# CSE 250: Graphs

## Lecture 18

Oct 13, 2023

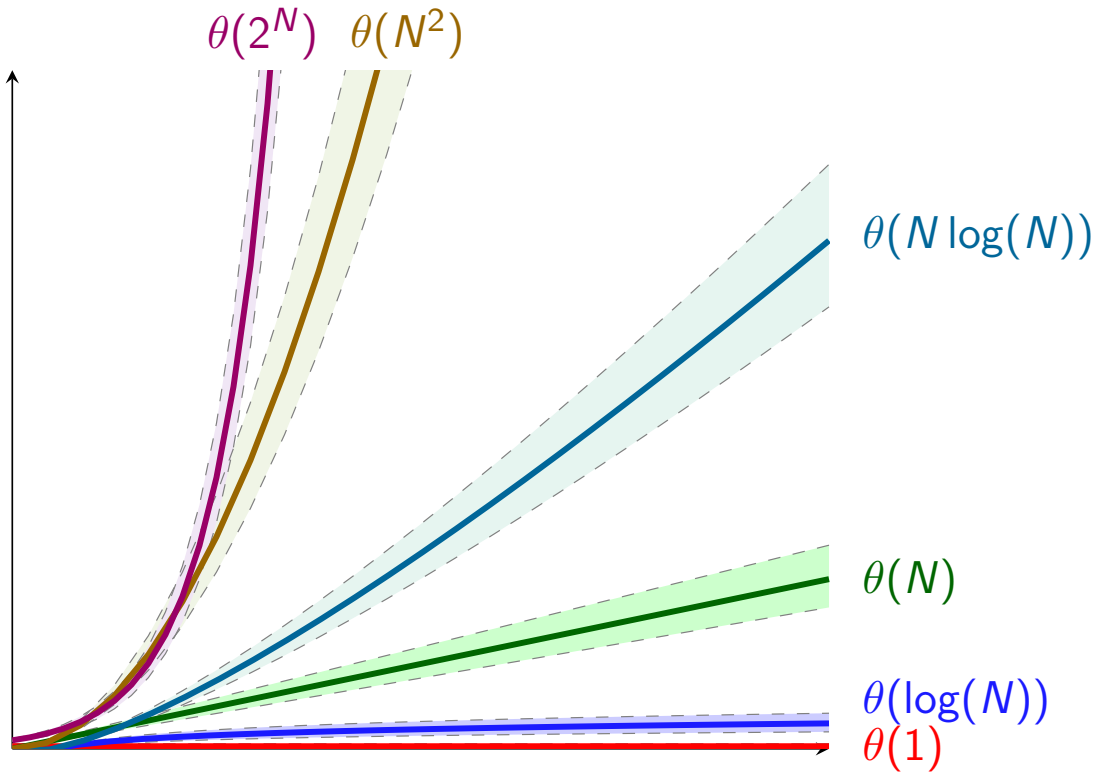
# Reminders

- WA3 due Sun, Oct 15 at 11:59 PM

# Graphs

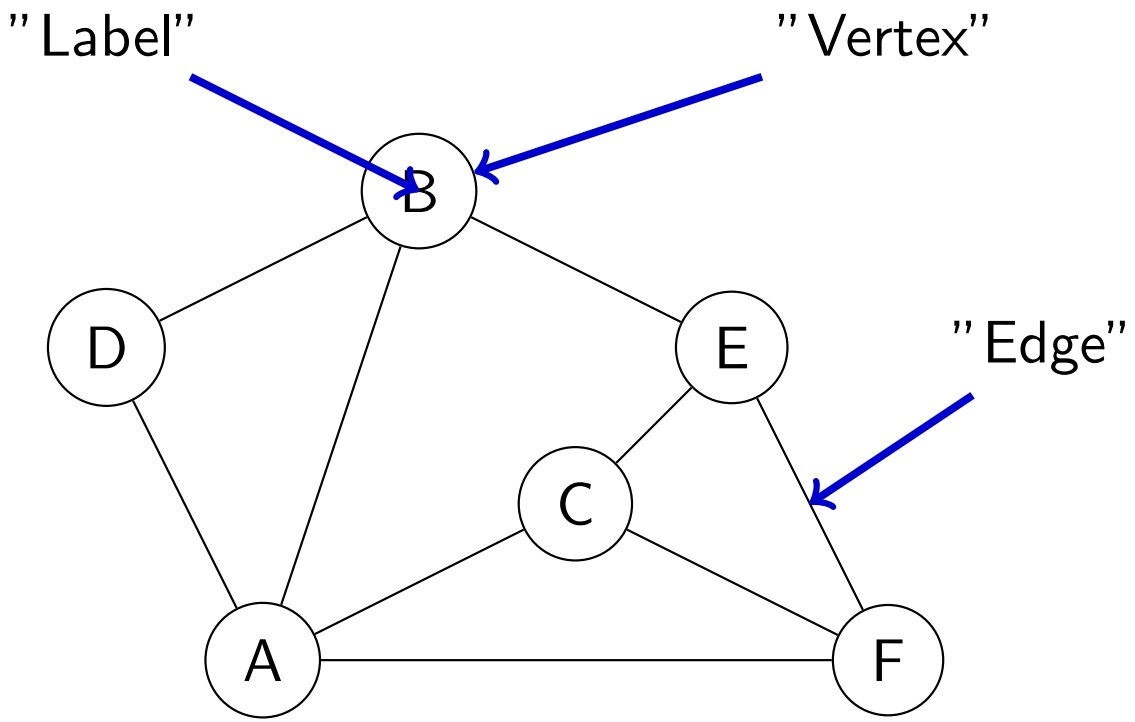
**What is a graph?**

# Graphs




**Not this kind of graph.**

# Graphs



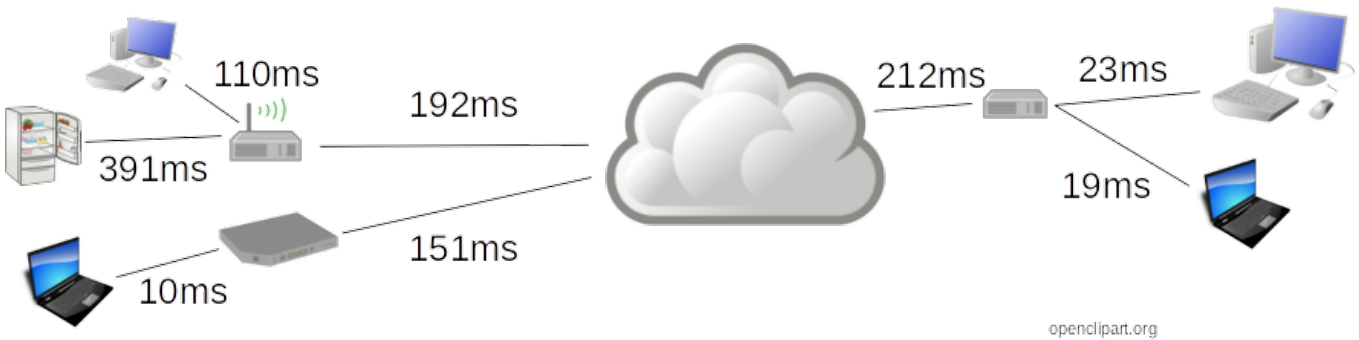
# Graphs

A **graph** is a pair (V, E), where

- $V$ is a set of **vertices** (sometimes nodes)
- $E$ is a set of vertex pairs called **edges**
- Edges and vertices may also store data (**labels**)

# Graph Examples

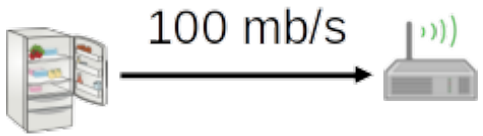
**Example:** A computer network
(edges store ping, nodes store IP addresses)



# Edge Types

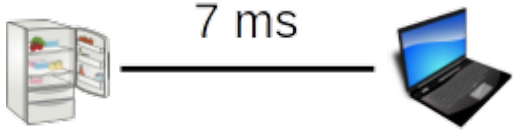
**Directed Edge** (e.g., transmit bandwidth)

- Ordered pair of vertices $(u, v)$
- origin $(u) \rightarrow$ destination $(v)$




**Undirected Edge** (e.g., latency)

- Unordered pair of vertices $(u, v)$




**Directed Graph**

- All edges are directed

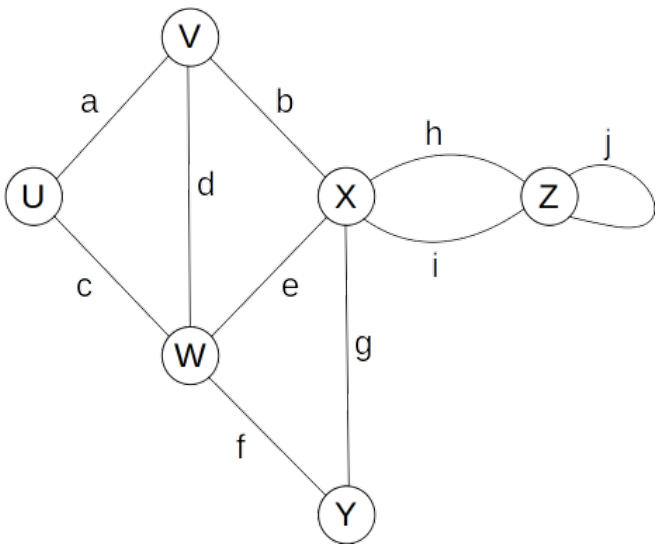
**Undirected Graph**

- All edges are undirected

# Other Applications of Graphs

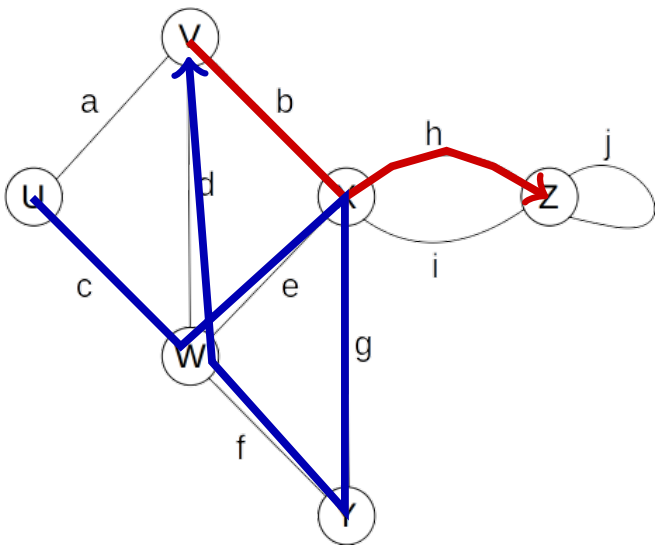
- Transportation (Flight/Road/Rail Routing)
- Protein/Protein interactions
- Social Networks
- Dependency Tracking
- Game Development (Planning, Routefinding)
- Taxonomies

# Graph Terminology




- **Endpoints of an edge**
  $U, V$ are the endpoints of $a$.
- **Edges incident on a vertex**
  $a, b, d$ are incident on $V$.
- **Adjacent Vertices**
  $U, V$ are adjacent.
- **Degree of a vertex (# of incident edges)**
  $X$ has degree 5.
- **Parallel Edges** (same endpoints)
  $h, i$ are parallel.
- **Self-loop** (same vertex is start and end)
  $j$ is a self-loop.
- **Simple Graph**
  A graph with no parallel edges or self-loops.

# Paths



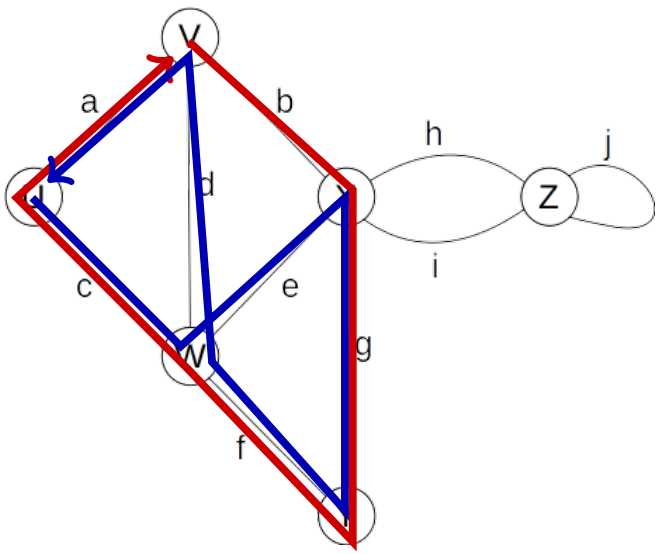

- **Path**

  A sequence of alternating vertices and edges
  - Begins with a vertex
  - Ends with a vertex
  - Each edge is preceded/followed by its endpoints
- **Simple Path**

  A path that never crosses the same vertex/edge twice
- **Examples**

  $V, b, X, h, Z$ is a simple path.
  $U, c, W, e, X, g, Y, f, W, d, V$ is a path that is not simple.

# Cycles




- **Cycle**
  A path that starts and ends on the same vertex.
  - Must contain at least one edge
- **Simple Cycle**
  A cycle where all of the edges and vertices are distinct (except the start/end vertex).
- **Examples**
  $V, b, X, g, Y, f, W, c, U, a, V$ is a simple cycle.
  $U, c, W, e, X, g, Y, f, W, d, V, a, U$ is a cycle that is not simple.

# Notation

- $N$: The number of vertices
- $M$: The number of edges
- $\deg(v)$: The degree of a vertex

# Handshake Theorem

$$\sum_{v \in V} \deg(v) = 2M$$

**Proof (sketch)**: Each edge adds 1 to the degree of 2 vertices.

# Edge Limit

In a directed graph with no self-loops and no parallel edges:

$$M \leq N \cdot (N - 1)$$

**Proof (sketch)**:

- Each pair is connected at most once (no parallel edges)
- $N$ possible start vertices
- $(N - 1)$ possible end vertices (no self-loops)
- $N \cdot (N - 1)$ distinct combinations possible

# Hey...

**Isn't this supposed to be a data structures class?**

# The Directed Graph ADT

## Interfaces

- `Graph<V, E>`
  - `V`: The vertex *label* type.
  - `E`: The edge *label* type.
- `Vertex<V, E>`
  - ... represents a single element (like a LinkedListNode)
  - ... stores a single value of type `V`
- `Edge<V, E>`
  - ... represents an edge (a pair of vertices)
  - ... stores a single value of type `E`

# The Directed Graph ADT

**What can we do with a (directed) graph?**

- Iterate over its vertices
- Iterate over its edges
- Add a vertex
- Add an edge
- Remove a vertex
- Remove an edge

# The Directed Graph ADT

```
1  interface Graph<V, E> {
2    public Iterator<Vertex> vertices()
3    public Iterator<Edge> edges()
4    public Vertex addVertex(V label): Vertex
5    public Edge addEdge(Vertex orig, Vertex dest, E label): Edge
6    public void removeVertex(Vertex vertex)
7    public void removeEdge(Edge edge)
8  }
```

# The Directed Graph ADT

**What can we do with a vertex?**

- What is the vertex's label?
- What edges are incident on the vertex?
- What edges is the vertex the origin of?
- What edges is the vertex the destination of?
- Is there an edge to another vertex?

# The Directed Graph ADT

```
1  interface Vertex<V, E> {
2    public V getLabel();
3    public Iterator<Edge> getOutEdges();
4    public Iterator<Edge> getInEdges();
5    public Iterator<Edge> getIncidentEdges();
6    public boolean hasEdgeTo(Vertex v);
7  }
```

# The Directed Graph ADT

**What can we do with an edge?**

- What is the edge's label?
- What is the edge's origin?
- What is the edge's destination?

# The Directed Graph ADT

```
1  interface Edge<V, E> {
2    public E getLabel();
3    public Vertex getOrigin();
4    public Vertex getDestination();
5  }
```

# Graph ADT Guarantees

**What guarantees does the ADT provide?**

- The origin and destination of each edge are in the graph.

# Graph Data Structures

What do we need to store for a graph $((V, E))$?

- A collection of vertices
- A collection of edges

# Edge List

```
1  class EdgeList<V, E> implements Graph<V, E>
2  {
3    List<Vertex> vertices = new ArrayList<Vertex>();
4    List<Edge>   edges    = new ArrayList<Edge>();
5
6    /*...*/
7  }
```

# addVertex

```
1  public Vertex addVertex(V label)
2  {
3    Vertex v = new Vertex(label);
4    vertices.append(v);
5    return v;
6  }
```

**What's the complexity?** $O(1)$ (amortized)

# addEdge

```
1  public Edge addEdge(Vertex origin, Vertex dest, E label)
2  {
3    Edge e = new Edge(origin, dest, label);
4    edges.append(e);
5    return e;
6  }
```

**What's the complexity?** $O(1)$ (amortized)

# removeEdge

```
1   public void removeEdge(Edge edge)
2   {
3     Iterator<Edge> i = edges.iterator();
4     while(i.hasNext())
5     {
6       if(i.next().equals(edge)){
7         i.remove();
8         return;
9       }
10    }
11  }
```

**What's the complexity?** $O(M)$

# Edge List

Could we do better?

# Edge List

```
1  class EdgeListV2<V, E> implements Graph<V, E>
2  {
3    List<Vertex> vertices = new LinkedList<Vertex>();
4    List<Edge>   edges    = new LinkedList<Edge>();
5
6    /*...*/
7  }
```

# removeEdge

```
1   public void removeEdge(Edge edge)
2   {
3     Iterator<Edge> i = edges.iterator();
4     while(i.hasNext())
5     {
6       if(i.next().equals(edge)){
7         i.remove();
8         return;
9       }
10    }
11  }
```

**What's the complexity?** $O(M)$

**Problem:** Finding the edge index/node is expensive

# Edge List

**Idea:** Store a reference to the linked list node

# Edge List

```
1  class EdgeListV2<V, E> implements Graph<V, E>
2  {
3    List<Vertex> vertices = new BetterLinkedList<Vertex>();
4    List<Edge>   edges    = new BetterLinkedList<Edge>();
5
6    /*...*/
7  }
```

# Edge List

```
1   class BetterLinkedList<T> implements List<T>
2   {
3     public Node<T> add(T element);
4       /* O(1) with tail pointer */
5
6     public void remove(Node<T> node)
7       /* O(1) */
8
9     public ListIterator<T> iterator
10      /* O(1) + O(1) per call to next */
11
12    /*...*/
13  }
```

# addVertex

```
1  class Vertex<V, E>
2  {
3    Node<Vertex> node = null;
4    /*...*/
5  }
```

```
1  public Vertex addVertex(V label)
2  {
3    Vertex vertex = new Vertex(label);
4    Node<Vertex> node = vertices.append(vertex);
5    vertex.node = node;
6    return vertex;
7  }
```

**What's the complexity?** $O(1)$

# addEdge

```
1  class Edge<V, E>
2  {
3    Node<Edge> node = null;
4    /*...*/
5  }
```

```
1  public Edge addEdge(Vertex origin, Vertex dest, E label)
2  {
3    Edge edge = new Edge(origin, dest, label);
4    Node<Edge> node = edges.append(edge);
5    edge.node = node;
6    return edge;
7  }
```

**What's the complexity?** $O(1)$

# removeEdge

```
1  public void removeEdge(Edge edge)
2  {
3    edges.remove(edge.node);
4    edge.node = null;
5  }
```

**What's the complexity?** $O(1)$

# removeVertex

```
1  public void removeVertex(Vertex vertex)
2  {
3    vertices.remove(vertex.node);
4    vertex.node = null;
5  }
```

**What's the complexity?** $O(1)$

# Enforcing Guarantees

Remember our guarantee: If an en edge is in the graph, its incident vertices must be too?
Which operations might violate this guarantee?

- `addVertex`: No violations possible
- `addEdge`: The edge's incident vertices might not be in the graph
- `removeEdge`: No violations possible
- `removeVertex`: The vertex might be incident on an edge.

# addEdge

```
1  public Edge addEdge(Vertex origin, Vertex dest, E label)
2  {
3    assert(origin.node != null);
4    assert(dest.node != null);
5    Edge edge = new Edge(label);
6    Node<Edge> node = edges.append(edge);
7    edge.node = node;
8    return edge;
9  }
```

**What's the complexity?** $O(1)$

# removeVertex

```
1  public void removeVertex(Vertex vertex)
2  {
3    for(edge : vertex.getIncidentEdges)
4    {
5      removeEdge(edge.node)
6    }
7    vertices.remove(vertex.node);
8    vertex.node = null;
9  }
```

**What's the complexity?**

$O(|\texttt{deg}(\texttt{vertex}) + T_{\texttt{getIncidentEdges}}(N, M))$

# getIncidentEdges

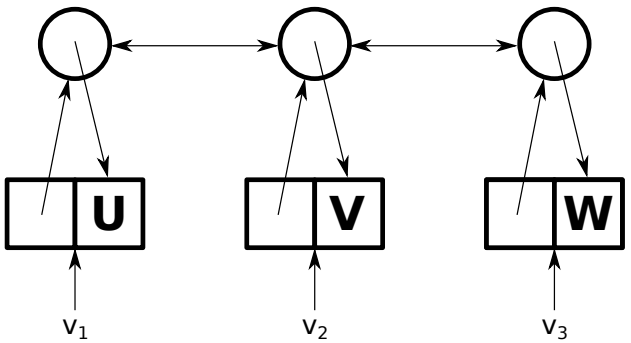
```
1   public Iterator<Edge> getIncidentEdges(Vertex vertex)
2   {
3     ArrayList<Edge> incidentEdges = new ArrayList<>();
4     for(edge : edges)
5     {
6       if(edge.origin.equals(vertex)
7          edge.dest.equals(vertex))
8       {
9         incidentEdges.add(edge);
10      }
11    }
12    return incidentEdges.iterator();
13  }
```

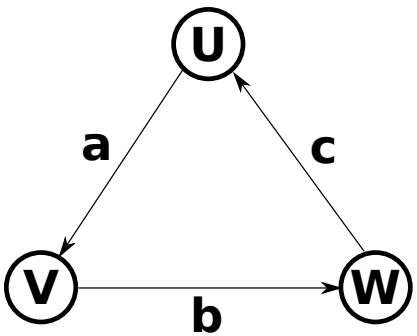
**What's the complexity?** $O(M) = O(N^2)$

# Edge List Summary

**LinkedList[Vertex]**

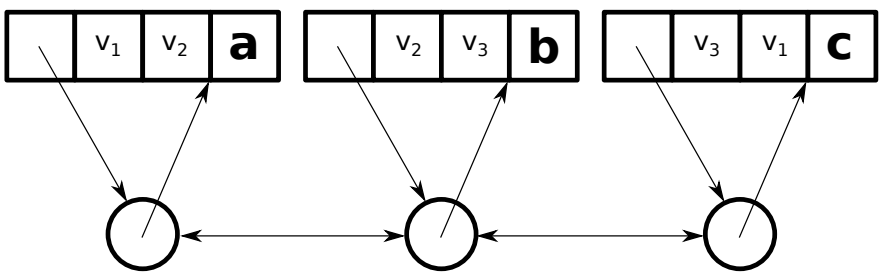
**Vertex**




**Edge**

**LinkedList[Edge]**



# Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(M)$
- `incidentEdges`: $O(M)$
- `hasEdgeTo`: $O(M)$

**Space Used**: $O(N + M)$
(constant space per vertex, edge)

# Improving on the Edge List

How can we avoid searching every edge in the edge list to find the incident edges?

**Idea:** Store each edges in/out edge list.

# Adjacency List

```
1  public class Vertex<V, E>
2  {
3    Node<Vertex> node = null;
4    List<Edge> inEdges = new BetterLinkedList<Edge>();
5    List<Edge> outEdges = new BetterLinkedList<Edge>();
6    /*...*/
7  }
```

# addEdge

```
1   public Edge addEdge(Vertex origin, Vertex dest, E label)
2   {
3     assert(origin.node != null);
4     assert(dest.node != null);
5     Edge edge = new Edge(label);
6     Node<Edge> node = edges.append(edge);
7     edge.node = node;
8     origin.outEdges.add(edge);
9     dest.inEdges.add(edge);
10    return edge;
11  }
```

**What's the complexity?** $O(1)$

# removeEdge

```
1   public void removeEdge(Edge edge)
2   {
3     edges.remove(edge.node);
4     for(iter : edge.orig.outEdges)
5     {
6       if(iter.next().equals(edge)){ iter.remove(); }
7     }
8     for(iter : edge.dest.inEdges)
9     {
10      if(iter.next().equals(edge)){ iter.remove(); }
11    }
12    edge.node = null;
13  }
```

**What's the complexity?** $O(M)$

# removeEdge

**Idea:** Store the outEdge/inEdge node with the Edge.

```
1  public class Edge<V, E>
2  {
3    Node<Edge> node = null;
4    Node<Edge> inNode = null;
5    Node<Edge> outNode = null;
6    /*...*/
7  }
```

# addEdge

```
1   public Edge addEdge(Vertex origin, Vertex dest, E label)
2   {
3     assert(origin.node != null);
4     assert(dest.node != null);
5     Edge edge = new Edge(label);
6     Node<Edge> node = edges.append(edge);
7     edge.node = node;
8     edge.outNode = origin.outEdges.add(edge);
9     edge.inNode = dest.inEdges.add(edge);
10    return edge;
11  }
```

**What's the complexity?** $O(1)$

# removeEdge

```
1  public void removeEdge(Edge edge)
2  {
3    edges.remove(edge.node);
4    edge.orig.outEdges.remove(edge.outNode);
5    edge.dest.inEdges.remove(edge.inNode);
6    edge.node = null;
7    edge.inNode = null;
8    edge.outNode = null;
9  }
```

**What's the complexity?** $O(1)$

# removeVertex

```
1   public void removeVertex(Vertex vertex)
2   {
3     for(edge : vertex.inEdges)
4     {
5       removeEdge(edge.node)
6     }
7     for(edge : vertex.outEdges)
8     {
9       removeEdge(edge.node)
10    }
11    vertices.remove(vertex.node);
12    vertex.node = null;
13  }
```

**What's the complexity?** $O(\texttt{deg(vertex)})$

# Adjacency List Summary

**Starting with an edge list**:

- Store a linked list of in/out edges with each vertex
- Store the linked list node for the in/out lists with each edge

# Edge List Summary

- `addEdge`, `addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(\texttt{deg}(v))$
- `incidentEdges`: $O(1) + O(1)$ per `next()`
- `hasEdgeTo`: $O(\texttt{deg}(v))$

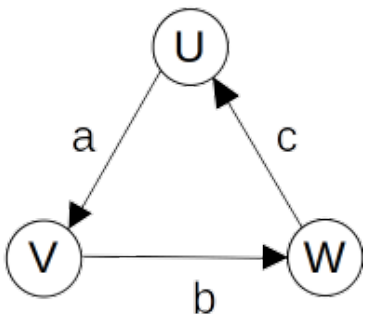
**Space Used**: $O(N + M)$
(constant space per vertex, edge)

# hasEdgeTo

Can we cut `hasEdgeTo` down to $O(1)$?

# The Adjacency Matrix Data Structure

| Origin \ Destination | U | V | W |
|---|---|---|---|
| U | - | a | - |
| V | - | - | b |
| W | c | - | - |



# Edge List Summary

- `addEdge`, `removeEdge`: $O(1)$
- `addVertex`, `removeVertex`: $O(N^2)$
- `incidentEdges`: $O(N)$
- `hasEdgeTo`: $O(1)$

**Space Used**: $O(N^2)$