Multidimensional Arrays & Graphs

CMSC 420: Lecture 3

Mini-Review

- **Abstract Data Types:**
  - *List*
  - *Stack*
  - *Queue*
  - *Deque*
  - *Dictionary*
  - *Set*

- **Implementations:**
  - Linked Lists
  - Circularly linked lists
  - Doubly linked lists
  - XOR Doubly linked lists
  - Ring buffers
  - Double stacks
  - Bit vectors

Techniques: Sentinels, Zig-zag scan, link inversion, bit twiddling, self-organizing lists, *constant-time initialization*

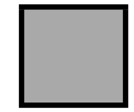
Constant-Time Initialization

- Design problem:
  - Suppose you have a long array, most values are 0.
  - Want constant time access and update
  - Have as much space as you need.

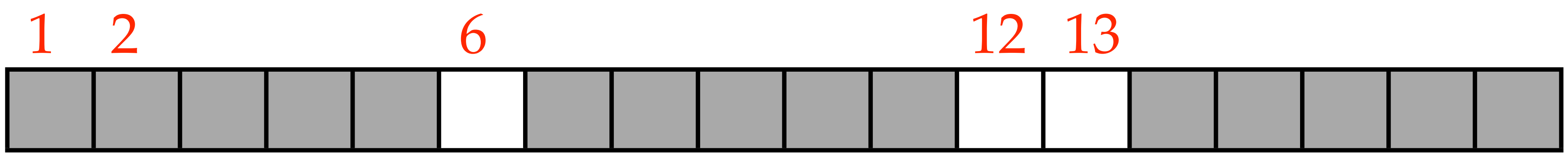
- Create a big array:
  - a = **new int**[LARGE_N];
  - **Too slow: for**(i=0; i < LARGE_N; i++) a[i] = 0

- Want to somehow *implicitly* initialize all values to 0 in constant time...

Constant-Time Initialization

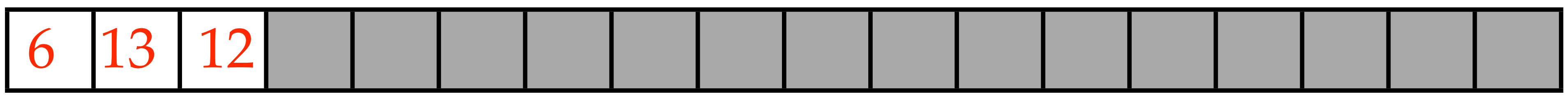
 means **unchanged**

Data[] =



- Access(i): if (0≤ When[i] < count and Where[When[i]] == i) return

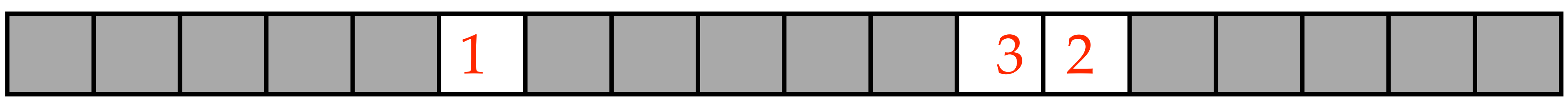
Where[] =



Count = 3

Count holds # of elements changed
Where holds indices of the changed elements.

When[] =



When maps from index *i* to the time step when item *i* was first changed.

```
Access(i):
    if 0 ≤ When[i] < Count and Where[When[i]] == i:
        return Data[i]
    else:
        return DEFAULT
```

Multidimensional Arrays

- Often it’s more natural to index data items by keys that have several dimensions. E.g.:
  - (longitude, latitude)
  - (row, column) of a matrix
  - (x,y,z) point in 3d space

- Aside: why is a plane “2-dimensional”?

Row-major vs. Column-major order

- 2-dimensional arrays can be mapped to linear memory in two ways:

	1	2	3	4	5
1	1	2	3	4	5
2	6	7	8	9	10
3	11	12	13	14	15
4	16	17	18	19	20

Row-major order

	1	2	3	4	5
1	1	5	9	13	17
2	2	6	10	14	18
3	3	7	11	15	19
4	4	8	12	16	20

Column-major order

$\mathrm{Addr}(i,j) = \mathrm{Base} + 5(i\text{-}1) + (j\text{-}1)$

$\mathrm{Addr}(i,j) = \mathrm{Base} + (i\text{-}1) + 4(j\text{-}1)$

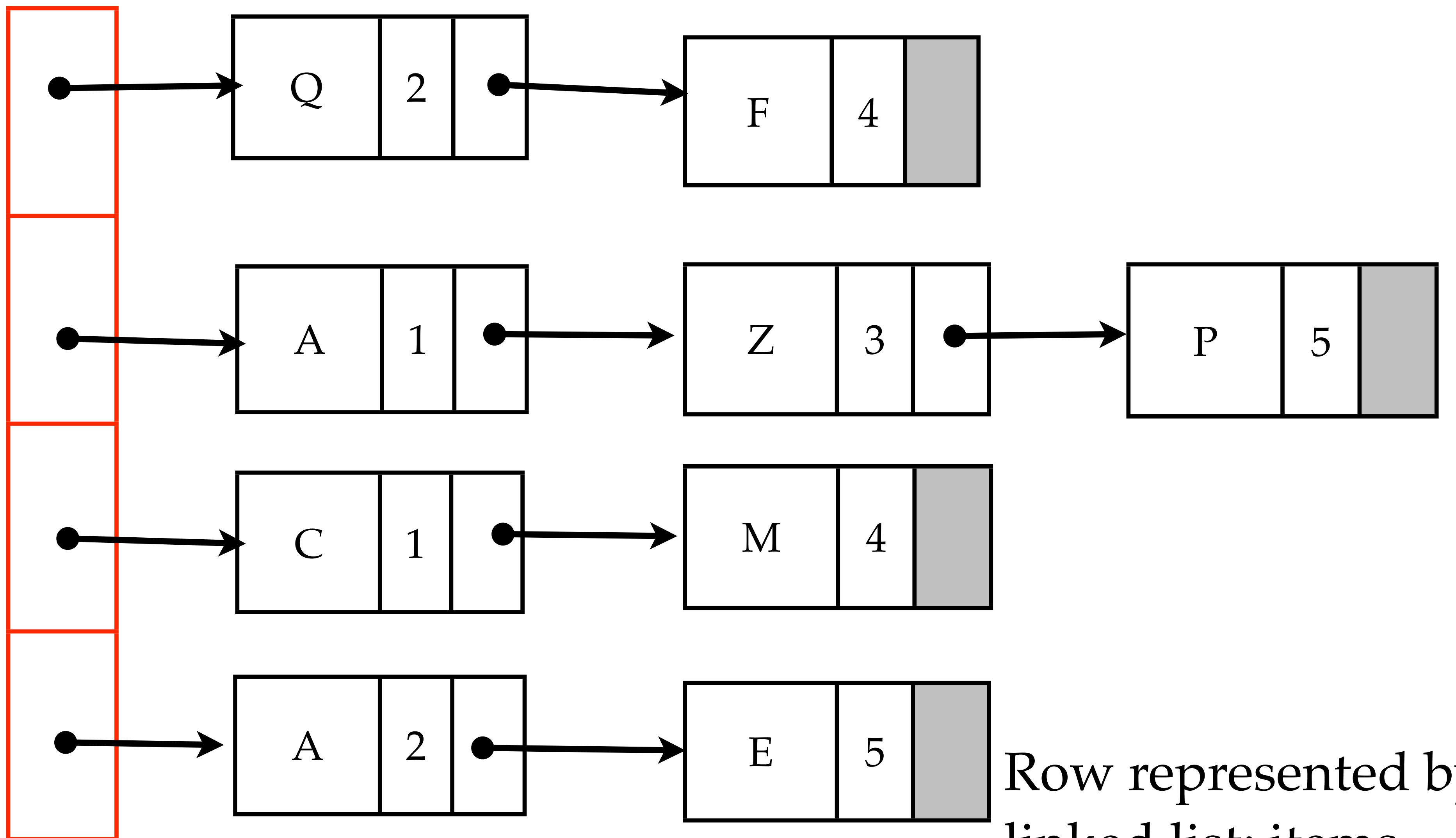
Row-major vs. Column-major order

- Generalizes to more than 2 dimensions
- Think of indices $<i_1, i_2, i_3, i_4, i_5,\ldots,i_d>$ as an odometer.
  - *Row-major order:* last index varies fastest
  - *Column-major order:* first index varies fastest

Sparse Matrices

- Sometimes many matrix elements are either uninteresting or all equal to the same value.
- Would like to *implicitly* store these items, rather than using memory for them.

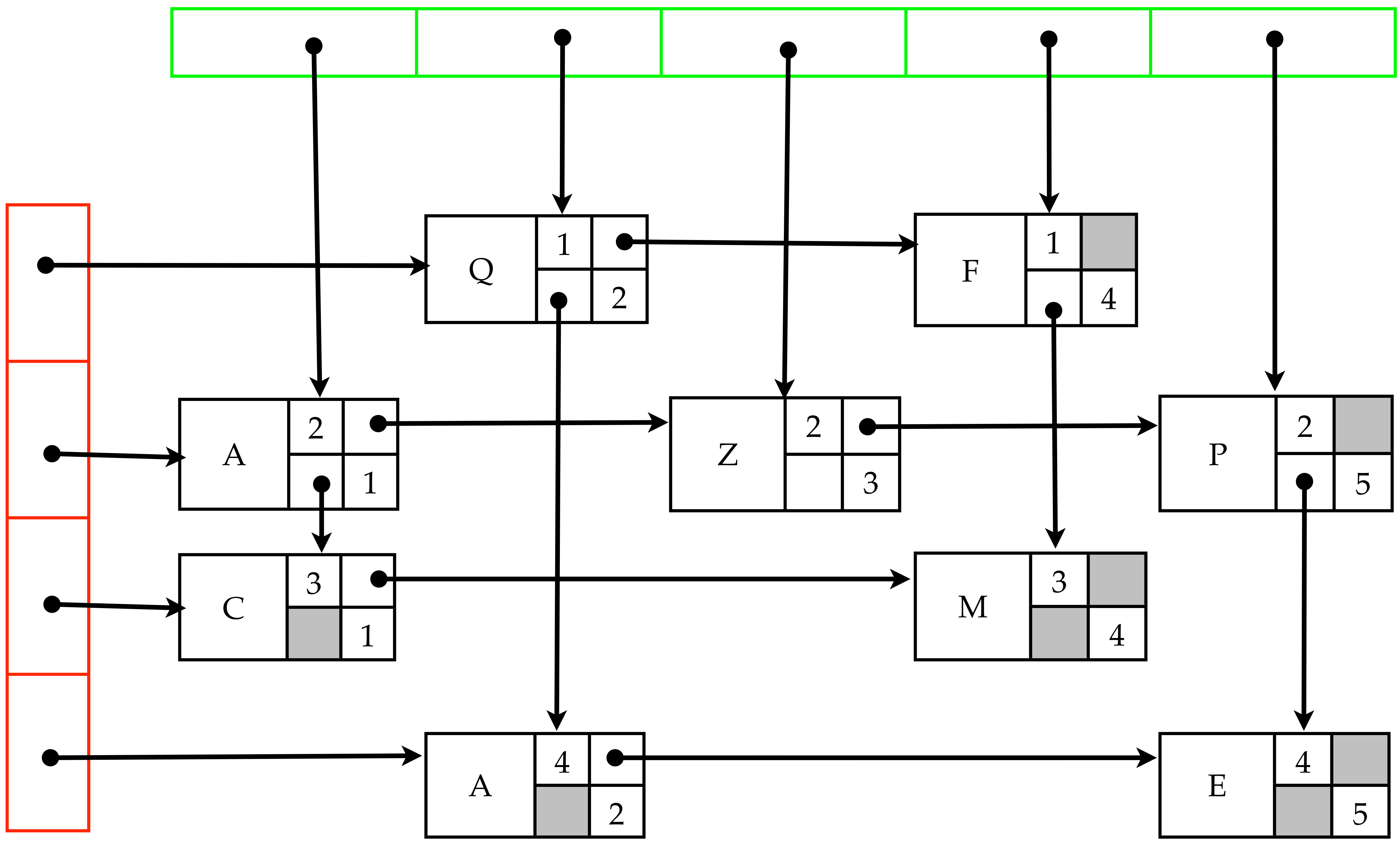
Linked 2-d Array Allocation



Array with random access to linked list representing rows

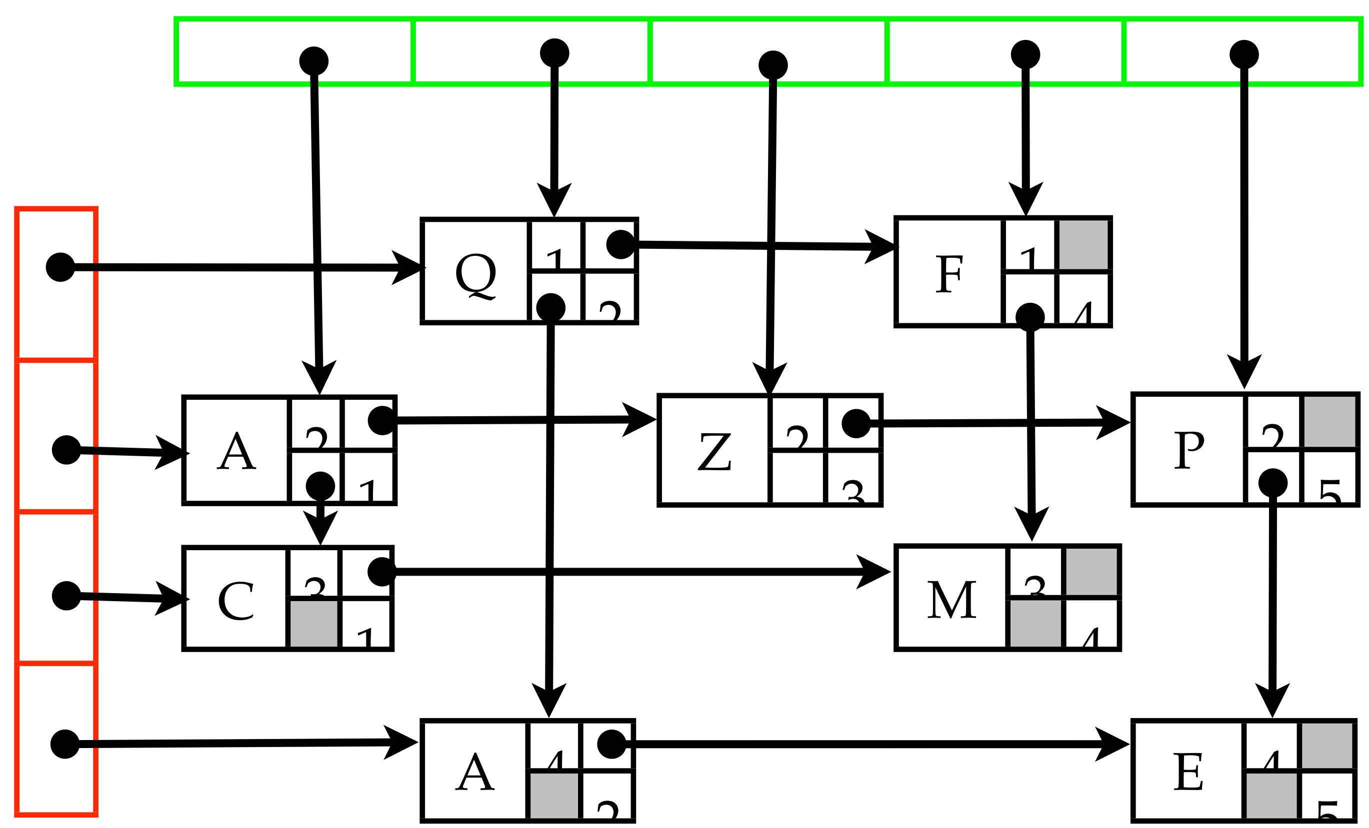
Row represented by linked list; items contain column numbers

Linked 2-d Array Allocation

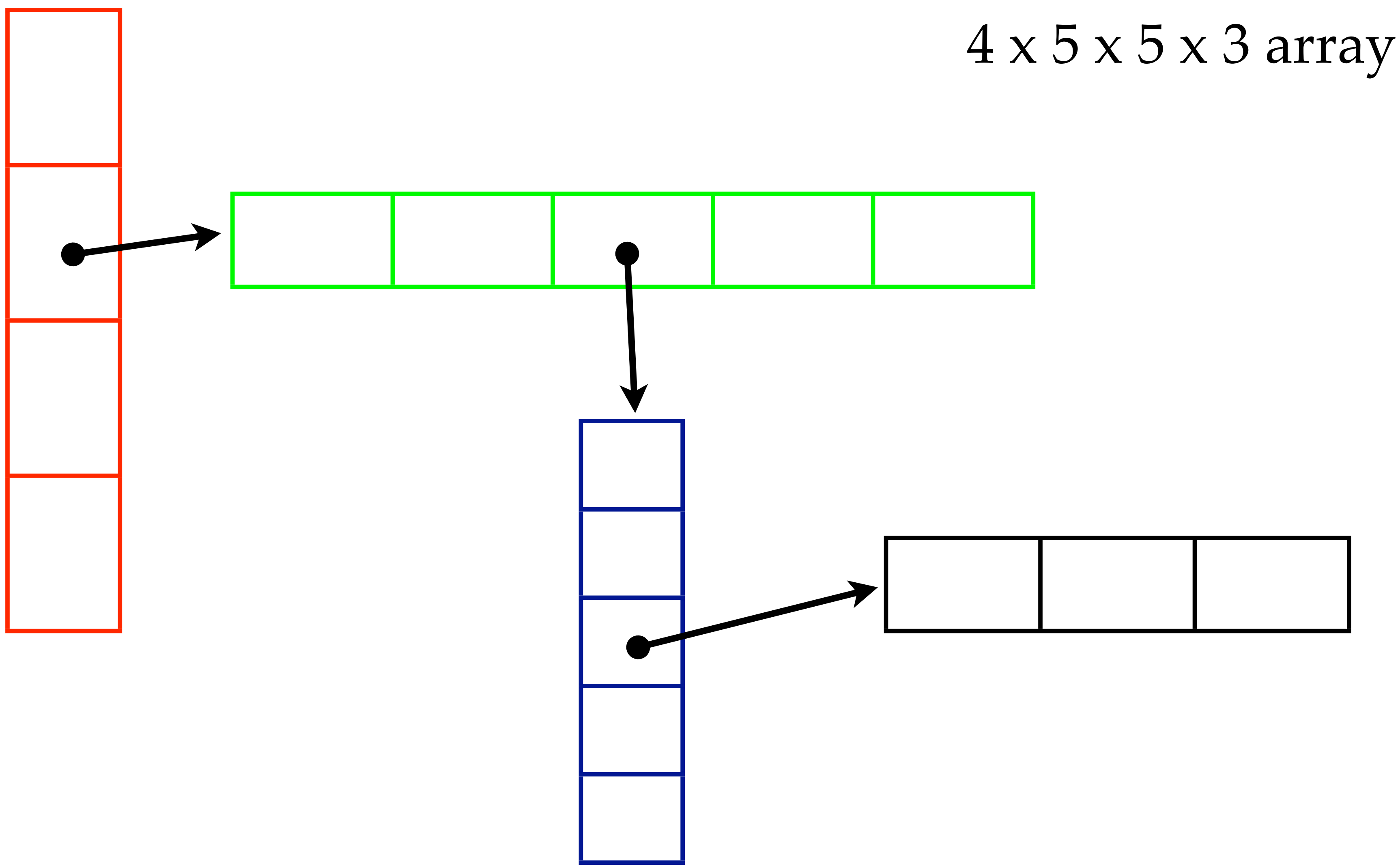


Threading

- Column pointers allow iteration through items with same column index.
- Example of *threading*: adding additional pointers to make iteration faster.
- Threading useful when the definition of "next" depends on context.
- We'll see additional examples of threading with trees.



Hierarchical Tables



- Combination of sequential and linked allocation.
- Particularly useful when filled elements cluster together, or when all entries in one dimension are always known.
- Natural to implement by combining Perl arrays, C++ `vectors`, etc.

Upper Triangular Matrices

- Sometimes "empty" elements are arranged in a pattern.
- Example: symmetric distance matrix.
- Want to store in contiguous memory.
- How do you access item *i, j*?

elements taken up by the first (i-1) rows:

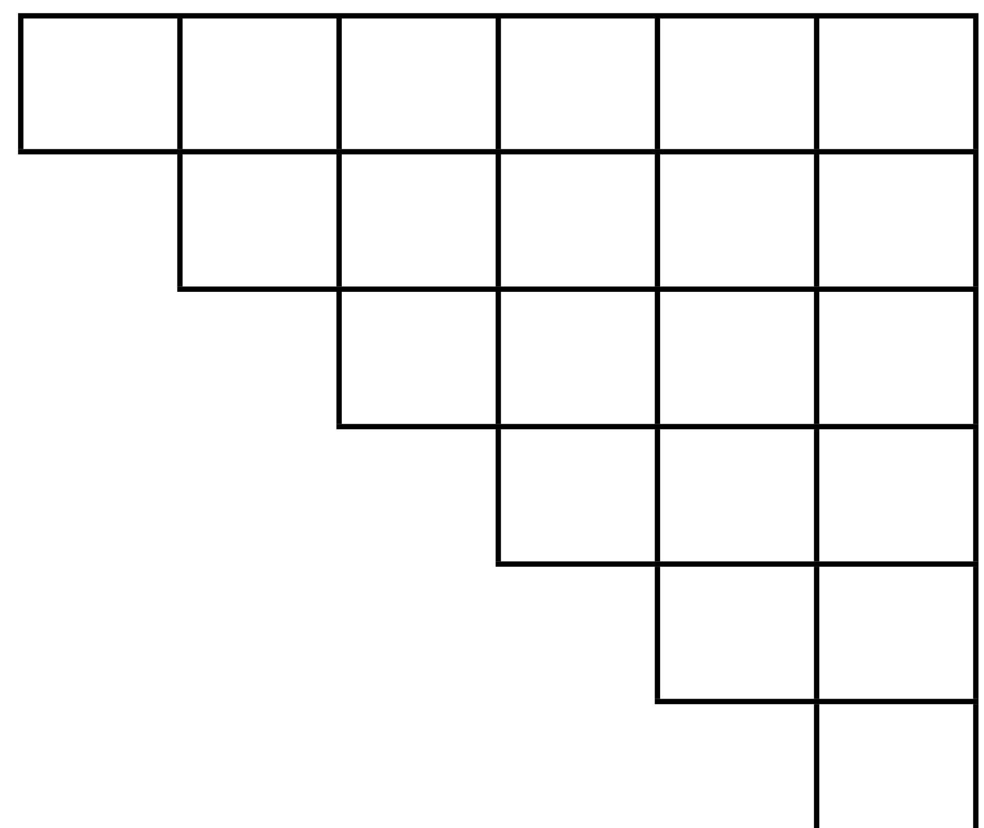
$$n + (n-1) + (n-2) + \ldots + (n - i + 1)$$

$$= \sum_{k=1}^{n} k - \sum_{k=1}^{n-i} k$$

$$= \frac{n(n+1)}{2} - \frac{(n-i)(n-i+1)}{2}$$

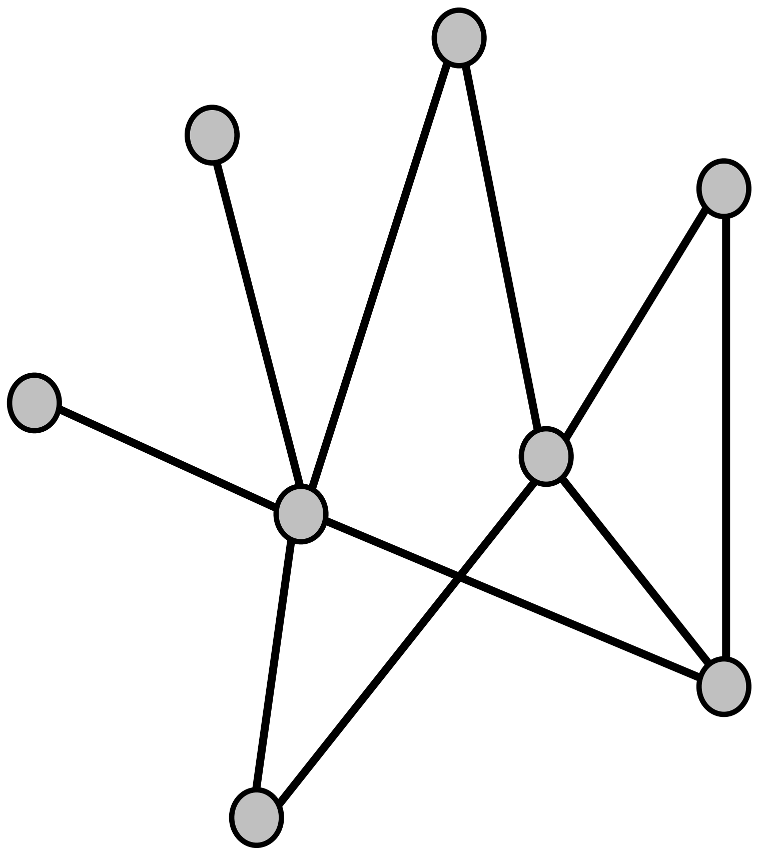
$$= ni + \tfrac{i-i^2}{2}$$

plus j-i come before the j^th element in the i^{th} row



Graphs – Examples

- Computer Networks
- Street map connecting cities
- Airline routes.
- Dependencies between jobs (must finish A before starting B)
- Protein interactions



Used to represent *relationships* between pairs of objects.

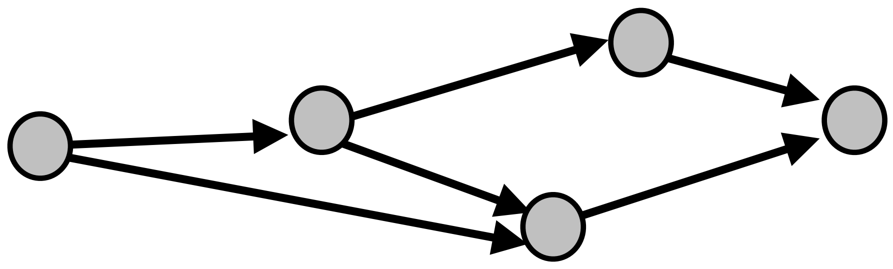
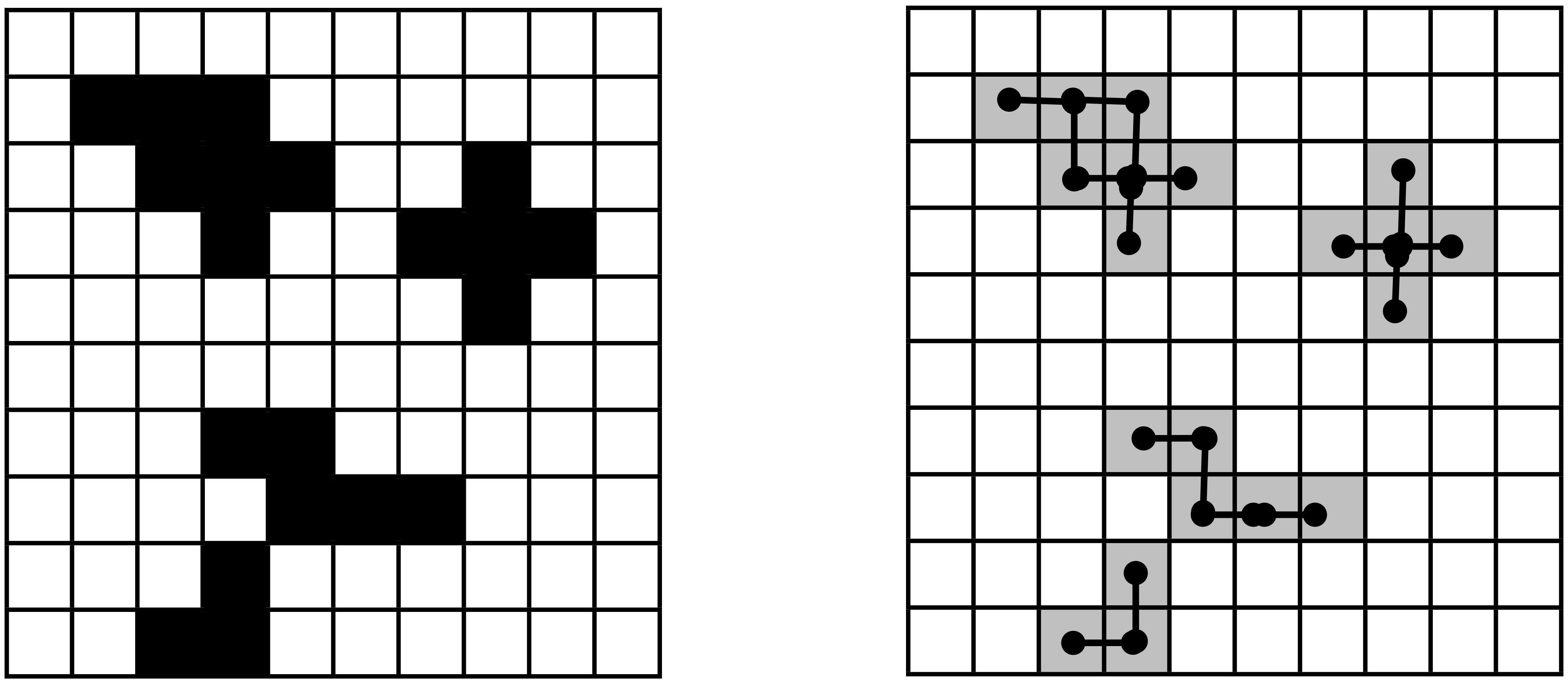


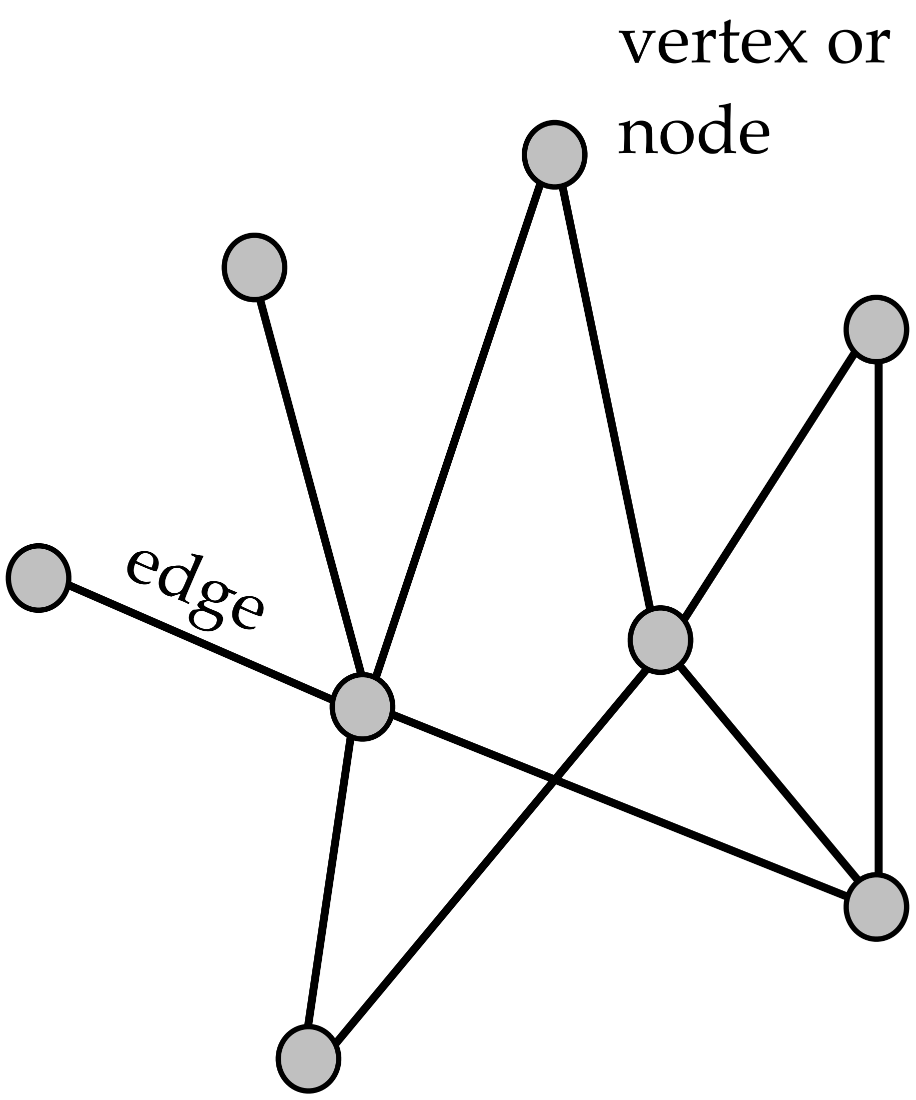
Image Graphs



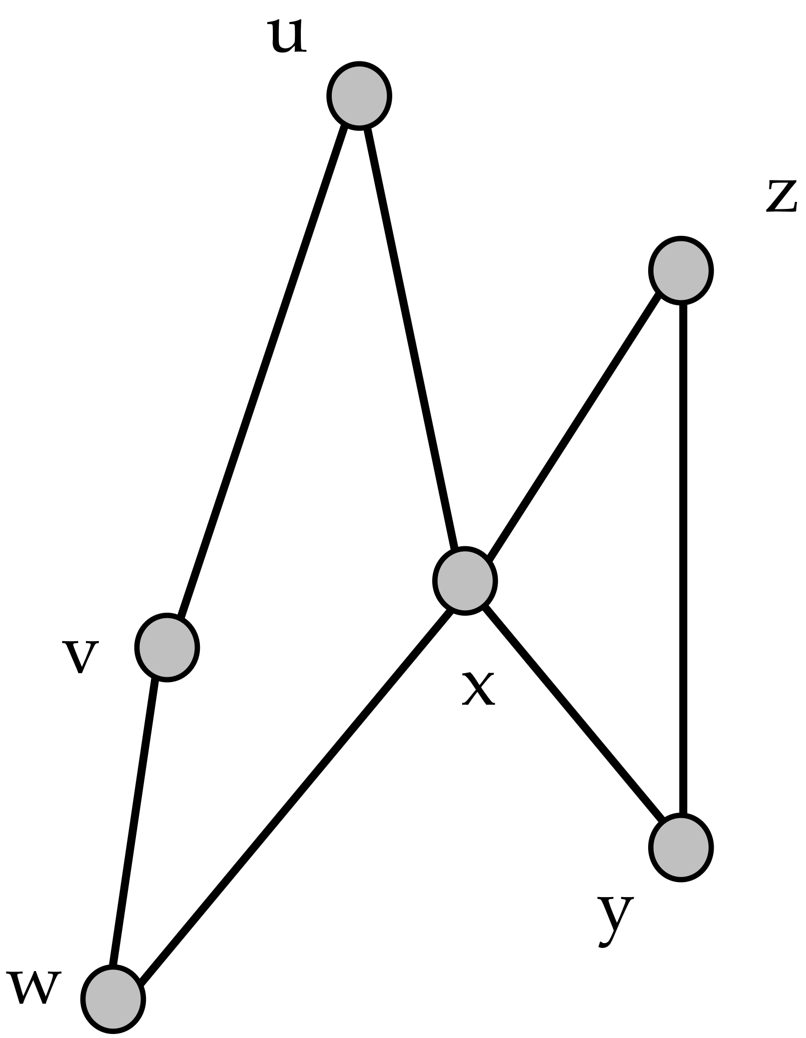
- Black & white image, 0/1 pixels (crossword puzzle, e.g.)
- G = (V,E), a set of vertices V and edges E
  - V = {set of pixels}
  - {u,v} in E if pixels u and v are next to each other.
- Separate connected parts of the graph = disjoint regions of the image (space fill, e.g.)
- Graph defined this way is *planar* (can be drawn without edge crossings).

Graphs – Terminology

- Graph G = (E, V)
  - V = set of vertices
  - E = set of pairs of vertices, represents edges
- *Degree* of vertex = # of edges adjacent to it
- If there is an edge $\{u,v\}$ then u is adjacent to v.
- Edge is *incident* to its *endpoints*.
- *Directed* graph = edges are arrows
  - *out-degree, in-degree*
- The set of vertices adjacent to a node u is called its *neighbors*.



Graphs – Example



- V = {u,v,w,x,y,z}
- E = { {u,v}, {v,w}, {u,x}, {w,x}, {z,y}, {x,y}}

Graphs – More Terminology

- A *path* is a sequence of vertices u_1, u_2, u_3, ... such that each edge (u_i, u_{i+1}) is present.
- A path is *simple* if each of the u_i is distinct.
- A *subgraph* of G = (V, E) is a graph H = (V′, E′) such that V′ is a subset of V and an edge (u,v) is in E′ iff (u,v) is in E and u and v are in V′.
- A graph is *connected* if there is a path connecting every pair of vertices.
- A *connected component* of G is a maximally sized, connected subgraph of G.

Graphs – Still More Terminology

- A *cycle* is a path $u_1, u_2, u_3, \ldots, u_k$ such that $u_1 = u_k$.
- A graph without any cycles is called *acyclic*.
- An undirected acyclic graph is called a *free tree* (or usually just a *tree*)
- A **directed** acyclic graph is called a DAG (for "Directed Acyclic Graph")
- *Weighted* graph means that either vertices or edges (or both) have weights associated with them.
- *Labeled* graph = nodes are labeled.

Graphs – Basic properties

- **Undirected graphs:**
  - What's the maximum number of edges?
    (A graph that contains all possible edges is called *complete*)
  - What's the sum of the all the degrees?

- **Directed graphs:**
  - What's the maximum number of edges?
  - What's the sum of all the degrees?

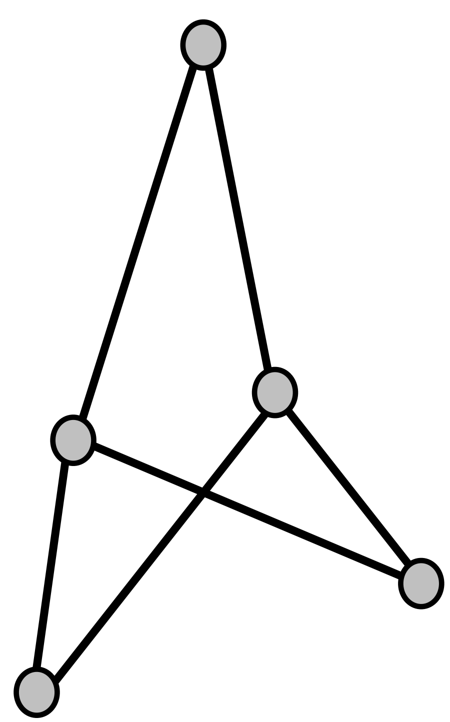
Graphs – Isomorphism

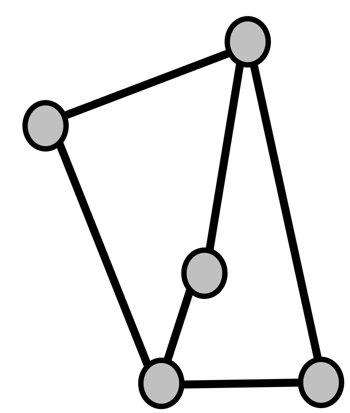
- Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if

  there's a 1-to-1 and onto mapping $f(v)$ between V_1 and V_2 such that:

  $\{u,v\}$ in E_1 iff $\{f(u), f(v)\}$ in E_2.

- In other words, G_1 and G_2 represent the same *topology*.





Does checking whether two graphs are isomorphic seem like an easy problem or a hard problem?

Graphs – ADT

- S = *vertices*()
- S = *edges*()
- *neighbors*(G, v)
- *insert_edge*(G, u,v)
- *insert_vertex*(G, u)
- *remove_edge*(G, u,v)
- *remove_vertex*(G, u)

Return *sets* - set ADT we talked about last time may be useful

Time to perform these tasks will depend on implementation.

What are ways to implement graphs?

Graphs – Implementations

1. List of edges
2. Adjacency matrix
3. Adjacency list

Edge List Representation

- Simple: store edges (aka vertex pairs) in a list.
- **Good if:** the "structure" of the graph is not needed, and iterating through all the edges is the common operation.
- **Bad because:**
  - testing whether an edge is present may take $O(|E|)$.
  - Relationships between edges are not evident from the list (hard to do shortest path, etc.).

Adjacency Matrix

2-dimensional matrix: 1 in entry (u,v) if edge (u,v) is present; 0 otherwise

What's special about the adjacency matrix for an *undirected* graph?

What kind of adjacency matrix makes sense for undirected graphs?

	1	2	3	4	5	6	7
1	1						
2			1				1
3					1		
4			1				
5							
6					1		
7		1					

Undirected Adjacency Matrix

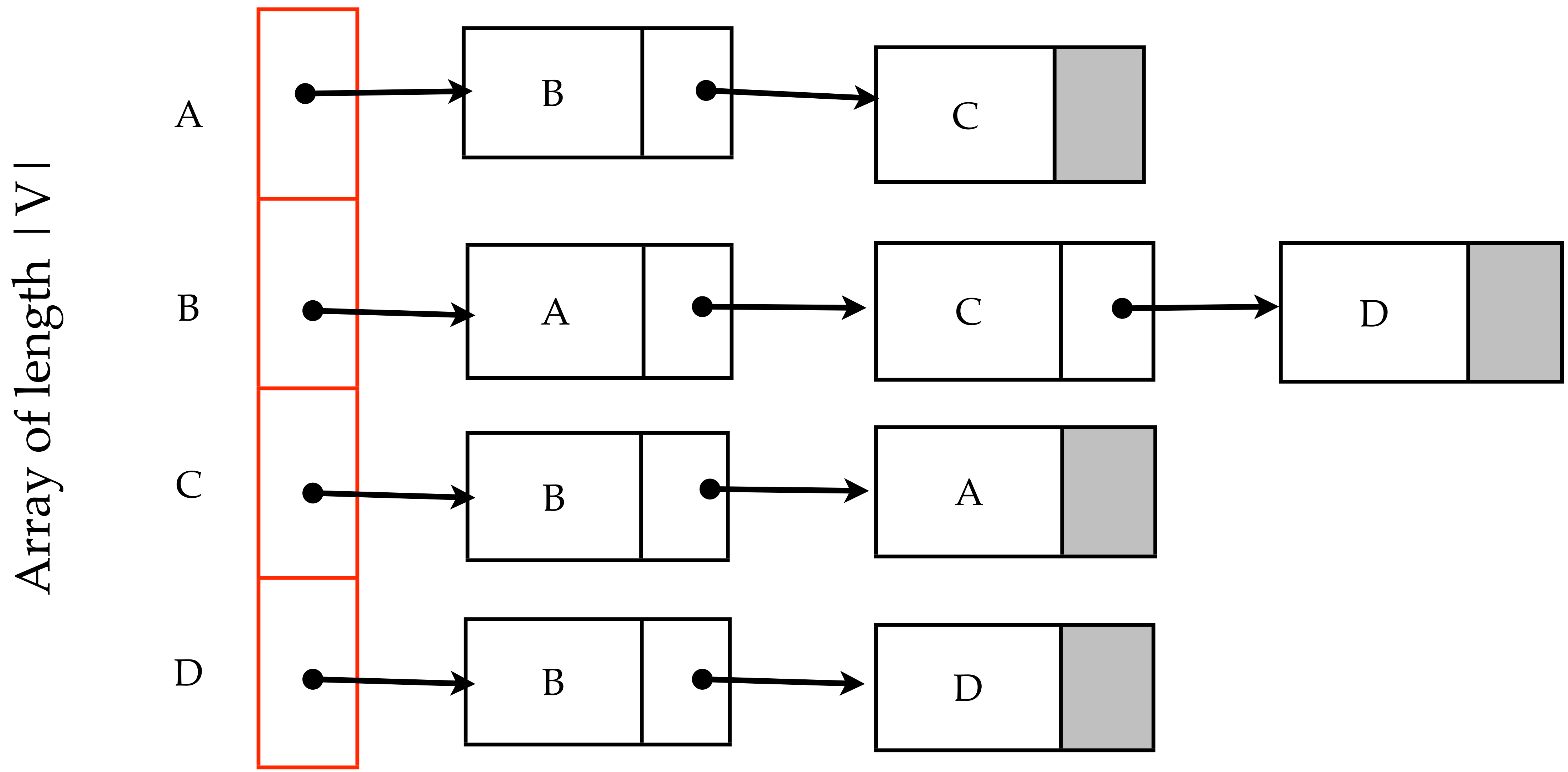
- Undirected graph = symmetric adjacency matrix because edge {u,v} is the same as edge {v, u}.

- Can use upper triangular matrix we discussed above.

- Weights on the edges can be represented by numbers in the matrix (as long as there is some “out of band” number to mean “no edge present”)

- What if most edges are absent? Say $|E| = O(|V|)$. Graph is *sparse*.

Adjacency Lists



In an undirected graph, each edge is stored twice (each edge is adjacent to two vertices)

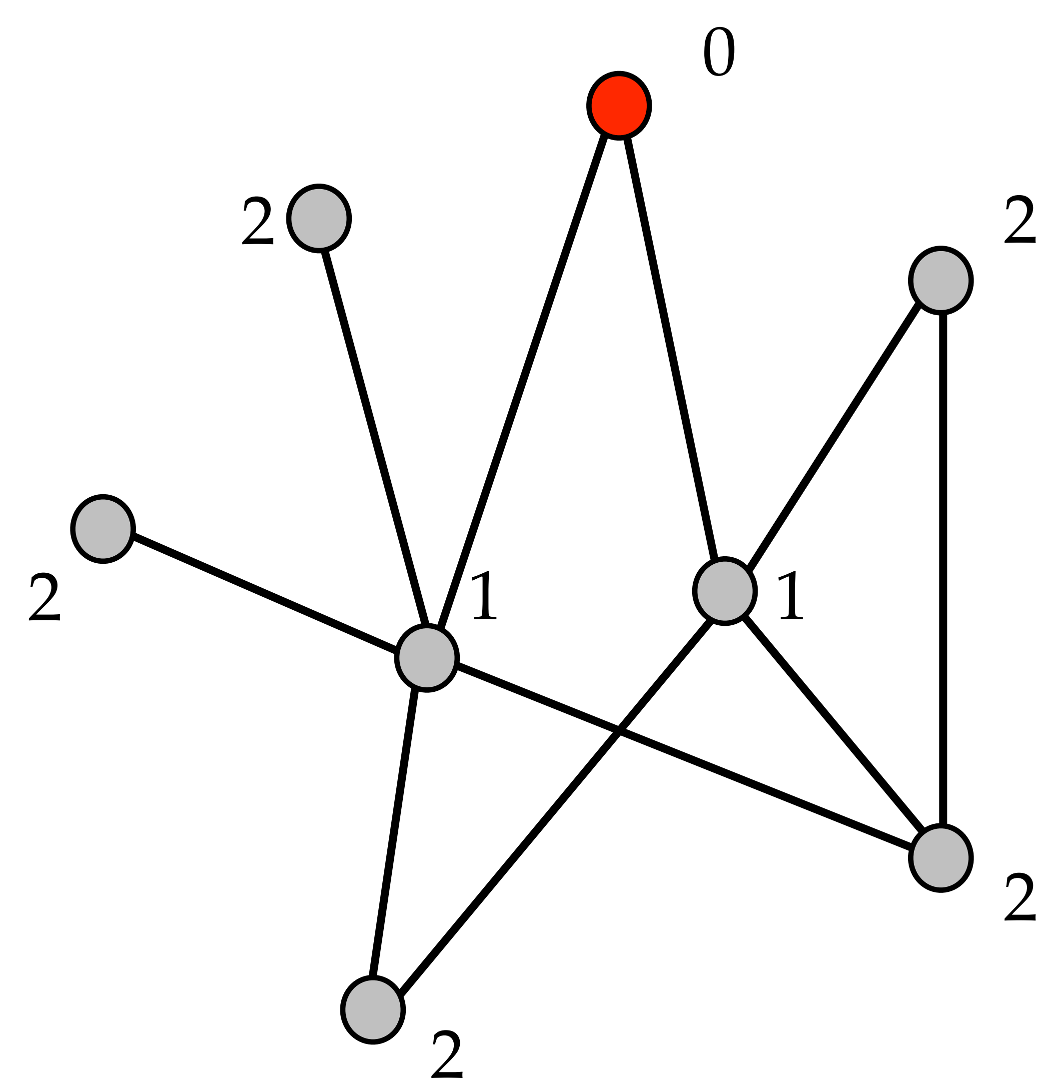
Adjacency MATRIX vs. Adjacency LISTS

- **Matrix**:
  - No pointer overhead
  - More space efficient if G is dense
  - Neighbor() operation is slow! O(n)

- **List**:
  - More space efficient if G is sparse
  - Neighbor() operation proportional to the degree.
  - Asymptotic running times often faster

Breadth-First Search

- Visit the nodes of a graph, starting at a given node v.
- We visit the vertices in increasing order according to their distance from u.
- I.e. we visit v, then v's neighbors, then their neighbors, ...
- If G is connected, we'll eventually visit all nodes.



Numbers indicate the shortest distance from v (minimum # of edges you must traverse to get from v to the node).

Breadth-First Search

```
BFS(G, u):
    mark each vertex unvisited
    Q = new Queue
    enqueue(Q, u)

    while not empty(Q):
        w = dequeue(Q)
        if w is unvisited:
            VISIT(w)
            mark w as visited
            for v in Neighbors(G, w):
                enqueue(Q, v)
```

Initially, every vertex is "unvisited"

Q maintains a queue of vertices that we've seen but not yet processed.

While there are vertices that we've seen but not processed...

Process one of them

and add its unseen neighbors to the queue and mark them seen.

Why a queue?

Breadth-First Search – Running time

```
BFS(G, u):
    mark each vertex unvisited
    Q = new Queue
    enqueue(Q, u)



    while not empty(Q):
        w = dequeue(Q)
        if w is unvisited:
            VISIT(w)
            mark w as visited
            for v in Neighbors(G, w):
                enqueue(Q, v)
```

If G is represented by adjacency LIST, then BFS takes time $O(|V| + |E|)$:

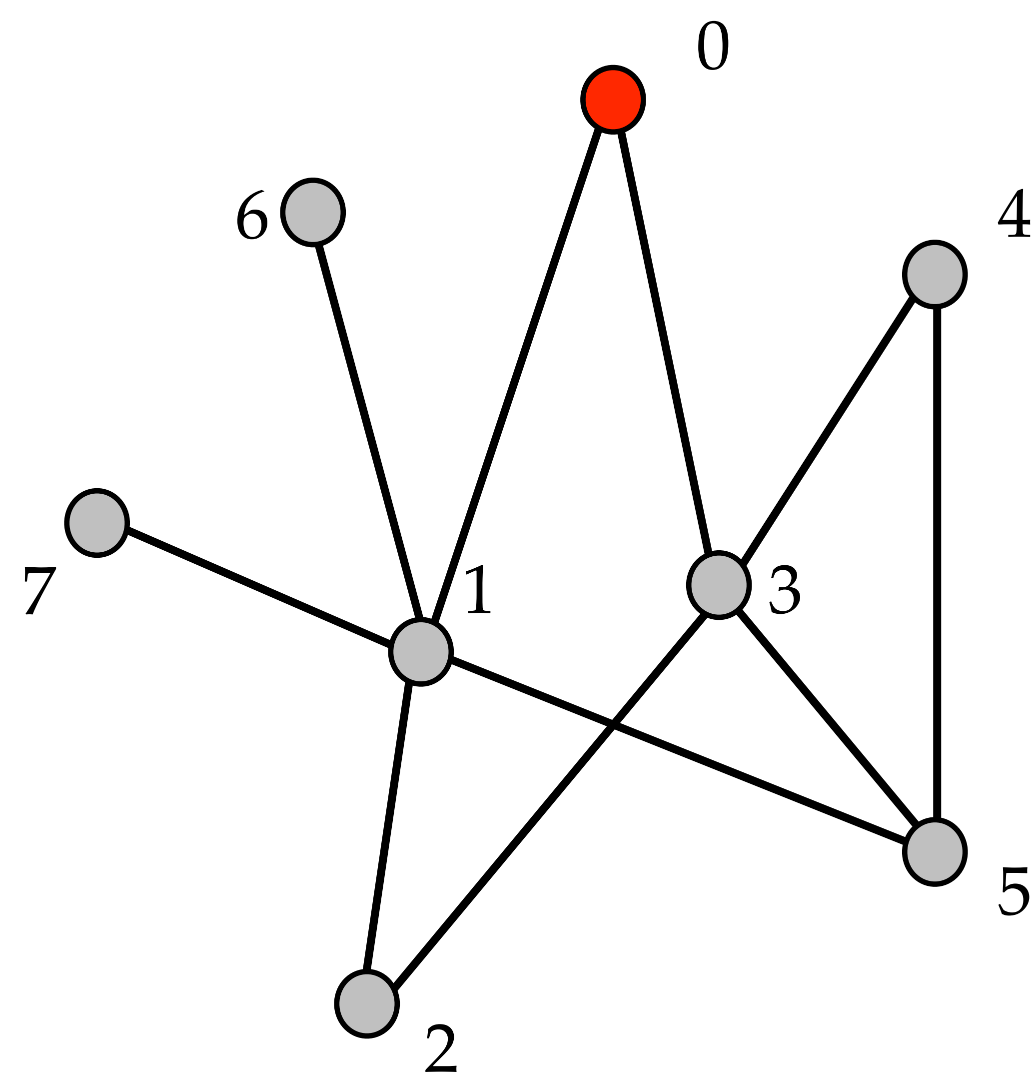
$|V|$ because you need to visit each node at least once to mark them unseen

$|E|$ because each edge is considered at most twice.

What if G is represented by adjacency MATRIX?

Depth-First Search

- Visit the nodes of a graph, starting at a given node v.
- Immediately after visiting a node u, visit its neighbors.
- I.e. we walk as far as we can, and only then "backtrack"
- If G is connected, we'll eventually visit all nodes.



Numbers indicate a possible sequence of visits.

Depth-First Search

```
DFS(G, u):
    mark each vertex unvisited
    S = new Stack
    push(S, u)

    while not empty(S):
        w = pop(S)
        if w is unvisited:
            VISIT(w)
            mark w as visited
            for v in Neighbors(G, w):
                    push(S, v)
```

Initially, everything vertex is "unvisited"

Q maintains a **stack** of vertices that we've seen but not yet processed.

Using a stack means that we'll move to one of the neighbors immediately after seeing them.

Depth-First Search vs. Breadth-First Search

```
DFS(G, u):
    mark each vertex unvisited
    S = new Stack
    push(S, u)

    while not empty(S):
        w = pop(S)
        if w is unvisited:
            VISIT(w)
            mark w as visited
            for v in Neighbors(G, w):
                    push(S, v)
```

```
BFS(G, u):
    mark each vertex unvisited
    Q = new Queue
    enqueue(Q, u)

    while not empty(Q):
        w = dequeue(Q)
        if w is unvisited:
            VISIT(w)
            mark w as visited
            for v in Neighbors(G, w):
                enqueue(Q, v)
```

Recursive DFS

```
Recursive_DFS(G, u):
    ProcessOnEnter(u)
    mark u visited
    for w in Neighbors(u):
        if w is unvisited:
            DFS(G, w)
   ProcessOnExit(u)
```

What if G is not connected?

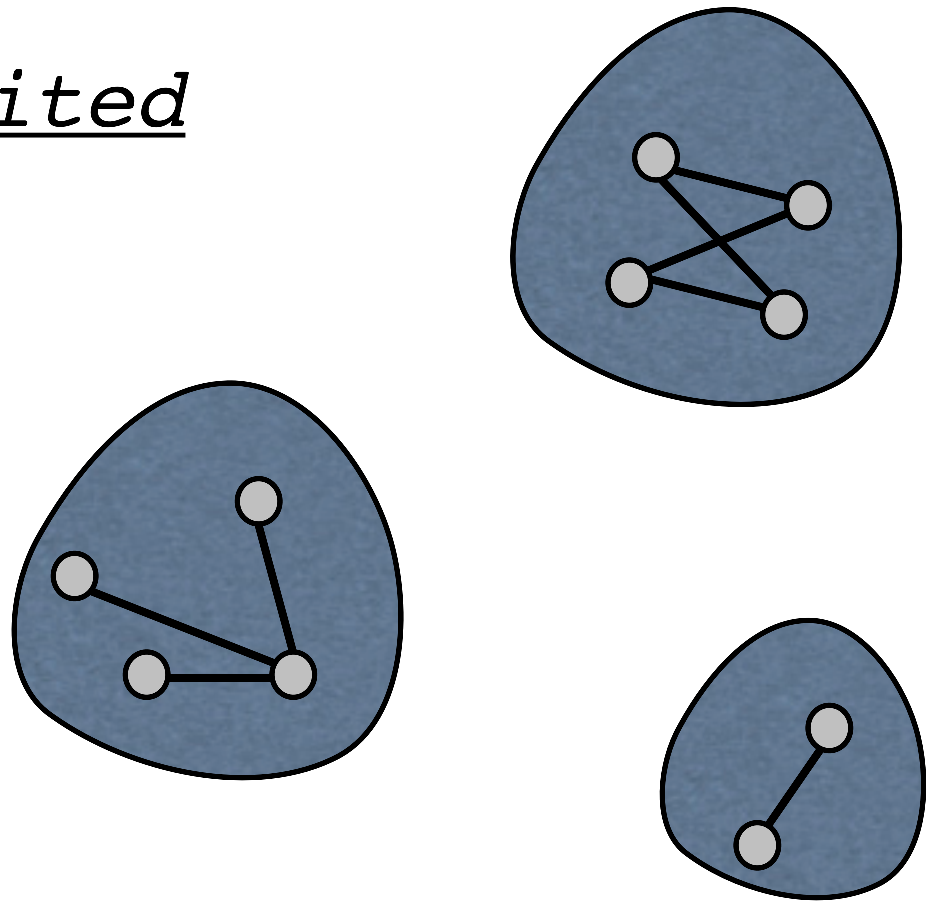
```
Traverse(G):
    mark all vertices as unvisited
    for u in Vertices(G):
        if u is unvisited:
            DFS(G, u)
```

Can use BFS search as well

Connected Components

```
Connected_Components(G):
    mark all vertices as unvisited
    cc = 0
    for u in Vertices(G):
        if u is unvisited:
            DFS(G, u, ++cc)
```

DFS (or BFS) will explore all vertices of a component



Connected components: path between every pair of nodes within a component; no path between components.