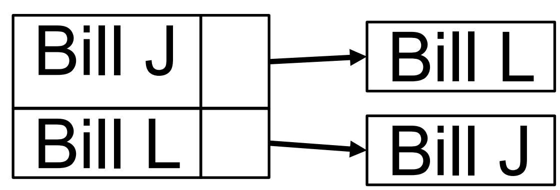
CSCI 136
Data Structures &
Advanced Programming

Lecture 29

Fall 2018

Instructors:



Last Time

- Recursive Depth-First Search
  - Tips on writing recursive methods
- Graph Data Structures: Implementation
  - Graph Interface

Today’s Outline

- Graph Data Structures: Implementation
  - Adjacency Array Implementation
  - Adjacency List Implementation
    - Featuring many Iterators!

Recall: Desired Functionality

- What are the basic operations we need to describe algorithms on graphs?
  - Given vertices u and v: are they adjacent?
  - Given vertex v and edge e, are they incident?
  - Given an edge e, get its incident vertices (*ends*)
  - How many vertices are adjacent to v? (*degree* of v)
    - The vertices adjacent to v are called its *neighbors*
  - Get a list of the neighbors of v (or the edges incident with v)

Graphs in structure5

- We want to store information at vertices and at edges, but we favor vertices
  - Let V and E represent the types of information held by vertices and edges respectively
  - Interface Graph<V,E> extends Structure<V>
    - Vertices are the building blocks; edges depend on them
- Type V holds a *label* for a (hidden) vertex type
- Type E holds a *label* for an (available) edge type
  - Label: Application-specific data for a vertex/edge

Graphs in structure5

- The methods described in the Structure interface deal with *vertices*
  - but also impact edges: e.g., clear()
- We'll want to add a number of similar methods to provide information about edges, and the graph itself

Graph Interface Methods

- void add(V vLabel)
- V remove(V vLabel)
  - Add/remove vertex to graph
- void addEdge(V vLabel1, V vLabel2, E edgeLabel),
  E removeEdge(V vLabel1, V vLabel2)
  - Add/remove edge between vLabel1 and vLabel2
- boolean containsEdge(V vLabel1, V vLabel2)
  - Returns true iff there is an edge between vLabel1 and vLabel2
- Edge<V,E> getEdge(V vLabel1, V vLabel2)
  - Returns edge between vLabel1 and vLabel2
- void clear()
  - Remove all nodes (and edges) from graph

Graph Interface Methods

- boolean visit(V vLabel)
  - Mark vertex as “visited” and return *previous* value of visited flag
- boolean visitEdge(Edge<V,E> e)
  - Mark edge as “visited”
- boolean isVisited(V vLabel), boolean isVisitedEdge(Edge<V,E> e)
  - Returns true iff vertex/edge has been visited
- Iterator<V> neighbors(V vLabel)
  - Get iterator for all neighbors of vLabel
  - For directed graphs, out-edges only
- Iterator<V> iterator()
  - Get vertex iterator
- void reset()
  - Remove visited flags for all nodes/edges

Edge Class

- Graph *edges* are defined in their own public class
  -
    ```
    Edge<V,E>(    V vLabel1, V vLabel2,
              E label, boolean directed)
    ```
  - Construct a (possibly directed) edge between two labeled vertices (`vLabel1` → `vLabel2`)
  - vLabel1 : here; vLabel2 : there
- Useful methods:

```
label(), here(), there()
setLabel(), isVisited(), isDirected()
```

Recursive Depth-First Search

```
// 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 Search

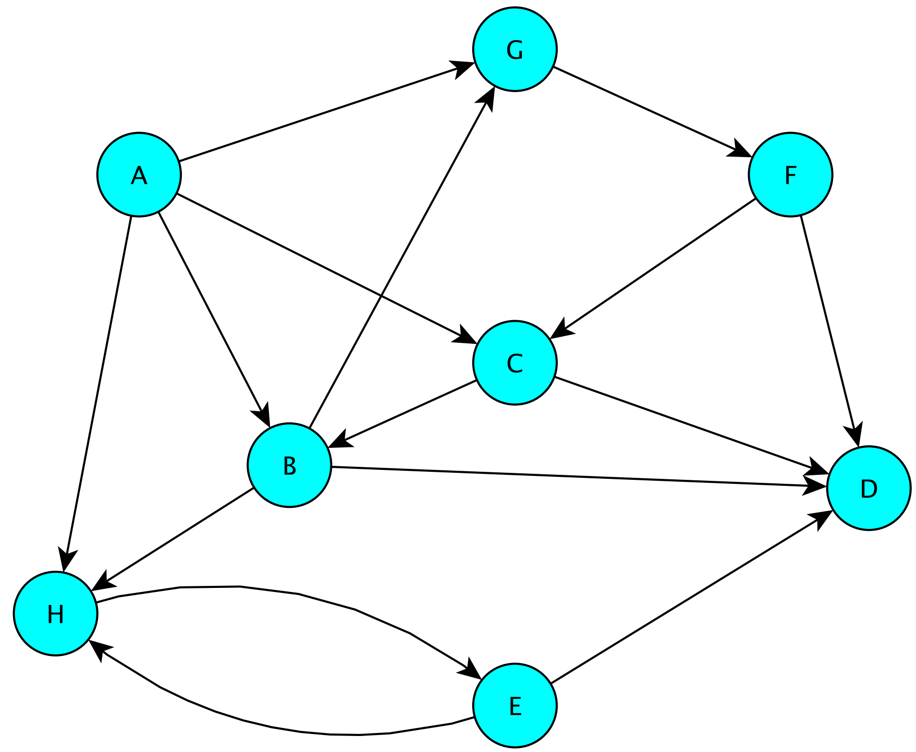
```
int DFS(Graph<V,E> g, V src) {
    g.visit(src);
    int count = 1;
    Iterator<V> neighbors = g.neighbors(src);
    while (neighbors.hasNext()) {
       V next = neighbors.next();
       if (!g.isVisited(next))
              count += DFS(g, next);
    }
  }
  return count;
}
```

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

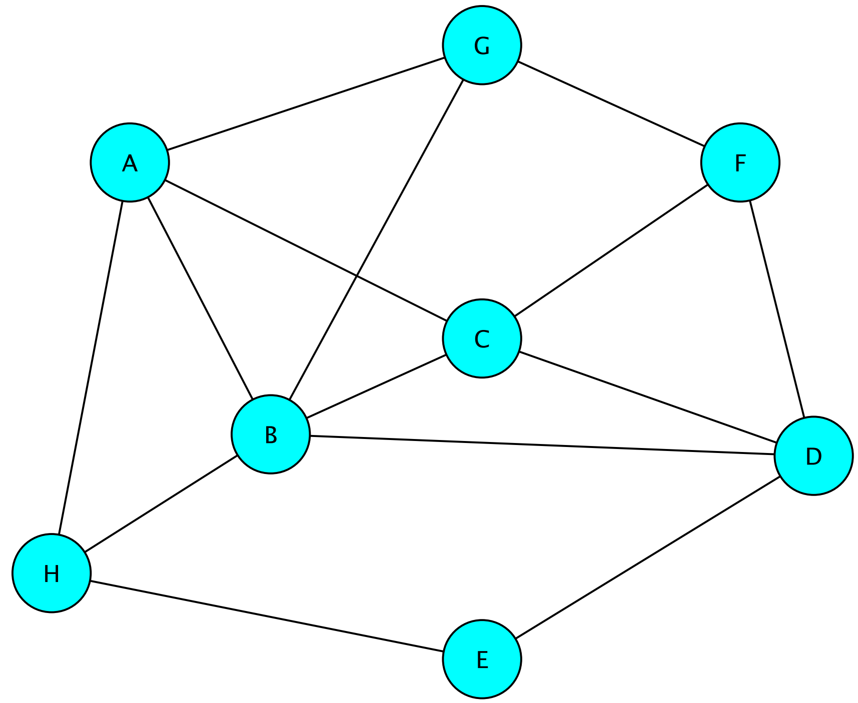
	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	1	1
B	0	0	0	1	0	0	1	1
C	0	1	0	1	0	0	0	0
D	0	0	0	0	0	0	0	0
E	0	0	0	1	0	0	0	1
F	0	0	1	1	0	0	0	0
G	0	0	0	0	0	1	0	0
H	0	0	0	0	1	0	0	0



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

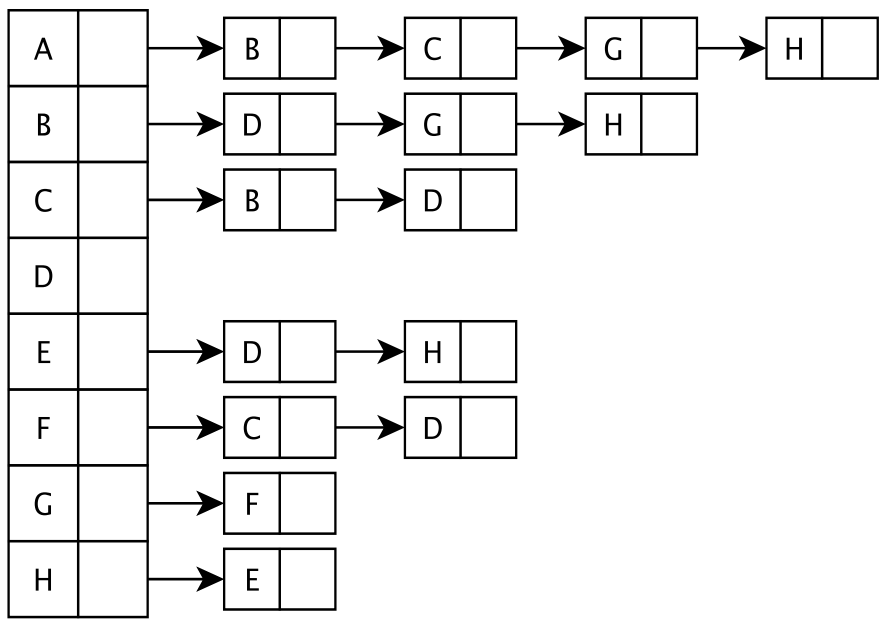
Adjacency Array: Undirected Graph

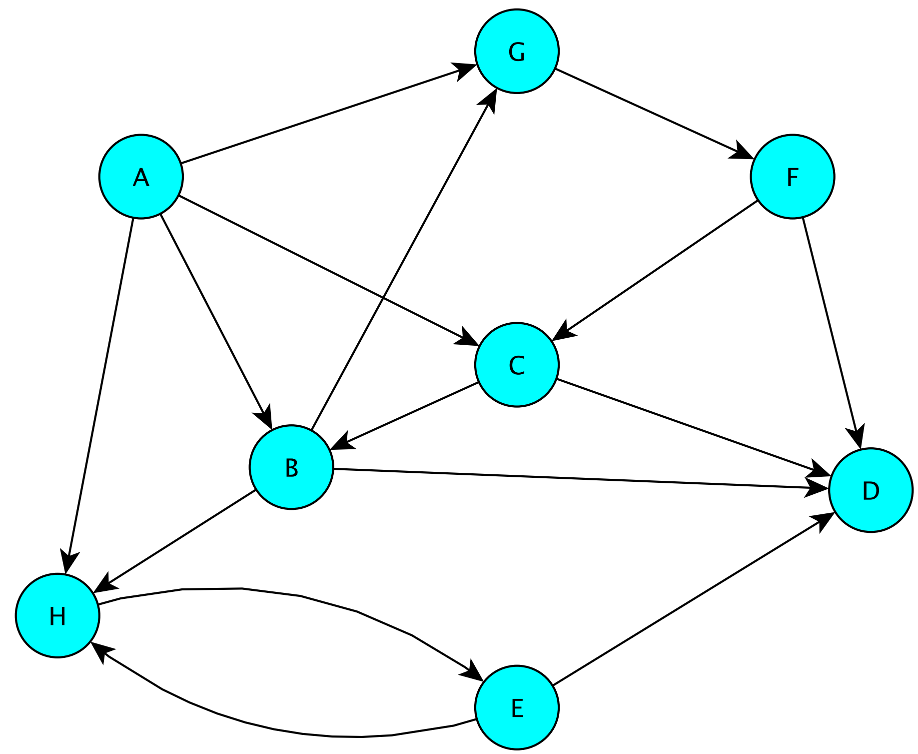
	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	1	1
B	1	0	1	1	0	0	1	1
C	1	1	0	1	0	1	0	0
D	0	1	1	0	1	1	0	0
E	0	0	0	1	0	0	0	1
F	0	0	1	1	0	0	1	0
G	1	1	0	0	0	1	0	0
H	1	1	0	0	1	0	0	0



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)

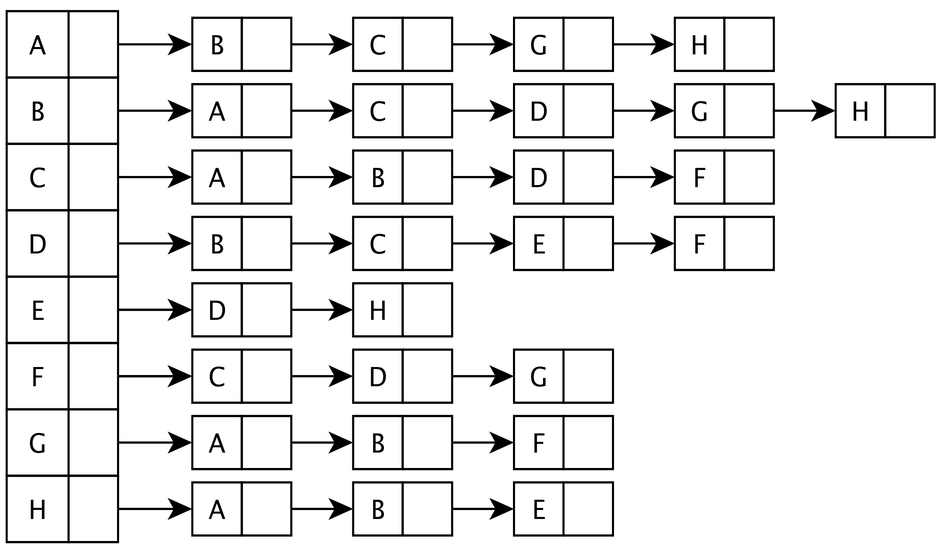
Adjacency List : Directed Graph

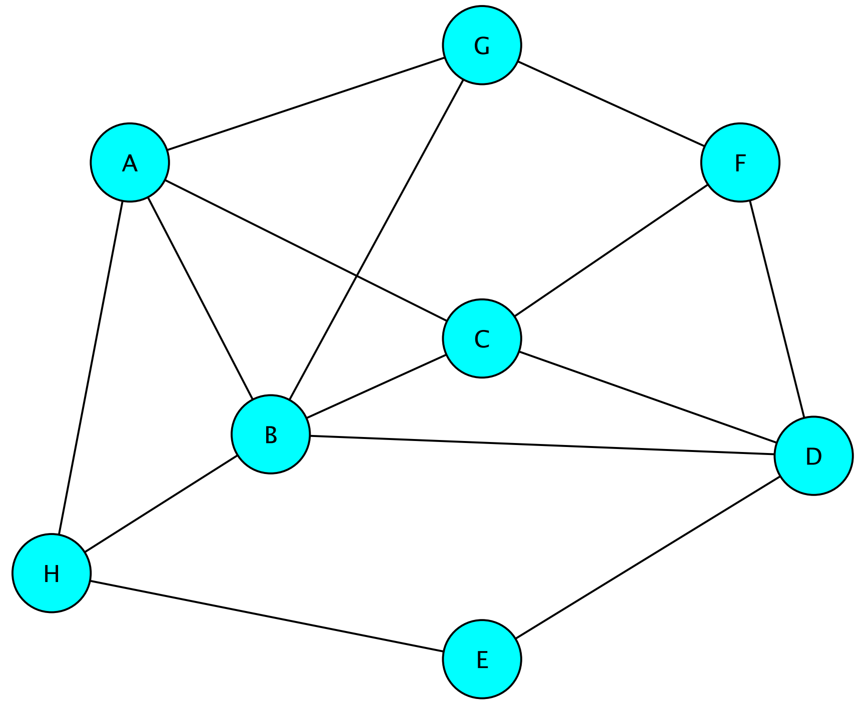




The vertices are stored in an array V[]
V[] contains a linked list of edges having a given source

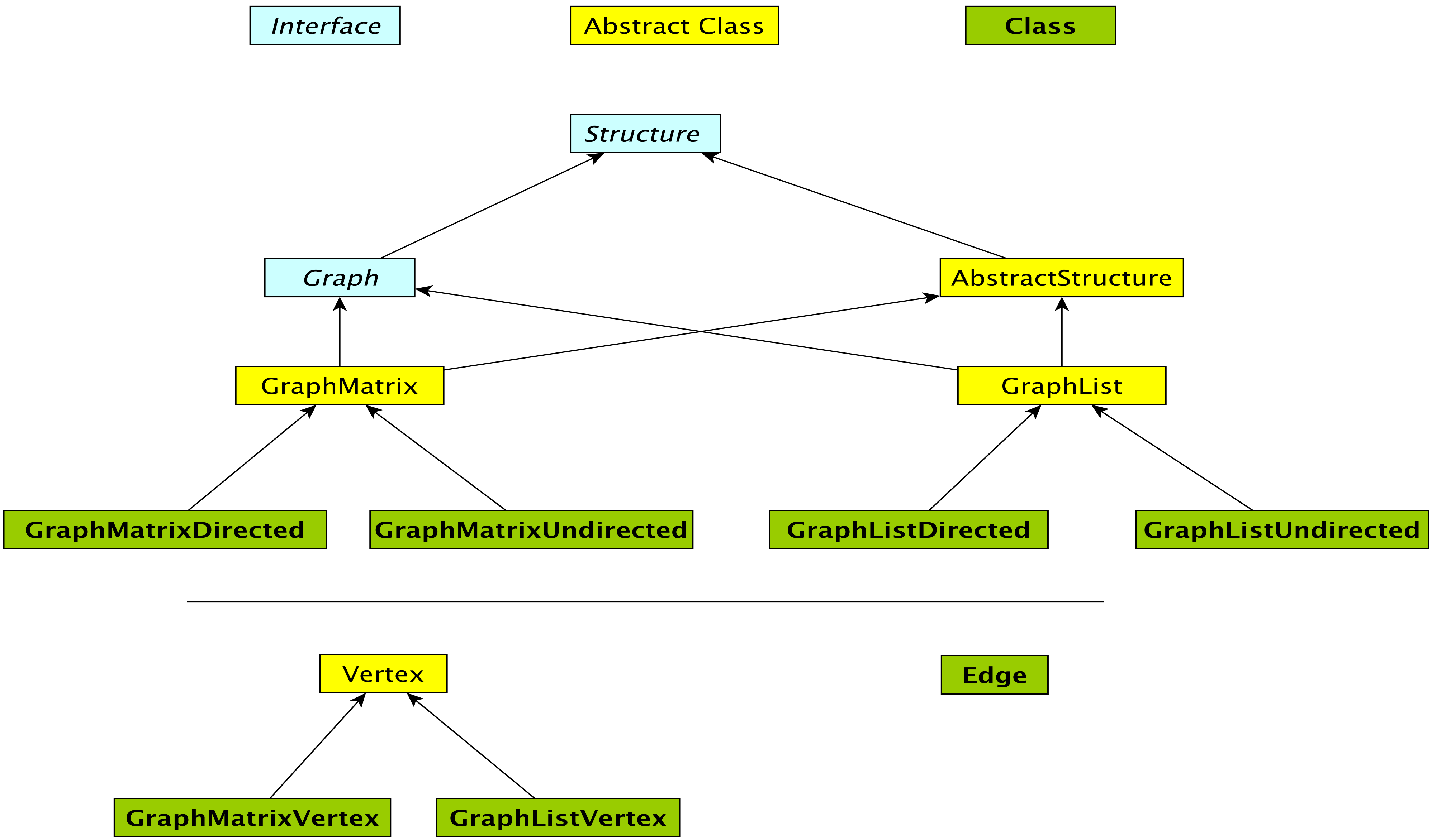
Adjacency List : Undirected Graph





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

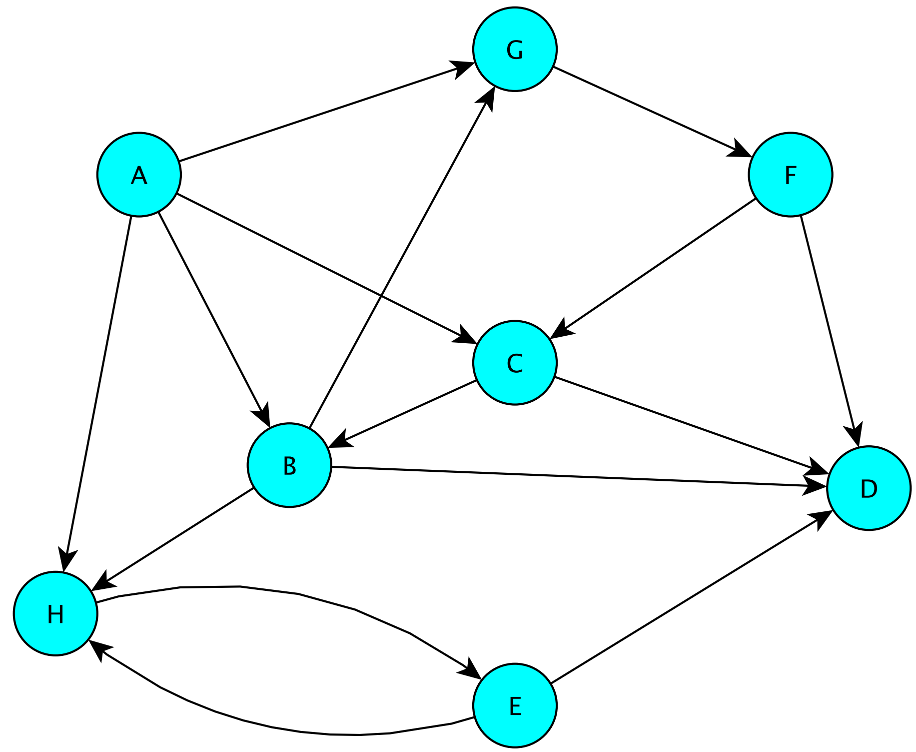
Why so many?!

- 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

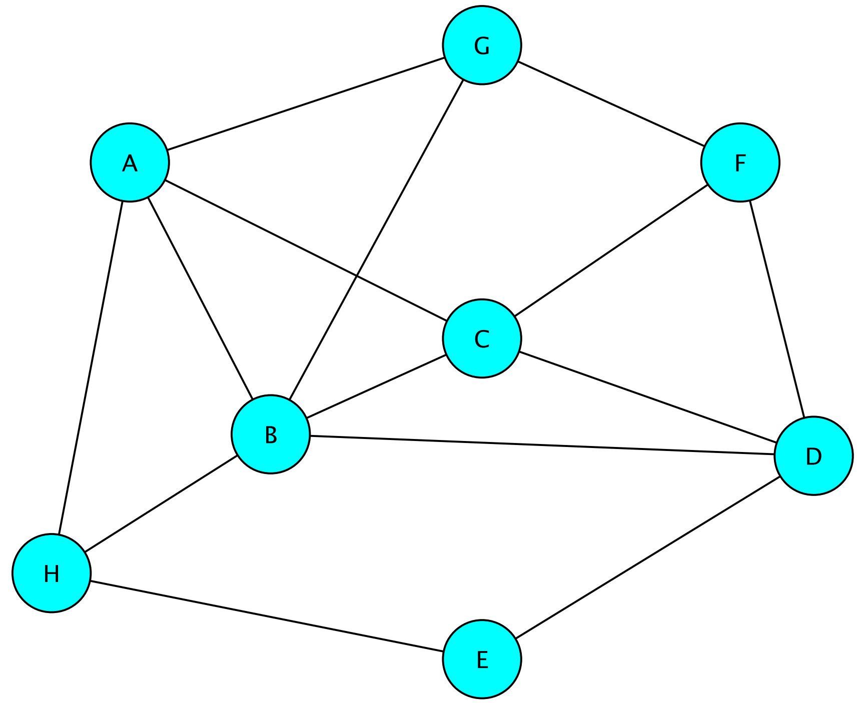
	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	1	1
B	0	0	0	1	0	0	1	1
C	0	1	0	1	0	0	0	0
D	0	0	0	0	0	0	0	0
E	0	0	0	1	0	0	0	1
F	0	0	1	1	0	0	0	0
G	0	0	0	0	0	1	0	0
H	0	0	0	0	1	0	0	0



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 Array: Undirected Graph

Halving the Space (not in structure5)

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	1
1	1	0	1	1	0	0	1
2	1	1	0	1	0	1	0
3	0	1	1	0	1	1	0
4	0	0	0	1	0	0	0
5	0	0	1	1	0	0	1
6	1	1	0	0	0	1	0

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	1
1		0	1	1	0	0	1
2			0	1	0	1	0
3				0	1	1	0
4					0	0	0
5						0	1
6							0

0	1	2	3	4	5	6	7	8	9	…																	
0	1	1	0	0	0	1	0	1	1	0	0	1	0	1	0	1	0	0	1	1	0	0	0	0	0	1	0

(i,j) maps to $7i + j - \frac{i(i+1)}{2}$

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?
- In these slides, we write GMV for GraphMatrixVertex

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....

Implementing the Matrix Model

- Abstract class – partially implements Graph

```
public abstract class GraphMatrix<V,E> implements Graph<V,E>
```

- This class will implement features common to directed and undirected graphs
- Instance variables

```
protected int size;  //max size of matrix
protected Object data[][];  //matrix of edges
protected Map<V, GMV<V>> dict; //labels -> vertices
// This is structure5.Map, NOT java.util.Map!
protected List<Integer> freeList; //avail indices
protected boolean directed;
```

GraphMatrix Constructor

(Yes, abstract classes can have constructors!)

```
protected GraphMatrix(int size, boolean dir) {
    this.size = size; // set maximum size
    directed = dir; // fix direction of edges

    // the following constructs a size x size matrix
    // (the “Objects” will be “Edges”)
    // (can’t use generics with arrays!)
    data = new Object[size][size];

    // label→index translation table
    dict = new Hashtable<V,GraphMatrixVertex<V>>(size);

    // put all indices in the free list
    freeList = new SinglyLinkedList<Integer>();
    for (int row = size-1; row >= 0; row--)
        freeList.add(new Integer(row));
}
```

GraphMatrix add()

```
public void add(V label) {
    // if there already, do nothing
    if (dict.containsKey(label)) return;

    Assert.pre(!freeList.isEmpty(), "Matrix not full");
    // allocate a free row and column
    int row = freeList.removeFirst().intValue();
    // add vertex to dictionary
    dict.put(label, new GraphMatrixVertex<V>(label, row));
}
```

GraphMatrix remove()

```
public V remove(V label) {
       // find and extract vertex
       GraphMatrixVertex<V> vert;
       vert = dict.remove(label);
       if (vert == null) return null;
       // remove vertex from matrix
       int index = vert.index();
       // clear row and column entries
       for (int row=0; row<size; row++) {
           data[row][index] = null;
           data[index][row] = null;
       }
       // add node index to free list
       freeList.add(new Integer(index));
       return vert.label();
}
```

Neighbors Iterator : GraphMatrix

neighbors Iterator

```
public Iterator<V> neighbors(V label) {
      GraphMatrixVertex<V> vert = dict.get(label);
      List<V> list = new SinglyLinkedList<V>();
      for (int row=size-1; row>=0; row--) {
            Edge<V,E> e = (Edge<V,E>)data[vert.index()][row];
            if (e != null)
                  if (e.here().equals(vert.label()))
                        list.add(e.there());
                        else list.add(e.here());
      }
      return list.iterator();
   }
```