Fundamentals of Data Structure

Mahesh Shirole

VJTI, Mumbai-19

Slides are prepared from

1. Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014
2. Data Structures and Algorithms in Java, by Robert Lafore, Second Edition, Sams Publishing

Data Structures for Graphs

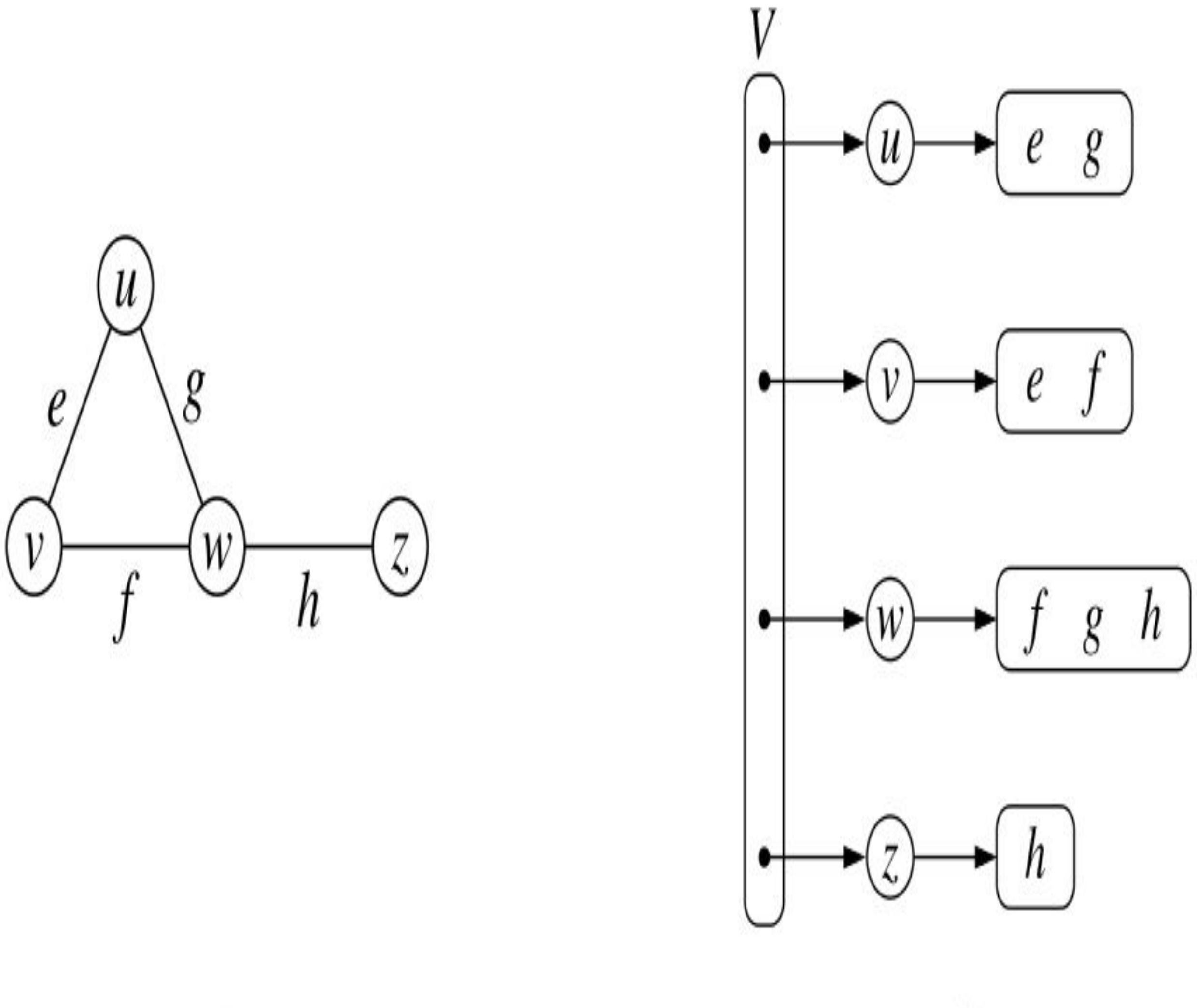
- There are four data structures for representing a graph
- In each representation, we maintain a collection to store the vertices of a graph
- The four representations differ greatly in the way they organize the edges
  - In an **edge list**, we maintain an unordered list of all edges. This minimally suffices, but there is no efficient way to locate a particular edge (u,v), or the set of all edges incident to a vertex v
  - In an **adjacency list**, we additionally maintain, for each vertex, a separate list containing those edges that are incident to the vertex. This organization allows us to more efficiently find all edges incident to a given vertex
  - An **adjacency map** is similar to an adjacency list, but the secondary container of all edges incident to a vertex is organized as a map, rather than as a list, with the adjacent vertex serving as a key. This allows more efficient access to a specific edge (u,v), for example, in O(1) expected time with hashing
  - An **adjacency matrix** provides worst-case O(1) access to a specific edge (u,v) by maintaining an n ×n matrix, for a graph with n vertices. Each slot is dedicated to storing a reference to the edge (u,v) for a particular pair of vertices u and v; if no such edge exists, the slot will store null

Adjacency List Structure

- The adjacency list structure for a graph *adds extra information to the edge list structure* that supports direct access to the incident edges (and thus to the adjacent vertices) of each vertex
- Specifically, for each vertex ***v***, we maintain a collection ***I(v)***, called the incidence collection of ***v***, whose entries are edges incident to ***v***
- In the case of a directed graph, outgoing and incoming edges can be respectively stored in two separate collections, $I_{out}(v)$ and $I_{in}(v)$
- Traditionally, the incidence collection ***I(v)*** for a vertex ***v*** is a list, which is why we call this way of representing a graph the adjacency list structure

Adjacency List Structure

- The *primary structure* for an adjacency list maintain the collection ***V*** of vertices in a way so that we can locate the *secondary structure* ***I(v)*** for a given vertex ***v*** in ***O(1)*** time
- We use a positional list to represent ***V***, with each Vertex instance maintaining a direct reference to its ***I(v)*** incidence collection
- We use a primary array-based structure to access the appropriate secondary lists



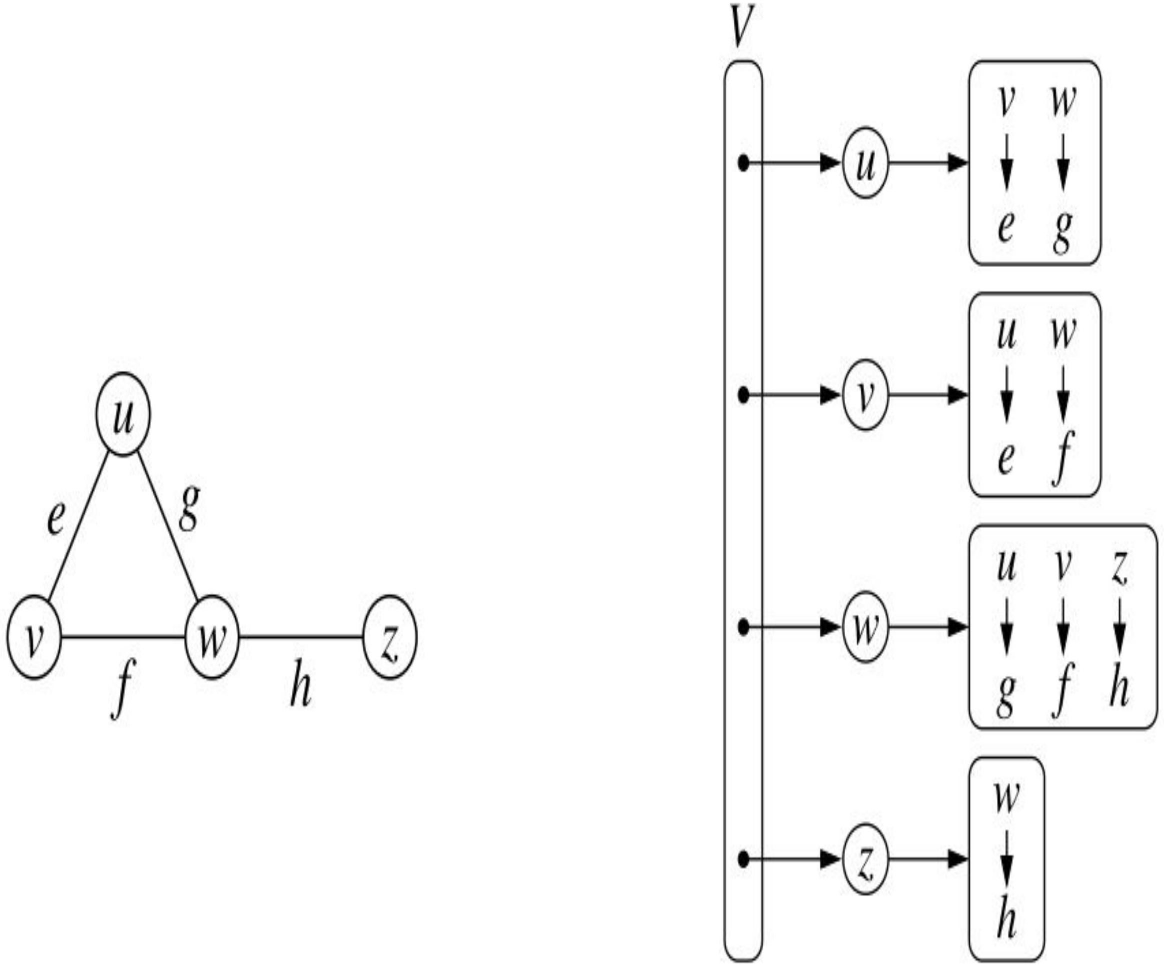
(a) (b)

Performance of the Adjacency List Structure

Method	Running Time
numVertices( ), numEdges( )	$O(1)$
vertices( )	$O(n)$
edges( )	$O(m)$
getEdge(u, v)	$O(\min(\deg(u), \deg(v)))$
outDegree(v), inDegree(v)	$O(1)$
outgoingEdges(v), incomingEdges(v)	$O(\deg(v))$
insertVertex(x), insertEdge(u, v, x)	$O(1)$
removeEdge(e)	$O(1)$
removeVertex(v)	$O(\deg(v))$

Adjacency Map Structure

- We can improve the performance of the adjacency list structure by using a hash-based map to implement ***I(v)*** for each vertex ***v***
- Specifically, we let the opposite endpoint of each incident edge serve as a key in the map, with the edge structure serving as the value
- The advantage of the adjacency map, relative to an adjacency list, is that the ***getEdge(u, v)*** method can be implemented in expected ***O(1)*** time by searching for vertex ***u*** as a key in ***I(v)***, or vice versa

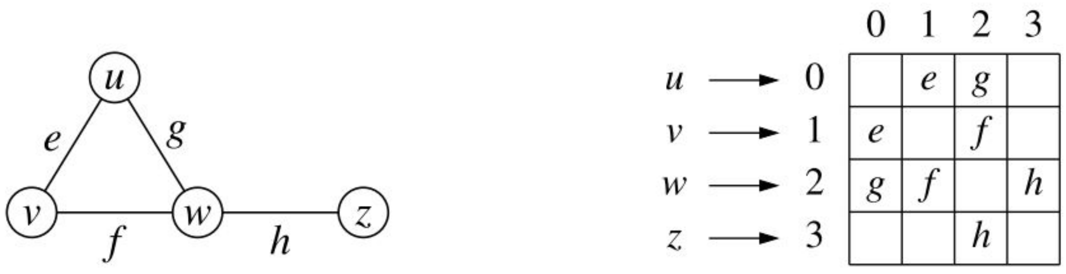


(a) (b)

Performance of the Adjacency Map Structure

Method	Edge List	Adj. List	Adj. Map	Adj. Matrix
numVertices()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
numEdges()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vertices()	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges()	$O(m)$	$O(m)$	$O(m)$	$O(m)$
getEdge(u, v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.	$O(1)$
outDegree(v) inDegree(v)	$O(m)$	$O(1)$	$O(1)$	$O(n)$
outgoingEdges(v) incomingEdges(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
insertVertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
removeVertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insertEdge(u, v, x)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$
removeEdge(e)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$

Adjacency Matrix Structure

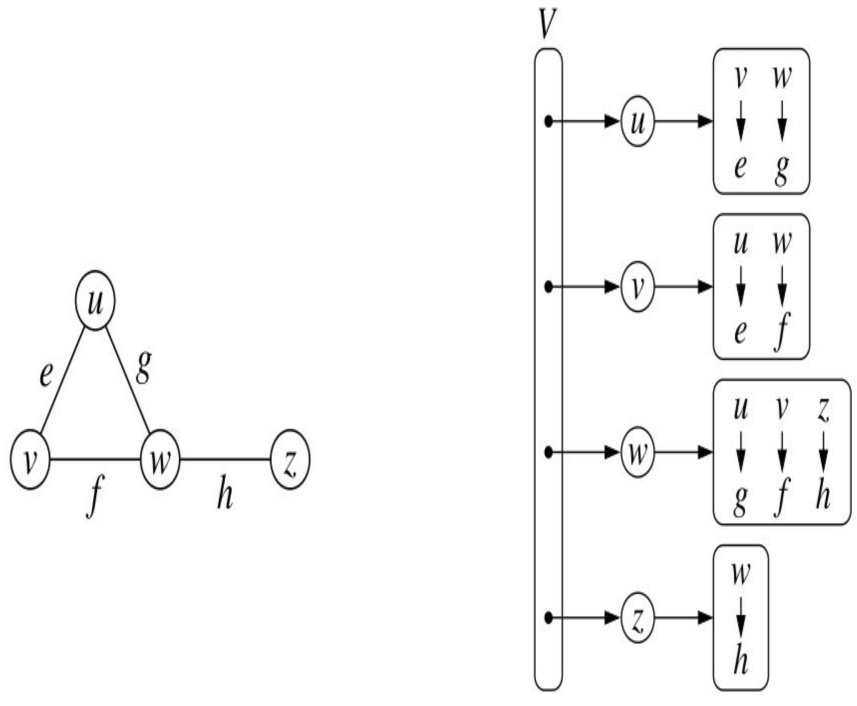


(a) (b)

- The adjacency matrix structure for a graph ***G*** augments the edge list structure with a matrix ***A***
- It allows us to locate an edge between a given pair of vertices in worst-case constant time
- In the adjacency matrix representation, we think of the vertices as being the integers in the set ***{0,1, . . . ,n−1}*** and the edges as being pairs of such integers
- This allows us to store references to edges in the cells of a two-dimensional ***n×n*** array ***A***
- Specifically, the cell ***A[i][j]*** holds a reference to the edge ***(u,v)***, if it exists, where ***u*** is the vertex with index ***i*** and ***v*** is the vertex with index ***j***. If there is no such edge, then ***A[i][j] = null***
- Array ***A*** is symmetric if graph ***G*** is undirected, as ***A[i][j] = A[j][i]*** for all pairs ***i*** and ***j***

Java Implementation

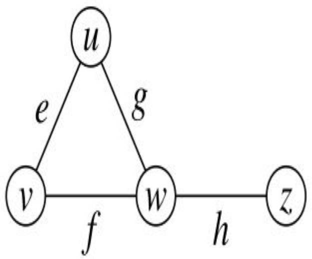
- An implementation of the Graph ADT, based on the adjacency map representation
- We use positional lists to represent each of the primary lists ***V*** and ***E***, as described in the edge list representation
- Additionally, for each vertex ***v***, we use a hash-based map to represent the secondary incidence map ***I(v)***
- To gracefully support both undirected and directed graphs, each vertex maintains two different map references: ***outgoing*** and ***incoming***
- In the directed case, these are initialized to two distinct map instances, representing $I_{out}(v)$ and $I_{in}(v)$
- In the case of an undirected graph, we assign both outgoing and incoming as aliases to a single map instance



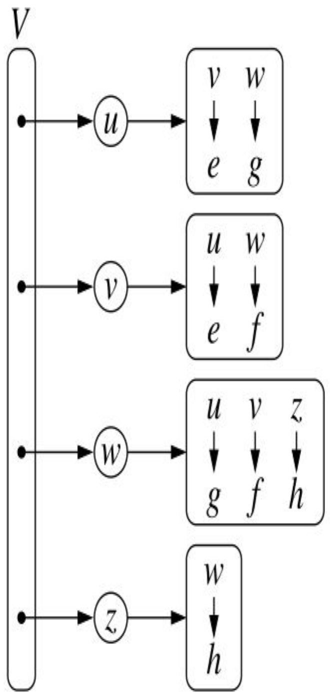
(a) (b)

Java Implementation

- Our implementation is organized as follows
- Define ***Vertex***, ***Edge***, and ***Graph***
- Define a concrete *AdjacencyMapGraph* class, with nested classes *InnerVertex* and *InnerEdge* to implement the vertex and edge abstractions
- A graph instance maintains a boolean variable that designates whether the graph is directed, and it maintains the vertex list and edge list



(a)



(b)

Java Implementation - Inner Class Vertex

```
/** A vertex of an adjacency map graph representation. */
private class InnerVertex<V> implements Vertex<V> {
  private V element;
  private Position<Vertex<V>> pos;
  private Map<Vertex<V>, Edge<E>> outgoing, incoming;
  /** Constructs a new InnerVertex instance storing the given element. */
  public InnerVertex(V elem, boolean graphIsDirected) {
    element = elem;
    outgoing = new ProbeHashMap<>();
    if (graphIsDirected)
      incoming = new ProbeHashMap<>();
    else
      incoming = outgoing;                    // if undirected, alias outgoing map
  }
```

Java Implementation - Inner Class Vertex

```
/** Returns the element associated with the vertex. */
public V getElement() { return element; }
/** Stores the position of this vertex within the graph's vertex list. */
public void setPosition(Position<Vertex<V>> p) { pos = p; }
/** Returns the position of this vertex within the graph's vertex list. */
public Position<Vertex<V>> getPosition() { return pos; }
/** Returns reference to the underlying map of outgoing edges. */
public Map<Vertex<V>, Edge<E>> getOutgoing() { return outgoing; }
/** Returns reference to the underlying map of incoming edges. */
public Map<Vertex<V>, Edge<E>> getIncoming() { return incoming; }
} //----------- end of InnerVertex class -----------
```

Java Implementation - Inner Class Edge

```
/** An edge between two vertices. */
private class InnerEdge<E> implements Edge<E> {
  private E element;
  private Position<Edge<E>> pos;
  private Vertex<V>[ ] endpoints;
  /** Constructs InnerEdge instance from u to v, storing the given element. */
  public InnerEdge(Vertex<V> u, Vertex<V> v, E elem) {
    element = elem;
    endpoints = (Vertex<V>[ ]) new Vertex[ ]{u,v};  // array of length 2
  }
```

Java Implementation - Inner Class Edge

```
  /** Returns the element associated with the edge. */
  public E getElement() { return element; }
  /** Returns reference to the endpoint array. */
  public Vertex<V>[ ] getEndpoints() { return endpoints; }
  /** Stores the position of this edge within the graph's vertex list. */
  public void setPosition(Position<Edge<E>> p) { pos = p; }
  /** Returns the position of this edge within the graph's vertex list. */
  public Position<Edge<E>> getPosition() { return pos; }
} //----------- end of InnerEdge class -----------
```

Java Implementation - AdjacencyMapGraph

```java
public class AdjacencyMapGraph<V,E> implements Graph<V,E> {
  // nested InnerVertex and InnerEdge classes defined here...
  private boolean isDirected;
  private PositionalList<Vertex<V>> vertices = new LinkedPositionalList<>();
  private PositionalList<Edge<E>> edges = new LinkedPositionalList<>();
  /** Constructs an empty graph (either undirected or directed). */
  public AdjacencyMapGraph(boolean directed) { isDirected = directed; }
  /** Returns the number of vertices of the graph */
  public int numVertices() { return vertices.size(); }
  /** Returns the vertices of the graph as an iterable collection */
  public Iterable<Vertex<V>> vertices() { return vertices; }
  /** Returns the number of edges of the graph */
  public int numEdges() { return edges.size(); }
  /** Returns the edges of the graph as an iterable collection */
  public Iterable<Edge<E>> edges() { return edges; }
```

Java Implementation - AdjacencyMapGraph

```
/** Returns the number of edges for which vertex v is the origin. */
public int outDegree(Vertex<V> v) {
  InnerVertex<V> vert = validate(v);
  return vert.getOutgoing().size();
}
/** Returns an iterable collection of edges for which vertex v is the origin. */
public Iterable<Edge<E>> outgoingEdges(Vertex<V> v) {
  InnerVertex<V> vert = validate(v);
  return vert.getOutgoing().values();  // edges are the values in the adjacency map
}
/** Returns the number of edges for which vertex v is the destination. */
public int inDegree(Vertex<V> v) {
  InnerVertex<V> vert = validate(v);
  return vert.getIncoming().size();
}
```

Java Implementation - AdjacencyMapGraph

```
/** Returns an iterable collection of edges for which vertex v is the destination. */
public Iterable<Edge<E>> incomingEdges(Vertex<V> v) {
  InnerVertex<V> vert = validate(v);
  return vert.getIncoming( ).values( );  // edges are the values in the adjacency map
}
public Edge<E> getEdge(Vertex<V> u, Vertex<V> v) {
/** Returns the edge from u to v, or null if they are not adjacent. */
  InnerVertex<V> origin = validate(u);
  return origin.getOutgoing( ).get(v);  // will be null if no edge from u to v
}
/** Returns the vertices of edge e as an array of length two. */
public Vertex<V>[ ] endVertices(Edge<E> e) {
  InnerEdge<E> edge = validate(e);
  return edge.getEndpoints( );
}
```

Java Implementation - AdjacencyMapGraph

```
/** Returns the vertex that is opposite vertex v on edge e. */
public Vertex<V> opposite(Vertex<V> v, Edge<E> e)
                                        throws IllegalArgumentException {
  InnerEdge<E> edge = validate(e);
  Vertex<V>[ ] endpoints = edge.getEndpoints( );
  if (endpoints[0] == v)
    return endpoints[1];
  else if (endpoints[1] == v)
    return endpoints[0];
  else
    throw new IllegalArgumentException("v is not incident to this edge");
}
/** Inserts and returns a new vertex with the given element. */
public Vertex<V> insertVertex(V element) {
  InnerVertex<V> v = new InnerVertex<>(element, isDirected);
  v.setPosition(vertices.addLast(v));
  return v;
}
```

Java Implementation - AdjacencyMapGraph

```
/** Inserts and returns a new edge between u and v, storing given element. */
public Edge<E> insertEdge(Vertex<V> u, Vertex<V> v, E element)
                                          throws IllegalArgumentException {
  if (getEdge(u,v) == null) {
    InnerEdge<E> e = new InnerEdge<>(u, v, element);
    e.setPosition(edges.addLast(e));
    InnerVertex<V> origin = validate(u);
    InnerVertex<V> dest = validate(v);
    origin.getOutgoing().put(v, e);
    dest.getIncoming().put(u, e);
    return e;
  } else
    throw new IllegalArgumentException("Edge from u to v exists");
}
```

Java Implementation - AdjacencyMapGraph

```
/** Removes a vertex and all its incident edges from the graph. */
public void removeVertex(Vertex<V> v) {
  InnerVertex<V> vert = validate(v);
  // remove all incident edges from the graph
  for (Edge<E> e : vert.getOutgoing().values())
    removeEdge(e);
  for (Edge<E> e : vert.getIncoming().values())
    removeEdge(e);
  // remove this vertex from the list of vertices
  vertices.remove(vert.getPosition());
}
}
```