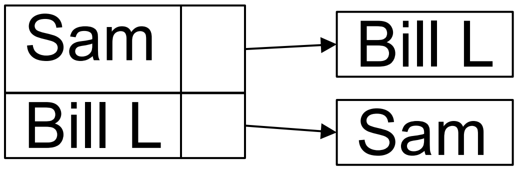
CSCI 136
Data Structures & Advanced Programming

Lecture 29

Fall 2019

Instructors:



Admin

- Lab 10 (last lab!) out
- Fill out form by Monday at midnight

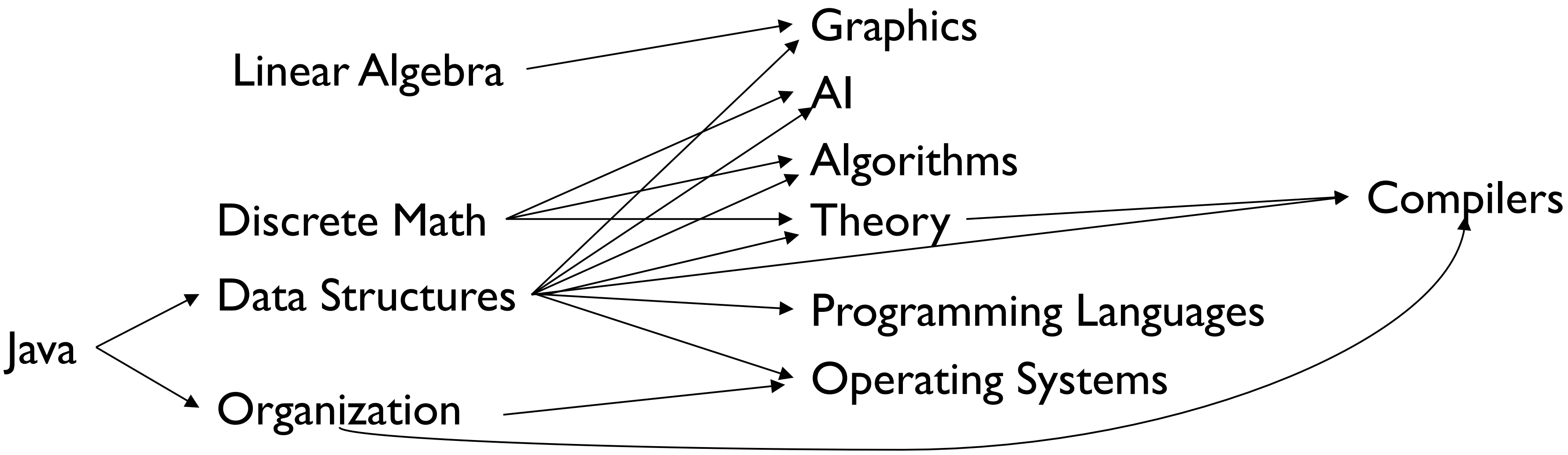
Last Time

- BFS and DFS
- Intro to directed graphs

Today's Outline

- Directed graphs
- Graph Data Structures
  - Graph Interface
  - How do we actually store a graph?

Directed Graphs

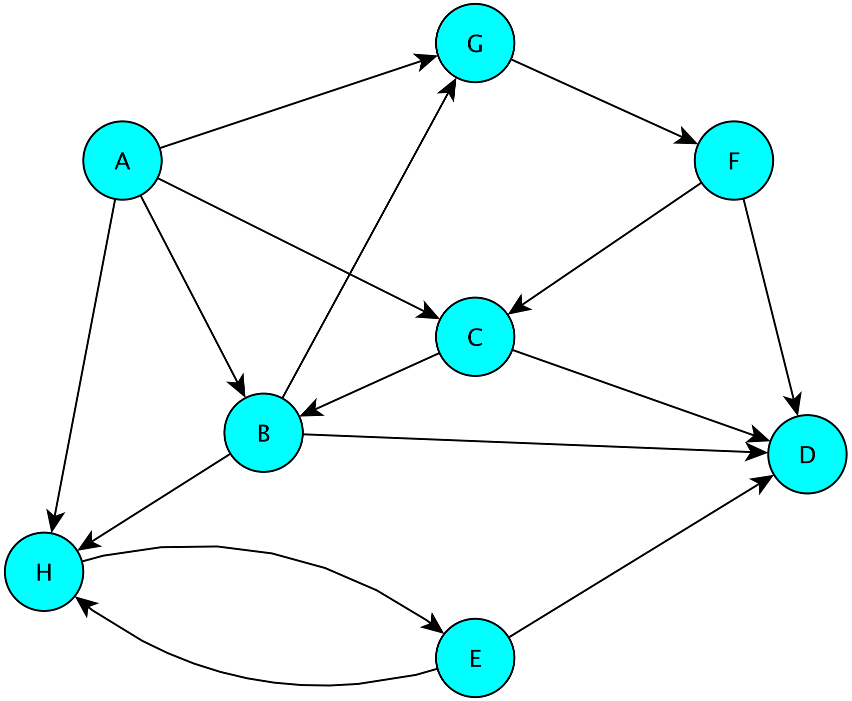


Def'n: In a *directed graph G = (V,E)*, each edge e in E is an *ordered* pair: e = (u,v) vertices: its *incident vertices*. The *source* of e is u; the *destination/target* is v.

Note: (u,v) ≠ (v,u)

Directed Graphs

- The (out) neighbors of B are D, G, H: B has out-degree 3
- The in neighbors of B are A, C: B has in-degree 2
- A has in-degree 0: it is a *source* in G; D has out-degree 0: it is a *sink* in G



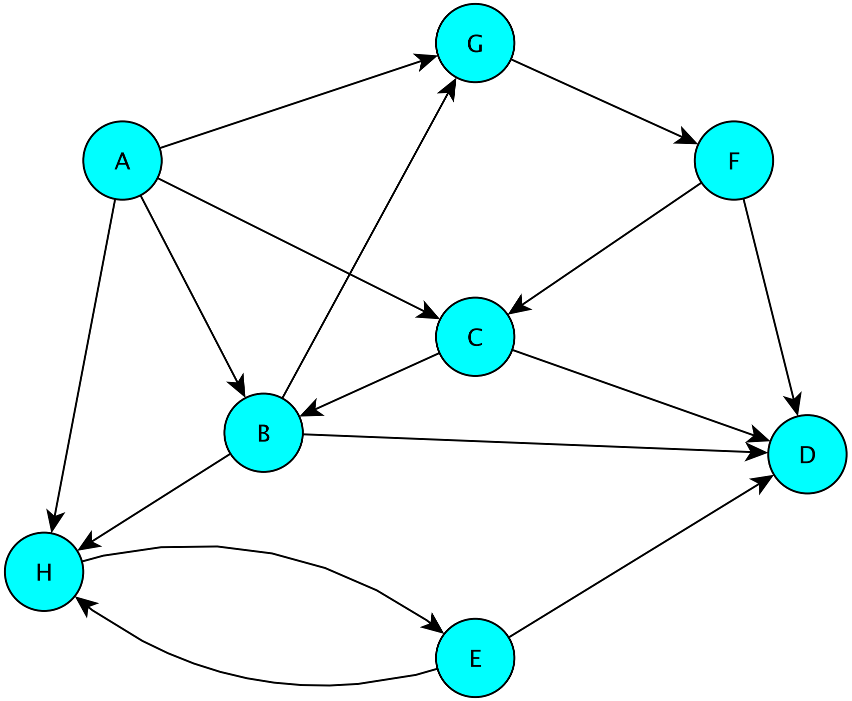
A walk is still an alternating sequence of vertices and edges

$$u = v_0, e_1, v_1, e_2, v_2, \ldots, v_{k-1}, e_k, v_k = v$$

but now $e_i = (v_{i-1}, v_i)$: all edges *point along direction* of walk

Directed Graphs

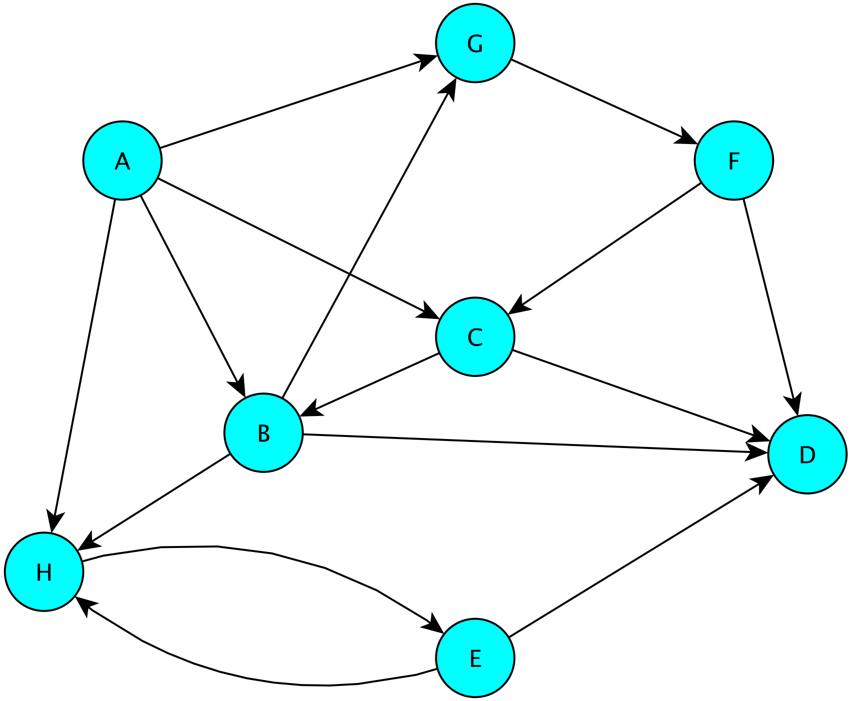
- A, B, H, E, D is a walk from A to D
- It's also a (simple) path
- D, E, H, B, A is *not* a walk from D to A
- B, G, F, C, B is a (directed) cycle (it's a 4-cycle)
- So is H, E, H (a 2-cycle)



- D is reachable from A (via path A, B, D), but A is not reachable from D
- In fact, every vertex is reachable from A

Directed Graphs

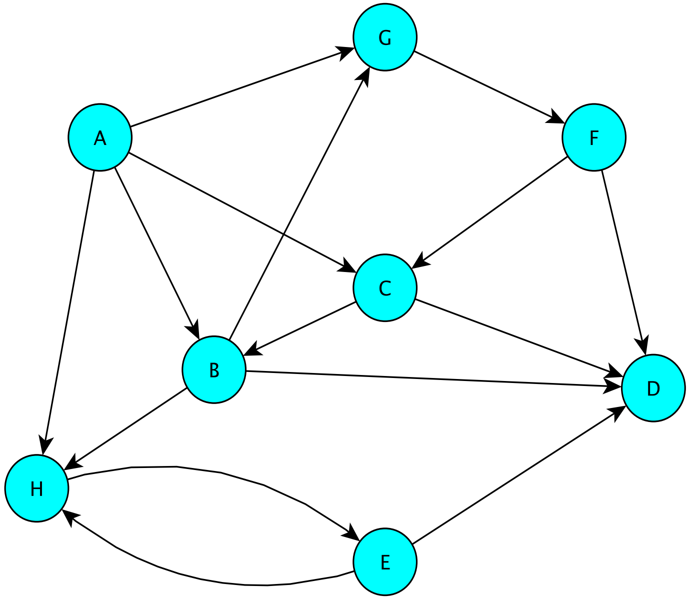
- A BFS of *G* from A visits every vertex
- A BFS of *G* from F visits all vertices but A
- A BFS of *G* from E visits only E, H, D

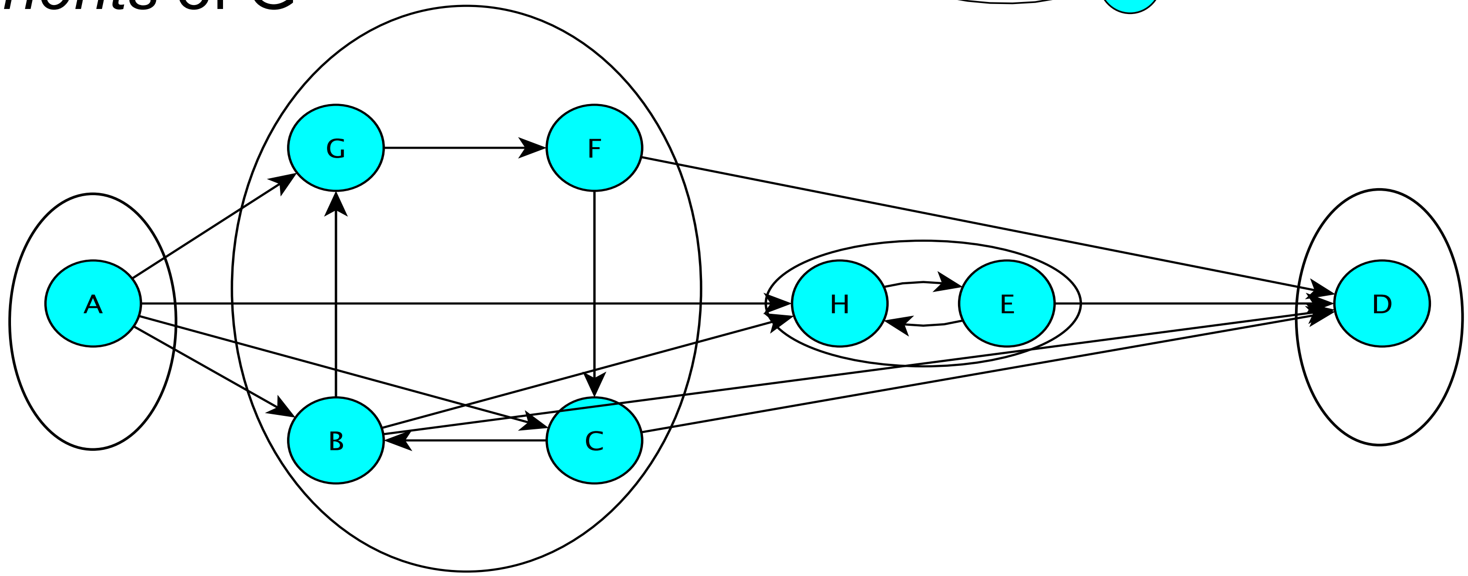


- Connectivity in directed graphs is more subtle than in undirected graphs!

Directed Graphs

- Vertices u and v are *mutually reachable* vertices if there are paths from u to v and v to u
- *Maximal* sets of mutually reachable vertices form *the strongly connected components* of G





Implementing Graphs

- Involves a number of implementation decisions, depending on intended uses
  - What kinds of graphs will be availabe?
    - Undirected, directed, mixed
  - What underlying data structures will be used?
  - What functionality will be provided
  - What aspects will be public/protected/private
- We'll focus on popular implementations for undirected and directed graphs (separately)

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

- So, the methods described in the Structure<V> interface are about 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

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)

Graph Interface Methods

- void add(V vtx), V remove(V vtx)
  - Add/remove vertex to/from graph
- void addEdge(V vtx1, V vtx2, E edgeLabel),

  E removeEdge(V vtx1, V vtx2)
  - Add/remove edge between vtx1 and vtx2
- boolean containsEdge(V vtx1, V vtx2)
  - Returns true iff there is an edge between vtx1 and vtx2
- Edge<V,E> getEdge(V vtx1, V vtx2)
  - Returns edge between vtx1 and vtx2
- void clear()
  - Remove all nodes (and edges) from graph

Graph Interface Methods

- boolean visit(V vertexLabel)
  - Mark vertex as “visited” and return *previous* value of visited flag
- boolean visitEdge(Edge<V,E> e)
  - Mark edge as “visited”
- boolean isVisited(V vtx), boolean isVisitedEdge(Edge<V,E> e)
  - Returns true iff vertex/edge has been visited
- Iterator<V> neighbors(V vtx1)
  - Get iterator for all neighbors of vtx1
  - 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 vtx1, V vtx2,
                  E label, boolean directed)
    ```
  - Construct a (possibly directed) edge between two labeled vertices (vtx1->vtx2)
- Useful methods:

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

Reachability: Breadth-First Traversal

```
BFS(G, v)        // Do a breadth-first search of G starting at v
// pre: all vertices are marked as unvisited
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;
```

Now compare value returned from BFS(G,v) to size of V

Breadth-First Traversal

```
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 Traversal 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 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

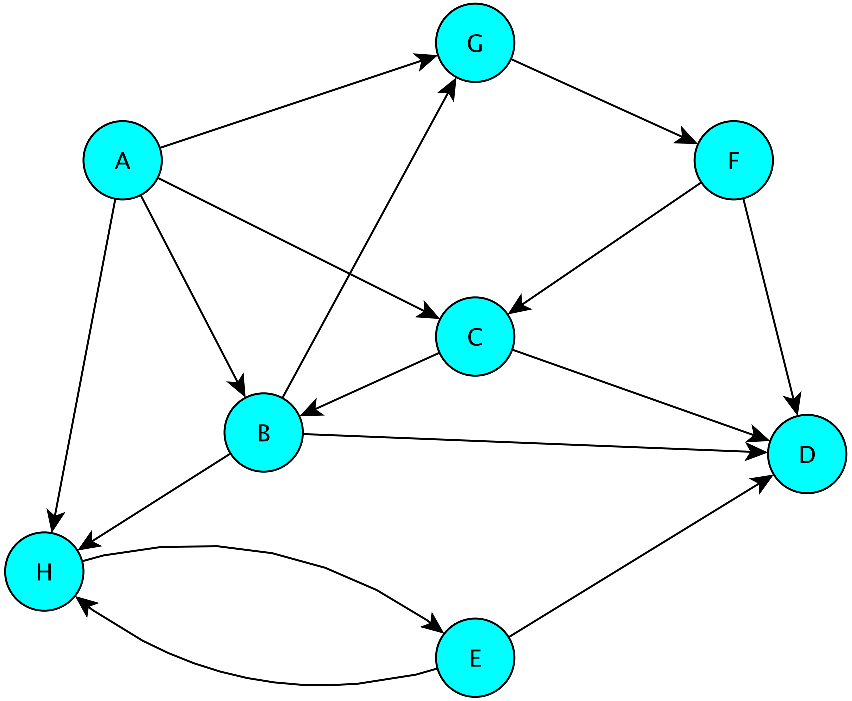
```
int DFS(Graph<V,E> g, V src) {
    g.visit(src);
    int count = 1;
    Iterator<V> neighbors = g.neighbors(src);
    while (neighbors.hasNext()) {
       V next = neighbors.next();
       if (!g.isVisited(next))
             count += DFS(g, next);
    }
  }
  return count;
}
```

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

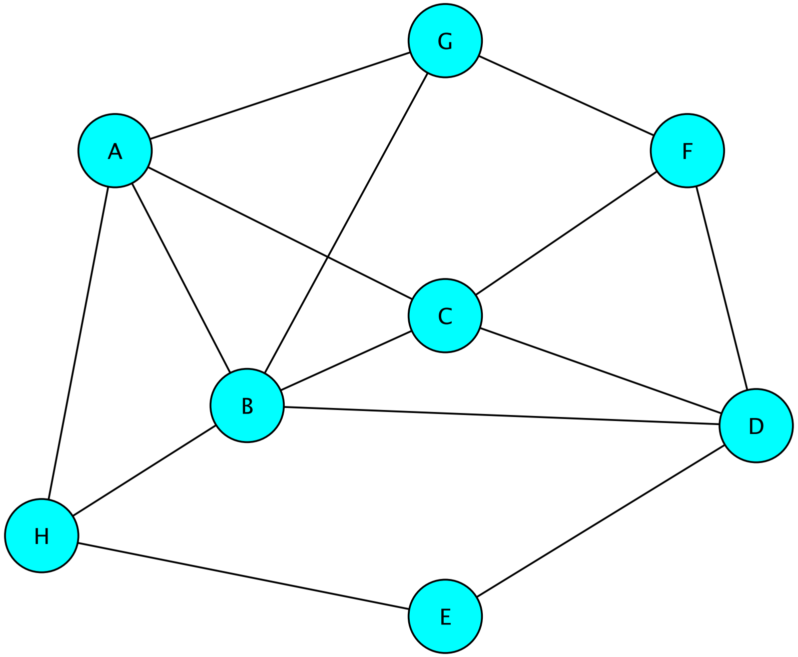
	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	1	1
B	0	0	0	1	0	0	1	1
C	0	1	0	1	0	0	0	0
D	0	0	0	0	0	0	0	0
E	0	0	0	1	0	0	0	1
F	0	0	1	1	0	0	0	0
G	0	0	0	0	0	1	0	0
H	0	0	0	0	1	0	0	0



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

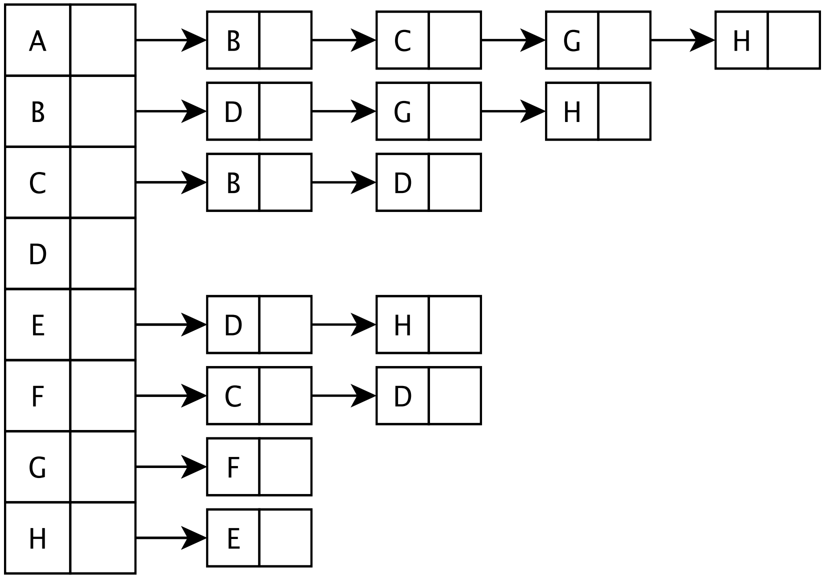
Adjacency Array: Undirected Graph

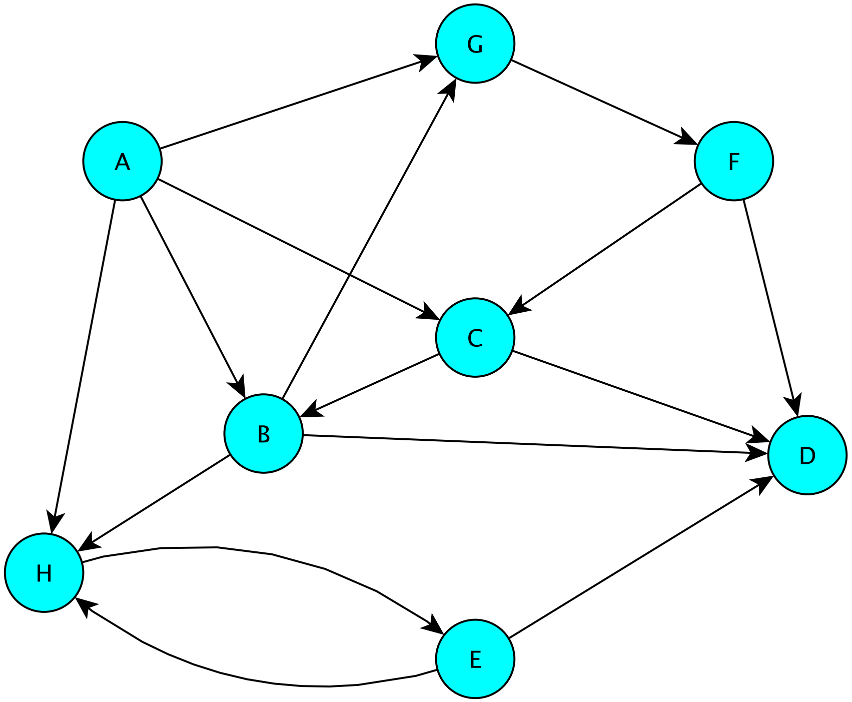
	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	1	1
B	1	0	1	1	0	0	1	1
C	1	1	0	1	0	1	0	0
D	0	1	1	0	1	1	0	0
E	0	0	0	1	0	0	0	1
F	0	0	1	1	0	0	1	0
G	1	1	0	0	0	1	0	0
H	1	1	0	0	1	0	0	0



Entry (i,j) stores 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

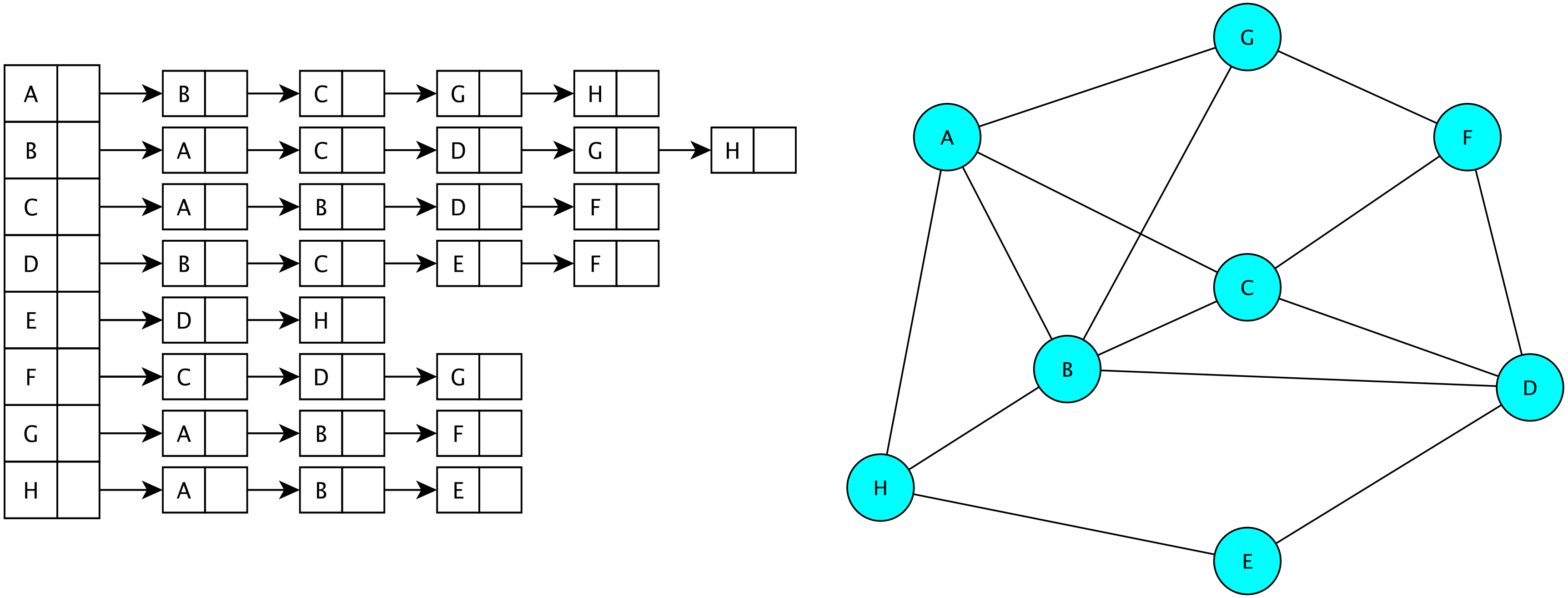




The vertices are stored in a data structure (we'll see how in a second)

This structure contains a linked list of **edges** having a given source

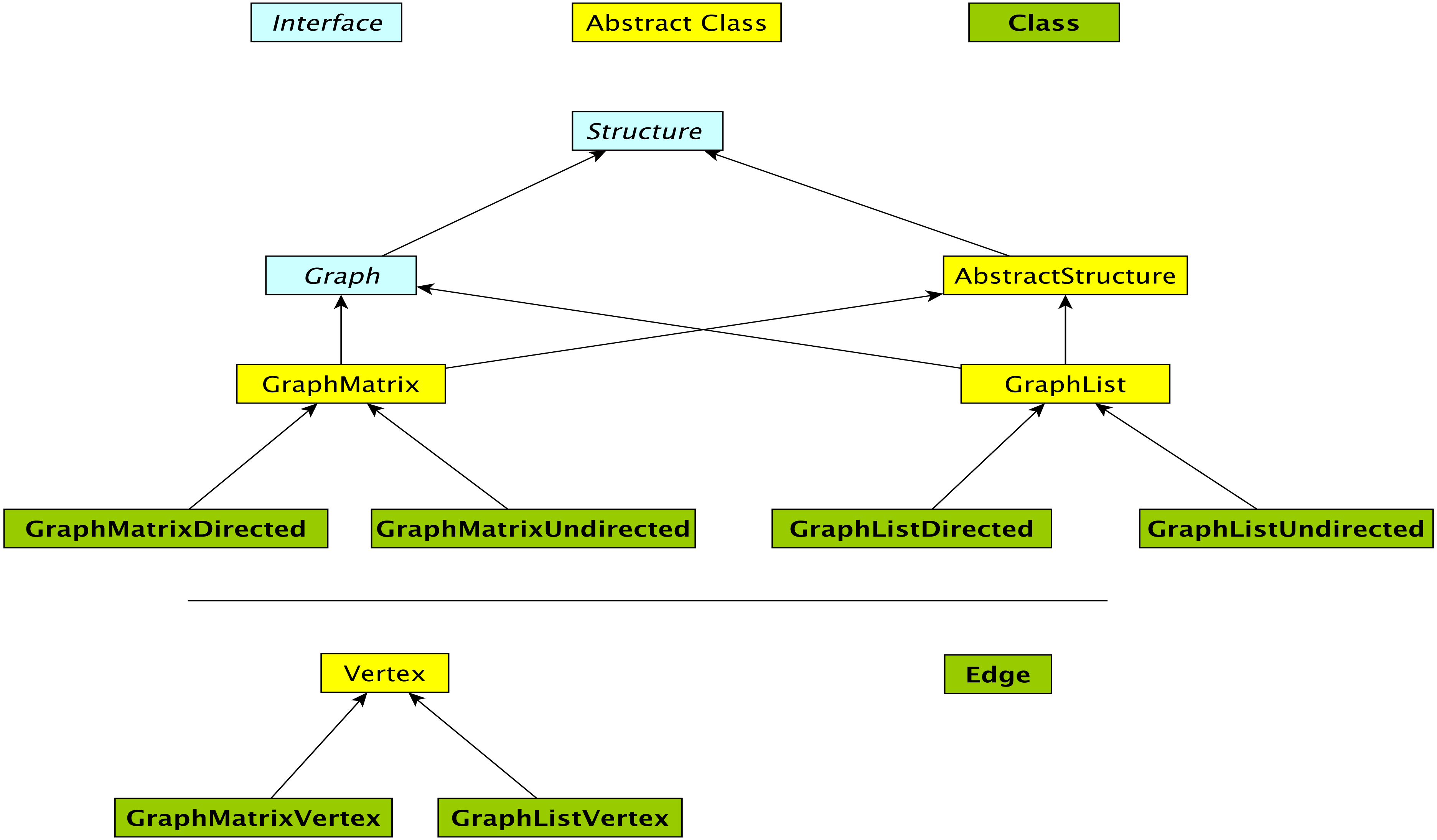
Adjacency List : Undirected Graph



The vertices are stored in a data structure (we’ll see how in a second)

This structure contains a linked list of **edges** incident to a given vertex

Graph Classes in structure5



Interface

Abstract Class

Class

Structure

Graph

AbstractStructure

GraphMatrix

GraphList

GraphMatrixDirected

GraphMatrixUndirected

GraphListDirected

GraphListUndirected

Vertex

Edge

GraphMatrixVertex

GraphListVertex

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....

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.
  - For now, see MapDemo.java example for simple use

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();
   }
```

GraphMatrixDirected

- Completes the implementation of GraphMatrix to ensure graph is directed
- GraphMatrixUndirected is very similar…
- How do we implement GraphMatrixDirected?
  - We'll discuss some methods
  - Read Ch 16 for complete details…

GraphMatrixDirected

- Constructor

```
public GraphMatrixDirected(int size) {
   // pre: size > 0
   // post: constructs an empty graph that may be
   //       expanded to at most size vertices. Graph
   //       is directed if dir true and undirected
   //       otherwise

   // call GraphMatrix constructor
   super(size,true);
}
```

GraphMatrixDirected

- addEdge

```
// pre: vLabel1 and vLabel2 are labels of existing vertices
public void addEdge(V vLabel1, V vLabel2, E label) {
   GraphMatrixVertex<V> vtx1,vtx2;
   vtx1 = dict.get(vLabel1);
   vtx2 = dict.get(vLabel2);
   Edge<V,E> e = new Edge<V,E>(vtx1.label(), vtx2.label(),
                               label, true);
   data[vtx1.index()][vtx2.index()] = e;
}
```

GraphMatrixDirected

- removeEdge

```
// pre: vLabel1 and vLabel2 are labels of existing vertices
public E removeEdge(V vLabel1, Vlabel2) {
    // get indices
   int row = dict.get(vLabel1).index();
   int col = dict.get(vLabel2).index();
   // cache old value
   Edge<V,E> e = (Edge<V,E>)data[row][col];
   // update matrix
   data[row][col] = null;
   if (e == null) return null;
   else return e.label(); // return old value
}
```

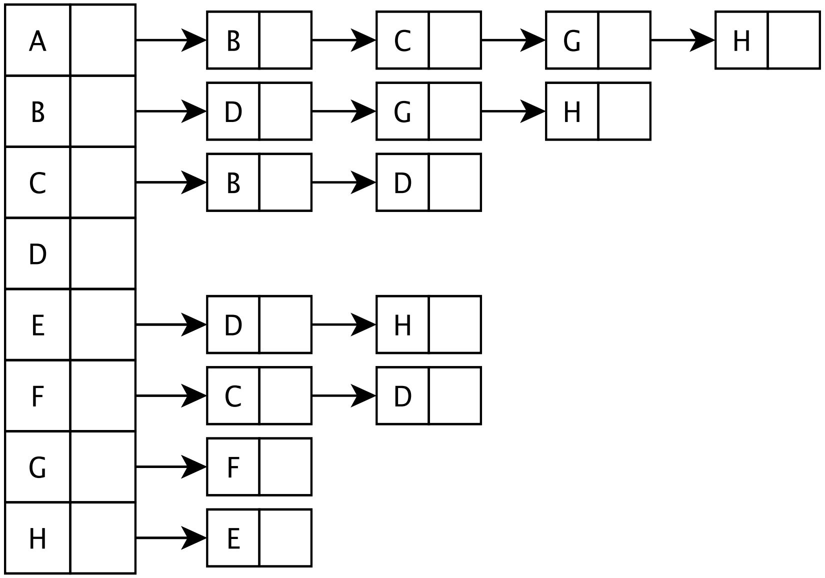
GraphMatrix Efficiency

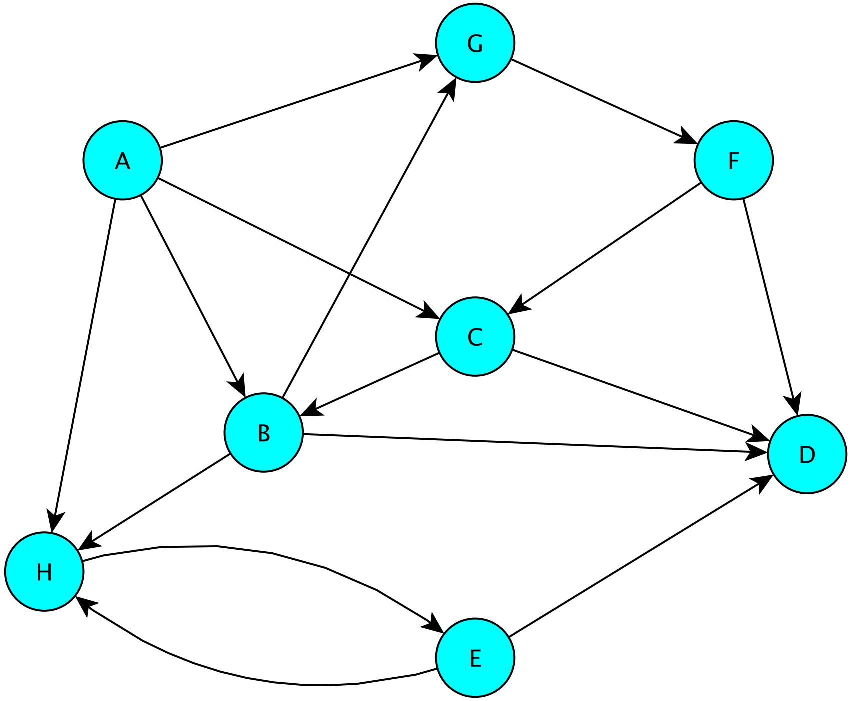
- Assume Map operations are O(1) (for now)
  - |E| = number of edges
  - |V| = number of vertices
- Runtime of add, addEdge, getEdge, removeEdge, remove?
- Space usage?
- Conclusions
  - Matrix is good for dense graphs
  - Have to commit to maximum # of vertices in advance

Efficiency : Assuming Fast Map

	GraphMatrix
add	O(1)
addEdge	O(1)
getEdge	O(1)
removeEdge	O(1)
remove	O(|V|)
space	$O(\|V\|^2)$

Adjacency List : Directed Graph

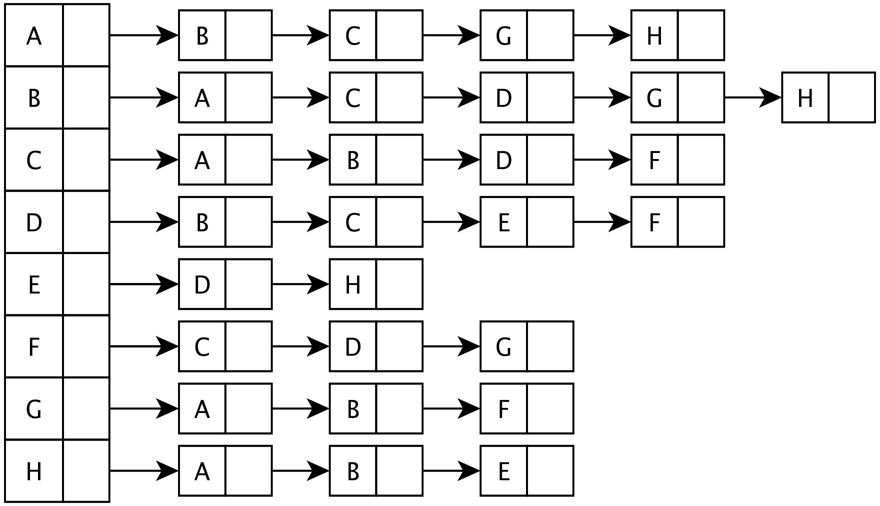


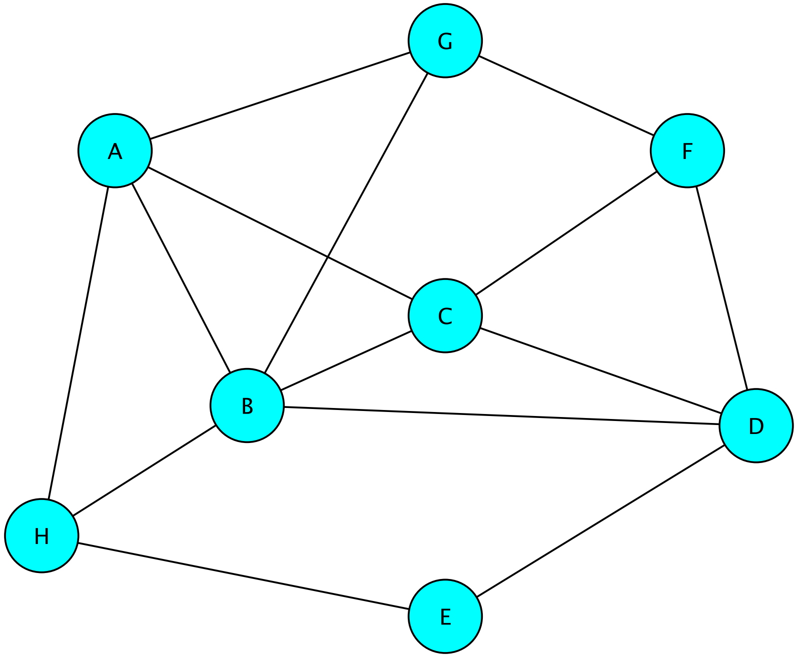


The vertices are stored in an map

Each vertex contains a linked list of edges having a given source

Adjacency List : Undirected Graph





The vertices are stored in a map
Each vertex contains a linked list of edges incident to a given vertex

GraphList

- Rather than keep an adjacency matrix, maintain an *adjacency list of edges* at each vertex (only keep outgoing edges for directed graphs)
- Support both directed and undirected graphs (GraphListDirected, GraphListUndirected)

Vertex and GraphListVertex

- We use the same Edge class for list-based graphs
- We extend Vertex to include an Edge list
- GraphListVertex class adds to Vertex class
  - A Structure to store edges adjacent to the vertex

    protected Structure<Edge<V,E>> adjacencies; // adjacent edges

    - adjacencies is created as a SinglyLinkedList of edges
  - 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

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

public void addEdge(Edge<V,E> e){
        if (!containsEdge(e)) adjacencies.add(e);
}

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

// 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

GraphListAIterator uses two instance variables

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

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

GraphListEIterator

GraphListEIterator uses one instance variable

```
protected AbstractIterator<Edge<V,E>> edges;
```

GraphListEIterator

- Takes the Map storing the vertices
- Uses it to build a linked list of all edges
- Gets an iterator for this linked list and stores it, using it in its own methods

GraphList

- To implement GraphList, we use the GraphListVertex (GLV) class
- GraphListVertex class
  - Maintain linked list of edges at each vertex
  - Instance vars: label, visited flag, linked list of edges
- GraphList abstract class
  - Instance vars:
    - Map<V,GraphListVertex<V,E>> dict; // label -> vertex
    - boolean directed; // is graph directed?
- How do we implement key GL methods?
  - GraphList(), add(), getEdge(), …

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

public void add(V label) {
      if (dict.containsKey(label)) return;
      GraphListVertex<V,E> v = new
            GraphListVertex<V,E>(label);
      dict.put(label,v);
}

public Edge<V,E> getEdge(V label1, V label2) {
      Edge<V,E> e = new Edge<V,E> (get(label1),
      get(label2), null, directed);
      return dict.get(label1).getEdge(e);
}
```

GraphListDirected

- GraphListDirected (GraphListUndirected) implements the methods requiring different treatment due to (un)directedness of edges
  - addEdge, remove, removeEdge, …

```
// addEdge 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);
}
```

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

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

        //Skip over the vertex label we're removing
        //(Nodes don't have edges to themselves...)
        if (!label.equals(v2)) {
            //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();
}
```

```
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<V,E>(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();
}
```