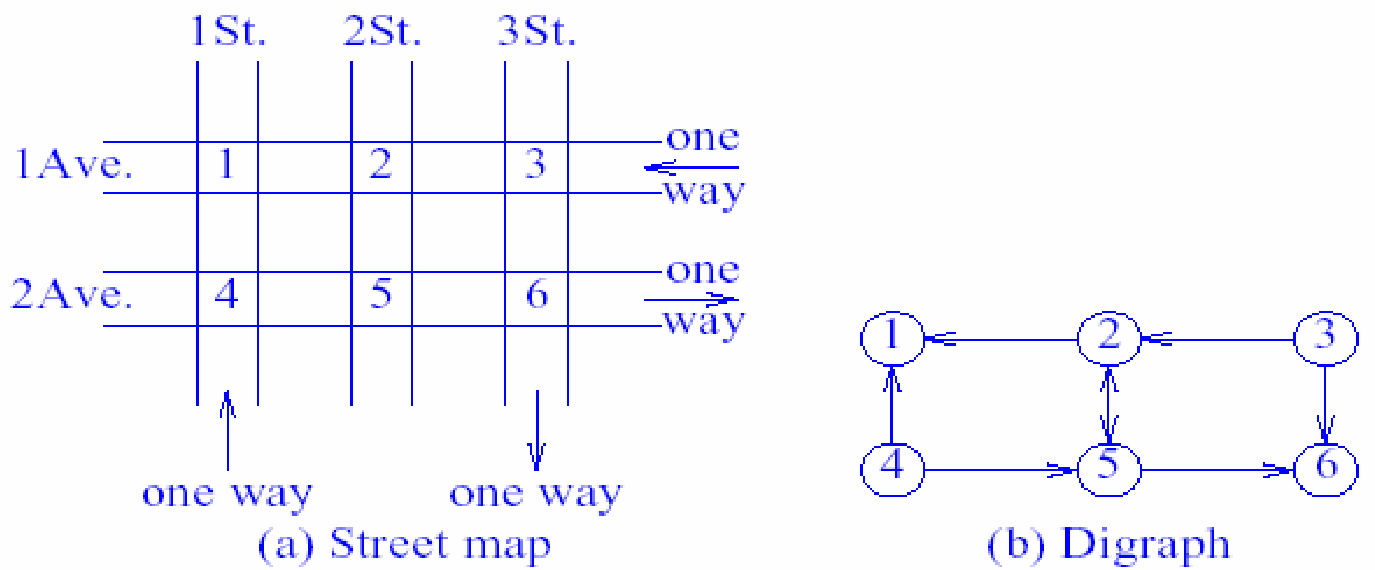
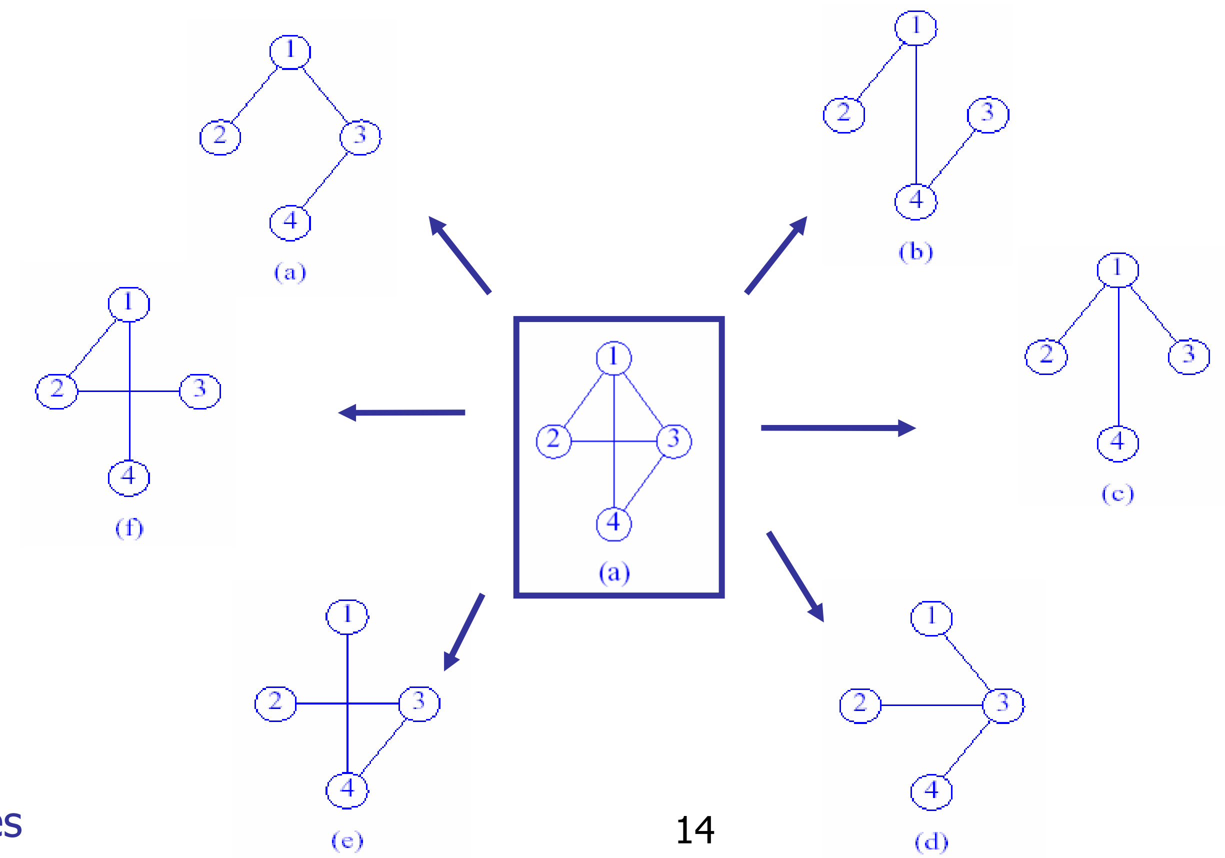


Figure 17.2 Street map and corresponding digraph

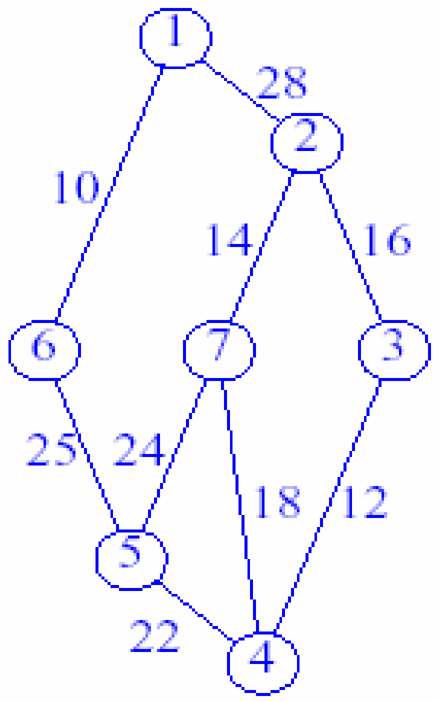
- Simple path: Path in which all vertices, except possibly the first and last, are different ➔ 5,2,1 (yes)   2,5,2,1 (no)
- Length: Each edge in a graph may have an associated length

# Graph Application: Spanning Trees

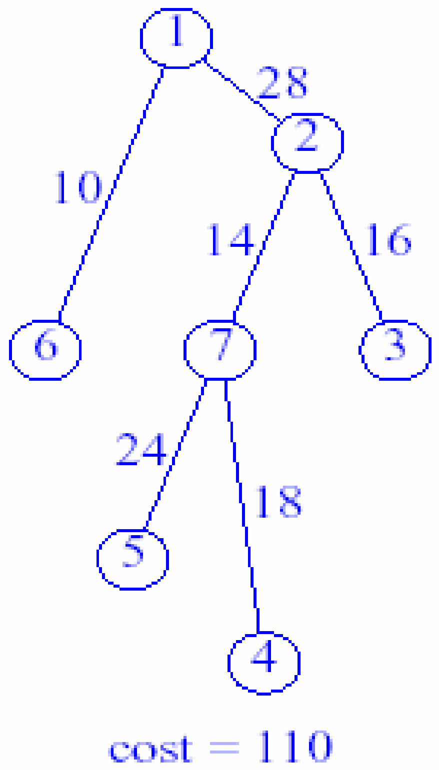
- A spanning tree is a tree and a subgraph of G that contains all the vertices of G



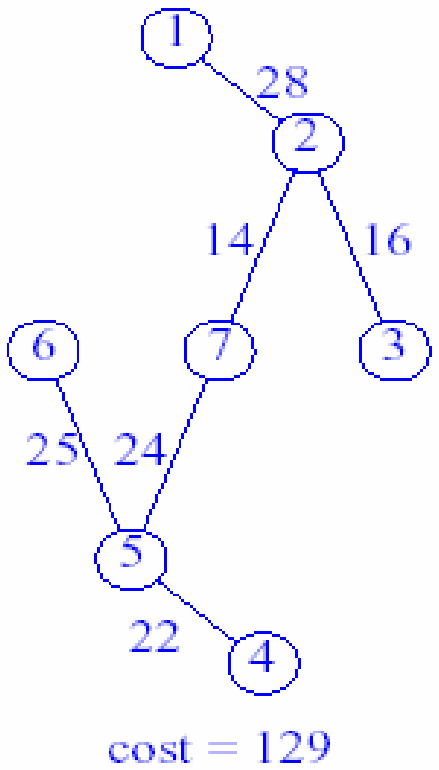
# Graph Application:
## Weighted Graph and its Spanning Trees




(a) Graph




(b) Spanning tree




(c) Spanning tree

# Graph Application:
# Bipartite Graphs

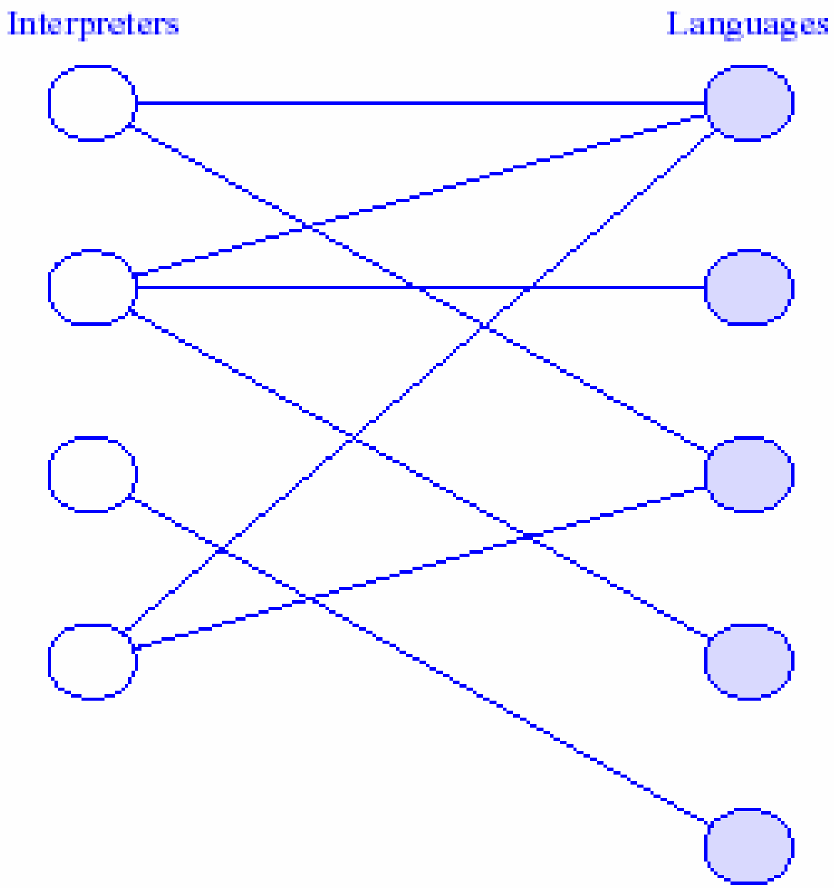



Figure 17.5 Interpreters and languages

Bipartite graphs

- Partition the vertex set into two subsets, A (the interpreter vertices) and B (the language vertices), so that every edge has one endpoint in A and the other in B.

# Table of Contents

# The ADT Graph

AbstractDataType Graph {

instances

a set V of vertices and a set E of edges

operations

vertices(), edges(), existsEdge(i, j),

putEdge(i, j), removeEdge(i, j), degree(i),

inDegree(i), outDegree(i)

}

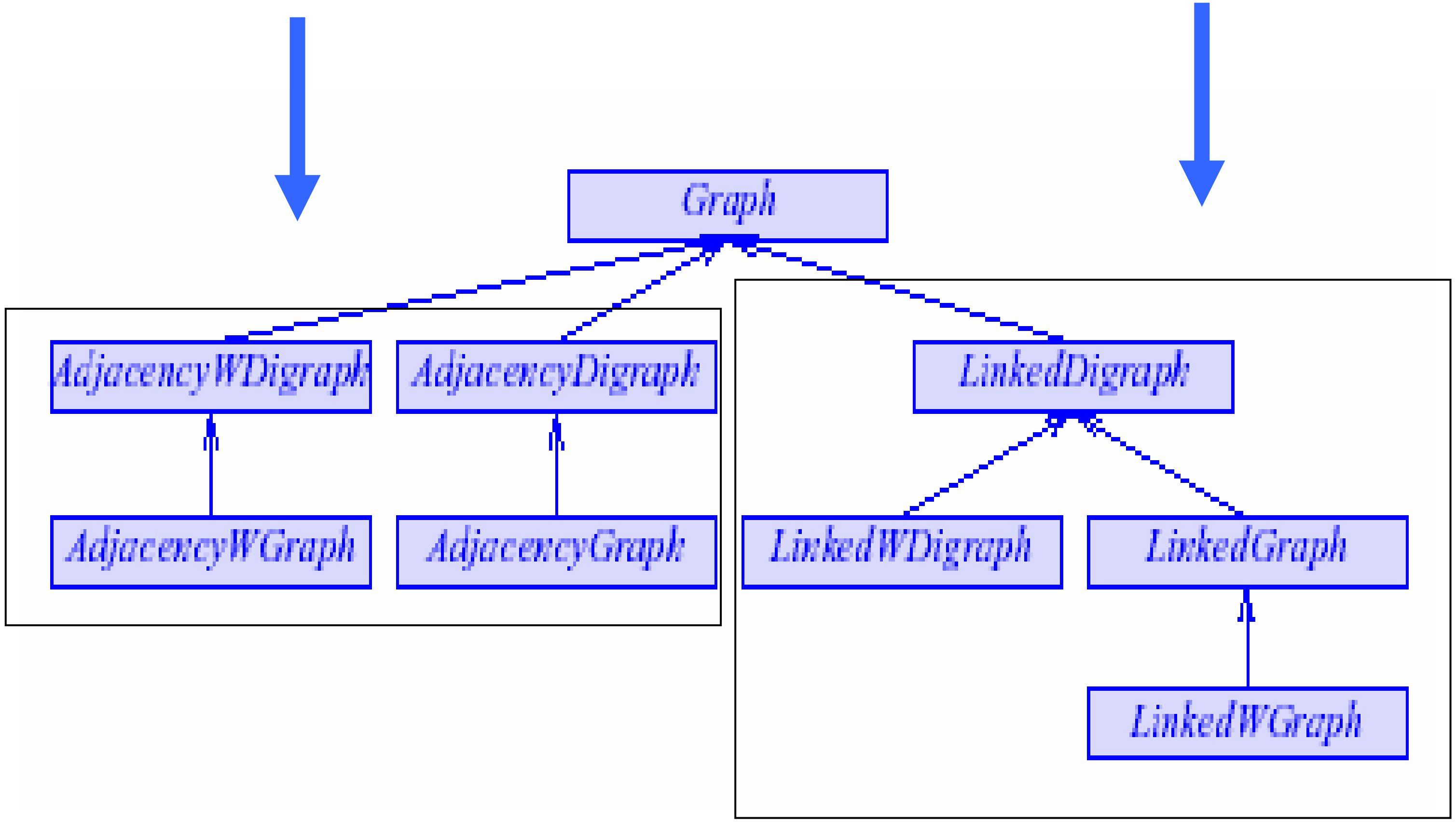
- ADT graph refers to all varieties of graphs whether directed, undirected, weighted, or unweigted

# The abstract class Graph

```
public abstract class Graph {

    //ADT method
    public abstract int         vertices();
    public abstract int         edges();
    public abstract boolean existEdge(int i, int j);
    public abstract void      putEdge(Object theEdge);
    public abstract void      removeEdge(int i, int j);
    public abstract int        degree(int i);
    public abstract int        inDegree(int i);
    public abstract int        outDegree(int i);

    //create an iterator for vertex i
    public abstract iterator  iterator(int i);
}
```

# Class derivation hierarchy for Graph

* Left side: array-based   vs   Right-side: linked-based



# Table of Contents

- Definition and Applications
- The ADT Graph
- *Representation of Unweighted Graphs*
- Representation of Weighted Graphs
- Class Implementations
- Graph Search Methods
- Application Revisited
  - Find a path in a Digraph
  - Connected Graph in a Graph
  - Component Labeling Problem in a Graph
  - Spanning Tree in a Graph

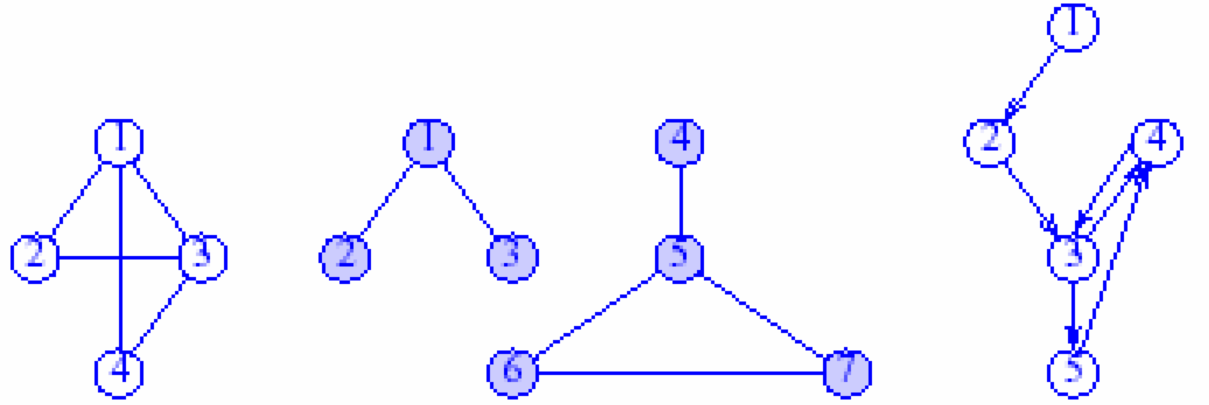
# Adjacency Matrix for Unweighted Graph (1)

- The adjacency matrix of an n-vertex graph G=(V,E) is an n×n matrix A
  - Each element of A is either 0 or 1
  - V={1, 2, …, n}

- G is an undirected Graph
  - $A(i, j) = \begin{cases} 1 & \text{If } (i, j) \in E \text{ or } (j, i) \in E \\ 0 & \text{otherwise} \end{cases}$

- G is a directed graph
  - $A(i, j) = \begin{cases} 1 & \text{If } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$

# Adjacency Matrix for Unweighted Graph (2)




(a) (b) (c)

$$
\begin{array}{c|cccc|}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 1 & 1 & 1 \\
2 & 1 & 0 & 1 & 0 \\
3 & 1 & 1 & 0 & 1 \\
4 & 1 & 0 & 1 & 0 \\
\hline
\end{array}
$$

(a)

$$
\begin{array}{c|ccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
6 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
7 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
\hline
\end{array}
$$

(b)

$$
\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 0 \\
2 & 0 & 0 & 1 & 0 & 0 \\
3 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 0 & 1 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 \\
\hline
\end{array}
$$

(c)

# Adjacency Matrix for Unweighted Graph (3)

- An n×n adjacency matrix can be mapped into array
  - (n+1)×(n+1) array: $(n+1)^2 = n^2 + 2n + 1$ bits
  - n×n array: $n^2$ bits
  - The diagonal may be eliminated
    - (n - 1) × n matrix: $(n^2-n)/2$ bits
    - But, potential mismatch in coding

- Array-based AM is simple
  - But you have to check every item in a row if you want to do something with adjacent nodes
  - 0(n) to determine the set of vertices adjacent to or from any given vertex

- Later we use the irregular array instead!

# Adjacency Matrix for Unweighted Graph (4)

$$
\begin{array}{c|c}
 & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} &
\left[\begin{array}{cccc}
0 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 \\
1 & 0 & 1 & 0
\end{array}\right]
\end{array}
\qquad
\begin{array}{c|c}
 & \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} &
\left[\begin{array}{ccccccc}
0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 0
\end{array}\right]
\end{array}
\qquad
\begin{array}{c|c}
 & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} &
\left[\begin{array}{ccccc}
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0
\end{array}\right]
\end{array}
$$

(a) (b) (c)

$$
\begin{array}{c|c}
 & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \end{array} &
\left[\begin{array}{cccc}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 \\
1 & 0 & 1 & 1
\end{array}\right]
\end{array}
\qquad
\begin{array}{c|c}
 & \begin{array}{ccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} &
\left[\begin{array}{ccccccc}
1 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1
\end{array}\right]
\end{array}
\qquad
\begin{array}{c|c}
 & \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} &
\left[\begin{array}{ccccc}
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 0
\end{array}\right]
\end{array}
$$

(a) (b) (c)

Figure 17.10 Adjacency matrices of Figure 17.9 with diagonals eliminated

# Linked Adjacency Lists for Unweighted Graph (1)

- An adjacency list for vertex i is a linear list that includes all vertices adjacent from the vertex i
  - aList[i].firstNode pointer is pointing to the first node
  - If x points to a node in the chain aList[i], (i, x.element.vertex) is an edge of the graph
- Space requirement for n vertex graph

  4*(n+1) : for the array aList

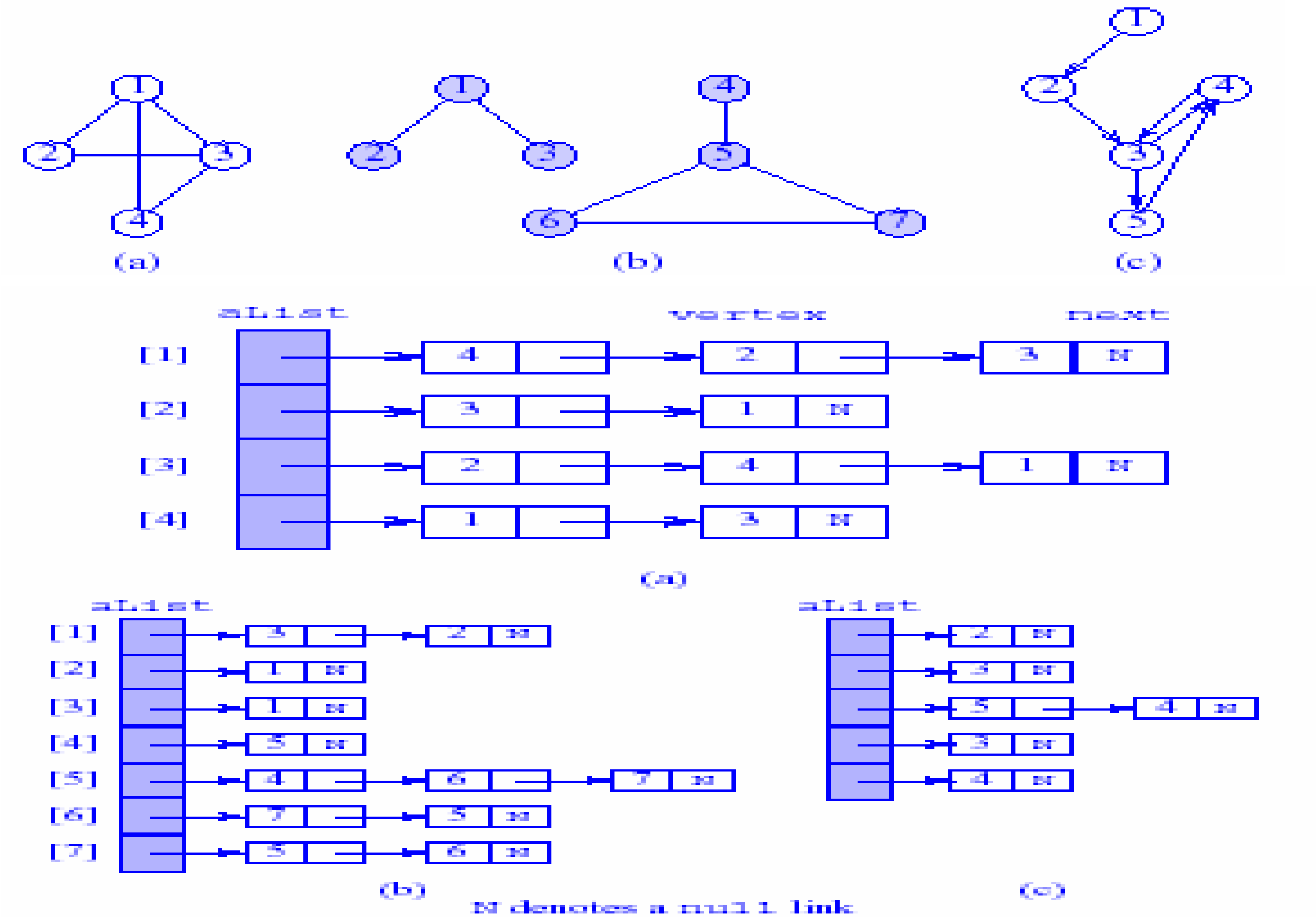
  + 4*n : for the n firstNode pointers

  + 4*3*m : for m chain nodes (next, element, vertex of element))

  ➔ 4(2n+3m+1)

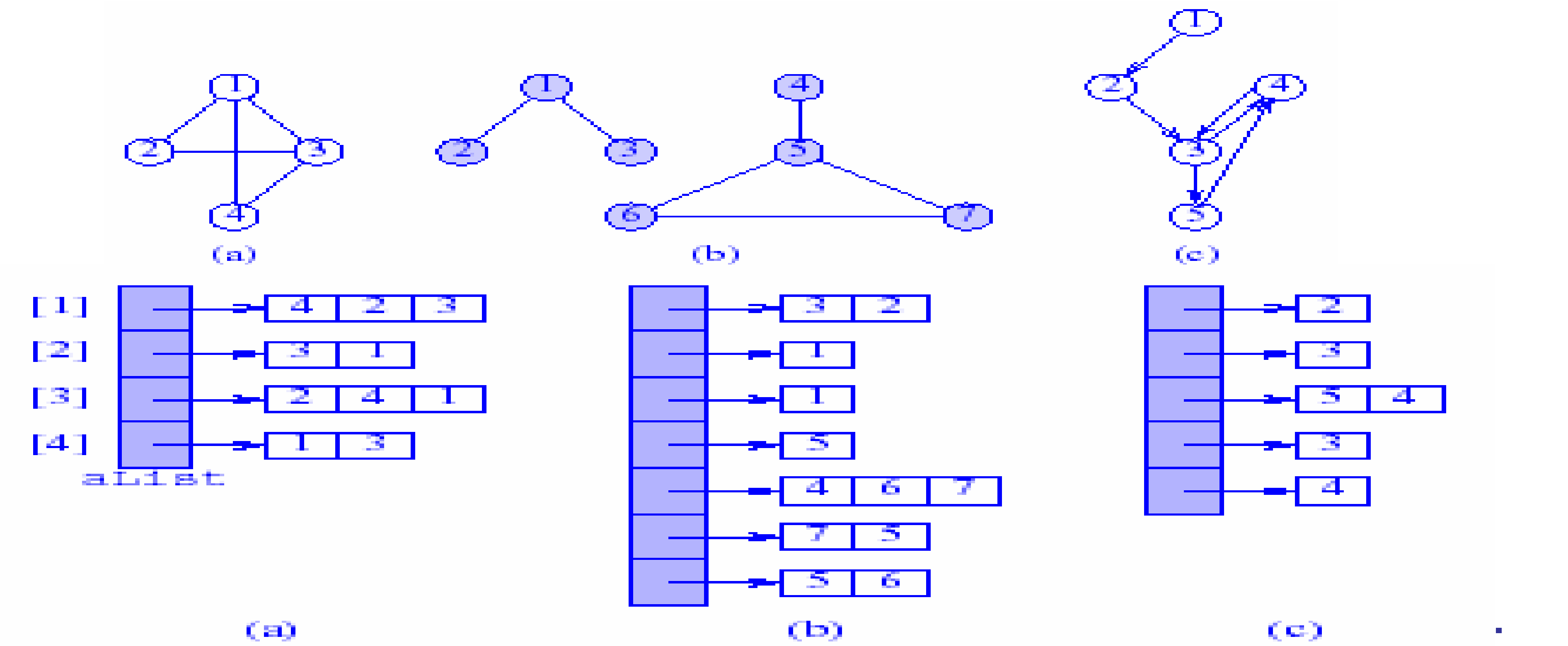
  - Undirected graph : m = 2e & Digraph graph : m = e where e is the number of edges

# Linked Adjacency Lists for Unweighted Graph (2)



# 2D Irregular Array for Adjacency Lists of Unweighted Graph

- 2D irregular array-based linear list representation rather than a chain
- Space requirement
  - 4*m bytes less than that of linked adjacency lists because we do not have m "next" pointers



# Table of Contents

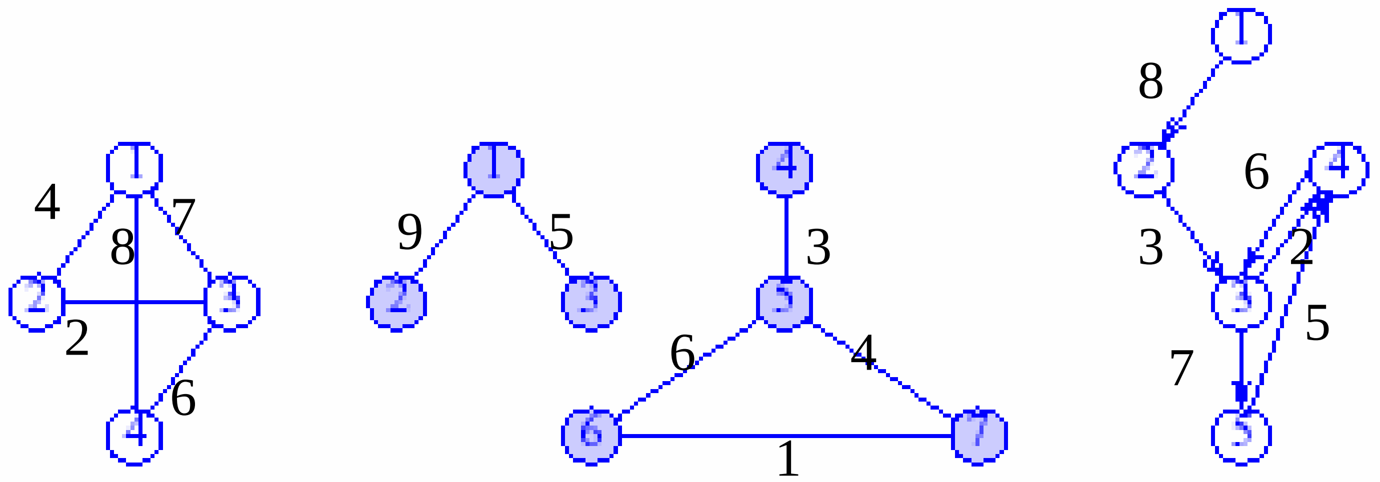
# Representation of Weighted Graphs

- Cost-adjacency-matrix
  - Use a matrix C
  - A(i, j) is 1 ➔ C(i, j) is cost (weight)
  - A(i, j) is 0 ➔ C(i, j) is null

- Linked Adjacency-list representation
  - Chain (see the next page)
  - Elements have the two fields vertex and weight

- Array Adjacency-list representation
  - Easily derived from that of unweighted graph
  - But determining the adjacent nodes is 0(n)

# Cost-adjacency matrices for Weighted Graph



(a) (b) (c)

(a)

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | - | 4 | 7 | 8 |
| 2 | 4 | - | 2 | - |
| 3 | 7 | 2 | - | 6 |
| 4 | 8 | - | 6 | - |

(b)

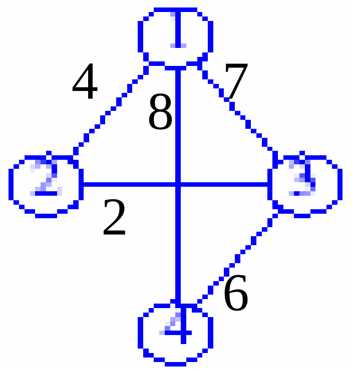
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | - | 9 | 5 | - | - | - | - |
| 2 | 9 | - | - | - | - | - | - |
| 3 | 5 | - | - | - | - | - | - |
| 4 | - | - | - | - | 3 | - | - |
| 5 | - | - | - | 3 | - | 6 | 4 |
| 6 | - | - | - | - | 6 | - | 1 |
| 7 | - | - | - | - | 4 | 1 | - |

(c)

| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | - | 8 | - | - | - |
| 2 | - | - | 3 | - | - |
| 3 | - | - | - | 2 | 7 |
| 4 | - | - | 6 | - | - |
| 5 | - | - | - | 5 | - |

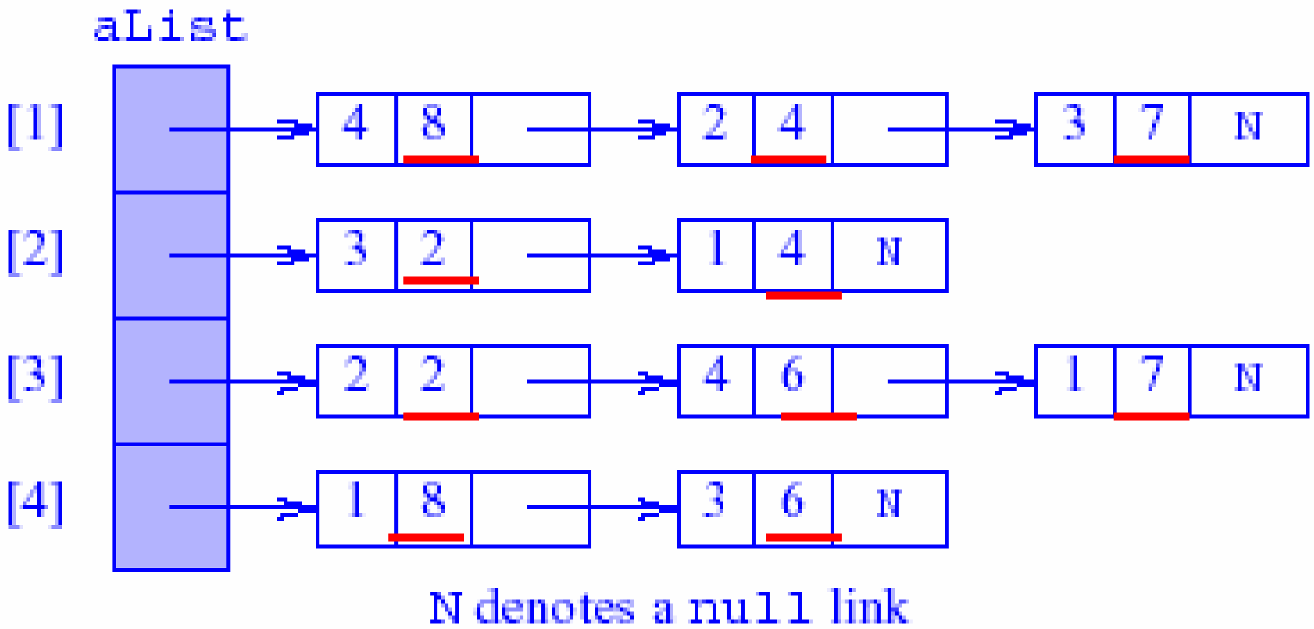
- denotes a null value

# Linked adjacency list for weighted graph




$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & - & 4 & 7 & 8 \\ 2 & 4 & - & 2 & - \\ 3 & 7 & 2 & - & 6 \\ 4 & 8 & - & 6 & - \end{array}$$

(a)



# Table of Contents

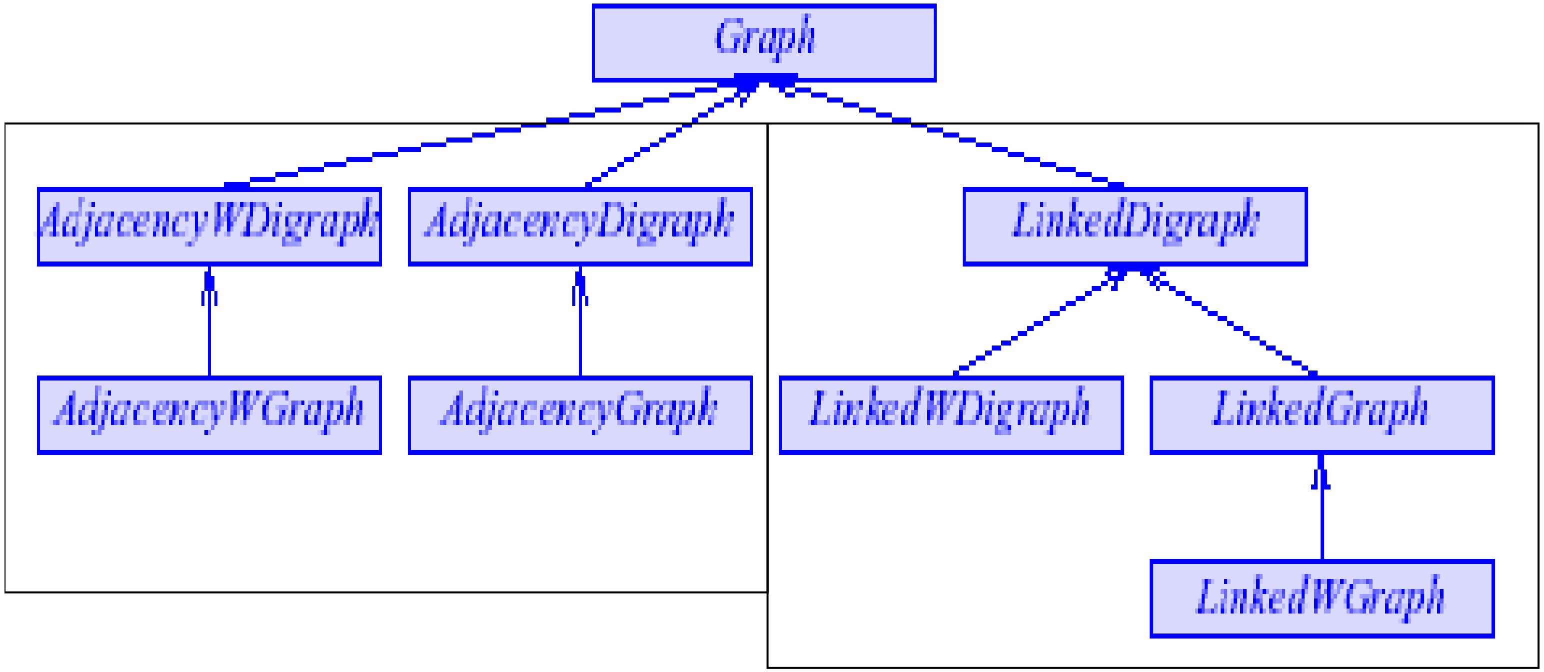
- Definition and Applications
- The ADT Graph
- Representation of Unweighted Graphs
- Representation of Weighted Graphs
- *Class Implementations*
- Graph Search Methods
- Application Revisited
  - Find a path in a Digraph
  - Connected Graph in a Graph
  - Component Labeling Problem in a Graph
  - Spanning Tree in a Graph

# The Different Classes for Graph

- Four graph types
  - unweighted undirected graph
  - weighted undirected graph
  - unweighted directed graph
  - weighted directed graph
- Three representations
  - matrix
  - Array
  - linked list
- Several pairs of 4 graph types have “IsA” relationship

# Class derivation hierarchy for Graph

* Left side: array-based vs Right-side: linked-based



# Array-based Adjacency-Matrix Classes for Graph

- Weighted-edge classes: 2D array of type Object
  - The class AdjacencyWDigraph & AdjacencyWGraph
- Unweighted-edge classes: 2D array of type boolean
  - The class AdjacencyDigraph & AdjacencyGraph

- We describe only AdjacencyWDigraph and AdjacencyWGraph here

# Remember: The abstract class Graph

```
public abstract class Graph
{
    //ADT method
    public abstract int vertices();
    public abstract int edges();
    public abstract boolean existEdge(int i, int j);
    public abstract void putEdge(Object theEdge);
    public abstract void removeEdge(int i, int j);
    public abstract int degree(int i);
    public abstract int inDegree(int i);
    public abstract int outDegree(int i);

    //create an iterator for vertex i
    public abstract iterator iterator(int i);
}
```

# The Class AdjacencyWDigraph (1)

```
public class AdjacencyWDigraph extends Graph {
   // data members
   int n;              // number of vertices
   int e;              // number of edges
   Object [][] a;   // adjacency array

   // constructors
   public AdjacencyWDigraph(int theVertices) {
      // validate theVertices
      if (theVertices < 0)
         throw new IllegalArgumentException ("no of vertices must be >= 0");
      n = theVertices;
      a = new Object [n + 1] [n + 1];
      // default values are e = 0 and a[i][j] = null
   }
  // default is a 0 vertex graph
   public AdjacencyWDigraph()    { this(0); }
```

# The Class AdjacencyWDigraph (2)

```
/** put edge e into the digraph; if already there, update its weight e.weight
  * @throws IllegalArgumentException when theEdge is invalid */
  public void putEdge(Object theEdge) {
     WeightedEdge edge =  (WeightedEdge) theEdge;
     int v1 = edge.vertex1;
     int v2 = edge.vertex2;
     if (v1 < 1 || v2 < 1 || v1 > n || v2 > n || v1 == v2)  throw new
            IllegalArgumentException ("(" + v1 + "," + v2 + ") is not a permissible edge");
     if (a[v1][v2] == null) e++;  // new edge
     a[v1][v2] = edge.weight;
  }

/** remove the edge (i,j) */
  public void removeEdge (int i, int j) {
     if (i >= 1 && j >= 1 && i <= n && j <= n && a[i][j] != null)  {
        a[i][j] = null;
        e--;  }
  }
```

# The Class AdjacencyWDigraph (3)

```
/** this method is undefined for directed graphs */
  public int degree (int i) {   throw new NoSuchMethodError();}

/** @return out-degree of vertex i
    * @throws IllegalArgumentException when i is not a valid vertex */
 public int outDegree(int i)  {
     if (i < 1 || i > n)
        throw new IllegalArgumentException("no vertex " + i);
     // count out edges from vertex i
     int sum = 0;
     for (int j = 1; j <= n; j++)
        if (a[i][j] != null)    sum++;
     return sum;
  }
```

# The Class AdjacencyWDigraph (4)

```
/** create and return an iterator for vertex i
   * @throws IllegalArgumentException when i is an invalid vertex */
 public Iterator iterator(int i)  {
      if (i < 1 || i > n)  throw new IllegalArgumentException("no vertex " + i);
      return new VertexIterator(i);
 }

private class VertexIterator implements Iterator {
      // data members
      private int v;                    // the vertex being iterated
      private int nextVertex;
      // constructor
      public VertexIterator(int i) {
          v = i;  // find first adjacent vertex
          for (int j = 1; j <= n; j++)
              if (a[v][j] != null)  {
                   nextVertex = j;
                   return;  }
        // no edge out of vertex i
        nextVertex = n + 1;
      }
```

# The Class AdjacencyWDigraph (5)

```
// iterator methods
  public boolean hasNext()  {  return nextVertex <= n;  } //true if there is next vertex
 /** @return next adjacent vertex and edge weight */
  public Object next() {
        if (nextVertex <= n) {
           int u = nextVertex; // find next adjacent vertex
           for (int j = u + 1; j <= n; j++)
               if (a[v][j] != null) {
                  nextVertex = j;
                  return new WeightedEdgeNode(u, a[v][u]); }
         // no next adjacent vertex for v
          nextVertex = n + 1;
          return new WeightedEdgeNode(u, a[v][u]);
       } else throw new NoSuchElementException("no next vertex");
    } // end of next()

  public void remove() { throw new UnsupportedOperationException();`} //unsupported
 }  // end of the class VertexIterator

} // end of the class AdjacencyWDigraph
```

# The Class AdjacencyWGraph (for undirected graph)

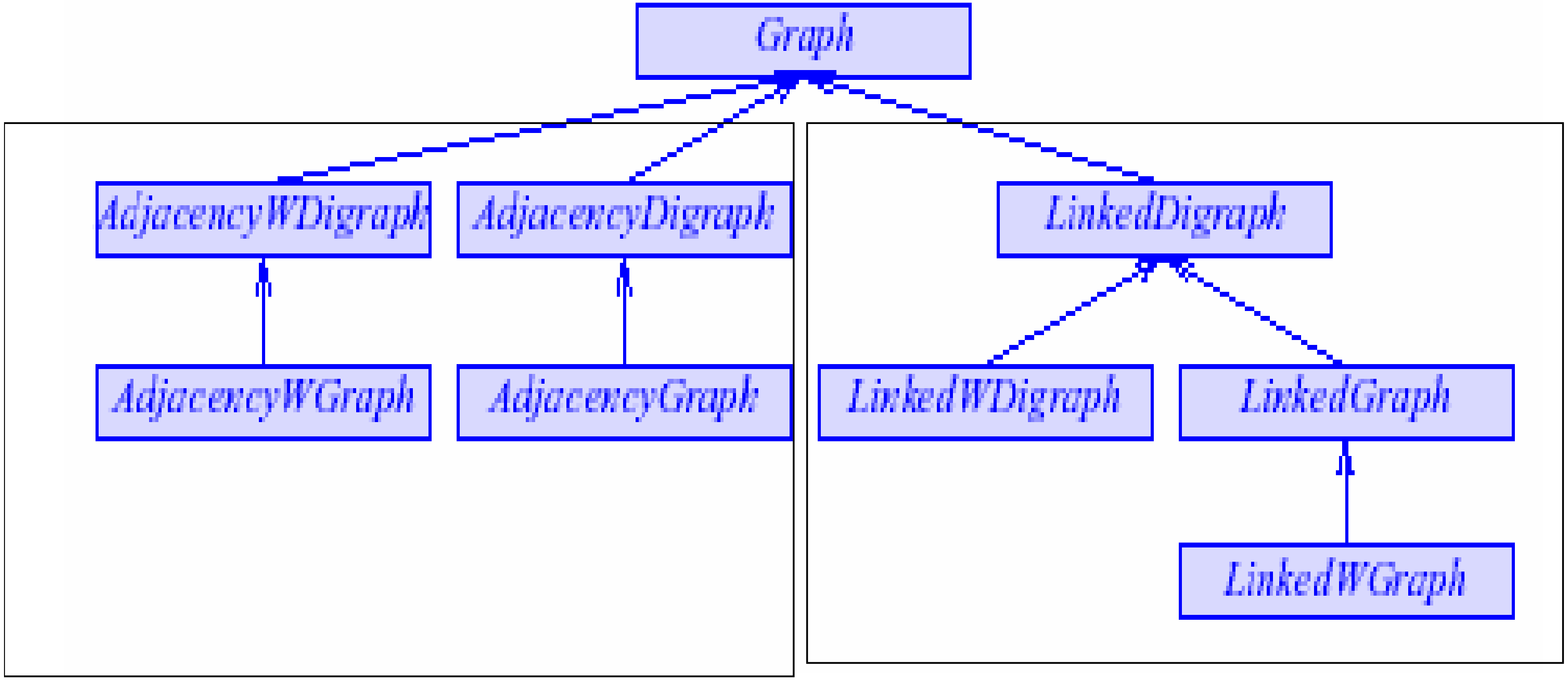
```
public class AdjacencyWGraph extends AdjacencyWDigraph {
   public void removeEdge(int i, int j) { /** remove the edge (i,j) */
      if (i >= 1 && j >= 1 && i <= n && j <= n && a[i][j] != null)  {
            a[i][j] = null;
            a[j][i] = null;
            e--;  }
   }
   /** put edge e into the graph; if already there, update its weight to e.weight
     * @throws IllegalArgumentException when theEdge cannot be an edge*/
   public void putEdge(Object theEdge) {
      WeightedEdge edge =  (WeightedEdge) theEdge;
      int v1 = edge.vertex1;
      int v2 = edge.vertex2;
      if (v1 < 1 || v2 < 1 || v1 > n || v2 > n || v1 == v2) throw new
             IllegalArgumentException ("(" + v1 + "," + v2 + ") is not a permissible edge");
      if (a[v1][v2] == null)   e++; // new edge
      a[v1][v2] = edge.weight;
      a[v2][v1] = edge.weight;
   }
}
```

# Linked-list classes for Graph

- Linked-list representation of a graph
  - each object is represented as an array element of chains
- In array-based graph ➔ removeEdge()
- Here, removeElement(vertex)
  - first search the chain
  - If matching element is found ➔ delete it from the chain
- Extension to the class Chain that includes the new methods is called the class GraphChain
- The class LinkedDigraph is a subclass of the class Graph

# Class derivation hierarchy for Graph

* Left side: array-based vs Right-side: linked-based



# Remember: The abstract class Graph

```
public abstract class Graph {
    //ADT method
    public abstract int vertices();
    public abstract int edges();
    public abstract boolean existEdge(int i, int j);
    public abstract void putEdge(Object theEdge);
    public abstract void removeEdge(int i, int j);
    public abstract int degree(int i);
    public abstract int inDegree(int i);
    public abstract int outDegree(int i);

    //create an iterator for vertex i
    public abstract iterator iterator(int i);
}
```

# The class LinkedDigraph (1)

```
public class LinkedDigraph extends Graph {
   // data members
   int n;                  // number of vertices
   int e;                  // number of edges
   GraphChain [] aList;    // adjacency lists

   // constructors
   public LinkedDigraph(int theVertices) {
      // validate theVertices
      if (theVertices < 0)
      throw new IllegalArgumentException ("number of vertices must be >= 0");
      n = theVertices;
      aList = new GraphChain [n + 1];
      for (int i = 1; i <= n; i++)    aList[i] = new GraphChain();
       // default value of e is 0
     }

    // default is a 0-vertex graph
     public LinkedDigraph()   { this(0); }
```

# The class LinkedDigraph (2)

```
// Graph methods
public int vertices()  {  return n;  }        /** @return number of vertices */
public int edges()     {  return e;  }        /** @return number of edges */
public boolean existsEdge(int i, int j) {  /** @return true iff (i,j) is an edge */
  if (i < 1 || j < 1 || i > n || j > n
        || aList[i].indexOf(new EdgeNode(j)) == -1)   return false;
    else   return true;
  }

// put theEdge into the digraph & throws IllegalArgumentException when theEdge is invalid
public void putEdge (Object theEdge) {
    Edge edge =  (Edge) theEdge;
    int v1 = edge.vertex1;
    int v2 = edge.vertex2;
    if (v1 < 1 || v2 < 1 || v1 > n || v2 > n || v1 == v2) throw new
          illegalArgumentException ("("+ v1 + "," + v2 + ") is not a permissible edge");
    if (aList[v1].indexOf(new EdgeNode(v2)) == -1) { // new edge
        aList[v1].add(0, new EdgeNode(v2)); // put v2 at front of chain aList[v1]
        e++; }
  }
```

# The class LinkedDigraph (3)

```
/** remove the edge (i,j) */
public void removeEdge(int i, int j)  {
      if (i >= 1 && j >= 1 && i <= n && j <= n) {
         Object v = aList[i].removeElement(j);
         if (v != null)     e--;   // edge (i,j) did exist
       }
   }

/** @return in-degree of vertex i
   * @throws IllegalArgumentException when i is an invalid vertex */
 public int inDegree(int i) {
      if (i < 1 || i > n)  throw new IllegalArgumentException("no vertex " + i);
      // count in edges at vertex i
      int sum = 0;
      for (int j = 1; j <= n; j++)
         if (aList[j].indexOf(new EdgeNode(i)) != -1)    sum++;
       return sum;
   }

} // end of the class LinkedDigraph
```

# Complexity of LinkedDGraph

- First Constructor: $O(n)$
- Second Constructor: $\Theta(1)$
- ExistsEdge(i, j): $O(d_i^{out})$
- Put(): $O(d_{v1}^{out})$
- removeEdge(): $O(d_i^{out})$
- outDegree(): $\Theta(1)$
- inDegree(): $O(n+e)$

# Table of Contents

- Definition and Applications
- The ADT Graph
- Representation of Unweighted Graphs
- Representation of Weighted Graphs
- Class Implementations
- *Graph Search Methods*
- Application Revisited
  - Find a path in a Digraph
  - Connected Graph in a Graph
  - Component Labeling Problem in a Graph
  - Spanning Tree in a Graph

# Graph Search Methods

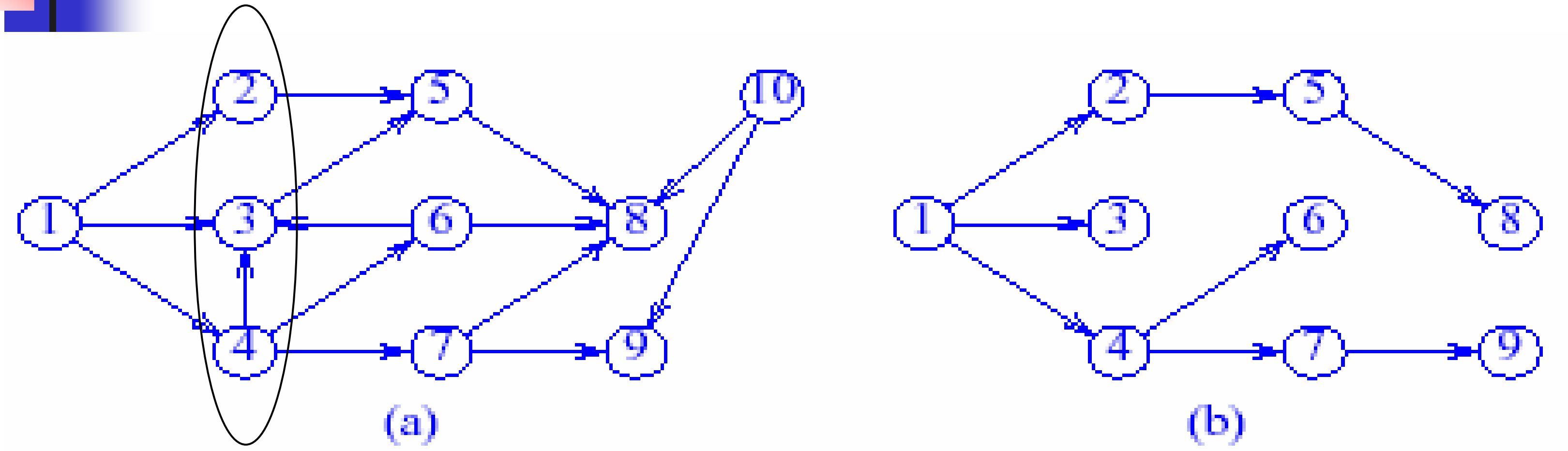
- Two standard ways
  - Breadth-first search (BFS)
  - Depth-first search (DFS)

- The DFS method is used more frequently to obtain efficient graph algorithms than the BFS method

- Here BFS & DFS examples are on directed graphs, but graph types do no matter

- Search a node ➔ Visit a node ➔ Put a label into a node

# Breadth-First Search (1)




- To determine all the vertices reachable from vertex 1
  - {1} first determine
  - {2,3,4} is set of vertices adjacent from 1
  - {5,6,7} is set of vertices adjacent from {2,3,4}
  - {8,9} is set of vertices adjacent from {5,6,7}
  - {10} is set of vertices adjacent from {8,9}
  - {1,2,3,4,5,6,7,8,9} is the set of vertices reachable from vertex 1
- We need QUEUE!

# Breadth-First Search (2)

- For finding reachable vertices from a given vertex V
  - BFS can be implemented using a queue
  - BFS ≒ Level-order traversal of a binary tree
- The pseudo-code labels all vertices that are reachable from v

```
breadthFirstSearch(v) {
    Label vertex v as reached
    Initialize Q to be a queue with only v in it
    while (Q is not empty) {
        Delete a vertex w from the queue;
        Let u be a vertex (if any) adjacent from w;
        while (u) {
            if (u has not been labeled) {
                Add u to the queue;
                Label u as reached; }
        u = next vertex that is adjacent from w
        }
    }
}
```

# Implementations of BFS

- BSF can be performed
  - independently of graph types
    - undirected graph, digraph, weighted undirected graph, or weighted digraph
  - independently of the particular representation
- BFS codes
  - Graph.java, AdjacenyDigraph.java, LinkedDigraph.java

- Next program assumes
  - reach[i]=0 initially for all vertices i
  - label $\neq$ 0

# BFS code in Graph.java

```
/* reach[i] is set to label for all vertices reachable from vertex v ;
   bfs(1, reach, 1):  set "1" to all nodes in reach[] reachable from vertex 1 */
public void bfs(int v, int [] reach, int label)   {
      ArrayQueue q = new ArrayQueue(10);
      reach[v] = label;
      q.put(new Integer(v));
      while (!q.isEmpty())  {  // remove a labeled vertex from the queue
          int w = ((Integer) q.remove()).intValue();
         // mark all unreached vertices adjacent from w
         Iterator iw = iterator(w);
         while (iw.hasNext()) {  // visit an adjacent vertex of w
              int u = ((EdgeNode) iw.next()).vertex;
              if (reach[u] == 0) { // u is an unreached vertex
                 q.put(new Integer(u));
                 reach[u] = label; // mark reached }
         }
      }
} // end of bfs
```

# BFS code in AdjacencyDigraph.java

* Array-based implementation

```
public void bfs (int v, int [] reach, int label)  {
      ArrayQueue q = new ArrayQueue(10);
      reach[v] = label;
      q.put(new Integer(v));
      while (!q.isEmpty())  {
         int w = ((Integer) q.remove()).intValue();
         for (int u = 1; u <= n; u++) {
            if (a[w][u] && reach[u] == 0){ // u is an unreached vertex
                  q.put(new Integer(u));
                  reach[u] = label; }
         }
      }
 } // end of bfs
```

# BFS code in LinkedDigraph.java

* Link-based implementation

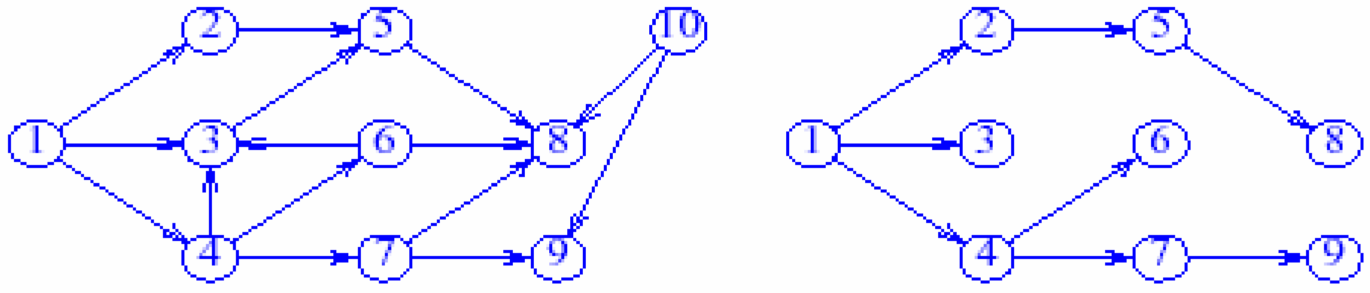
```
public void bfs (int v, int [] reach, int label)  {
      ArrayQueue q = new ArrayQueue(10);
      reach[v] = label;
      q.put(new Integer(v));
      while (!q.isEmpty()) {
         int w = ((Integer) q.remove()).intValue();
        for (ChainNode p = aList[w].firstNode; p != null; p = p.next) {
            int u = ((EdgeNode) p.element).vertex;
            if (reach[u] == 0) {   // u is an unreached vertex
               q.put(new Integer(u));
               reach[u] = label; }
         }
     } // end of while
} // end of bfs
```

# Complexity Analysis of BFS

- BFS steps
  - Add the adjacent vertices to the queue exactly once
  - Delete the vertices from queue exactly once
  - Traverse the adjacent vertices exactly once

- Time for these operations
  - If s vertices are labeled and there are n vertices in a graph
    - O(s*n) : if array-based adjacency matrix is used
    - $O(\sum_i d_i^{out})$ : if linked adjacency list is used

# Depth-First Search (1)




(a) (b)

- To determine all the vertices reachable from vertex 1
  - v=1, candidate of u : 2, 3, 4 // 2 is selected
  - v=2, candidate of u : 5 // 5 is selected
  - v=5, candidate of u : 8 // 8 is selected
  - v=8, no unreached adjacent node // back up vertex 5.
  - v=5, no unreached adjacent node // back up vertex 2
  - v=2, no unreached adjacent noide // back up vertex 1
  - v=1, candidate of u : 3, 4 //3 is selected
  - Keep going…….
- We need STACK!

# Depth-First Search (2)

```
depthFirstSearch(v) {
    Label vertex v as reached.
    for(each unreached vertex u adjacent from v)
        depthFirstSearch(u);
}
```

- DFS can be implemented using a stack
- Theorem 17.2: Let G be an arbitrary graph and let v be any vertex of G. The pseudo-code depthFirstSearch labels all vertices that are reachable from v (including vertex v)

# DFS in Graph.java

```
/* reach[i] is set to label for all vertices reachable from vertex v
   dfs(1, reach, 1): set "1" to all nodes in reach[] rechable from vertex 1  */
public void dfs (int v, int [] reach, int label) {
    Graph.reach = reach;
    Graph.label = label;
    rDfs(v);
}
/** recursive dfs method */
private void rDfs(int v) {
      reach[v] = label;
      Iterator iv = iterator(v);
      while (iv.hasNext()) { // visit an adjacent vertex of v
         int u = ((EdgeNode) iv.next()).vertex;
         if (reach[u] == 0)    rDfs(u);    // u is an unreached vertex
      }
}
```

# Other DFS codes

- DFS in AdjacencyDigraph.java
- DFS in LinkedDigraph.java

- Yes, it is your job!

# Complexity Comparison of DFS vs BFS

- We can prove DFS & BFS have the same time & space complexities
- DFS ➔ stack space for the recursion
- BFS ➔ queue space

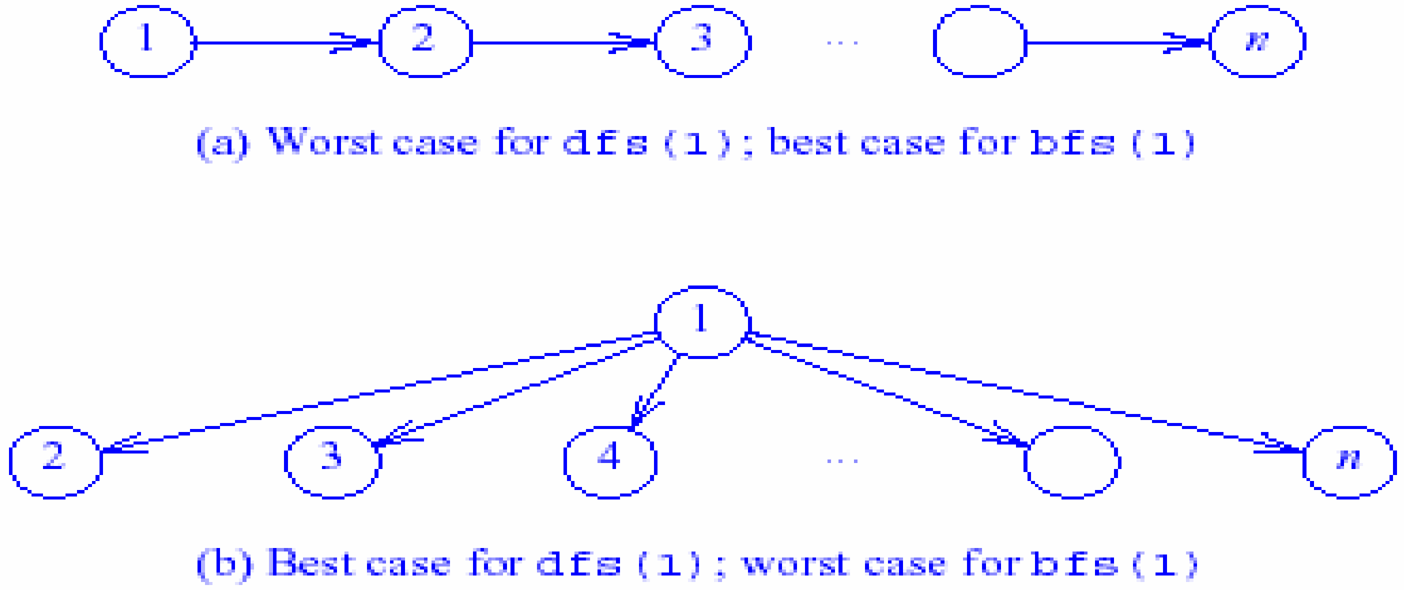



Figure 17.19 Worst-case and best-case space-complexity graphs

# Table of Contents

- Definition and Applications
- The ADT Graph
- Representation of Unweighted Graphs
- Representation of Weighted Graphs
- Class Implementations
- Graph Search Methods
- *Application Revisited*
  - Find a path in a Digraph (using DFS)
  - Connected Graph in a Graph (using DFS)
  - Component Labeling Problem in a Graph (using BFS)
  - Spanning Tree in a Graph (using DFS & BFS)

# Finding a Path in a Digraph

- To actually construct the path in a directed graph, we need to remember the edges used to move from one vertex to the next

- findpath()
  - Return null if no path
  - Return an array p[length] from s to d if find a path where p[0] = s and p[p.length -1] = d

- findPath() calls rFindPath() which is a modified DFS

# Code for findPath()

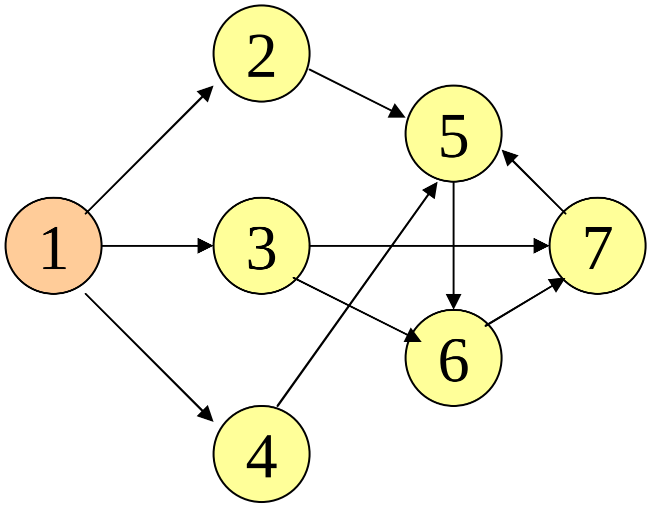
```
/** find a path from s to d
  * @return the path in an array using positions 0 on up
  * @return null if there is no path */
public int [] findPath(int s, int d) {  // initialize for recursive path finder
    int n = vertices();
    path = new int [n];
    path[0] = s;             // first vertex is always s
    length = 0;              // current path length
    destination = d;
    reach = new int [n + 1];  // by default reach[i] = 0 initially
   // search for path
    if (s == d || rFindPath(s)) { // a path was found, trim array to path size
      int [] newPath = new int [length + 1];
      System.arraycopy(path, 0, newPath, 0, length + 1); // copy from old space to new space
      return newPath; }
    else return null;
 } // end of findPath
```

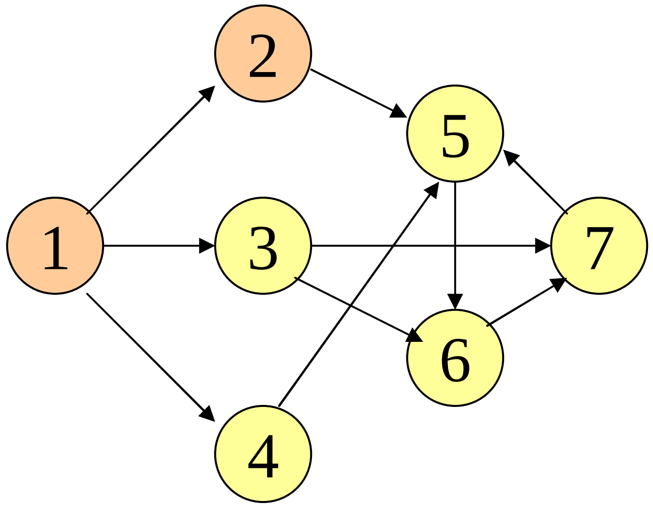
# Code for rFindPath()

```
private boolean rFindPath(int s); {
   reach[s] = 1;
   Iterator is = iterator(s);
   while (is.hashNext()) { // visit an adjacent vertex of s
     int u = ((EdgeNode) is.next()).vertex;
     if (reach[u] == 0) { /* u is an unreached vertex & move to vertex u
       length++;
       path[length] = u; // add u to path
       if (u == destination) return true;
       if (rFindPath(u)) return true;
       // no path from u to destination
       length--; } // remove u from path
   }
  return false
} // end of rFindPath
```

# An Example: findPath() (1)

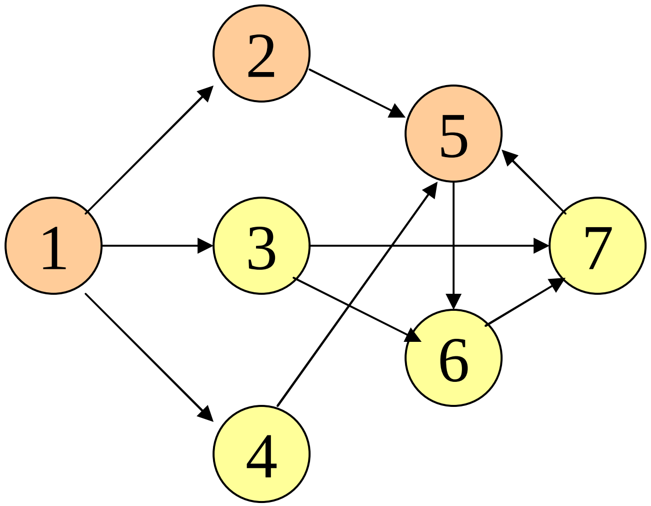
- Source = 1, Destination =7:  length=1, path={1,2}

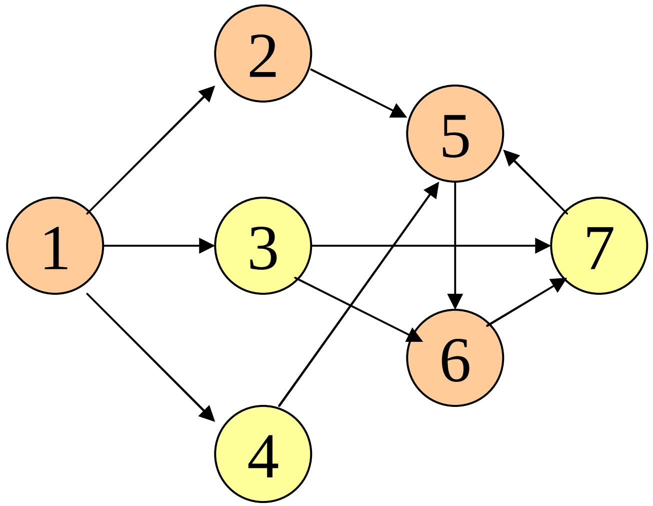


# An Example: findPath() (2)

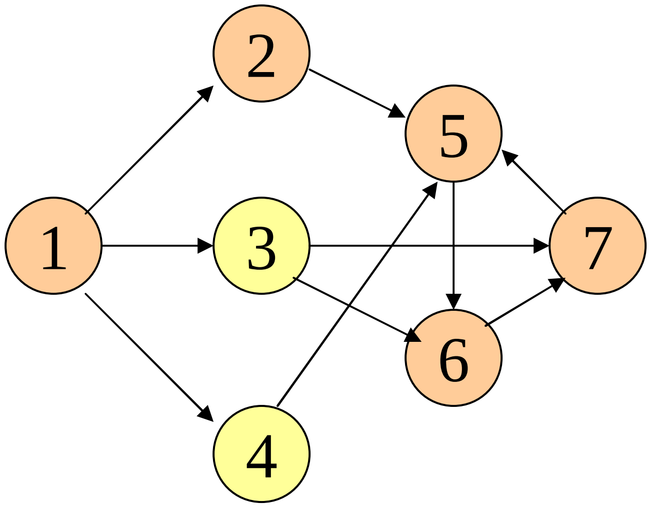
- Source = 1, Destination =7: length=3, path={1,2,5,6}



# An Example: findPath() (3)

- Source = 1, Destination = 7: length = 4, path={1,2,5,6,7}




Find!!

# Table of Contents

# Connected Graph

- Determine whether an undirected graph G is connected by performing DFS or BFS

- Here connected() is based on DFS

- connected() return
  - true : if the graph is connected
  - false : if the graph is not connected

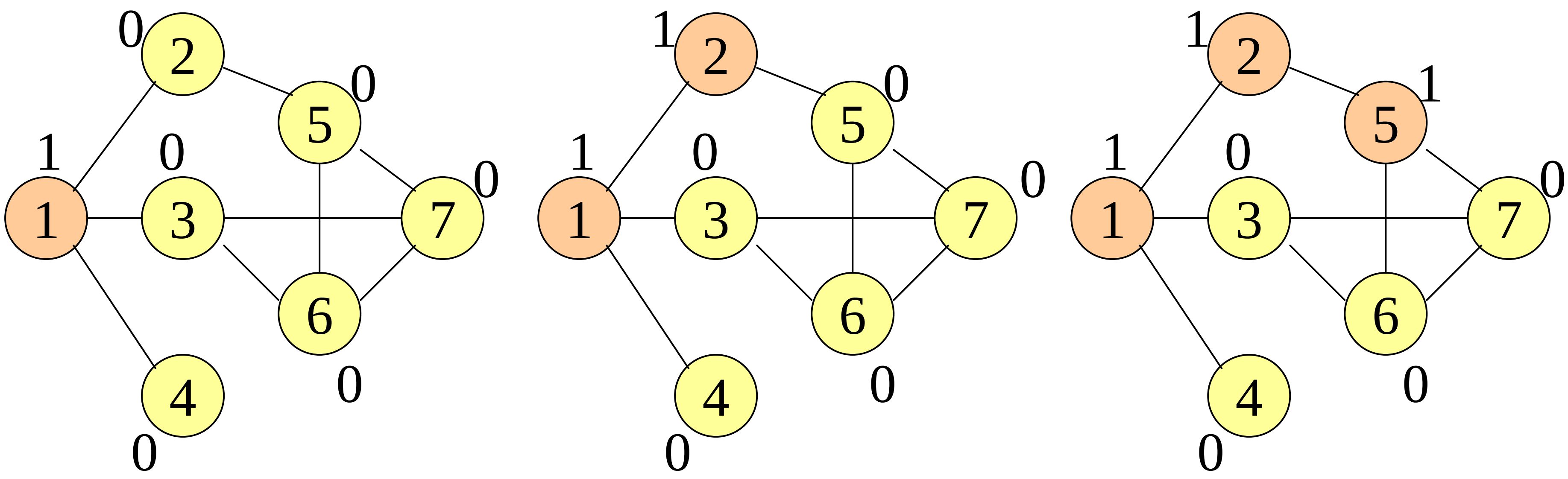
- Connected component
  - Vertices that are reachable from a vertex i, together with the edges that connect pair of vertices in a graph

# Code for connected()

```
/** @return true iff graph is connected */
 public boolean connected() {
    // make sure this is an undirected graph
    verifyUndirected("connected");

    int n = vertices();
    reach = new int [n + 1];  // by default reach[i] = 0

   dfs(1, reach, 1); // mark vertices reachable from vertex 1

    for (int i = 1; i <= n; i++) // check if all vertices marked
       if (reach[i] == 0)   return false;
    return true;
 }
```

# An Example: connected() (1)

- Mark vertices reachable from vertex 1
- DFS & visit the node with the smallest key first

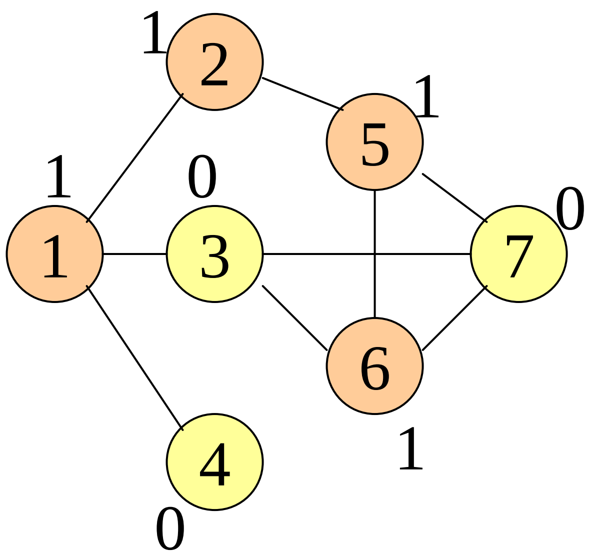


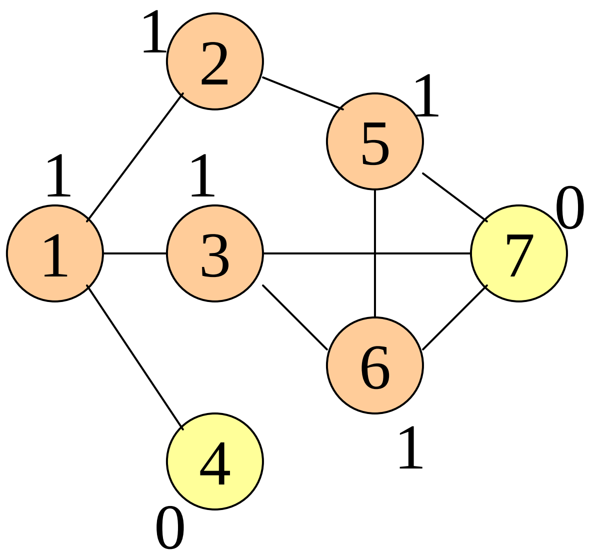



SNU IDB Lab.

# An Example: connected() (2)

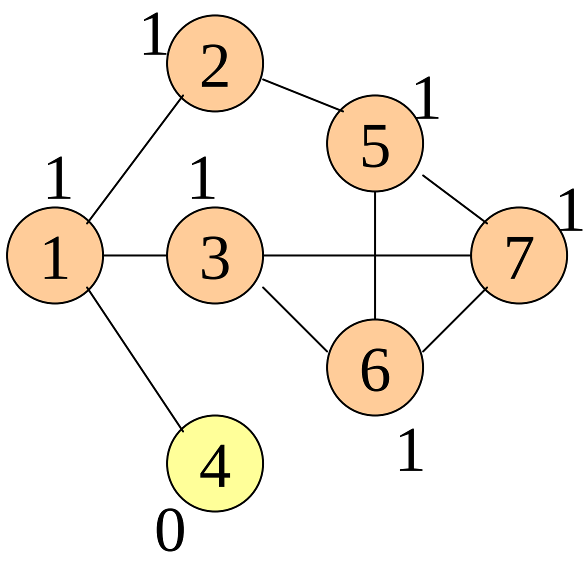
- From node 5: visit node 6
- From node 6: visit node 3

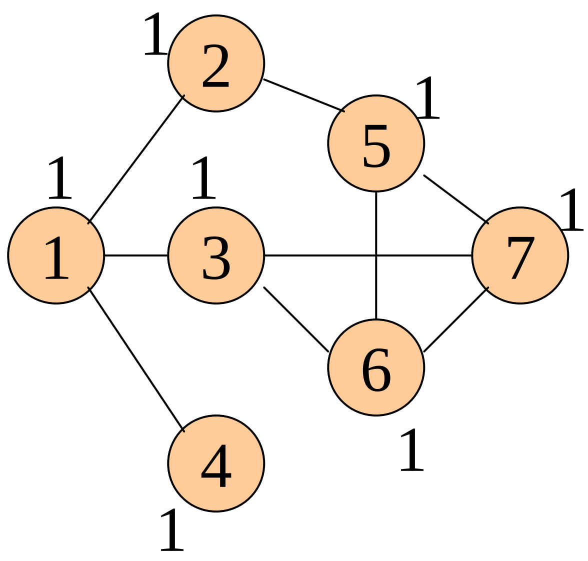


# An Example: connected() (3)

- From node 3: visit node 7, then finish up the first DFS
- Go back to node 1, and visit the remaining node 4. ➔ Done!



# Table of Contents

- Definition and Applications
- The ADT Graph
- Representation of Unweighted Graphs
- Representation of Weighted Graphs
- Class Implementations
- Graph Search Methods
- *Application Revisited*
  - Find a path in a Digraph
  - Connected Graph in a Graph
  - Component Labeling Problem in a Graph
  - Spanning Tree in a Graph

# Component Labeling Problem in a Graph

- Component-labeling problem
  - we are to label the vertices of an undirected graph so that two vertices are assigned the same label iff they belong to the same component
  - verifyUndirected() is needed

- Label components
  - making repeated invocations of either a DFS or BFS

- Graph.labelComponent()
  - Different components have different labels
  - Here, solve the component-labeling problem using BFS
  - $O(n^2)$ : if adjacency matrix is used for n nodes
  - $O(n + e)$ : if linked-adjacency-list representation is used for n nodes & e edges

# Code for verifyUndirected()

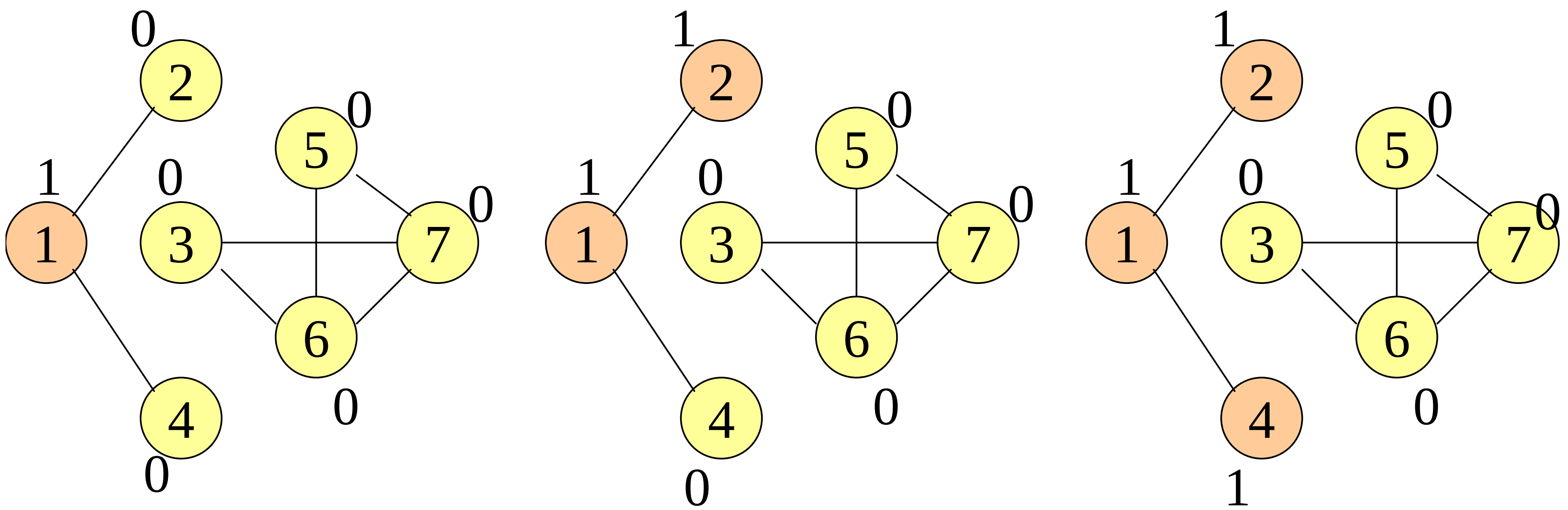
```
/** verify that the graph is an undirected graph
  * @exception UndefinedMethodException if graph is directed */
public void verifyUndirected(String theMethodName) {
      Class c = getClass();    // class of this
      if (c == AdjacencyGraph.class ||
          c == AdjacencyWGraph.class ||
          c == LinkedGraph.class ||
          c == LinkedWGraph.class)
         return;

       // if not an undirected graph
     throw new UndefinedMethodException
          ("Graph." + theMethodName + " is for undirected graphs only");
}
```

# Code for labelComponents()

```
/** label the components of an undirected graph
  *  @return the number of components
  *  set c[i] to be the component number of vertex i */
 public int labelComponents(int [] c)  { // make sure this is an undirected graph
   verifyUndirected("labelComponents");
   int n = vertices();
   // assign all vertices to no component
   for (int i = 1; i <= n; i++)   c[i] = 0;
   label = 0;  // ID of last component
   // identify components
   for (int i = 1; i <= n; i++)
      if (c[i] == 0) {       // vertex i is unreached  // vertex i is in a new component
         label++;            // new label for new component
         bfs(i, c, label);   // mark new component
      }
   return label;
 }
```

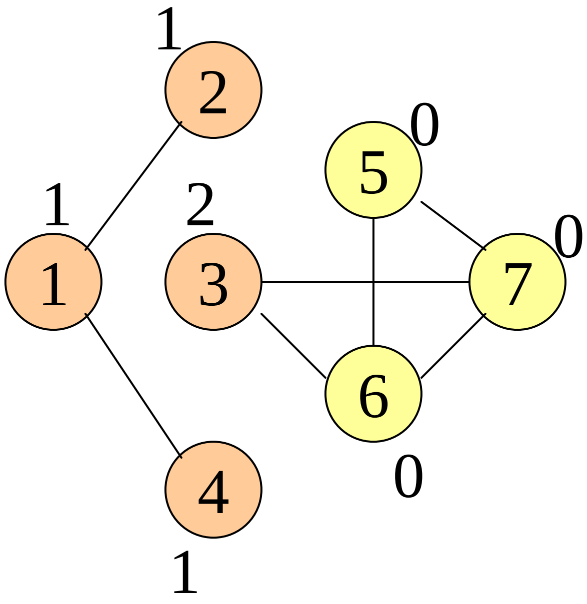
# Example: labelComponents() (1)

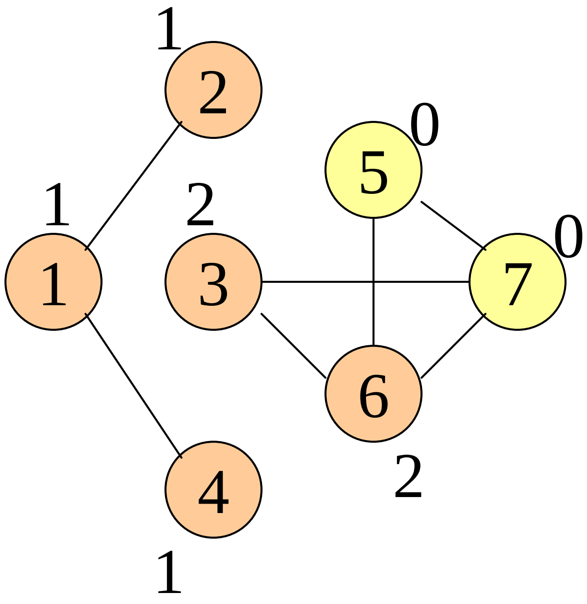
* label = 1,  execute bfs(1, c, label) from node 1



# Example: labelComponents() (2)

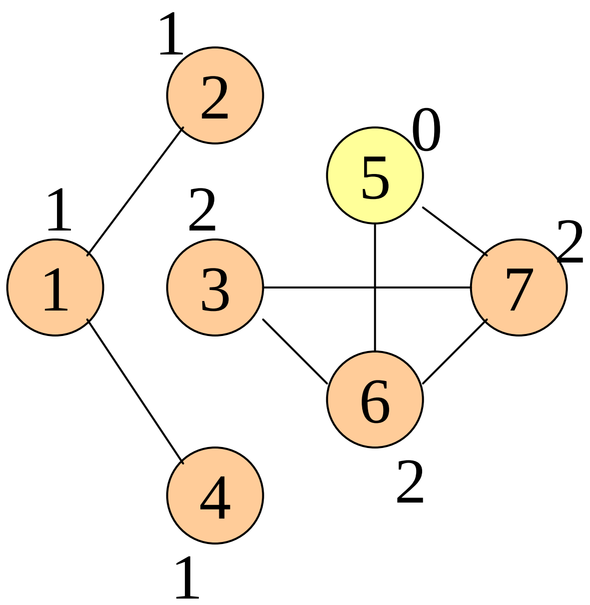
* Now label = 2,  execute bfs(3, c, label)  from node 3

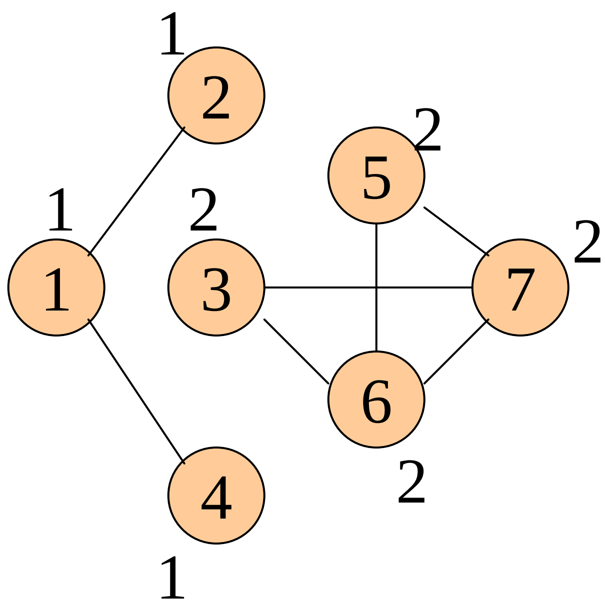


# Example: labelComponents() (3)

- label = 2,  bfs(3, c, label)
- From node 3, visit node 6 & 7
- From node 6, visit node 5



# Table of Contents

# Spanning Trees

- Spanning Tree
  - A set of edges contains a path from v to every other vertex in the graph
  - It defines a connected subgraph
  - Normally applied to connected undirected graphs

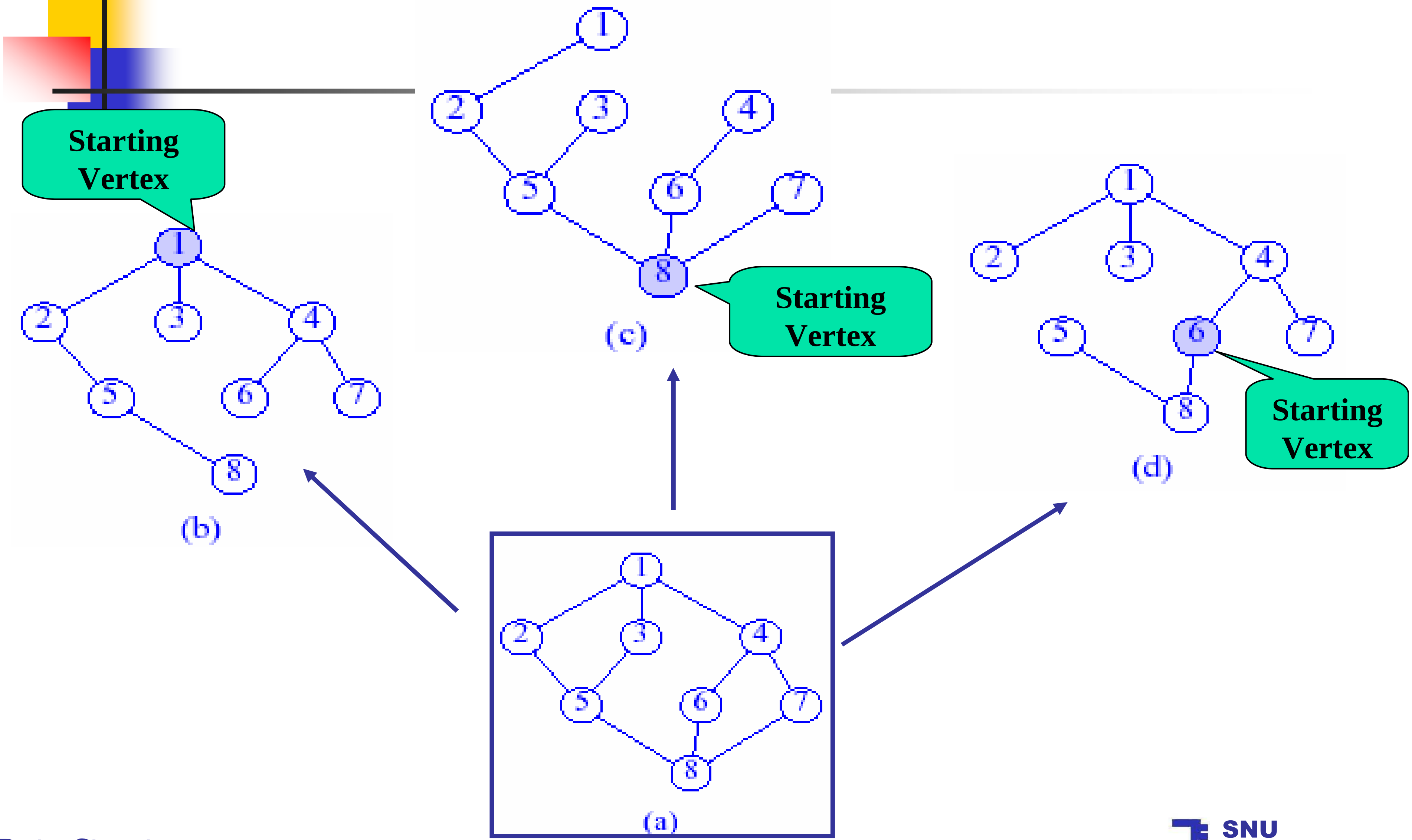
- Bread-first spanning tree (BF spanning tree)
  - Spanning tree obtained in the manner from BFS

- Depth-first spanning tree (DF Spanning tree)
  - Spanning tree obtained in the manner from DFS

# Code for BF Spanning Tree

```
/** SpanningTree with BFS
   * reach[i] is set to label for all vertices reachable from vertex v
   * sTreebfs(v, reach, 1): set "1" to the nodes in reach[] from the node v */
public void sTreebfs(int v, int [] reach, int label) {
      ArrayQueue q = new ArrayQueue(10);
      reach[v] = label;
      q.put(new Integer(v));
      while (!q.isEmpty()) {
         int w = ((Integer) q.remove()).intValue(); // remove a labeled vertex from the queue
         iterator iw = iterator(w); // mark all unreached vertices adjacent from w
        while (iw.hasNext()) { // visit an adjacent vertex of w
            int u = ((EdgeNode) iw.next()).vertex;
            if (reach[u] == 0) { // u is an unreached vertex
               q.put(new Integer(u));
               reach[u] = label; // mark reached
               for (i=1; i <=q.length(); i++) {  // remove edge!!
                       int t = ((Integer) q.remove()).intValue();
                       if (existsEdge(u,t)) {  removeEdge(u, t) }
                       q.put(new Integer(t)); }
              } // end of if
         } // end fo while
      }
} // end of sTreeBfs
```

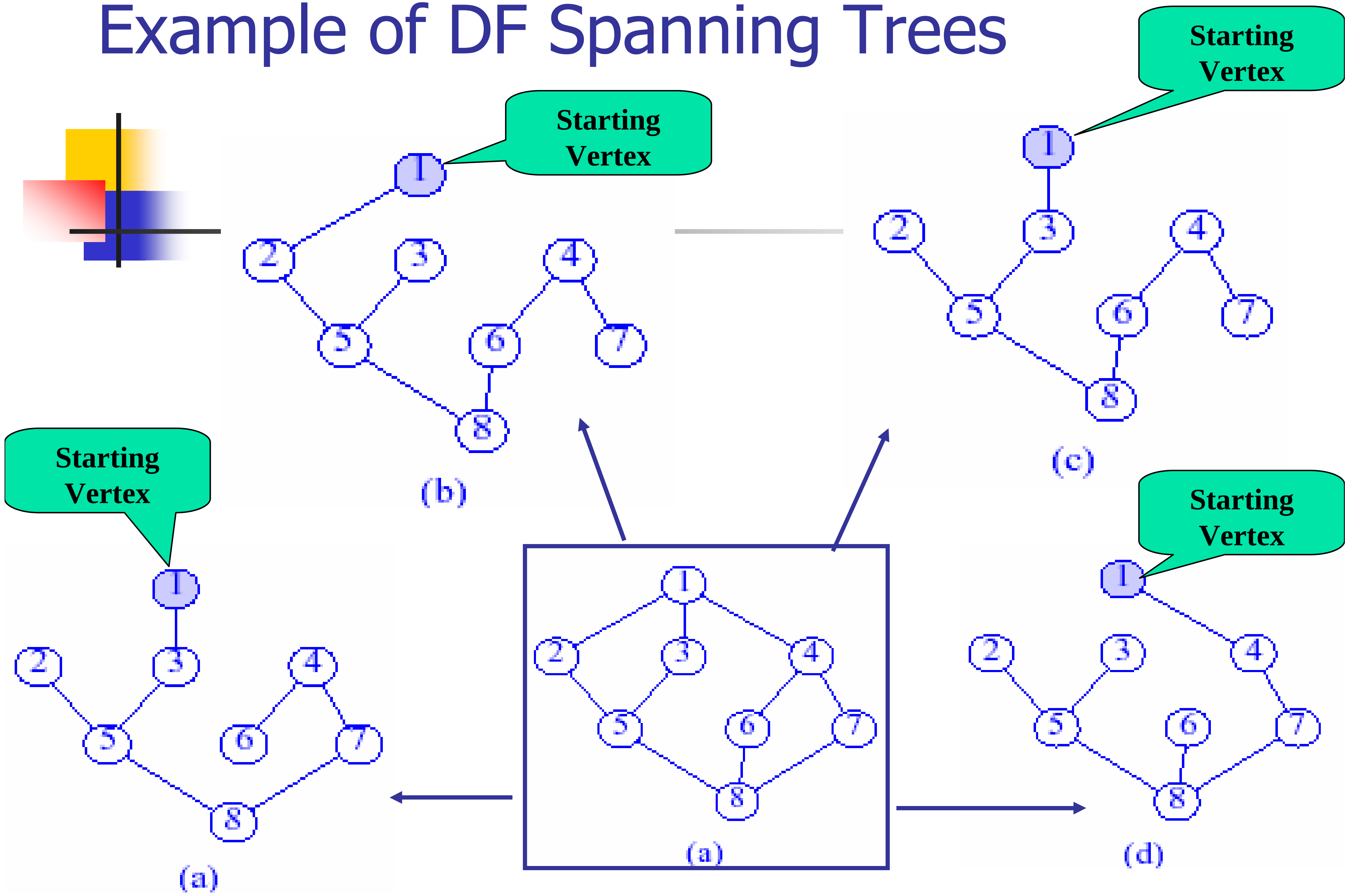
# Example of BF Spanning Trees



# Code for DF SpanningTree

```
/* SpanningTree with DFS
   reach[i] is set to label for all vertices reachable from vertex v*/
public void sTreedfs(int v, int [] reach, int label) {
      reach[v] = label;
      Iterator iv = iterator(v);
      // visit an adjacent vertex of v
      int u1 = ((EdgeNode) iv.next()).vertex;
      if (reach[u] == 0)  // u is an unreached vertex
      while (iv.hasNext()) {
          // remove edge
          int u2 = ((EdgeNode) iv.next()).vertex;
          if (existsEdge(v,u2))  {  removeEdge(v,u2)  }
       }
      sTreedfs(u1);
 }
```

# Example of DF Spanning Trees



# Summary

- Graphs
  - Used to model many real-world problems
- In this chapter
  - Graph terminology
  - Different types of graphs
  - Common graph representations
  - Standard graph search methods
  - Algorithms to find a path in a graph
  - Specifying an abstract data type as an abstract class

# Sahni class: dataStructures.LinkedDigraph (p.672)

public class ***LinkedDigraph*** extends Graph{

constructors

*LinkedDigraph()*: Constructs an empty directed graph

methods

*int inDegree(int i)*: Returns the in-degree of vertex *i*

*int outDegree(int i)*: Returns the out-degree of vertex *i*

*int putEdge(theEdge)*: Puts *theEdge* into the digraph

*int removeEdge(int i, int j)*: Removes the edge (*i*, *j*) from the digraph

*int existsEdge(int i, int j)*: Returns *true* iff the graph contains (*i*, *j*)

}

# Sahni class: dataStructures.LinkedGraph (p.672)

public class ***LinkedGraph*** extends LinkedDigraph{

constructors

*LinkedGraph()*: Constructs an empty undirected graph

methods

*int degree(int i)*: Returns the degree of vertex *i*

*int putEdge(Object theEdge)*: Puts *theEdge* into the graph

*int removeEdge(int i, int j)*: Removes the edge (*i*, *j*) from the graph

*int existsEdge(int i, int j)*: Returns *true* iff the graph contains (*i*, *j*)

}

# Data Structures

- Chapter 2-4: Complexity of algorithms
- Chapter 5-8: Linear List
- Chapter 9-11: Stack & Queue
- Chapter 12-16: Tree
- Chapter 17: Graph