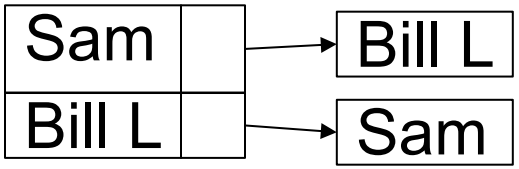
CSCI 136
Data Structures & Advanced Programming

Lecture 30

Fall 2019

Instructors:



Admin

- Lab 10 today
- Lab 7 back
- Lab 8 and PS 3 soon

- No lab next week (Thanksgiving) or the week after

Last Time

- Graph Data Structures: Implementation
  - Graph Interface

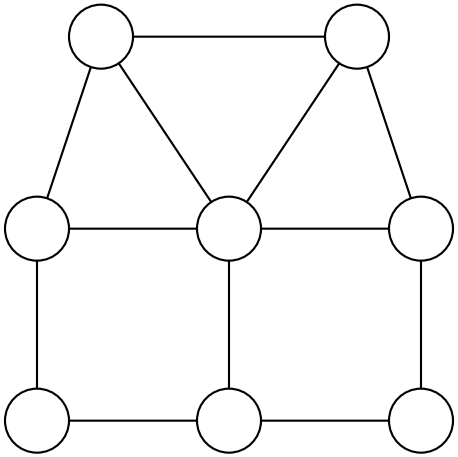
Today's Outline

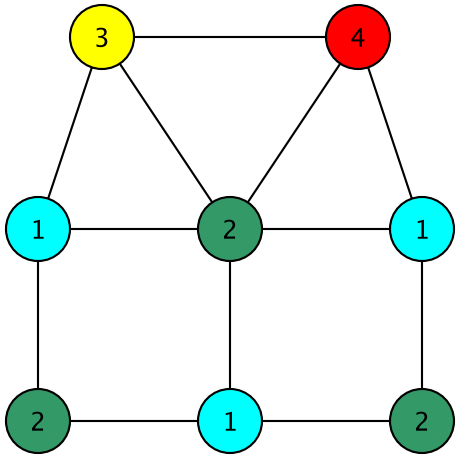
- Lab 10: Greedy Algorithms
  - Exam Scheduling
- Graph Data Structures: Implementation
  - Adjacency Array Implementation
  - Adjacency List Implementation
    - Featuring many Iterators!

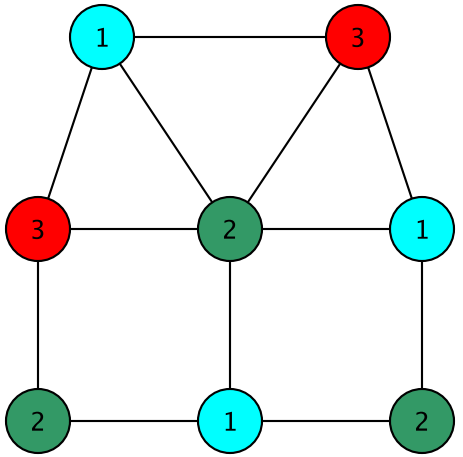
Greedy Algorithms

- A *greedy algorithm* attempts to find a globally optimum solution to a problem by making locally optimum (greedy) choices
- Example: Graph Coloring
  - A *(proper) coloring* of a graph G = (V,E) is an assignment of a value (color) to each vertex so that adjacent vertices get different values (colors)
  - Typically one strives to minimize the number of colors used

Greedy Coloring







Greedy Coloring : CS

Here's a greedy coloring algorithm

Create a structure C to hold a collection of lists

while G is not empty

    pick a vertex v; create an empty list L; add v to L

    for each vertex u != v in G

        if u is not adjacent to any vertex of L

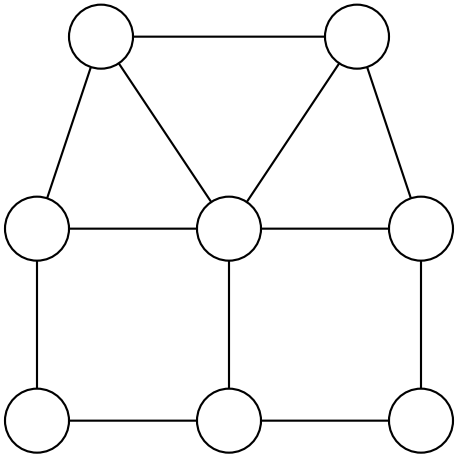
            add u to L

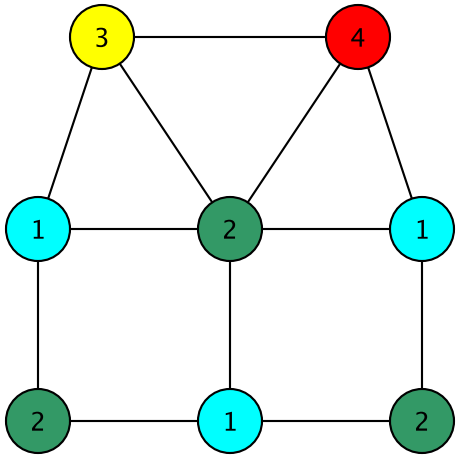
    remove all vertices of L from G

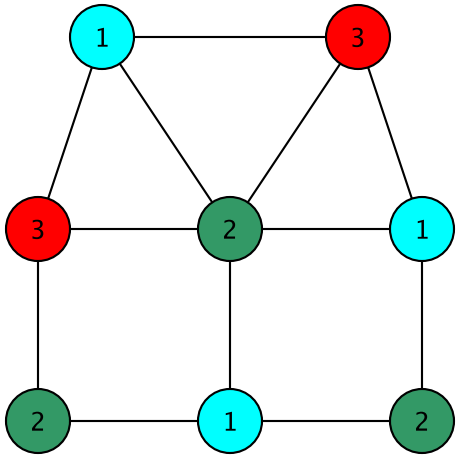
    add L to C

Return C as the coloring

Greedy Coloring







Greedy Coloring

Some observations

- Each list (color class) L is a set of vertices no two of which are adjacent (an *independent set*)
- Each color class is maximal: cannot be made any larger
  - The hope is that this results in fewer colors being needed
  - But the solution is not always optimum!
  - This is a *very hard problem*
- The coloring problem is the same as finding a *partition* of the vertex set into independent sets
  - Partition means union of disjoint sets

Lab 10 : Exam Scheduling

Find a schedule (set of time slots) for exams so that

- No student has two exams in the same slot
- Every course is in a slot
- The number of slots is as small as possible

This is just the graph coloring problem in disguise!

- Each course is a vertex
- Two vertices are adjacent if the courses share students
- A slot must be an independent set of vertices (that is, a color class)

Lab 10 Notes: Using Graphs

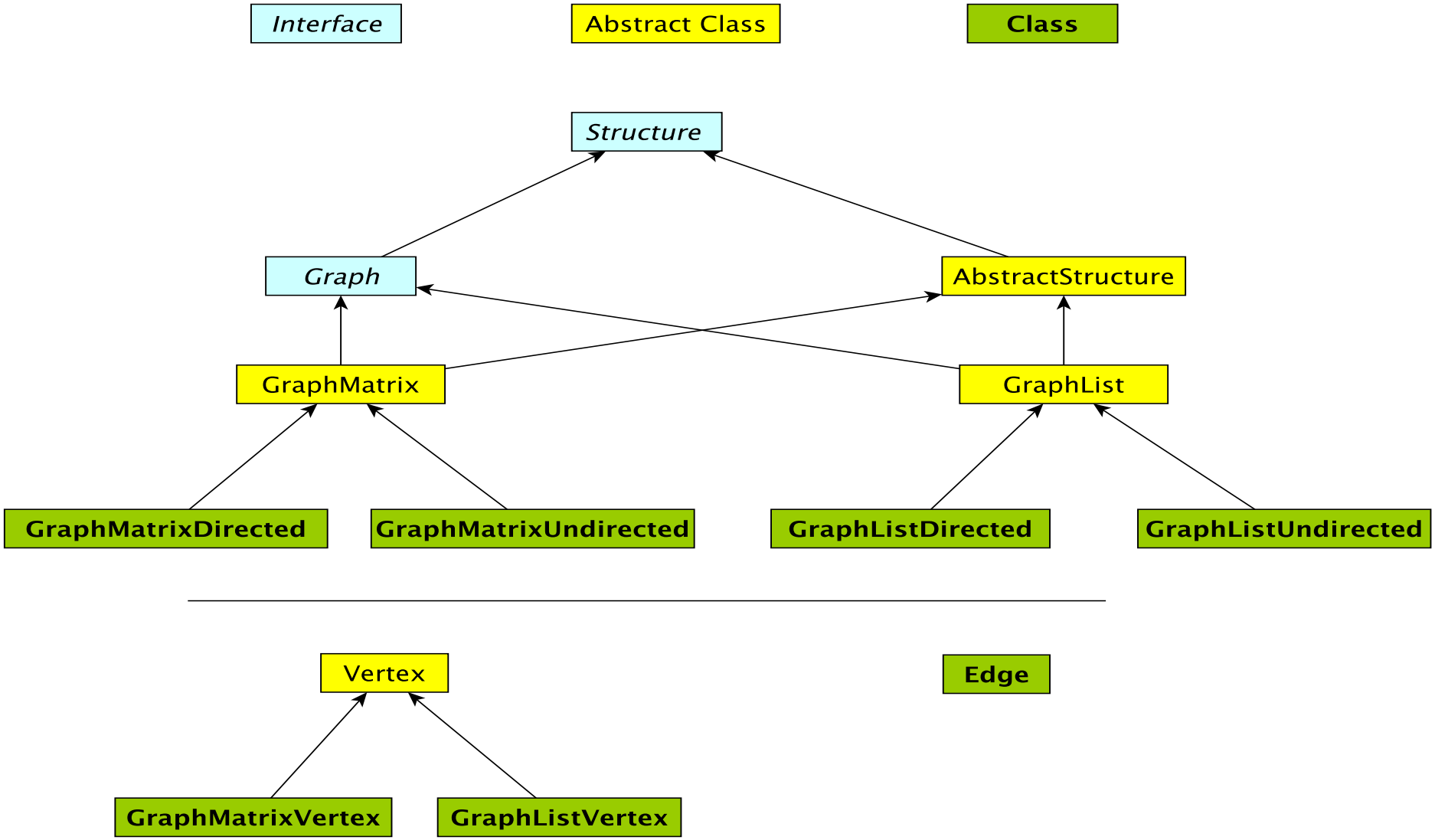
- Create a new graph in structure5
  - GraphListDirected, GraphListUndirected,
  - GraphMatrixDirected, GraphMatrixUndirected

- Graph<V,E> conflictGraph = new GraphListUndirected<V,E>();

Lab 10 : Useful Graph Methods

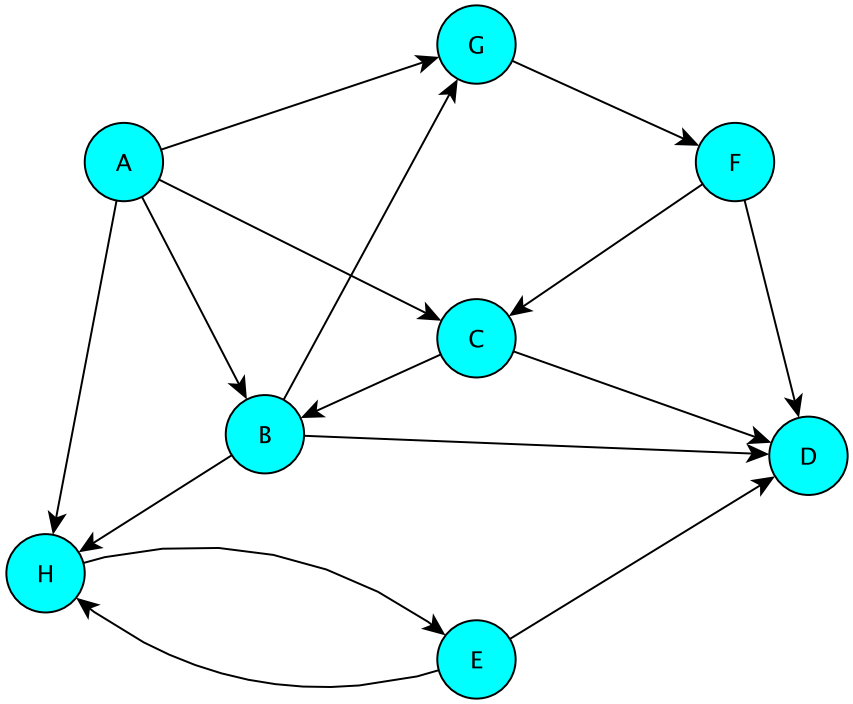
- `void add(V label)`
  - add vertex to graph
- `void addEdge(V vtx1, V vtx2, E label)`
  - add edge between vtx1 and vtx2
- `Iterator<V> neighbors(V vtx1)`
  - Get iterator for all neighbors to vtx1
- `boolean isEmpty()`
  - Returns true iff graph is empty
- `Iterator<V> iterator()`
  - Get vertex iterator
- `V remove(V label)`
  - Remove a vertex from the graph
- `E removeEdge(V vLabel1, V vLabel2)`
  - Remove an edge from graph

Graph Classes in structure5



Adjacency Array: Directed Graph

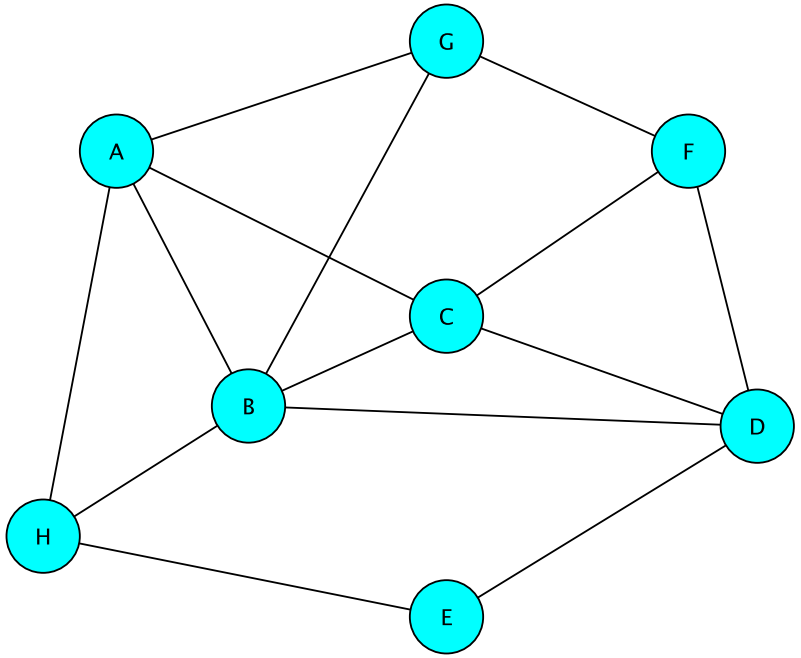
	A	B	C	D	E	F	G	H
A	0	1	1	0	0	0	1	1
B	0	0	0	1	0	0	1	1
C	0	1	0	1	0	0	0	0
D	0	0	0	0	0	0	0	0
E	0	0	0	1	0	0	0	1
F	0	0	1	1	0	0	0	0
G	0	0	0	0	0	1	0	0
H	0	0	0	0	1	0	0	0



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	I	I	0	0	0	I	I
B	I	0	I	I	0	0	I	I
C	I	I	0	I	0	I	0	0
D	0	I	I	0	I	I	0	0
E	0	0	0	I	0	0	0	I
F	0	0	I	I	0	0	I	0
G	I	I	0	0	0	I	0	0
H	I	I	0	0	I	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)

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

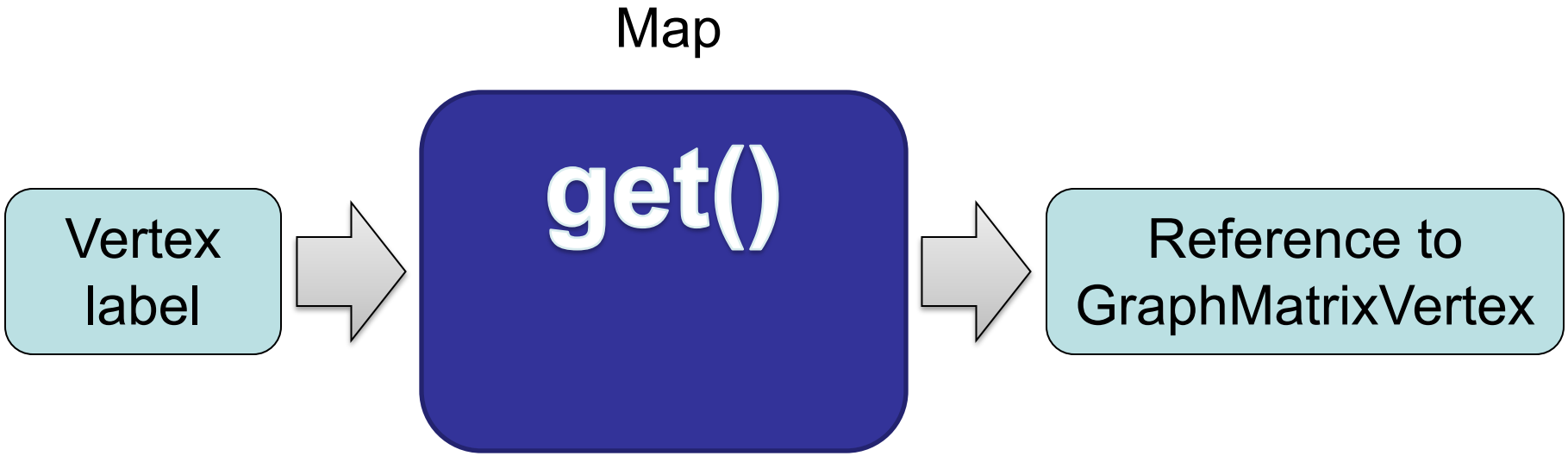
Choosing a Dictionary Structure

- 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

For now:



A map's get(V label) method takes a Vertex label as input, and outputs the corresponding GraphMatrixVertex reference in O(1) time

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 (null if none)
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;
}
```

Why do we get the vertex labels if we already know them?!

- vLabel1 (or 2) may only contain *partial* label information
- Think *Association…*

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)

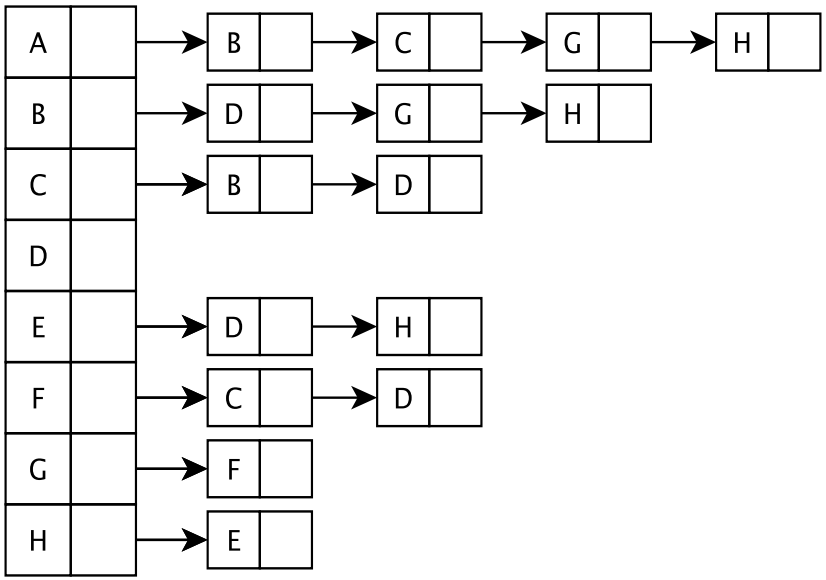
  (For now---even though they are not!)

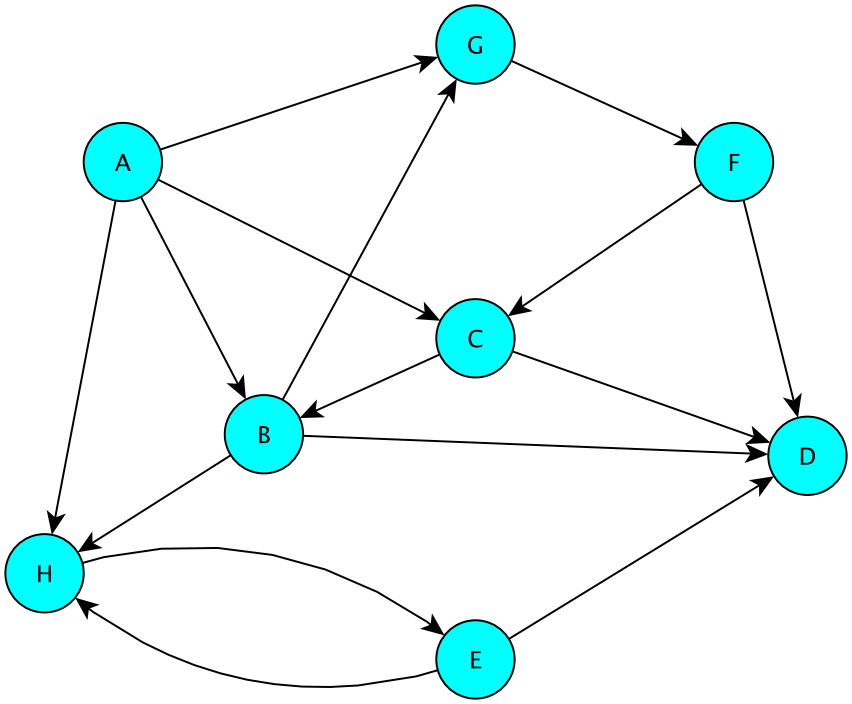
  - |E| = number of edges (often folks write m = |E|)
  - |V| = number of vertices (often folks write n = |V|)
- Runtime of add, addEdge, getEdge, removeEdge, remove?
- Space usage?
- Conclusions
  - Matrix is good for dense graphs
  - Have to commit to maximum # of vertices in advance

Efficiency : Assuming Fast Map Ops

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

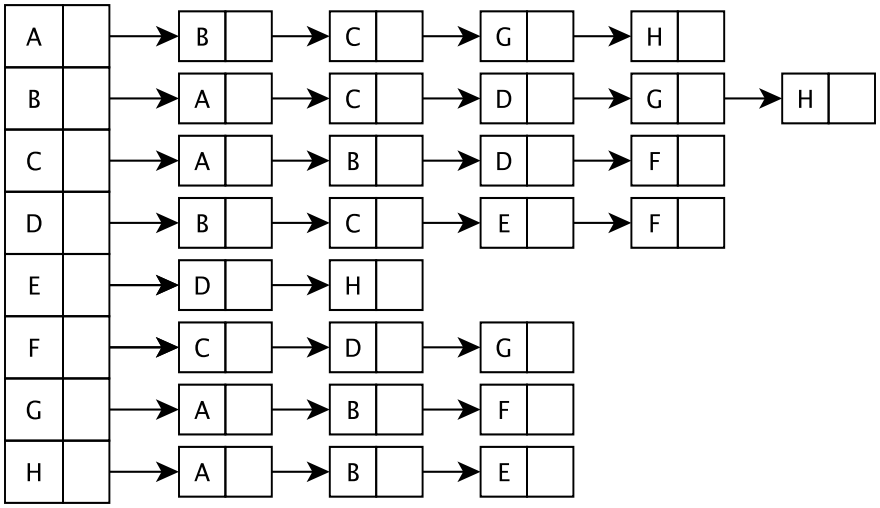
Adjacency List : Directed Graph

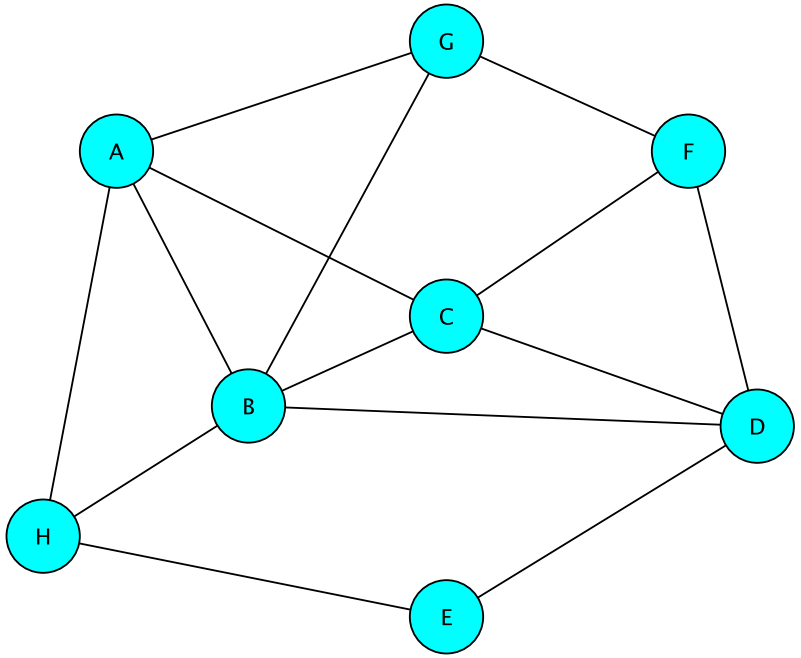




The vertices are stored in an array V[]
V[] contains a linked list of edges having a given source

Adjacency List : Undirected Graph





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

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();
}
```

Efficiency Revisited

- 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 better for dense graphs
  - List is better for sparse graphs
  - For graphs "in the middle" there is no clear winner

Efficiency : Assuming Fast Map

	Matrix	GraphList
add	O(1)	O(1)
addEdge	O(1)	O(1)
getEdge	O(1)	O(|V|)
removeEdge	O(1)	O(|V|)
remove	O(|V|)	O(|E|)
space	$O(\|V\|^2)$	O(|V|+|E|)