CSCI 136 Data Structures & Advanced Programming

Lecture 28

Fall 2019

Instructors:

$$\begin{tikzcd} \text{Bill} \arrow[r, bend left] & \text{Sam} \arrow[l, bend left] \end{tikzcd}$$

Last Time

- More on Graphs
  - Applications and Problems
    - Testing connectedness
    - Counting connected components
    - Breadth-first search
    - Depth-first search
      - And recursive depth-first search

Today's Outline

- Directed Graphs (from Lecture 27)
  - Definition and Properties
  - Reachability and (Strong) Connectedness
- Graph Data Structures: Implementation
  - Graph Interface (from Lecture 27)
  - Adjacency Array Implementation Basic Concepts
  - Adjacency List Implementation Basic Concepts
  - Adjacency Array Implementation Details

Implementing Breadth-First Search

```
BFS(G, v)          // Do a breadth-first search of G starting at v
// pre: all vertices are marked as unvisited
// post: return number of visited vertices
count ← 0;
Create empty queue Q; enqueue v; mark v as visited; count++
While Q isn't empty
        current ← Q.dequeue();
        for each unvisited neighbor u  of current :
                add u to Q; mark u as visited; count++
return count;
```

Breadth-First Search

```
int BFS(Graph<V,E> g, V src) {
  Queue<V> todo = new QueueList<V>(); int count = 0;
  g.visit(src); count++;
  todo.enqueue(src);
  while (!todo.isEmpty()) {
    V node = todo.dequeue();
    Iterator<V> neighbors = g.neighbors(node);
    while (neighbors.hasNext()) {
       V next = neighbors.next();
       if (!g.isVisited(next)) {
          g.visit(next); count++;
          todo.enqueue(next);
       }
    }
  }
  return count;
}
```

Breadth-First Search of Edges

```
int BFS(Graph<V,E> g, V src) {
  Queue<V> todo = new QueueList<V>(); int count = 0;
  g.visit(src); count++;
  todo.enqueue(src);
  while (!todo.isEmpty()) {
    V node = todo.dequeue();
    Iterator<V> neighbors = g.neighbors(node);
    while (neighbors.hasNext()) {
       V next = neighbors.next();
       if (!g.isVisitedEdge(node,next)) g.visitEdge(next,node);
       if (!g.isVisited(next)) {
          g.visit(next); count++;
          todo.enqueue(next);
       }
    }
  }
  return count;
}
```

Recursive Depth-First Traversal

```
// Before first call to DFS, set all vertices to unvisited
//Then call DFS(G,v)
DFS(G, v)
        Mark v as visited; count=1;
        for each unvisited neighbor u of v:
                count += DFS(G,u);
        return count;
```

Recursive Depth-First Traversal

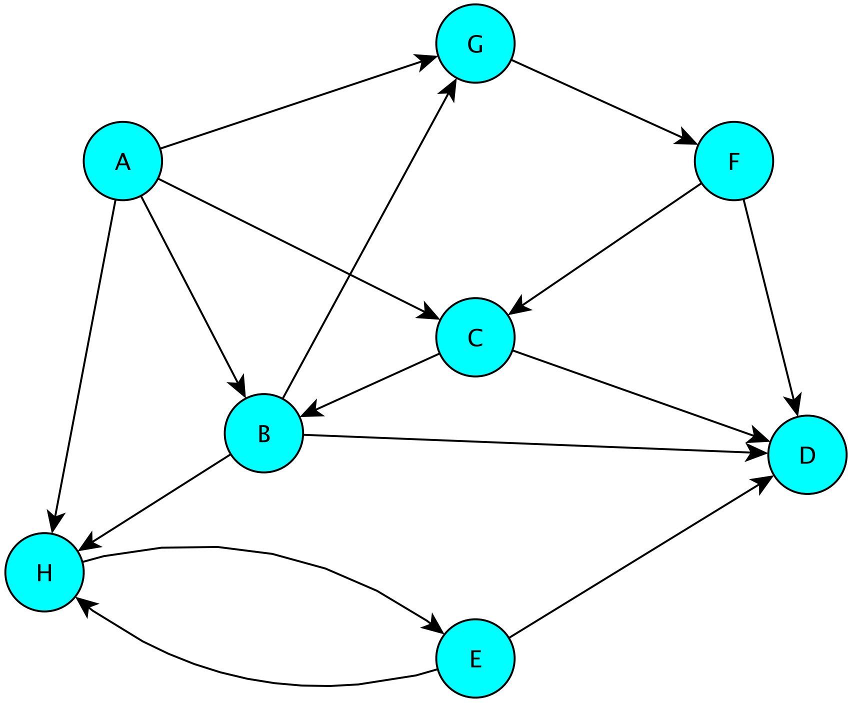
```
int DFS( Graph<V,E> g, V src ) {
    g.visit(src);
    int count = 1;
    Iterator<V> neighbors = g.neighbors(src);
    while (neighbors.hasNext()) {
       V next = neighbors.next();
       if (!g.isVisited(next))
              count += DFS(g, next);
    }
  }
  return count;
}
```

Representing Graphs

- Two standard approaches
  - Option 1: Array-based (directed and undirected)
  - Option 2: List-based (directed and undirected)
- We'll look at both
  - Array-based graphs store the edge information in a 2-dimensional array indexed by the vertices
  - List-based graphs store the edge information in a (1-dimensional) array of lists
    - The array is indexed by the vertices
    - Each array element is a list of edges incident with that vertex

Adjacency Array: Directed Graph

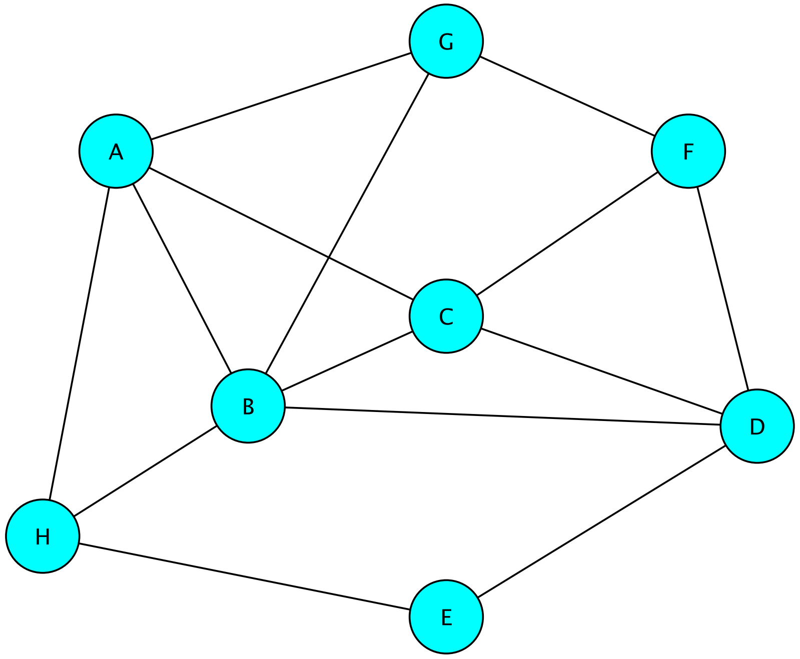
	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	1	1
B	0	0	0	1	0	0	1	1
C	0	1	0	1	0	0	0	0
D	0	0	0	0	0	0	0	0
E	0	0	0	1	0	0	0	1
F	0	0	1	1	0	0	0	0
G	0	0	0	0	0	1	0	0
H	0	0	0	0	1	0	0	0



Entry (i,j) stores 1 if there is an edge from i to j; 0 otherwise
E.G.: edges(B,C) = 1 but edges(C,B) = 0

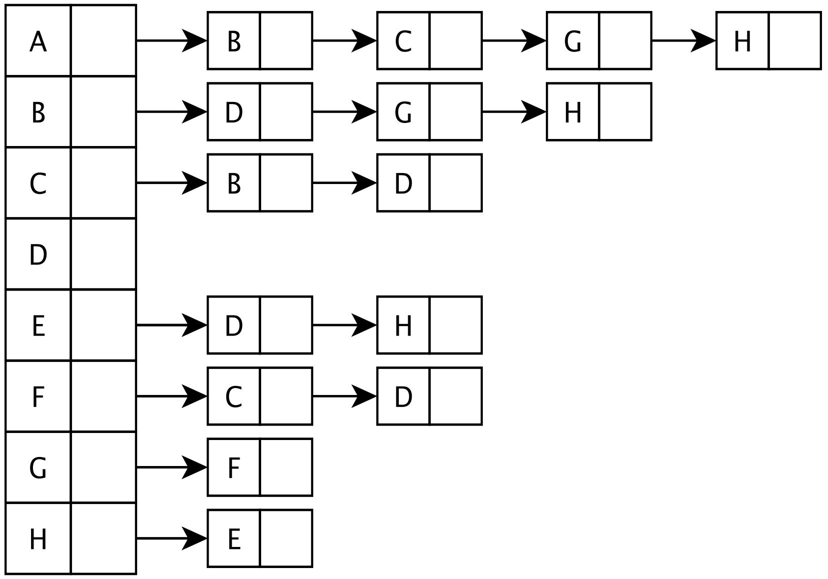
Adjacency Array: Undirected Graph

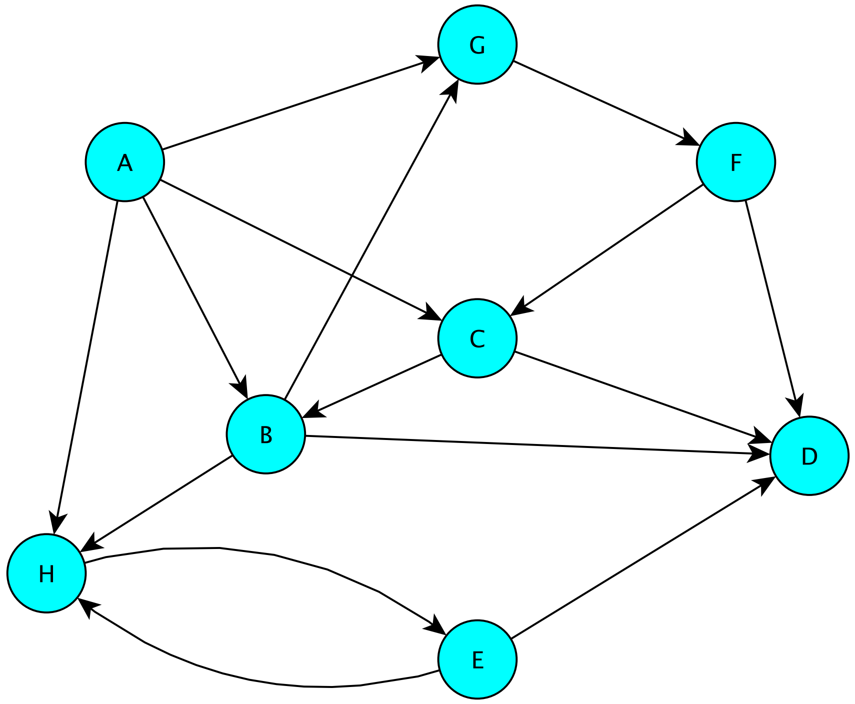
	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	1	1
B	1	0	1	1	0	0	1	1
C	1	1	0	1	0	1	0	0
D	0	1	1	0	1	1	0	0
E	0	0	0	1	0	0	0	1
F	0	0	1	1	0	0	1	0
G	1	1	0	0	0	1	0	0
H	1	1	0	0	1	0	0	0



Entry (i,j) store 1 if there is an edge between i and j; else 0
E.G.: edges(B,C) = 1 = edges(C,B)

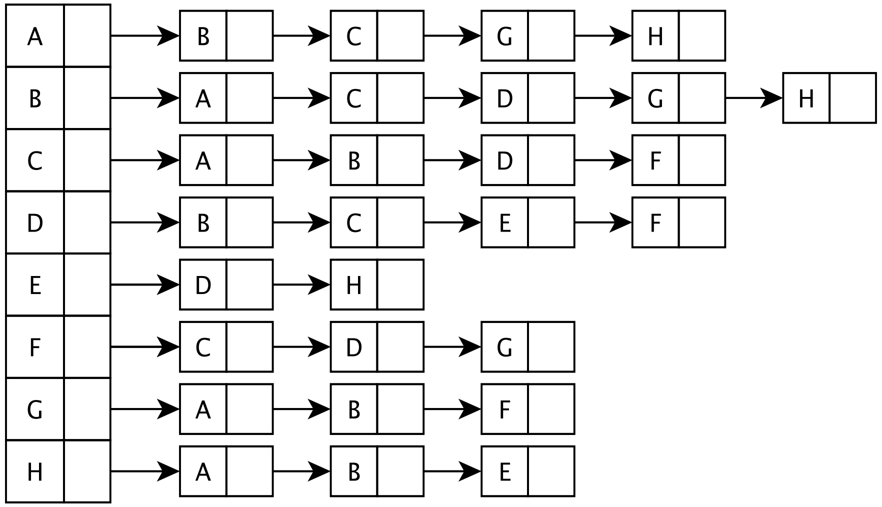
Adjacency List : Directed Graph

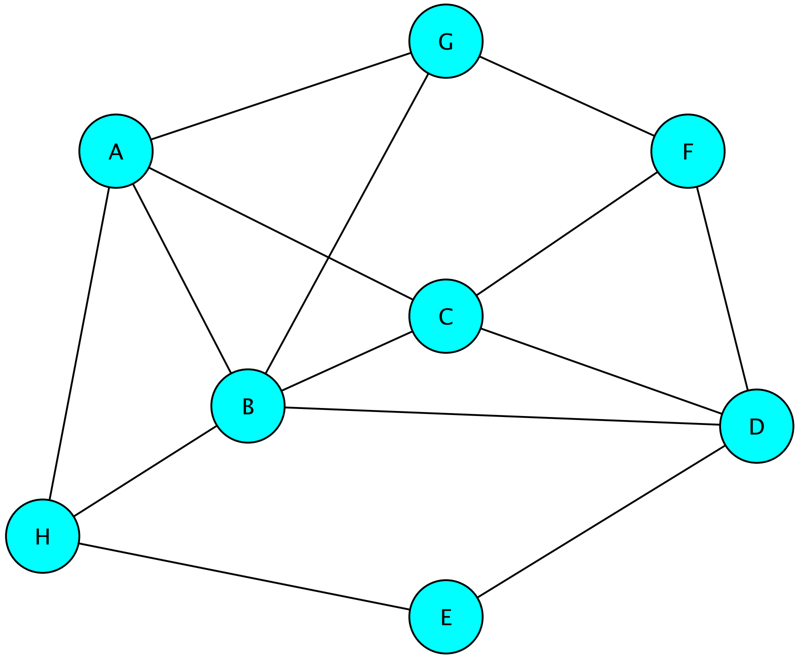




The vertices are stored in an array V[]
V[] contains a linked list of edges having a given source

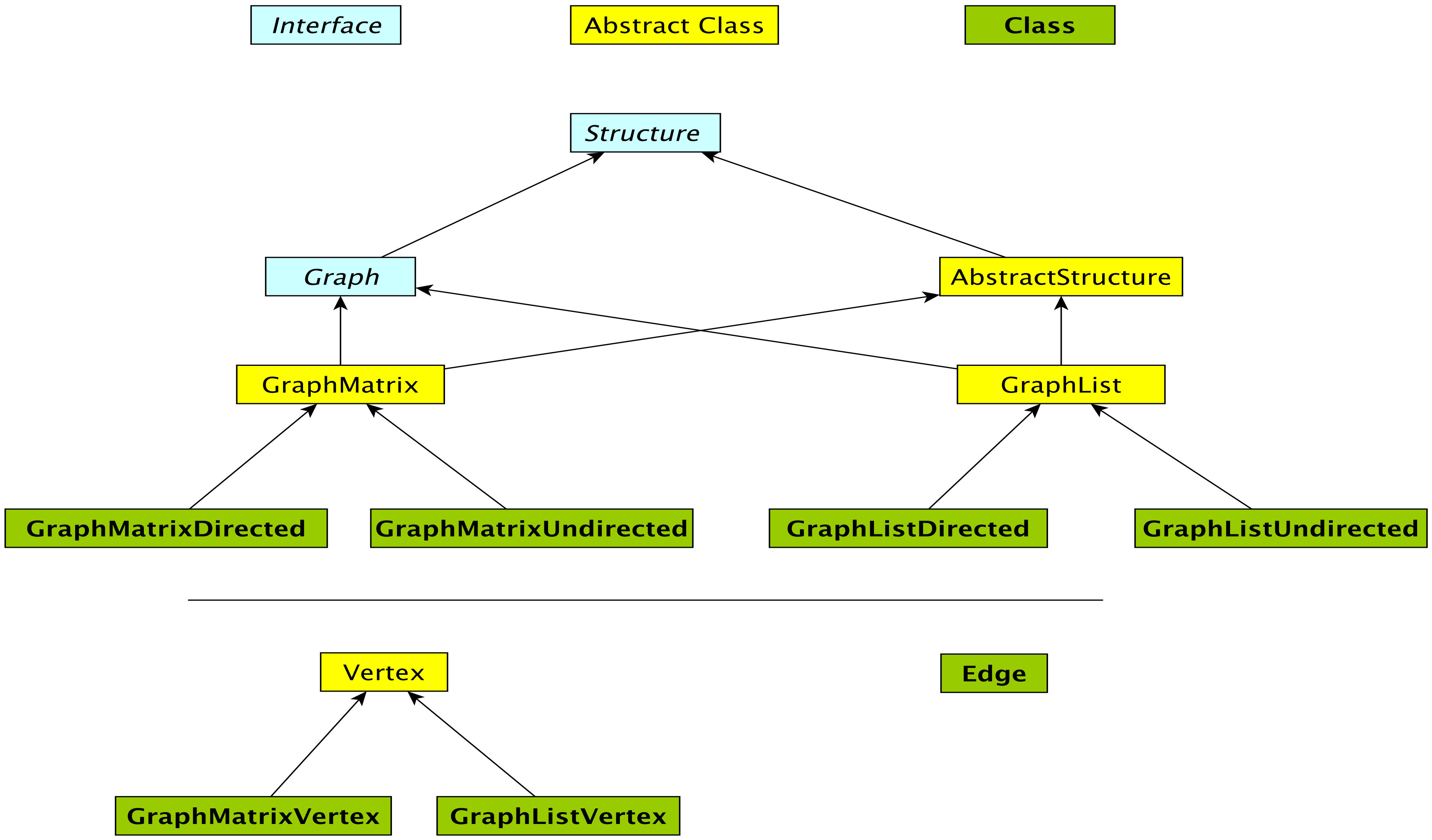
Adjacency List : Undirected Graph





The vertices are stored in an array V[]
V[] contains a linked list of edges incident to a given vertex

Graph Classes in structure5



Graph Classes in structure5

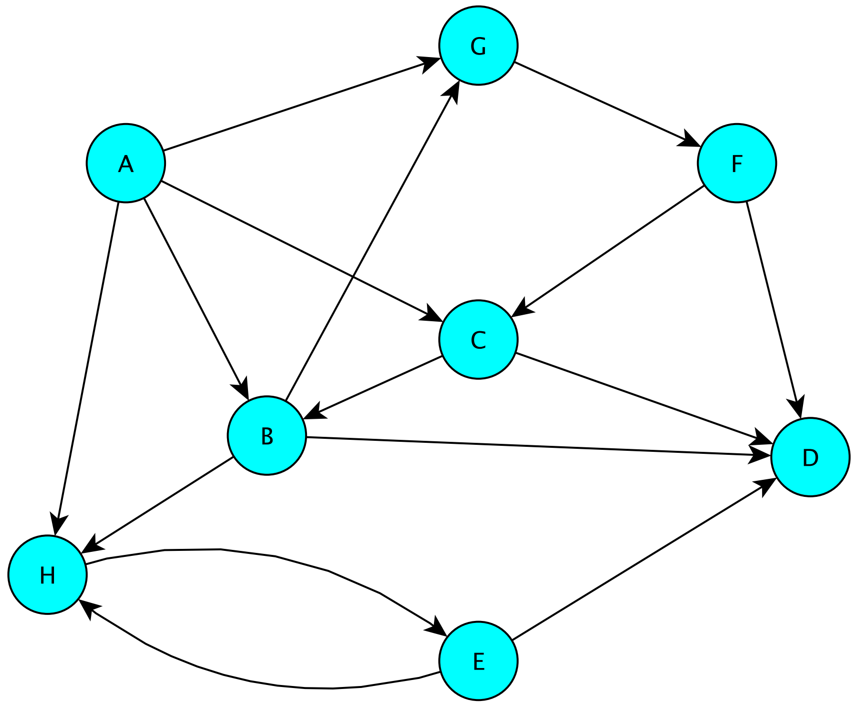
Why so many?!

- There are two types of graphs: undirected & directed
- There are two implementations: arrays and lists
- We want to be able to avoid large amounts of identical code in multiple classes
- We abstract out features of implementation common to both directed and undirected graphs

We'll tackle array-based graphs first....

Adjacency Array: Directed Graph

	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	1	1
B	0	0	0	1	0	0	1	1
C	0	1	0	1	0	0	0	0
D	0	0	0	0	0	0	0	0
E	0	0	0	1	0	0	0	1
F	0	0	1	1	0	0	0	0
G	0	0	0	0	0	1	0	0
H	0	0	0	0	1	0	0	0



Challenges
•Can’t use Objects as array indices
•How does deleting a vertex work?!

Vertex and GraphMatrixVertex

- We need to define a Vertex class
  - Unlike the Edge class, Vertex class **is not public**
  - Useful Vertex methods:

    ```
    V label(), boolean visit(),
    boolean isVisited(), void reset()
    ```
  - GraphMatrixVertex class adds one more useful attribute to Vertex class
    - Index of node (int) in adjacency matrix

      ```
      int index()
      ```
    - Why do we only need one int to represent index?

Choosing a Dictionary Structure

- We need a structure that will let us retrieve the index of a vertex given the vertex label (a dictionary)
- Many choices
  - Vector of associations:
    - Vector<Association<V, GraphMatrixVertex<V>>>
  - Ordered Vector of Associations
  - BinarySearchTree of Associations
- Problem: We don’t want to allow multiple vertices with same label.... [Why?]
- We’ll use the Map Interface [Chapter 15]
  - Maps require a unique key for each entry

Digression : Map Interface

- Methods for Map<K, VAL>
  - int size() - returns number of entries in map
  - boolean isEmpty() - true iff there are no entries
  - boolean containsKey(K key) - true iff key exists in map
  - boolean containsValue(VAL val) - true iff val exists at least once in map
  - VAL get(K key) - get value associated with key
  - VAL put(K key, VAL val) - insert mapping from key to val, returns value replaced (old value) or null
  - VAL remove(K key) - remove mapping from key to val
  - void clear() - remove all entries from map
- We'll study this more in a week or so....