Implementation of Graphs

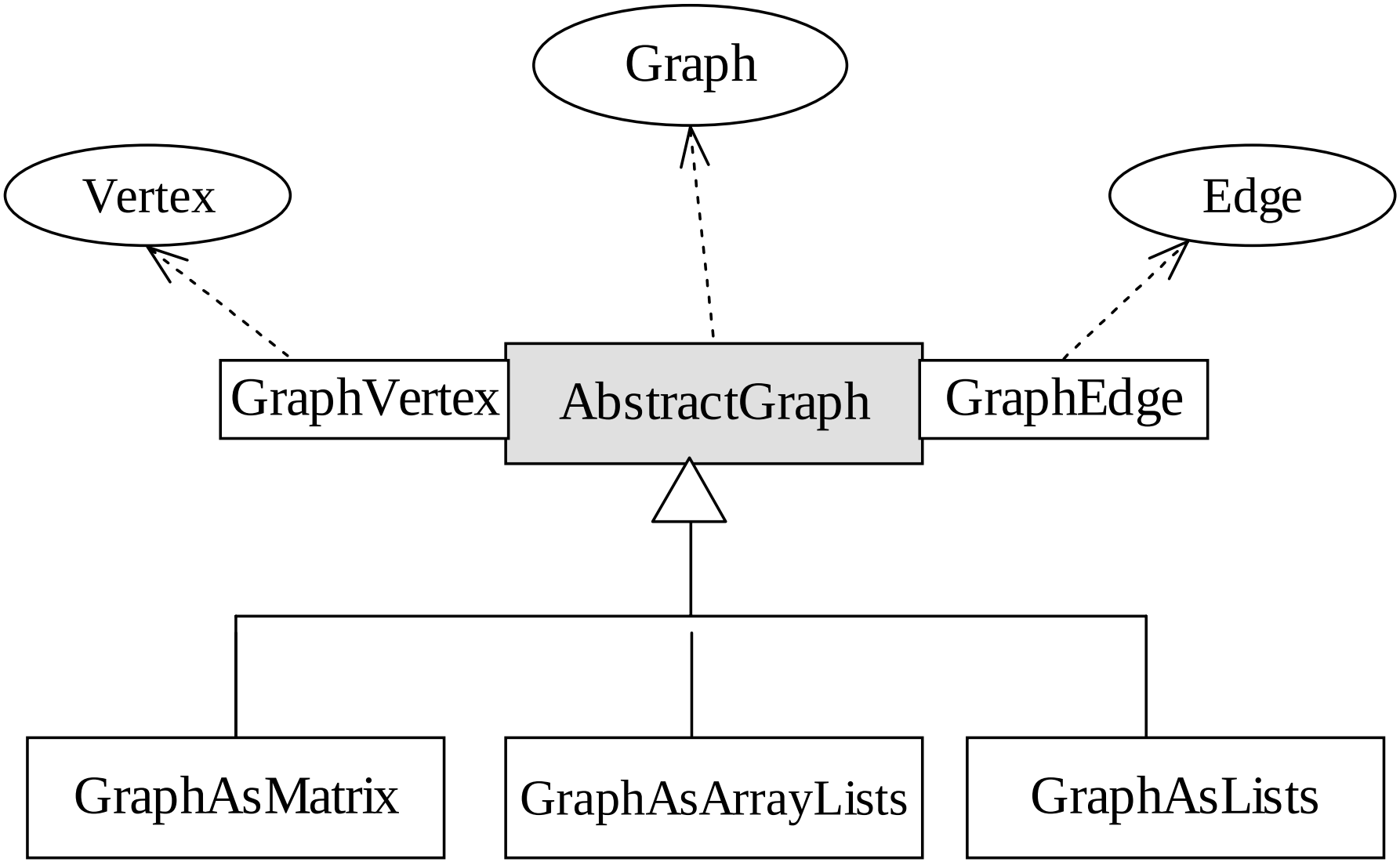
- Identification of Classes and Interfaces
- Concrete Implementations for Graph
- Review Questions.

Identification of Classes

- A graph contains vertices and edges.
- We can identify three kinds of objects: vertices, edges, and graphs.
- Accordingly, we define three interfaces:
  - Vertex
  - Edge
  - Graph

- A graph can be represented in different ways (we know three of them). Accordinglg, we use the following six classes
  - AbstractGraph (having the following two inner classes)
    - GraphVertex
    - GraphEdge
  - GraphAsMatrix
  - GraphAsArrayLists
  - GraphAsLists

Classes and Interfaces



The Vertex Interface

- Each vertex must be distinguishable from other vertices. Thus, each vertex should have a unique label.

- Some applications require vertex-weighted graphs.

- A vertex type object belongs to exactly one graph. It is neither independent nor shared between two graphs. Otherwise the getIncidentEdges(), and other methods will not make sense.

```
public interface Vertex  extends Comparable
{
    public String getLabel();
    public Comparable getWeight();
    public Iterator getIncidentEdges();
    public Iterator getEmanatingEdges();
    public Iterator getPredecessors();
    public Iterator getSuccessors();
}
```

The Edge Interface

- An edge in a directed graph is an ordered pair of vertices; while in an undirected graph it is a set of two vertices.

- We use the same class for both--the context determines whether it is directed or undirected.

- Some edges may have weight

```
public interface Edge extends Comparable
{
    public abstract Vertex getFromVertex();
    public abstract Vertex getToVertex();
    public abstract Comparable getWeight();
    public abstract boolean isDirected();
    public abstract Vertex getMate(Vertex vertex);

}
```

The Graph Interface

- It represents both directed and undirected graphs.

```
public interface Graph {
    public int getNumberOfEdges();
    public int getNumberOfVertices();
    public Iterator getVertices();
    public Iterator getEdges();

    public void addVertex(String label);
    public void addVertex(String label, Comparable weight);
    public Vertex getVertex(String label);
    public int getIndex(Vertex v);// vertices are indexed starting from zero

    public void addEdge(String from, String to);
    public void addEdge(String from, String to, Comparable weight);
    public Edge getEdge(String from, String to);
    public boolean isReachable(String from, String to);

    public boolean isDirected();
    public boolean isWeighted();
    public boolean isConnected();
    public abstract boolean isStronglyConnected();
    public abstract boolean isWeaklyConnected();
    public boolean isCyclic();
    public void preorderDepthFirstTraversal(Visitor visitor, Vertex start);
    public void postorderDepthFirstTraversal(Visitor visitor, Vertex start);
    public void breadthFirstTraversal(Visitor visitor, Vertex start);
    public abstract int topologicalOrderTraversal(Visitor visitor);
}
```

The AbstractGraph class

- The following introduces the AbstractGraph class.

```
public abstract class AbstractGraph implements Graph {
   protected int numberOfVertices;
   protected int numberOfEdges;
   protected boolean directed;

   public AbstractGraph(boolean directed){
        numberOfVertices = 0;
        numberOfEdges = 0;
       this.directed = directed;
   }
   public int getNumberOfVertices(){return numberOfVertices;}
   public int getNumberOfEdges(){return numberOfEdges;}
   public void purge() {
        numberOfVertices = 0;
        numberOfEdges = 0;
   }
   public void addVertex(String label){addVertex(label, null);}
   public void addEdge(String from, String to){addEdge(from, to, null);}

   public boolean isDirected() {return directed;}
```

implemented in subclasses

The AbstractGraph class- Cont'd

```
public boolean isWeighted(){
   Iterator p = getEdges();
   if(((Edge)p.next()).getWeight() == null) return false;
   return true;
}
public Vertex getVertex(String label){
   Iterator i = getVertices();
   while (i.hasNext()){
      Vertex v = (Vertex) i.next();
      if (v.getLabel().equals(label)) return v;
   }
   return null;
}
public Edge getEdge(String from, String to){
   Iterator i = getEdges();
   while (i.hasNext()){
      Edge e = (Edge) i.next();
      if (e.getFromVertex().getLabel().equals(from) &&
                              e.getToVertex().getLabel().equals(to))
         return e;
   }
   return null;
}
```

The AbstractGraph class- Cont'd

```
public Iterator getEmanatingEdges(Vertex from) {
   Iterator i = getEdges();
   MyLinkedList emEdges = new MyLinkedList();
   while (i.hasNext()){
      Edge edge = (Edge) i.next();
      if (edge.getFromVertex().equals(from)) emEdges.append(edge);
   }
   return emEdges.iterator();
}
public Iterator getIncidentEdges(Vertex to) {
   Iterator i = getEdges();
   MyLinkedList inEdges = new MyLinkedList();
   while (i.hasNext()){
      Edge edge = (Edge) i.next();
      if (edge.getToVertex().equals(to)) inEdges.append(edge);
   }
   return inEdges.iterator();
}

public int getIndex(Vertex v){ return getIndex(v.getLabel());}
protected abstract int getIndex(String label);
```

Other methods of the AbstractGraph class will be discussed in coming lectures

The GraphVertex class

- The GraphVertex class is implemented as an inner class:

```
protected final class GraphVertex implements Vertex  {
   protected String label; protected Comparable weight;
   protected GraphVertex(String s, Comparable w) {
      label = s; weight = w;
   }
   protected GraphVertex(String s) {this(s, null);}
   public int compareTo(Object obj) {
      return label.compareTo(((GraphVertex)obj).getLabel());
   }
   public Iterator getIncidentEdges() {
      return AbstractGraph.this.getIncidentEdges(this);
   }
   public Iterator getPredecessors() {
      return new Iterator() {
         Iterator edges = getIncidentEdges();
         public boolean hasNext() {return edges.hasNext();}
         public Object next() {
            Edge edge = (Edge)edges.next();
            return edge.getMate(GraphVertex.this);
         }
      };
   }
}
```

The getEmanatingEdges and getSuccessors methods are implemented in the same way.

The GraphEdge class

- The GraphEdge class is also implemented as an inner class:

```
protected final class GraphEdge implements Edge {
   protected Vertex startVertex, endVertex;
   protected Comparable weight;
   protected GraphEdge(Vertex v1, Vertex v2, Comparable w) {
      startVertex = v1; endVertex = v2; weight = w;
   }
   protected GraphEdge(Vertex v1, Vertex v2) {this(v1, v2, null);}
   public Vertex getFromVertex() {return startVertex;}
   public Vertex getToVertex() {return endVertex;}
   public Comparable getWeight() {return weight;}
   public Vertex getMate(Vertex v) {
      if(v.equals(startVertex)) return endVertex;
      if(v.equals(endVertex)) return startVertex;
      else throw new InvalidOperationException("invalid vertex");
   }

   public boolean isDirected() {
      return AbstractGraph.this.isDirected();
   }
   // …
}
```

Implementing GraphAsMatrix (Adjacency Matrix)

- The following describes the concrete class, GraphAsMatrix:

```
public class GraphAsMatrix extends AbstractGraph {
   private int size;
   private Vertex[] vertices;
   private Edge[][] edges;
   public GraphAsMatrix(int size, boolean directed) {
      super(directed);
      this.size = size;
      vertices = new GraphVertex[size];
      edges = new Edge[size][size];
   }
   public void purge()  {
      for (int i=0;i<size;i++){
         vertices[i] = null;
         for (int j=0;j<size;j++) edges[i][j] = null;
      }
      super.purge();
   }
   public int getIndex(String label){
      for (int i=0;i<numberOfVertices;i++)
         if (vertices[i].getLabel().equals(label)) return i;
      return -1;
   }
```

Implementing GraphAsMatrix - Cont'd

```
public void addVertex(String label, Comparable weight){
   if (getIndex(label)!=-1)
      throw new IllegalArgumentException("Duplicate vertex");
   if (numberOfVertices == size)
      throw new IllegalArgumentException("Graph is full");
   vertices[numberOfVertices++] = new GraphVertex(label, weight);
}

public void addEdge(String from, String to, Comparable weight){
   int i = getIndex(from);
   int j = getIndex(to);
   if (i==-1 || j==-1)
      throw new IllegalArgumentException("Vertex not in this graph");
   if (i == j)
      throw new IllegalArgumentException("Loops not supported");
   if (edges[i][j] == null){
      edges[i][j] = new GraphEdge(vertices[i], vertices[j], weight);
      numberOfEdges++;
      if (!isDirected() && edges[j][i]==null){
         edges[j][i]=new GraphEdge(vertices[j], vertices[i],weight);
         numberOfEdges++;
      }
   }
}
```

Implementing GraphAsMatrix - Cont'd

```
public Iterator getVertices(){
   return new Iterator(){
      int index = 0;
      public boolean hasNext(){return index < numberOfVertices;}
      public Object next(){return vertices[index++];}
   };
}
public Iterator getEdges() {
   return new Iterator(){
      int count = 0,i=0,j=0;
      public boolean hasNext(){return count < numberOfEdges;}
      public Object next(){
         if (count==numberOfEdges) throw new NoSuchElementException();
         while (i<numberOfVertices && j<numberOfVertices &&
                                                edges[i][j]==null){
            j++; if (j==numberOfVertices){j=0;i++;}
         }
         Edge r = edges[i][j];
         count++;
         // for next call, adust i and j
         j++; if (j==numberOfVertices){j=0;i++;}
         return r;
      }
   };
}
```

Implementing GraphAsLists (Simple List)

- The following describes the concrete class, GraphAsLists:

```
public class GraphAsLists extends AbstractGraph {
   private MyLinkedList listOfVertices, listOfEdges;
   public GraphAsLists(boolean directed) {
      super(directed);
      listOfVertices = new MyLinkedList();
      listOfEdges = new MyLinkedList();
   }
   public void purge()  {
      listOfVertices.purge();
      listOfEdges.purge();
      super.purge();
   }
   public int getIndex(String label){
      int index = -1;
      MyLinkedList.Element e = listOfVertices.getHead();
      while (e != null){
         index++;
         Vertex v = (Vertex) e.getData();
         if (label.equals(v.getLabel())) return index;
         e = e.getNext();
      }
      return -1;
   }
```

Implementing GraphAsLists – Cont'd

```
public void addVertex(String label, Comparable weight){
   if (getIndex(label)!=-1)
      throw new IllegalArgumentException("Duplicate vertex");
   listOfVertices.append(new GraphVertex(label, weight));
   numberOfVertices++;
}
public void addEdge(String from, String to, Comparable weight){
   Vertex fromVertex = getVertex(from);
   Vertex toVertex = getVertex(to);
   if (fromVertex==null || toVertex==null)
      throw new IllegalArgumentException("Vertex not in this graph");
   if (fromVertex == toVertex)
      throw new IllegalArgumentException("Loops not supported");
   if (getEdge(from, to)==null){
      listOfEdges.append(new GraphEdge(fromVertex, toVertex, weight));
      numberOfEdges++;
      if (!isDirected() && getEdge(to, from)==null){
         listOfEdges.append(new GraphEdge(toVertex, fromVertex, weight));
         numberOfEdges++;
      }
   }
}
public Iterator getEdges() {return listOfEdges.iterator();}
public Iterator getVertices(){return listOfVertices.iterator();}
```

Implementing GraphAsArrayLists (Adjacency List)

- The following describes implementation of graph as Adjacency List

```
public class GraphAsArrayLists extends AbstractGraph {
   private int size;
   private Vertex[] vertices;
   private MyLinkedList[] edges;
   public GraphAsArrayLists(int size, boolean directed) {
      super(directed);
      this.size = size;
      vertices = new GraphVertex[size];
      edges = new MyLinkedList[size];
      for (int i=0;i<size;i++) edges[i] = new MyLinkedList();
   }

   // These methods are similar to those in GraphAsMatrix class
   public int getIndex(String label)
   public void addVertex(String label, Comparable weight)
   public Iterator getVertices()

   // These methods will be implemented in the lab
   public void purge()
   public void addEdge(String from, String to, Comparable weight)
   public Iterator getEdges()
}
```

Simple Programming Questions on Graphs

1. Write an instance method **public Edge minWeightEdge()** in one of the concrete Graph classes that returns the minimum-weight edge. Your method must throw an appropriate exception if the graph is not weighted. Your method must not use any Iterator.

2. Write an instance method public int **countSpecialEdges()** of **AbstractGraph** that counts the number of edges in the invoking object that have starting vertex greater than ending vertex (based on compareTo method).

Solution for Q1 (In GraphAsMatrix class)

```
public Edge minWeightEdge(){
   boolean isFirstEdge = true;
   Edge min = null;
   for(int i = 0;i < numberOfVertices; i++){
      for(int j = 0;j < numberOfVertices; j++){
         if(edges[i][j]!=null){
            if(edges[i][j].getWeight()==null)
               throw new IllegalArgumentException(
                   "The graph is not weighted");

            if(isFirstEdge){
               min = edges[i][j];
               isFirstEdge = false;
            } else if(edges[i][j].getWeight().compareTo(
                      min.getWeight())<0)
               min = edges[i][j];
         }
      }
   }
   return min;
}
```