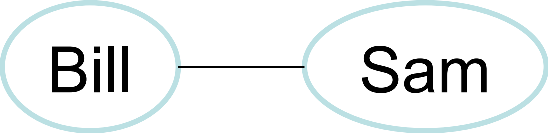
# CSCI 136
# Data Structures & Advanced Programming

Lecture 28

Fall 2019

Instructors:



Bill

Sam

# Last Time

- Lab 9: Super Lexicon!
- Introduction To Graphs
  - Definitions and Properties: Undirected Graphs

# Today' s Outline

- More on Graphs
  - Applications and Problems
    - Testing connectedness
    - Counting connected components
      - Breadth-first and Depth-first search
  - Directed Graphs
    - Definition and Properties
  - Reachability and (Strong) Connectedness
- Graph Data Structures: Preliminaries
  - Graph Interface

# Reachability and Connectedness

- Def’n: A vertex v in G is *reachable* from a vertex u in G if there is a path from u to v
- v is reachable from u *iff* u is reachable from v
- Def’n: An undirected graph G is *connected* if for every pair of vertices u, v in G, v is reachable from u (and vice versa)
- The set of all vertices reachable from v, along with all edges of G connecting any two of them, is called the *connected component of v*

# Operations on Graphs

- 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 vertices *adjacent* to v
    - From which we can get the edges *incident* with v

# Testing Connectedness

- How can we determine whether G is connected?
  - Pick a vertex v; see if every vertex u is reachable from v
- How could we do this?
  - Visit the neighbors of v, then visit their neighbors, etc. See if you reach all vertices
    - Assume we can mark a vertex as "visited"
- How do we *efficiently* manage all this visiting?

# Reachability: Breadth-First Search

```
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

# BFS Theorem

Thm. BFS(G,v) visits exactly those vertices u reachable from v.

Proof: We'll show that if u is reachable from v then BFS(G,v) visits u by induction on d = d(v,u)

- Base Case: d = 0. Then u = v.
  - v is reachable from v and BFS(G,v) visits v
- Induction Hypothesis: For some d ≥ 0, if d(u,v) = d then BFS(G,v) visits u.

# BFS Theorem

- Induction Step: Assume now that d(u,v) = d+1
  - Let v = $v_0$, $e_1$, $v_1$, $e_2$, $v_2$, … , $v_d$, $e_{d+1}$, $v_{d+1}$ = u be a path of length d+1 from v to u
  - Then v = $v_0$, $e_1$, $v_1$, $e_2$, $v_2$, … , $v_d$ is a path of length d from v to $v_d$
  - By I.H., $v_d$ is visited by BFS(G,v) and put in Q
  - So $v_d$ will be dequeued and all of its unvisited neighbors, including u, will be marked as visited

A similar argument shows that if u is visited by BFS(G,v) then u is reachable from v

# BFS Reflections

- The BFS algorithm can be modified to build a tree $T_v$: the edges connecting a visited vertex to (as yet) unvisited neighbors
- $T_v$ is called a *BFS tree of G with root v* (or *from v*)
- The vertices of $T_v$ are visited in *level-order*
- Every path in $T_v$ from v to a vertex u is a *shortest possible path* from v to u
    - That is, the path has length d(v,u)

# Reachability: Depth-First Search

```
DFS(G, v)        // Do a depth-first search of G starting at v
// pre: all vertices are marked as unvisited
count ←0;
Create empty stack S; push v; mark v as visited; count++;
While S isn't empty
        current ←S.pop();
        for each unvisited neighbor u  of current :
                add u to S; mark u as visited; count++
return count;
```

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

# DFS Reflections

- The DFS algorithm traced out a tree different from that produced by BFS
  - It still consists of the edges connecting a visited vertex to (as yet) unvisited neighbors
- It is called a DFS *tree of G with root v* (or *from v*)
- Vertices are processed in pre-order w.r.t. the tree
- By manipulating the stack differently, we could produce a post-order version of DFS
- And perhaps write DFS recursively….

# 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 neighbor u of v:
        if u is unvisited:
            count += DFS(G,u);
    return count;
```

Is it even clear that this method does what we want?!
Let's prove some facts about it....

# What *Exactly* Does DFS Do?

- Given a graph G = (V, E), a vertex v, let X ⊆ V, where v ∉ X.
- Assume X are exactly the vertices of V that have been marked as visited
- Claim: DFS(G,v) will visit exactly those unvisited vertices that are in the connected component of G – X that contains v
  - G – X is the graph obtained by deleting the vertices of X–and edges using X–from G
  - Prove by induction on |V – X|

# Recursive Depth-First Search

Claim: DFS visits all vertices w reachable from v

- Proof: Induction on length d of shortest path from v to w
  - Base case: d = 0: Then v = w ✓
  - Ind. Hyp.: Assume DFS visits all vertices w of distance at most d from v (for some d ≥ 0).
  - Ind. Step: Suppose now that w is distance d+1 from v. Consider a path of length d+1 from v to w and let u be the next-to-last vertex on the path

# Recursive Depth-First Search

Claim: DFS visits all vertices w reachable from v

- Proof: Induction on length d of shortest path from v to w
  - The path is $v = v_0, v_1, v_2, \ldots, v_d = u, v_{d+1} = w$
    - The edges are implied so not explicitly written!
  - By Ind. Hyp., u is visited. At this point, if w has not yet been visited, it will be one of the unvisited vertices on which DFS() is recursively called, so it will then be visited.

# Recursive Depth-First Search

Claim: DFS visits *only* vertices reachable from v

- Idea: Prove by induction on number of times DFS is called that DFS is only called on vertices w reachable from v

Claim: DFS counts correctly the number of vertices reachable from v

- Idea: Induction on number of unvisited vertices reachable from v
  - DFS will never be called on same vertex twice

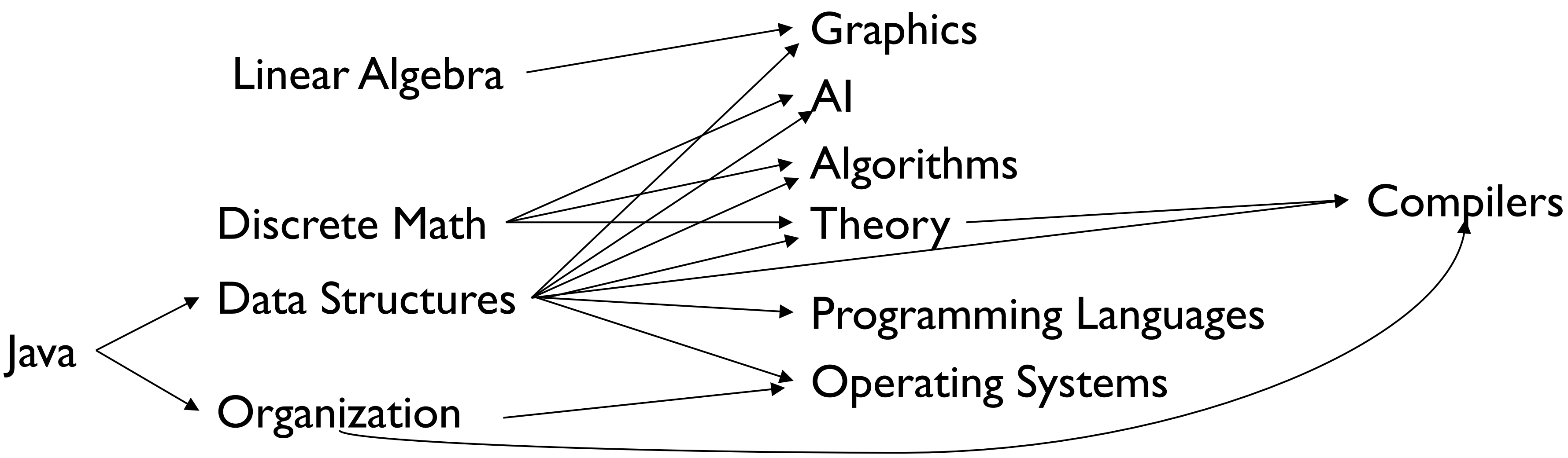
# Recursive Depth-First Search

Claim: DFS(G,v) returns the number of unvisited nodes reachable from v

Proof: Uses previous two observations

- DFS visits every node reachable from v
- DFS doesn’t visit any node *not* reachable from v

# Directed Graphs


Linear Algebra

Discrete Math

Data Structures

Java

Organization

Graphics

AI

Algorithms

Theory

Programming Languages

Operating Systems

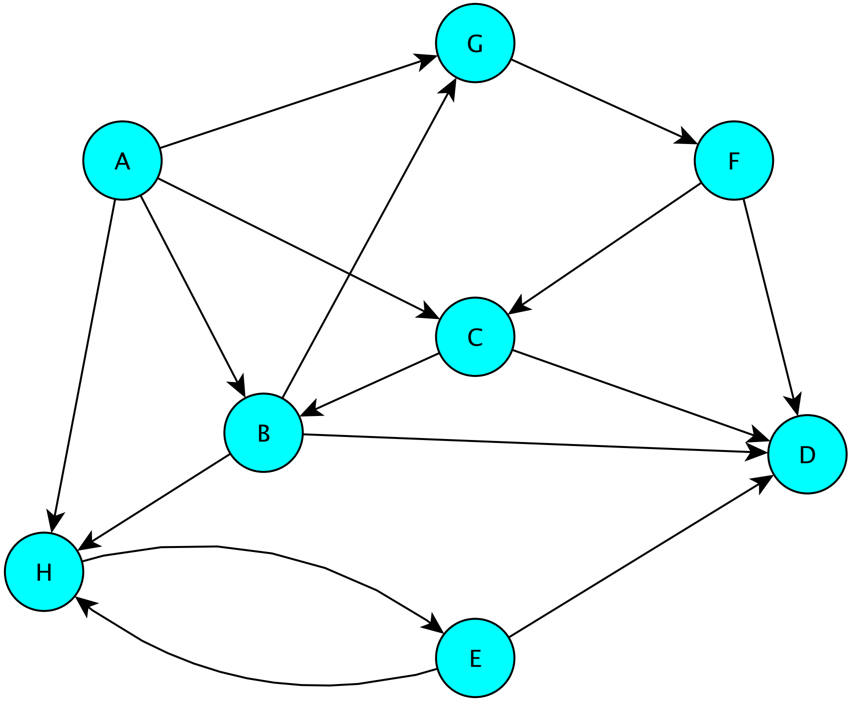
Compilers

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



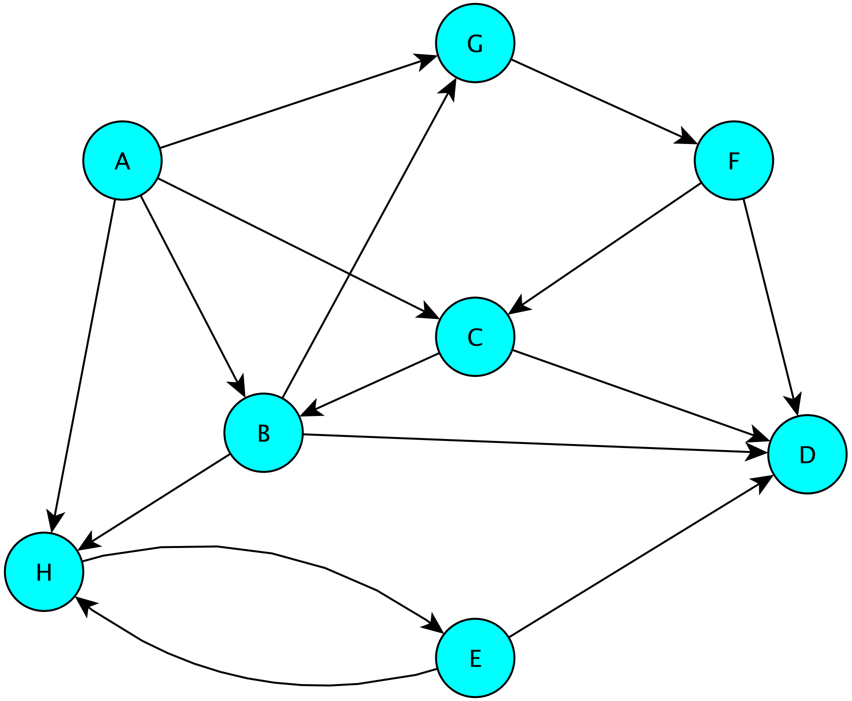
A walk is still an alternating sequence of vertices and edges

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

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

# Directed Graphs

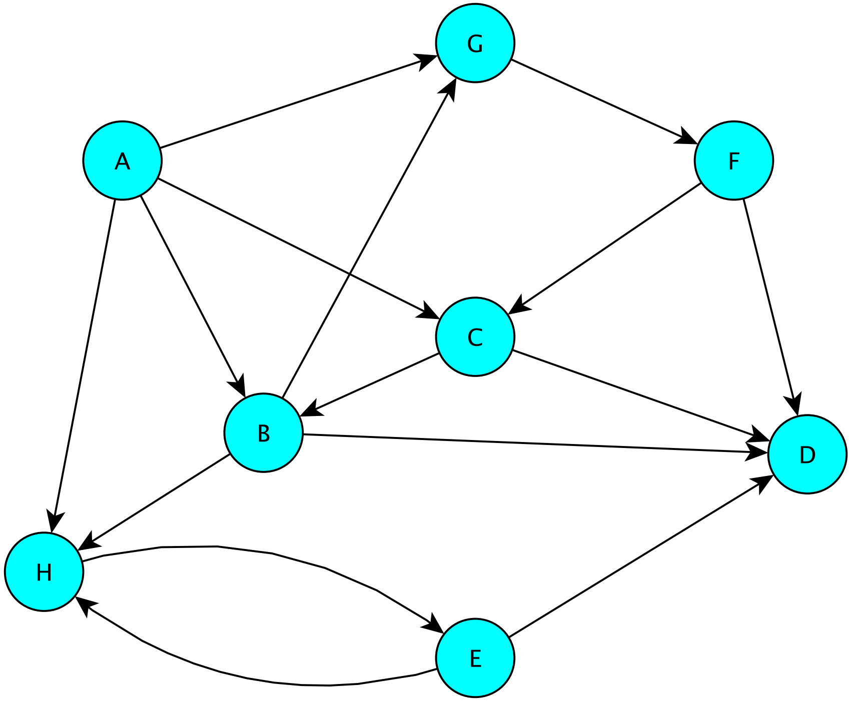
- A, B, H, E, D is a walk from A to D
- It's also a (simple) path
- D, E, H, B, A is *not* a walk from D to A
- B, G, F, C, B is a (directed) cycle (it's a 4-cycle)
- So is H, E, H (a 2-cycle)



- 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

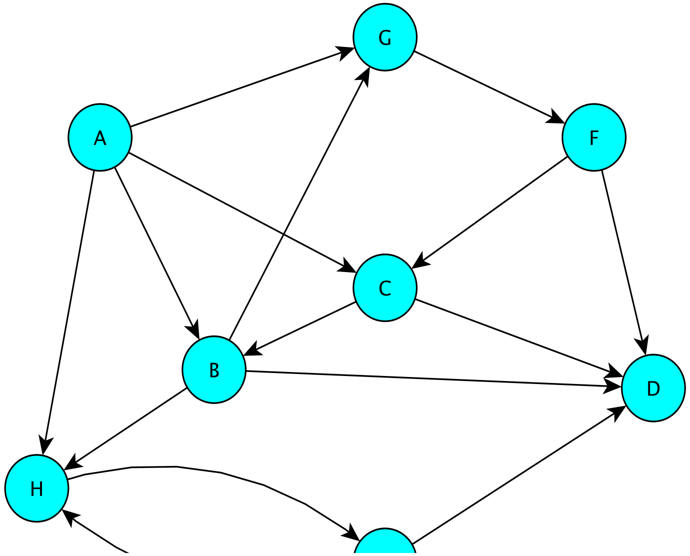
- A BFS of *G* from A visits every vertex
- A BFS of *G* from F visits all vertices but A
- A BFS of *G* from E visits only E, H, D

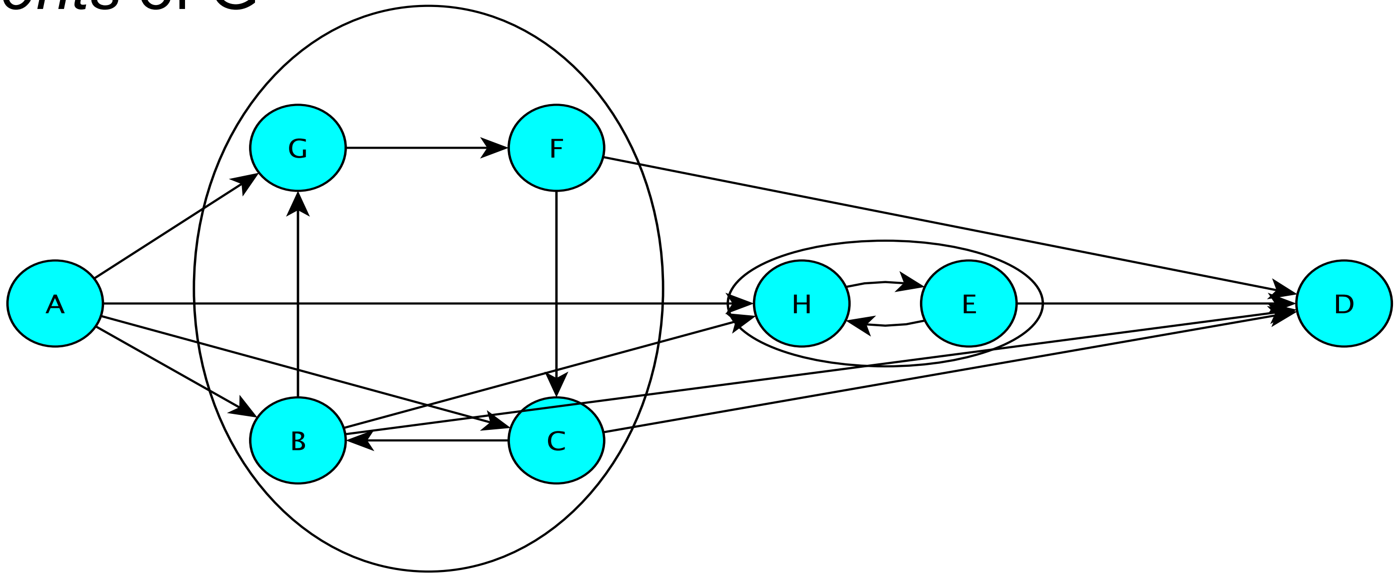


- Connectivity in directed graphs is more subtle than in undirected graphs!

# 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

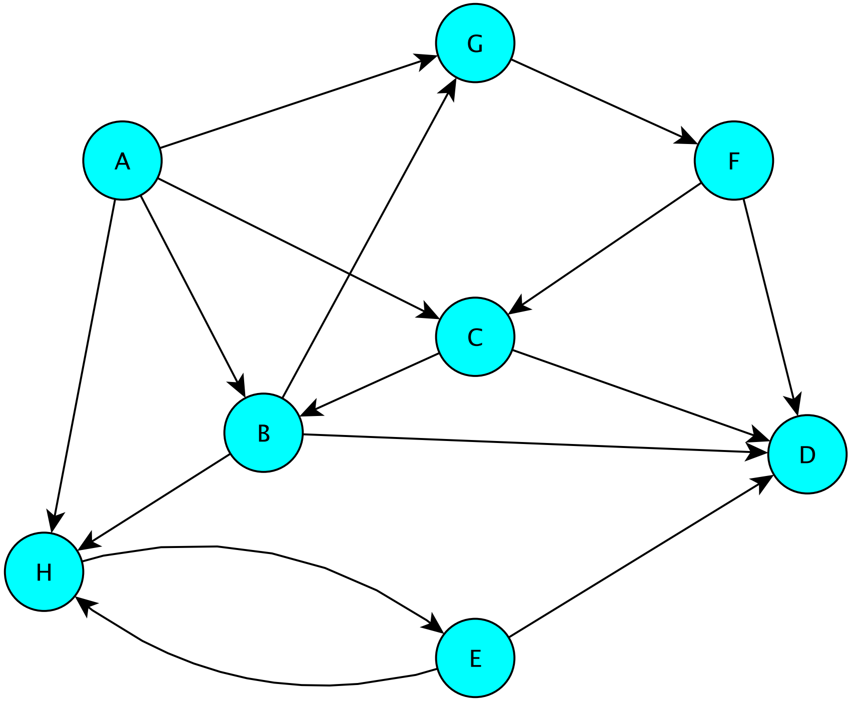
```
int DFS(Graph<V,E> g, V src) {
    g.visit(src);
    int count = 1;
    Iterator<V> neighbors = g.neighbors(src);
    while (neighbors.hasNext()) {
       V next = neighbors.next();
       if (!g.isVisited(next))
              count += DFS(g, next);
    }
  }
  return count;
}
```

# 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

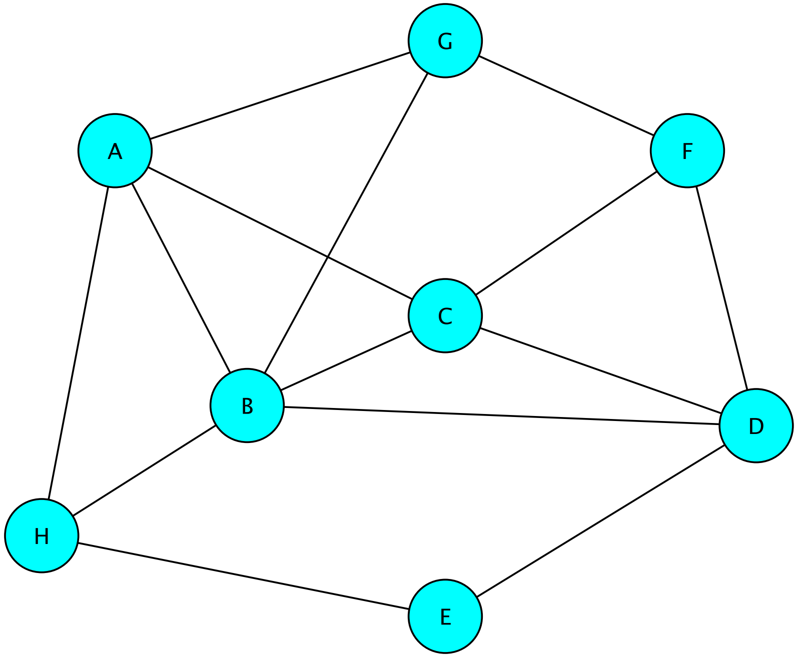
| | A | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| B | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| C | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| D | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| E | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| F | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |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

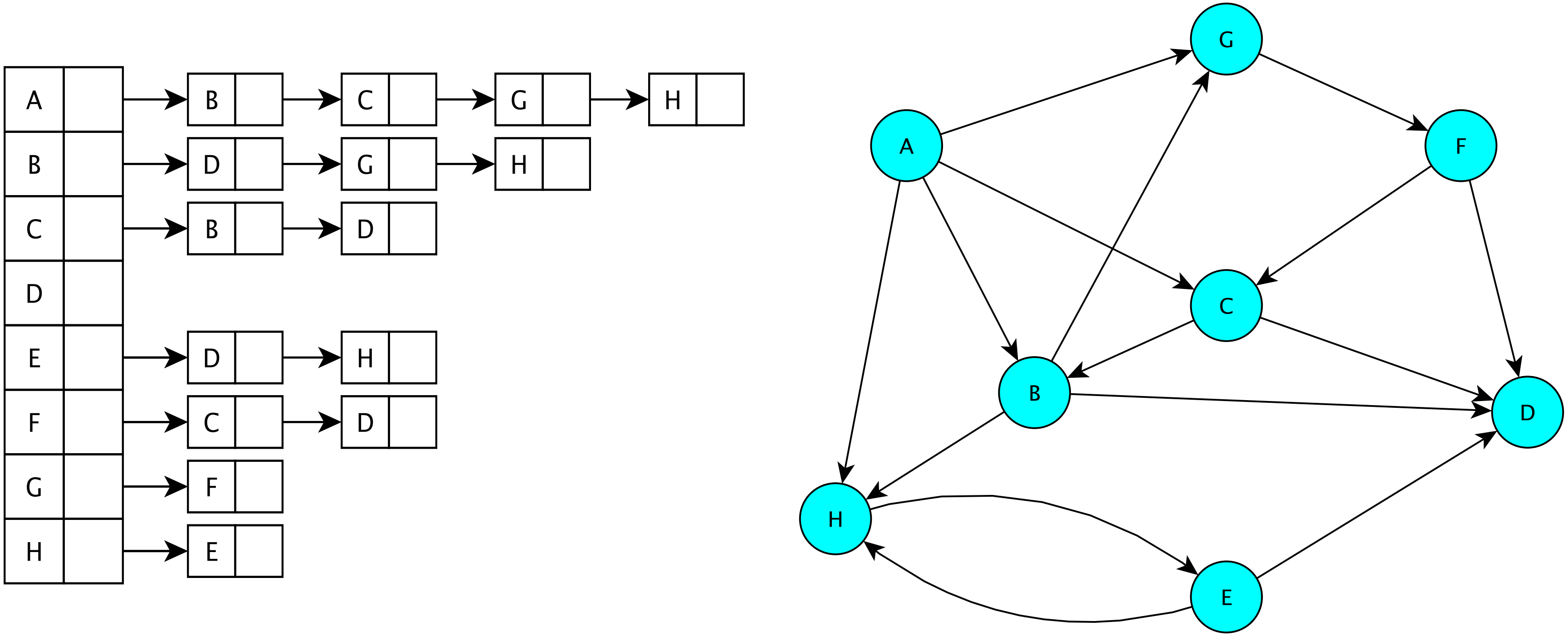
# Adjacency Array: Undirected Graph

|   | 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 |



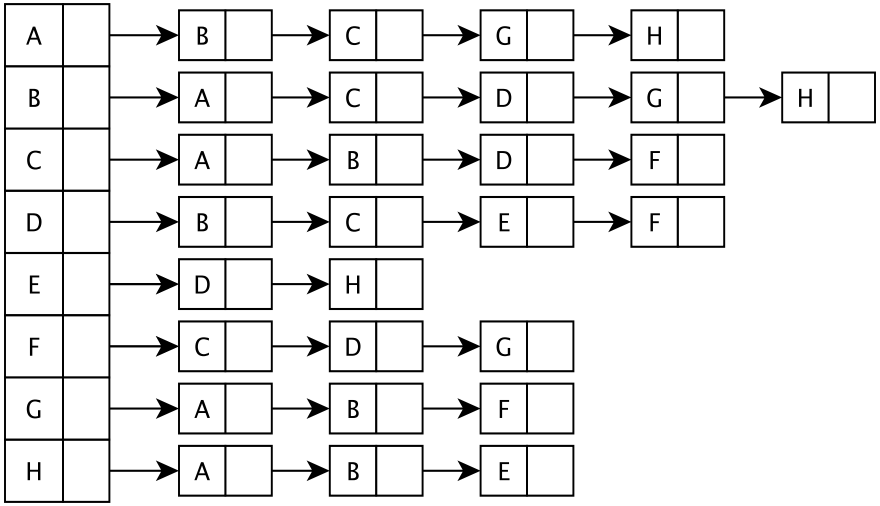
Entry (i,j) store 1 if there is an edge between i and j; else 0
E.G.: edges(B,C) = 1 = edges(C,B)

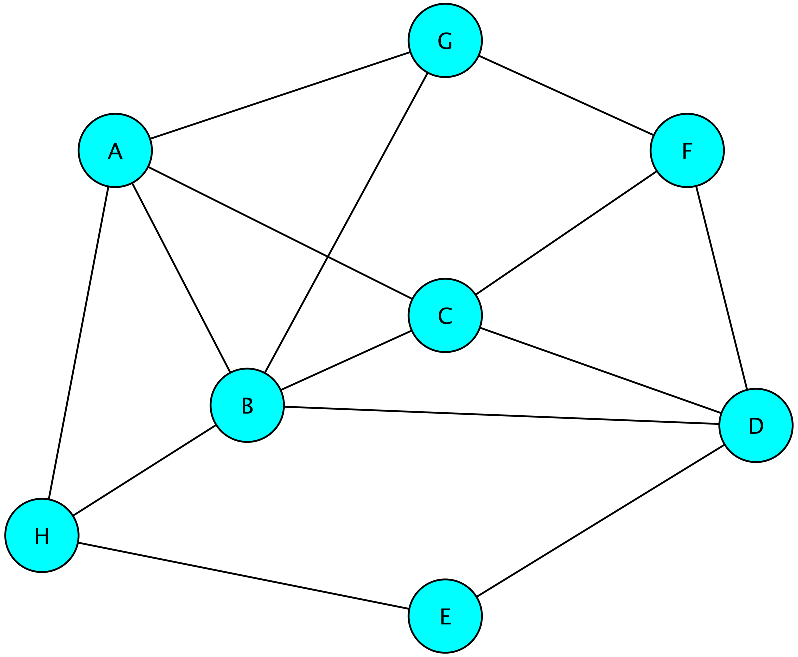
# Adjacency List : Directed Graph



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

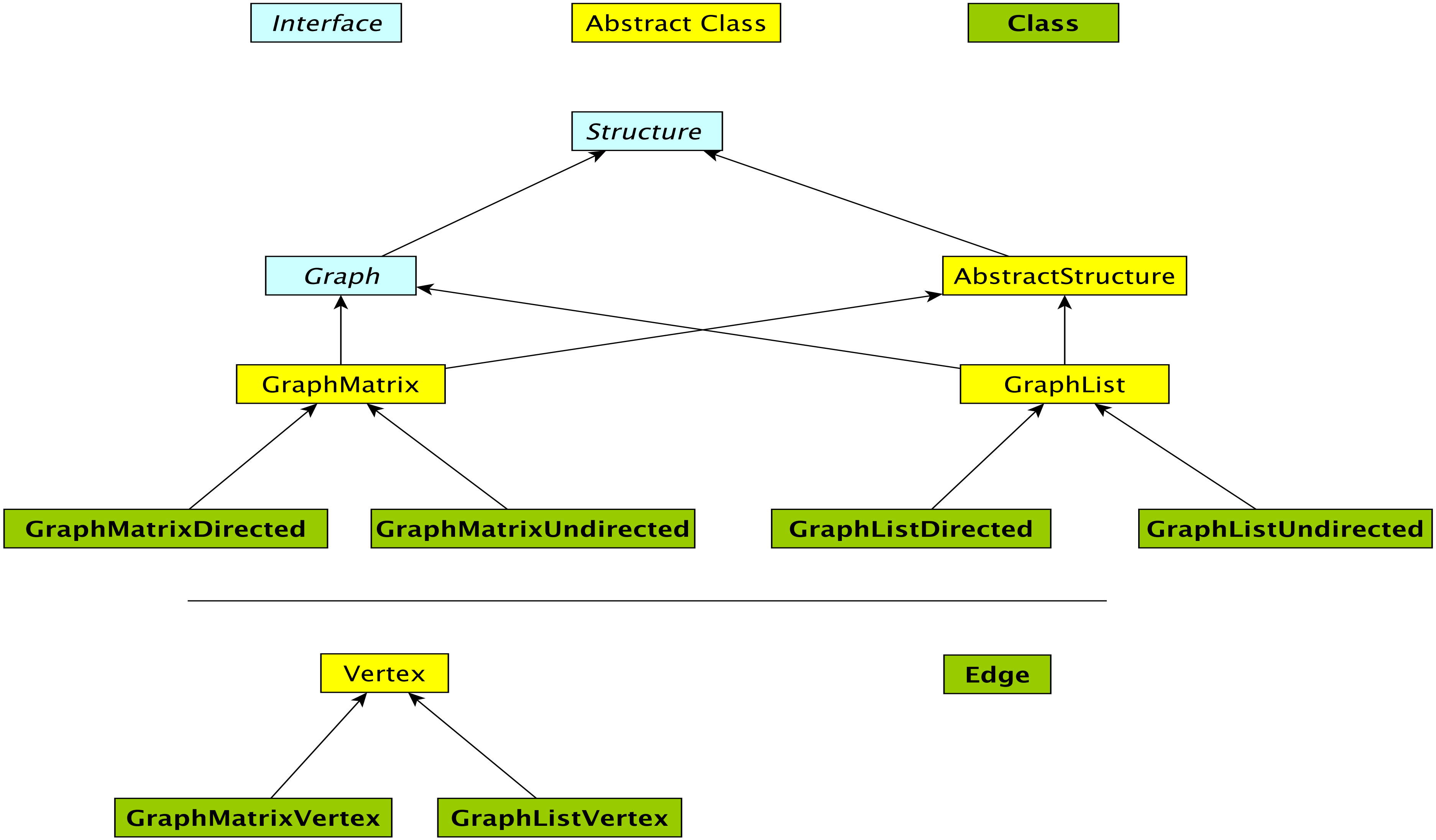
# Adjacency List : Undirected Graph





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

# Graph Classes in structure5



*Interface*

Abstract Class

**Class**

*Structure*

*Graph*

AbstractStructure

GraphMatrix

GraphList

**GraphMatrixDirected**

**GraphMatrixUndirected**

**GraphListDirected**

**GraphListUndirected**

Vertex

**Edge**

**GraphMatrixVertex**

**GraphListVertex**