CS 10:
Problem solving via Object Oriented Programming

Relationships

Main goals

- Implement graphs

Agenda

➡ 1. Graphs

2. Four common representations

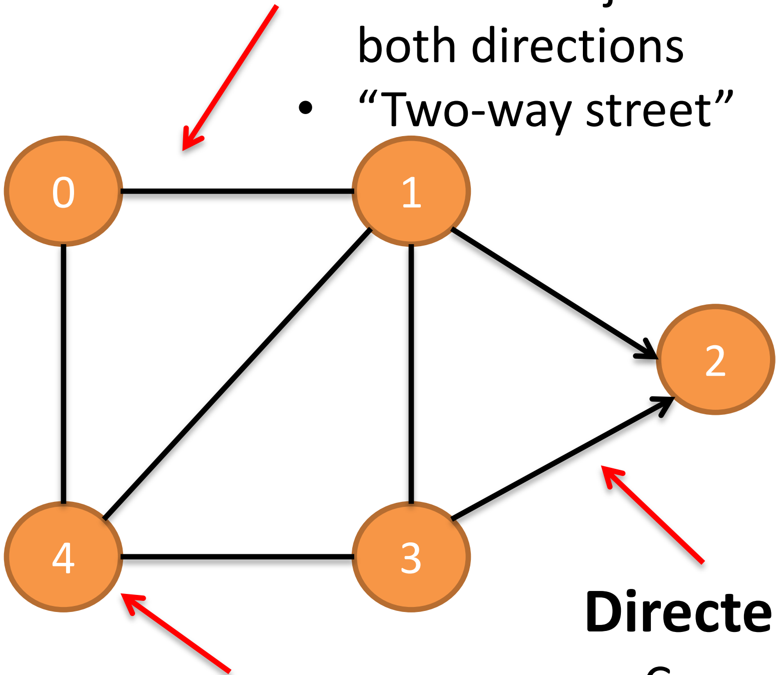
3. Implementation

Graphs represent directed or undirected relationships with nodes and edges

Graphs

Undirected edges

- Connect objects in both directions
- “Two-way street”



Directed edges

- Connect objects in a single directions
- “One-way street”

Nodes (vertices)

- Represent objects
- Could be a person or city or computer or intersection of roads…

Undirected graph

Only undirected edges

Directed graph

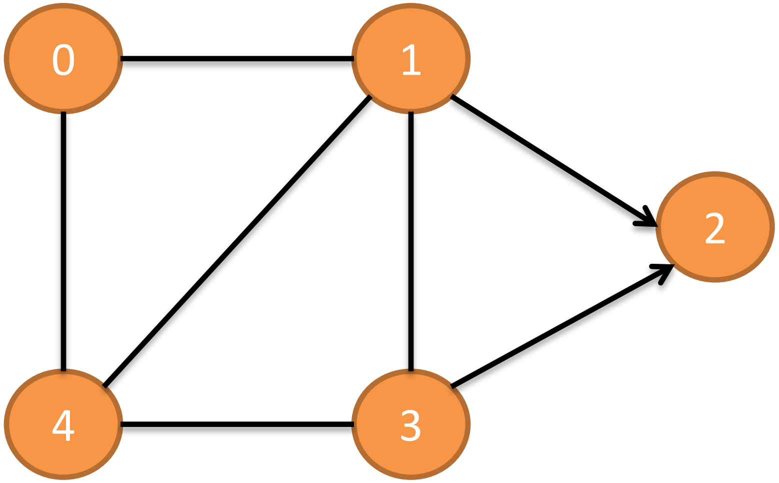
Only directed edges

Mixed graph

Has both directed and undirected edges

Both nodes and edges can hold information about the relationship

Graphs



Nodes

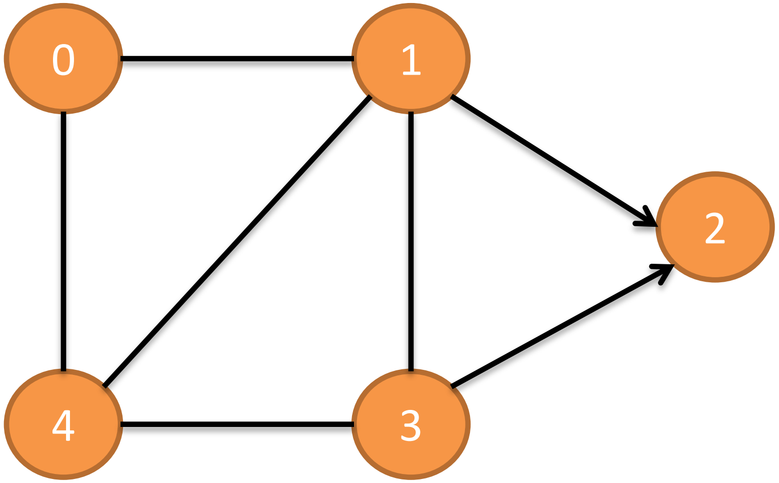
- Represent an Object
- Can be as simple as a String
- Could be more complex like an Object from a Person Class

Edges

- Can hold information about relationship
  - Distance between cities
  - Capacity of a pipe
  - Label of relationship type (“follower”, “friend”, “co-worker”)

Graph ADT defines several useful methods

Graph.java



Create/alter graph structure

`insertVertex(v)`

Add node `v` to graph

`insertDirected(u,v)/Undirected(u,v)`

Add edge to graph between node `u` and node `v`

`removeVertex(v)/removeDirected(u,v)/removeUndirected(u,v)`

Remove node `v` or edge from `u to v`

Traverse graph

`outDegree(v)/inDegree(v)`

Count of edges out of or into node `v`

`outNeighbors(v)/inNeighbors(v)`

Other nodes connected from/to node `v`

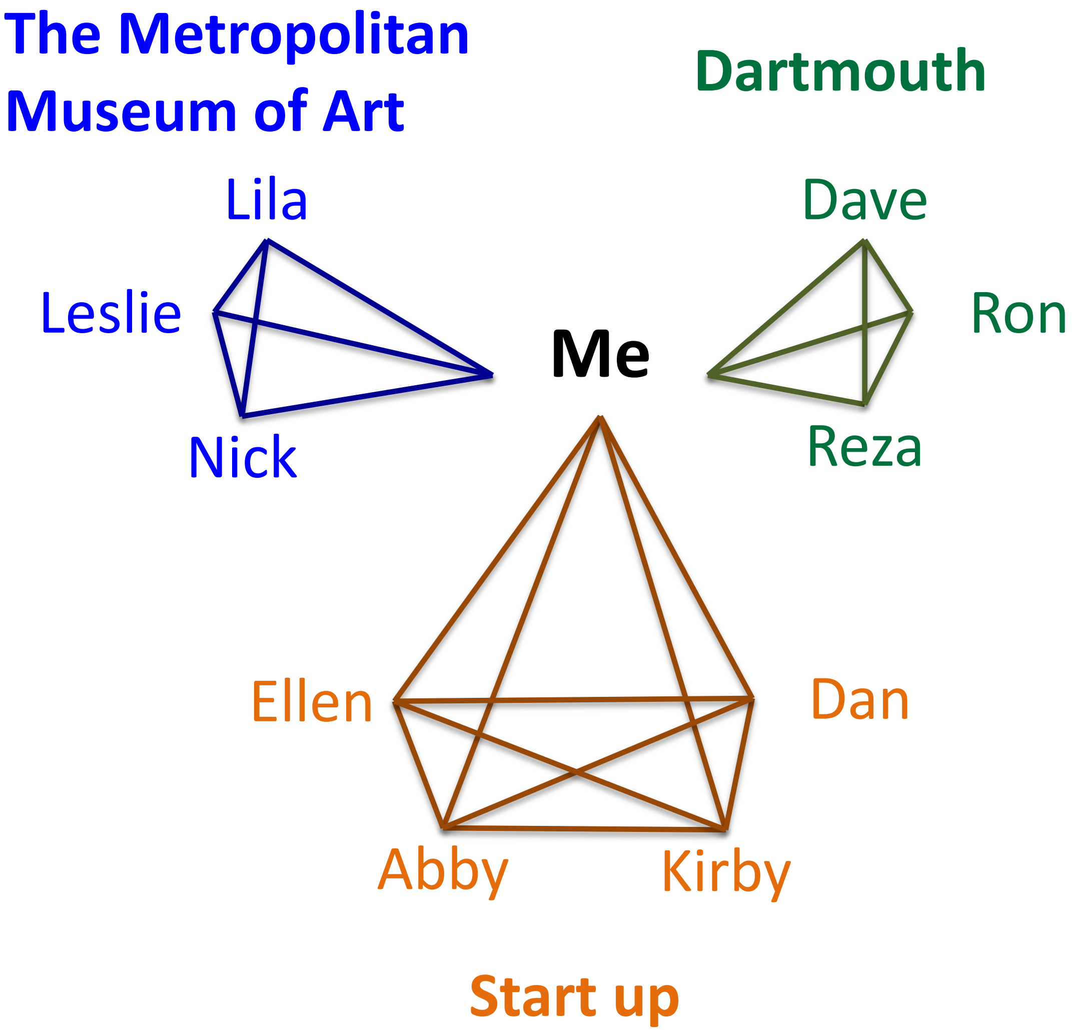
`hasEdge(u,v)`

True if node `v` connected to node `u`

`getLabel(u,v)`

Return label on edge from node `u` to node `v`

We can use Graph ADT methods to answer interesting questions



Questions we can answer

- Who is the most connected? (most in edges)
- Who are mutual acquaintances ("cliques" where all nodes have edges to each other)
- Who is a friend-of-a-friend but is not yet a friend? (breadth-first search, next class)

Agenda

1. Graphs

2. Four common representations

3. Implementation

Graphs are commonly represented in one of four different ways

Common Graph representations

➡ 1. Edge List

2. Adjacency List

3. Adjacency Matrix

4. Adjacency Map

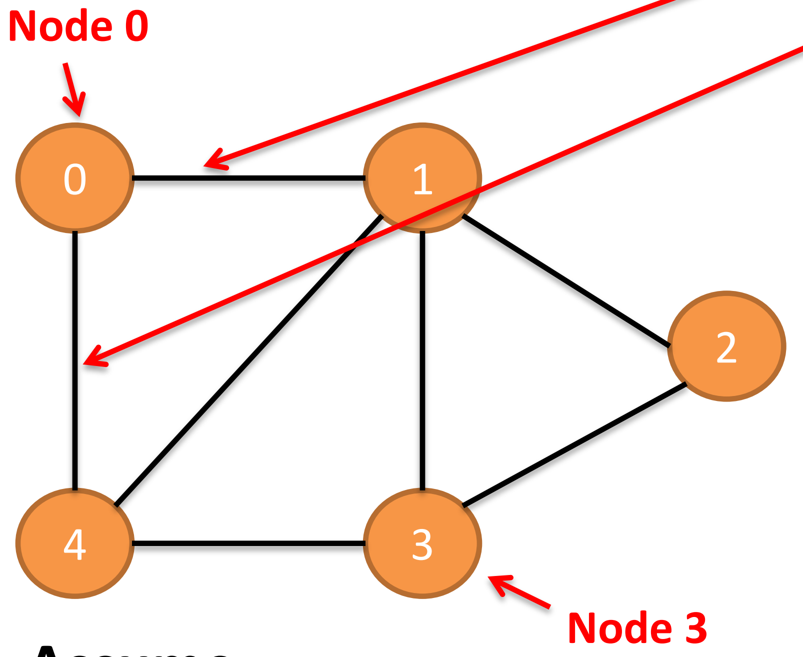
Edge Lists create an unordered list of vertex pairs where each entry is an edge

1. Edge List

List of edges

Edge list
{node #, node #}

```
{ {0,1}, {0,4}, {1,2}, {1,3},
{1,4}, {2,3}, {3,4} }
```



Assume:

n nodes (here 5)

m edges (here 7)

Graphs are commonly represented in one of four different ways

Common Graph representations

1. Edge List

➡ 2. Adjacency List

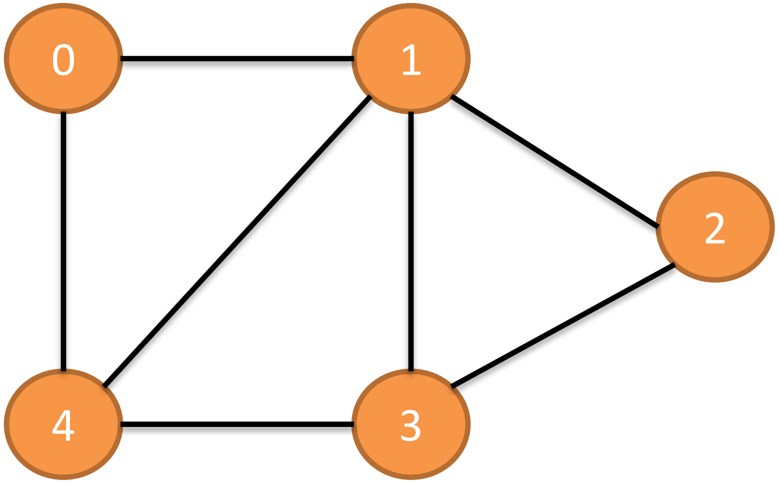
3. Adjacency Matrix

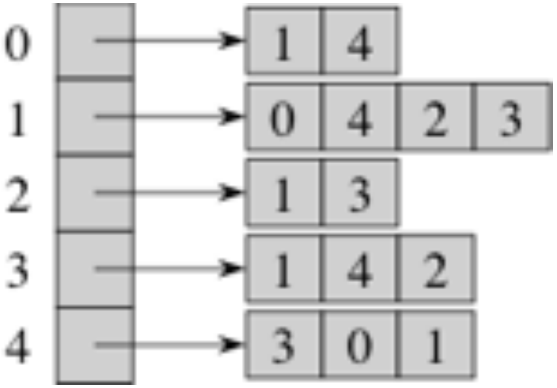
4. Adjacency Map

Adjacency Lists store adjacent nodes in a List; gives improved performance

2. Adjacency List

List of Lists





Assume:

n nodes (here 5)

m edges (here 7)

Graphs are commonly represented in one of four different ways

Common Graph representations

1. Edge List

2. Adjacency List

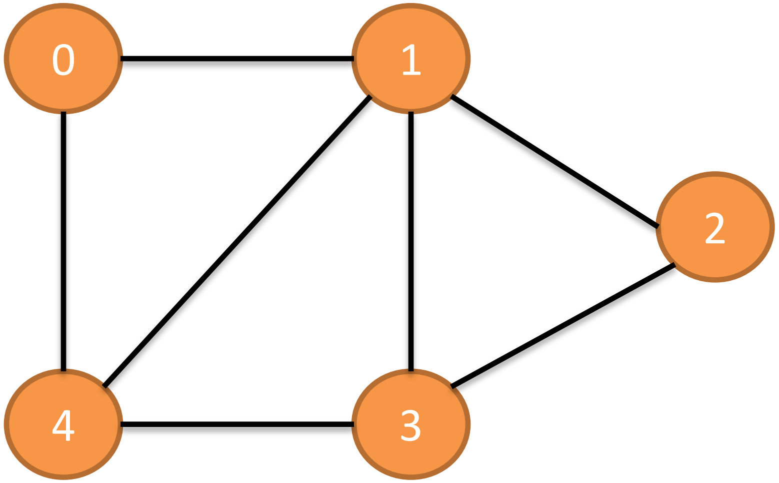
3. Adjacency Matrix

4. Adjacency Map

Adjacency Matrices create an n x n array to indicate existence of edges

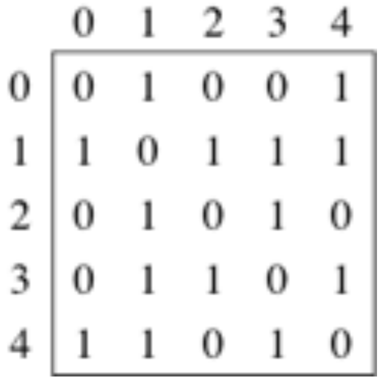
3. Adjacency Matrix

n x n array



To

From



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Assume:

n nodes (here 5)

m edges (here 7)

Graphs are commonly represented in one of four different ways

Common Graph representations

1. Edge List

2. Adjacency List

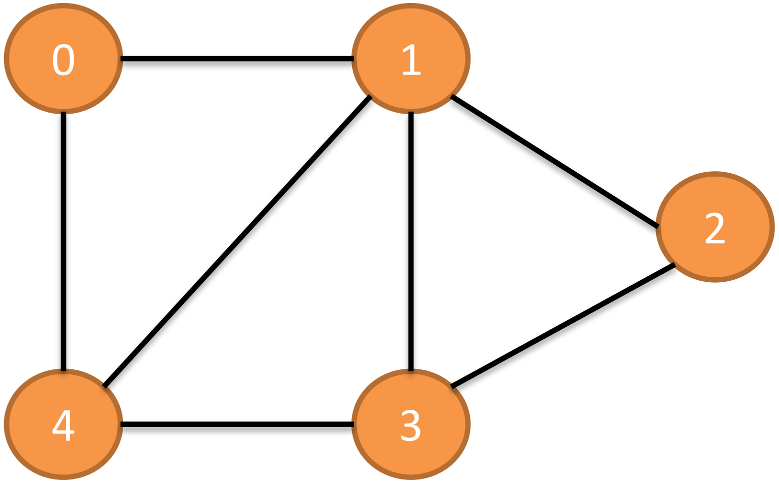
3. Adjacency Matrix

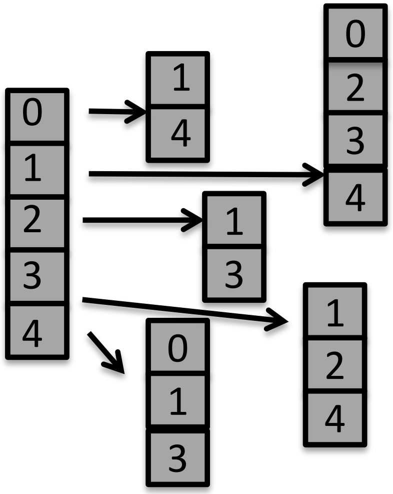
4. Adjacency Map

Adjacency Maps create a Map for each node and a second Map to adjacent nodes

4. Adjacency Map

Map of Maps



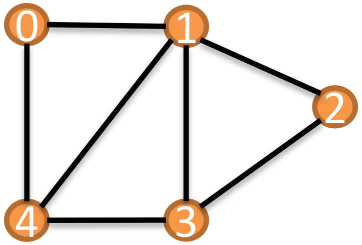


Assume:

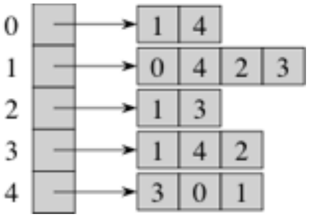
n nodes (here 5)

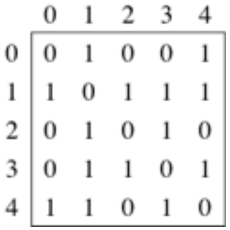
m edges (here 7)

How a Graph is implemented has a big impact on run-time performance

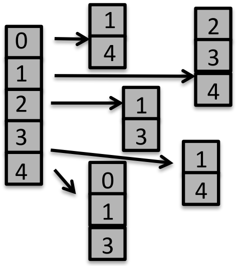


```
{{0,1},
{0,4}, {1,2},
{1,3}, {1,4},
{2,3}, {3,4}}
```





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



Method	Edge List	Adjacency List	Adjacency Matrix	Adjacency Map
`in/outDegree(v)`	O(m)	O(1)	O(n)	O(1)
`in/outNeighbors(v)`	O(m)	O(d_v)	O(n)	O(d_v)
`hasEdge(u,v)`	O(m)	O(d_u)	O(1)	O(1)
`insertVertex(v)`	O(1)	O(1)	O(n^2)	O(1)
`removeVertex(v)`	O(m)	O(d_v)	O(n^2)	O(d_v)
`insertEdge(u,v,e)`	O(1)	O(1)	O(1)	O(1)
`removeEdge(u,v)`	O(m)	O(1)	O(1)	O(1)

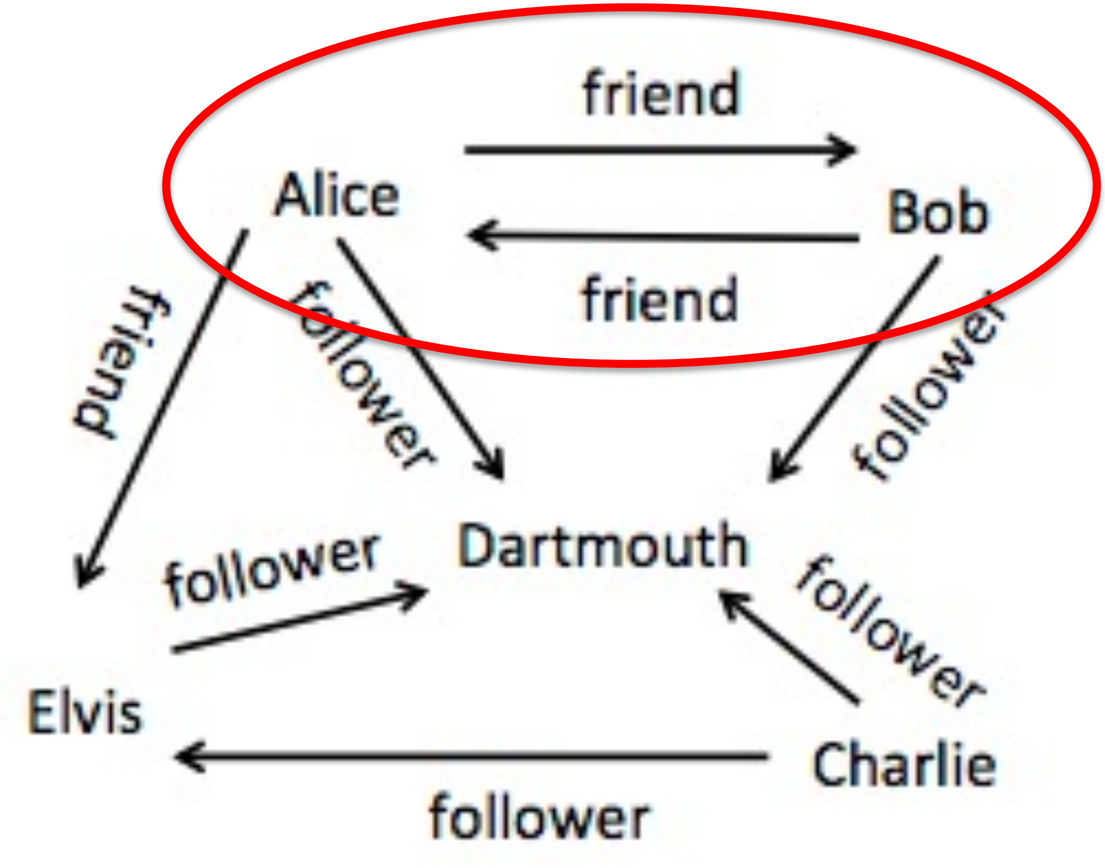
Best performance is shown in red

n = number of nodes (5), m = number of edges (7), d_v = degree of node v

Agenda

1. Graphs
2. Four common representations
3. Implementation

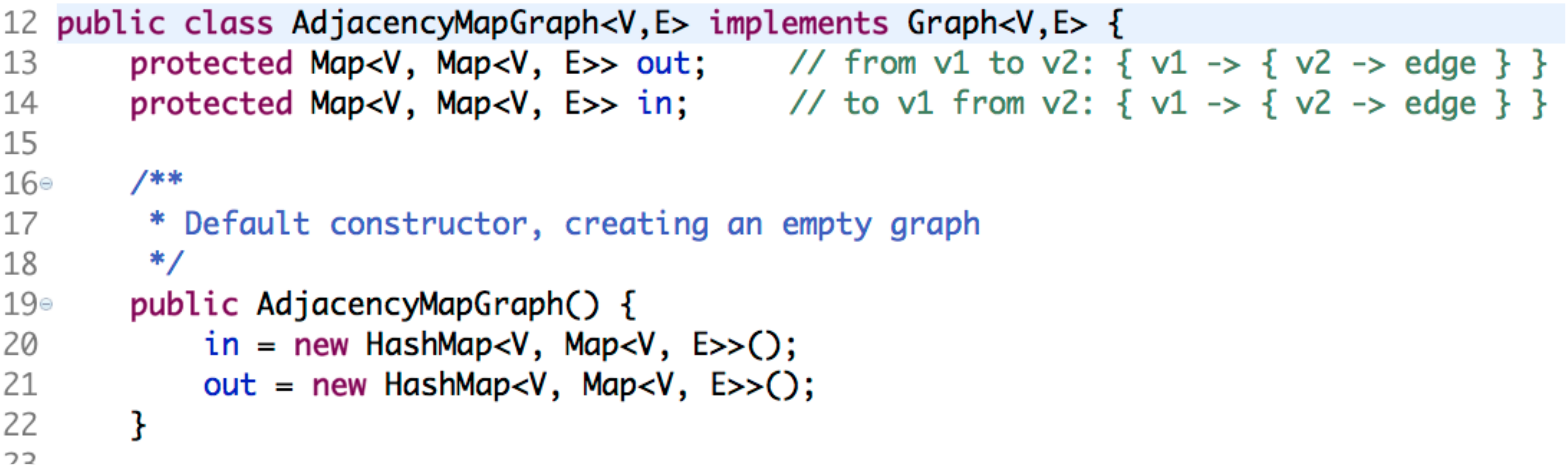
Our implementation will allow a mixed graph (directed and undirected edges)



Undirected edges are two directed edges, one in each direction

AdjancyMapGraph.java tracks *in* and *out* edges in two different Maps

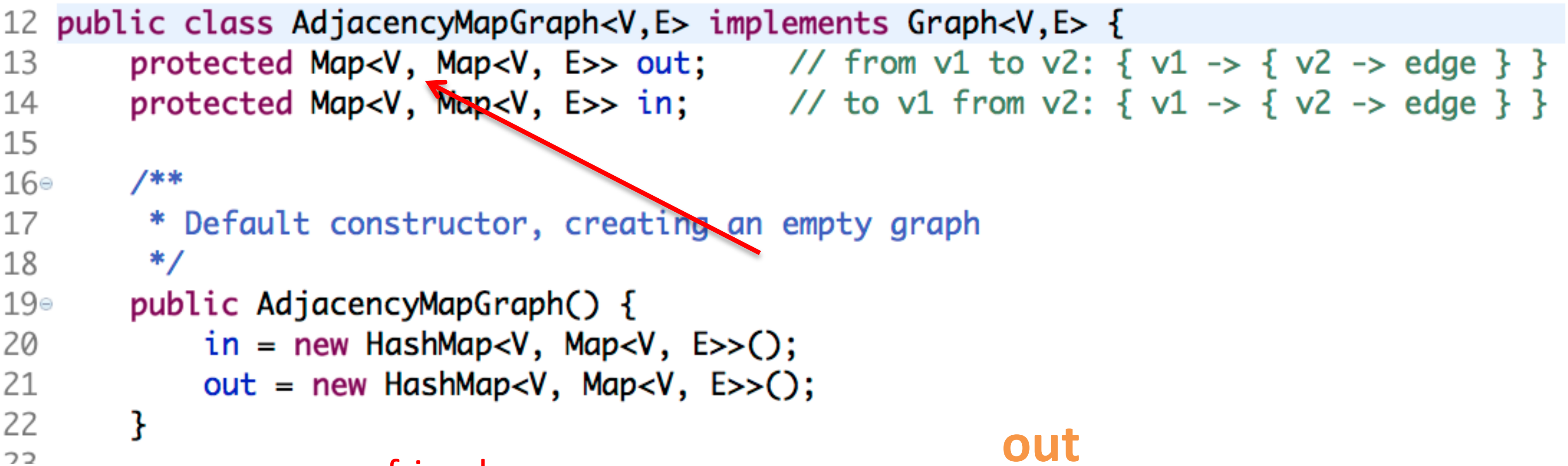
AdjacencyMapGraph.java



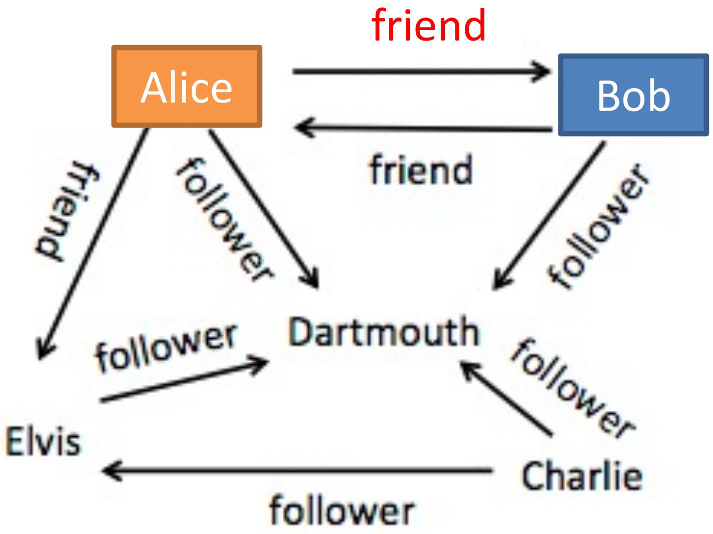
```
public class AdjacencyMapGraph<V,E> implements Graph<V,E> {
    protected Map<V, Map<V, E>> out;     // from v1 to v2: { v1 -> { v2 -> edge } }
    protected Map<V, Map<V, E>> in;      // to v1 from v2: { v1 -> { v2 -> edge } }

    /**
     * Default constructor, creating an empty graph
     */
    public AdjacencyMapGraph() {
        in = new HashMap<V, Map<V, E>>();
        out = new HashMap<V, Map<V, E>>();
    }
```

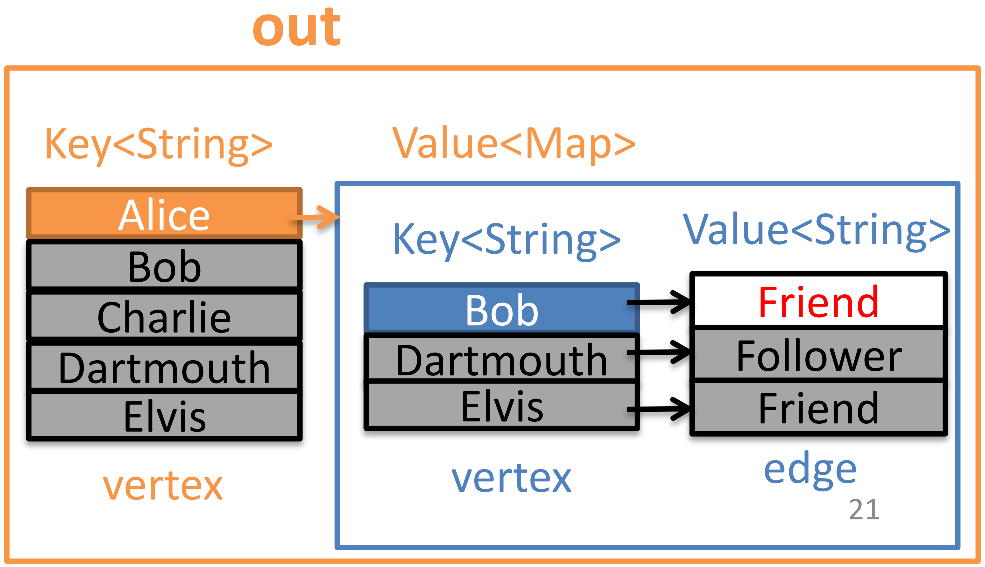
out tracks edges leaving a vertex

AdjacencyMapGraph.java



```java
public class AdjacencyMapGraph<V,E> implements Graph<V,E> {
    protected Map<V, Map<V, E>> out;    // from v1 to v2: { v1 -> { v2 -> edge } }
    protected Map<V, Map<V, E>> in;     // to v1 from v2: { v1 -> { v2 -> edge } }

    /**
     * Default constructor, creating an empty graph
     */
    public AdjacencyMapGraph() {
        in = new HashMap<V, Map<V, E>>();
        out = new HashMap<V, Map<V, E>>();
    }
```





out

Key<String>

Value<Map>

Alice
Bob
Charlie
Dartmouth
Elvis

vertex

Key<String>

Value<String>

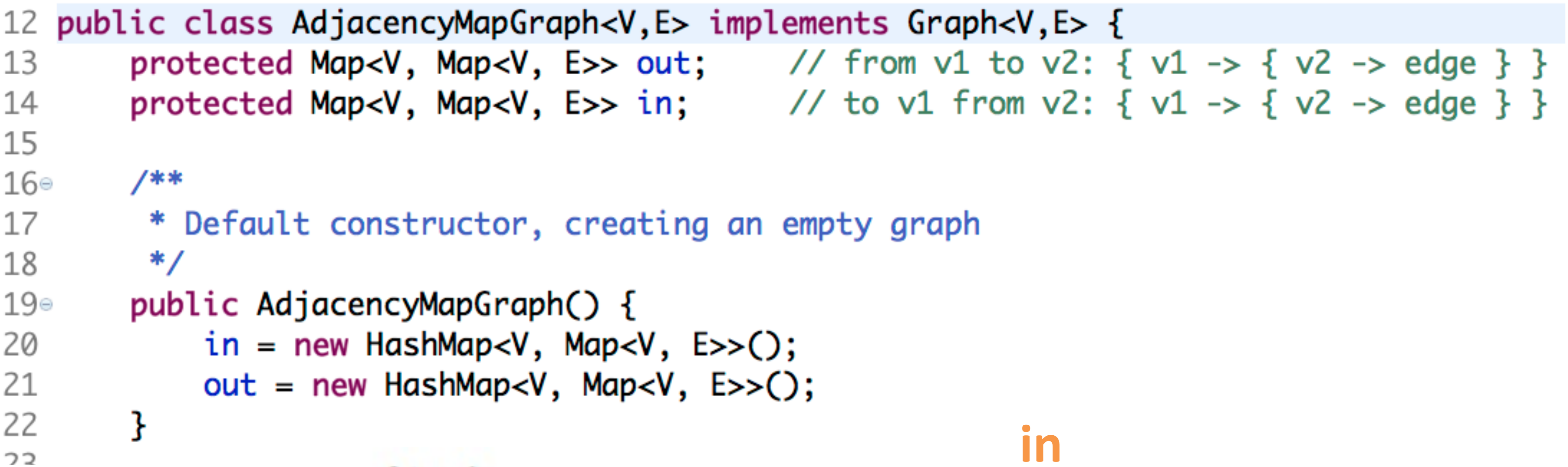
Bob → Friend
Dartmouth → Follower
Elvis → Friend

vertex

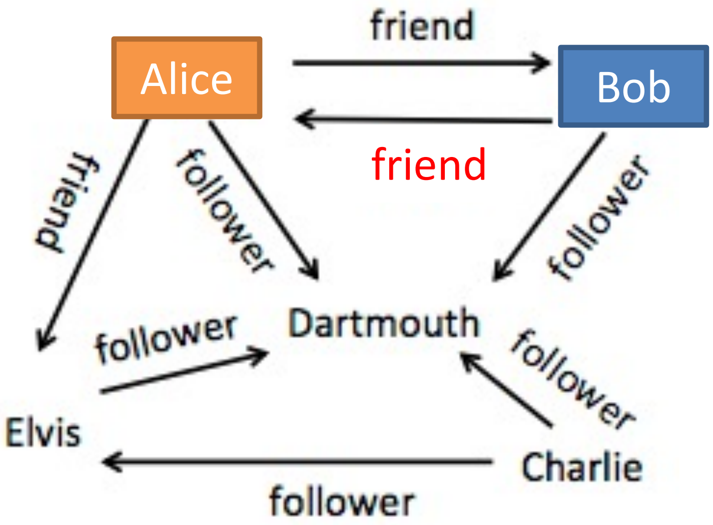
edge

in tracks edges entering a vertex

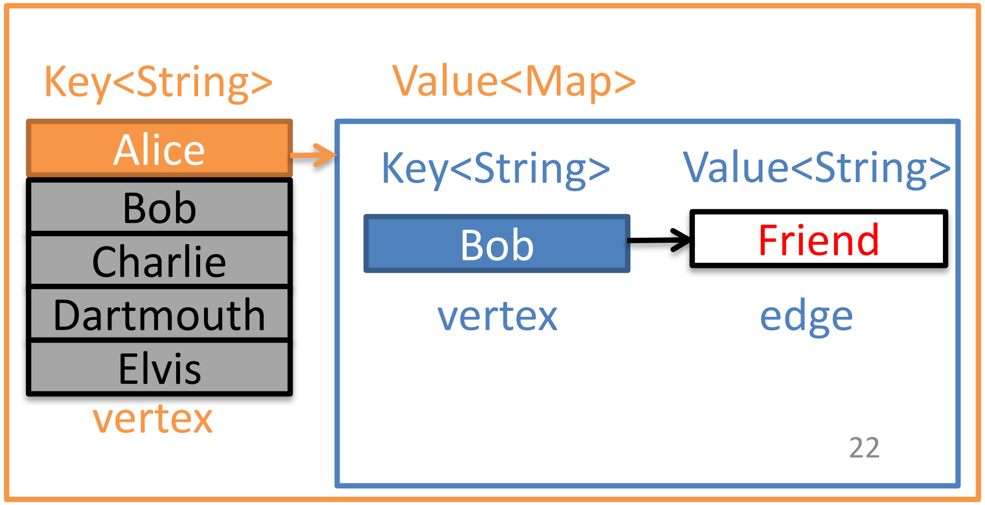
AdjacencyMapGraph.java



```java
public class AdjacencyMapGraph<V,E> implements Graph<V,E> {
    protected Map<V, Map<V, E>> out;    // from v1 to v2: { v1 -> { v2 -> edge } }
    protected Map<V, Map<V, E>> in;     // to v1 from v2: { v1 -> { v2 -> edge } }

    /**
     * Default constructor, creating an empty graph
     */
    public AdjacencyMapGraph() {
        in = new HashMap<V, Map<V, E>>();
        out = new HashMap<V, Map<V, E>>();
    }
```



in



Key<String>

Value<Map>

Alice

Bob

Charlie

Dartmouth

Elvis

vertex

Key<String>

Value<String>

Bob

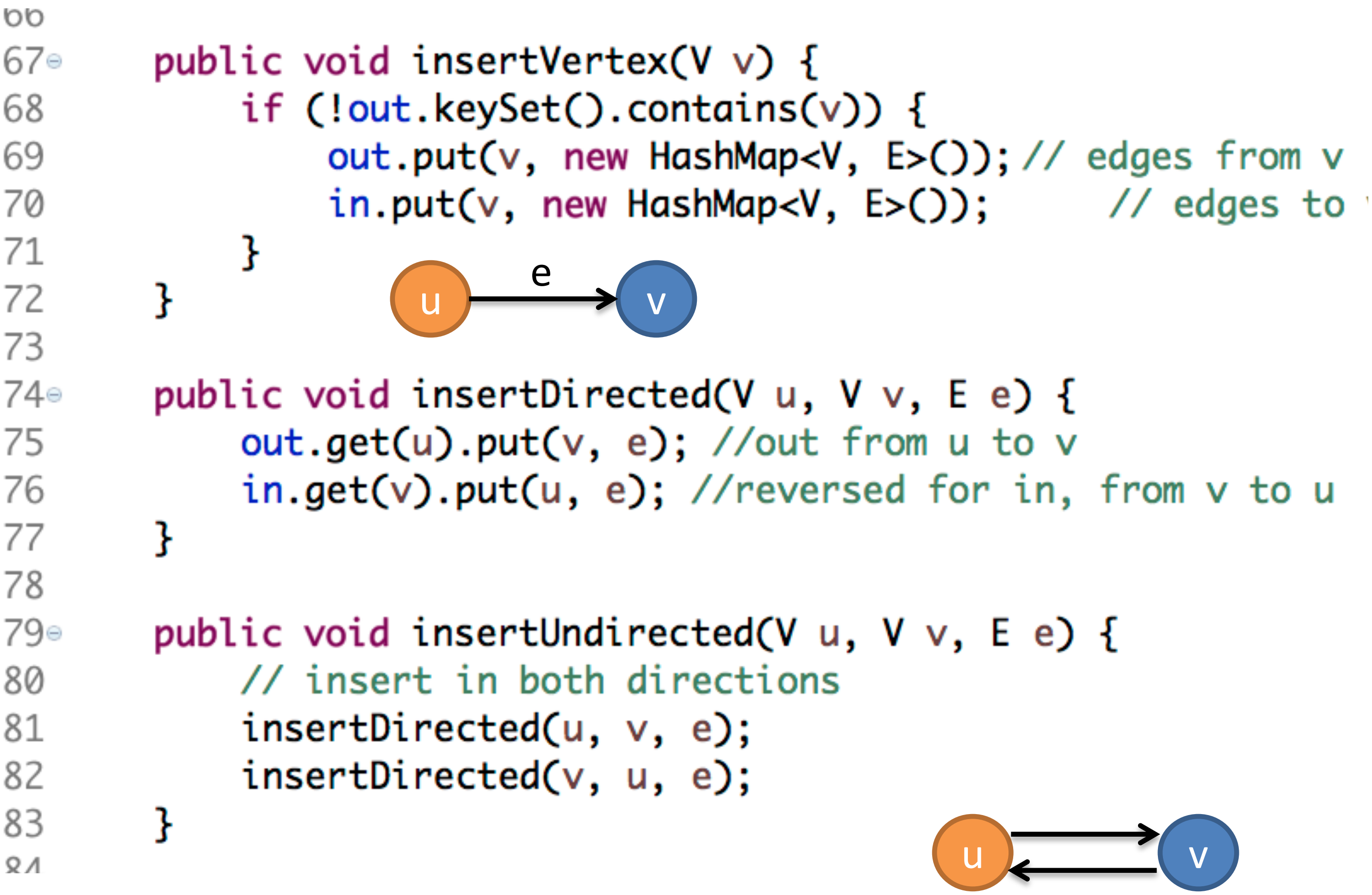
Friend

vertex

edge

Inserting vertices and edges requires updating both *in* and *out*

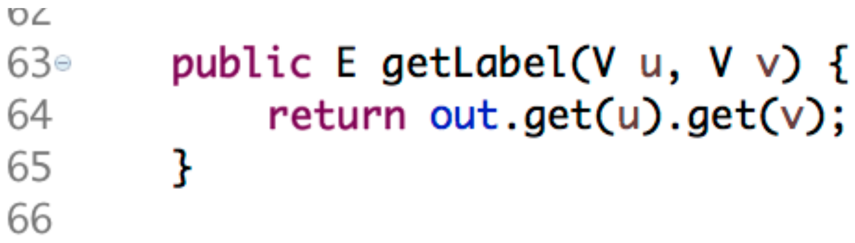
AdjacencyMapGraph.java



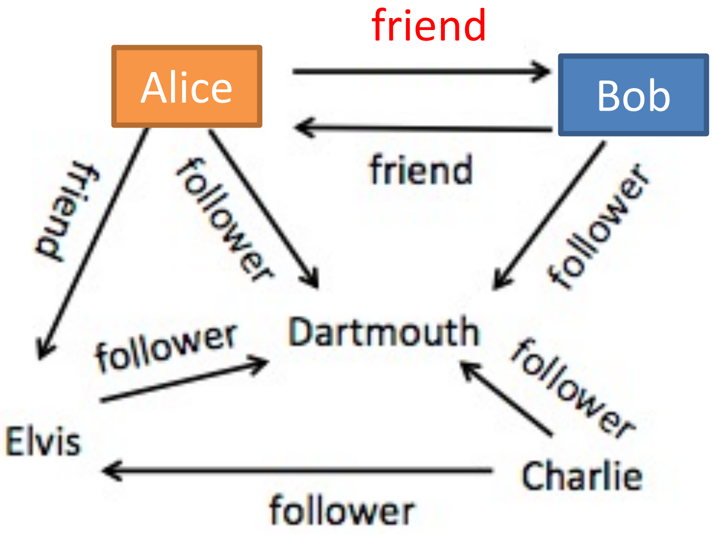
```
public void insertVertex(V v) {
    if (!out.keySet().contains(v)) {
        out.put(v, new HashMap<V, E>()); // edges from v
        in.put(v, new HashMap<V, E>());      // edges to
    }
}

public void insertDirected(V u, V v, E e) {
    out.get(u).put(v, e); //out from u to v
    in.get(v).put(u, e); //reversed for in, from v to u
}

public void insertUndirected(V u, V v, E e) {
    // insert in both directions
    insertDirected(u, v, e);
    insertDirected(v, u, e);
}
```

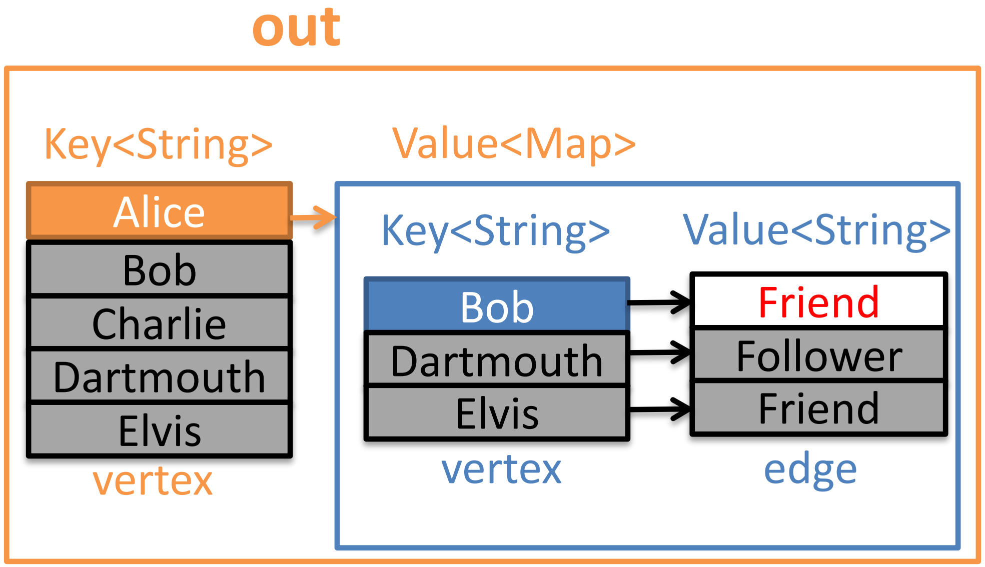
getLabel(u,v) returns the label on the edge between *u* and *v*

AdjacencyMapGraph.java



```
    public E getLabel(V u, V v) {
        return out.get(u).get(v);
    }
```





out

Key<String>

Value<Map>

Alice

Bob

Charlie

Dartmouth

Elvis

vertex

Key<String>

Value<String>

Bob → Friend

Dartmouth → Follower

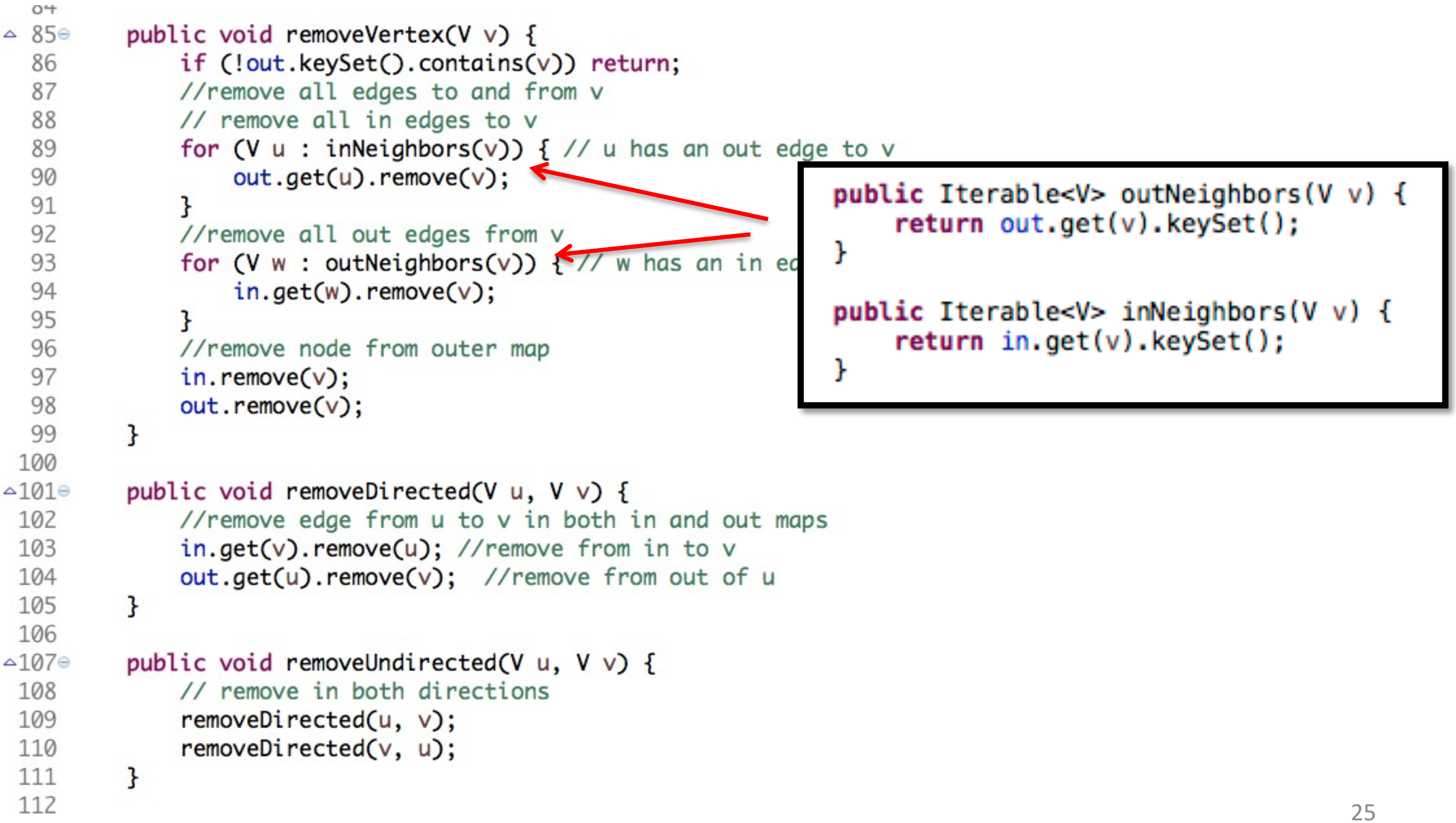
Elvis → Friend

vertex

edge

When removing edges and vertices, must remove from both *in* and *out* Maps

AdjacencyMapGraph.java

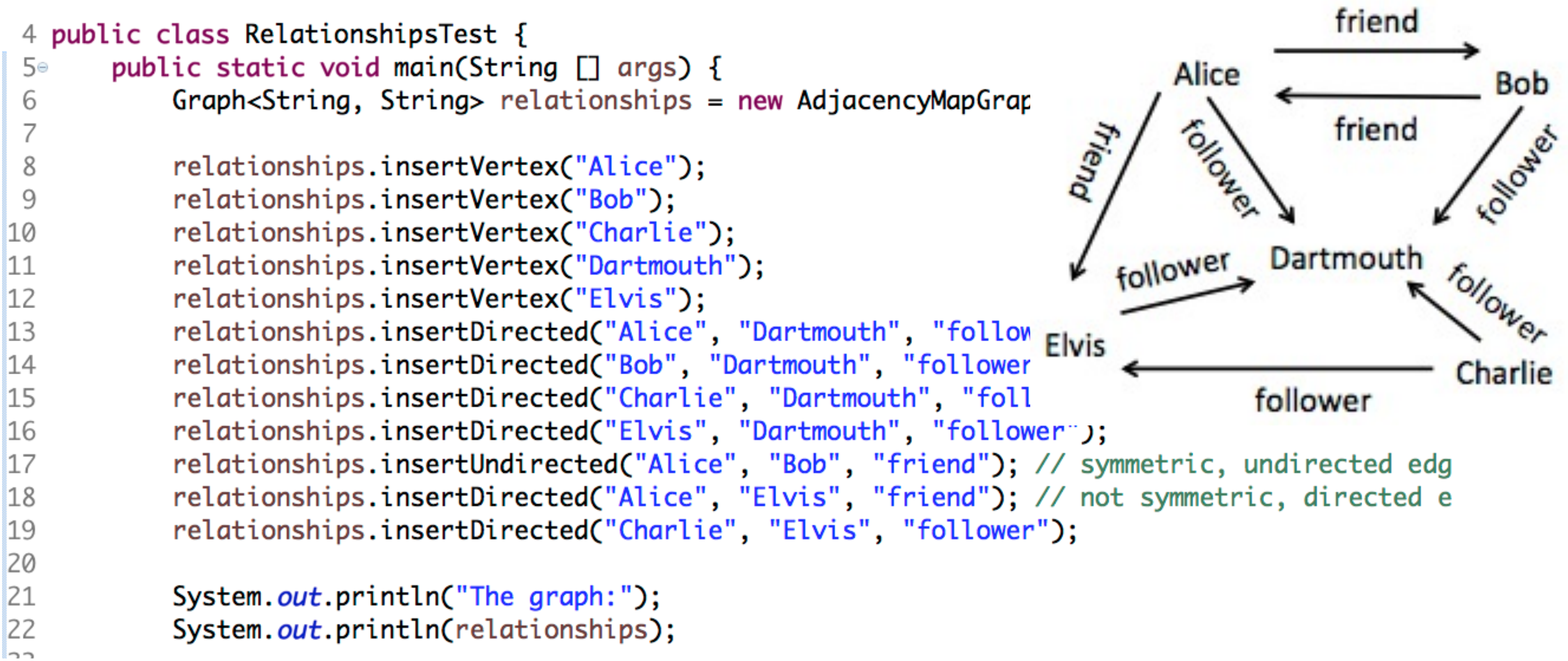


```java
public void removeVertex(V v) {
    if (!out.keySet().contains(v)) return;
    //remove all edges to and from v
    // remove all in edges to v
    for (V u : inNeighbors(v)) { // u has an out edge to v
        out.get(u).remove(v);
    }
    //remove all out edges from v
    for (V w : outNeighbors(v)) { // w has an in ed
        in.get(w).remove(v);
    }
    //remove node from outer map
    in.remove(v);
    out.remove(v);
}

public void removeDirected(V u, V v) {
    //remove edge from u to v in both in and out maps
    in.get(v).remove(u); //remove from in to v
    out.get(u).remove(v);  //remove from out of u
}

public void removeUndirected(V u, V v) {
    // remove in both directions
    removeDirected(u, v);
    removeDirected(v, u);
}
```

```java
public Iterable<V> outNeighbors(V v) {
    return out.get(v).keySet();
}

public Iterable<V> inNeighbors(V v) {
    return in.get(v).keySet();
}
```

RelationshipTest.java: create graph with both directed and non-directed edges

RelationshipTest.java



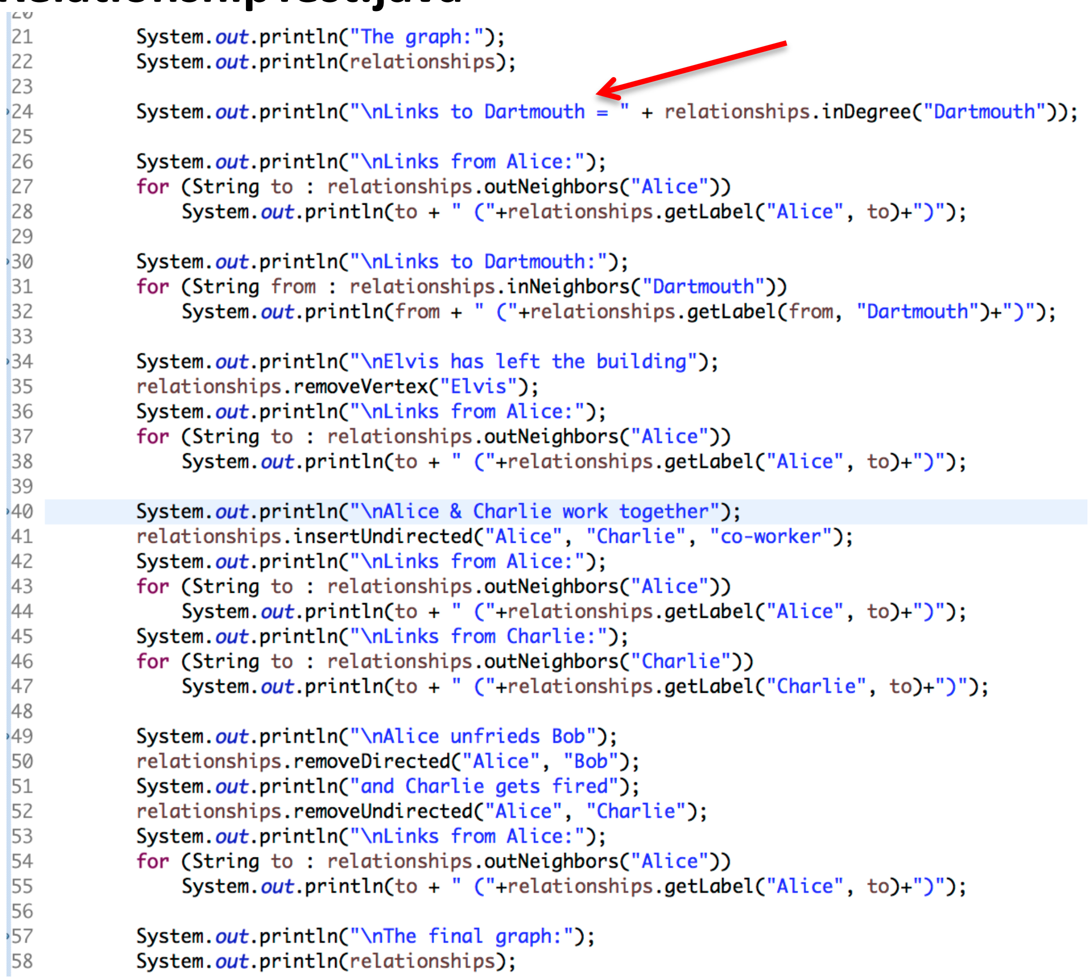
```
public class RelationshipsTest {
    public static void main(String [] args) {
        Graph<String, String> relationships = new AdjacencyMapGrap

        relationships.insertVertex("Alice");
        relationships.insertVertex("Bob");
        relationships.insertVertex("Charlie");
        relationships.insertVertex("Dartmouth");
        relationships.insertVertex("Elvis");
        relationships.insertDirected("Alice", "Dartmouth", "follow
        relationships.insertDirected("Bob", "Dartmouth", "follower
        relationships.insertDirected("Charlie", "Dartmouth", "foll
        relationships.insertDirected("Elvis", "Dartmouth", "follower");
        relationships.insertUndirected("Alice", "Bob", "friend"); // symmetric, undirected edg
        relationships.insertDirected("Alice", "Elvis", "friend"); // not symmetric, directed e
        relationships.insertDirected("Charlie", "Elvis", "follower");

        System.out.println("The graph:");
        System.out.println(relationships);
```

Output (from implicit *toString()* call):

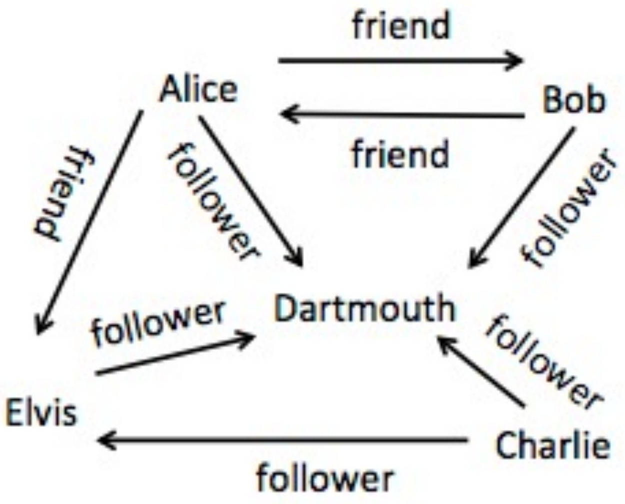
```
The graph:
Vertices: [Bob, Dartmouth, Alice, Elvis, Charlie]
Out edges: {Bob={Dartmouth=follower, Alice=friend}, Dartmouth={},
Alice={Dartmouth=follower, Bob=friend, Elvis=friend}, Elvis={Dartmouth=follower},
Charlie={Dartmouth=follower, Elvis=follower}}
```

RelationshipTest.java: create graph with both directed and non-directed edges

RelationshipTest.java



```
        System.out.println("The graph:");
        System.out.println(relationships);

        System.out.println("\nLinks to Dartmouth = " + relationships.inDegree("Dartmouth"));

        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nLinks to Dartmouth:");
        for (String from : relationships.inNeighbors("Dartmouth"))
            System.out.println(from + " ("+relationships.getLabel(from, "Dartmouth")+")");

        System.out.println("\nElvis has left the building");
        relationships.removeVertex("Elvis");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nAlice & Charlie work together");
        relationships.insertUndirected("Alice", "Charlie", "co-worker");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");
        System.out.println("\nLinks from Charlie:");
        for (String to : relationships.outNeighbors("Charlie"))
            System.out.println(to + " ("+relationships.getLabel("Charlie", to)+")");

        System.out.println("\nAlice unfrieds Bob");
        relationships.removeDirected("Alice", "Bob");
        System.out.println("and Charlie gets fired");
        relationships.removeUndirected("Alice", "Charlie");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nThe final graph:");
        System.out.println(relationships);
```

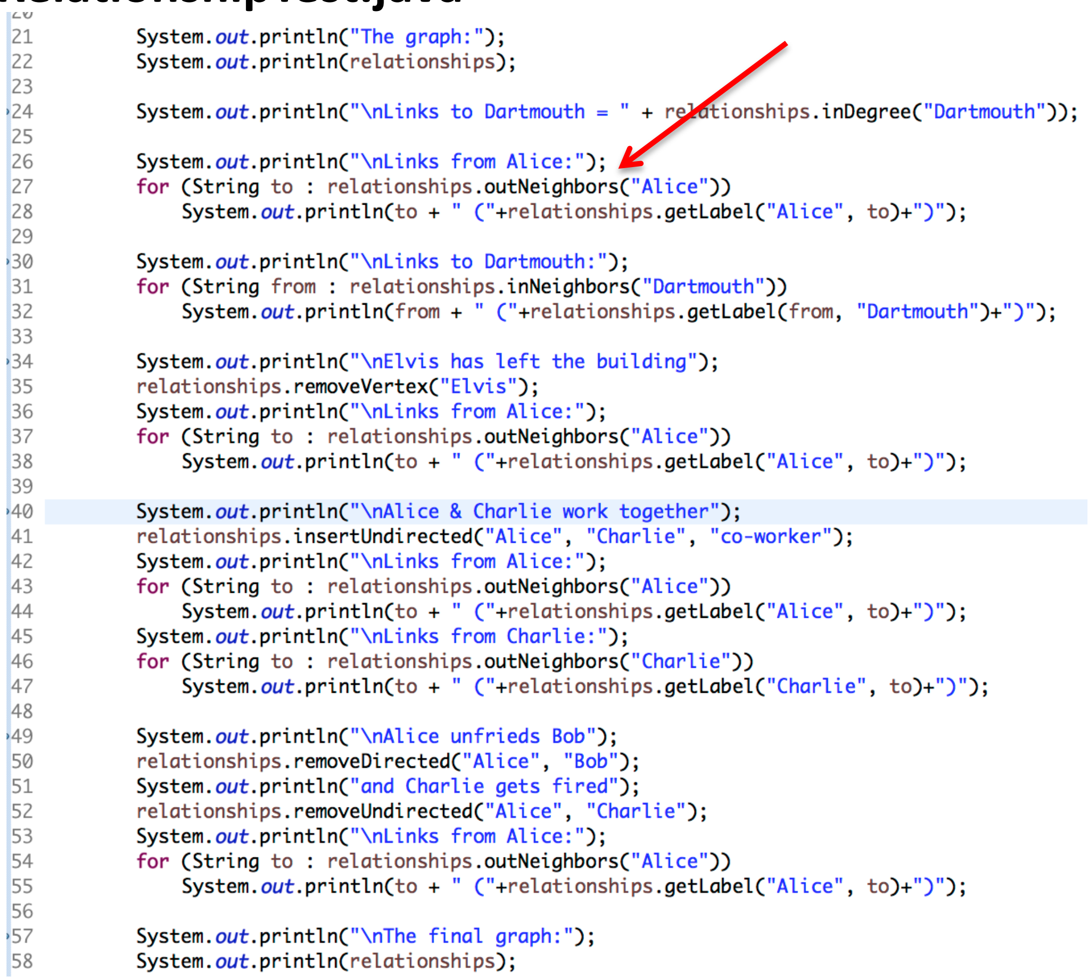


Output:

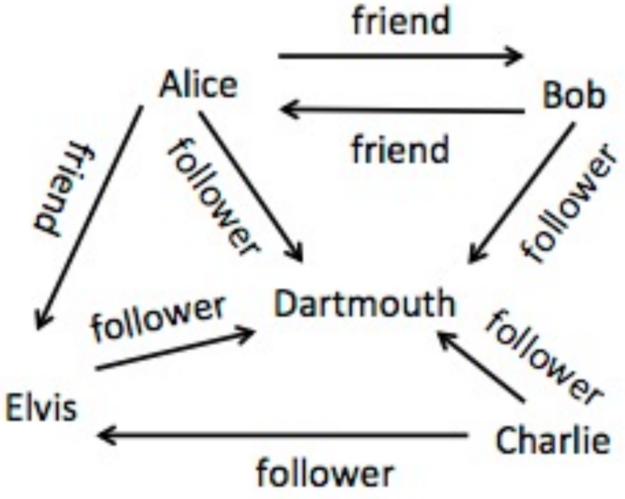
Links to Dartmouth = 4

RelationshipTest.java: create graph with both directed and non-directed edges

RelationshipTest.java



```java
        System.out.println("The graph:");
        System.out.println(relationships);

        System.out.println("\nLinks to Dartmouth = " + relationships.inDegree("Dartmouth"));

        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nLinks to Dartmouth:");
        for (String from : relationships.inNeighbors("Dartmouth"))
            System.out.println(from + " ("+relationships.getLabel(from, "Dartmouth")+")");

        System.out.println("\nElvis has left the building");
        relationships.removeVertex("Elvis");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nAlice & Charlie work together");
        relationships.insertUndirected("Alice", "Charlie", "co-worker");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");
        System.out.println("\nLinks from Charlie:");
        for (String to : relationships.outNeighbors("Charlie"))
            System.out.println(to + " ("+relationships.getLabel("Charlie", to)+")");

        System.out.println("\nAlice unfrieds Bob");
        relationships.removeDirected("Alice", "Bob");
        System.out.println("and Charlie gets fired");
        relationships.removeUndirected("Alice", "Charlie");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nThe final graph:");
        System.out.println(relationships);
```



Output:

Links from Alice:

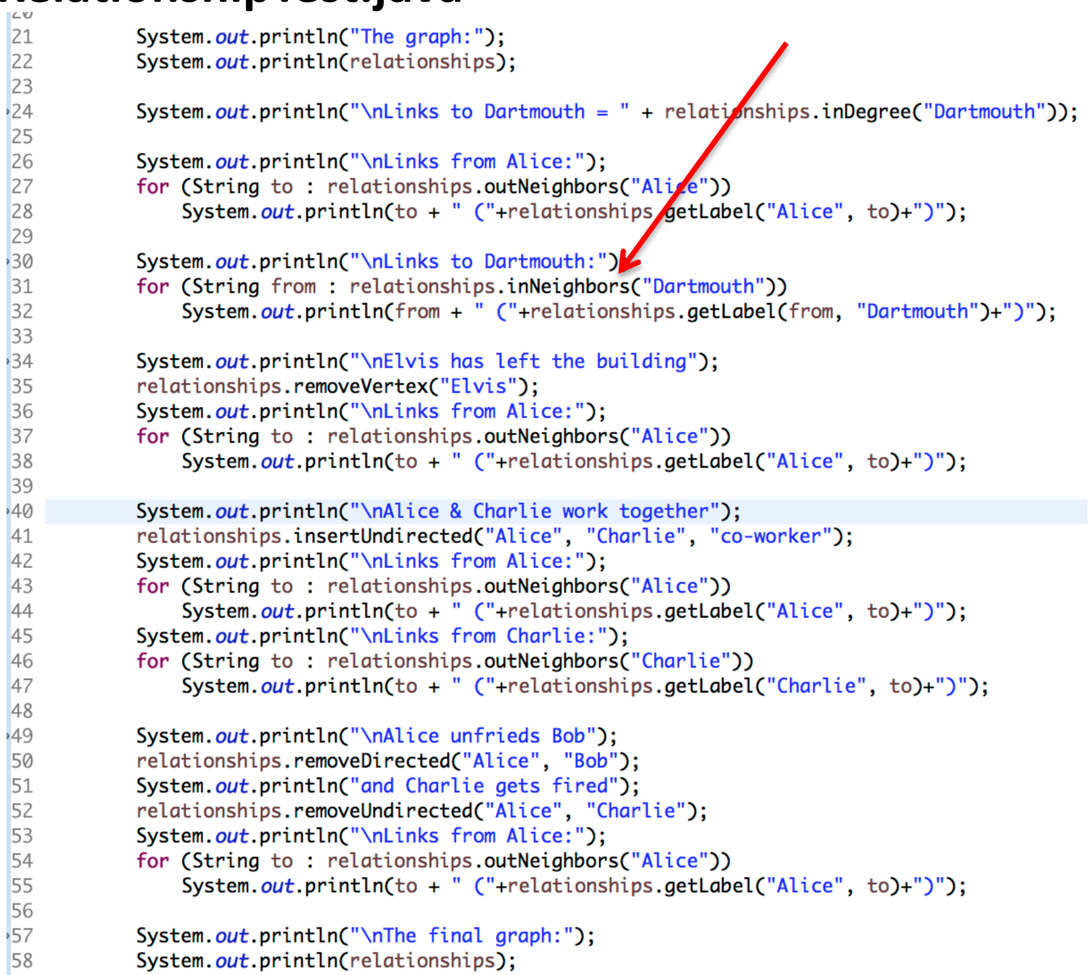
Dartmouth (follower)

Bob (friend)

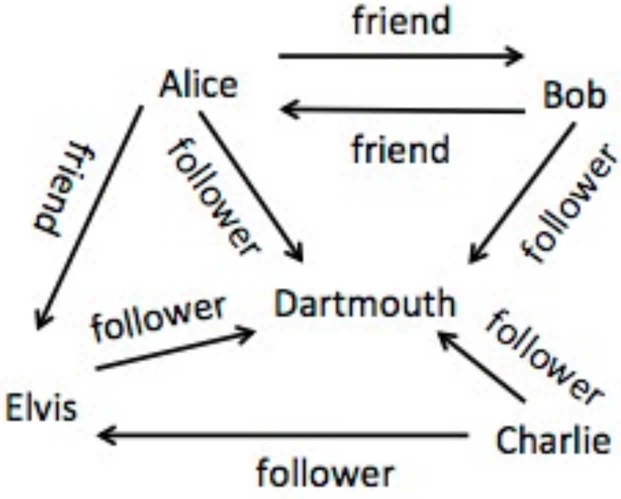
Elvis (friend)

RelationshipTest.java: create graph with both directed and non-directed edges

RelationshipTest.java



```java
        System.out.println("The graph:");
        System.out.println(relationships);

        System.out.println("\nLinks to Dartmouth = " + relationships.inDegree("Dartmouth"));

        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nLinks to Dartmouth:");
        for (String from : relationships.inNeighbors("Dartmouth"))
            System.out.println(from + " ("+relationships.getLabel(from, "Dartmouth")+")");

        System.out.println("\nElvis has left the building");
        relationships.removeVertex("Elvis");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nAlice & Charlie work together");
        relationships.insertUndirected("Alice", "Charlie", "co-worker");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");
        System.out.println("\nLinks from Charlie:");
        for (String to : relationships.outNeighbors("Charlie"))
            System.out.println(to + " ("+relationships.getLabel("Charlie", to)+")");

        System.out.println("\nAlice unfrieds Bob");
        relationships.removeDirected("Alice", "Bob");
        System.out.println("and Charlie gets fired");
        relationships.removeUndirected("Alice", "Charlie");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nThe final graph:");
        System.out.println(relationships);
```

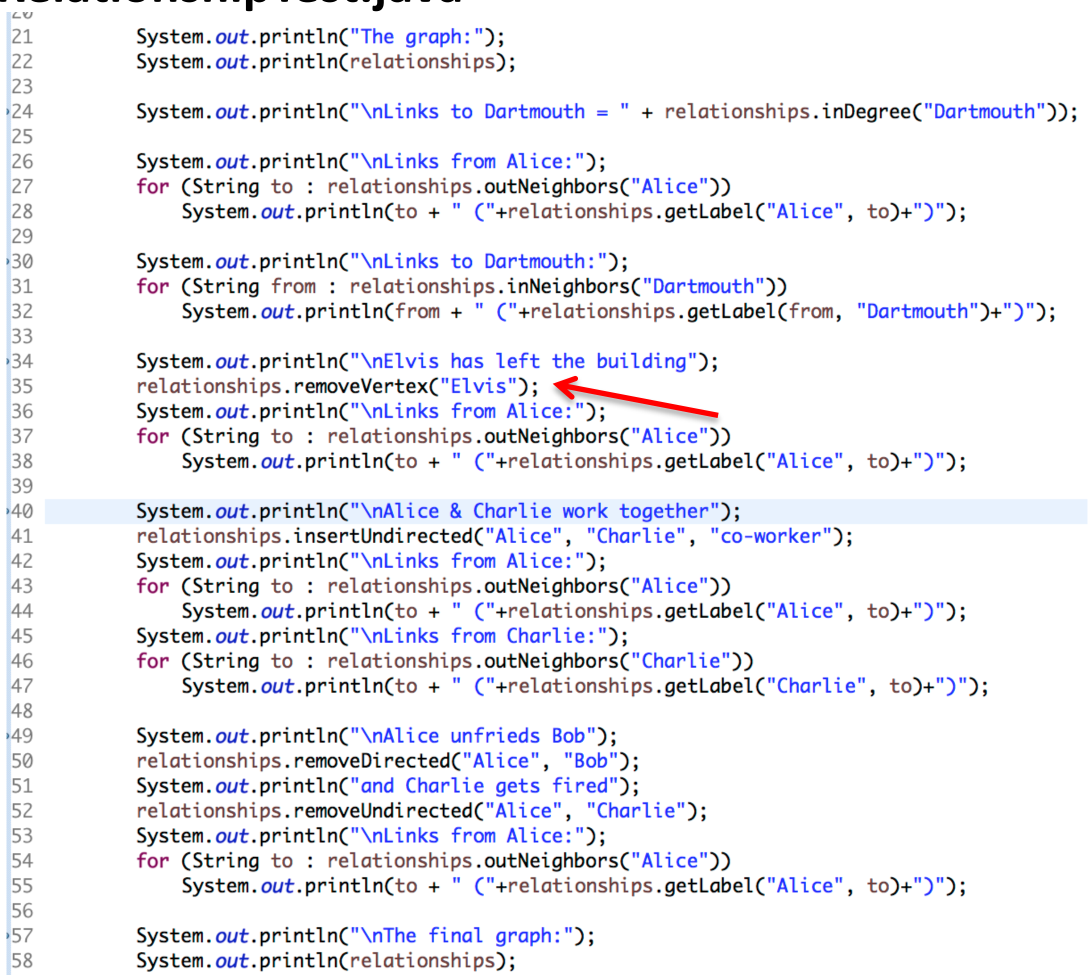


Output:

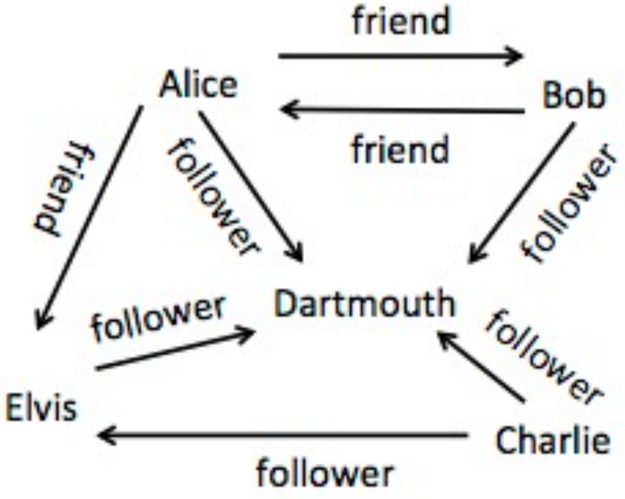
```
Links to Dartmouth:
Bob (follower)
Alice (follower)
Elvis (follower)
Charlie (follower)
```

RelationshipTest.java: create graph with both directed and non-directed edges

RelationshipTest.java

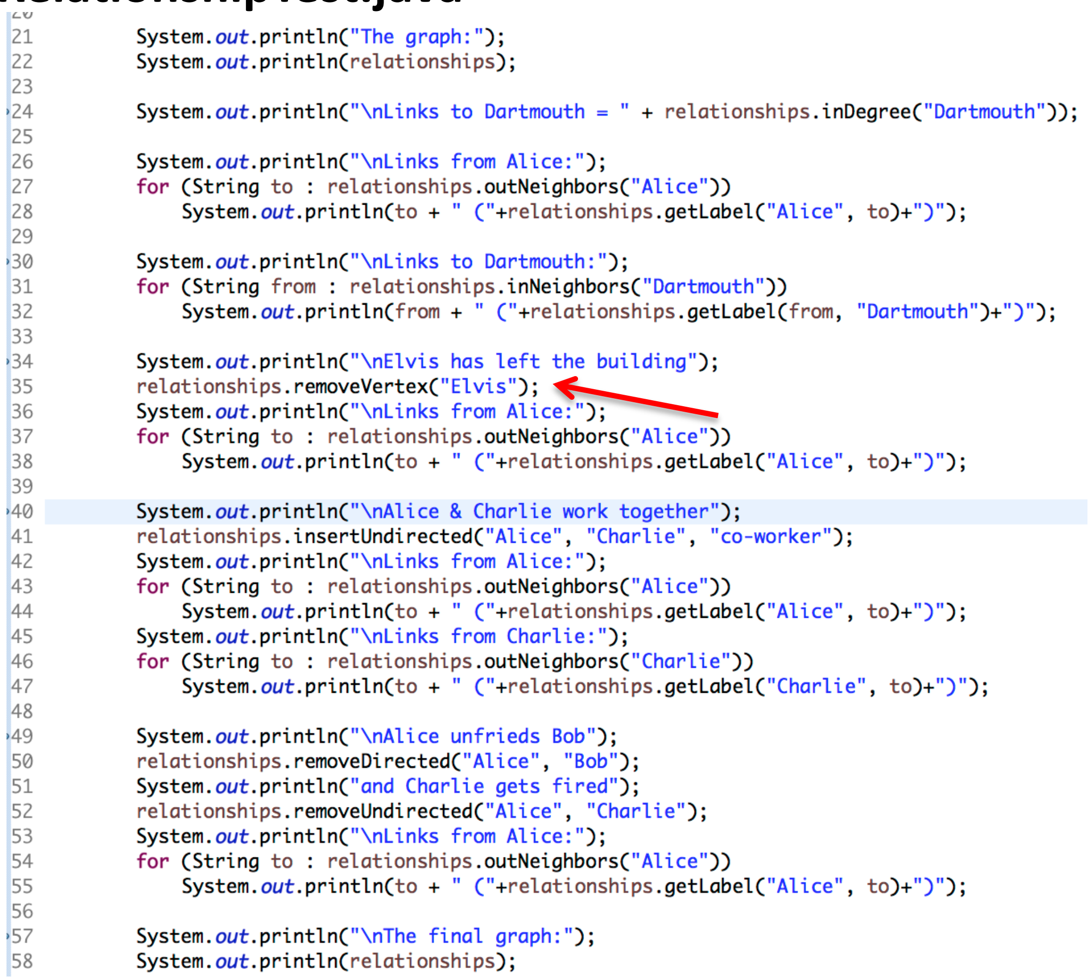


```java
        System.out.println("The graph:");
        System.out.println(relationships);

        System.out.println("\nLinks to Dartmouth = " + relationships.inDegree("Dartmouth"));

        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nLinks to Dartmouth:");
        for (String from : relationships.inNeighbors("Dartmouth"))
            System.out.println(from + " ("+relationships.getLabel(from, "Dartmouth")+")");

        System.out.println("\nElvis has left the building");
        relationships.removeVertex("Elvis");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nAlice & Charlie work together");
        relationships.insertUndirected("Alice", "Charlie", "co-worker");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");
        System.out.println("\nLinks from Charlie:");
        for (String to : relationships.outNeighbors("Charlie"))
            System.out.println(to + " ("+relationships.getLabel("Charlie", to)+")");

        System.out.println("\nAlice unfrieds Bob");
        relationships.removeDirected("Alice", "Bob");
        System.out.println("and Charlie gets fired");
        relationships.removeUndirected("Alice", "Charlie");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nThe final graph:");
        System.out.println(relationships);
```

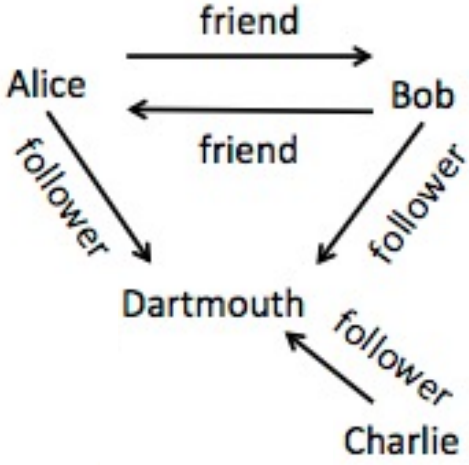


RelationshipTest.java: create graph with both directed and non-directed edges

RelationshipTest.java



```java
        System.out.println("The graph:");
        System.out.println(relationships);

        System.out.println("\nLinks to Dartmouth = " + relationships.inDegree("Dartmouth"));

        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nLinks to Dartmouth:");
        for (String from : relationships.inNeighbors("Dartmouth"))
            System.out.println(from + " ("+relationships.getLabel(from, "Dartmouth")+")");

        System.out.println("\nElvis has left the building");
        relationships.removeVertex("Elvis");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nAlice & Charlie work together");
        relationships.insertUndirected("Alice", "Charlie", "co-worker");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");
        System.out.println("\nLinks from Charlie:");
        for (String to : relationships.outNeighbors("Charlie"))
            System.out.println(to + " ("+relationships.getLabel("Charlie", to)+")");

        System.out.println("\nAlice unfrieds Bob");
        relationships.removeDirected("Alice", "Bob");
        System.out.println("and Charlie gets fired");
        relationships.removeUndirected("Alice", "Charlie");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nThe final graph:");
        System.out.println(relationships);
```



Output:

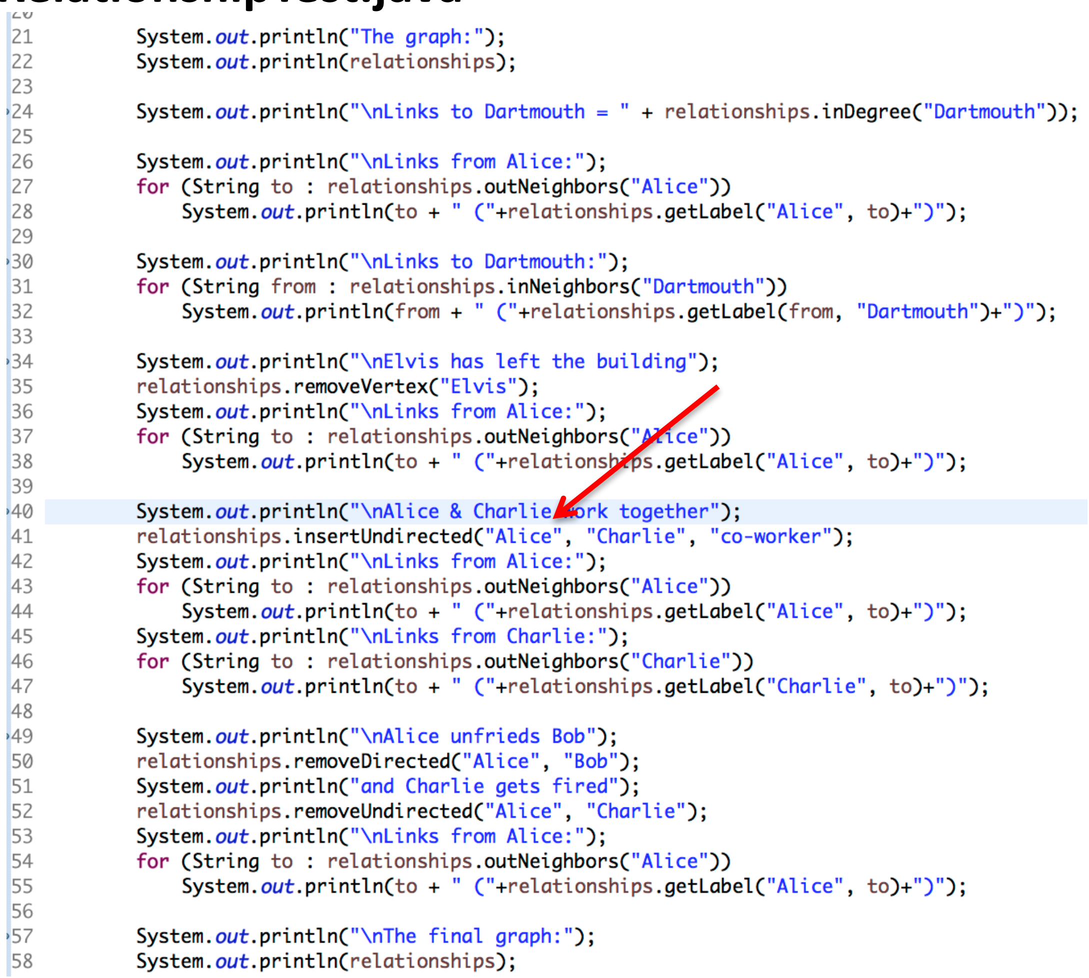
Links from Alice:

Dartmouth (follower)

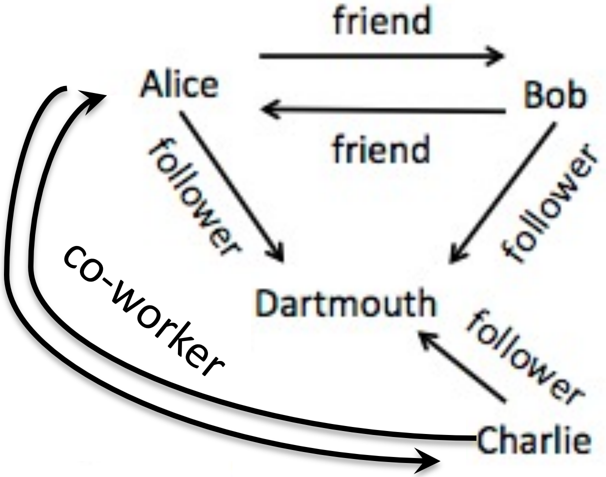
Bob (friend)

RelationshipTest.java: create graph with both directed and non-directed edges

RelationshipTest.java



```java
        System.out.println("The graph:");
        System.out.println(relationships);

        System.out.println("\nLinks to Dartmouth = " + relationships.inDegree("Dartmouth"));

        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nLinks to Dartmouth:");
        for (String from : relationships.inNeighbors("Dartmouth"))
            System.out.println(from + " ("+relationships.getLabel(from, "Dartmouth")+")");

        System.out.println("\nElvis has left the building");
        relationships.removeVertex("Elvis");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nAlice & Charlie work together");
        relationships.insertUndirected("Alice", "Charlie", "co-worker");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");
        System.out.println("\nLinks from Charlie:");
        for (String to : relationships.outNeighbors("Charlie"))
            System.out.println(to + " ("+relationships.getLabel("Charlie", to)+")");

        System.out.println("\nAlice unfrieds Bob");
        relationships.removeDirected("Alice", "Bob");
        System.out.println("and Charlie gets fired");
        relationships.removeUndirected("Alice", "Charlie");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nThe final graph:");
        System.out.println(relationships);
```



Output:

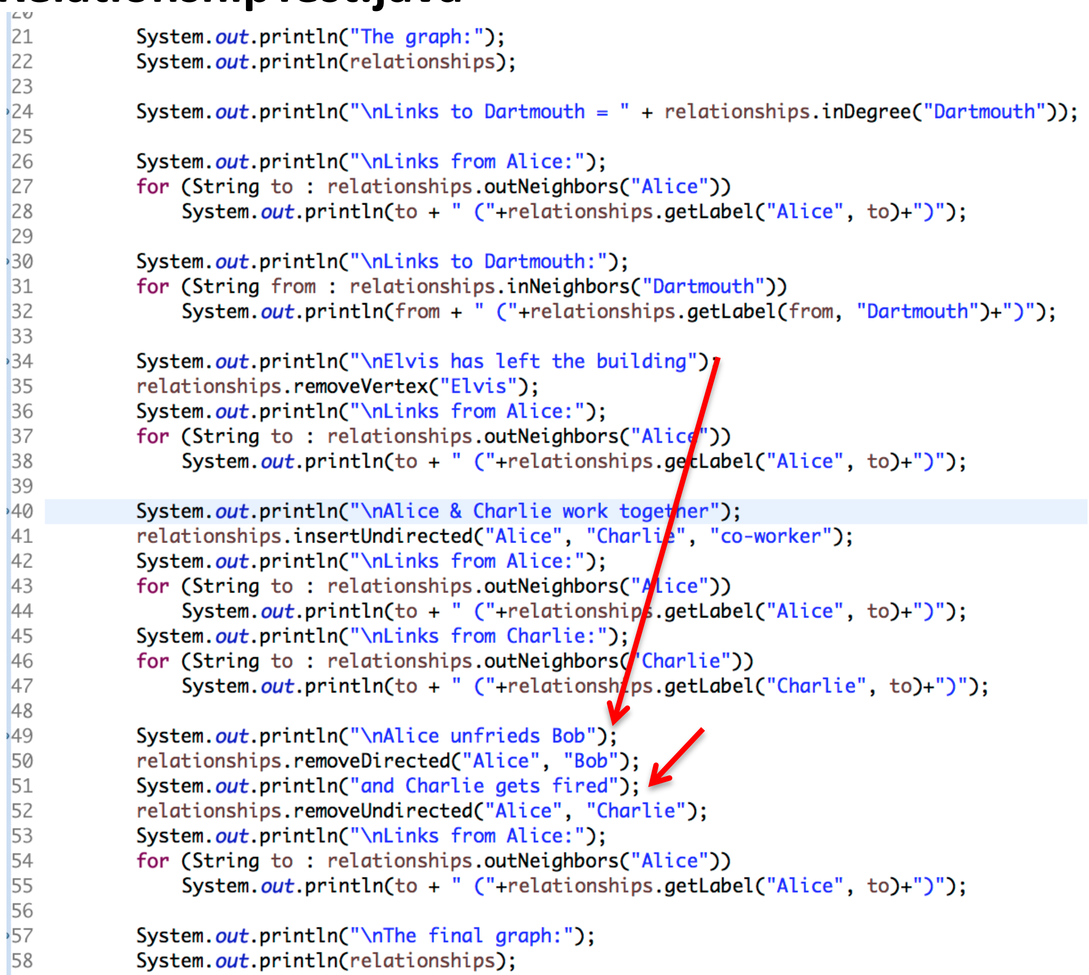
Alice & Charlie work together

Links from Alice:
Dartmouth (follower)
Bob (friend)
Charlie (co-worker)

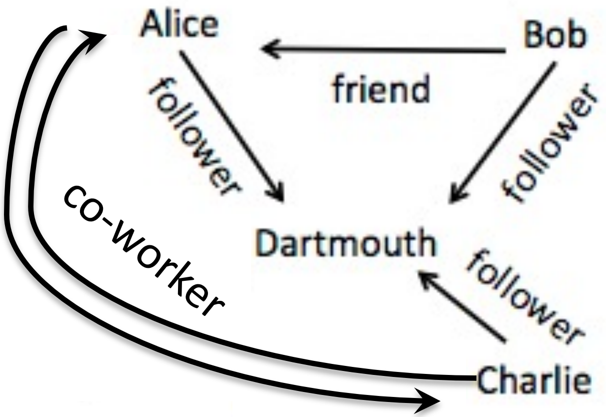
Links from Charlie:
Dartmouth (follower)
Alice (co-worker)

RelationshipTest.java: create graph with both directed and non-directed edges

RelationshipTest.java



```java
        System.out.println("The graph:");
        System.out.println(relationships);

        System.out.println("\nLinks to Dartmouth = " + relationships.inDegree("Dartmouth"));

        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nLinks to Dartmouth:");
        for (String from : relationships.inNeighbors("Dartmouth"))
            System.out.println(from + " ("+relationships.getLabel(from, "Dartmouth")+")");

        System.out.println("\nElvis has left the building");
        relationships.removeVertex("Elvis");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nAlice & Charlie work together");
        relationships.insertUndirected("Alice", "Charlie", "co-worker");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");
        System.out.println("\nLinks from Charlie:");
        for (String to : relationships.outNeighbors("Charlie"))
            System.out.println(to + " ("+relationships.getLabel("Charlie", to)+")");

        System.out.println("\nAlice unfrieds Bob");
        relationships.removeDirected("Alice", "Bob");
        System.out.println("and Charlie gets fired");
        relationships.removeUndirected("Alice", "Charlie");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nThe final graph:");
        System.out.println(relationships);
```



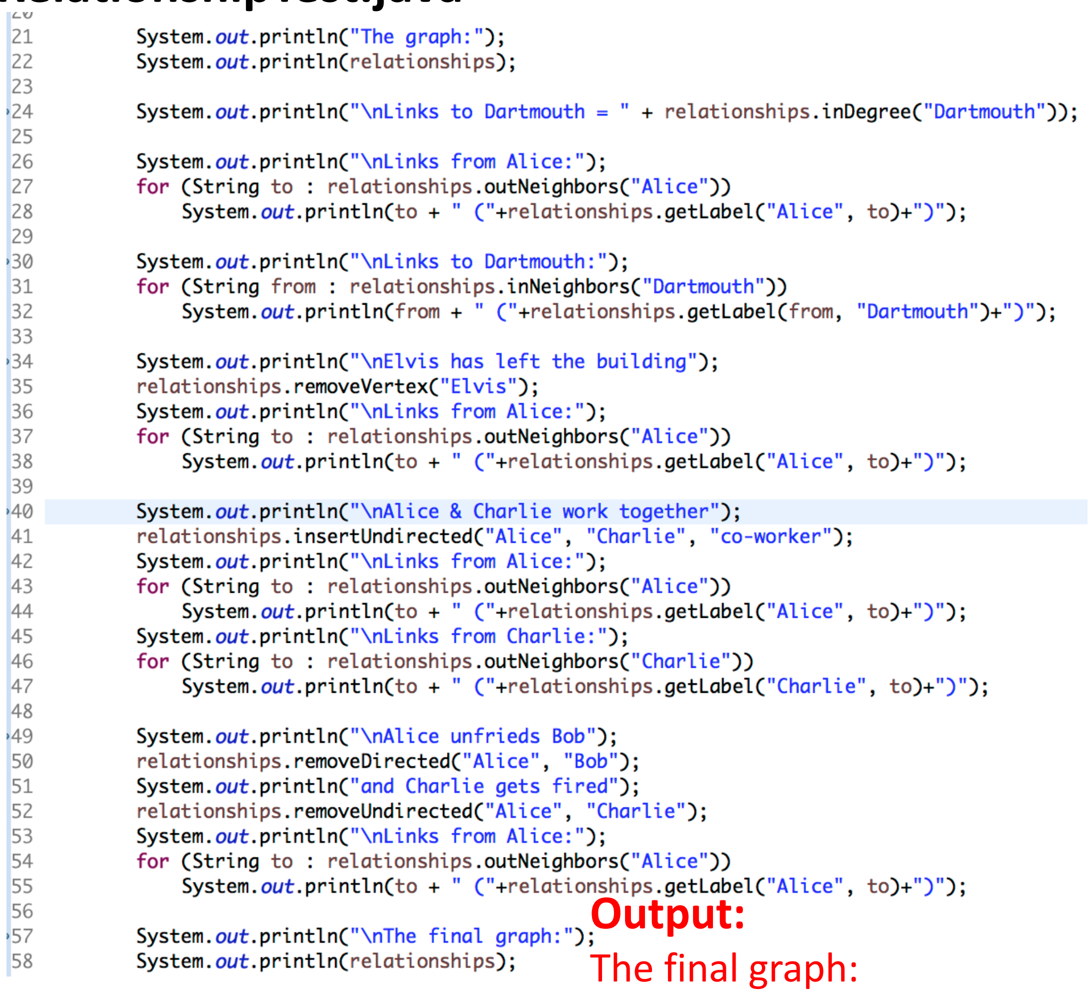
Output:

Alice unfriends Bob
and Charlie gets fired

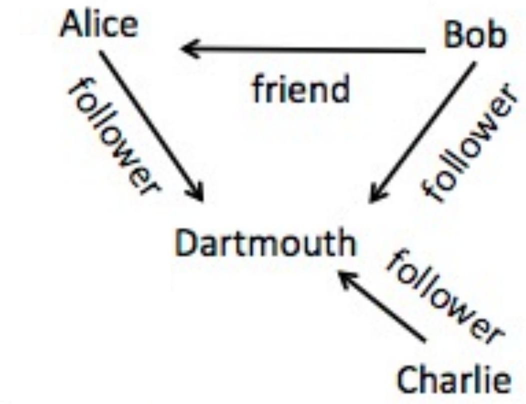
Links from Alice:
Dartmouth (follower)

RelationshipTest.java: create graph with both directed and non-directed edges

RelationshipTest.java



```
        System.out.println("The graph:");
        System.out.println(relationships);

        System.out.println("\nLinks to Dartmouth = " + relationships.inDegree("Dartmouth"));

        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nLinks to Dartmouth:");
        for (String from : relationships.inNeighbors("Dartmouth"))
            System.out.println(from + " ("+relationships.getLabel(from, "Dartmouth")+")");

        System.out.println("\nElvis has left the building");
        relationships.removeVertex("Elvis");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nAlice & Charlie work together");
        relationships.insertUndirected("Alice", "Charlie", "co-worker");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");
        System.out.println("\nLinks from Charlie:");
        for (String to : relationships.outNeighbors("Charlie"))
            System.out.println(to + " ("+relationships.getLabel("Charlie", to)+")");

        System.out.println("\nAlice unfrieds Bob");
        relationships.removeDirected("Alice", "Bob");
        System.out.println("and Charlie gets fired");
        relationships.removeUndirected("Alice", "Charlie");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nThe final graph:");
        System.out.println(relationships);
```



Output:

```
The final graph:
Vertices: [Bob, Dartmouth, Alice, Charlie]
Out edges: {Bob={Dartmouth=follower, Alice=friend}, Dartmouth={},
Alice={Dartmouth=follower},Charlie={Dartmouth=follower}}
```

Summary

- Graphs are used to represent relationships
    - Directed vs. undirected
    - Four different implementations with pros and cons
- Implementation with adjacency map

Next

- Graph traversals

Additional Resources

Edge lists

ANNOTATED SLIDES

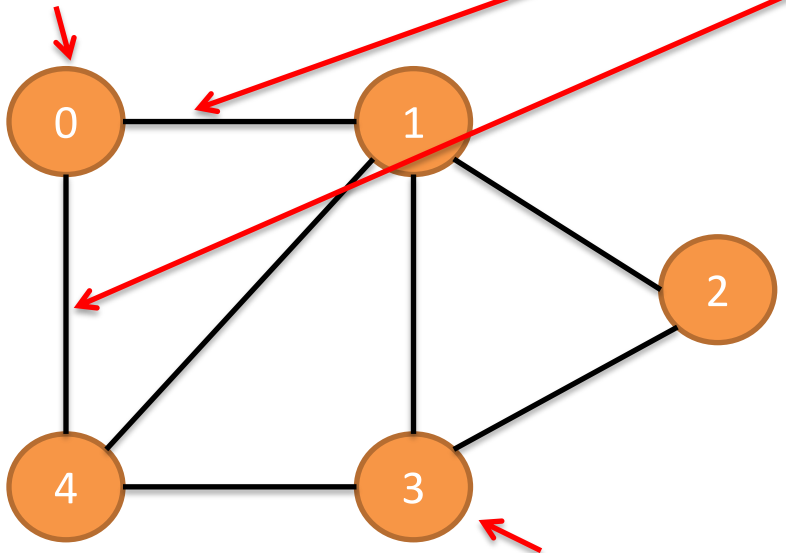
Edge Lists create an unordered list of vertex pairs where each entry is an edge

1. Edge List

List of edges

Edge list {node #, node #}

```
{ {0,1}, {0,4}, {1,2}, {1,3},
{1,4}, {2,3}, {3,4} }
```



Node 0

Node 3

Assume:

n nodes (here 5)

m edges (here 7)

Notes:

- Number nodes 0..n-1
- Edge List stores pairs of indexes that reference nodes
- Each Edge List entry represents an edge between two nodes
- *m* total entries in Edge List
- Can be ordered to show directed edges
- Insert edge fast, just add to list
- Everything else slow
- Example: `removeVertex` is O(m), have to remove all edges to/from node, so search all edges leading to or from node

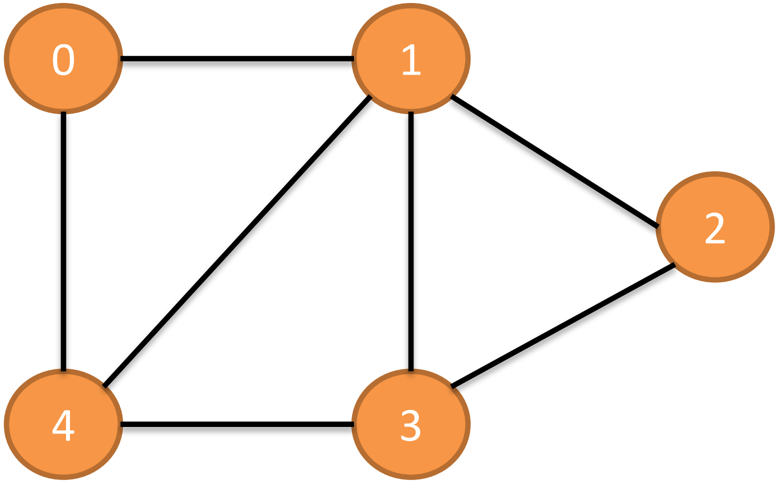
Adjacency lists

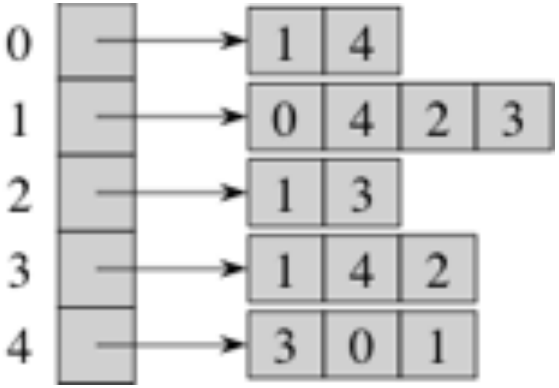
ANNOTATED SLIDES

Adjacency Lists store adjacent nodes in a List; gives improved performance

2. Adjacency List

List of Lists





0 → 1 4

1 → 0 4 2 3

2 → 1 3

3 → 1 4 2

4 → 3 0 1

Assume:

n nodes (here 5)

m edges (here 7)

Notes:

- Two vertices are said to be *adjacent* if there is an edge between them
- Store List of nodes in or out of each vertex (same if undirected graph)
- Might keep two lists, one for in neighbors and one for out neighbors
- Faster to get neighbors than Edge List, just iterate in O(degree(v)) vs. O(m)

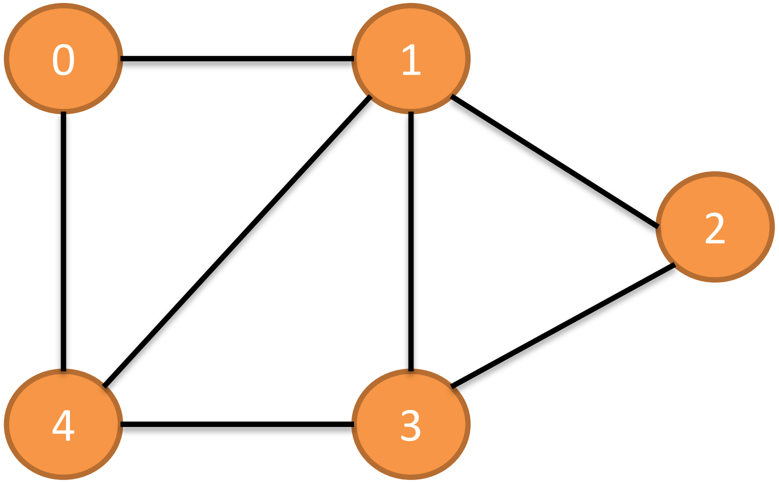
Adjacency matrix

ANNOTATED SLIDES

Adjacency Matrices create an n x n array to indicate existence of edges

3. Adjacency Matrix

n x n array



Assume:

n nodes (here 5)

m edges (here 7)

To

From

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Notes:

- Create n x n matrix A, set A[i,j] = 1 if edge from node i to node j, else 0
- Works if no parallel edges
- Undirected graph A[i,j] == A[j,i]
- `hasEdge(u,v)` is now O(1), whereas in Adjacency List it was O(degree(u))
- Finding neighbors now O(n) because have to check entire row or column
- Adding/removing vertices O(n^2), have to rebuild entire matrix

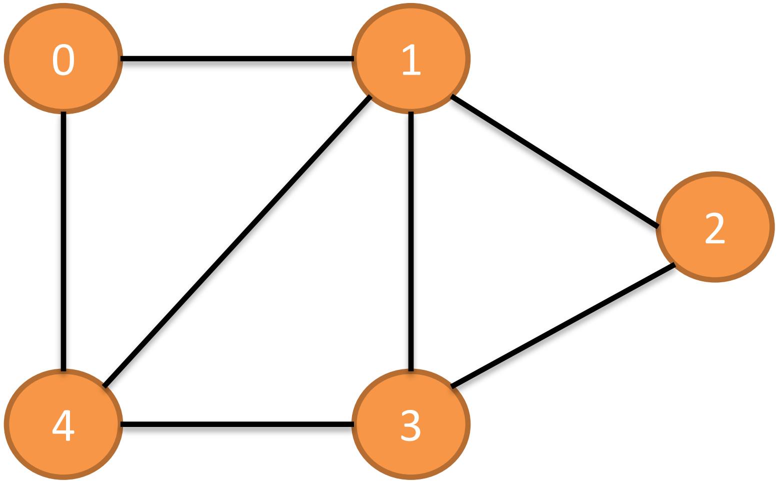
Adjacency map

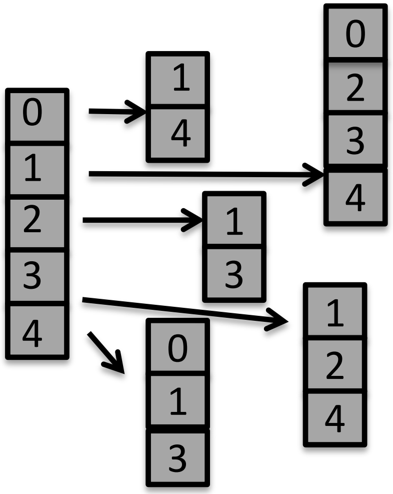
ANNOTATED SLIDES

Adjacency Maps create a Map for each node and a second Map to adjacent nodes

4. Adjacency Map

Map of Maps





Assume:

n nodes (here 5)

m edges (here 7)

Notes:

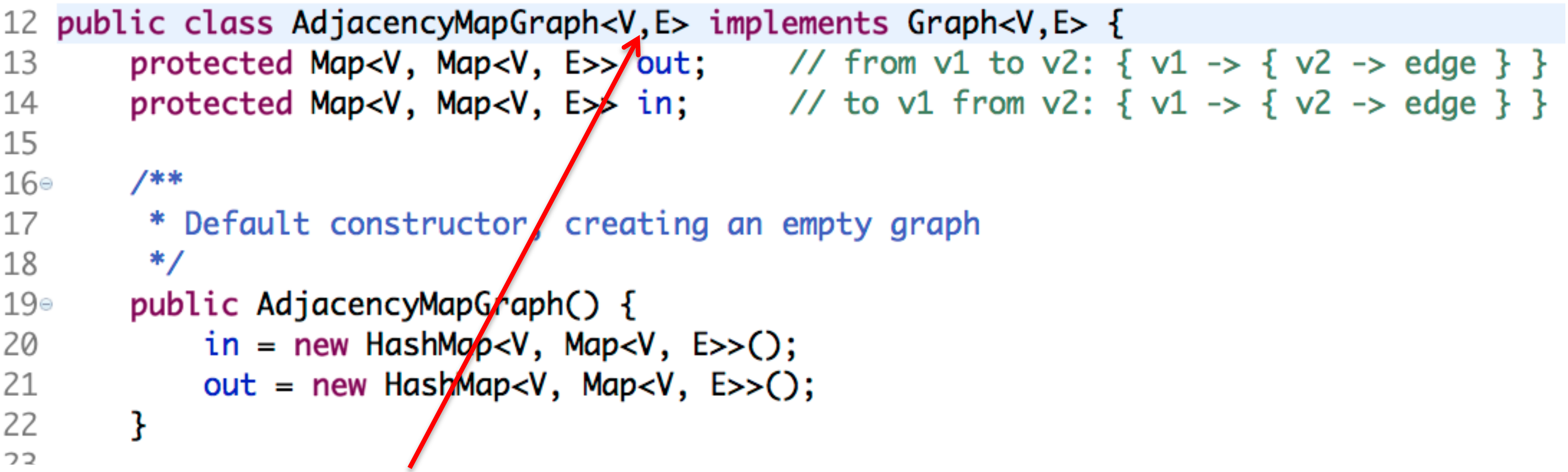
- Create Map with vertex names as Key
- Map Value is a second Map of adjacent vertices with vertex name as Key
- Value in second Map is edge label
- No need to number nodes in order
- `hasEdge(u,v)` now expected O(1)
  - Look up `u` in Map O(1)
  - Look up `v` in second Map O(1)

AdjacencyMapGraph.java

ANNOTATED SLIDES

AdjancyMapGraph.java tracks *in* and *out* edges in two different Maps

AdjacencyMapGraph.java



```
public class AdjacencyMapGraph<V,E> implements Graph<V,E> {
    protected Map<V, Map<V, E>> out;     // from v1 to v2: { v1 -> { v2 -> edge } }
    protected Map<V, Map<V, E>> in;      // to v1 from v2: { v1 -> { v2 -> edge } }

    /**
     * Default constructor, creating an empty graph
     */
    public AdjacencyMapGraph() {
        in = new HashMap<V, Map<V, E>>();
        out = new HashMap<V, Map<V, E>>();
    }
```

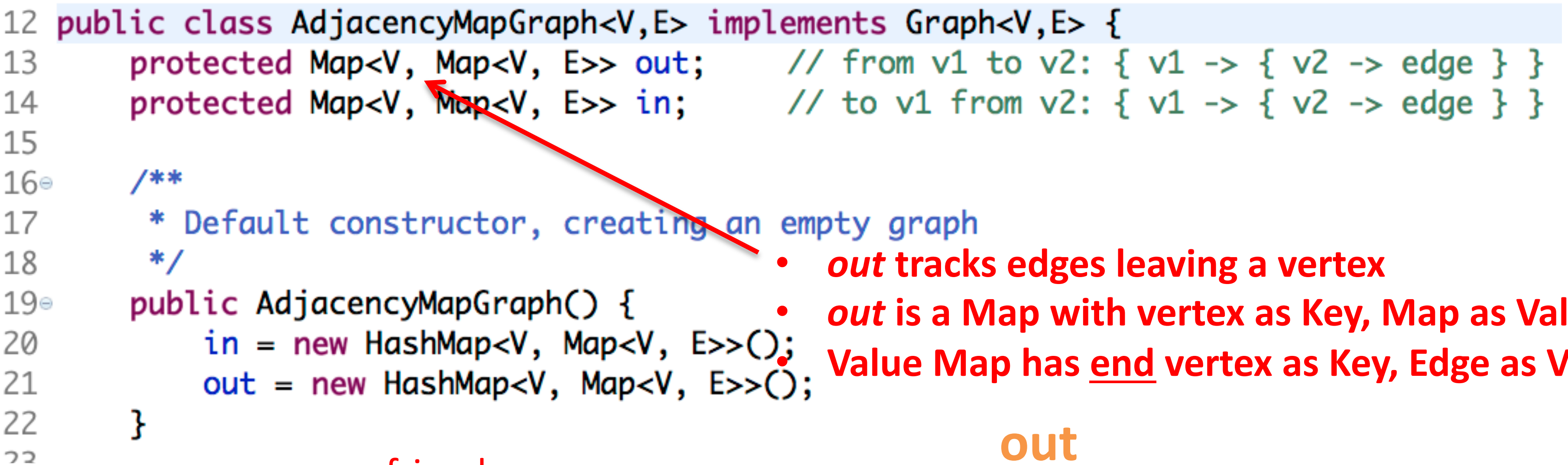
Will normally declare something like:

`Graph<String, String> relationships =` **`new AdjacencyMapGraph<String, String>();`**

Vertices V will be Strings (e.g., someone's name)
Edges E will be Strings (e.g., "follows" or "friend")

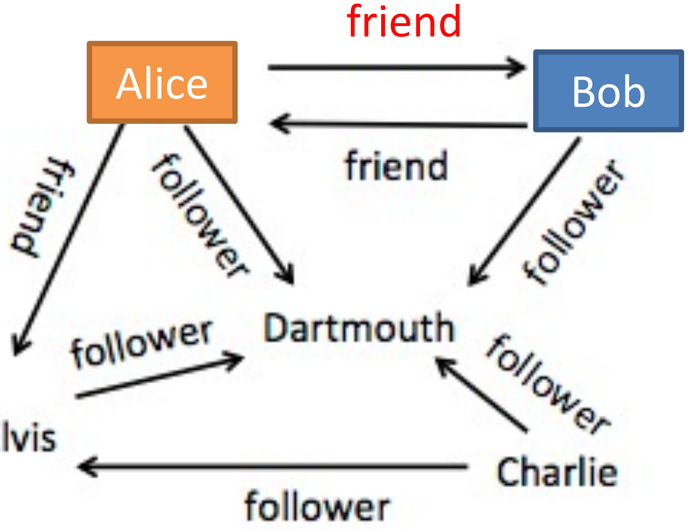
out tracks edges leaving a vertex

AdjacencyMapGraph.java

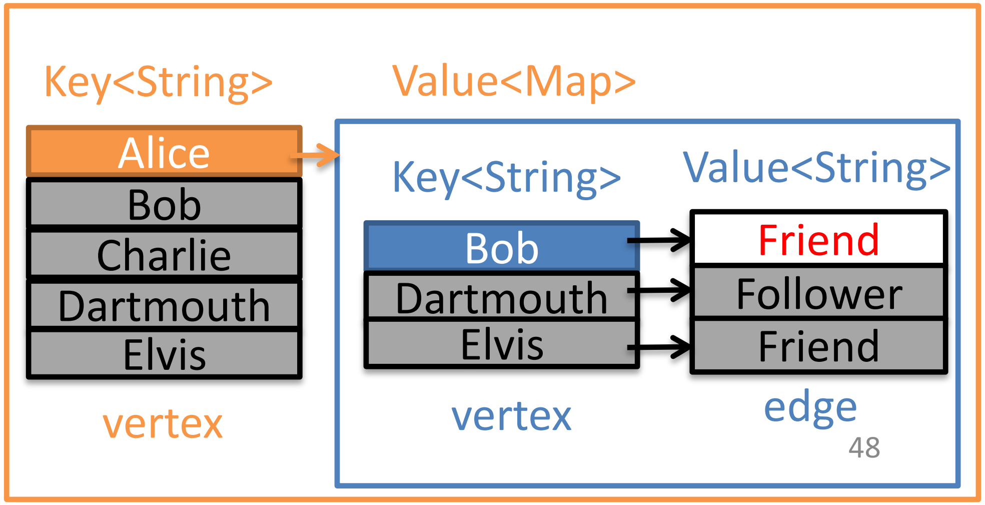


```java
public class AdjacencyMapGraph<V,E> implements Graph<V,E> {
    protected Map<V, Map<V, E>> out;     // from v1 to v2: { v1 -> { v2 -> edge } }
    protected Map<V, Map<V, E>> in;      // to v1 from v2: { v1 -> { v2 -> edge } }

    /**
     * Default constructor, creating an empty graph
     */
    public AdjacencyMapGraph() {
        in = new HashMap<V, Map<V, E>>();
        out = new HashMap<V, Map<V, E>>();
    }
```

- ***out*** **tracks edges leaving a vertex**
- ***out*** **is a Map with vertex as Key, Map as Value**
- **Value Map has end vertex as Key, Edge as Value**

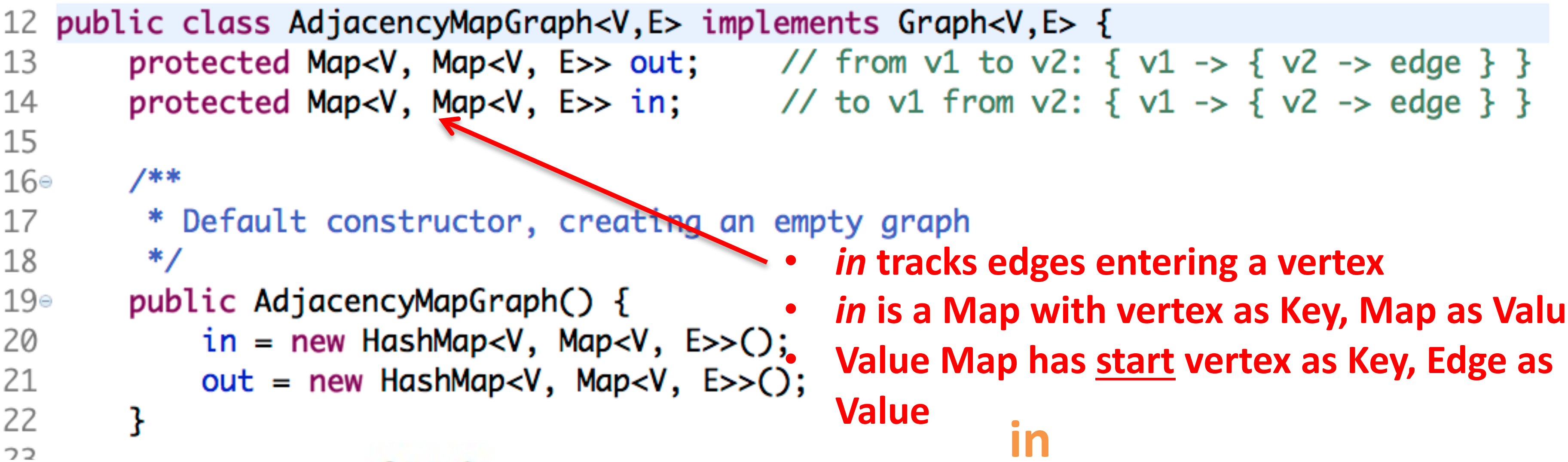


out



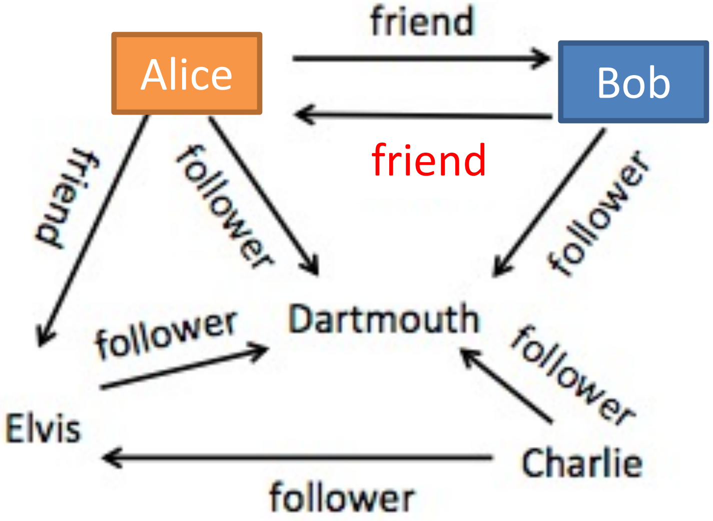
in tracks edges entering a vertex

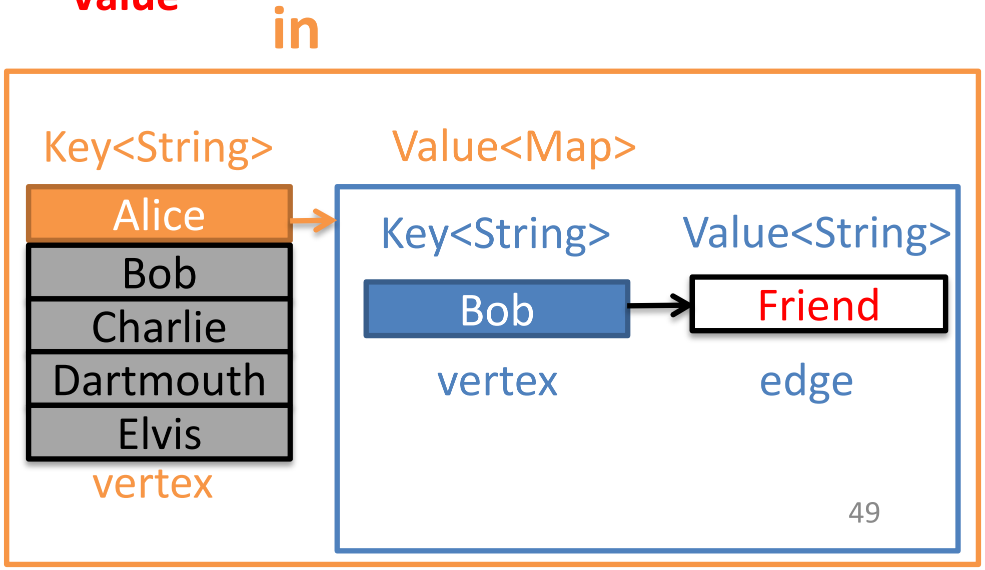
AdjacencyMapGraph.java



```
public class AdjacencyMapGraph<V,E> implements Graph<V,E> {
    protected Map<V, Map<V, E>> out;    // from v1 to v2: { v1 -> { v2 -> edge } }
    protected Map<V, Map<V, E>> in;     // to v1 from v2: { v1 -> { v2 -> edge } }

    /**
     * Default constructor, creating an empty graph
     */
    public AdjacencyMapGraph() {
        in = new HashMap<V, Map<V, E>>();
        out = new HashMap<V, Map<V, E>>();
    }
```

- ***in* tracks edges entering a vertex**
- ***in* is a Map with vertex as Key, Map as Value**
- **Value Map has start vertex as Key, Edge as Value**





in

Key<String>

Value<Map>

Key<String>
Alice
Bob
Charlie
Dartmouth
Elvis

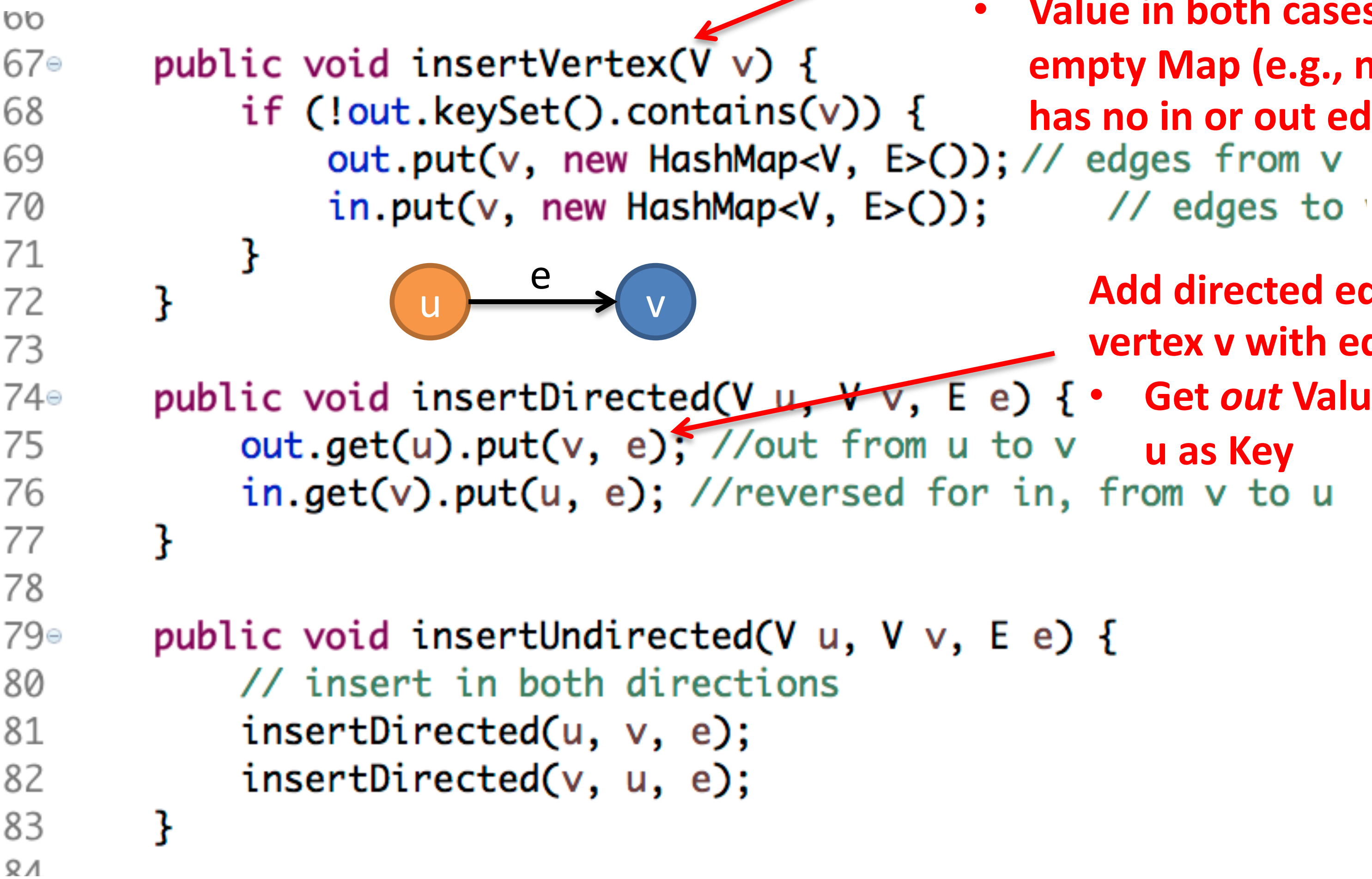
vertex

Key<String>	Value<String>
Bob	Friend
vertex	edge

Inserting vertices and edges requires updating both *in* and *out*

AdjacencyMapGraph.java

- **Adding new vertex adds Key to both *in* and *out***
- **Value in both cases is set to empty Map (e.g., new vertex has no in or out edges)**



```java
    public void insertVertex(V v) {
        if (!out.keySet().contains(v)) {
            out.put(v, new HashMap<V, E>()); // edges from v
            in.put(v, new HashMap<V, E>());      // edges to
        }
    }

    public void insertDirected(V u, V v, E e) {
        out.get(u).put(v, e); //out from u to v
        in.get(v).put(u, e); //reversed for in, from v to u
    }

    public void insertUndirected(V u, V v, E e) {
        // insert in both directions
        insertDirected(u, v, e);
        insertDirected(v, u, e);
    }
```

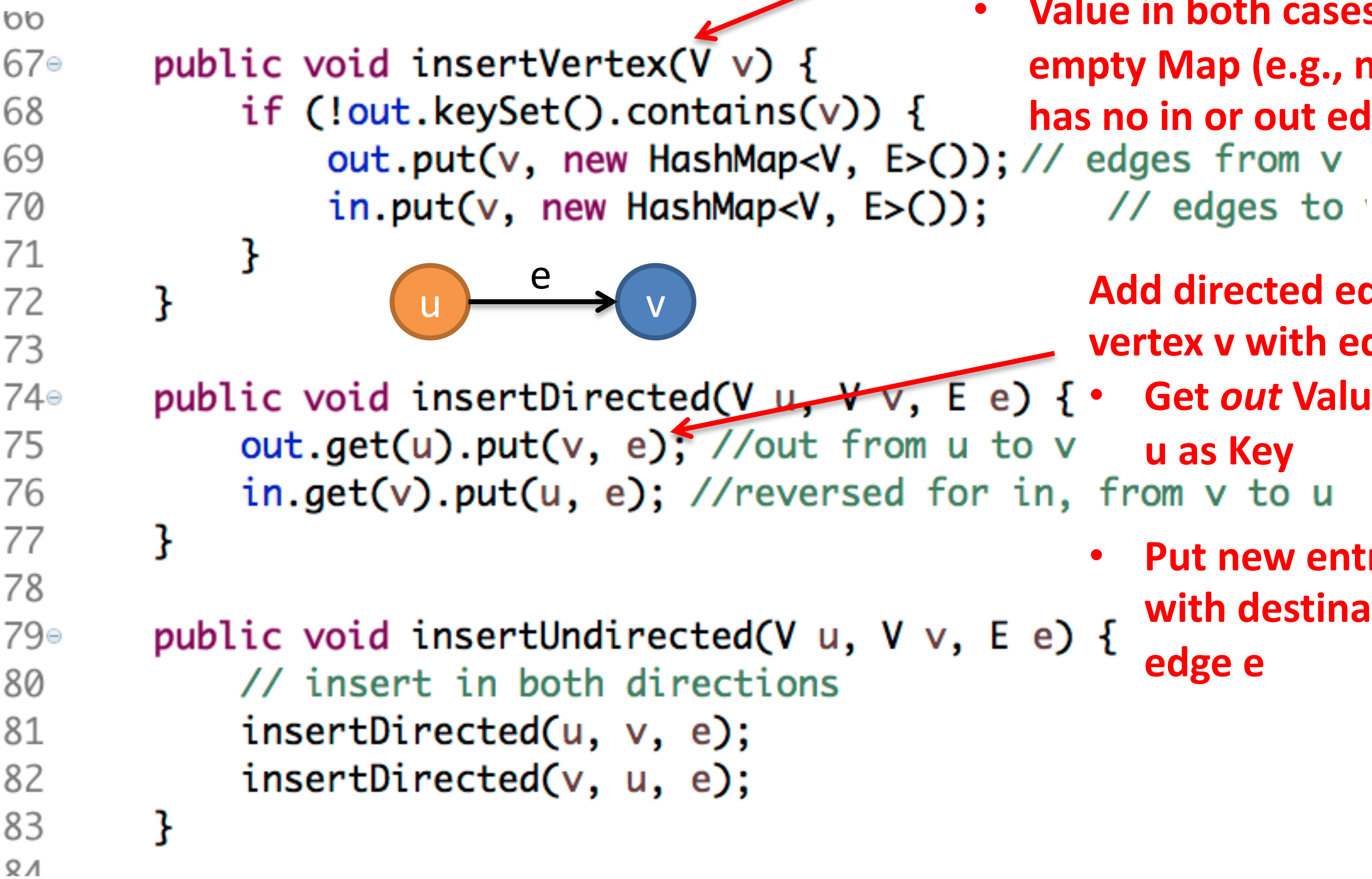
Add directed edge from vertex u to vertex v with edge label e

- **Get *out* Value Map using vertex u as Key**

Inserting vertices and edges requires updating both *in* and *out*

AdjacencyMapGraph.java

- **Adding new vertex adds Key to both *in* and *out***
- **Value in both cases is set to empty Map (e.g., new vertex has no in or out edges)**



```java
    public void insertVertex(V v) {
        if (!out.keySet().contains(v)) {
            out.put(v, new HashMap<V, E>()); // edges from v
            in.put(v, new HashMap<V, E>());     // edges to
        }
    }

    public void insertDirected(V u, V v, E e) {
        out.get(u).put(v, e); //out from u to v
        in.get(v).put(u, e); //reversed for in, from v to u
    }

    public void insertUndirected(V u, V v, E e) {
        // insert in both directions
        insertDirected(u, v, e);
        insertDirected(v, u, e);
    }
```

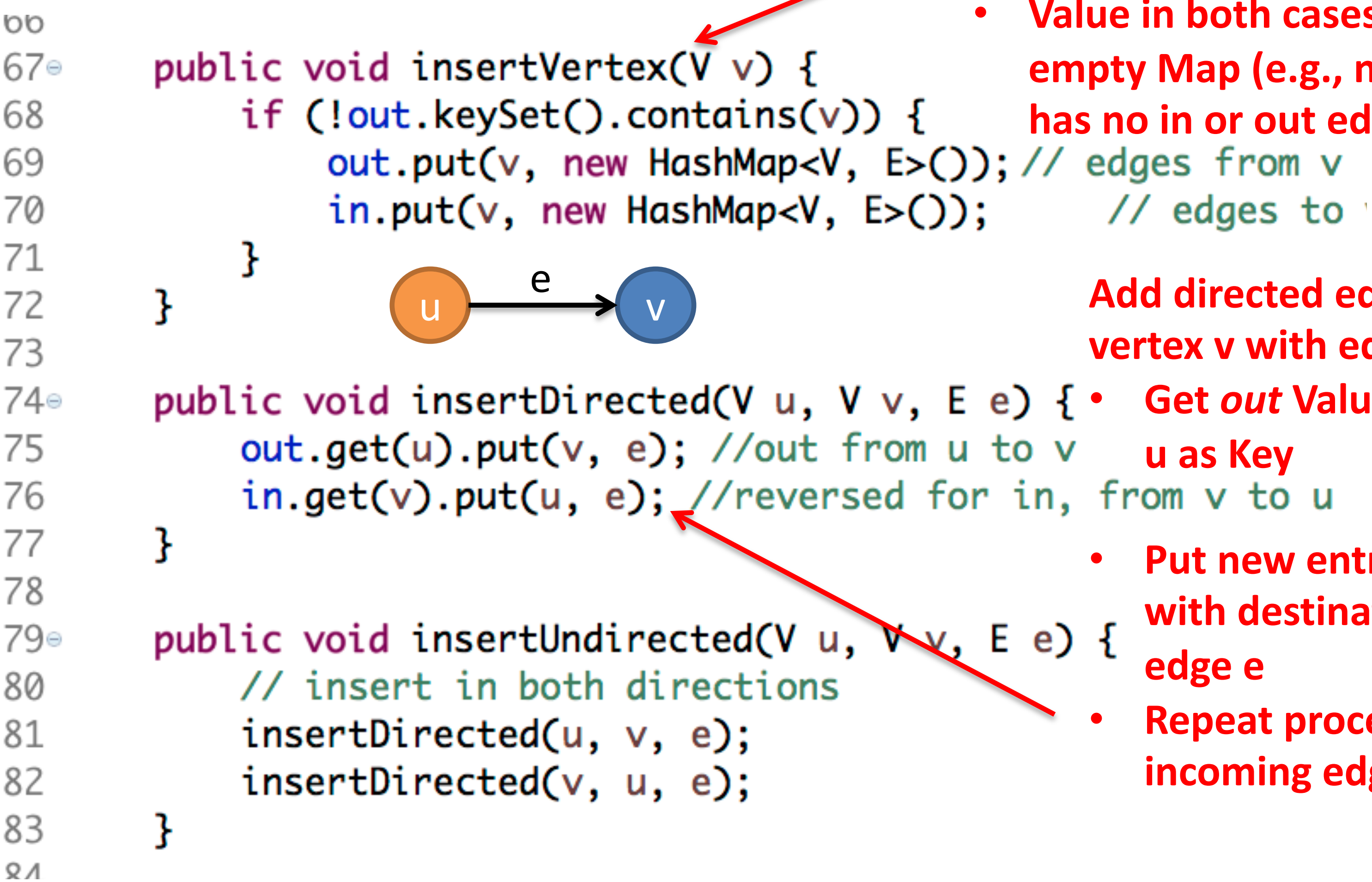
Add directed edge from vertex u to vertex v with edge label e

- **Get *out* Value Map using vertex u as Key**
- **Put new entry into Value Map with destination vertex v and edge e**

Inserting vertices and edges requires updating both *in* and *out*

AdjacencyMapGraph.java

- **Adding new vertex adds Key to both *in* and *out***
- **Value in both cases is set to empty Map (e.g., new vertex has no in or out edges)**



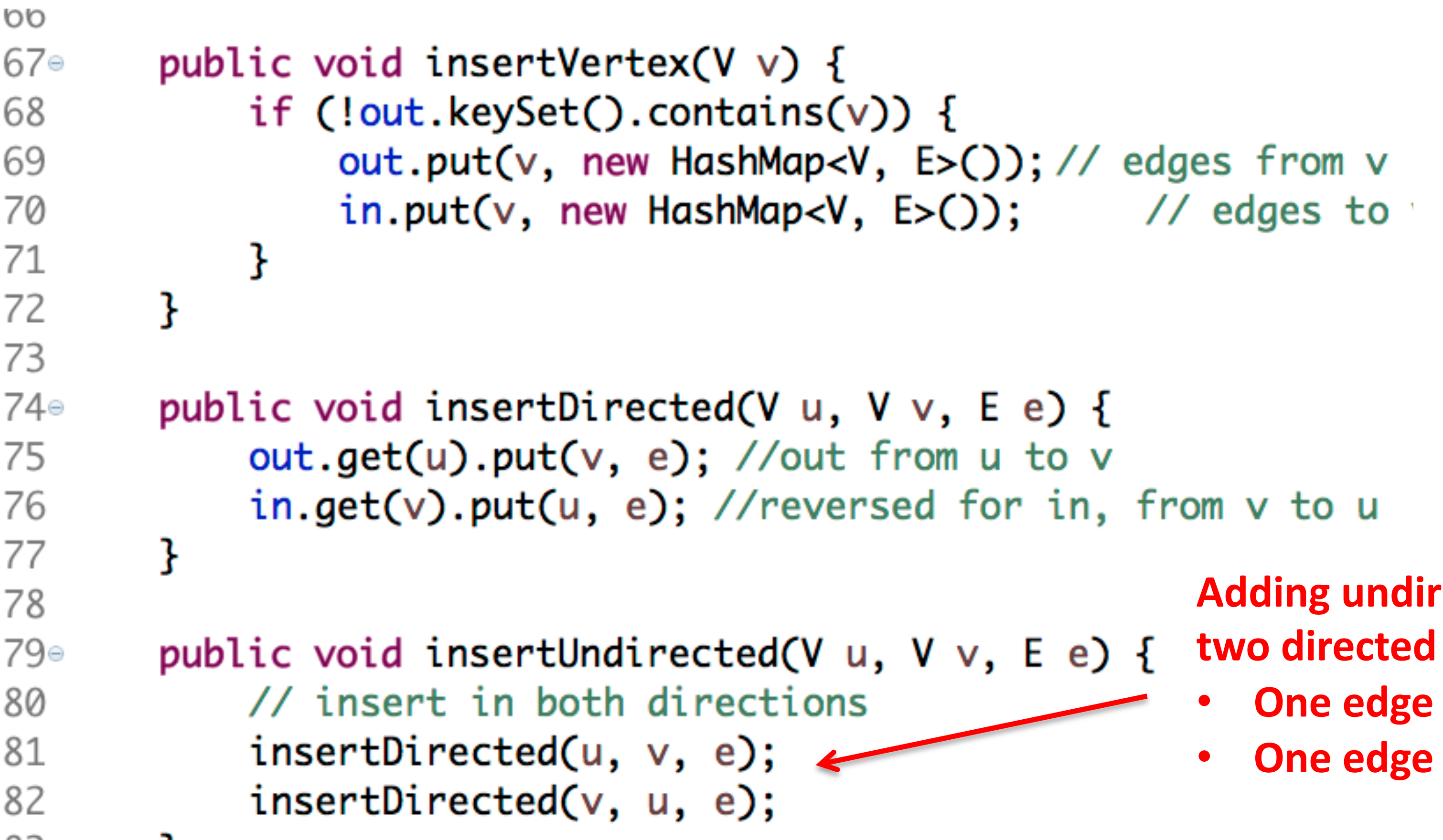
```java
    public void insertVertex(V v) {
        if (!out.keySet().contains(v)) {
            out.put(v, new HashMap<V, E>()); // edges from v
            in.put(v, new HashMap<V, E>());      // edges to
        }
    }

    public void insertDirected(V u, V v, E e) {
        out.get(u).put(v, e); //out from u to v
        in.get(v).put(u, e); //reversed for in, from v to u
    }

    public void insertUndirected(V u, V v, E e) {
        // insert in both directions
        insertDirected(u, v, e);
        insertDirected(v, u, e);
    }
```

Add directed edge from vertex u to vertex v with edge label e

- **Get *out* Value Map using vertex u as Key**
- **Put new entry into Value Map with destination vertex v and edge e**
- **Repeat process, updating *in* for incoming edge e into v from u**

We model undirected edges as directed edges going in both directions

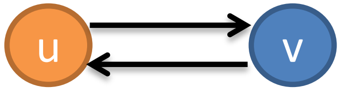
AdjacencyMapGraph.java



```
    public void insertVertex(V v) {
        if (!out.keySet().contains(v)) {
            out.put(v, new HashMap<V, E>());// edges from v
            in.put(v, new HashMap<V, E>());      // edges to
        }
    }

    public void insertDirected(V u, V v, E e) {
        out.get(u).put(v, e); //out from u to v
        in.get(v).put(u, e); //reversed for in, from v to u
    }

    public void insertUndirected(V u, V v, E e) {
        // insert in both directions
        insertDirected(u, v, e);
        insertDirected(v, u, e);
    }
```

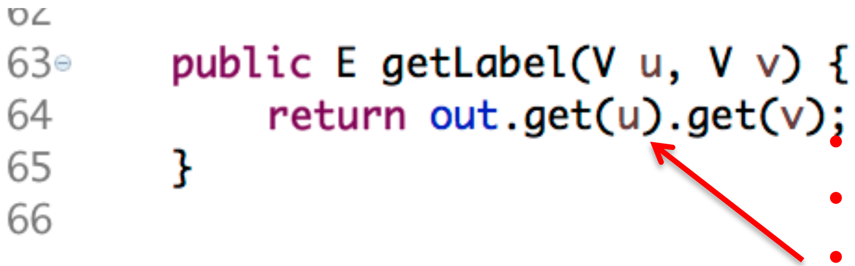
Adding undirected edge creates two directed edges

- One edge from u to v
- One edge from v to u



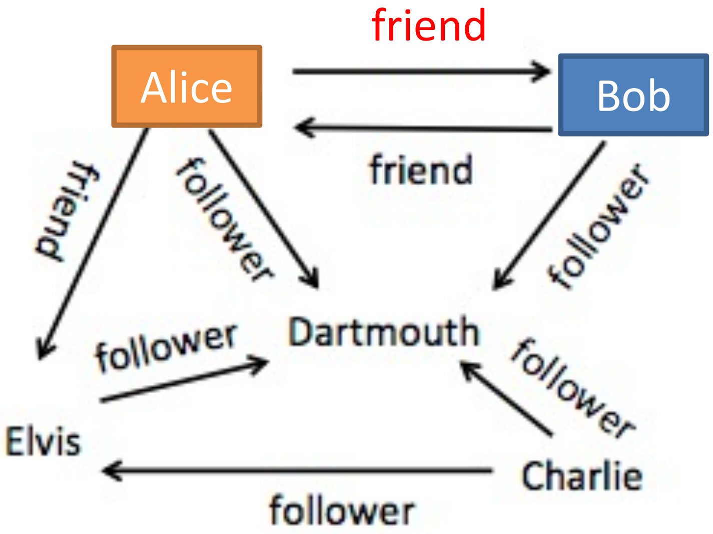
getLabel(u,v) returns the label on the edge between *u* and *v*

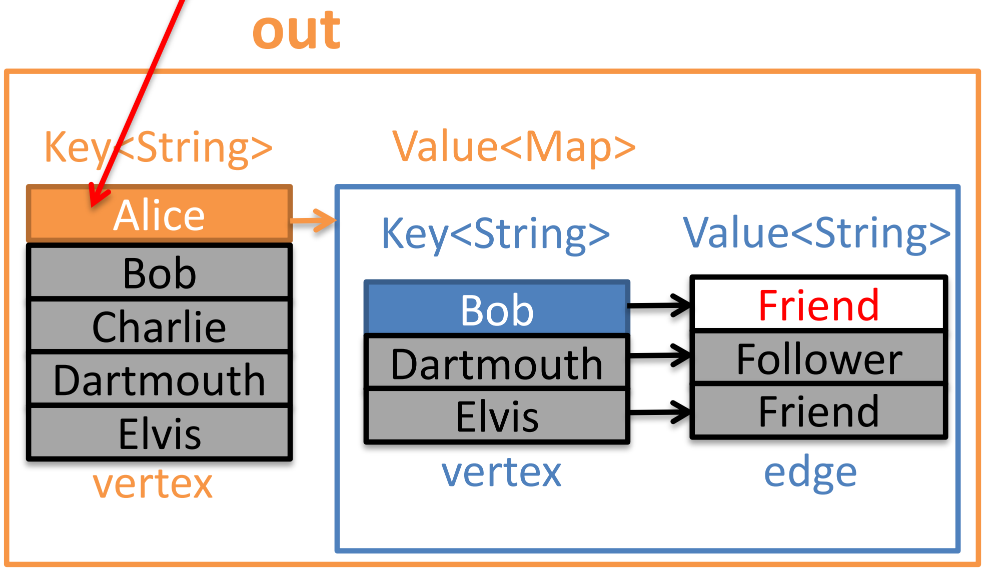
AdjacencyMapGraph.java



```
    public E getLabel(V u, V v) {
        return out.get(u).get(v);
    }
```

- ***getLabel(u,v)* returns label on edge from u to v**
- ***getLabel("Alice", "Bob")* returns *"Friend"***
- **First get Value Map for Key "Alice" from *out***





out

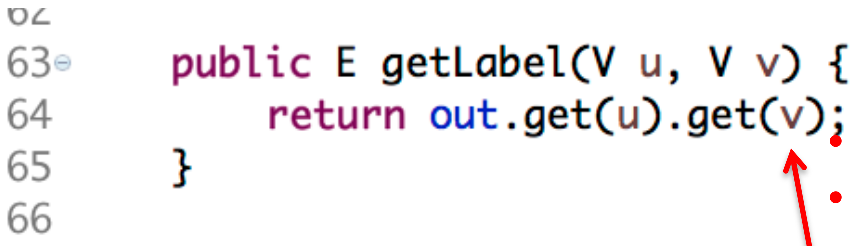
Key<String> Value<Map>

Alice
Bob
Charlie
Dartmouth
Elvis
vertex

Key<String>	Value<String>
Bob	Friend
Dartmouth	Follower
Elvis	Friend
vertex	edge

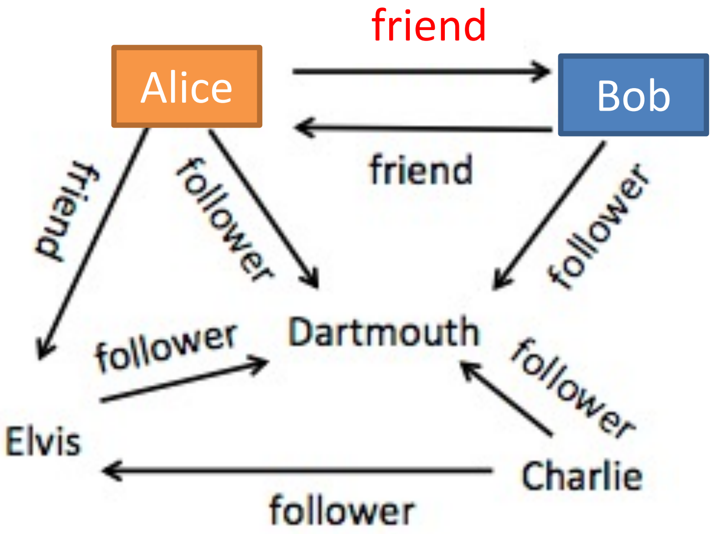
getLabel(u,v) returns the label on the edge between *u* and *v*

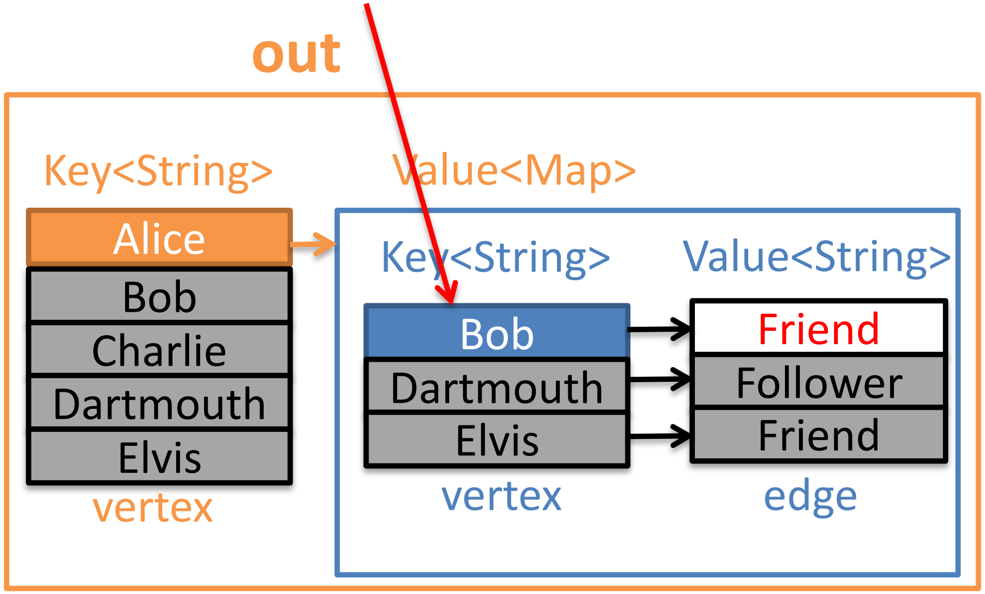
AdjacencyMapGraph.java



```
    public E getLabel(V u, V v) {
        return out.get(u).get(v);
    }

```

- ***getLabel(u,v)* returns label on edge from u to v**
- ***getLabel("Alice","Bob")* returns *"Friend"***
- **First get Value Map for Key "Alice" from *out***
- **Next get use Key "Bob" to get Value String**





out

Key<String>

Value<Map>

Alice

Bob

Charlie

Dartmouth

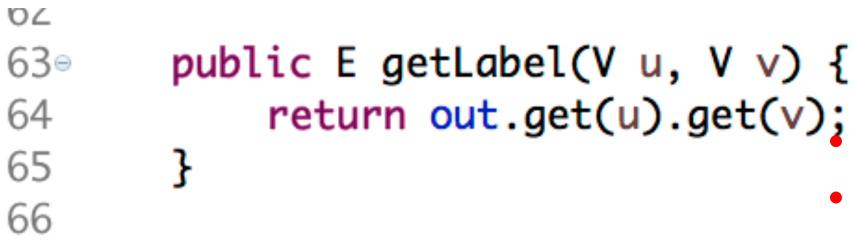
Elvis

vertex

Key<String>	Value<String>
Bob	Friend
Dartmouth	Follower
Elvis	Friend
vertex	edge

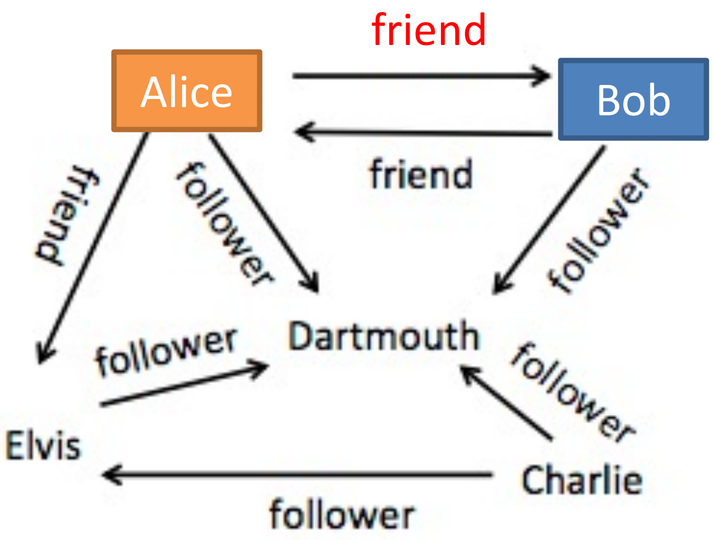
getLabel(u,v) returns the label on the edge between *u* and *v*

AdjacencyMapGraph.java

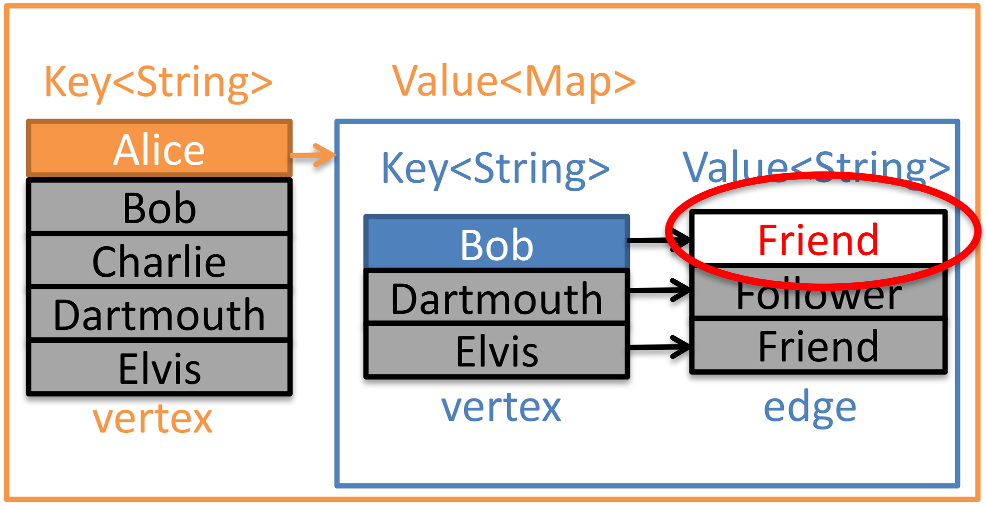


```
    public E getLabel(V u, V v) {
        return out.get(u).get(v);
    }
```

- ***getLabel(u,v)*** **returns label on edge from u to v**
- ***getLabel("Alice","Bob")*** **returns *"Friend"***
- **First get Value Map for Key "Alice" from *out***
- **Next get use Key "Bob" to get Value String**
- **Return "Friend"**



out

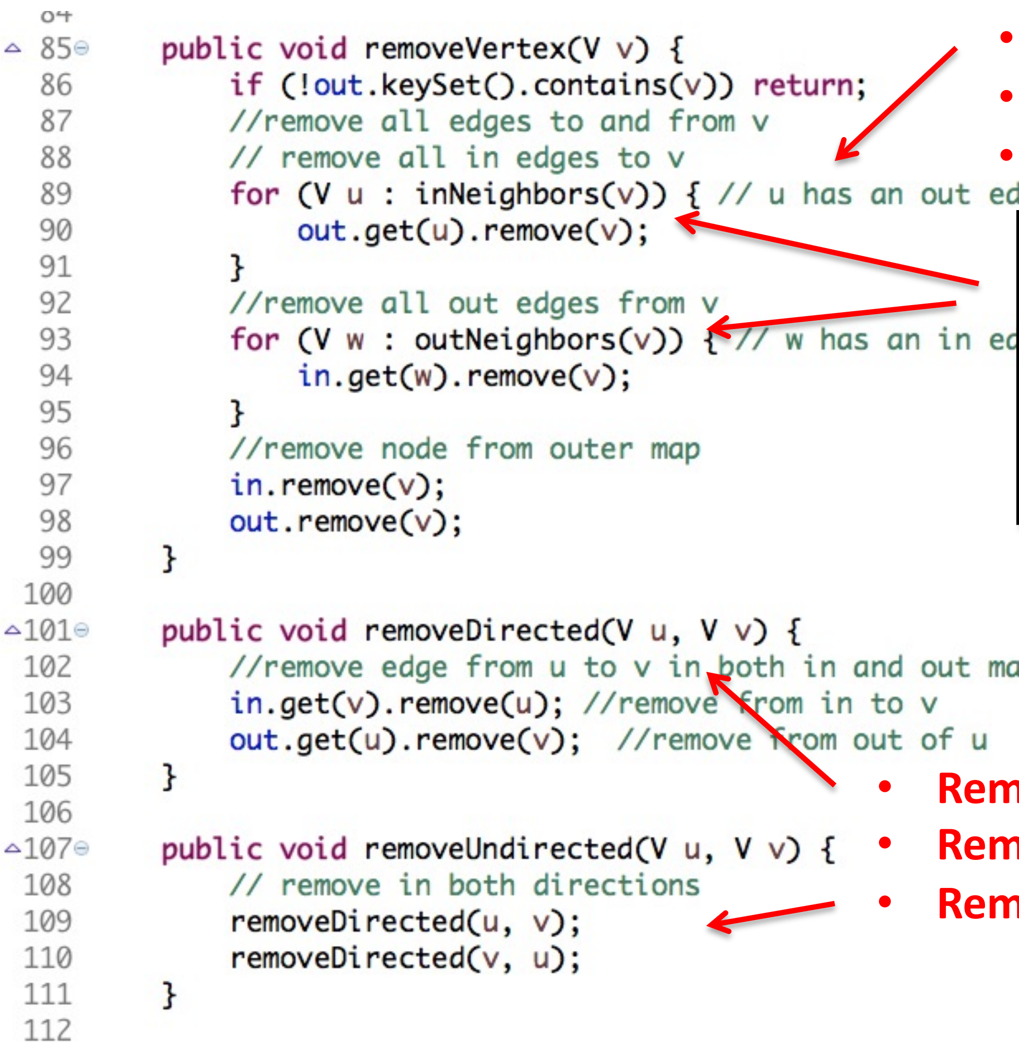


Key<String>	Value<Map>
Alice	
Bob	
Charlie	
Dartmouth	
Elvis	
vertex	

Key<String>	Value<String>
Bob	Friend
Dartmouth	Follower
Elvis	Friend
vertex	edge

When removing edges and vertices, must remove from both *in* and *out* Maps

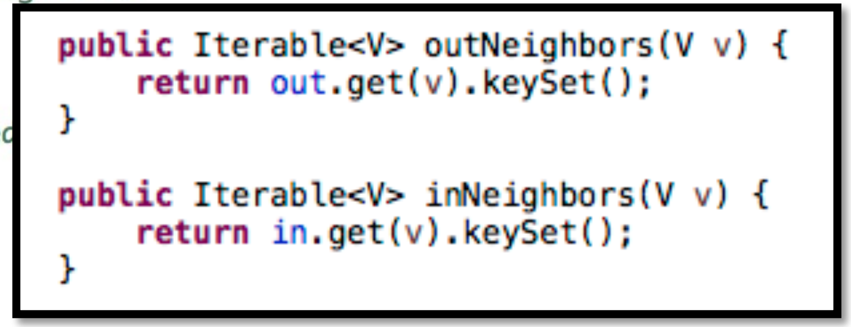
AdjacencyMapGraph.java



```
    public void removeVertex(V v) {
        if (!out.keySet().contains(v)) return;
        //remove all edges to and from v
        // remove all in edges to v
        for (V u : inNeighbors(v)) { // u has an out edge to v
            out.get(u).remove(v);
        }
        //remove all out edges from v
        for (V w : outNeighbors(v)) { // w has an in ed
            in.get(w).remove(v);
        }
        //remove node from outer map
        in.remove(v);
        out.remove(v);
    }

    public void removeDirected(V u, V v) {
        //remove edge from u to v in both in and out maps
        in.get(v).remove(u); //remove from in to v
        out.get(u).remove(v);  //remove from out of u
    }

    public void removeUndirected(V u, V v) {
        // remove in both directions
        removeDirected(u, v);
        removeDirected(v, u);
    }
```

Removing vertex *v*

- **Remove all *in* edges (out from neighbor)**
- **Remove all *out* edges (in from neighbor)**
- **Then remove *v* from *in* and *out* Maps**



```
public Iterable<V> outNeighbors(V v) {
    return out.get(v).keySet();
}

public Iterable<V> inNeighbors(V v) {
    return in.get(v).keySet();
}
```

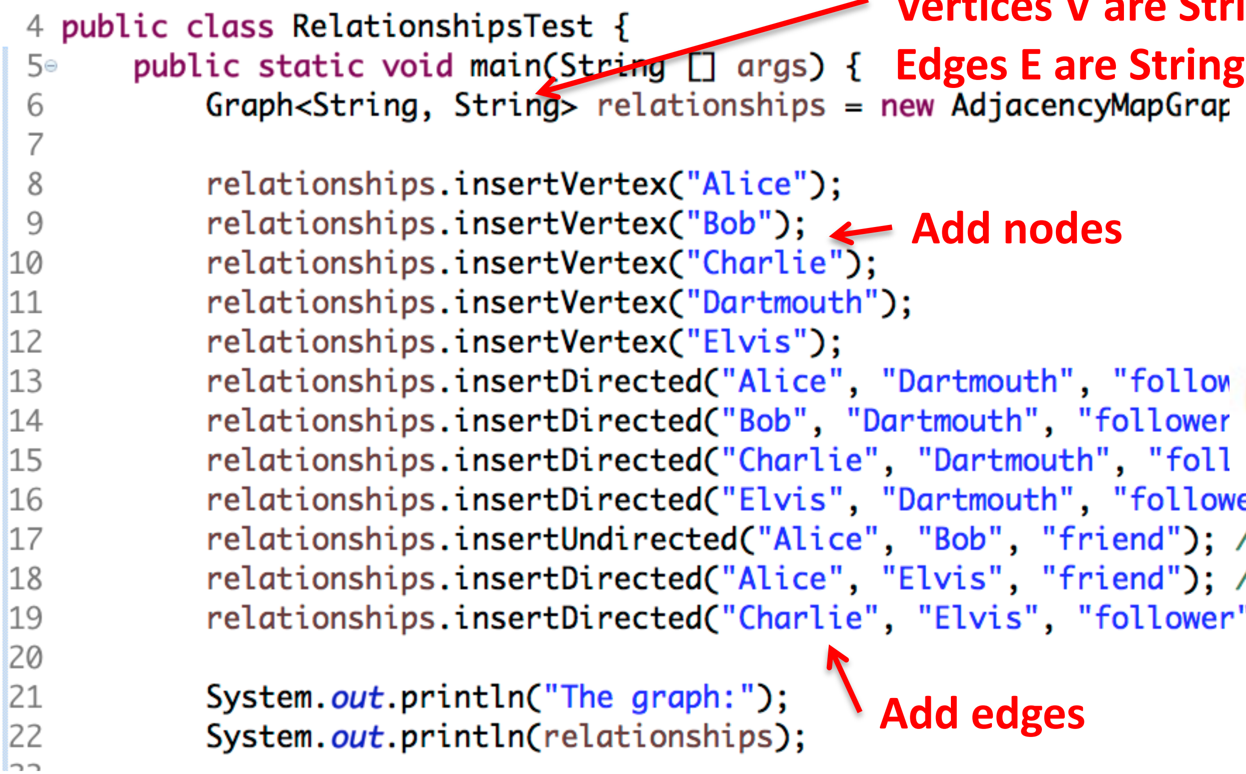
- **Removing directed edge from *u* to *v***
- **Remove from both *in* and *out* Maps**
- **Removing undirected, call *removeDirected()* twice**

RelationshipTest.java

ANNOTATED SLIDES

RelationshipTest.java: create graph with both directed and non-directed edges

RelationshipTest.java

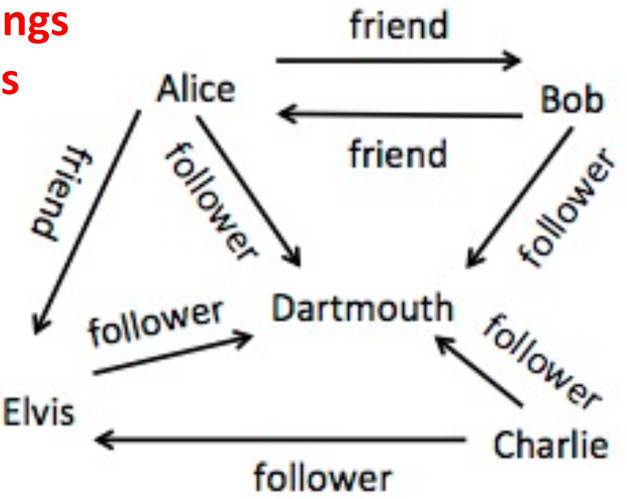


```java
public class RelationshipsTest {
    public static void main(String [] args) {
        Graph<String, String> relationships = new AdjacencyMapGrap

        relationships.insertVertex("Alice");
        relationships.insertVertex("Bob");
        relationships.insertVertex("Charlie");
        relationships.insertVertex("Dartmouth");
        relationships.insertVertex("Elvis");
        relationships.insertDirected("Alice", "Dartmouth", "follow
        relationships.insertDirected("Bob", "Dartmouth", "follower
        relationships.insertDirected("Charlie", "Dartmouth", "foll
        relationships.insertDirected("Elvis", "Dartmouth", "follower");
        relationships.insertUndirected("Alice", "Bob", "friend"); // symmetric, undirected edg
        relationships.insertDirected("Alice", "Elvis", "friend"); // not symmetric, directed e
        relationships.insertDirected("Charlie", "Elvis", "follower");

        System.out.println("The graph:");
        System.out.println(relationships);
```

Declare graph:
Vertices V are Strings
Edges E are Strings

Add nodes

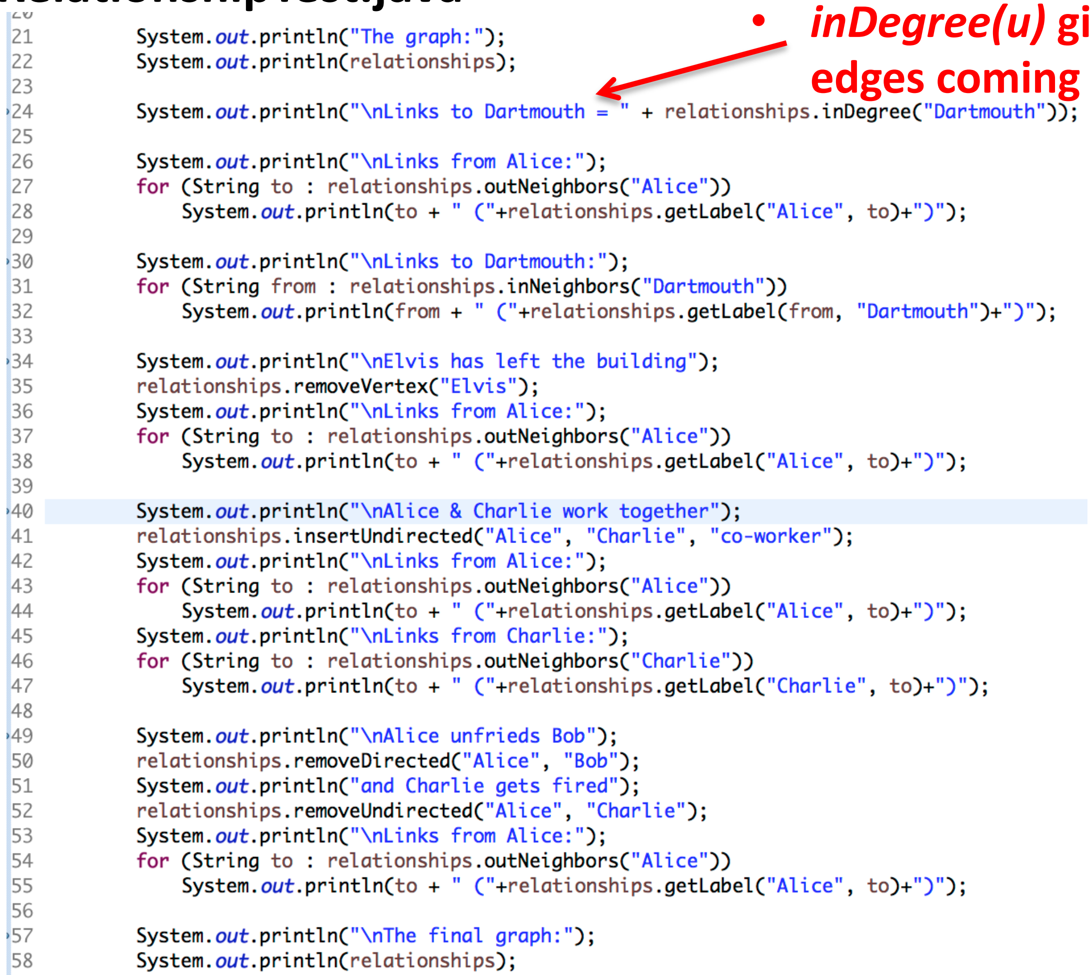
Add edges



Output (from implicit *toString()* call):
The graph:
Vertices: [Bob, Dartmouth, Alice, Elvis, Charlie]
Out edges: {Bob={Dartmouth=follower, Alice=friend}, Dartmouth={}, Alice={Dartmouth=follower, Bob=friend, Elvis=friend}, Elvis={Dartmouth=follower}, Charlie={Dartmouth=follower, Elvis=follower}}

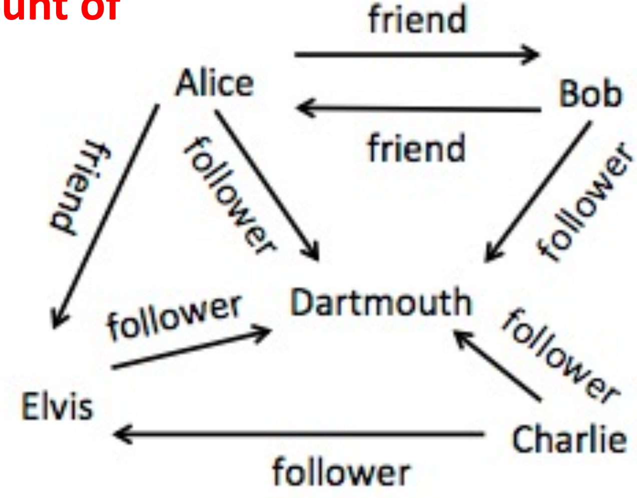
RelationshipTest.java: create graph with both directed and non-directed edges

RelationshipTest.java



```java
        System.out.println("The graph:");
        System.out.println(relationships);

        System.out.println("\nLinks to Dartmouth = " + relationships.inDegree("Dartmouth"));

        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nLinks to Dartmouth:");
        for (String from : relationships.inNeighbors("Dartmouth"))
            System.out.println(from + " ("+relationships.getLabel(from, "Dartmouth")+")");

        System.out.println("\nElvis has left the building");
        relationships.removeVertex("Elvis");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nAlice & Charlie work together");
        relationships.insertUndirected("Alice", "Charlie", "co-worker");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");
        System.out.println("\nLinks from Charlie:");
        for (String to : relationships.outNeighbors("Charlie"))
            System.out.println(to + " ("+relationships.getLabel("Charlie", to)+")");

        System.out.println("\nAlice unfrieds Bob");
        relationships.removeDirected("Alice", "Bob");
        System.out.println("and Charlie gets fired");
        relationships.removeUndirected("Alice", "Charlie");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nThe final graph:");
        System.out.println(relationships);
```

- ***inDegree(u)*** **gives count of edges coming into u**

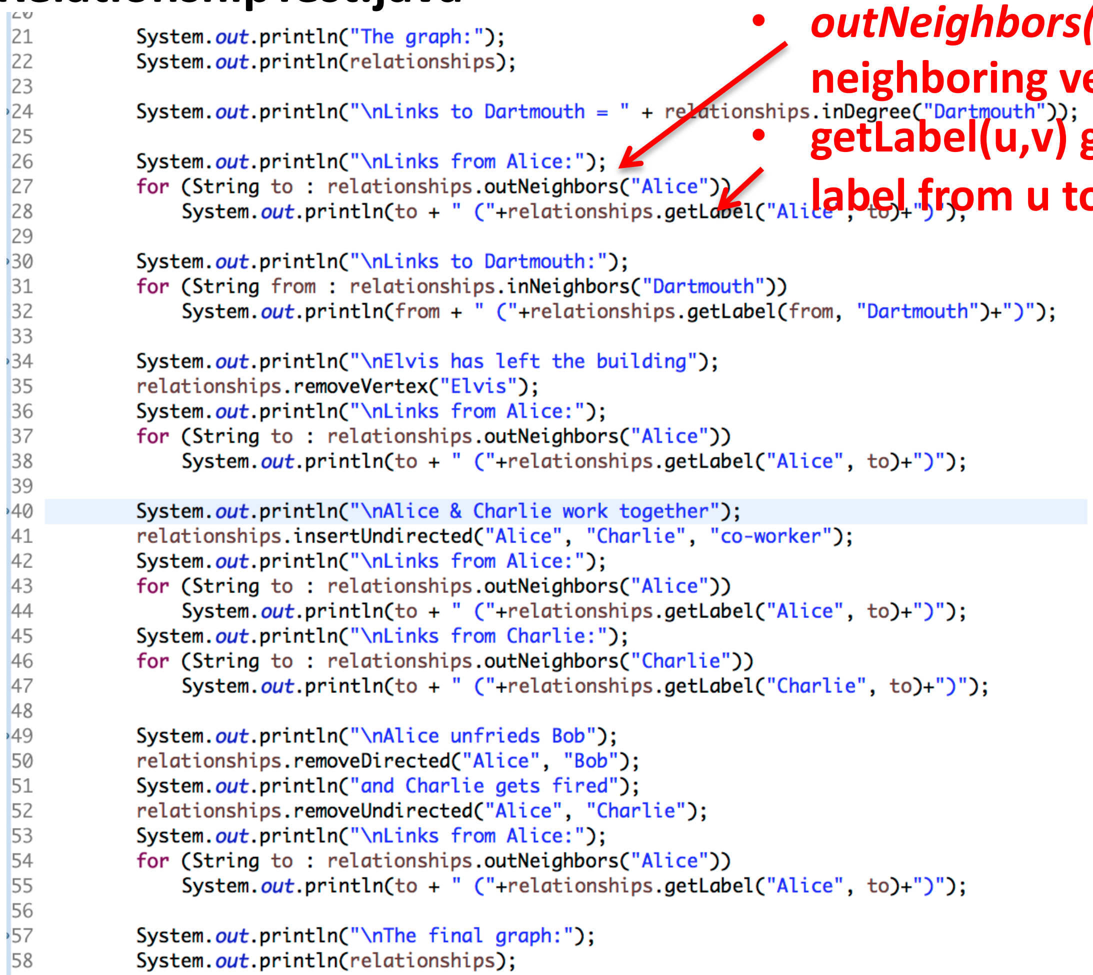


Output:

Links to Dartmouth = 4

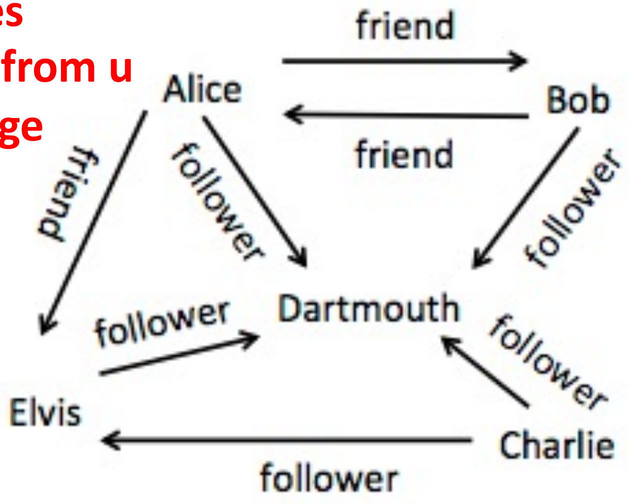
RelationshipTest.java: create graph with both directed and non-directed edges

RelationshipTest.java



```
        System.out.println("The graph:");
        System.out.println(relationships);

        System.out.println("\nLinks to Dartmouth = " + relationships.inDegree("Dartmouth"));

        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nLinks to Dartmouth:");
        for (String from : relationships.inNeighbors("Dartmouth"))
            System.out.println(from + " ("+relationships.getLabel(from, "Dartmouth")+")");

        System.out.println("\nElvis has left the building");
        relationships.removeVertex("Elvis");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nAlice & Charlie work together");
        relationships.insertUndirected("Alice", "Charlie", "co-worker");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");
        System.out.println("\nLinks from Charlie:");
        for (String to : relationships.outNeighbors("Charlie"))
            System.out.println(to + " ("+relationships.getLabel("Charlie", to)+")");

        System.out.println("\nAlice unfrieds Bob");
        relationships.removeDirected("Alice", "Bob");
        System.out.println("and Charlie gets fired");
        relationships.removeUndirected("Alice", "Charlie");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nThe final graph:");
        System.out.println(relationships);
```

- ***outNeighbors(u)*** **gives neighboring vertices from u**
- **getLabel(u,v) gets edge label from u to v**



Output:

Links from Alice:

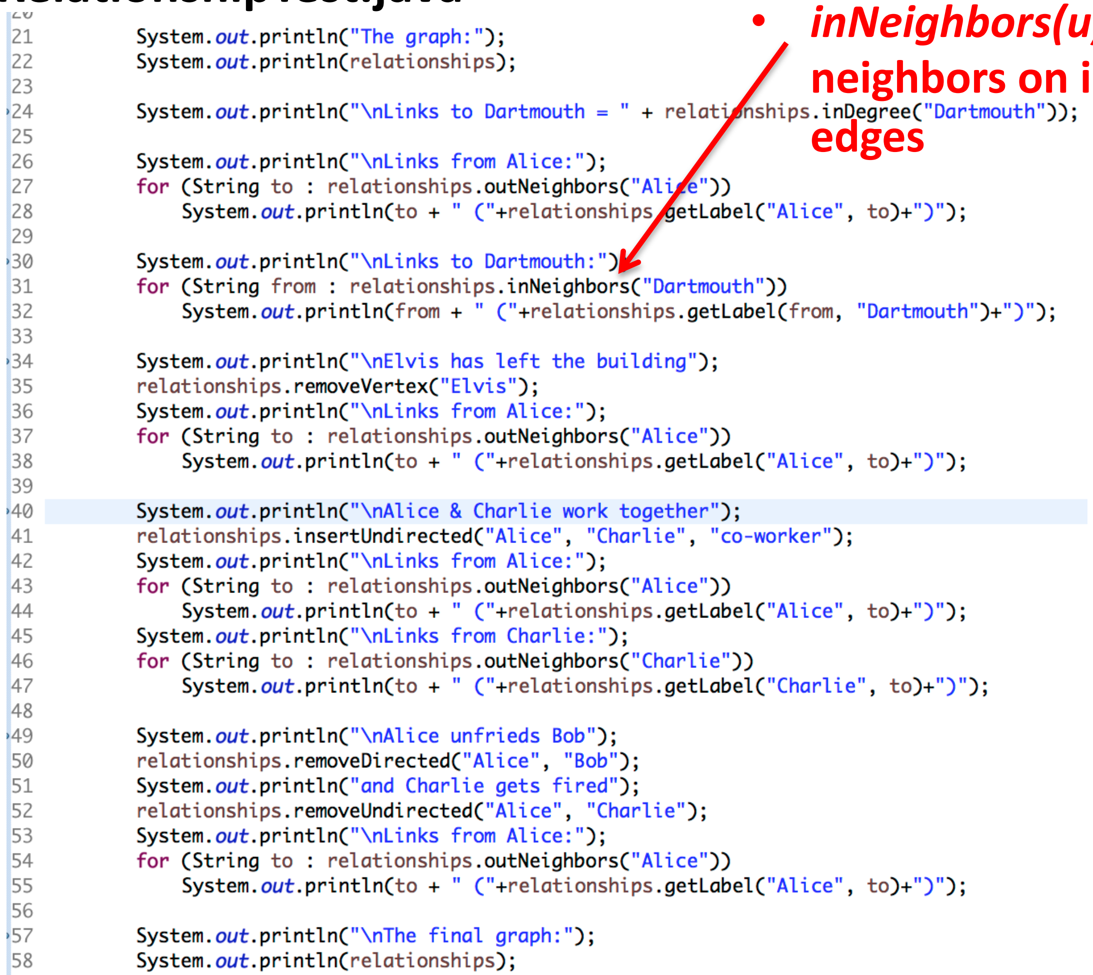
Dartmouth (follower)

Bob (friend)

Elvis (friend)

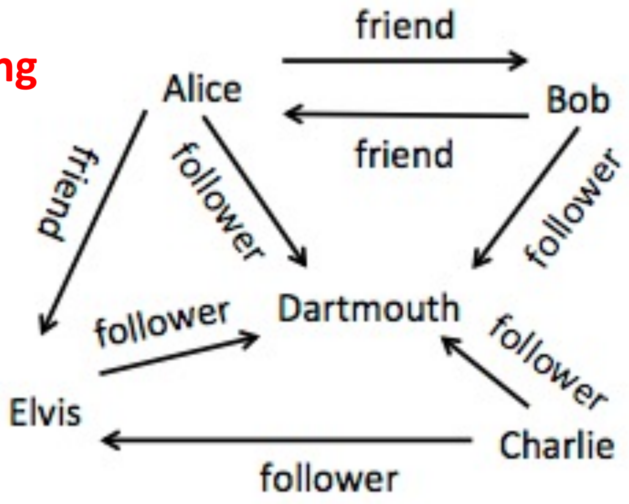
RelationshipTest.java: create graph with both directed and non-directed edges

RelationshipTest.java



```
        System.out.println("The graph:");
        System.out.println(relationships);

        System.out.println("\nLinks to Dartmouth = " + relationships.inDegree("Dartmouth"));

        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nLinks to Dartmouth:");
        for (String from : relationships.inNeighbors("Dartmouth"))
            System.out.println(from + " ("+relationships.getLabel(from, "Dartmouth")+")");

        System.out.println("\nElvis has left the building");
        relationships.removeVertex("Elvis");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nAlice & Charlie work together");
        relationships.insertUndirected("Alice", "Charlie", "co-worker");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");
        System.out.println("\nLinks from Charlie:");
        for (String to : relationships.outNeighbors("Charlie"))
            System.out.println(to + " ("+relationships.getLabel("Charlie", to)+")");

        System.out.println("\nAlice unfrieds Bob");
        relationships.removeDirected("Alice", "Bob");
        System.out.println("and Charlie gets fired");
        relationships.removeUndirected("Alice", "Charlie");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nThe final graph:");
        System.out.println(relationships);
```

- ***inNeighbors(u)* gives neighbors on incoming edges**

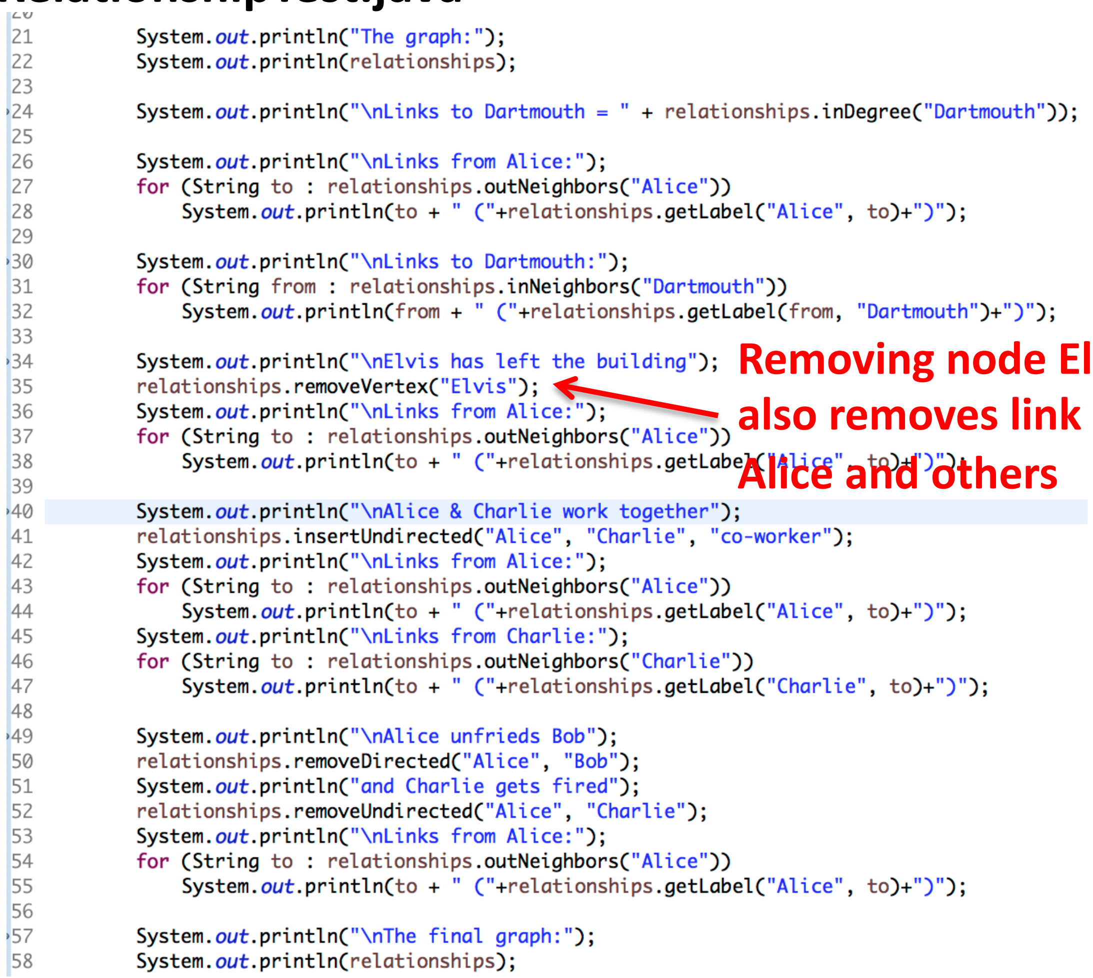


Output:

```
Links to Dartmouth:
Bob (follower)
Alice (follower)
Elvis (follower)
Charlie (follower)
```

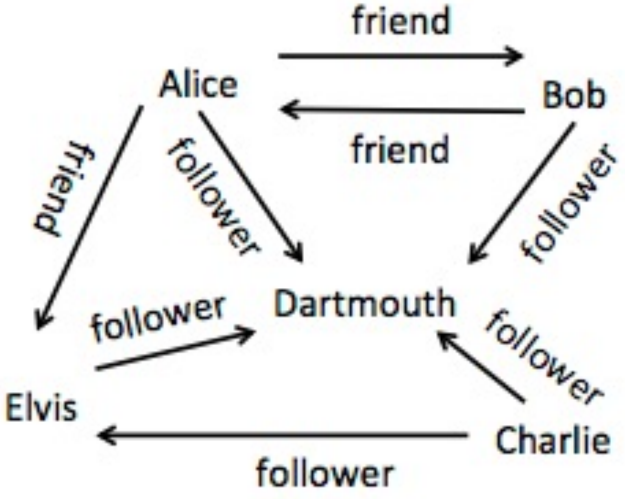
RelationshipTest.java: create graph with both directed and non-directed edges

RelationshipTest.java



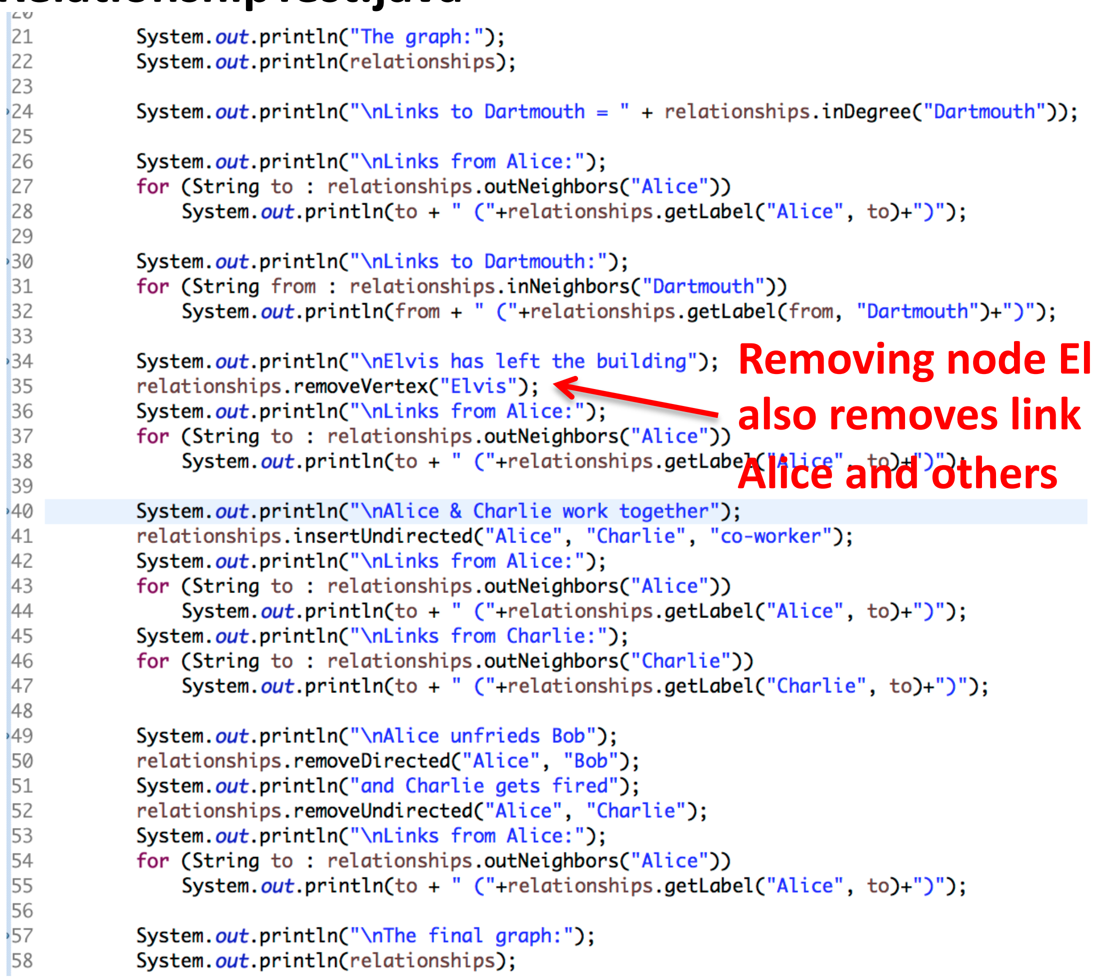
```java
        System.out.println("The graph:");
        System.out.println(relationships);

        System.out.println("\nLinks to Dartmouth = " + relationships.inDegree("Dartmouth"));

        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nLinks to Dartmouth:");
        for (String from : relationships.inNeighbors("Dartmouth"))
            System.out.println(from + " ("+relationships.getLabel(from, "Dartmouth")+")");

        System.out.println("\nElvis has left the building");
        relationships.removeVertex("Elvis");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nAlice & Charlie work together");
        relationships.insertUndirected("Alice", "Charlie", "co-worker");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");
        System.out.println("\nLinks from Charlie:");
        for (String to : relationships.outNeighbors("Charlie"))
            System.out.println(to + " ("+relationships.getLabel("Charlie", to)+")");

        System.out.println("\nAlice unfrieds Bob");
        relationships.removeDirected("Alice", "Bob");
        System.out.println("and Charlie gets fired");
        relationships.removeUndirected("Alice", "Charlie");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nThe final graph:");
        System.out.println(relationships);
```

Removing node Elvis also removes link from Alice and others



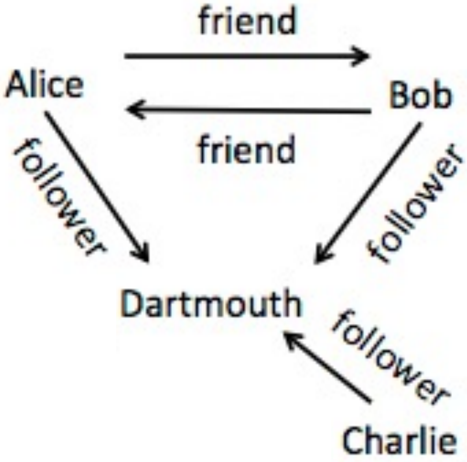
RelationshipTest.java: create graph with both directed and non-directed edges

RelationshipTest.java



```java
        System.out.println("The graph:");
        System.out.println(relationships);

        System.out.println("\nLinks to Dartmouth = " + relationships.inDegree("Dartmouth"));

        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nLinks to Dartmouth:");
        for (String from : relationships.inNeighbors("Dartmouth"))
            System.out.println(from + " ("+relationships.getLabel(from, "Dartmouth")+")");

        System.out.println("\nElvis has left the building");
        relationships.removeVertex("Elvis");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nAlice & Charlie work together");
        relationships.insertUndirected("Alice", "Charlie", "co-worker");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");
        System.out.println("\nLinks from Charlie:");
        for (String to : relationships.outNeighbors("Charlie"))
            System.out.println(to + " ("+relationships.getLabel("Charlie", to)+")");

        System.out.println("\nAlice unfrieds Bob");
        relationships.removeDirected("Alice", "Bob");
        System.out.println("and Charlie gets fired");
        relationships.removeUndirected("Alice", "Charlie");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nThe final graph:");
        System.out.println(relationships);
```

Removing node Elvis also removes link from Alice and others



Output:

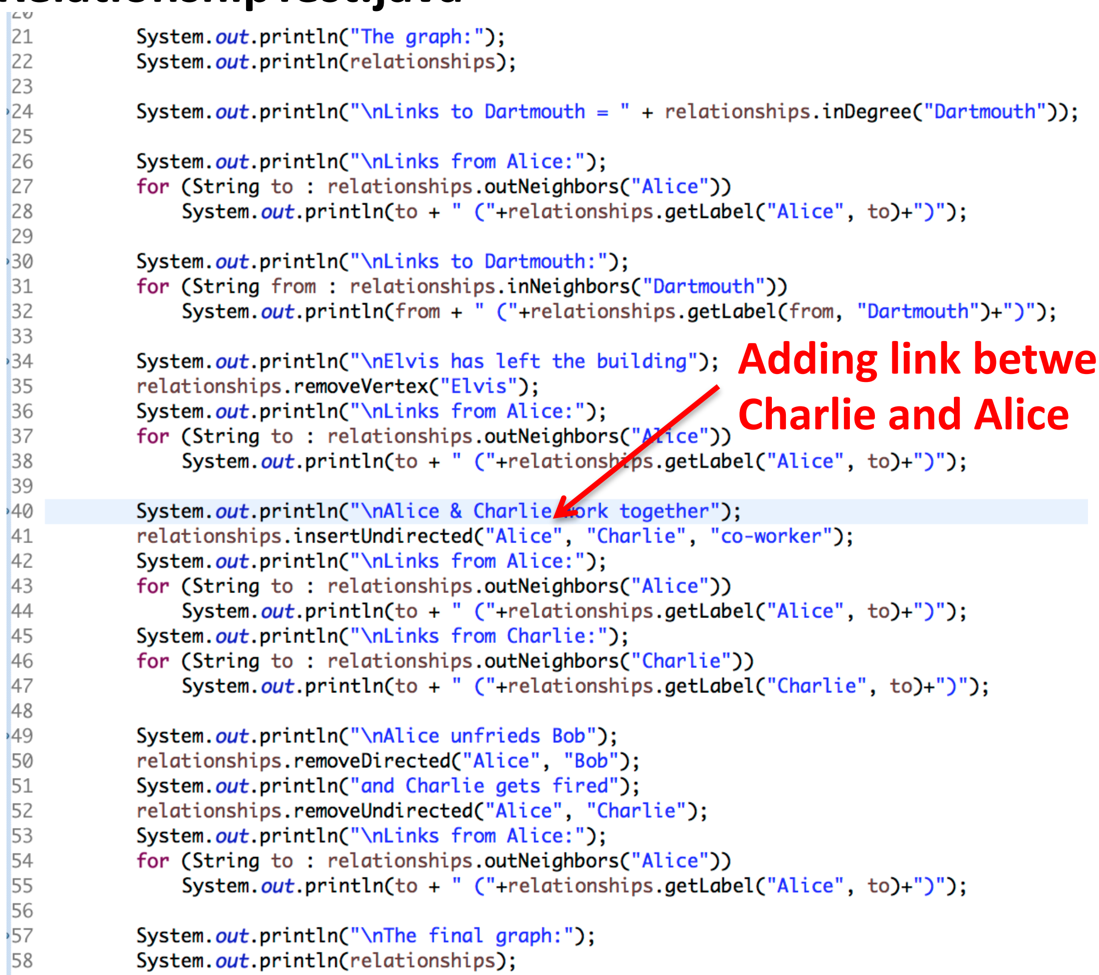
Links from Alice:

Dartmouth (follower)

Bob (friend)

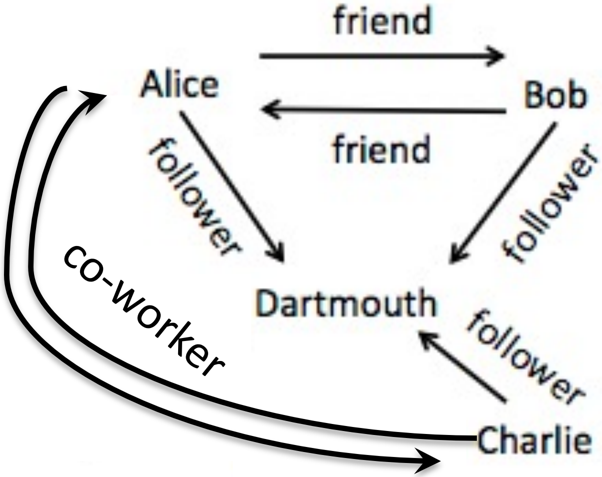
RelationshipTest.java: create graph with both directed and non-directed edges

RelationshipTest.java



```java
        System.out.println("The graph:");
        System.out.println(relationships);

        System.out.println("\nLinks to Dartmouth = " + relationships.inDegree("Dartmouth"));

        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nLinks to Dartmouth:");
        for (String from : relationships.inNeighbors("Dartmouth"))
            System.out.println(from + " ("+relationships.getLabel(from, "Dartmouth")+")");

        System.out.println("\nElvis has left the building");
        relationships.removeVertex("Elvis");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nAlice & Charlie work together");
        relationships.insertUndirected("Alice", "Charlie", "co-worker");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");
        System.out.println("\nLinks from Charlie:");
        for (String to : relationships.outNeighbors("Charlie"))
            System.out.println(to + " ("+relationships.getLabel("Charlie", to)+")");

        System.out.println("\nAlice unfrieds Bob");
        relationships.removeDirected("Alice", "Bob");
        System.out.println("and Charlie gets fired");
        relationships.removeUndirected("Alice", "Charlie");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nThe final graph:");
        System.out.println(relationships);
```

Adding link between Charlie and Alice

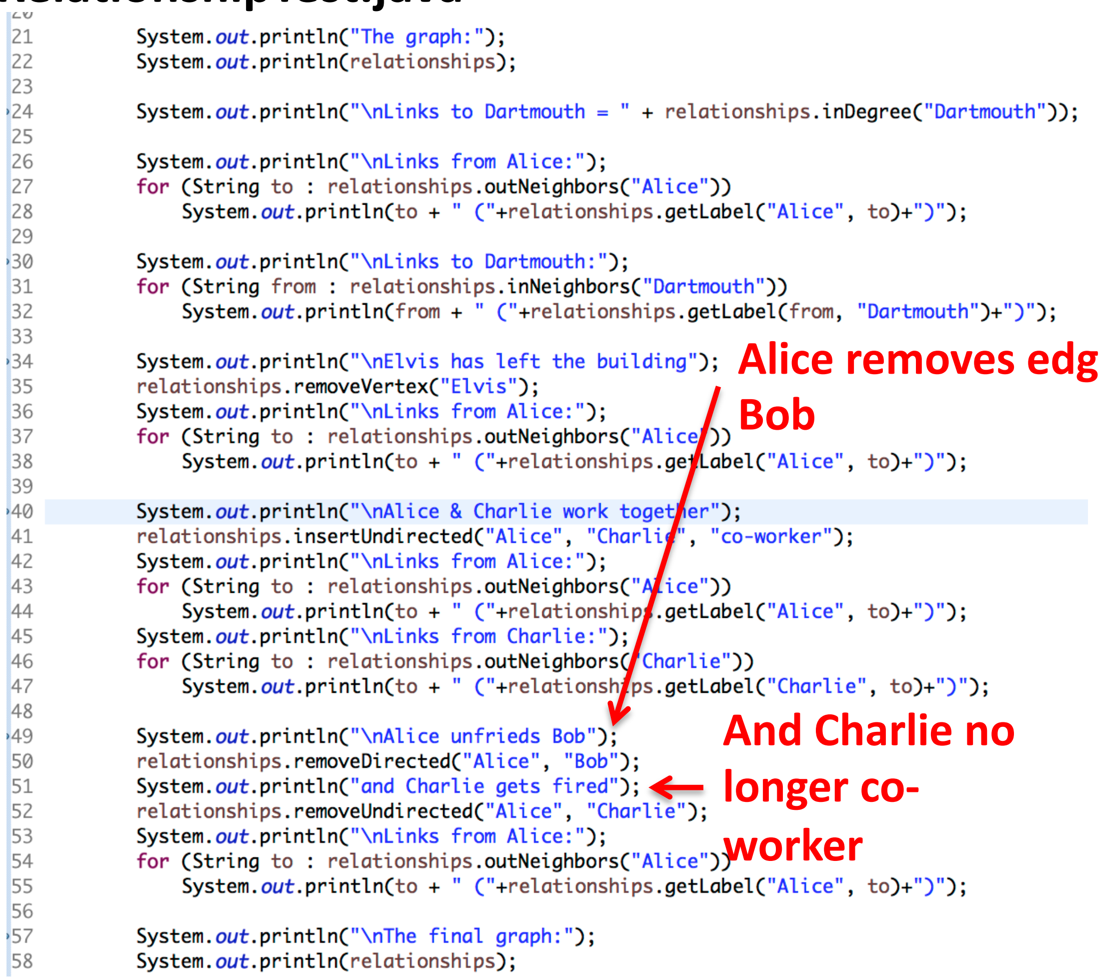


Output:

```
Alice & Charlie work together

Links from Alice:
Dartmouth (follower)
Bob (friend)
Charlie (co-worker)

Links from Charlie:
Dartmouth (follower)
Alice (co-worker)
```

RelationshipTest.java: create graph with both directed and non-directed edges

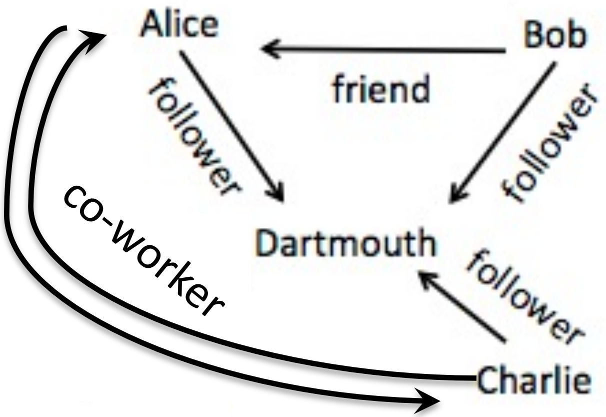
RelationshipTest.java



```
        System.out.println("The graph:");
        System.out.println(relationships);

        System.out.println("\nLinks to Dartmouth = " + relationships.inDegree("Dartmouth"));

        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nLinks to Dartmouth:");
        for (String from : relationships.inNeighbors("Dartmouth"))
            System.out.println(from + " ("+relationships.getLabel(from, "Dartmouth")+")");

        System.out.println("\nElvis has left the building");
        relationships.removeVertex("Elvis");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nAlice & Charlie work together");
        relationships.insertUndirected("Alice", "Charlie", "co-worker");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");
        System.out.println("\nLinks from Charlie:");
        for (String to : relationships.outNeighbors("Charlie"))
            System.out.println(to + " ("+relationships.getLabel("Charlie", to)+")");

        System.out.println("\nAlice unfrieds Bob");
        relationships.removeDirected("Alice", "Bob");
        System.out.println("and Charlie gets fired");
        relationships.removeUndirected("Alice", "Charlie");
        System.out.println("\nLinks from Alice:");
        for (String to : relationships.outNeighbors("Alice"))
            System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

        System.out.println("\nThe final graph:");
        System.out.println(relationships);
```

Alice removes edge to Bob

And Charlie no longer co-worker



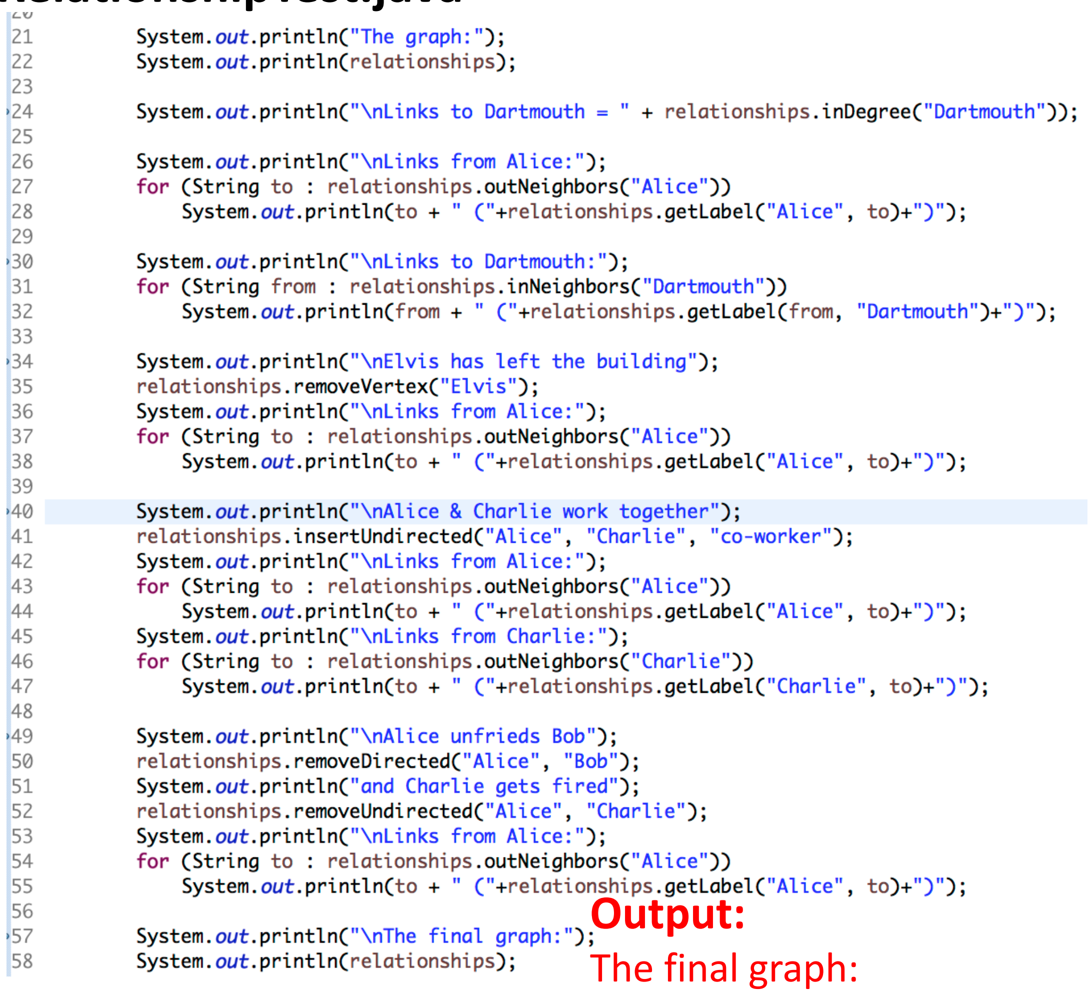
Output:

Alice unfriends Bob and Charlie gets fired

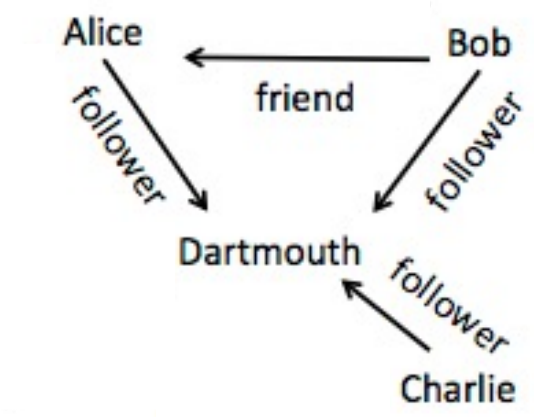
Links from Alice:
Dartmouth (follower)

RelationshipTest.java: create graph with both directed and non-directed edges

RelationshipTest.java



```
System.out.println("The graph:");
System.out.println(relationships);

System.out.println("\nLinks to Dartmouth = " + relationships.inDegree("Dartmouth"));

System.out.println("\nLinks from Alice:");
for (String to : relationships.outNeighbors("Alice"))
    System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

System.out.println("\nLinks to Dartmouth:");
for (String from : relationships.inNeighbors("Dartmouth"))
    System.out.println(from + " ("+relationships.getLabel(from, "Dartmouth")+")");

System.out.println("\nElvis has left the building");
relationships.removeVertex("Elvis");
System.out.println("\nLinks from Alice:");
for (String to : relationships.outNeighbors("Alice"))
    System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

System.out.println("\nAlice & Charlie work together");
relationships.insertUndirected("Alice", "Charlie", "co-worker");
System.out.println("\nLinks from Alice:");
for (String to : relationships.outNeighbors("Alice"))
    System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");
System.out.println("\nLinks from Charlie:");
for (String to : relationships.outNeighbors("Charlie"))
    System.out.println(to + " ("+relationships.getLabel("Charlie", to)+")");

System.out.println("\nAlice unfrieds Bob");
relationships.removeDirected("Alice", "Bob");
System.out.println("and Charlie gets fired");
relationships.removeUndirected("Alice", "Charlie");
System.out.println("\nLinks from Alice:");
for (String to : relationships.outNeighbors("Alice"))
    System.out.println(to + " ("+relationships.getLabel("Alice", to)+")");

System.out.println("\nThe final graph:");
System.out.println(relationships);
```



Output:

```
The final graph:
Vertices: [Bob, Dartmouth, Alice, Charlie]
Out edges: {Bob={Dartmouth=follower, Alice=friend}, Dartmouth={},
Alice={Dartmouth=follower},Charlie={Dartmouth=follower}}
```