CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lec 18: Graph ADT and EdgeLists

Announcements

- WA3 due on Sunday

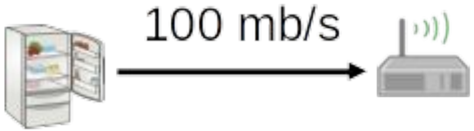
Edge Types

Directed Edge (asymmetric relationship)

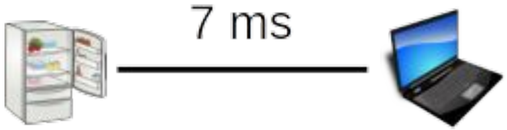
- Ordered pair of vertices (***u*, *v***)
- origin (***u***) →destination (***v***)

Undirected Edge (symmetric relationship)

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



transmit bandwidth



round-trip latency

Edge Types

Directed Edge (asymmetric relationship)

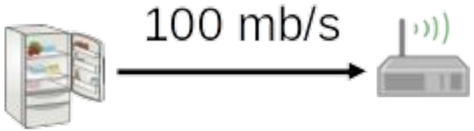
- Ordered pair of vertices (***u*****,** ***v***)
- origin (***u***) →destination (***v***)

Undirected Edge (symmetric relationship)

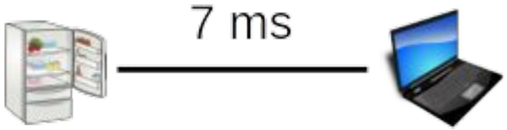
- Unordered pair of vertices (***u*****,*****v***)

Directed Graph: All edges are directed

Undirected Graph: All edges are undirected



transmit bandwidth



round-trip latency

Terminology

Endpoints of an edge

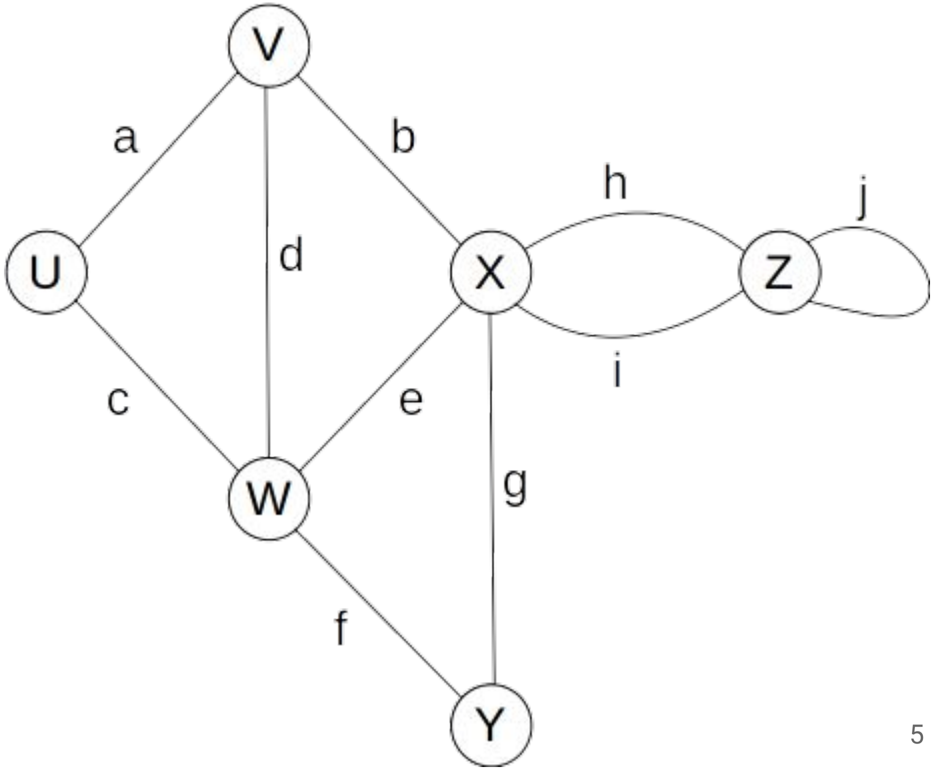
U, *V* are endpoints of *a*

Adjacent Vertices

U, *V* are adjacent

Degree of a vertex

X has degree 5



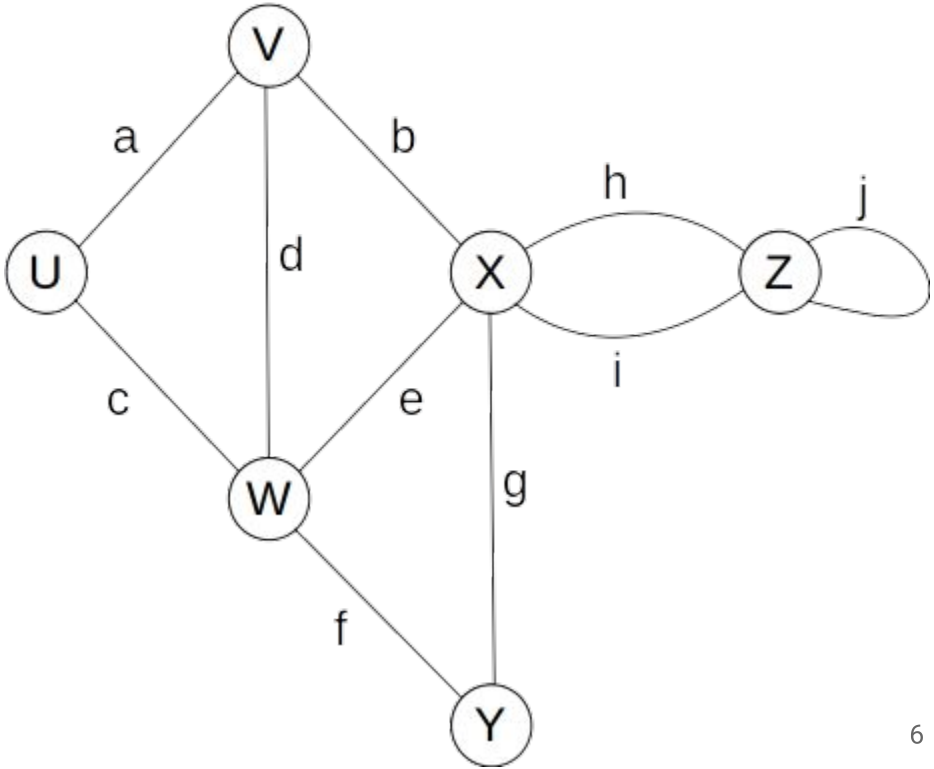
Terminology

Edges indecent on a vertex
a, *b*, *d* are incident on *V*

Parallel Edges
h, *i* are parallel

Self-Loop
j is a self-loop

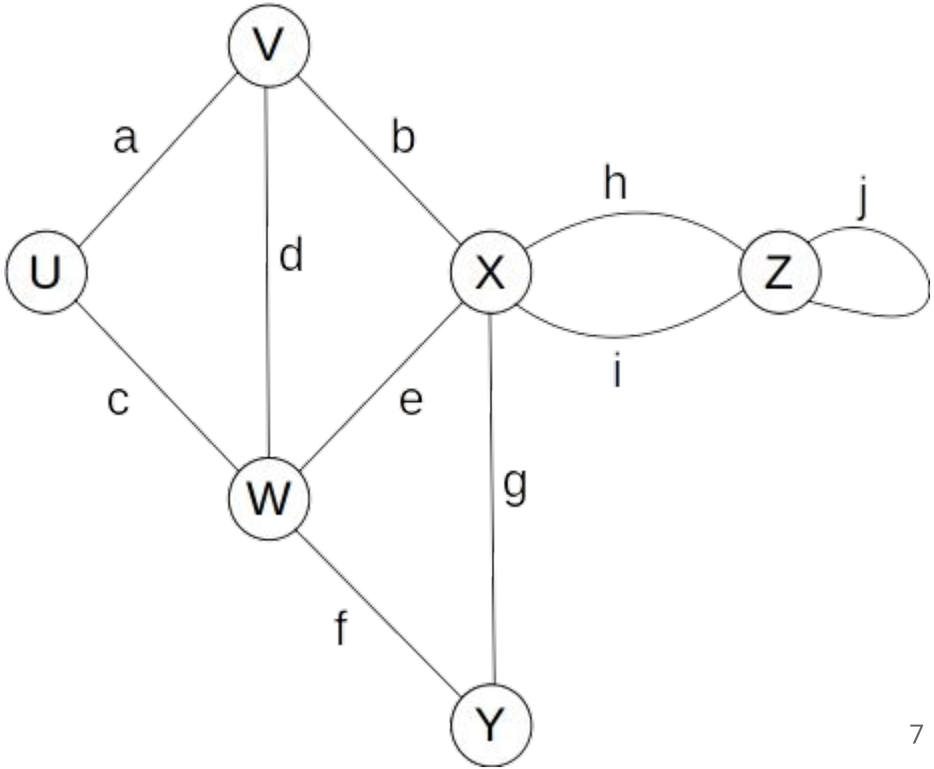
Simple Graph
A graph without parallel edges or self-loops



Terminology

Path
A sequence of alternating vertices and edges
- begins with a vertex
- ends with a vertex
- each edge preceded/followed by its endpoints

Simple Path
A path such that all of its vertices and edges are distinct

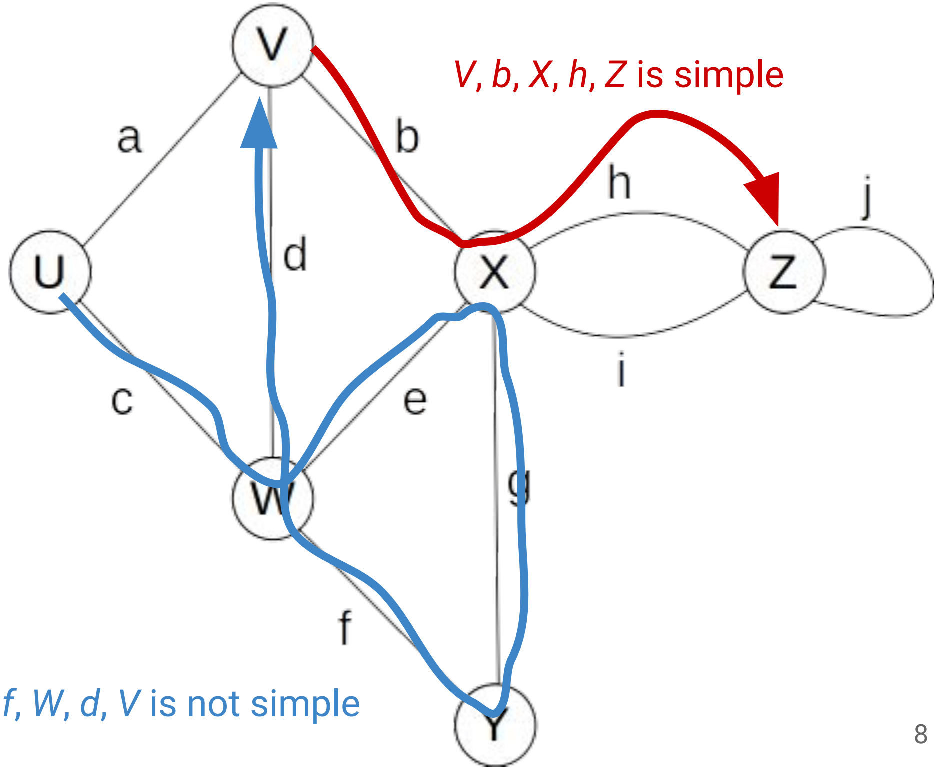


Terminology

Path
A sequence of alternating vertices and edges

- begins with a vertex
- ends with a vertex
- each edge preceded/followed by its endpoints

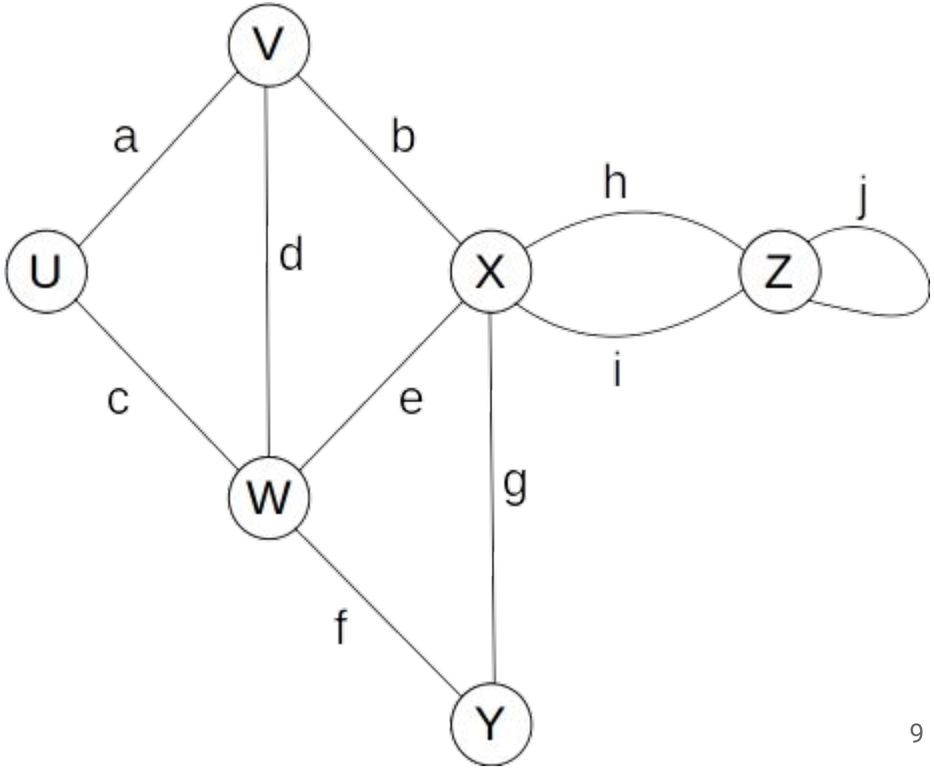
Simple Path
A path such that all of its vertices and edges are distinct



Terminology

Cycle
A path the begins and ends with the same vertex. Must contain at least one edge

Simple Cycle
A cycle such that all of its vertices and edges are distinct



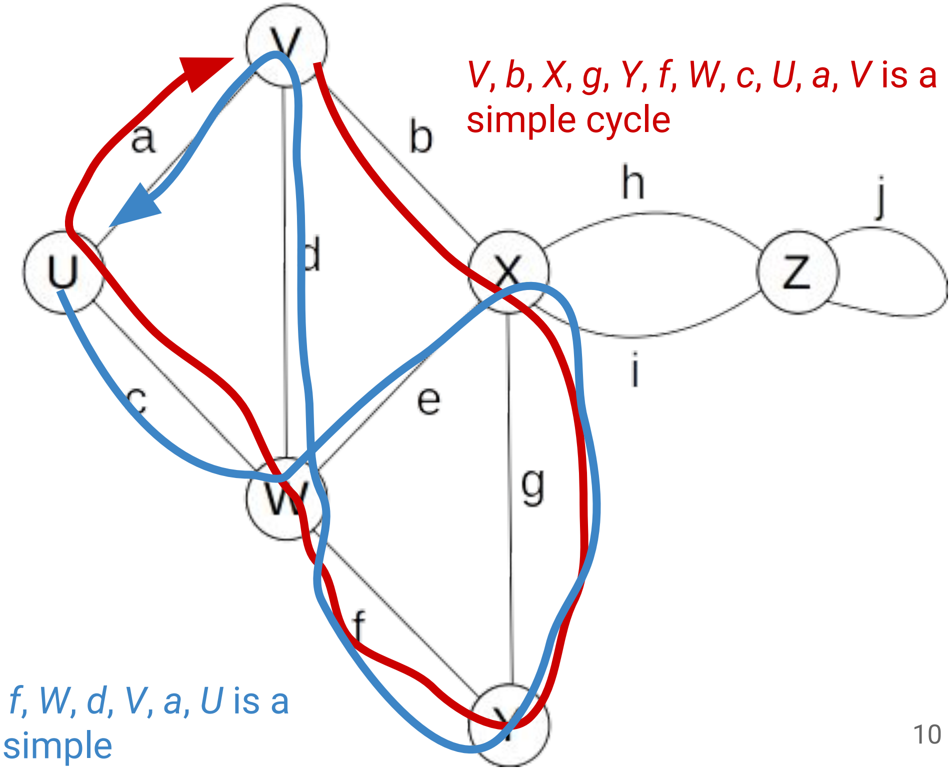
Terminology

Cycle

A path the begins and ends with the same vertex. Must contain at least one edge

Simple Cycle

A cycle such that all of its vertices and edges are distinct



Notation

n The number of vertices

m The number of edges

deg(*v*) The degree of vertex v

Graph Properties

$$\sum_{v} deg(v) = 2m$$

Graph Properties

$$\sum_{v} deg(v) = 2m$$

Proof: Each edge is counted twice

Graph Properties

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

$$m \leq n\,(n - 1)$$

Graph Properties

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

$$m \leq n\,(n - 1)$$

No parallel edges: each pair is connected at most once

No self-loops: pick each vertex only once

Graph Properties

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

$$m \leq n\ (n - 1)$$

No parallel edges: each pair is connected at most once

No self-loops: pick each vertex only once

n choices for the first vertex; **(*n* - 1)** choices for the second vertex. Therefore even if there was one edge between every possible pair, we still have at most ***n*(*n* - 1)** edges

A (Directed) Graph ADT

Two type parameters (`Graph[V,E]`)

`V`: The vertex label type

`E`: The edge label type

Vertices

…are elements

…store a value of type `V`

Edges

…are also elements

…store a value of type `E`

A (Directed) Graph ADT

What can we do with a Graph?

A (Directed) Graph ADT

What can we do with a Graph?

- Iterate through the vertices
- Iterate through the edges
- Add a vertex
- Add an edge
- Remove a vertex
- Remove an edge

A (Directed) Graph ADT

```java
public interface Graph<V, E> {
    public Iterator<Vertex> vertices();
    public Iterator<Edge> edges();
    public Vertex addVertex(V label);
    public Edge addEdge(Vertex orig, Vertex dest, E label);
    public void removeVertex(Vertex vertex);
    public void removeEdge(Edge edge);
}
```

A (Directed) Graph ADT

What can we do with a Vertex?

A (Directed) Graph ADT

What can we do with a Vertex?

- Get it's label
- Get the outgoing edges
- Get the incoming edges
- Get all incident edges
- Check if it's adjacent to another Vertex

A (Directed) Graph ADT

What can we do with an Edge?

- Get it's label
- Get the incident vertices

A (Directed) Graph ADT

```
public interface Vertex<V,E> {
  public V getLabel();
  public Iterator<Edge> getOutEdges();
  public Iterator<Edge> getInEdges();
  public Iterator<Edge> getIncidentEdges();
  public boolean hasEdgeTo(Vertex v);
}

public interface Edge<V,E> {
  public Vertex getOrigin();
  public Vertex getDestination();
  public E getLabel();
}
```

Implementation Attempt 1: Edge List

Data Model:

A List of Edges
(ArrayList)

A List of Vertices
(ArrayList)

Implementation Attempt 1: Edge List

```java
public class EdgeList<V,E> implements Graph<V,E> {
  List<Vertex> vertices = new ArrayList<Vertex>();
  List<Edge> edges = new ArrayList<Edge>();
  /*...*/
}
```

Implementation Attempt 1: Edge List

```
public Vertex addVertex(V label) {
  Vertex v = new Vertex(label);
  vertices.add(v);
  return v;
}

public Edge addEdge(Vertex orig, Vertex dest, E label) {
  Edge e = new Edge(orig, dest, label);
  edges.add(e);
  return e;
}
```

What's the complexity?

Implementation Attempt 1: Edge List

```java
public Vertex addVertex(V label) {
  Vertex v = new Vertex(label);
  vertices.add(v);
  return v;
}

public Edge addEdge(Vertex orig, Vertex dest, E label) {
  Edge e = new Edge(orig, dest, label);
  edges.add(e);
  return e;
}
```

Amortized $\Theta(1)$

Amortized $\Theta(1)$

What's the complexity?

Implementation Attempt 1: Edge List

```
public void removeEdge(Edge edge) {
  edges.remove(edge);
}
```

What's the complexity?

Implementation Attempt 1: Edge List

```
public void removeEdge(Edge edge) {
  edges.remove(edge);
}
```

What's the complexity?

We have to search for edge by value in an unsorted list! ***O*(*m*)**

Attempt 2: Linked Edge List

Data Model:

A List of Edges
(LinkedList)

A List of Vertices
(LinkedList)

Attempt 2: Linked Edge List

```java
public class LinkedEdgeList<V,E> implements Graph<V,E> {
  List<Vertex> vertices = new LinkedList<Vertex>();
  List<Edge> edges = new LinkedList<Edge>();
  /*...*/
}
```

Attempt 2: Linked Edge List

```
public Vertex addVertex(V label) {
  Vertex v = new Vertex(label);
  vertices.add(v);
  return v;
}

public Edge addEdge(Vertex orig, Vertex dest, E label) {
  Edge e = new Edge(orig, dest, label);
  edges.add(e);
  return e;
}
```

What's the complexity?

Attempt 2: Linked Edge List

```
public Vertex addVertex(V label) {
  Vertex v = new Vertex(label);
  vertices.add(v);
  return v;
}

public Edge addEdge(Vertex orig, Vertex dest, E label) {
  Edge e = new Edge(orig, dest, label);
  edges.add(e);
  return e;
}
```

$\Theta(1)$

$\Theta(1)$

What's the complexity?

Attempt 2: Linked Edge List

```
public void removeEdge(Edge<V,E> edge) {
  edges.remove(edge);
}
```

What's the complexity?

Attempt 2: Linked Edge List

```
public void removeEdge(Edge<V,E> edge) {
  edges.remove(edge);
}
```

What's the complexity?

We have to search for edge by value in an unsorted list! ***O*(*m*)**

Attempt 2: Linked Edge List

```
public void removeEdge(Edge<V,E> edge) {
  edges.remove(edge);
}
```

What's the complexity?

We have to search for edge by value in an unsorted list! ***O*(*m*)**

Solution: What if we stored a reference to the node?

Attempt 2: Linked Edge List

```
public class LinkedEdgeList<V,E> implements Graph<V,E> {
  List<Vertex> vertices = new CustomLinkedList<Vertex>();
  List<Edge> edges = new CustomLinkedList<Edge>();
  /*...*/
}
```

```
public class Vertex<V,E> {
  private Node<Vertex> node;
  /*...*/
}
public class Edge<V,E> {
  private Node<Edge> node;
  /*...*/
}
```

Attempt 2: Linked Edge List

```java
public Vertex addVertex(V label) {
  Vertex v = new Vertex(label);
  Node<Vertex> node = vertices.add(v);
  v.node = node;
  return v;
}

public Edge addEdge(Vertex orig, Vertex dest, E label) {
  Edge e = new Edge(orig, dest, label);
  Node<Edge> node = edges.add(e);
  e.node = node;
  return e;
}
```

Attempt 2: Linked Edge List

```java
public Vertex addVertex(V label) {
  Vertex v = new Vertex(label);
  Node<Vertex> node = vertices.add(v);
  v.node = node;
  return v;
}

public Edge addEdge(Vertex orig, Vertex dest, E label) {
  Edge e = new Edge(orig, dest, label);
  Node<Edge> node = edges.add(e);
  e.node = node;
  return e;
}
```

Both add methods still $\Theta(1)$

Attempt 2: Linked Edge List

```java
public void removeEdge(Edge edge) {
  edges.remove(edge.node);
}
```

What's the complexity?

Attempt 2: Linked Edge List

```java
public void removeEdge(Edge edge) {
  edges.remove(edge.node);
}
```

What's the complexity? $\Theta(1)$

Attempt 2: Linked Edge List

```
public void removeVertex(Vertex vertex) {
  vertices.remove(vertex.node);
}
```

What's the complexity?

Attempt 2: Linked Edge List

```
public void removeVertex(Vertex vertex) {
  vertices.remove(vertex.node);
}
```

What's the complexity? $\Theta(1)$

What's the problem?

Attempt 2: Linked Edge List

```java
public void removeVertex(Vertex vertex) {
  vertices.remove(vertex.node);
}
```

What's the complexity? $\Theta(1)$

What's the problem? The removed vertex may be incident to edges, which now have an endpoint that is not in the graph!

Attempt 2: Linked Edge List

```
public void removeVertex(Vertex vertex) {
  for(edge : vertex.getIncidentEdges()) {
    removeEdge(edge.node)
  }
  vertices.remove(vertex.node);
}
```

What's the complexity?

Attempt 2: Linked Edge List

```
public void removeVertex(Vertex vertex) {
  for(edge : vertex.getIncidentEdges()) {
    removeEdge(edge.node)
  }
  vertices.remove(vertex.node);
}
```

How do we get the incident edges with our current model?

What's the complexity?

Attempt 2: Linked Edge List

```java
public Iterator<Edge> getIncidentEdges(Vertex vertex) {
 ArrayList<Edge> incidentEdges = new ArrayList<>();
 for(edge : edges) {
   if(edge.origin.equals(vertex) || edge.dest.equals(vertex)) {
     incidentEdges.add(edge);
   }
 }
 return incidentEdges.iterator();
}
```

What is the complexity?

Attempt 2: Linked Edge List

```
public Iterator<Edge> getIncidentEdges(Vertex vertex) {
 ArrayList<Edge> incidentEdges = new ArrayList<>();
 for(edge : edges) {
   if(edge.origin.equals(vertex) || edge.dest.equals(vertex)) {
     incidentEdges.add(edge);
   }
 }
 return incidentEdges.iterator();
}
```

What is the complexity? ***O(m)***

Attempt 2: Linked Edge List

```
public void removeVertex(Vertex vertex) {
  for(edge : vertex.getIncidentEdges()) {
    removeEdge(edge.node)
  }
  vertices.remove(vertex.node);
}
```

How do we get the incident edges with our current model?

What's the complexity? $\boldsymbol{O(m) = O(n^2)}$

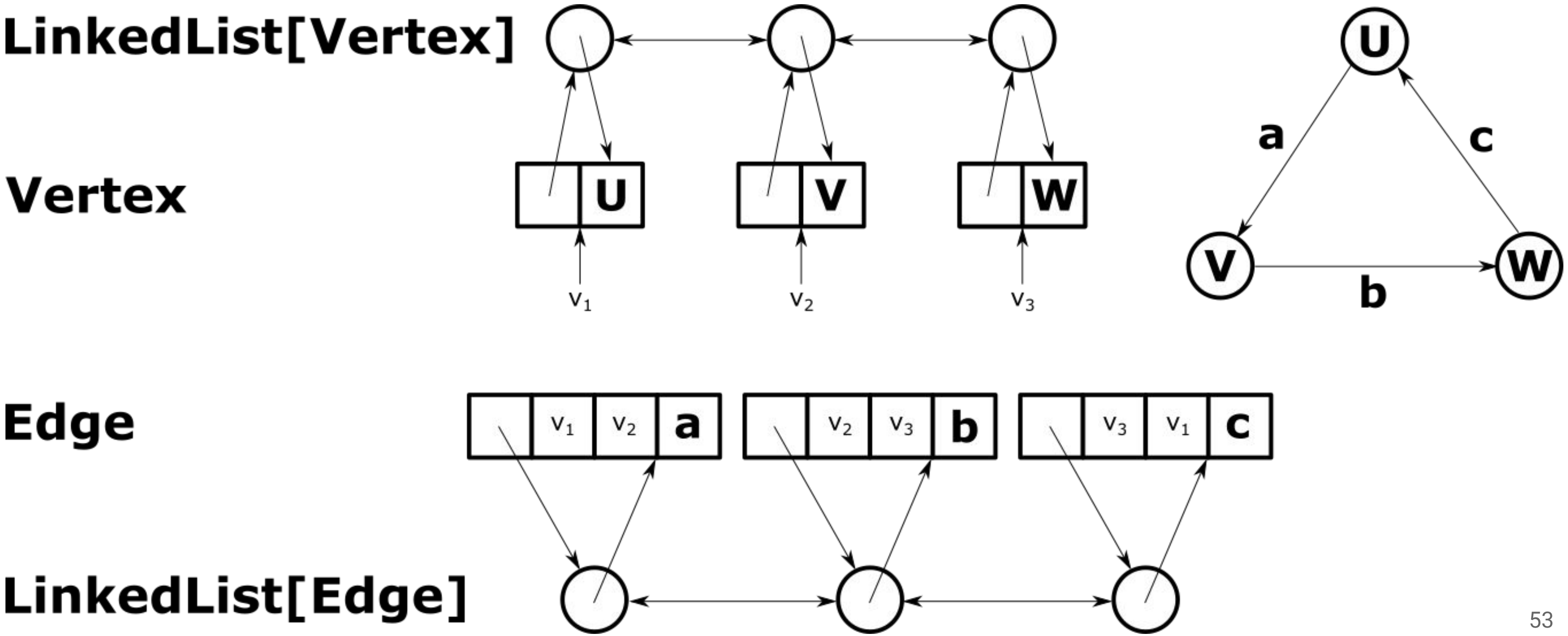
Edge List Summary

- `addEdge, addVertex`:
- `removeEdge`:
- `removeVertex`:
- `vertex.incidentEdges`:
- `vertex.edgeTo`:
- **Space Used:**

Edge List Summary

- `addEdge, addVertex`: $O(1)$
- `removeEdge`: $O(1)$
- `removeVertex`: $O(m)$
- `vertex.incidentEdges`: $O(m)$
- `vertex.edgeTo`: $O(m)$
- **Space Used:** $O(n) + O(m)$

Edge List Summary



How can we improve?

How can we improve?

Idea: Store the in/out edges for each vertex!

(Called an adjacency list)