Lecture 19: Swinging from Up-Trees to Graphs

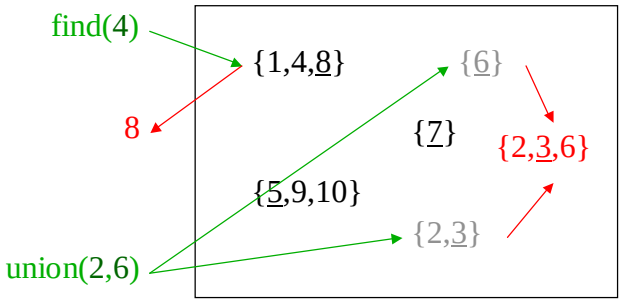
- Today's Agenda:
  - Smart Union and Find
    - Union-by-size/height and Path Compression
  - Run Time Analysis – as tough as it gets!
  - Introduction to Graphs

- Covered in Chapters 8 and 9 in the textbook

Recall: Disjoint Set ADT

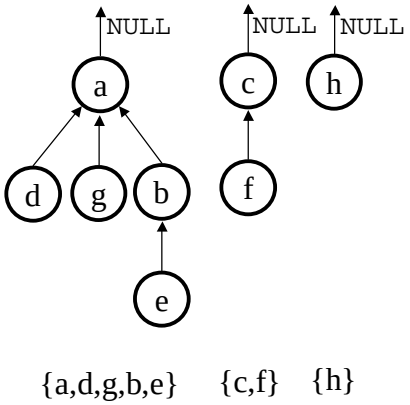
- Disjoint set ADT: Used to represent a collection of sets containing objects that are related to each other
  - Relations defined through Union operation
  - Union merges two sets – their objects become related.
  - Find returns the "name" of the set an object belongs to

Example:
Initial Classes = {1,4,8}, {2,3}, {6}, {7}, {5,9,10}
Name of equiv. class underlined



Recall: Up-Tree Data Structure

- Each equivalence class (or discrete set) is an up-tree with its root as its representative member
- All members of a given set are nodes in that set's up-tree

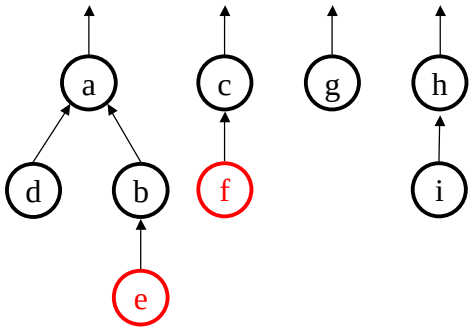


Example of Find

Find: Just follow parent pointers to the root!

find(f) = c
find(e) = a

Runtime depends on tree depth



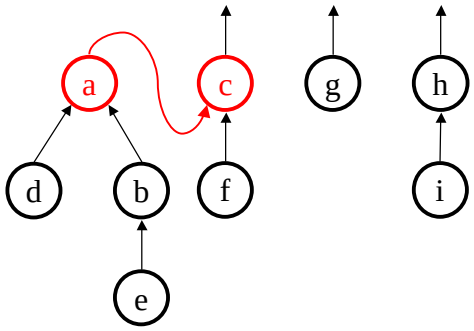
-(Tree size)

0 (a)	1 (b)	2 (c)	3 (d)	4 (e)	5 (f)	6 (g)	7 (h)	8 (i)
Array up: -4	0	-2	0	1	2	-1	-2	7

Example of Union

Union: Just hang one root from the other!

union(c,a)



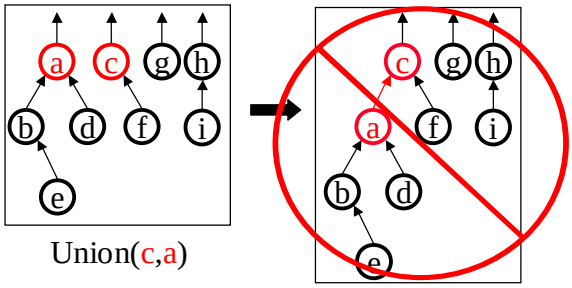
Runtime = O(1)

	0 (a)	1 (b)	2 (c)	3 (d)	4 (e)	5 (f)	6 (g)	7 (h)	8 (i)
Array up:	2	0	-6	0	1	2	-1	-2	7

Change a (from -4) to c (= 2), update size

Smart Union?

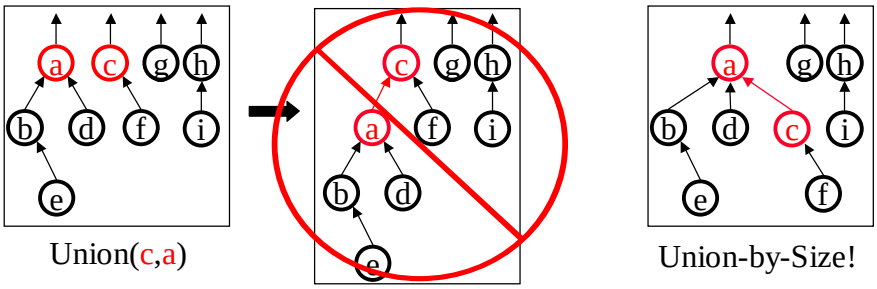
- For M Finds and N-1 Unions, worst case time is O(MN+N)
  - Can we speed things up by being clever about growing our up-trees?



Union(c,a)

Smart Union/Find: Union-by-Size

- Idea: In Union, always make root of *larger* tree the new root
- Why? Minimizes height of the new up-tree



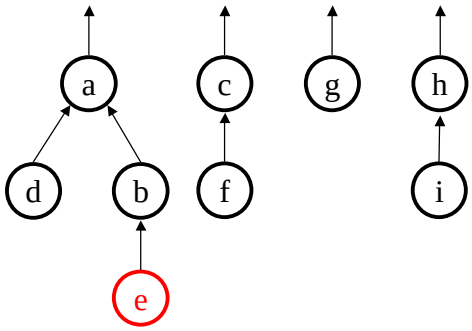
Union-by-Size: Run-Time Analysis

- Finds are O(max up-tree height) for a forest of up-trees containing N nodes
- Number of nodes in an up-tree of height *h* using union-by-size is $\geq 2^h$ (prove by induction)
  - Pick up-tree with max height
  - Then, $N \geq 2^{\text{max height}} \rightarrow$ max height $\leq \log N$
- Find takes **O(log N)** when Union-by-Size is used
  - Same result with Union-by-Height (see text)

Smart Find?

Find(e) = a

Runtime depends on tree depth

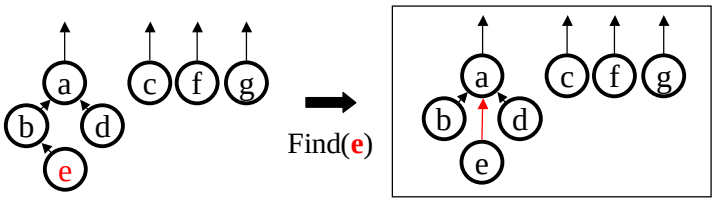


If we do M Finds on the same element → O(M log N) time

Can we make Find have *side-effects* so that next Find will be faster?

Introducing…Path Compression

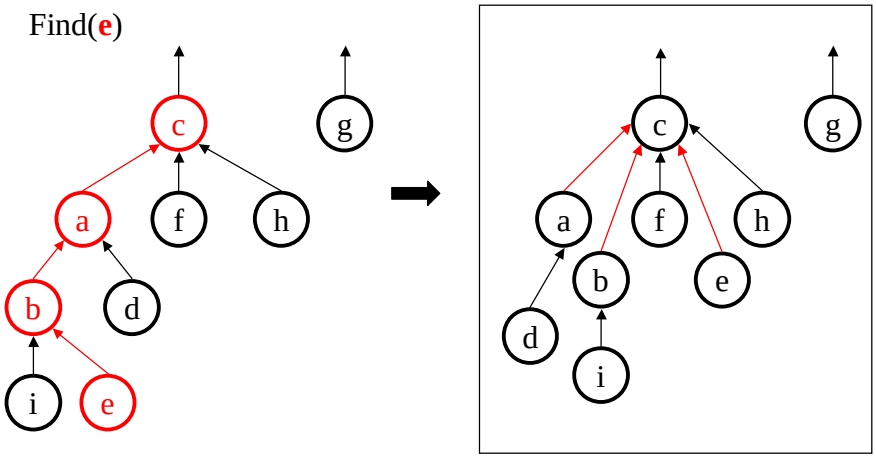
- Path Compression: Point everything along path of a Find to root
- Reduces height of entire access path to 1: Finds get faster!
  - Déjà vu?
  - Idea similar to the one behind your old buddy – the splay tree…



Path compression

A P.C. example with more meat…

Find(**e**)



How to P.C. – Path Compression Code

```
public int Find(int X)
{ // Assumes X = Hash(X_Element)
  // X_Element could be str/char etc.

  if (up[X] < 0) // Root
   return X; //Return root = set name
  else
   //Find parent and update pointer to root
   return up[X] = Find(up[X]);
}
```

Make all nodes along access path point to root

- Trivial modification of original recursive Find
- New running time = ?

How to P.C. – Path Compression Code

```
public int Find(int X)
{ // Assumes X = Hash(X_Element)
  // X_Element could be str/char etc.

  if (up[X] < 0) // Root
   return X; //Return root = set name
  else
   //Find parent and update pointer to root
   return up[X] = Find(up[X]);
}
```

Collapsing the tree by pointing to root

- Find still takes O(max up-tree height) worst case
- But what happens to the tree heights over time?
- What is the *amortized* run time of Find if we do M Finds?

What is the *amortized* run time per operation if we do a sequence of M Unions or Finds using Union-by-Size & P.C.?

10-second break to solve this problem…

What is the *amortized* run time per operation if we do a sequence of M Unions or Finds using Union-by-Size & P.C.?

If you succeeded in solving this…you shouldn't be in this class!

One of the toughest run-time analysis problems ever!

(no, a similar problem won't be in the final)

Fine print: The final has not been written up yet, so we are not responsible for any statements about the final made in this lecture or indeed any future lectures, but not including the final review lecture

Analysis of P.C. with Union-by-Size

- R. E. Tarjan (of the up-trees fame) showed that:
  - When both P.C. and Union-by-Size are used, the worst case run time for a sequence of M operations (Unions or Finds) is $\Theta(M\ \alpha(M,N))$
- What is $\alpha(M,N)$?
  - $\alpha(M,N)$ is the inverse of Ackermann's function
- What is Ackermann's function?

Digression: Them slow-growing functions…

- How fast does log N grow? log N = 4 for N = 16 = 2^{2^2}
  - ➪ Grows quite slowly
- Let $\log^{(k)} N = \log(\log(\log \ldots (\log N)))$ (k logs)
- Let log* N = minimum k such that $\log^{(k)} N \le 1$
- How fast does $\log^* N$ grow? $\log^* N = 4$ for $N = 65536 = 2^{2^{2^2}}$
  - ➪ Grows very slowly

Ackermann and his function

- Ackermann created a really explosive function A(i, j) whose inverse $\alpha(M, N)$ grows very, very slowly (slower than $\log^* N$)

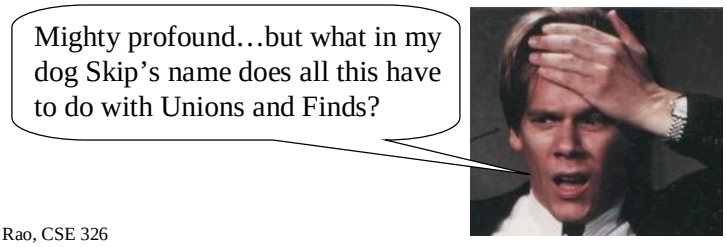
$$A(1, j) = 2^j \text{ for } j \ge 1$$
$$A(i,1) = A(i-1,2) \text{ for } i \ge 2$$
$$A(i, j) = A(i-1, A(i, j-1)) \text{ for } i, j \ge 2$$

$$\alpha(M, N) = \min\{i \ge 1 \mid A(i, \lfloor M / N \rfloor) > \log N\}$$

How slowly does $\alpha(M, N)$ grow?

$\alpha(M, N) \leq 4$ for all practical choices of M and N

($\alpha(M, N) = 4$ for M **far** larger than the number of atoms in the universe (2^{300})!! (assuming $M \geq N$))



Back to Smart Unions/Finds

- R. E. Tarjan showed that:
  - When both P.C. and Union-by-Size are used, the worst case total run time for any sequence of M Unions and Finds is $\Theta(M \cdot \alpha(M,N))$
- Textbook proves weaker result of O(M log* N) time
  - Requires 6 pages and 8 Lemmas! (Check it out!)
- Amortized run time per operation
  = total time/no. of operations = $\Theta(M \cdot \alpha(M,N))/M$
  = $\Theta(\alpha(M,N))$
  ≈ $\Theta(1)$ for all practical purposes ($\alpha(M, N) \leq 4$ for all practical M, N)
  ≈ constant time!

Summary of Disjoint Set and Union/Find

- The Disjoint Set ADT allows us to represent objects that fall into different equivalence classes or sets
- Two main operations: Union of two classes and Find class name for a given element
- Up-Tree data structure allows efficient array implementation
  - Unions take O(1) worst case time, Finds can take O(N)
  - Union-by-Size (or by-Height) reduces worst case time for Find to O(log N)
  - If we use both Union-by-Size/Height & Path Compression:
    - Any sequence of M Union/Find operations results in O(1) amortized time per operation (for all practical purposes)

Applications of Disjoint Sets

- Disjoint sets can be used to represent:
  - Cities on a map (disjoint sets of connected cities)
  - Electrical components on chip
  - Computers connected in a network
  - Groups of people related to each other by blood
  - Textbook example: Maze generation using Unions/Finds:
    - Start with walls everywhere and each cell in a set by itself
    - Knock down walls randomly and Union cells that become connected
    - Use Find to find out if two cells are already connected
    - Terminate when starting and ending cell are in same set i.e. connected (or when all cells are in same set)

We are now ready to tackle
the grandmama of all data structures…

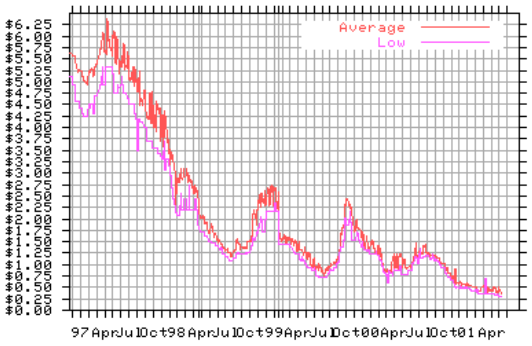
The crème de la crème…

The most general, the all-encompassing…

Graphs and their algorithms!

What are graphs? (Take 1)

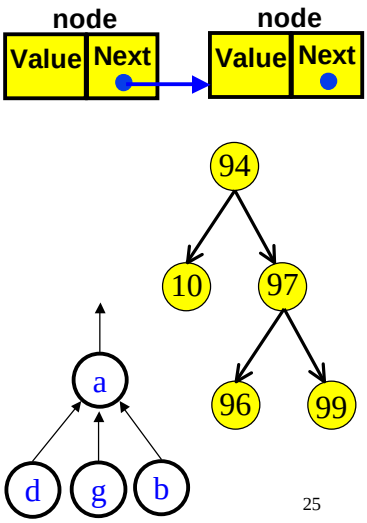
- Yes, this is a graph….



- But we are interested in a different kind of “graph”

Motivation for Graphs

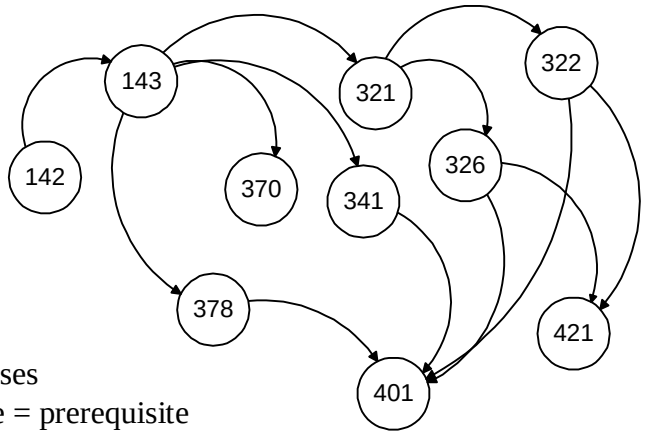
- Consider the data structures we have looked at so far…
- Linked list: nodes with 1 incoming edge + 1 outgoing edge
- Binary trees/heaps: nodes with 1 incoming edge + 2 outgoing edges
- Binomial trees/B-trees: nodes with 1 incoming edge + multiple outgoing edges
- Up-trees: nodes with multiple incoming edges + 1 outgoing edge



Motivation for Graphs

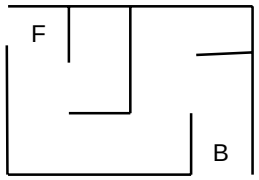
- What is common among these data structures?
- How can you generalize them?
- Consider data structures for representing the following problems…

Course Prerequisites for CSE at UW

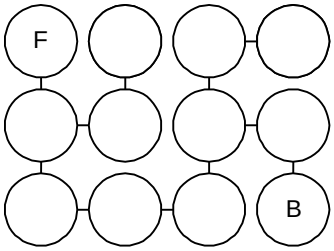


Nodes = courses
Directed edge = prerequisite

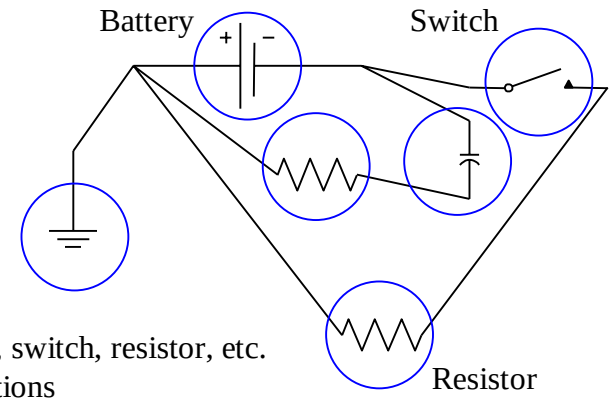
Representing a Maze or Floor Plan of a House



Nodes = rooms
Edge = door or passage



Representing Electrical Circuits

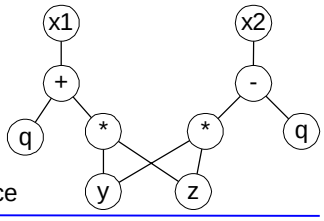


Nodes = battery, switch, resistor, etc.
Edges = connections

Representing Expressions in Compilers

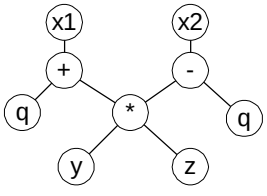
```
x1=q+y*z
x2=y*z-q
```

Naive:



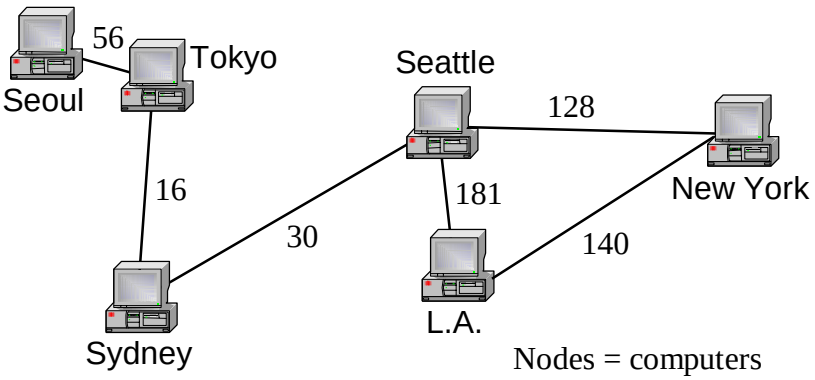
y*z calculated twice

common subexpression eliminated:



Nodes = symbols/operators
Edges = relationships

Information Transmission in a Computer Network

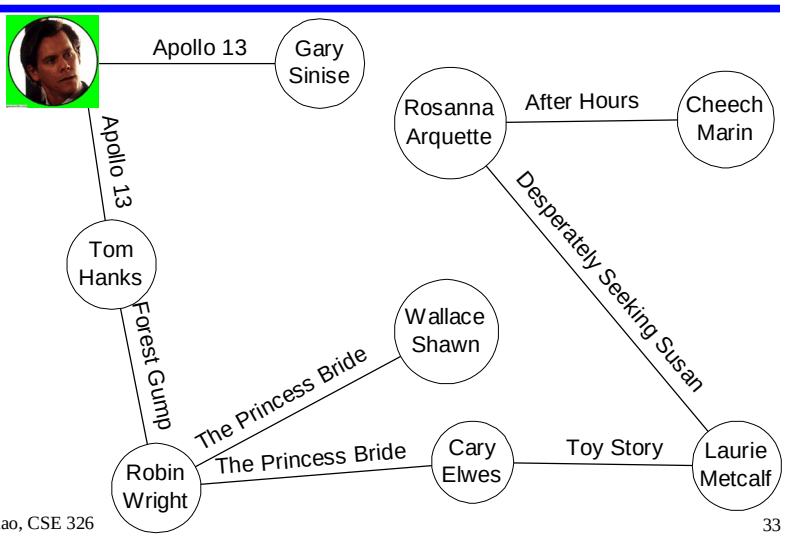


Nodes = computers
Edges = transmission rates

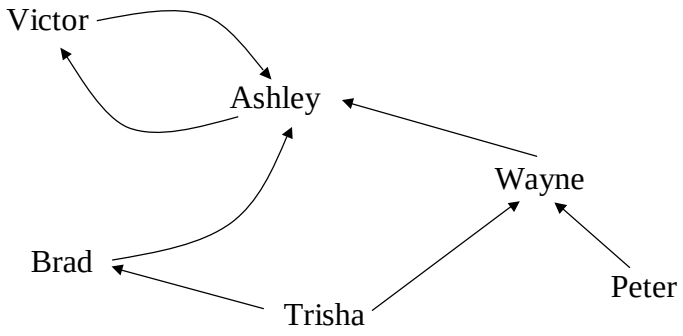
Traffic Flow on Highways

Nodes = cities
Edges = # vehicles on connecting highway

Six Degrees of Separation from Kevin Bacon



Soap Opera Relationships



Graphs: Definition

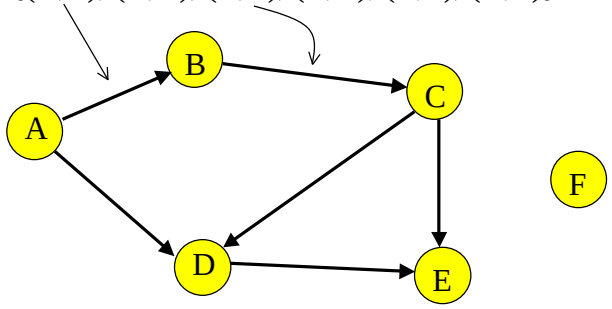
- A graph is simply a collection of nodes plus edges
  - Linked lists, trees, and heaps are all special cases of graphs
- The nodes are known as vertices (node = "vertex")
- Formal Definition: A graph *G* is a pair (*V*, *E*) where
  - *V* is a set of vertices or nodes
  - *E* is a set of edges that connect vertices

Graphs: An Example

- Here is a graph $G = (V, E)$
  - Each edge is a pair (v_1, v_2), where v_1, v_2 are vertices in V

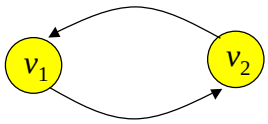
$V = \{A, B, C, D, E, F\}$

$E = \{(A,B), (A,D), (B,C), (C,D), (C,E), (D,E)\}$



Directed versus Undirected Graphs

- If the order of edge pairs (v_1, v_2) matters, the graph is directed (also called a digraph): $(v_1, v_2) \neq (v_2, v_1)$



- If the order of edge pairs (v_1, v_2) does not matter, the graph is called an undirected graph: in this case, $(v_1, v_2) = (v_2, v_1)$

Graph Representations

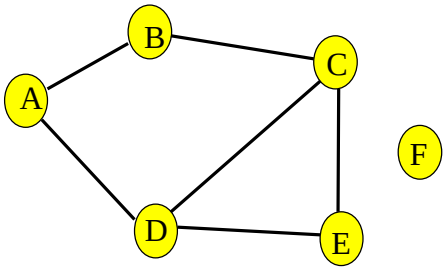
- Space and time are measured in terms of both:
  - Number of vertices = $|V|$ and
  - Number of edges = $|E|$
- There are two ways of representing graphs:
  - The *adjacency matrix* representation
  - The *adjacency list* representation

Graph Representation: Adjacency Matrix

The *adjacency matrix* representation:

Space = ?

$$M(v, w) = \begin{cases} 1 & \text{if } (v, w) \text{ is in E} \\ 0 & \text{otherwise} \end{cases}$$

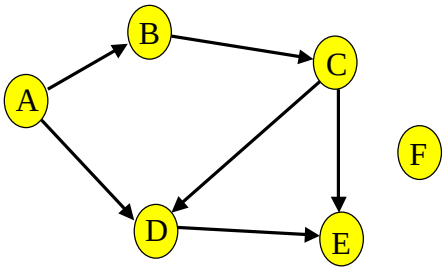


	A	B	C	D	E	F
A	0	1	0	1	0	0
B	1	0	1	0	0	0
C	0	1	0	1	1	0
D	1	0	1	0	1	0
E	0	0	1	1	0	0
F	0	0	0	0	0	0

Adjacency Matrix for a Digraph

Space = $|V|^2$

$$M(v, w) = \begin{cases} 1 & \text{if } (v, w) \text{ is in E} \\ 0 & \text{otherwise} \end{cases}$$

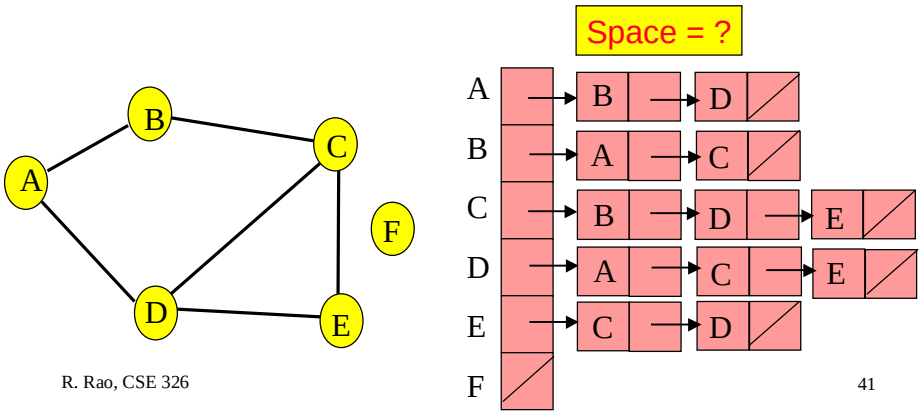


	A	B	C	D	E	F
A	0	1	0	1	0	0
B	0	0	1	0	0	0
C	0	0	0	1	1	0
D	0	0	0	0	1	0
E	0	0	0	0	0	0
F	0	0	0	0	0	0

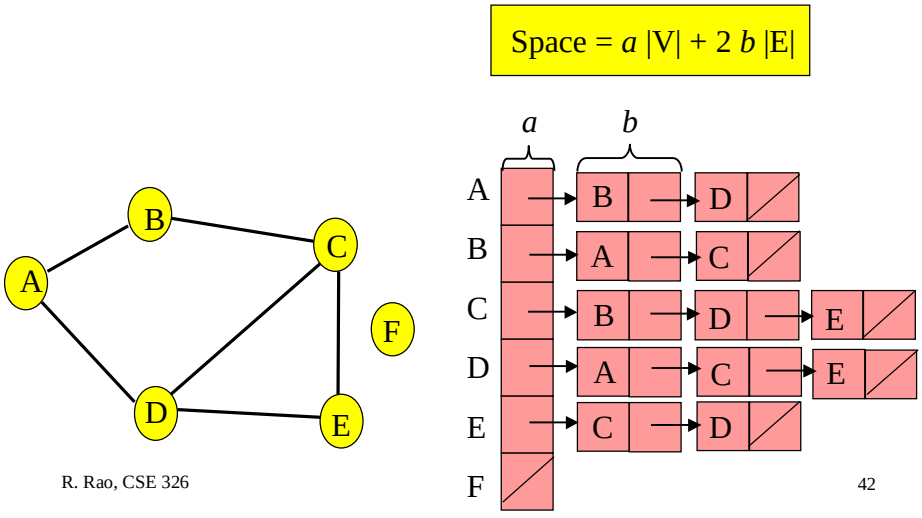
Graph Representation: Adjacency List

The *adjacency list* representation: For each v in V,

$L(v)$ = list of w such that (v, w) is in E

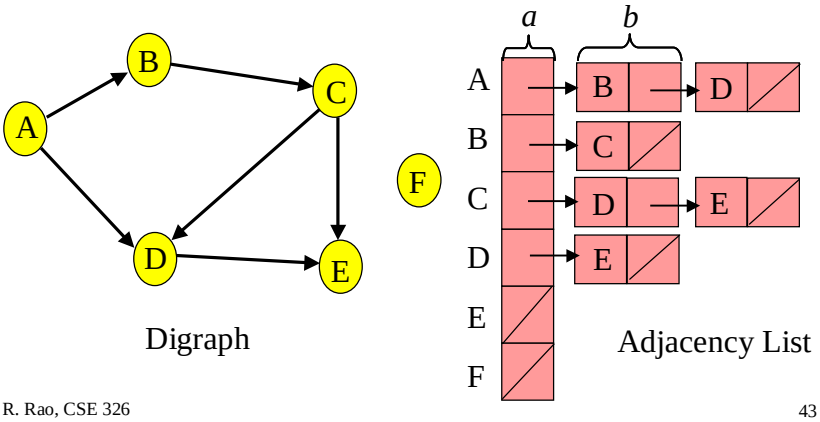


Graph Representation: Adjacency List



Adjacency List for a Digraph

Space = ?

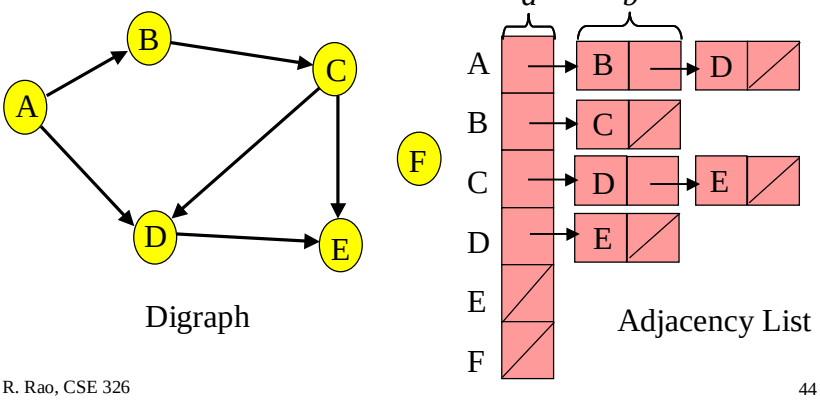


Digraph

Adjacency List

Adjacency List for a Digraph

Space = $a\,|V| + b\,|E|$

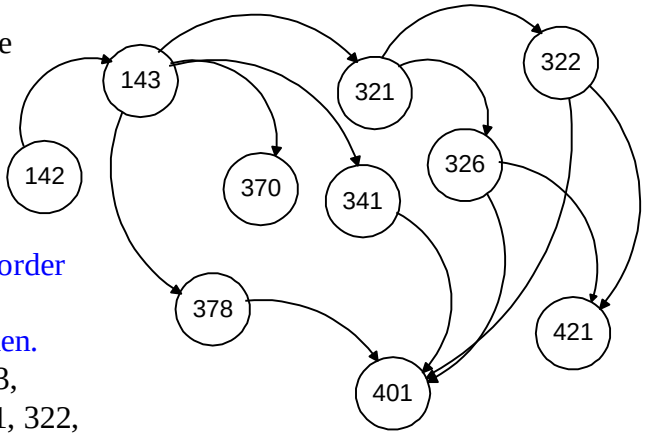


Digraph

Adjacency List

Graphs: Problem #1: Topological Sort

Graph of course prerequisites

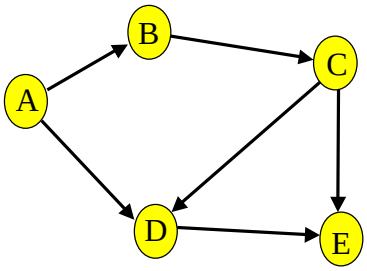


Problem: Find an order in which all these courses can be taken.
Example: 142, 143, 378, 370, 321, 341, 322, 326, 421, 401

To take a course, all its prerequisites must be taken first

Topological Sort Definition

Topological sorting problem: given digraph $G = (V, E)$, find a linear ordering of its vertices such that:
for any edge (v, w) in E, v precedes w in the ordering



F

How would you Topo-Sort this digraph given an adjacency list representation of $G = (V, E)$?

Next Class:

Getting intimate with Topo-sorts

Finding shortest ways to get to your classrooms

To Do:

Homework #4

(to be posted on class web Monday 2/24)

Read and enjoy chapter 9