CSE100: Design and Analysis of Algorithms Lecture 16 – Hashing (wrap up) and Graphs, DFS and BFS

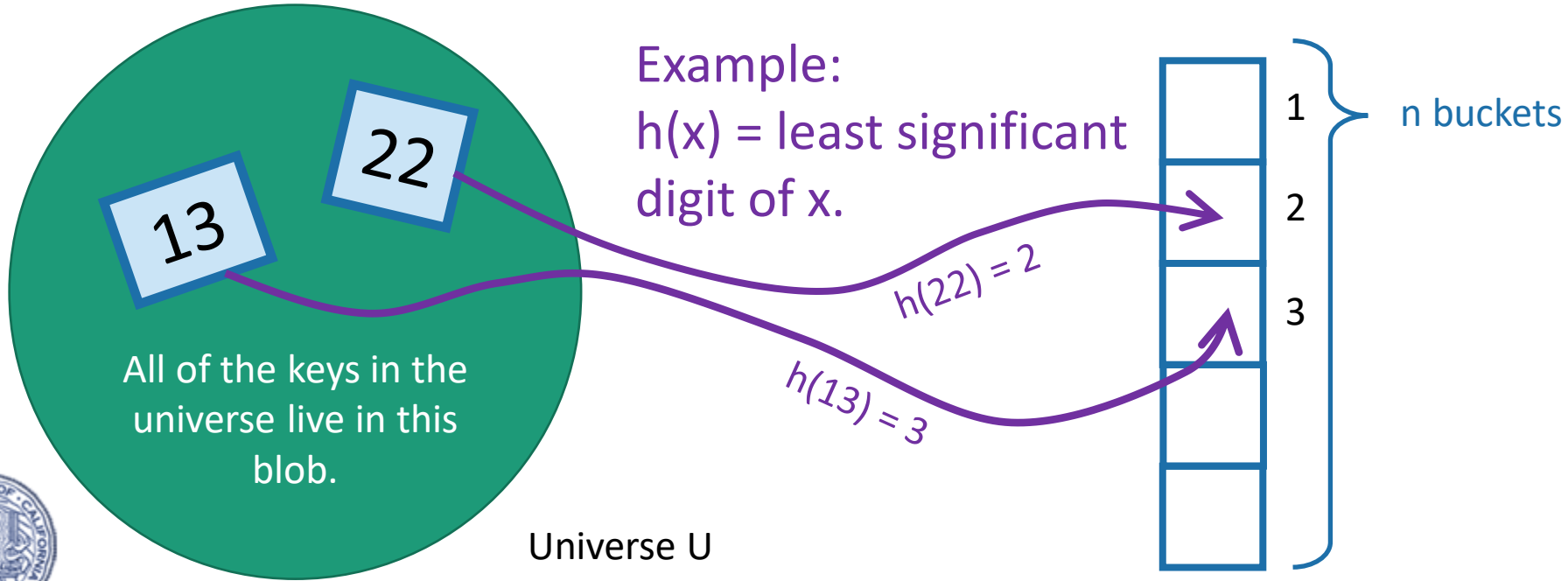
Mar 15th 2022

Hashing, Graphs, Depth First Search and Breadth First Search

Hash Table (review)

For this lecture, I'm assuming that the number of things is the same as the number of buckets, both are n.
This doesn't have to be the case, although we do want:
#buckets = O(#things which show up)

- We have a universe U, of size M.
  - at most n of which will show up.
- M is waaaayyyyyy bigger than n.
- We will put items of U into n buckets.
- There is a *hash function* $h: U \to \{1, \ldots, n\}$ which says what element goes in what bucket.



Worst-case analysis (review)

- Goal: Design a function $h: U \to \{1, \ldots, n\}$ so that:
  - No matter what input (fewer than n items of U) a bad guy chooses, the buckets will be balanced.
  - Here, balanced means O(1) entries per bucket.

- If we had this, then we'd achieve our dream of O(1) INSERT/DELETE/SEARCH

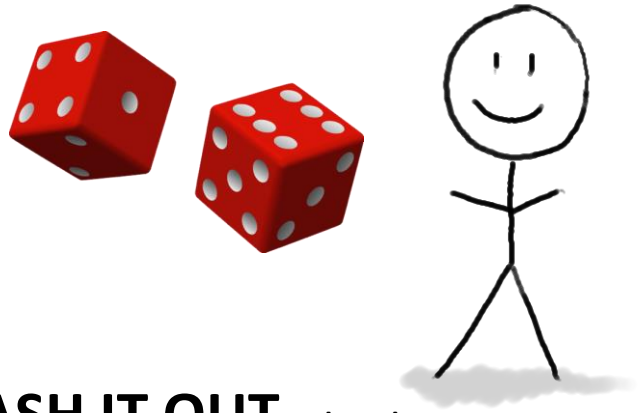
The game (review)

What does **random** mean here? Uniformly random?

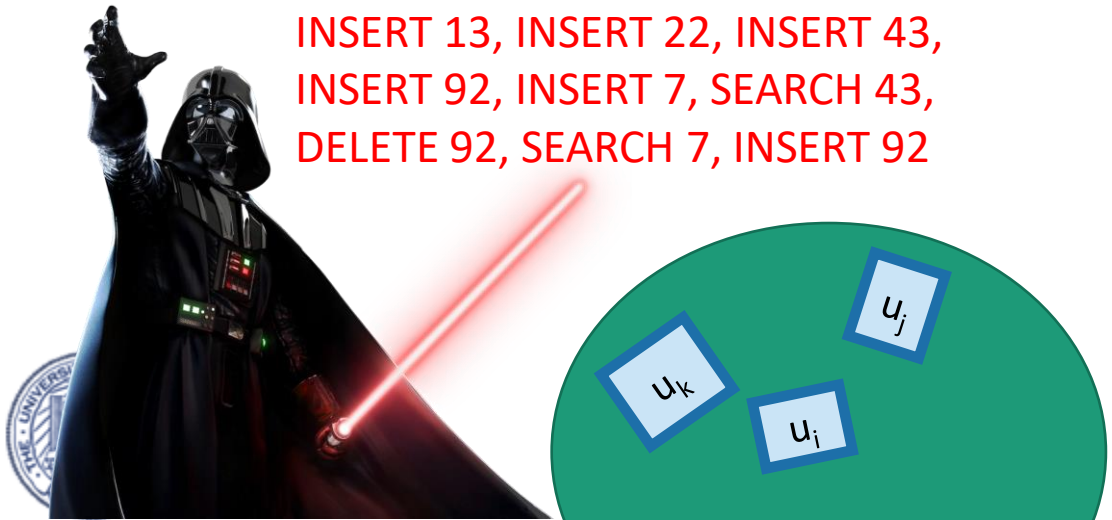
Plucky the pedantic penguin

1. An adversary chooses any n items $u_1, u_2, \ldots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.

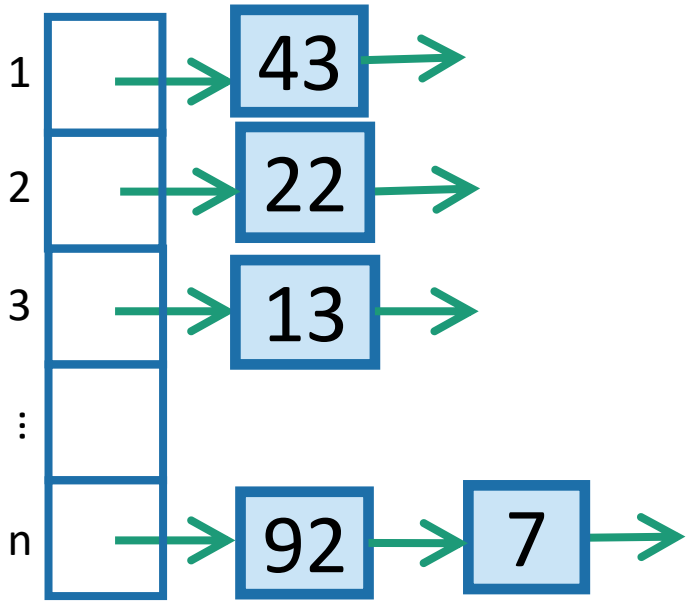
2. You, the algorithm, chooses a **random** hash function $h: U \to \{1, \ldots, n\}$.



3. **HASH IT OUT** #hashpuns



INSERT 13, INSERT 22, INSERT 43, INSERT 92, INSERT 7, SEARCH 43, DELETE 92, SEARCH 7, INSERT 92



With Uniformly Random Hash… (review)

- We showed:
  - For all ways a bad guy could choose $u_1, u_2, \ldots, u_n$, to put into the hash table, and for all $i \in \{1, \ldots, n\}$,

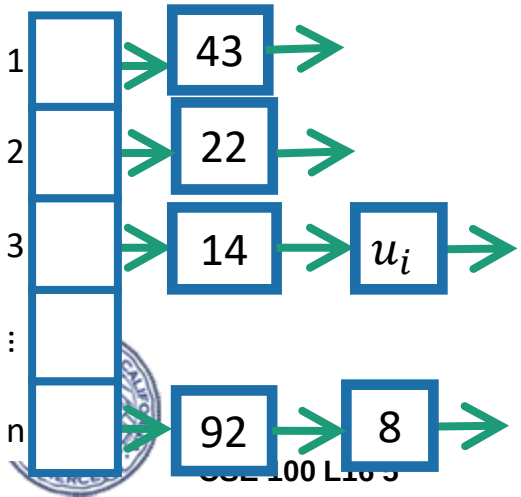
    E[number of items in u_i 's bucket] ≤ 2.

    - Which implies:
      - No matter what sequence of operations and items the bad guy chooses,

        E[time of INSERT/DELETE/SEARCH] = O(1)

So our solution is:

Pick a uniformly random hash function?

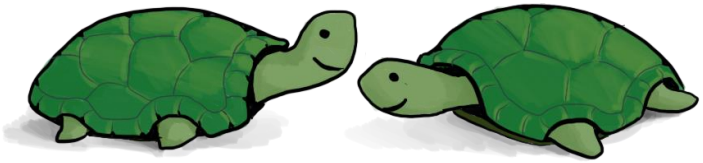


Outline

- Hash tables are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
  - like self-balancing binary trees
  - The difference is we can get better performance in expectation by using randomness.
- **Hash families** are the magic behind hash tables.
- **Universal hash families** are even more magical.

What's wrong with this plan?

- Hint: How would you implement (and store) and uniformly random function $h: U \to \{1, \ldots, n\}$?



Think-Pair-Share Terrapins

- If h is a uniformly random function:
  - That means that **h(1)** is a **uniformly random** number between 1 and n.
  - **h(2)** is also a **uniformly random** number between 1 and n, independent of h(1).
  - **h(3)** is also a **uniformly random** number between 1 and n, independent of h(1), h(2).
  - …
  - **h(n)** is also a **uniformly random** number between 1 and n, independent of h(1), h(2), …, h(n-1).

A uniformly random hash function is not a good idea.

- In order to store/evaluate a uniformly random hash function, we'd use a lookup table:

All of the M things in the universe

x	h(x)
AAAAAA	1
AAAAAB	5
AAAAAC	3
AAAAAD	3
…	
ZZZZZY	7
ZZZZZZ	3

- Each value of h(x) takes log(n) bits to store.
- Storing M such values requires Mlog(n) bits.
- In contrast, direct addressing (initializing a bucket for every item in the universe) requires only M bits….

Another thought…

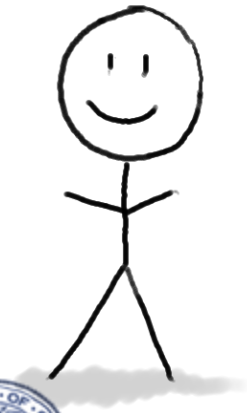
- Just remember h on the relevant values

13 22 43 92 7

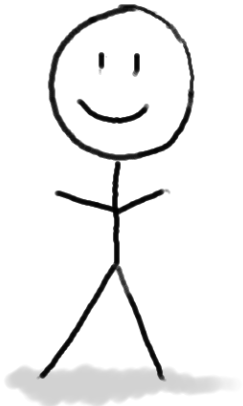
h(7) = 8
h(43) = 2
h(92) = 3
h(22) = 3
h(13) = 6

h(13) = 6
h(22) = 3

We need some way of storing keys and values with O(1) INSERT/DELETE/SEARCH…



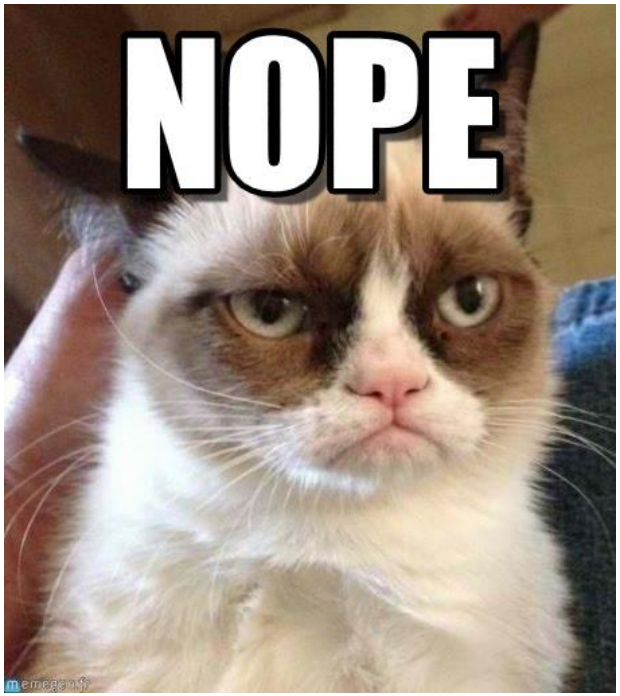
Algorithm now



Algorithm later

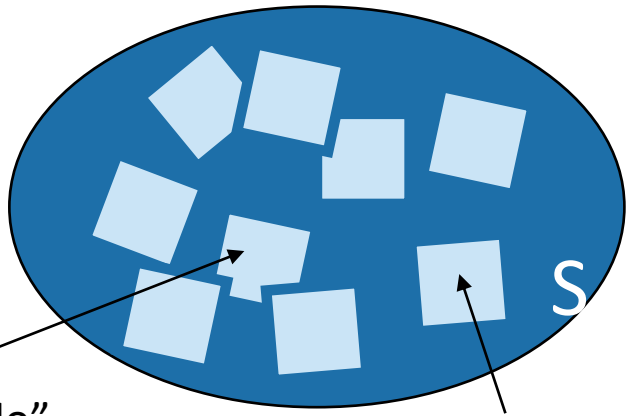
Could we store a uniformly random h without using a lookup table?

- Maybe there's a different way to store h that uses less space?



Aside: description length

- Say I have a set S with s things in it.
- I get to write down the elements of S however I like, in binary using b bits.
- Then $b \geq \log(s)$:
  - There are 2^b binary strings of length b.
  - I need to have at least as many strings as I have items in S.
  - So $s \leq 2^b$ aka $b \geq \log(s)$

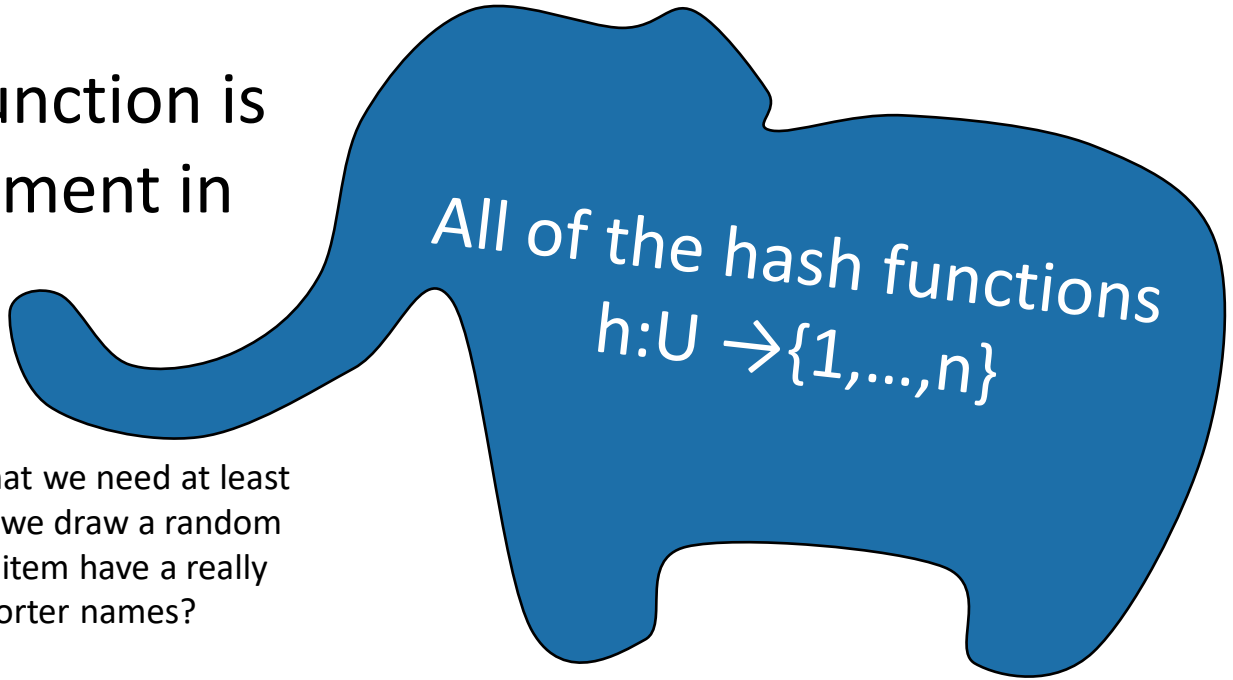


I'll call this one "Fido"
Or, 011011

This one is named "Hercules"
Or, 101111

We need Mlog(n) bits to store a random hash function h:U -> {1,…,n}

- Say that this elephant-shaped blob represents the set of all hash functions.
- It has size n^M. (Really big!)
- To write down a random hash function, we need $\log(n^M) = M\log(n)$ bits.
- A random hash function is just a random element in this set.

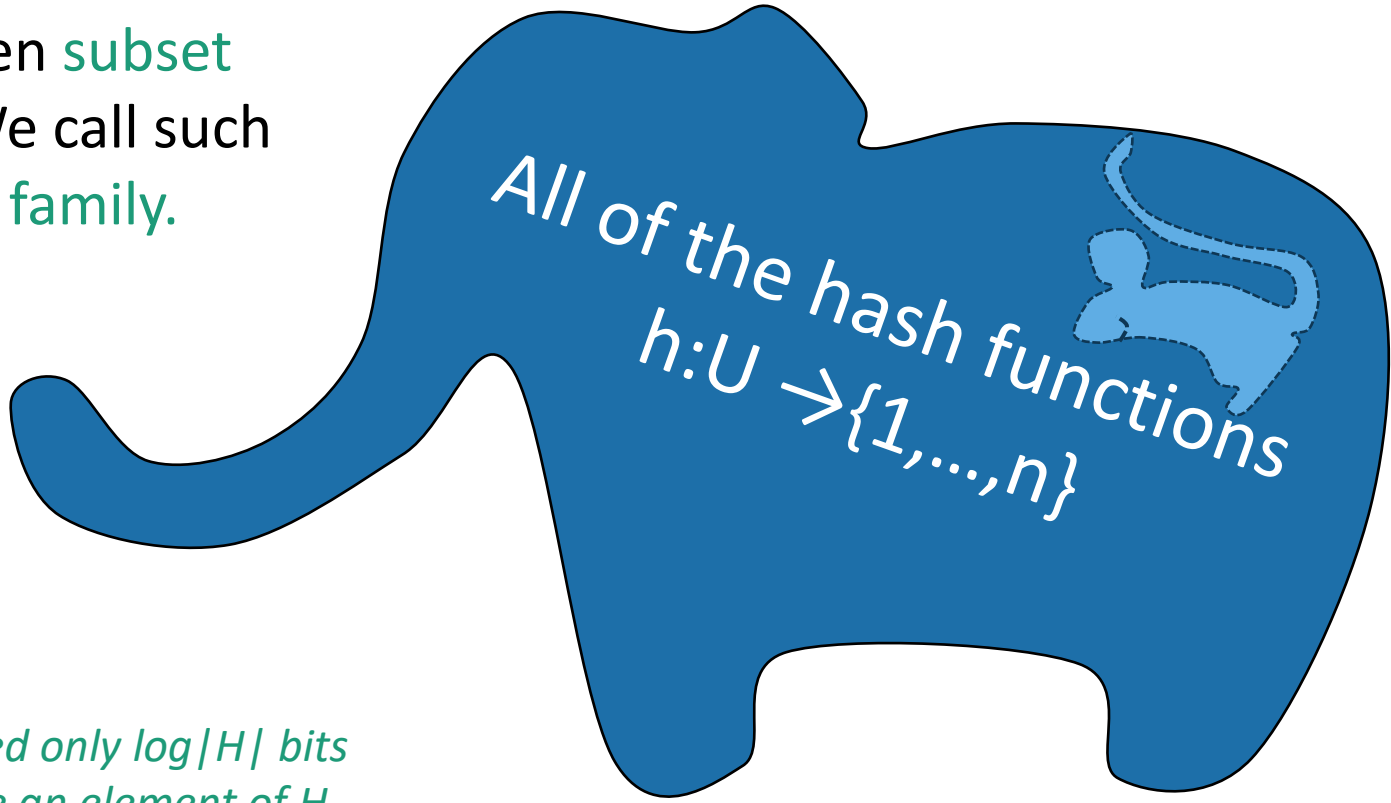


Technically we should argue that we need at least Mlog(n) bits on average when we draw a random hash function… why can't one item have a really long name and others have shorter names?

Solution

- Pick from a smaller set of functions.

A cleverly chosen subset of functions. We call such a subset a hash family.



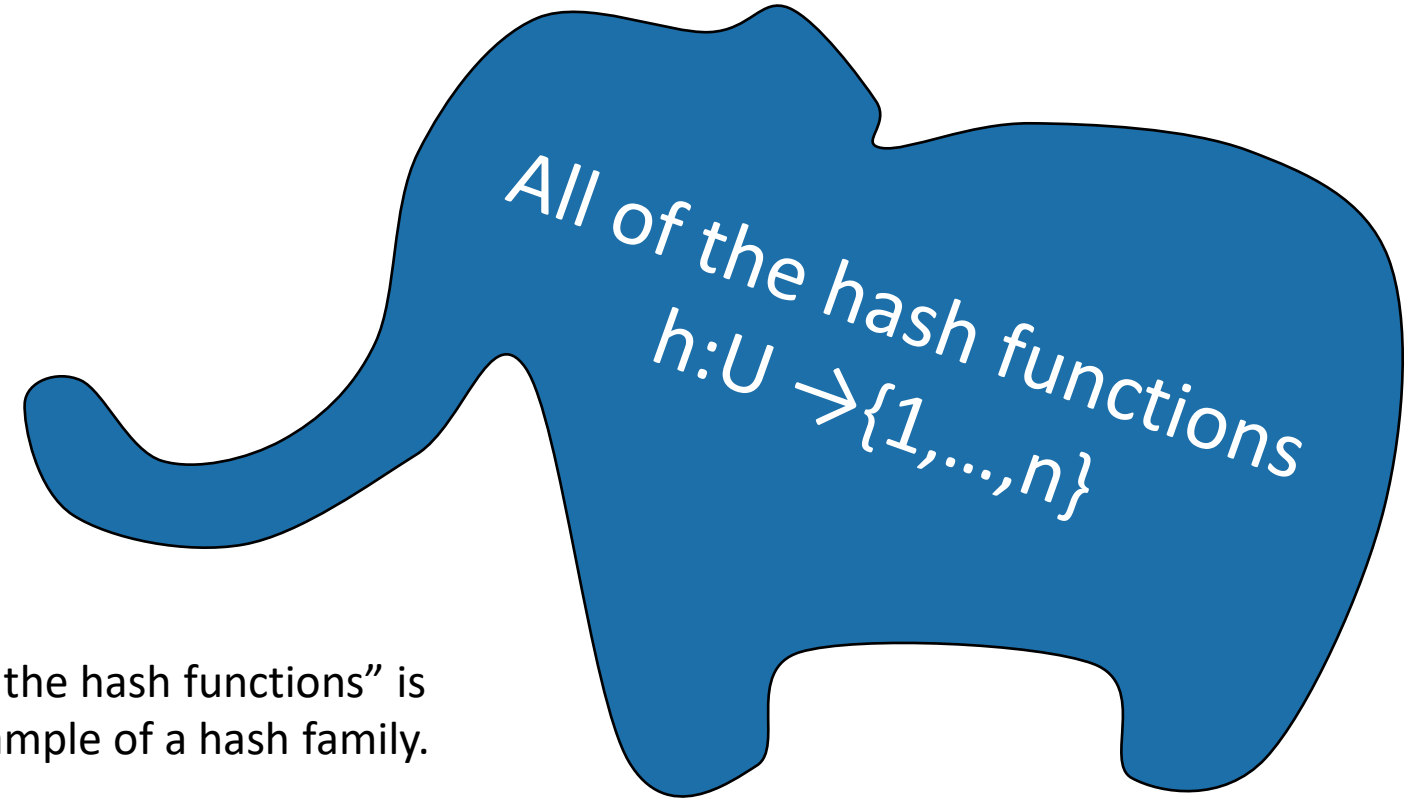
We need only log|H| bits to store an element of H.

Outline

- **Hash tables** are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
  - like self-balancing binary trees
  - The difference is we can get better performance in expectation by using randomness.
- **Hash families** are the magic behind hash tables.
- **Universal hash families** are even more magic.

Hash families

- A hash family is a collection of hash functions.

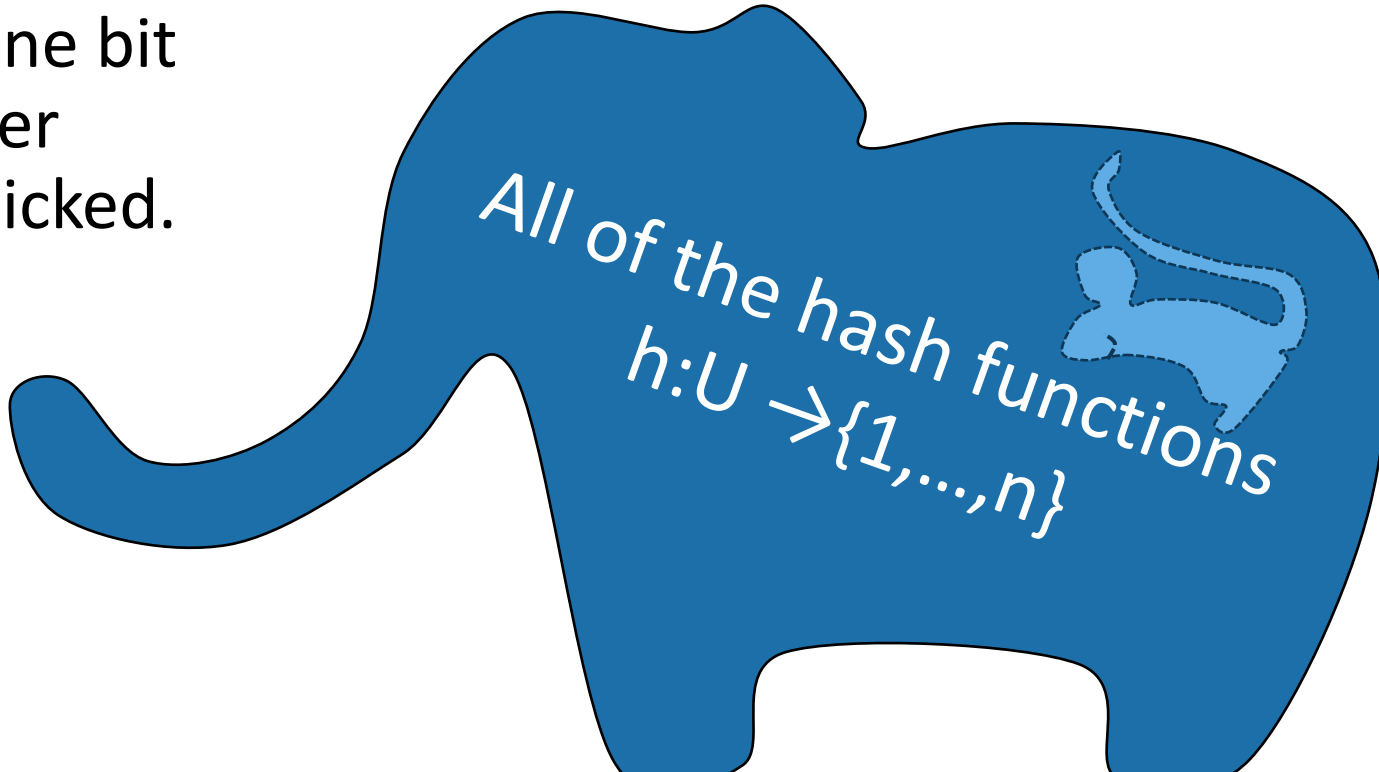


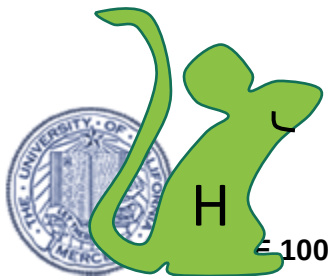
”All of the hash functions” is an example of a hash family.

Example:
a smaller hash family

This is still a terrible idea!
Don't use this example!
For pedagogical purposes only!

- H = { function which returns the least sig. digit,
  function which returns the most sig. digit }
- Pick h in H at random.
- Store just one bit to remember which we picked.



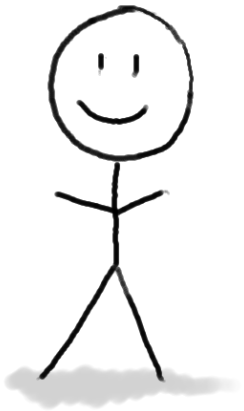


The game

h_0 = Most_significant_digit
h_1 = Least_significant_digit
$H = \{h_0, h_1\}$

2. You, the algorithm, chooses a **random** hash function $h: U \to \{0, \ldots, 9\}$. Choose it randomly from H.

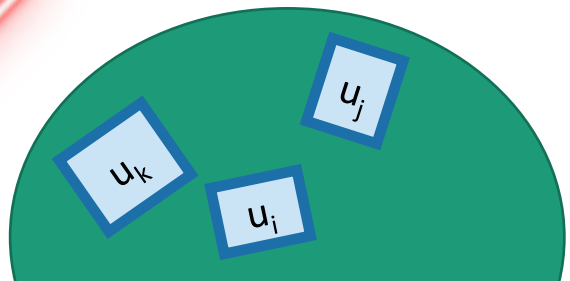
I picked h_1



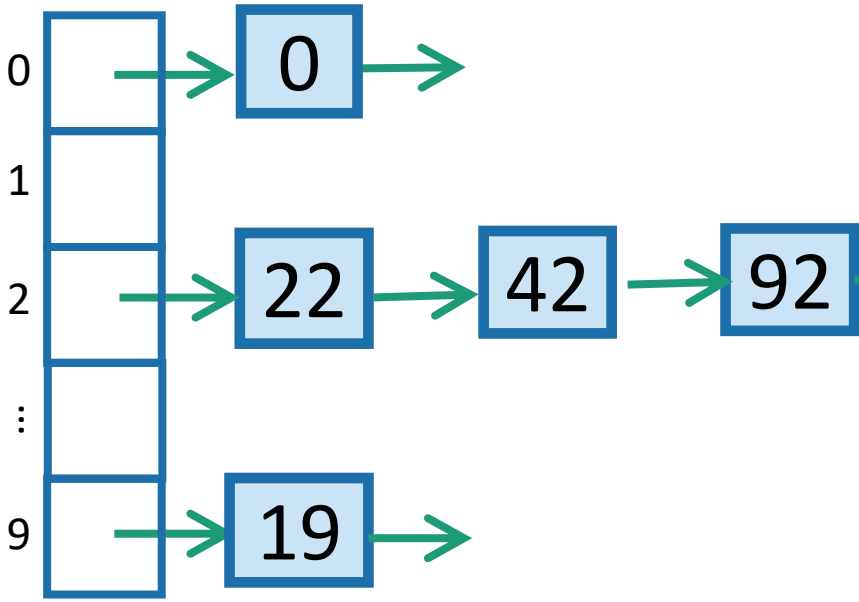
1. An adversary (who knows H) chooses any n items $u_1, u_2, \ldots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.

19	22	42	92	0

INSERT 19, INSERT 22, INSERT 42,
INSERT 92, INSERT 0, SEARCH 42,
DELETE 92, SEARCH 0, INSERT 92



3. HASH IT OUT #hashpuns



This is not a very good hash family

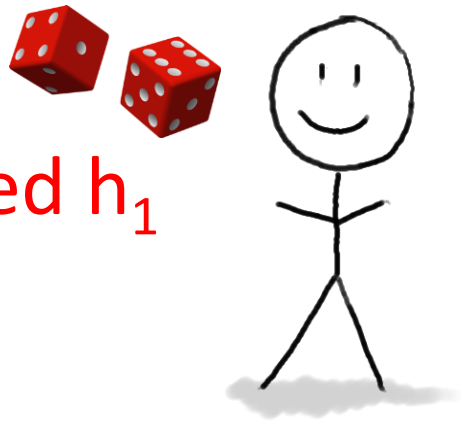
- H = { function which returns least sig. digit,

  function which returns most sig. digit }
- On the previous slide, the adversary could have been a lot more adversarial…

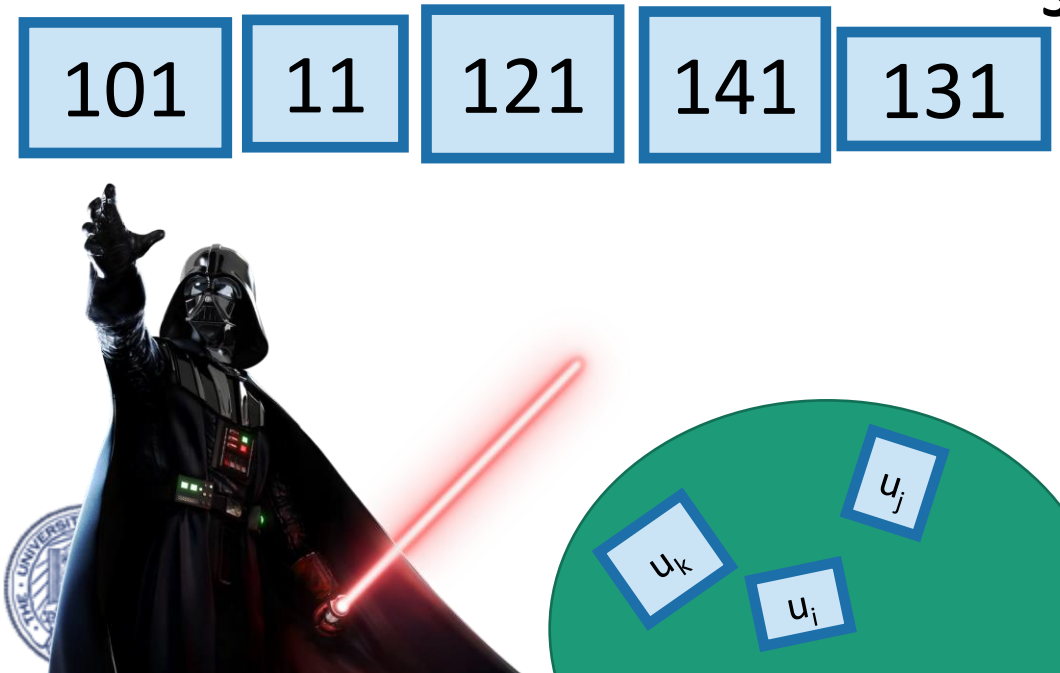
The game

h_0 = Most_significant_digit
h_1 = Least_significant_digit
$H = \{h_0, h_1\}$

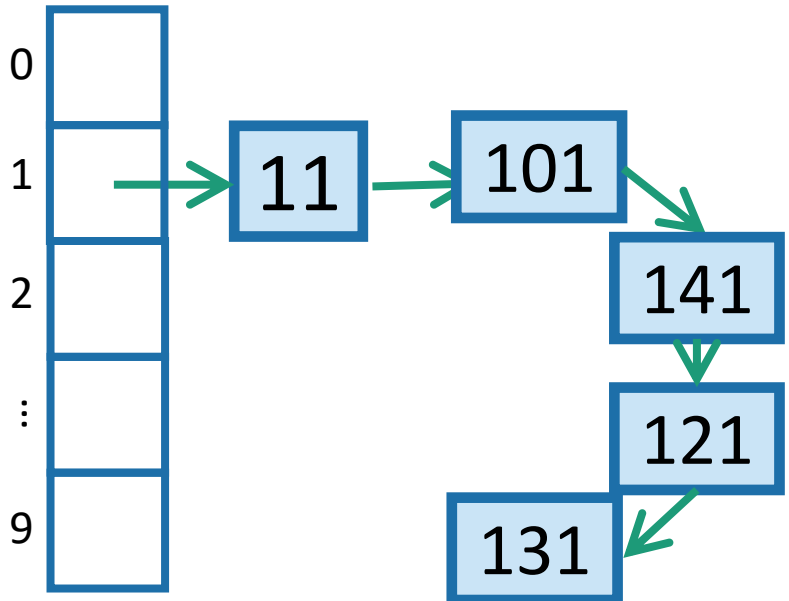
2. You, the algorithm, chooses a **random** hash function $h: U \to \{0, \ldots, 9\}$. Choose it randomly from H.



1. An adversary (who knows H) chooses any n items $u_1, u_2, \ldots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.



3. **HASH IT OUT** #hashpuns

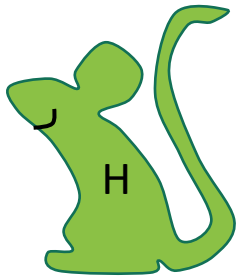


Outline

- **Hash tables** are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
  - like self-balancing binary trees
  - The difference is we can get better performance in expectation by using randomness.
- **Hash families** are the magic behind hash tables.
- **Universal hash families** are even more magic.

How to pick the hash family?

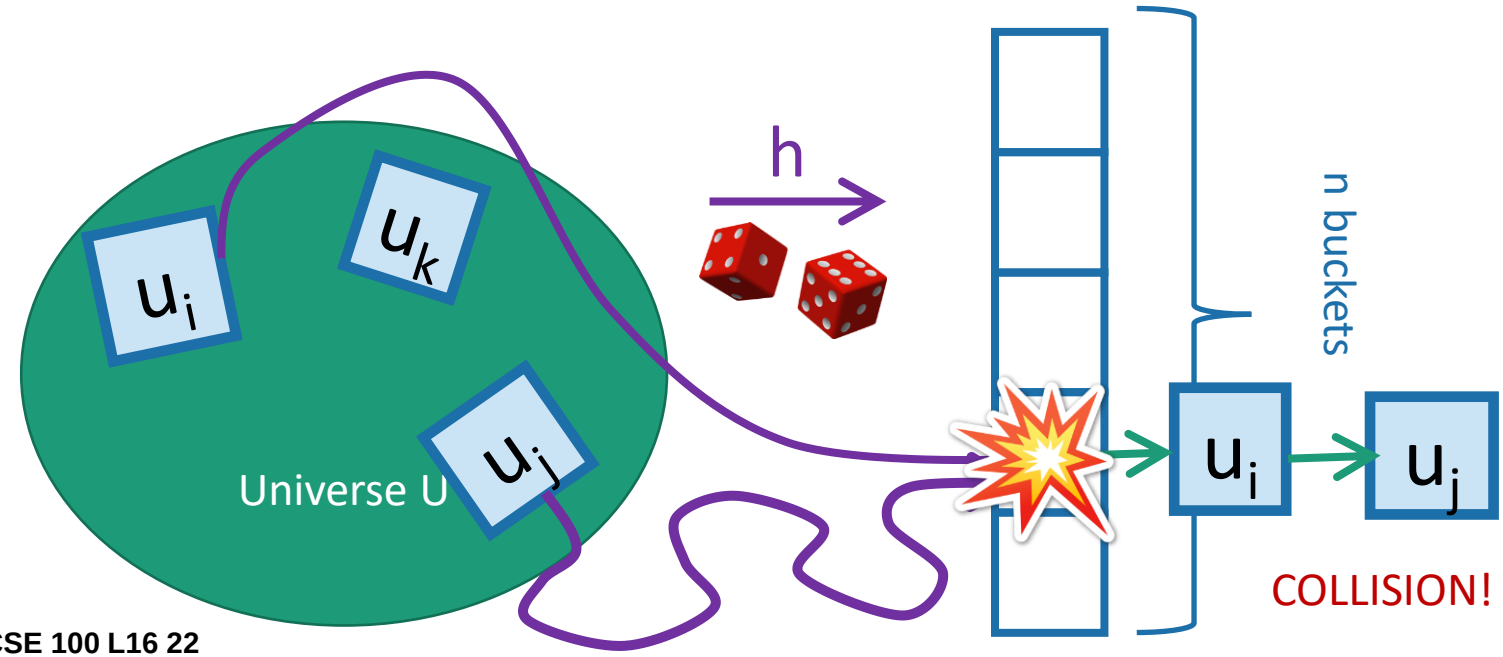
- Definitely not like in that example.
- Let's go back to that computation from earlier….



Expected number of items in u_i's bucket?

- $E[\ ] = \sum_{j=1}^{n} P\{ h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} P\{ h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} 1/n$
- $= 1 + \frac{n-1}{n} \leq 2.$

All that we needed was that this is 1/n



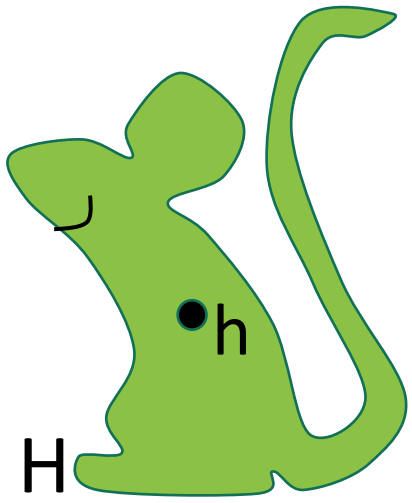
Strategy

- Pick a small hash family H, so that when I choose h randomly from H,

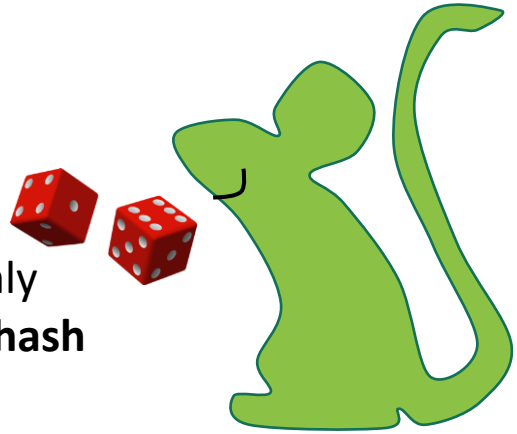
$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$
$$P_{h \in H}\{ h(u_i) = h(u_j)\} \leq \frac{1}{n}$$

In English: fix any two elements of U. The probability that they collide under a random h in H is small.

- A hash family H that satisfies this is called a **universal hash family.**
- Then we still get O(1)-sized buckets in expectation.
- But now the space we need is log(|H|) bits.
  - Hopefully pretty small!



So the whole scheme will be



Choose h randomly from a **universal hash family** H

We can store h in small space since H is so small.

u_1

u_k

u_i

u_j

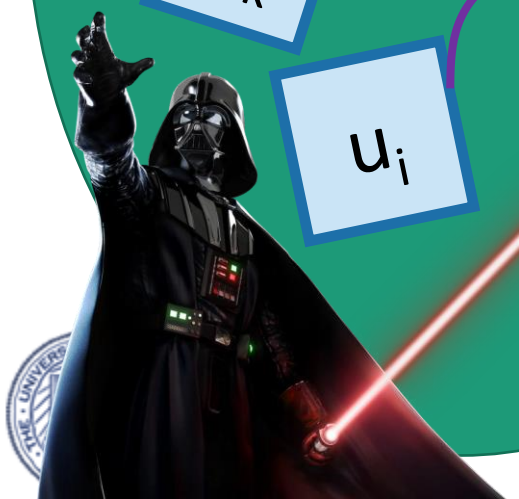
h

n buckets

u_i

Universe U

Probably these buckets will be pretty balanced.



Universal hash family

- H is a ***universal hash family*** if, when h is chosen uniformly at random from H,

$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$

$$P_{h \in H}\{ h(u_i) = h(u_j)\} \leq \frac{1}{n}$$

Example

- Pick a small hash family H, so that when I choose h randomly from H,

$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$
$$P_{h \in H}\{ h(u_i) = h(u_j)\} \leq \frac{1}{n}$$

- H is all of the functions $h: U \to \{1, \ldots, n\}$
  - We saw this earlier – it corresponds to picking a uniformly random hash function.
  - Unfortunately this H is really really large.

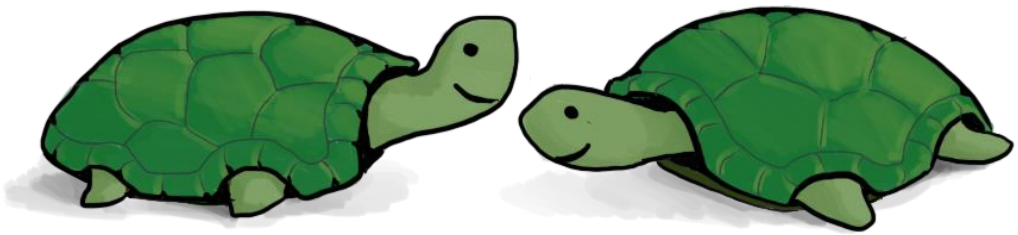
Non-example

- Pick a small hash family H, so that when I choose h randomly from H,

$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$
$$P_{h \in H}\{ h(u_i) = h(u_j)\} \leq \frac{1}{n}$$

- h_0 = Most_significant_digit
- h_1 = Least_significant_digit
- H = {h_0, h_1}

Prove that this choice of H is NOT a universal hash family!



Non-example

- Pick a small hash family H, so that when I choose h randomly from H,

$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$
$$P_{h \in H}\{ h(u_i) = h(u_j)\} \leq \frac{1}{n}$$

- h_0 = Most_significant_digit
- h_1 = Least_significant_digit
- H = {h_0, h_1}

NOT a universal hash family:

$$P_{h \in H}\{h(101) = h(111)\} = 1 > \frac{1}{10}$$

A small universal hash family??

- Here's one:
  - Pick a prime $p \geq M$.
  - Define

$$f_{a,b}(x) = ax + b \quad mod\ p$$

$$h_{a,b}(x) = f_{a,b}(x) \quad mod\ n$$

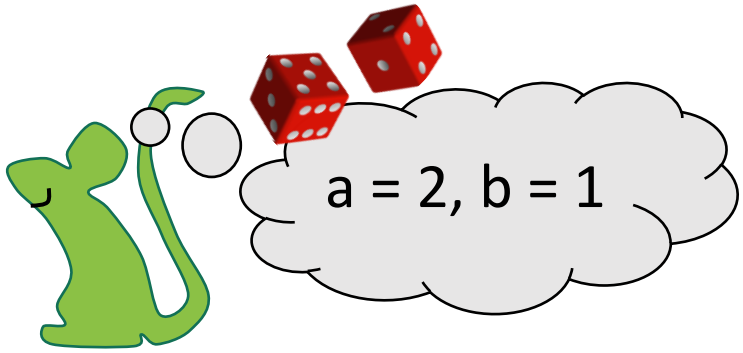
  - Define:

$$H = \{\, h_{a,b}(x) \ : \ a \in \{1, \ldots, p-1\}, b \in \{0, \ldots, p-1\} \,\}$$

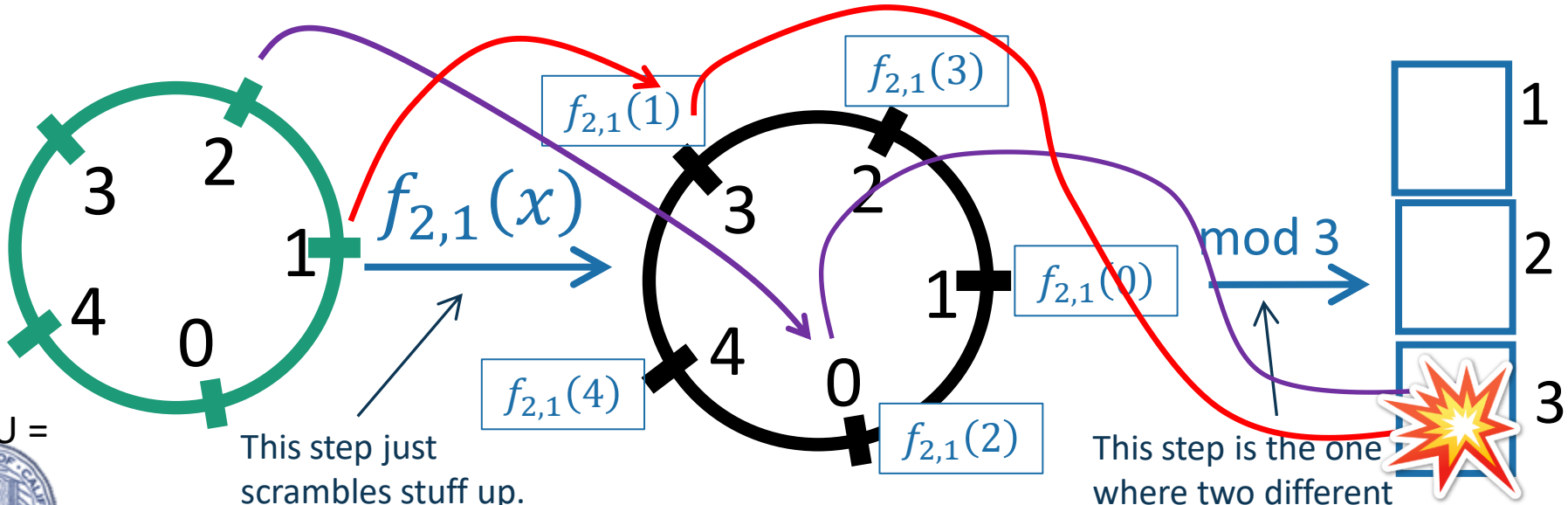
- Claim:

  H is a universal hash family.

Say what?

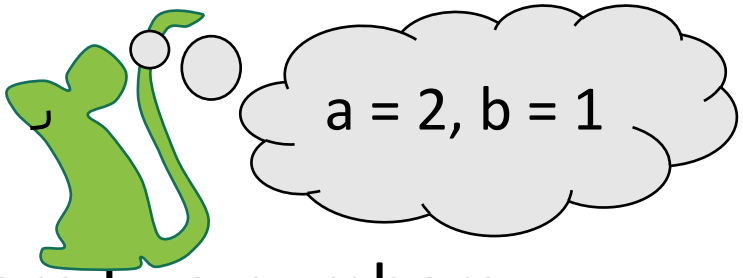


- Example: M = p = 5, n = 3
- To draw h from H:
  - Pick a random a in {1,…,4}, b in {0,…,4}
- As per the definition:
  - $f_{2,1}(x) = 2x + 1 \quad mod\ 5$
  - $h_{2,1}(x) = f_{2,1}(x) \quad mod\ 3$



Ignoring why this is a good idea

- Can we store h with small space?



- Just need to store two numbers:
  - a is in {1,…,p-1}
  - b is in {0,…,p-1}
  - So about 2log(p) bits
  - By our choice of p, that's O(log(M)) bits.

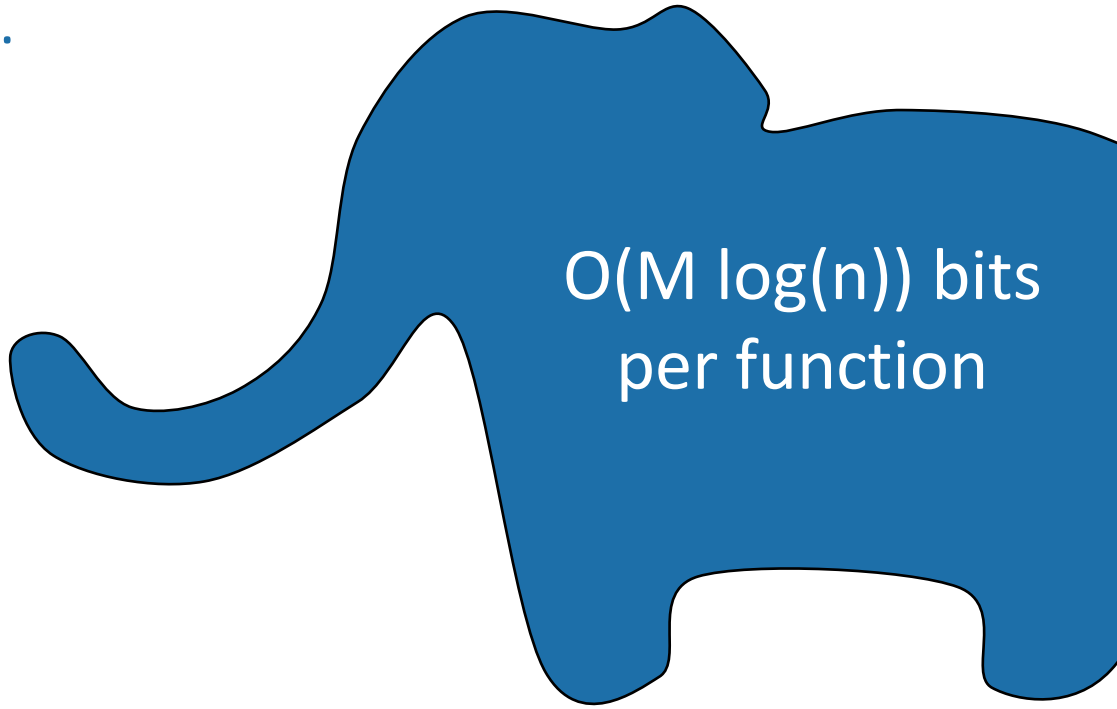
Compare: direct addressing was M bits!

Twitter example: log(M) = 140 log(128) = **980** vs M = $\mathbf{128^{140}}$

Another way to see that

We only need log(M) bits to store h

- We have p-1 choices for a, and p choices for b.
  - So $|H| = p(p-1) = O(M^2)$
- Space needed to store an element h:
  - $\log(M^2) = O(\log(M))$.



O(log(M)) bits per function

Why does this work?

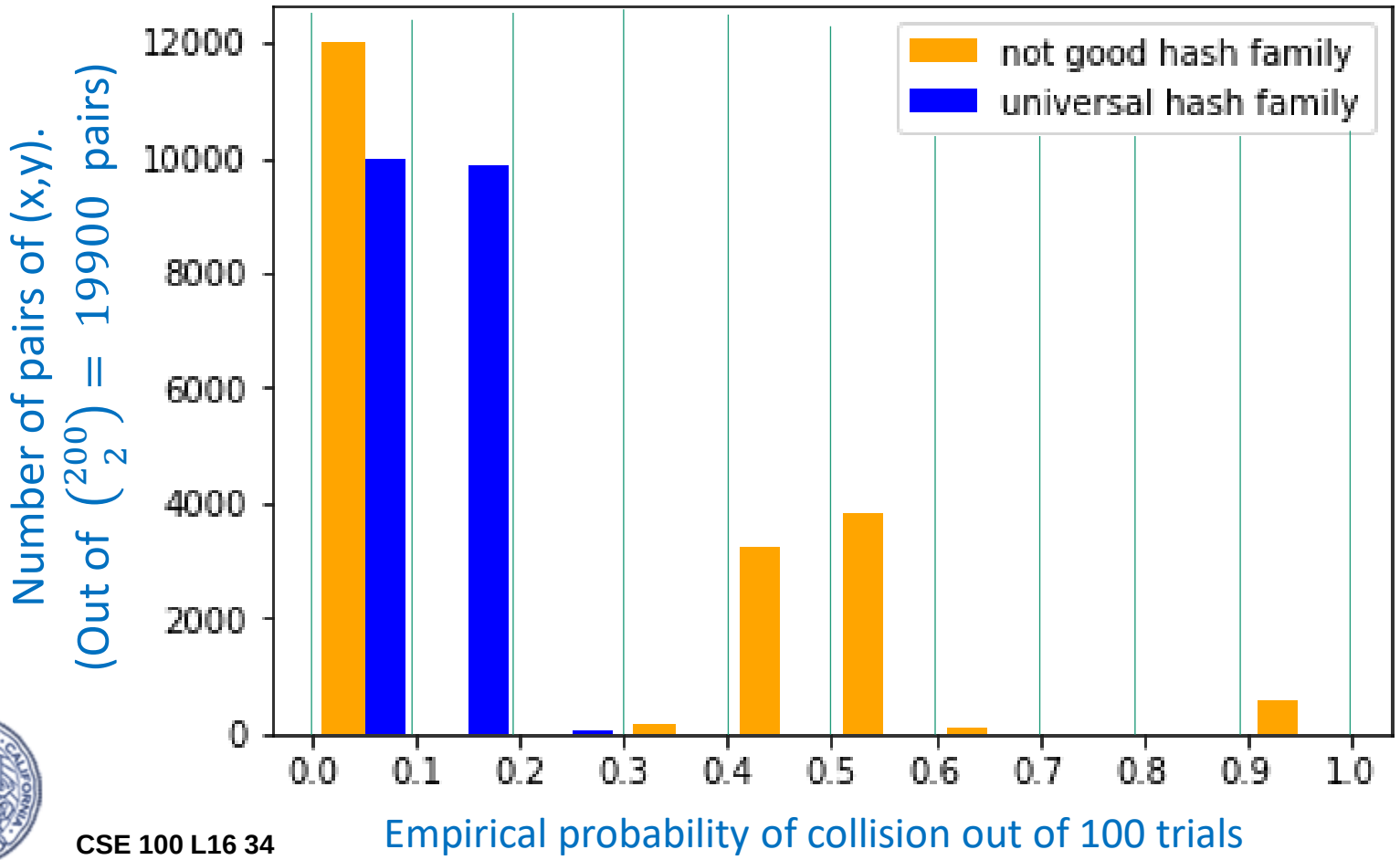
- This is actually a little complicated.
  - See CLRS (Thm 11.5) if you are curious.
  - You are NOT RESPONSIBLE for the proof in this class.
  - But you should know that a universal hash family of size $O(M^2)$ exists.

Try to prove that this is a universal hash family!

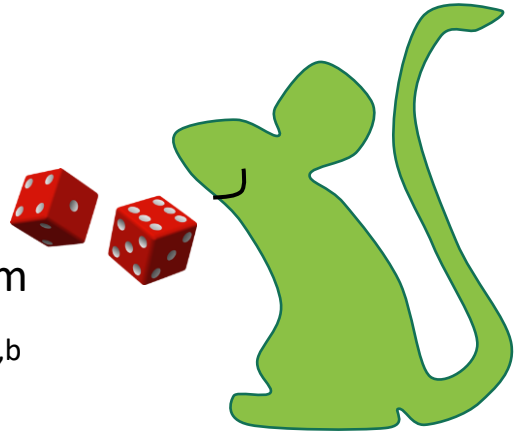
But let's check that it **does** work

- Run a Python program to check this out

M=200, n=10

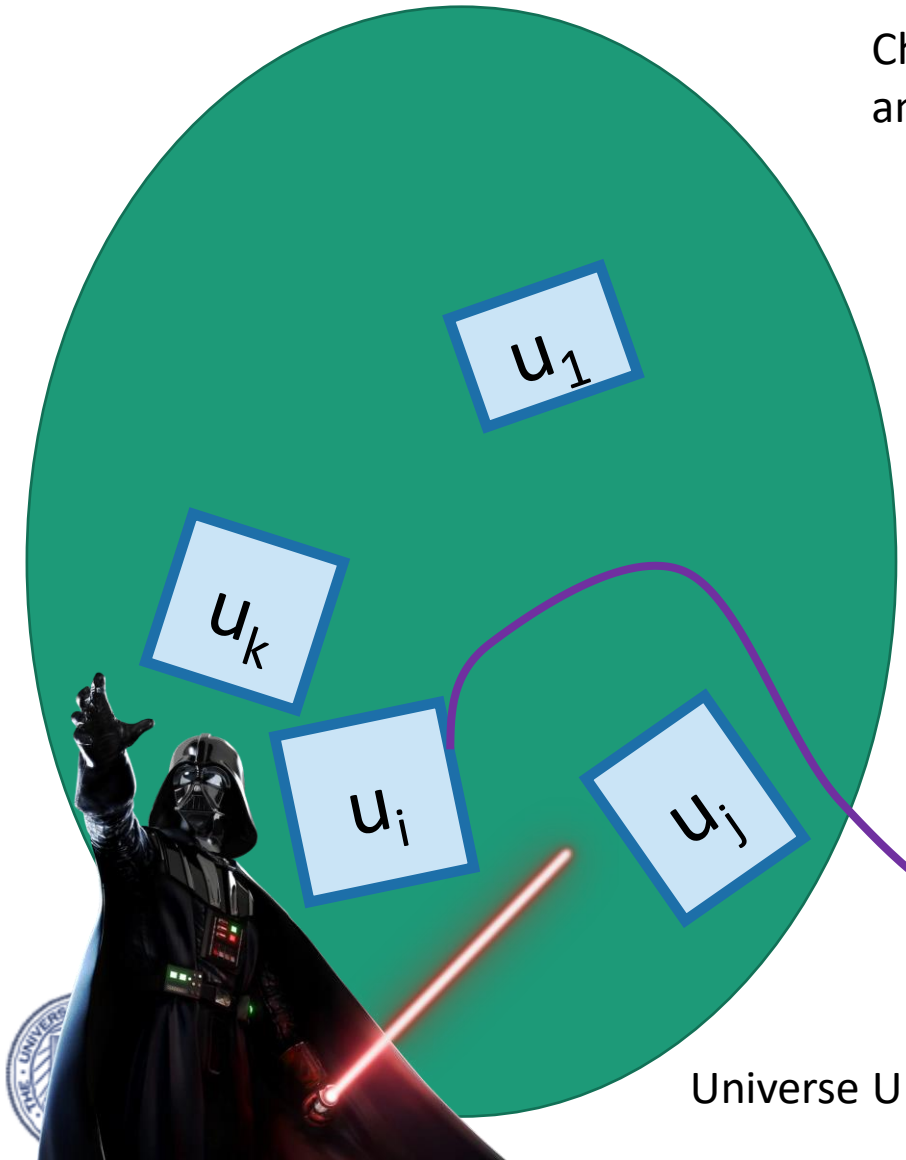


So the whole scheme will be



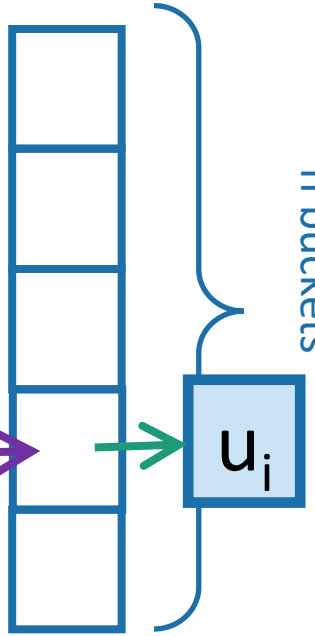
Choose a and b at random and form the function $h_{a,b}$

We can store h in space O(log(M)) since we just need to store a and b.



Universe U

$h_{a,b}$



Probably these buckets will be pretty balanced.

Outline

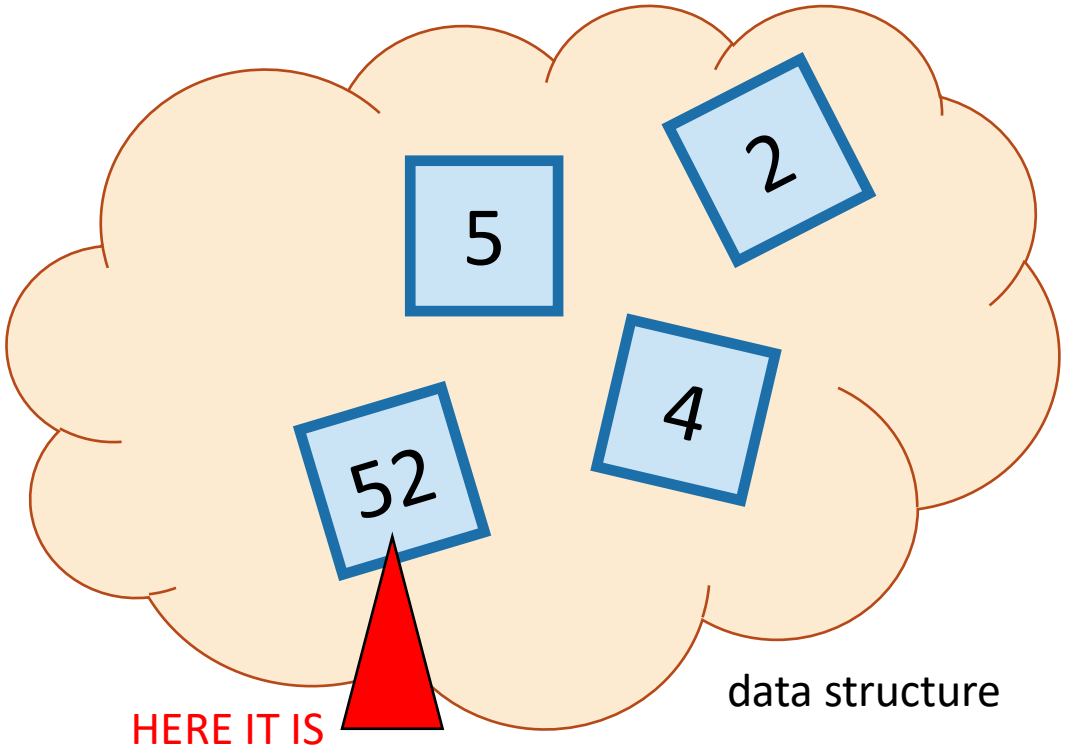
- **Hash tables** are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
  - like self-balancing binary trees
  - The difference is we can get better performance in expectation by using randomness.
- **Hash families** are the magic behind hash tables.
- **Universal hash families** are even more magic.

Recap 

Want O(1) INSERT/DELETE/SEARCH

- We are interesting in putting nodes with keys into a data structure that supports fast INSERT/DELETE/SEARCH.

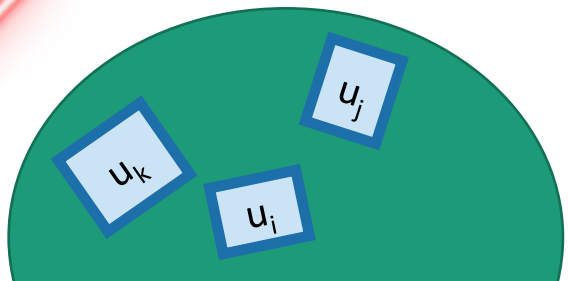
- INSERT 5
- DELETE 4
- SEARCH 52



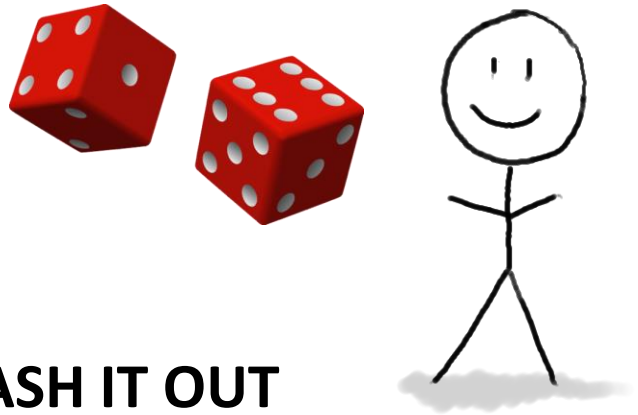
We studied this game

1. An adversary chooses any n items $u_1, u_2, \ldots, u_n \in U$, and any sequence of L INSERT/DELETE/SEARCH operations on those items.

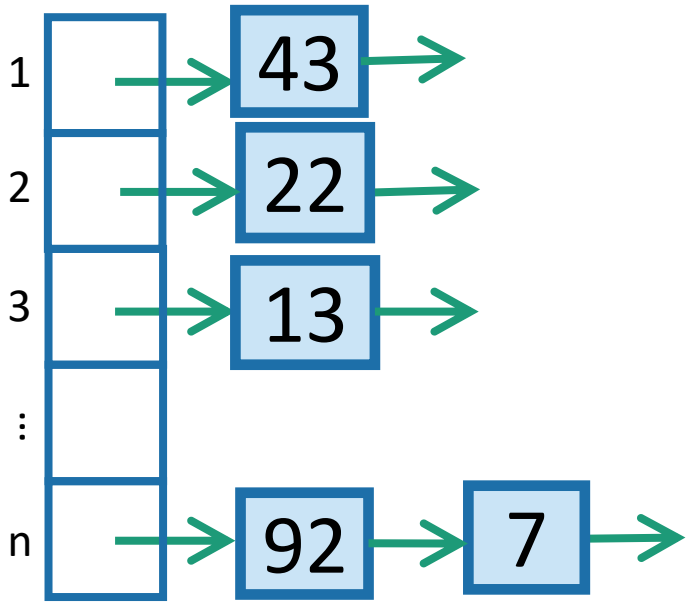
INSERT 13, INSERT 22, INSERT 43,
INSERT 92, INSERT 7, SEARCH 43,
DELETE 92, SEARCH 7, INSERT 92



2. You, the algorithm, chooses a **random** hash function $h: U \to \{1, \ldots, n\}$.



3. **HASH IT OUT**



Uniformly random h was good

- If we choose h uniformly at random,

$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$

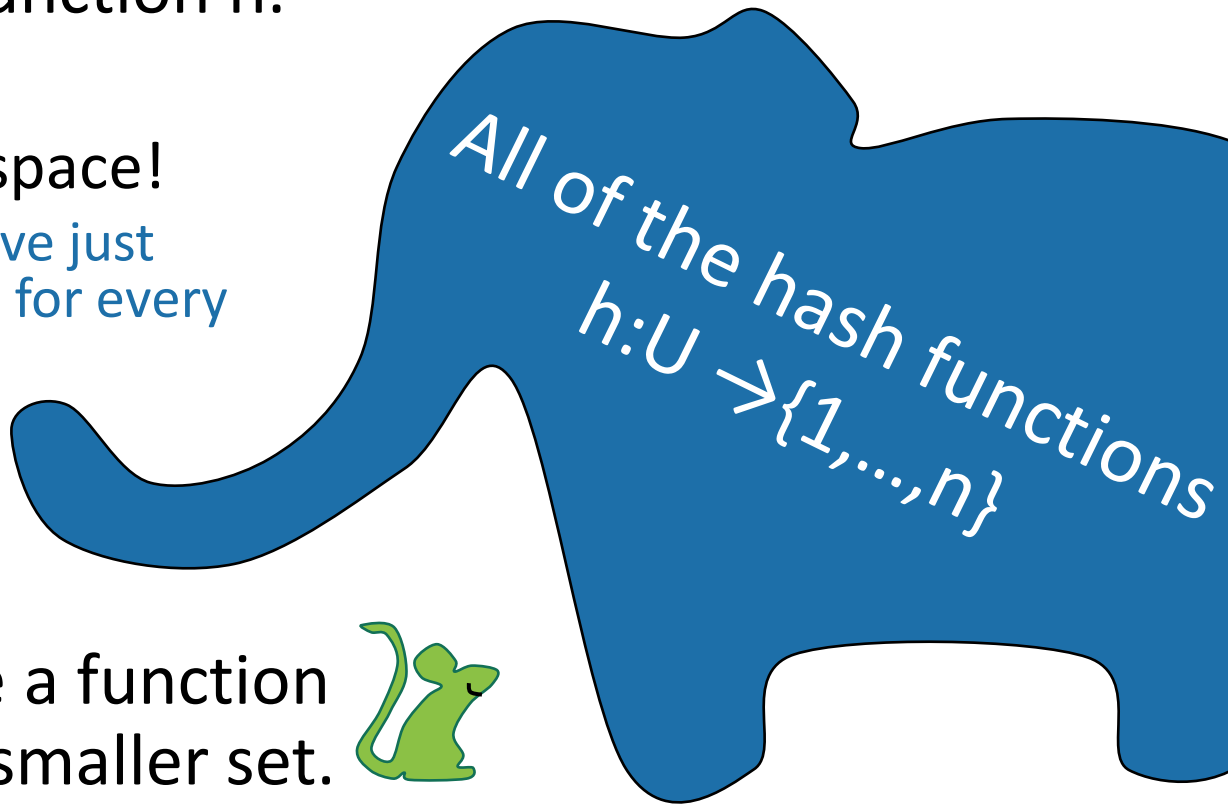
$$P_{h \in H}\{\, h(u_i) = h(u_j)\} \leq \frac{1}{n}$$

- That was enough to ensure that all INSERT/DELETE/SEARCH operations took O(1) time in expectation, even on adversarial inputs.

Uniformly random h was bad

- If we actually want to implement this, we have to store the hash function h.

- That takes a lot of space!
  - We may as well have just initialized a bucket for every single item in U.



- Instead, we chose a function randomly from a smaller set.

We needed a smaller set
that still has this property

- If we choose h uniformly at random in H,

$$\text{for all } u_i, u_j \in U \quad \text{with } u_i \neq u_j,$$

$$P_{h \in H}\{ h(u_i) = h(u_j)\} \leq \frac{1}{n}$$

This was all we needed to make sure that the buckets were balanced in expectation!

- We call any set with that property a

universal hash family.

- We gave an example of a really small one ☺

Hashing a universe of size M into n buckets, where at most n of the items in M ever show up.

Conclusion:

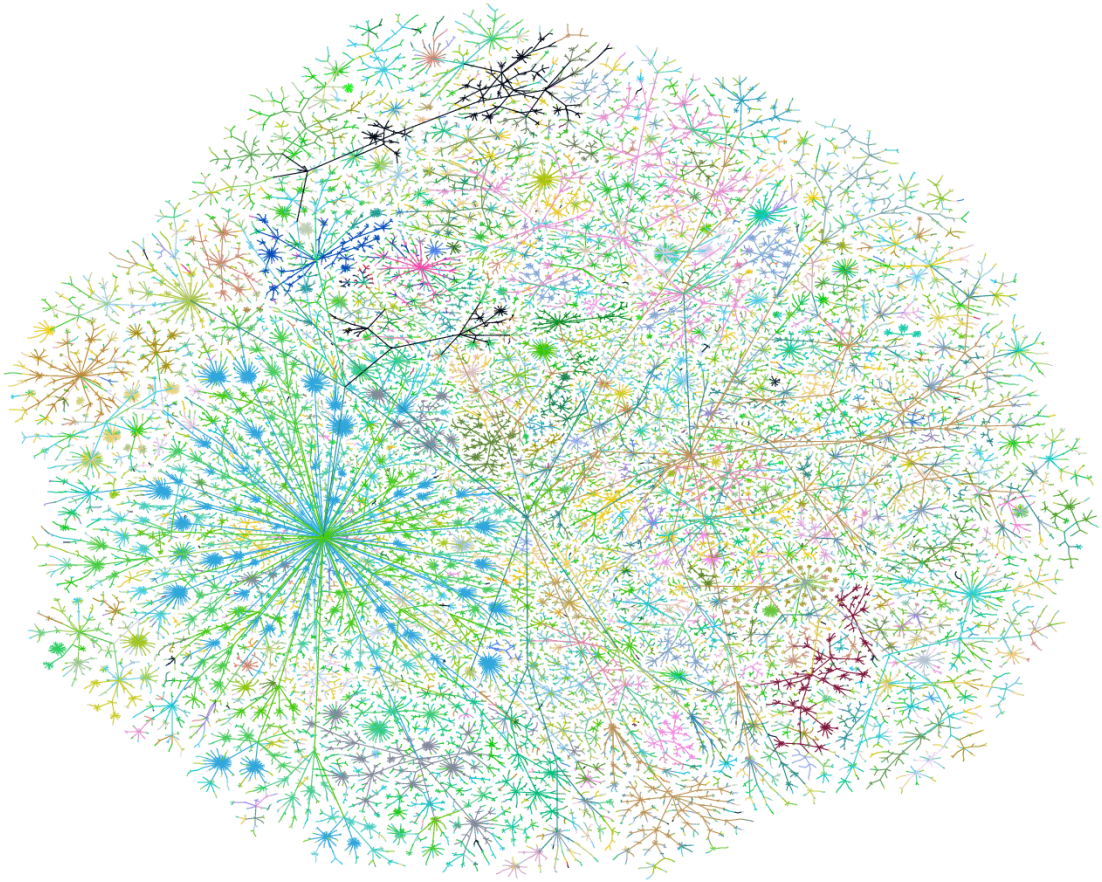
- We can build a hash table that supports INSERT/DELETE/SEARCH in O(1) expected time
- Requires O(n log(M)) bits of space.
  - O(n) buckets
  - O(n) items with log(M) bits per item
  - O(log(M)) to store the hash function

That's it for data structures (for now)

Now we can use these going forward!

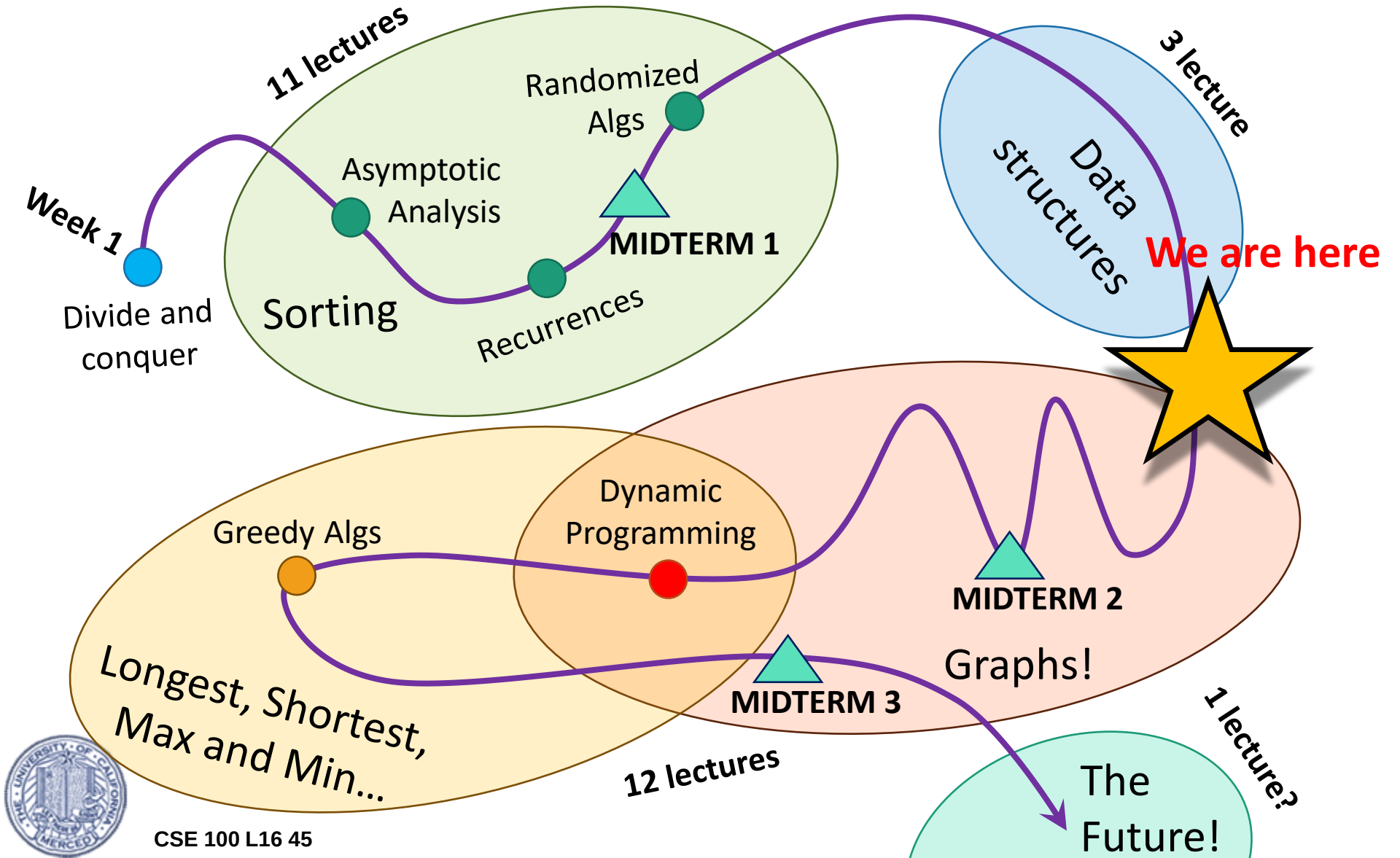
Next Part

- Graph algorithms!



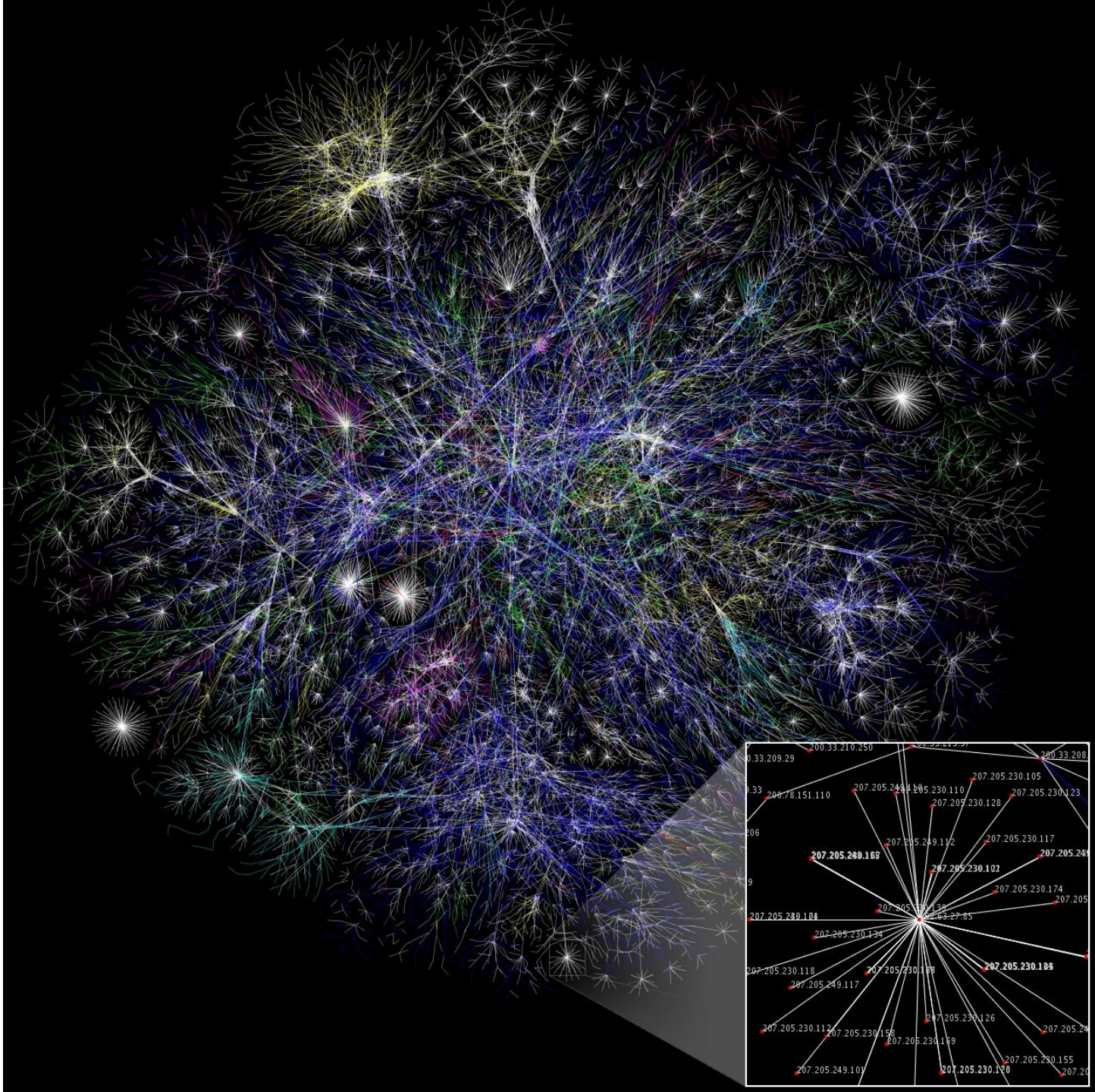
More detailed schedule on the website!

Roadmap



Graphs

Graph of the internet (circa 2005…it's a lot bigger now…)

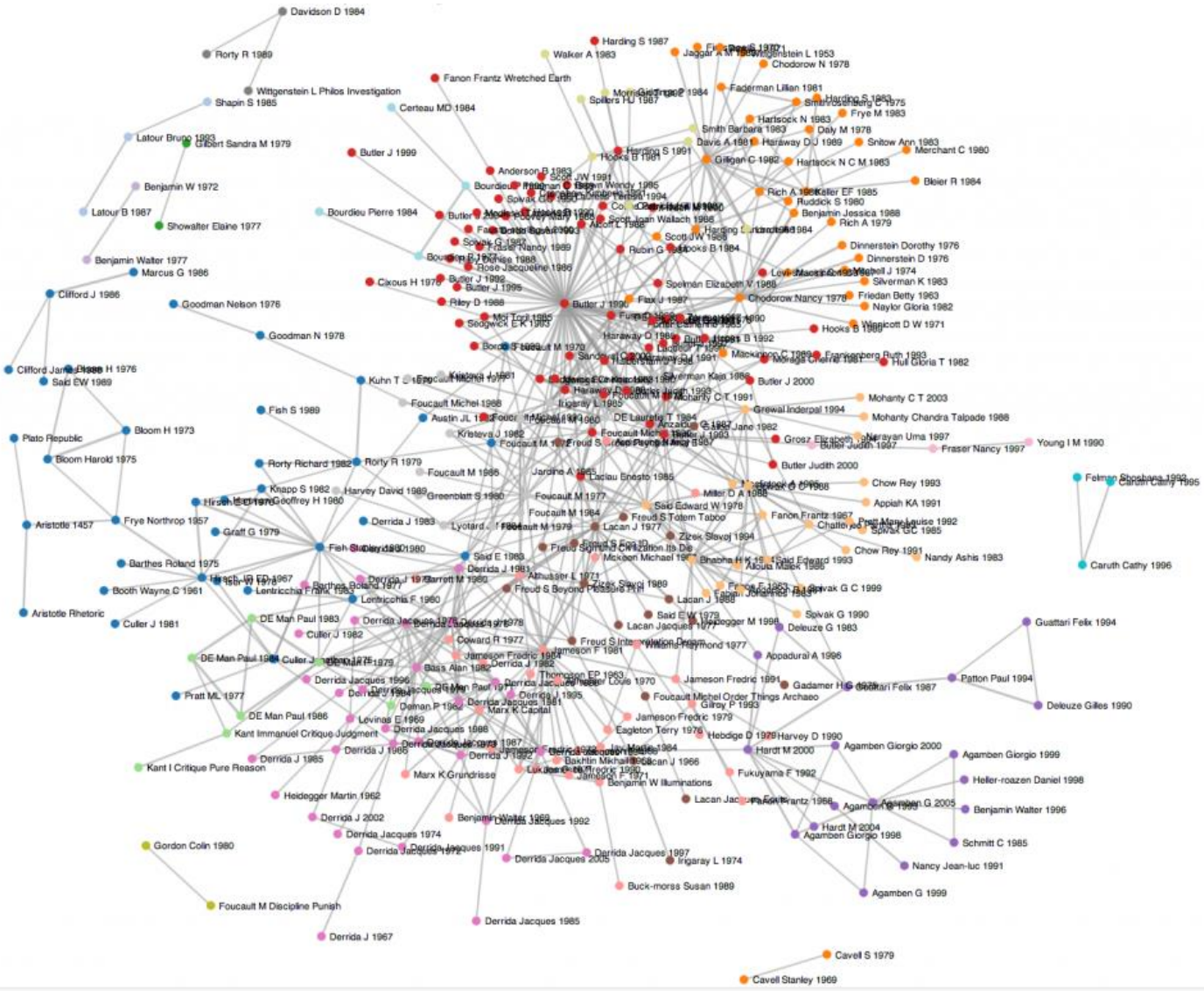


Outline

- Part A: Graphs and terminology
- Part B: Depth-first search
  - Application: topological sorting
  - Application: in-order traversal of BSTs
- Part C: Breadth-first search
  - Application: shortest paths
  - Application (if time): is a graph bipartite?

Part A: Graphs

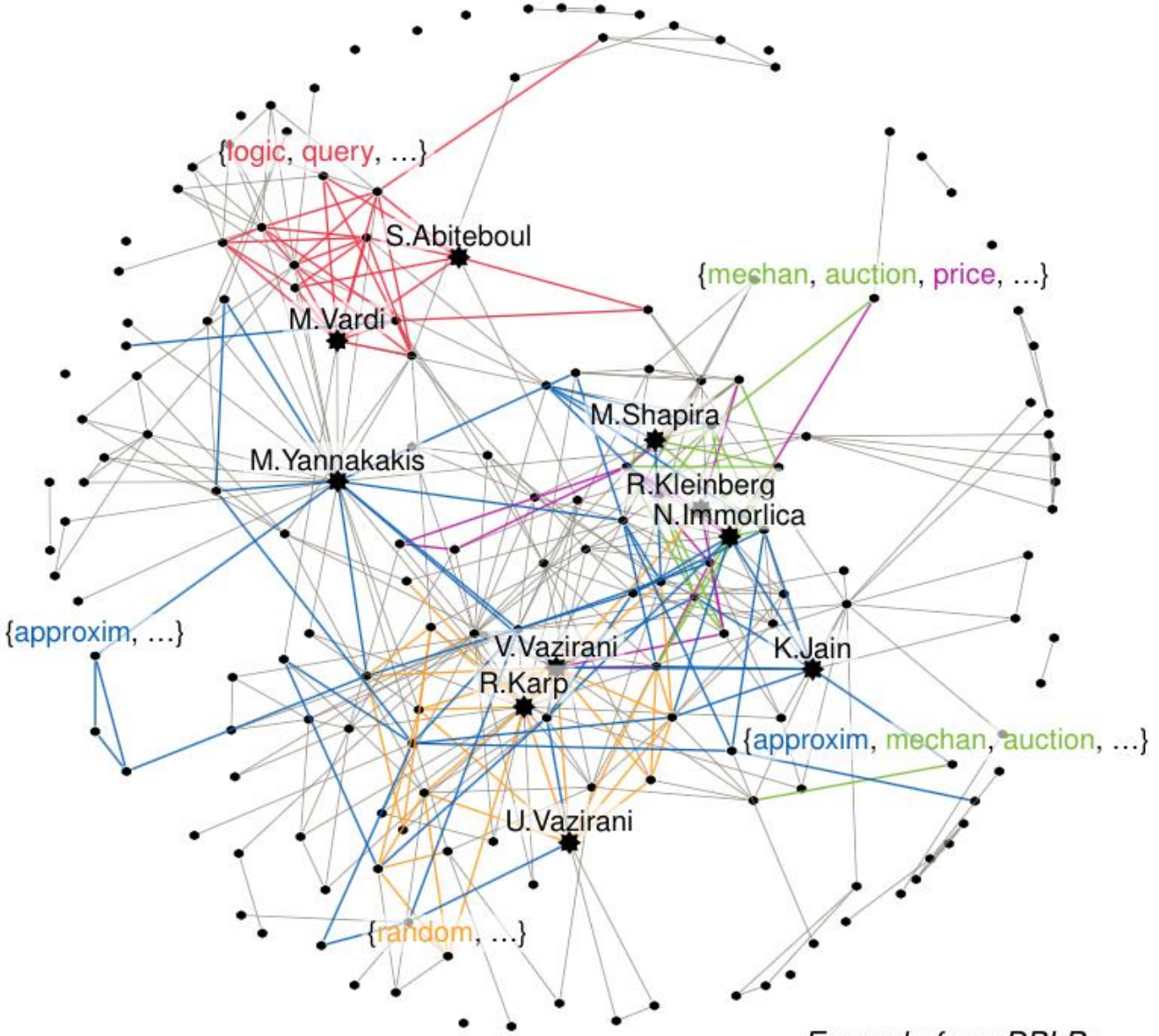
Graphs



Citation graph of literary theory academic papers

Graphs

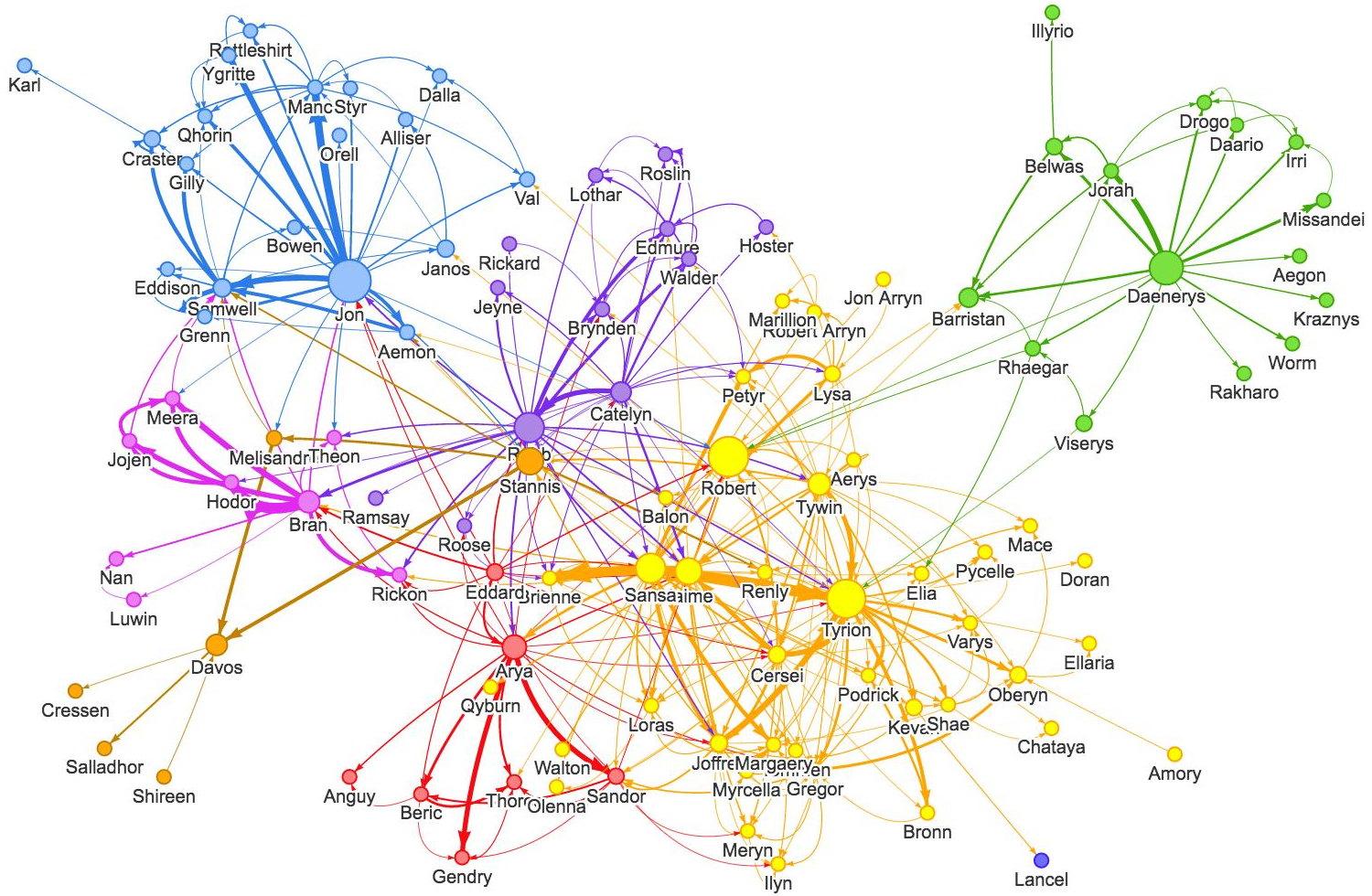
Theoretical Computer Science academic communities



Example from DBLP:
Communities within the co-authors of Christos H. Papadimitriou

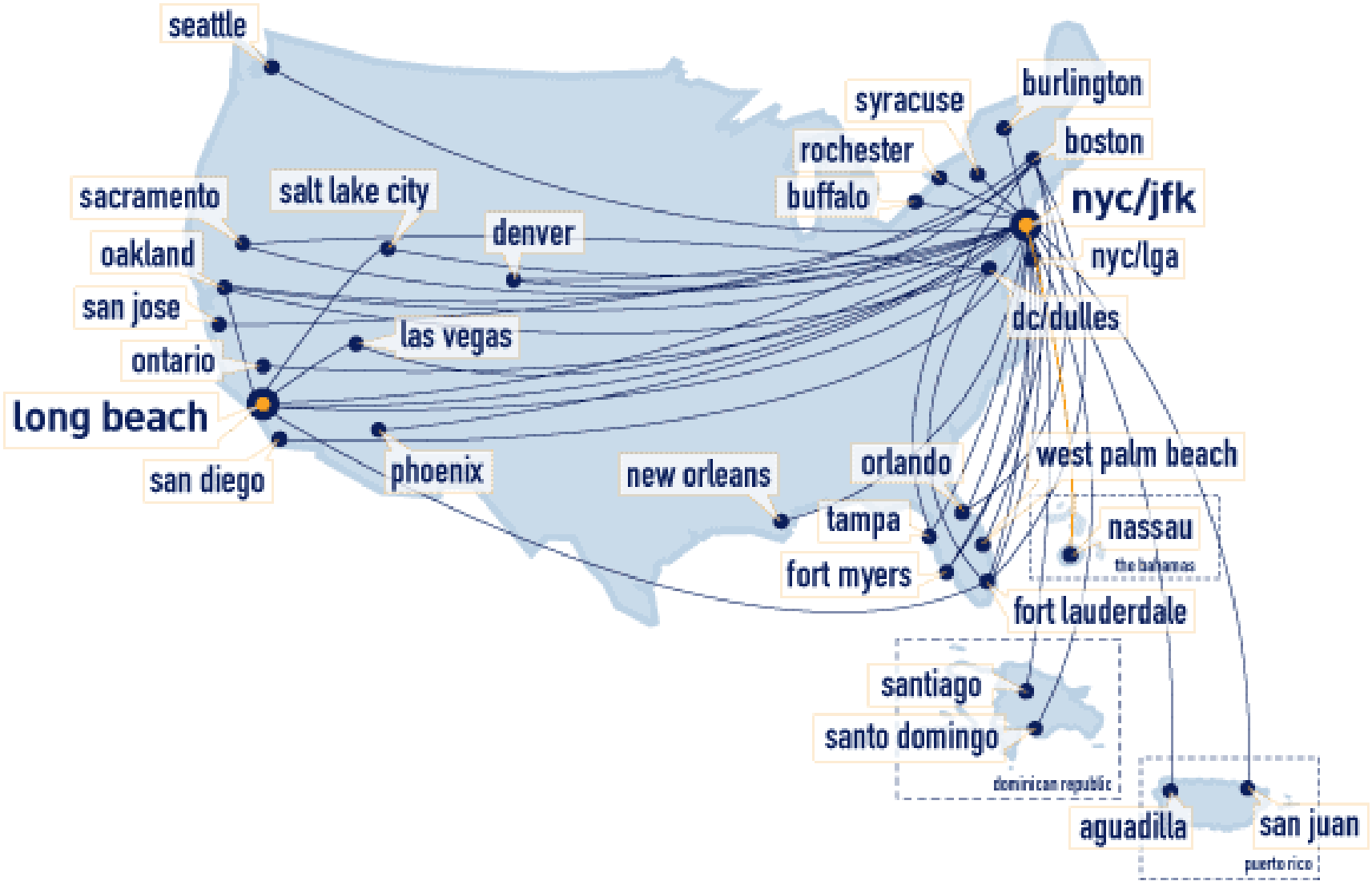
Graphs

Game of Thrones Character Interaction Network



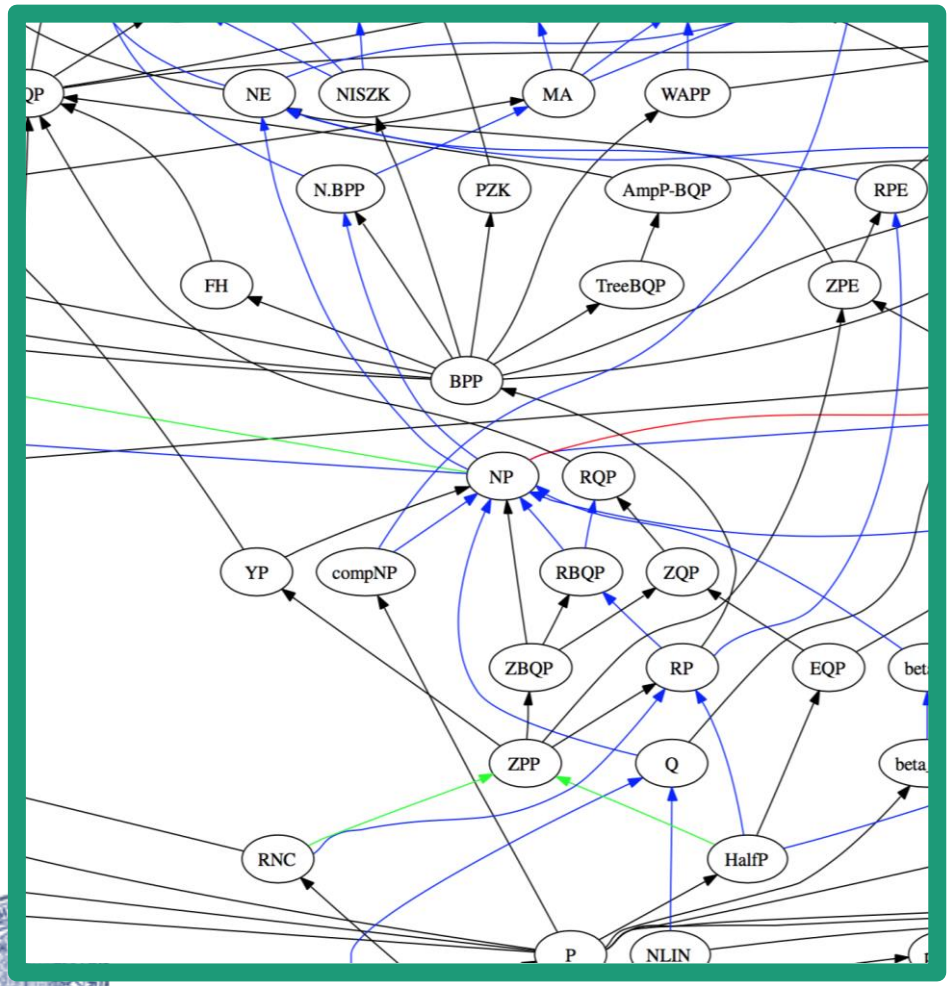
Graphs

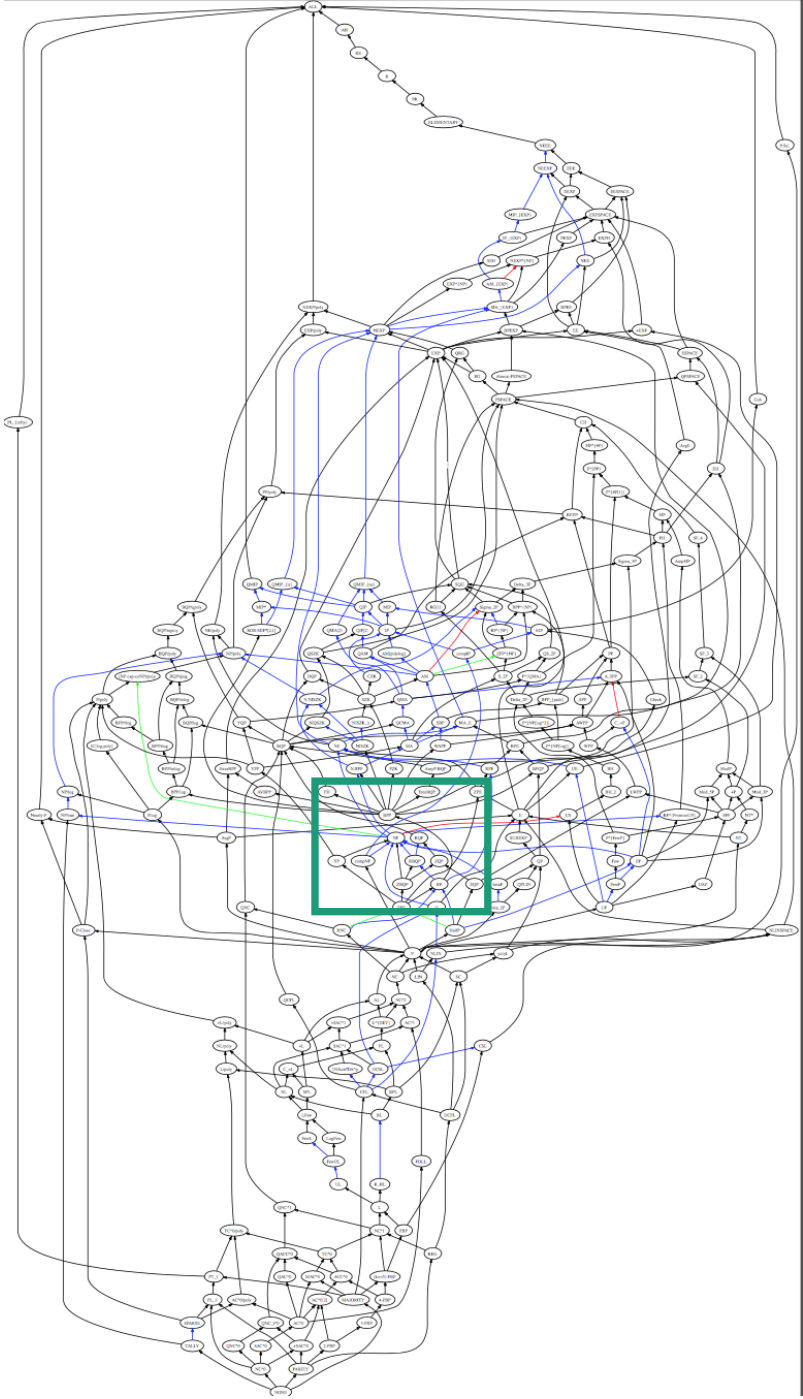
jetblue flights



Graphs

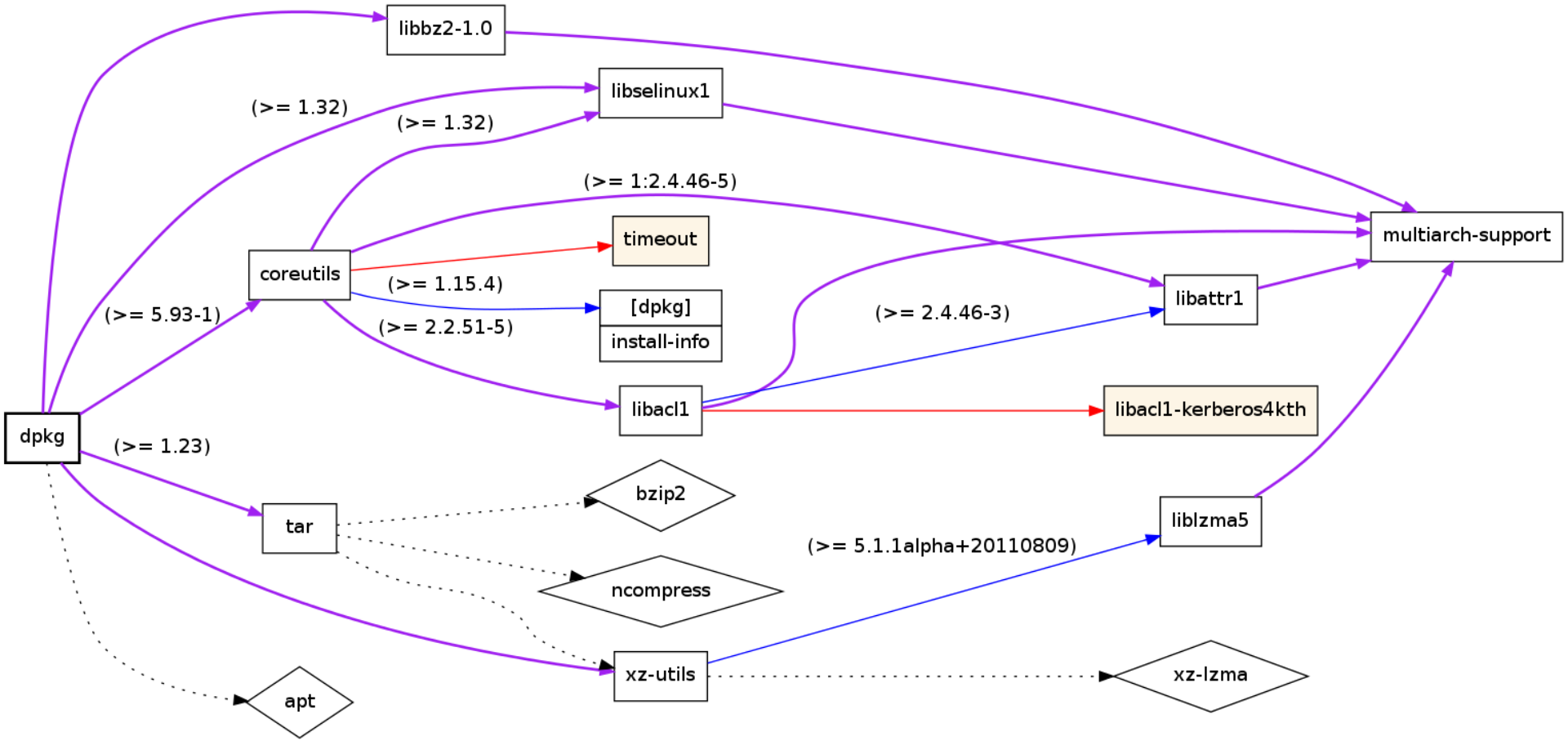
Complexity Zoo
containment graph



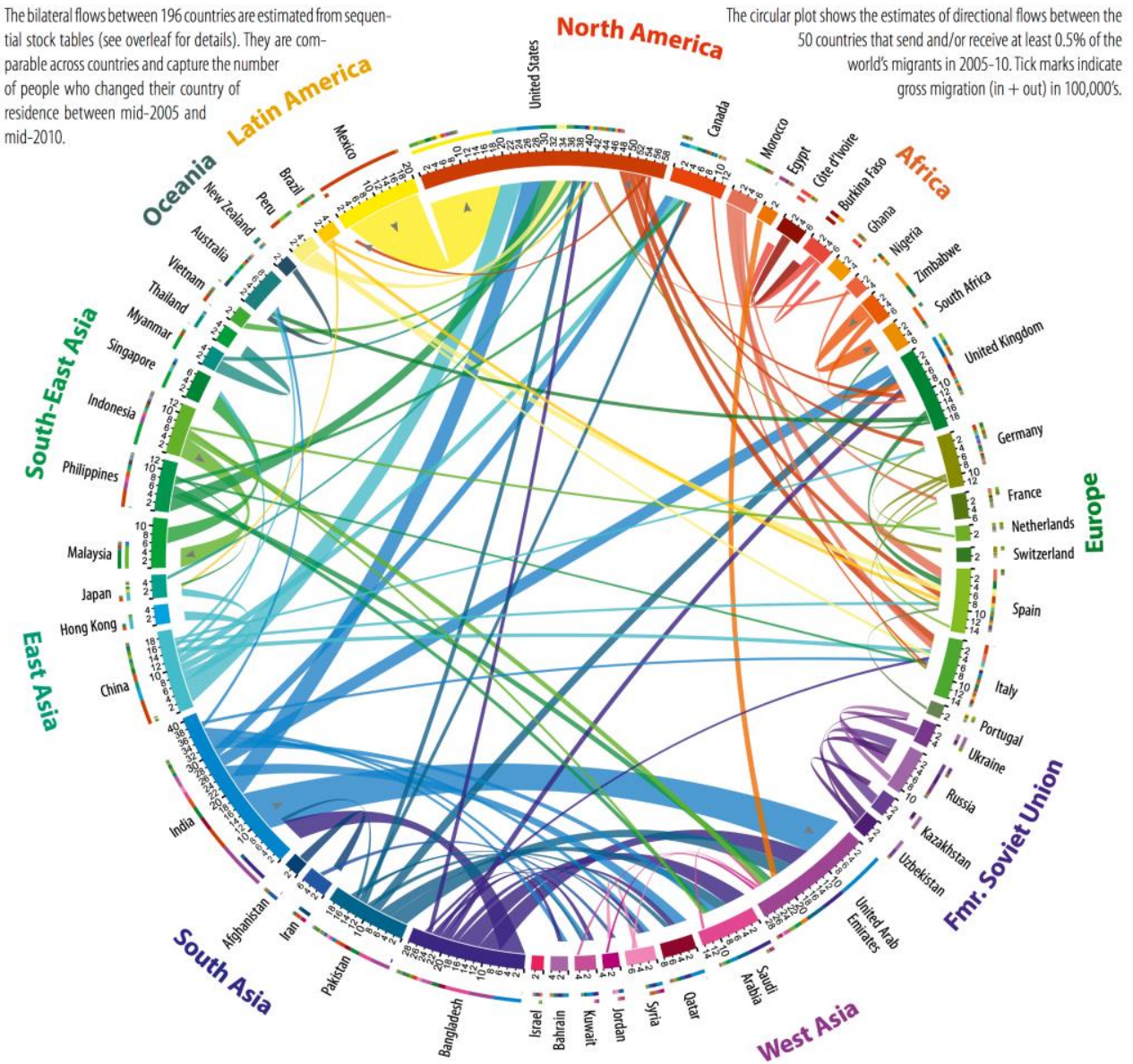


Graphs

debian dependency (sub)graph



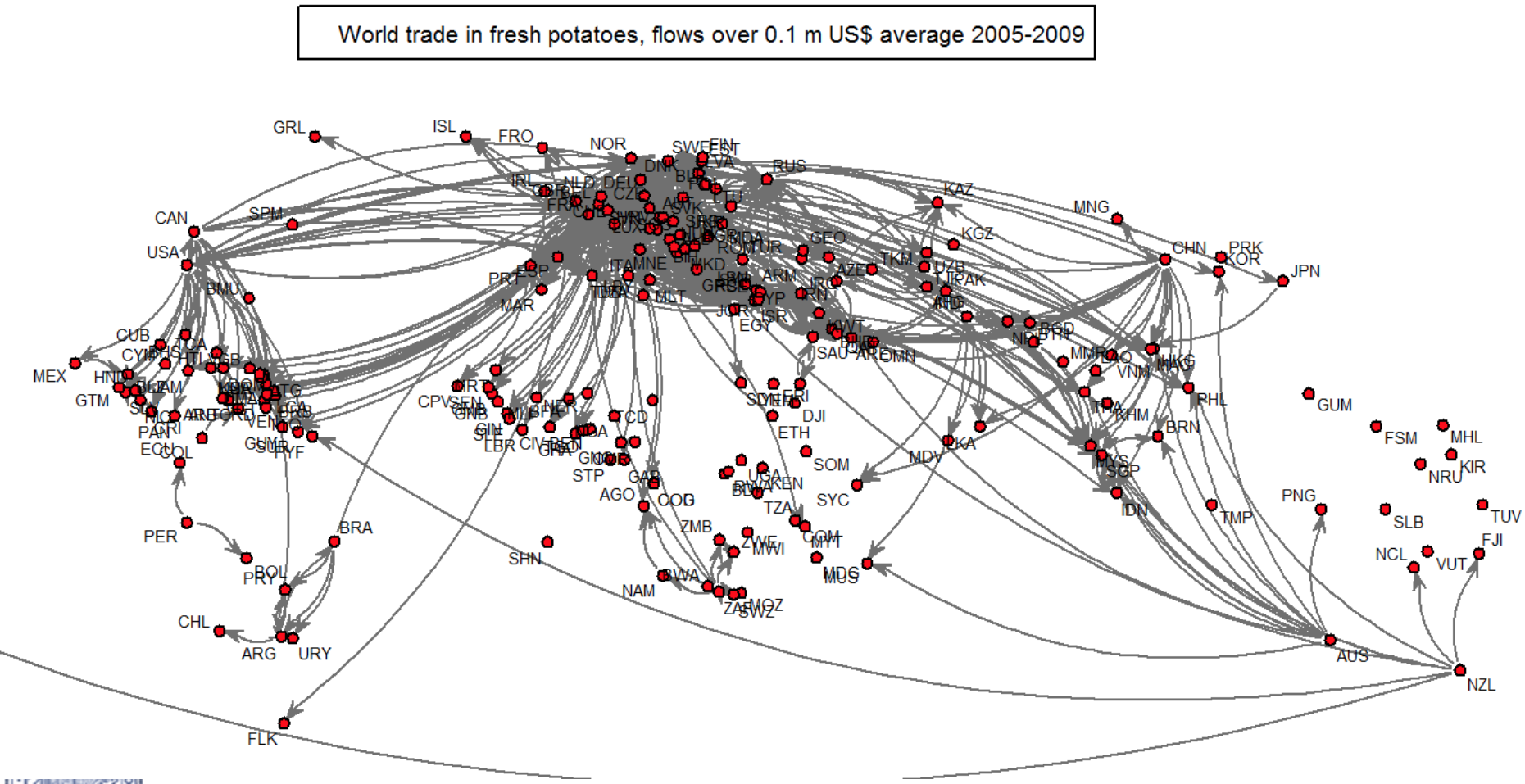
Graphs



Immigration flows

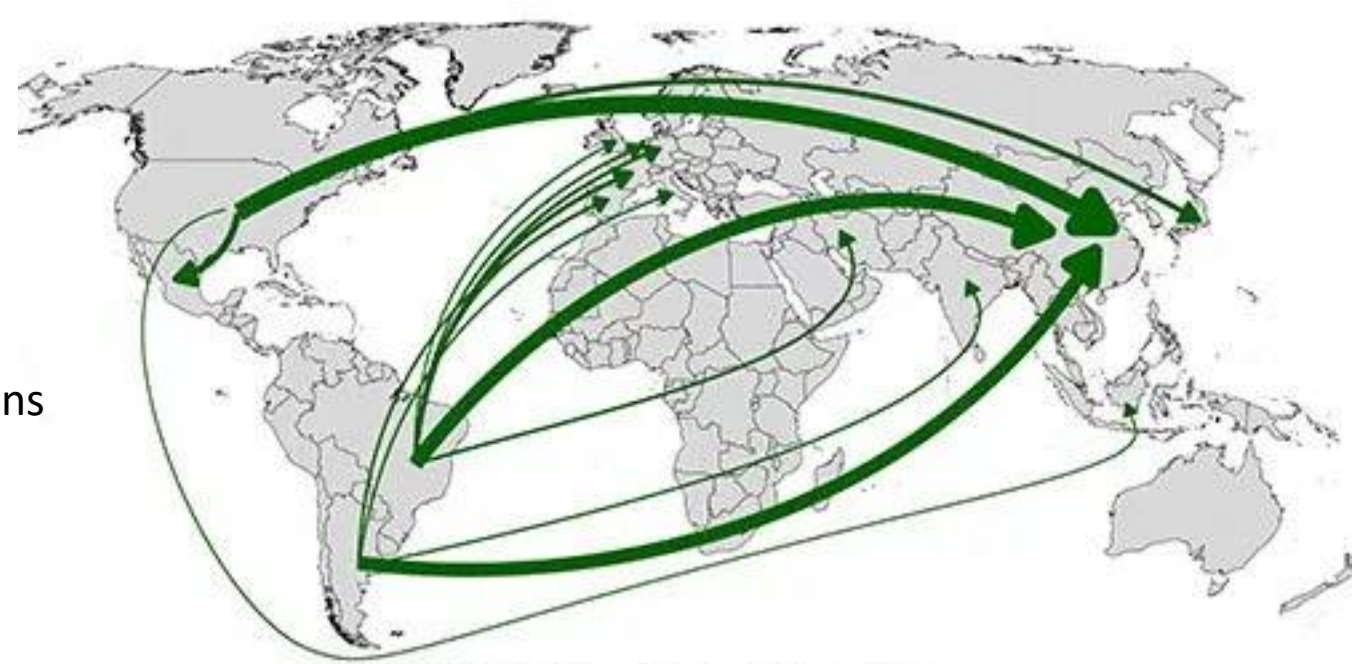
Graphs

Potato trade

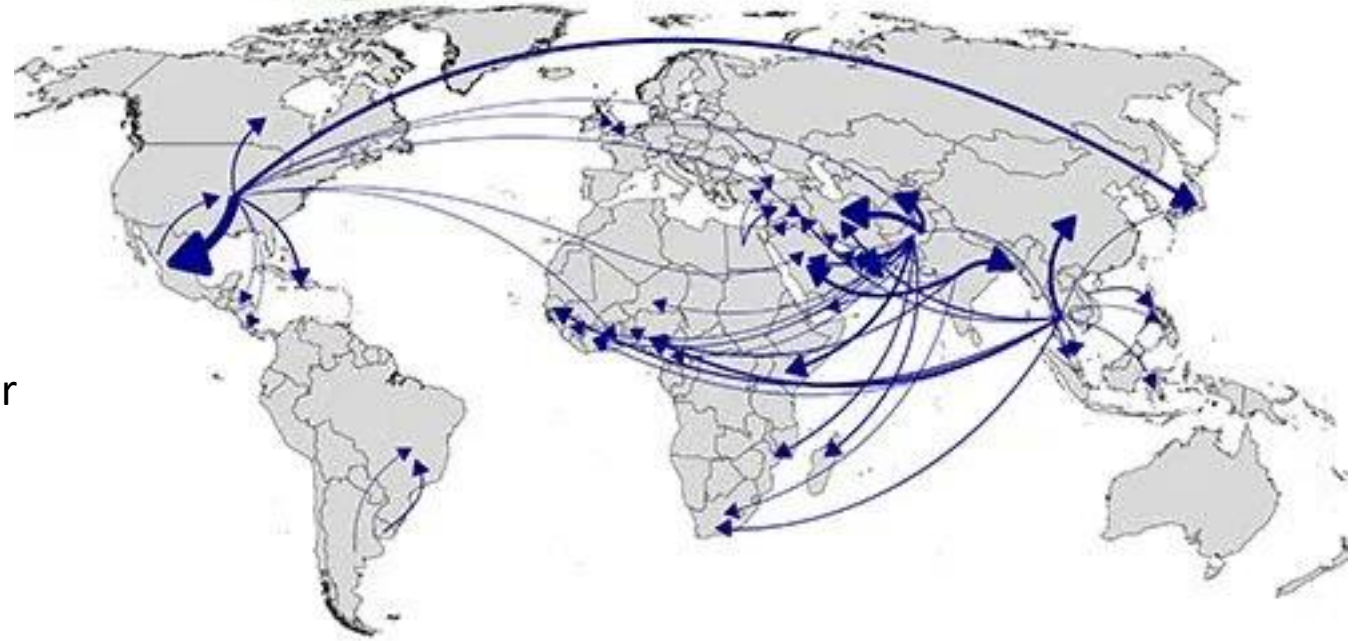


Graphs

Soybeans

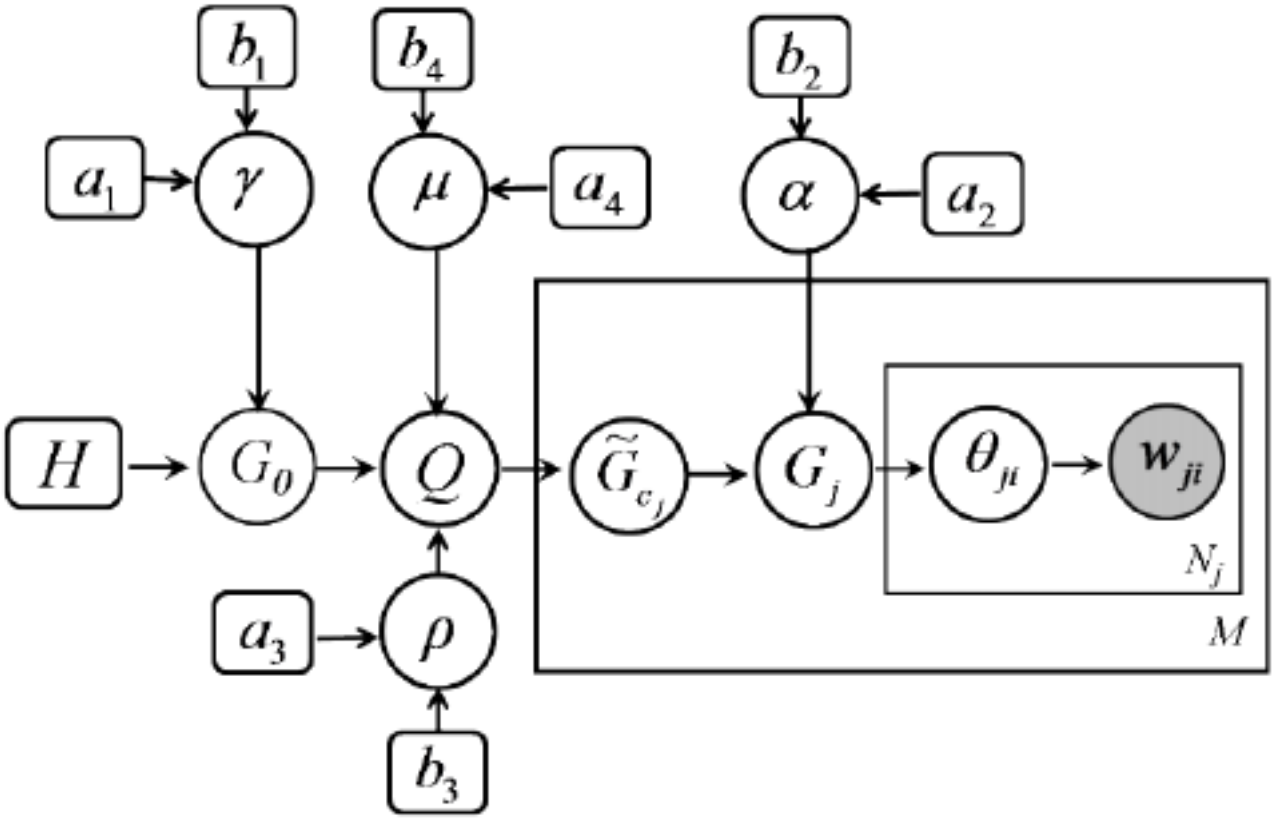


Water



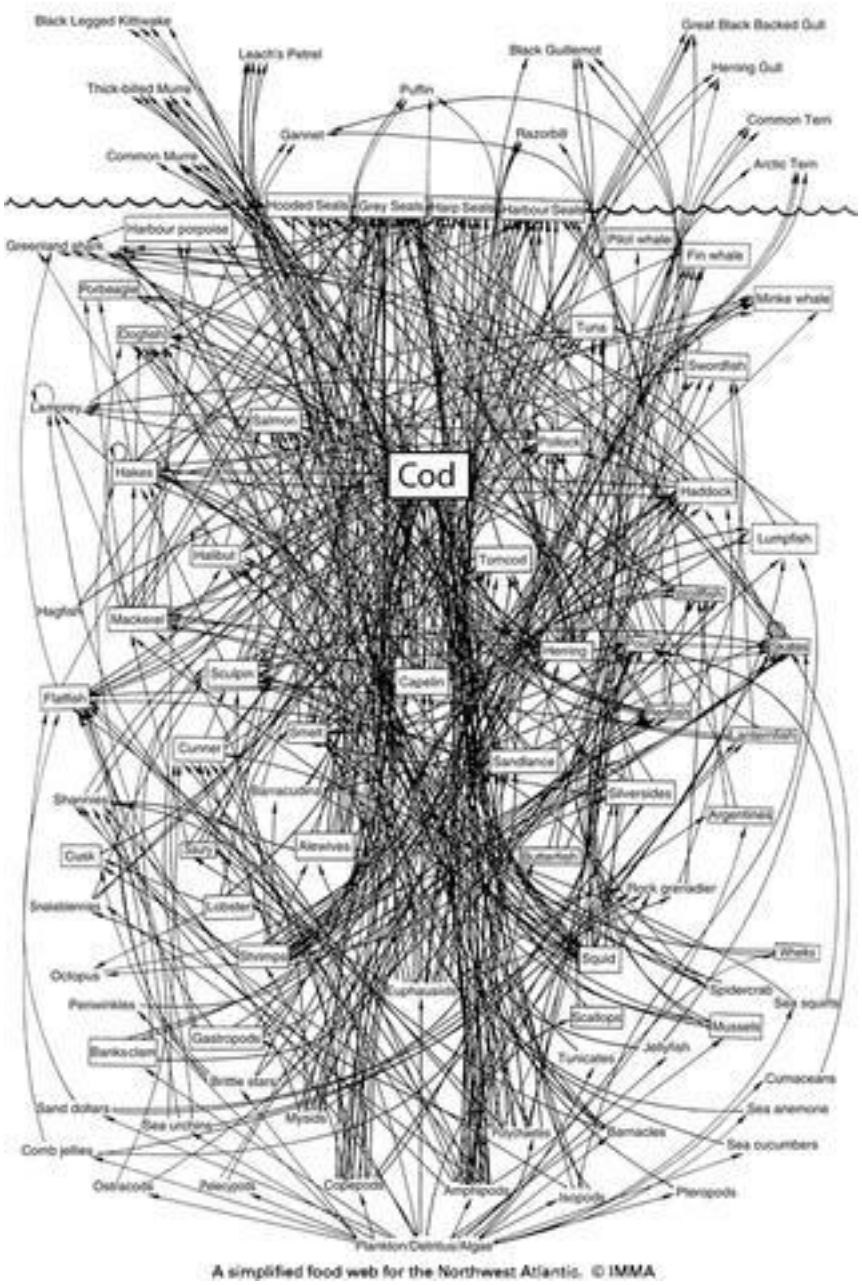
Graphs

Graphical models



Graphs

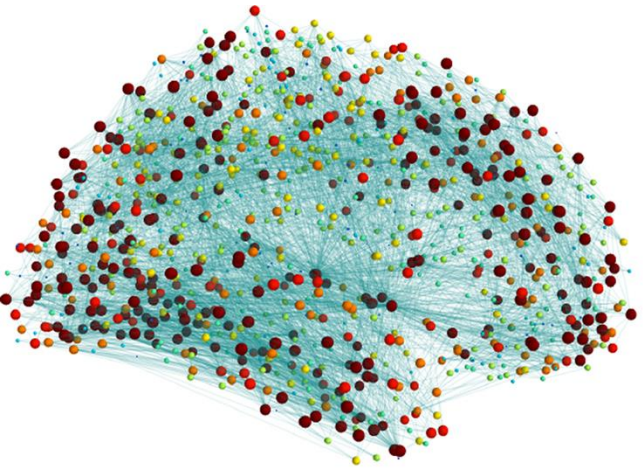
What eats what in the Atlantic ocean?

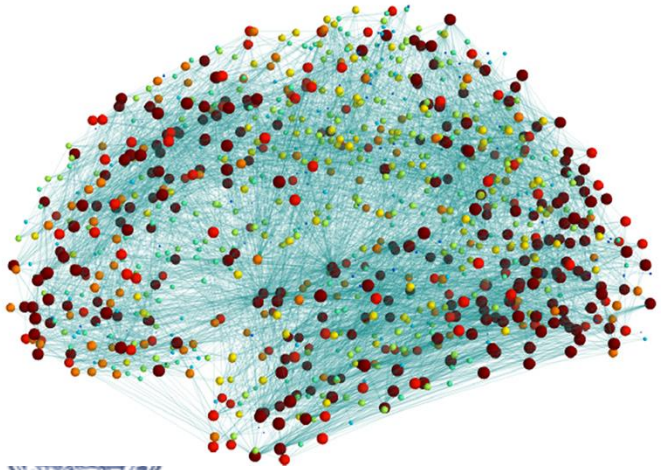


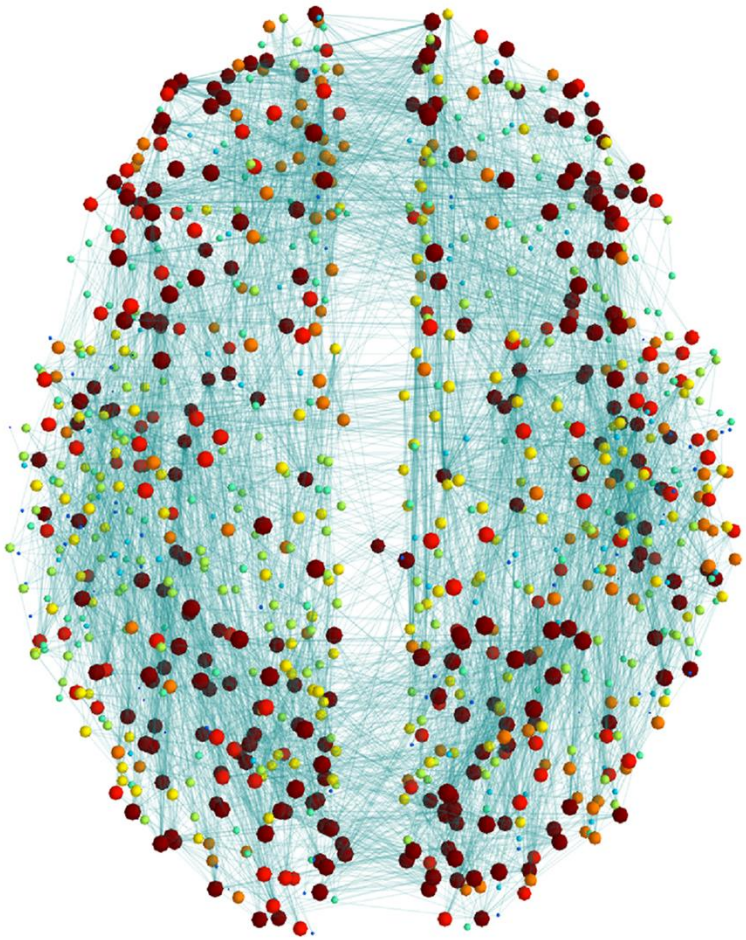
A simplified food web for the Northwest Atlantic. © IMMA

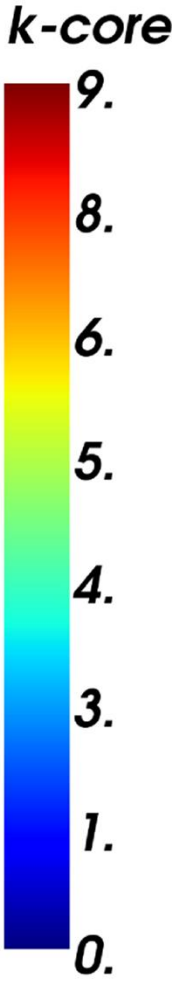
Graphs

Neural connections
in the brain









Graphs

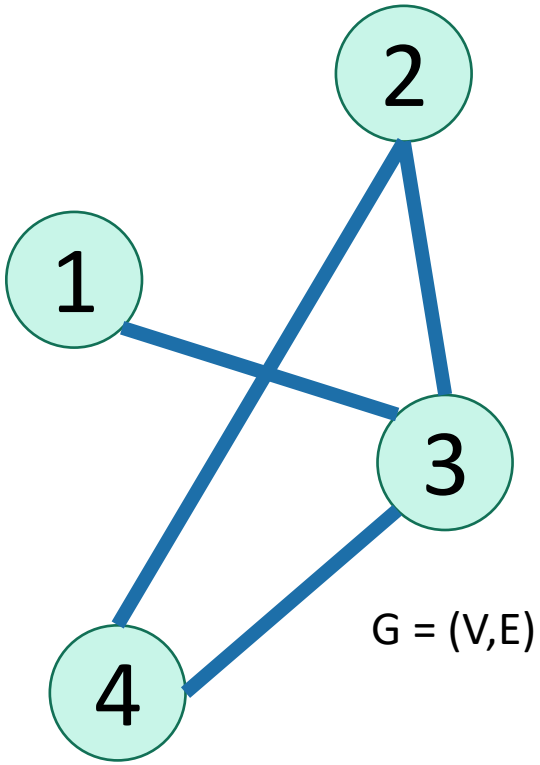
- **There are a lot of graphs.**

- We want to answer questions about them.
  - Efficient routing?
  - Community detection/clustering?
  - Problem examples:
    - Computing (Kevin) Bacon numbers
    - Signing up for classes without violating pre-req constraints
    - How to distribute fish in tanks so that none of them will fight.

- This is what we'll do for the next several lectures.

Undirected Graphs

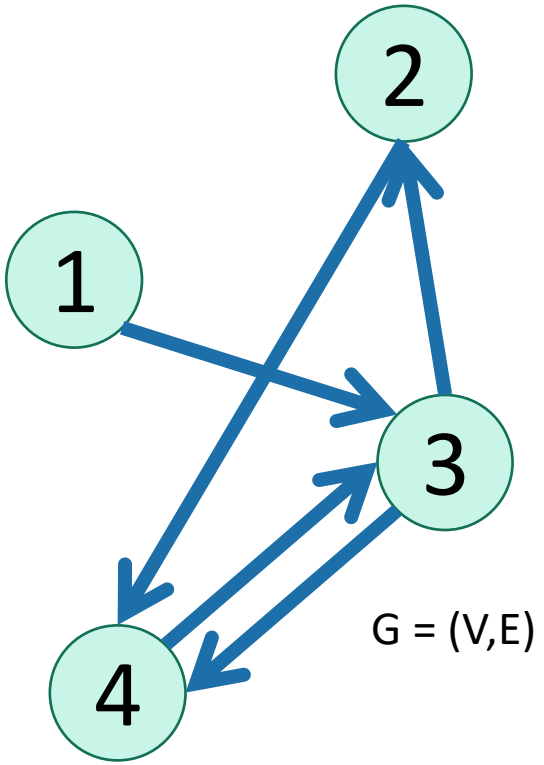
- Has vertices and edges
  - V is the set of vertices
  - E is the set of edges
  - Formally, a graph is G = (V,E)
- Example
  - V = {1,2,3,4}
  - E = { {1,3}, {2,4}, {3,4}, {2,3} }



- The **degree** of vertex 4 is 2.
  - There are 2 edges coming out.
- Vertex 4's **neighbors** are 2 and 3

Directed Graphs

- Has vertices and edges
  - V is the set of vertices
  - E is the set of **DIRECTED** edges
  - Formally, a graph is G = (V,E)
- Example
  - V = {1,2,3,4}
  - E = { (1,3), (2,4), (3,4), (4,3), (3,2) }

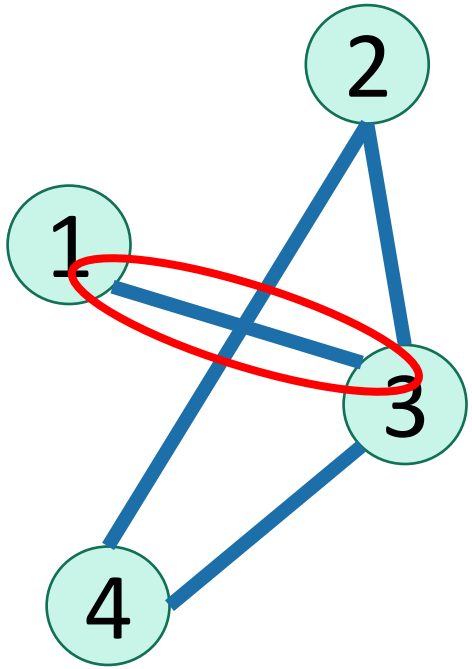


- The **in-degree** of vertex 4 is 2.
- The **out-degree** of vertex 4 is 1.
- Vertex 4's **incoming neighbors** are 2,3
- Vertex 4's **outgoing neighbor** is 3.

How do we represent graphs?

- Option 1: adjacency matrix

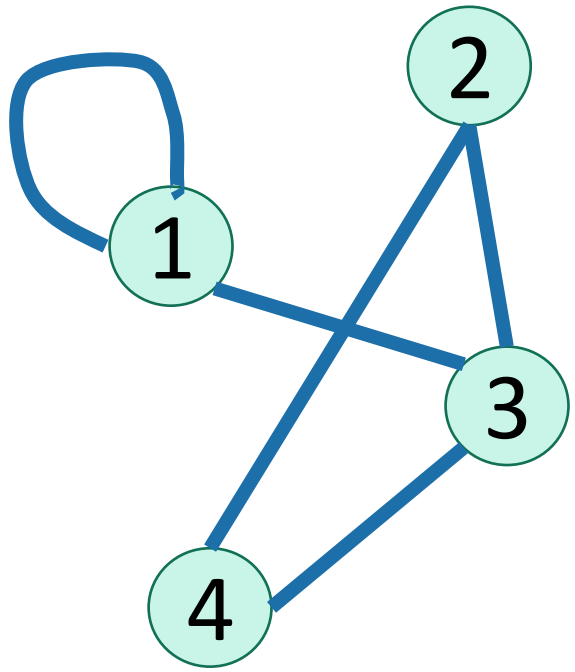
	1	2	3	4
1	0	0	1	0
2	0	0	1	1
3	1	1	0	1
4	0	1	1	0



How do we represent graphs?

- Option 1: adjacency matrix

	1	2	3	4
1	1	0	1	0
2	0	0	1	1
3	1	1	0	1
4	0	1	1	0

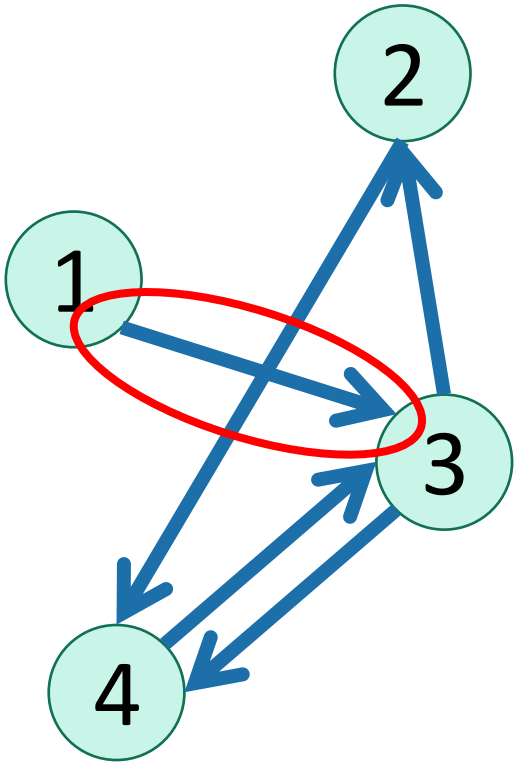


How do we represent graphs?

- Option 1: adjacency matrix

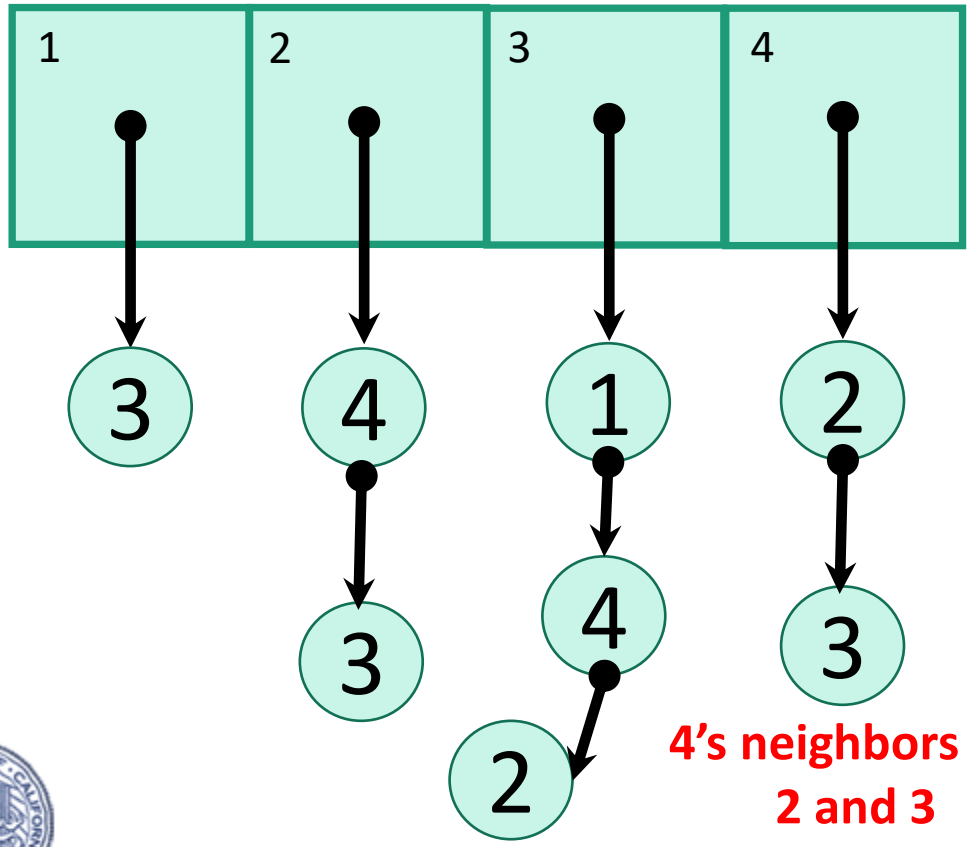
Destination

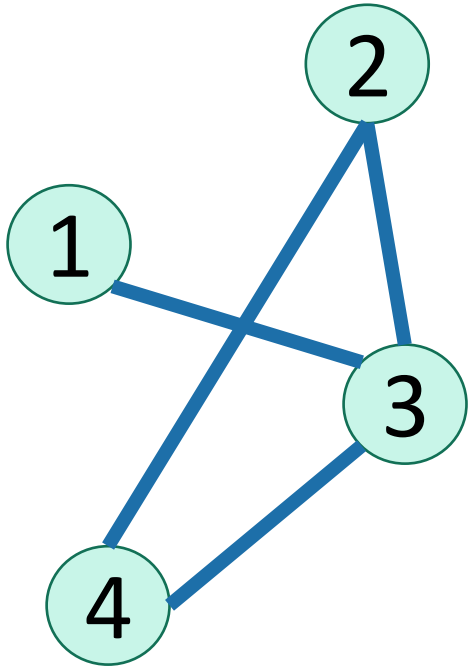
Source	1	2	3	4
1	0	0	1	0
2	0	0	0	1
3	0	1	0	1
4	0	0	1	0



How do we represent graphs?

- Option 2: adjacency lists.





4's neighbors are 2 and 3

How would you modify this for directed graphs?

In either case

- Vertices can store other information
  - Attributes (name, IP address, …)
  - helper info for algorithms that we will perform on the graph

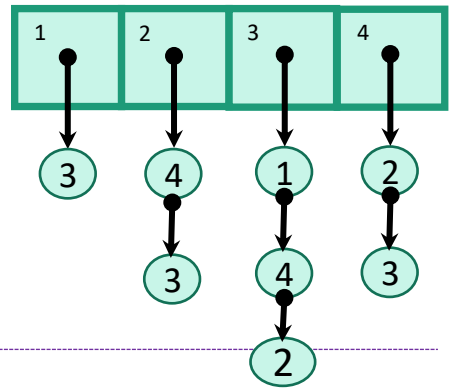
- Want to be able to do the following operations:
  - **Edge Membership**: Is edge e in E?
  - **Neighbor Query**: What are the neighbors of vertex v?

Trade-offs

Say there are n vertices and m edges.

$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Generally better for **sparse** graphs



Edge membership Is e = {v,w} in E?	O(1)	O(deg(v)) or O(deg(w))
Neighbor query Give me v's neighbors.	O(n)	O(deg(v))
Space requirements	$O(n^2)$	O(n + m)

We'll assume this representation for the rest of the class