CSCI 136
Data Structures & Advanced Programming

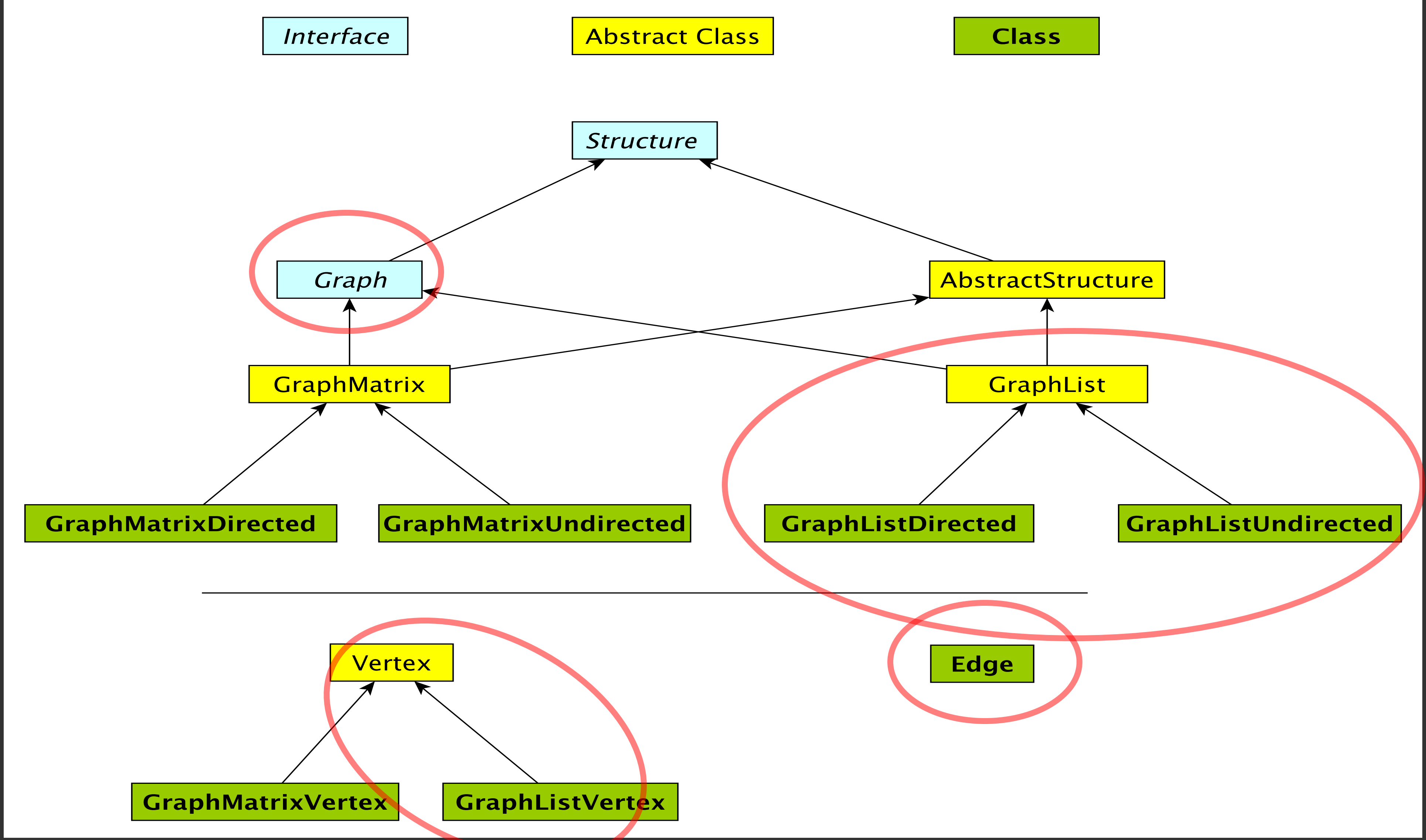
Implementing Graphs:
Adjacency Lists

Video Outline

- Graph Implementation Details
    - Adjacency Matrix – covered in another video
    - Adjacency List – covered in this video

- Time/Space Complexity

Graph Classes in structure5



GraphList: Big Picture

- Matrix representation wastes space storing non-edges

- Maintain an *adjacency list* of *edges* at each vertex (no adjacency matrix)
  - Keep only *outgoing* edges for directed graphs

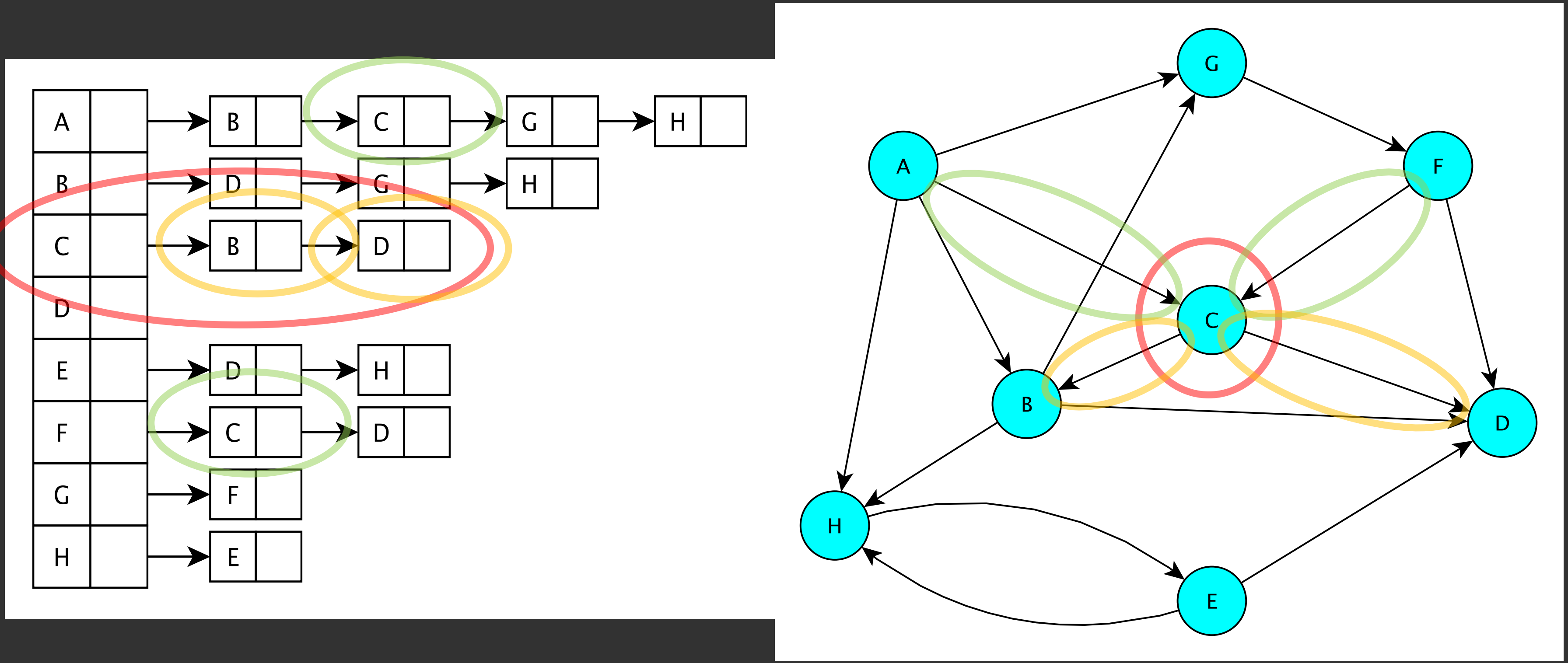
- Support both directed and undirected graphs
  - Abstract `GraphList` implements common functionality
  - Concrete classes `GraphListDirected` and `GraphListUndirected` complete implementation

GraphList

- Want to store edges adjacent to each vertex
- Let’s just store a List of them
  - specifically, a SinglyLinkedList

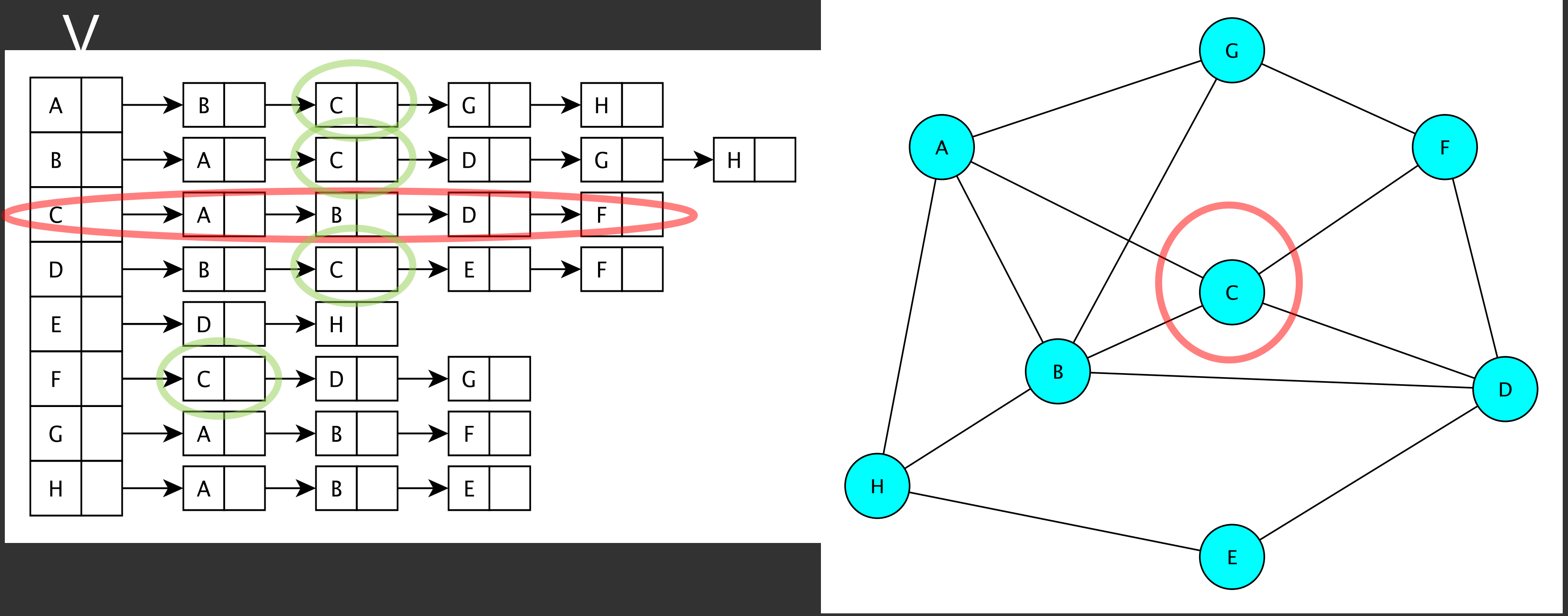
- If we do that for all vertices, then we’ve stored the whole graph!

Adjacency List : Directed Graph



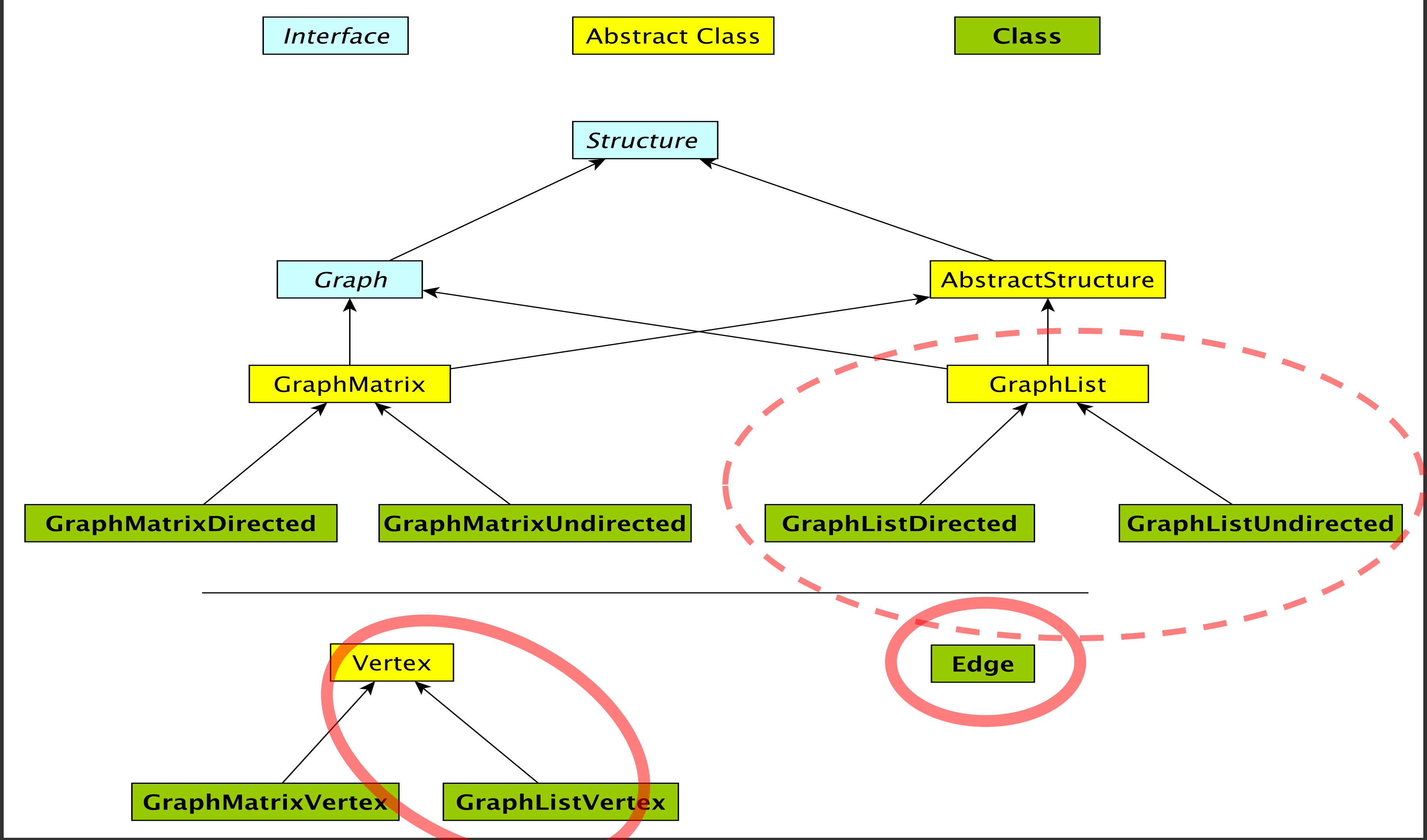
The vertices are stored in a Map<V, GraphListVertex>
Each GraphListVertex contains a linked list of all edges with a given source

Adjacency List : Undirected Graph



The vertices are stored in a Map<V, GraphListVertex>
Each GraphListVertex contains a linked list of all edges
incident to a given vertex

Graph Classes in structure5



Vertex and GraphListVertex

- We use the same `Edge` class for all graph types, but we will extend `Vertex` to include an `Edge` list

- `GraphListVertex` class adds to `Vertex` class:
  - A `Structure` to store edges adjacent to the vertex
  - Several methods

```
public void addEdge(Edge<V,E> e)
public boolean containsEdge(Edge<V,E> e)
public Edge<V,E> removeEdge(Edge<V,E> e)
public Edge<V,E> getEdge(Edge<V,E> e)
public int degree()
// and methods to produce Iterators...
```

GraphListVertex

```
class GraphListVertex<V,E> extends Vertex<V>
{
    protected Structure<Edge<V,E>> adjacencies;
```

GraphListVertex (extends Vertex)

```
public GraphListVertex(V label){
        super(label); // init superclass' fields (Vertex)
        adjacencies = new SinglyLinkedList<Edge<V,E>>();
}

public boolean containsEdge(Edge<V,E> e){
        return adjacencies.contains(e);
}

public void addEdge(Edge<V,E> e){
        if (!containsEdge(e)) { // no duplicate edges
            adjacencies.add(e);
        }
}

public Edge<V,E> removeEdge(Edge<V,E> e) {
        return adjacencies.remove(e);
}
```

GraphListVertex Iterators

```
// Iterator for incident edges
public Iterator<Edge<V,E>> adjacentEdges() {
        return adjacencies.iterator(); // use SLL’s iter
}

// Iterator for adjacent vertices
public Iterator<V> adjacentVertices() {
    ???
}
```

How can we iterate over the adjacent vertices?

Iterating over vertices

- Option 1: Store all the vertices in some list; return its iterator
  - Easy, but a bit fragile and space inefficient
- Option 2: Build class to iterate through edges one at a time using edge iterator; return vertex

We’ll do Option 2 this time

GraphListVertex Iterators

Note the A in GraphListAIterator

```
// Iterator for incident edges
public Iterator<Edge<V,E>> adjacentEdges() {
        return adjacencies.iterator(); // use SLL’s iter
}

// Iterator for adjacent vertices
public Iterator<V> adjacentVertices() {
        return new GraphListAIterator<V,E>
            (adjacentEdges(), label());
}
```

`GraphListAIterator` creates an Iterator over *vertices* based on the Iterator over *edges* produced by `adjacentEdges()`

GraphListAIterator: Dispenses Neighboing Vertices

GraphListAIterator is a class with two instance variables:

```
protected Iterator<Edge<V,E>> edges;
protected V vertex;

public GraphListAIterator(Iterator<Edge<V,E>> i, V v) {
        edges = i;
        vertex = v;
}

public V next() {
        Edge<V,E> e = edges.next();
        if (vertex.equals(e.here())) {
            return e.there();
        } else { // could be an undirected edge!
            return e.here();
        }
}
```

GraphList (Abstract base class)

- To implement `GraphList`, what data structures do we need?
  - (Recall: We maintain an *adjacency list* of *edges* at each vertex)
- `GraphListVertex` class
  - Instance vars: `label`, `visited` flag, linked list of edges

- We do NOT need a free list like `GraphMatrix`
- We do NOT need to know |V| ahead of time

GraphList

```
protected Map<V,GraphListVertex<V,E>> dict;
protected boolean directed;

protected GraphList(boolean dir){
    dict = new Hashtable<V,GraphListVertex<V,E>>();
    directed = dir;
}

public void add(V label) {
    if (dict.containsKey(label)) // unique vertices only
       return;

    GraphListVertex<V,E> v = new GraphListVertex<>(label);
    dict.put(label, v);
}
```

```
public Edge<V,E> getEdge(V labelA, V labelB) {
    // Create “dummy edge” for searching (ignore value)
    Edge<V,E> e = new Edge<>(get(labelA), get(labelB),
                  null, directed);
    return dict.get(labelA).getEdge(e);
}
```

(in GraphListVertex:)

```
public Edge<V,E> getEdge(Edge<V,E> e) {
      // Go through all V’s adjacent edges and compare
      Iterator<Edge<V,E>> edges = adjacencies.iterator();
      while (edges.hasNext()) {
            Edge<V,E> adjE = edges.next();
            if (e.equals(adjE))
                  return adjE;
      }
      return null;
}
```

GraphListDirected

- `GraphListDirected/Undirected` implement any methods requiring different treatment due to directedness of edges
  - `addEdge`, `remove`, `removeEdge`, …
  - edges appear in BOTH lists in undirected!
- We will only look at `GraphListDirected` in this video because the concepts are similar, and undirected version is slightly more straightforward

```java
// in GraphListDirected.java


// first vertex is source, second is destination
public void addEdge(V vLabel1, V vLabel2, E label) {
   // first get the vertices
   GraphListVertex<V,E> v1 = dict.get(vLabel1);
   GraphListVertex<V,E> v2 = dict.get(vLabel2);

   // create the new edge
   Edge<V,E> e = new Edge<V,E>(v1.label(), v2.label(), label, true);

   // add edge only to source vertex linked list (aka adjacency list)
   v1.addEdge(e);
}
```

```
// in GraphListDirected.java

public V remove(V label) {
   //Get vertex out of map/dictionary
   GraphListVertex<V,E> v = dict.get(label);

   //Iterate over all vertex labels
   Iterator<V> vi = iterator();
   while (vi.hasNext()) {
       //Get next vertex label in iterator
        V v2 = vi.next();

        //Remove all edges to "label"
        //If edge does not exist, removeEdge returns null
       removeEdge(v2,label);
    }

    //Remove vertex from map
    dict.remove(label);
    return v.label();
}
```

```java
// in GraphListDirected.java

public E removeEdge(V vLabel1, V vLabel2) {
   //Get vertices out of map
   GraphListVertex<V,E> v1 = dict.get(vLabel1);
   GraphListVertex<V,E> v2 = dict.get(vLabel2);

   //Create a “temporary” edge connecting two vertices
   Edge<V,E> e = new Edge<>(v1.label(), v2.label(), null, true);

   //Remove edge from source vertex linked list
   e = v1.removeEdge(e);

   if (e == null) {
    return null;
   } else {
    return e.label();
 }
}
```

GraphList: Big Picture

- Maintain an *adjacency list* of *edges* at each vertex (no adjacency matrix)
  - Keep only *outgoing* edges for directed graphs

- **Space:** we only “pay for what we store”
  - Vertex lists are as large as there are edges
- **Performance:** no “direct” way to access edges
  - We can quickly find a vertex, but need to scan through its unordered adjacency list

GraphList Efficiency

(assuming O(1) Map operations)

For a `GraphListDirected<V,E>`

- where |E| = number of edges, and |V| = number of vertices

Operation	**Big-O**
`add(V label)`	O(1)
`remove(V label)`	O(|V| + |E|)
`addEdge(V v1, V v2)`	O(min(|V|, |E|))
`getEdge(V v1, V v2)`	O(min(|V|, |E|))
`removeEdge(V v1, V v2)`	O(min(|V|, |E|)))
Space Usage	O(|V| + |E|)

O(min(|V|, |E|))?

- Why is removeEdge O(min(|V|, |E|))?
  - We don’t know which is smaller
    - For lots of graphs, |V| is smaller
    - But not always! Some graphs have few edges

  - To remove an edge, we must traverse all edges in a vertex’s adjacency list.
    - There cannot be more than |V| edges in the adjacency list
    - There cannot be more than |E| edges in the adjacency list.