CSE373: Data Structures & Algorithms
Lecture 13: Hash Collisions

Aaron Bauer

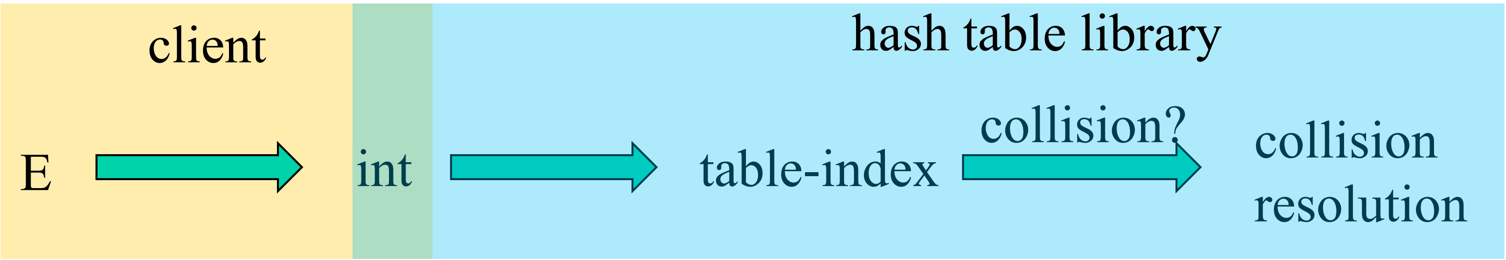
Winter 2014

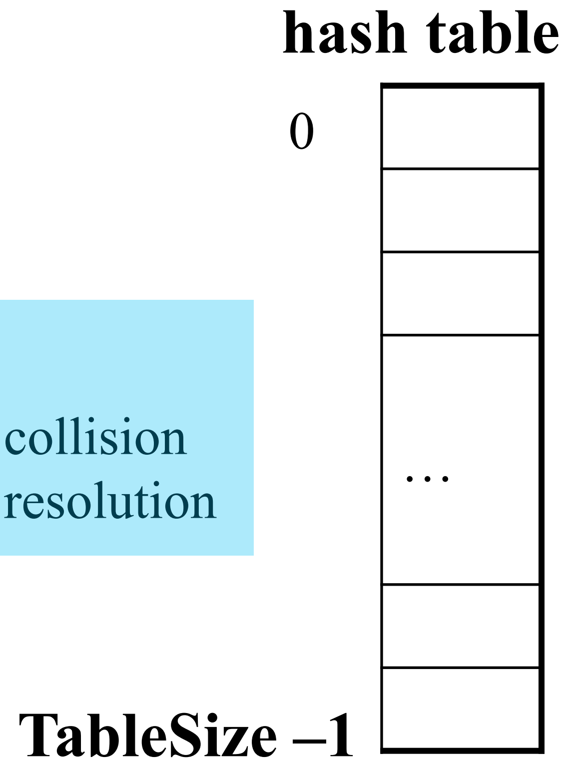
Announcements

- Homework 3 due at 11 p.m. (or later with late days)
- Homework 4 has been posted (due Feb. 20)
  - Can be done with a partner
  - Partner selection due Feb. 12
  - Partner form linked from homework

Hash Tables: Review

- Aim for constant-time (i.e., *O*(1)) **`find`**, **`insert`**, and **`delete`**
  - "On average" under some reasonable assumptions
- A hash table is an array of some fixed size
  - But growable as we'll see





One expert suggestion

- int result = 17;
- foreach field f
  - int fieldHashcode =
    - boolean: (f ? 1: 0)
    - byte, char, short, int: (int) f
    - long: (int) (f ^ (f >>> 32))
    - float: Float.floatToIntBits(f)
    - double: Double.doubleToLongBits(f), then above
    - Object: object.hashCode( )
  - result = 31 * result + fieldHashcode

Collision resolution

Collision:

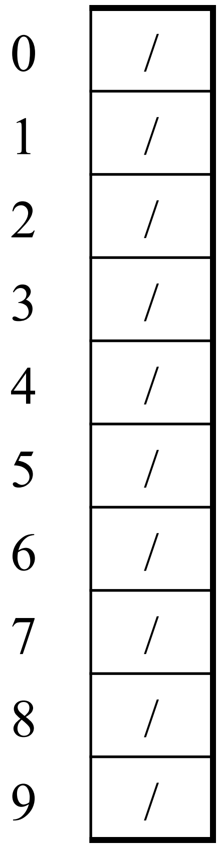
When two keys map to the same location in the hash table

We try to avoid it, but number-of-keys exceeds table size

So hash tables should support collision resolution

- Ideas?

Separate Chaining



Chaining:

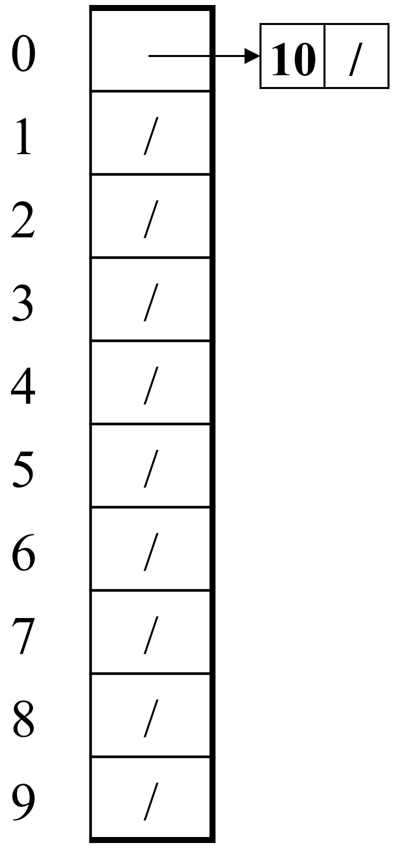
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42
with mod hashing
and **`TableSize`** = 10

Separate Chaining



Chaining:

All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

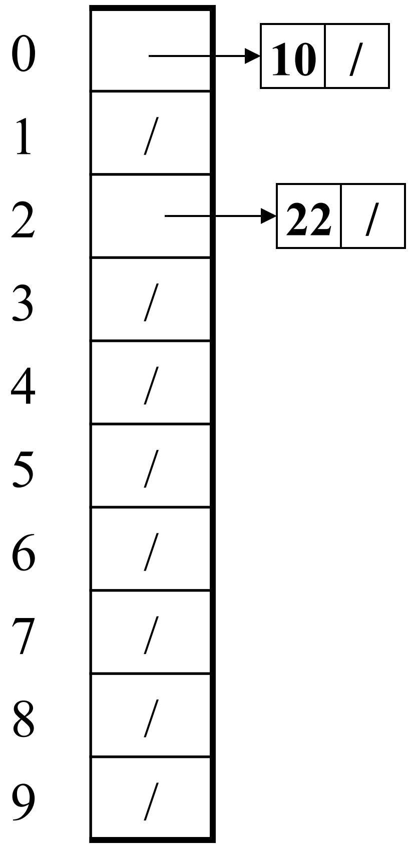
Example:

insert 10, 22, 107, 12, 42

with mod hashing

and **`TableSize`** = 10

Separate Chaining



Chaining:

All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

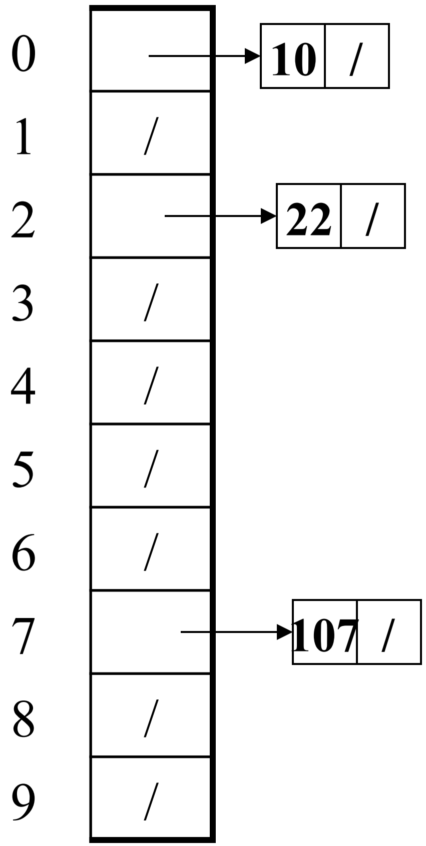
Example:

insert 10, 22, 107, 12, 42

with mod hashing

and **`TableSize`** = 10

Separate Chaining



Chaining:

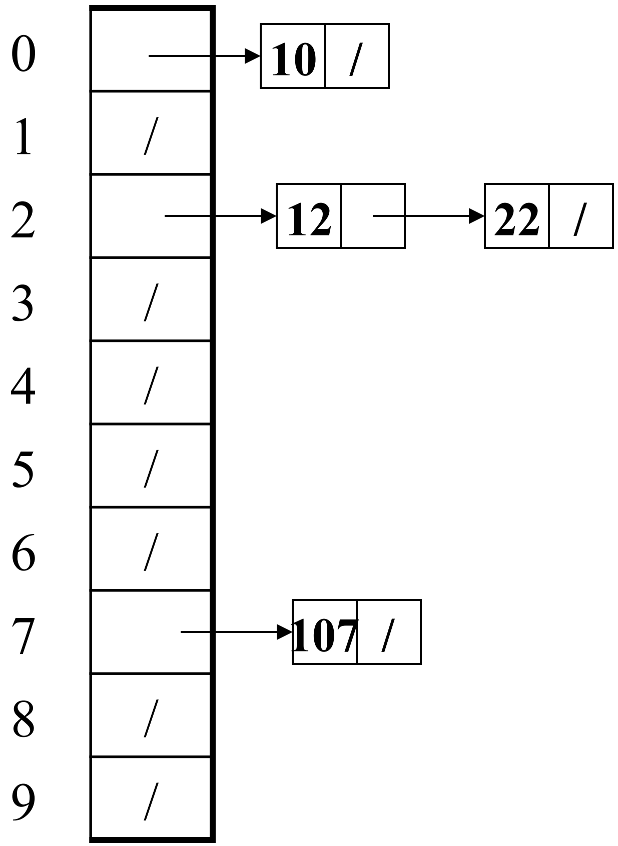
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42
with mod hashing
and **`TableSize`** = 10

Separate Chaining



Chaining:

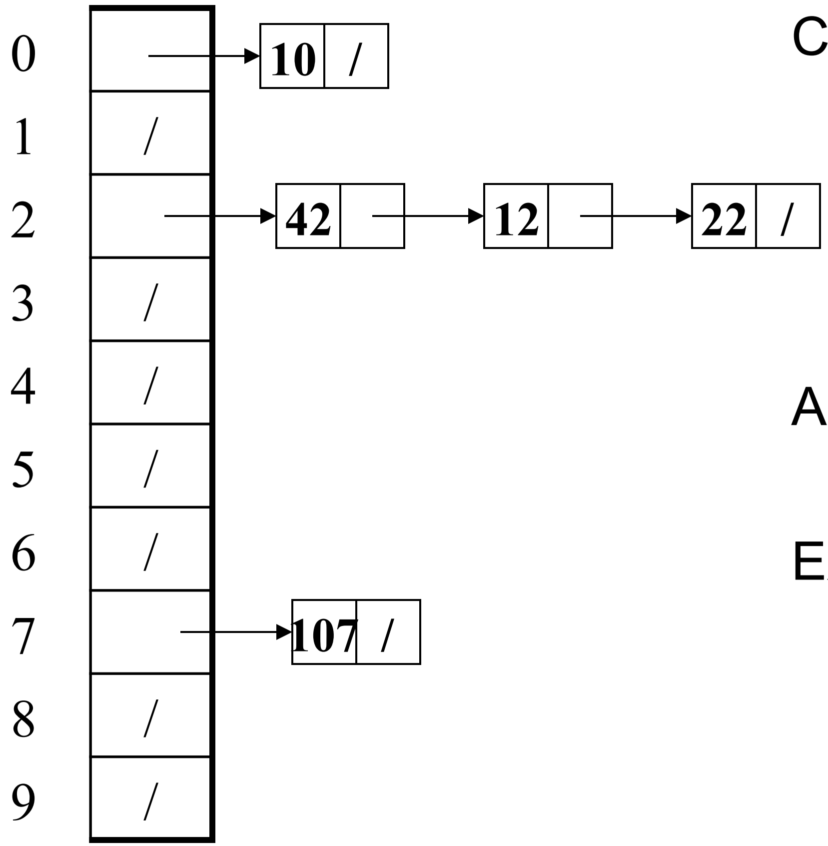
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

insert 10, 22, 107, 12, 42
with mod hashing
and **`TableSize`** = 10

Separate Chaining



Chaining:

All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

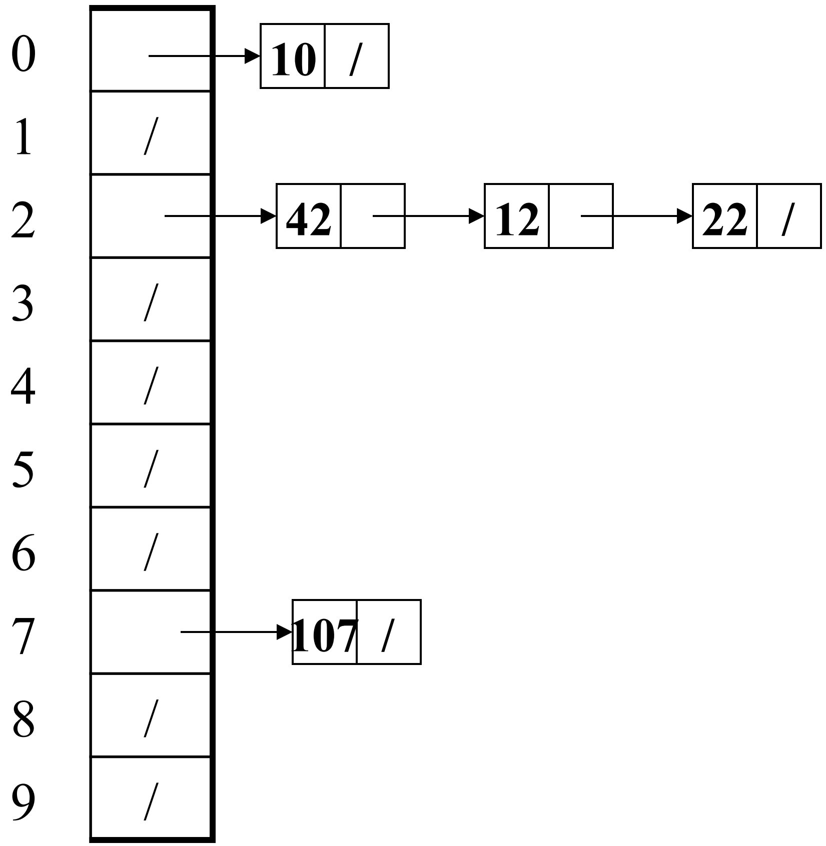
insert 10, 22, 107, 12, 42 with mod hashing and **`TableSize`** = 10

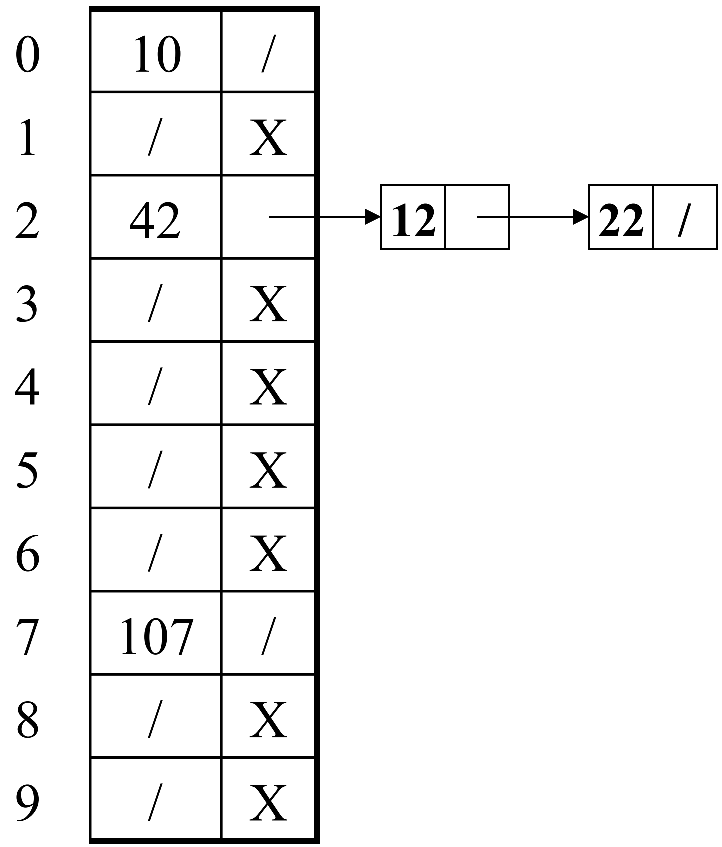
Thoughts on chaining

- Worst-case time for **`find`**?
  - Linear
  - But only with really bad luck or bad hash function
  - So not worth avoiding (e.g., with balanced trees at each bucket)

- Beyond asymptotic complexity, some “data-structure engineering” may be warranted
  - Linked list vs. array vs. chunked list (lists should be short!)
  - Move-to-front
  - Maybe leave room for 1 element (or 2?) in the table itself, to optimize constant factors for the common case
    - A time-space trade-off…

Time vs. space (constant factors only here)





More rigorous chaining analysis

Definition: The load factor, λ, of a hash table is

$$\lambda = \frac{N}{\text{TableSize}} \quad \leftarrow \textbf{number of elements}$$

Under chaining, the average number of elements per bucket is ___

More rigorous chaining analysis

Definition: The load factor, λ, of a hash table is

$$\lambda = \frac{N}{\text{TableSize}} \qquad \leftarrow \textbf{number of elements}$$

Under chaining, the average number of elements per bucket is λ

So if some inserts are followed by *random* finds, then on average:

- Each unsuccessful **`find`** compares against ____ items

More rigorous chaining analysis

Definition: The load factor, λ, of a hash table is

$$\lambda = \frac{\text{N}}{\text{TableSize}} \quad \leftarrow \textbf{number of elements}$$

Under chaining, the average number of elements per bucket is λ

So if some inserts are followed by *random* finds, then on average:

- Each unsuccessful **`find`** compares against λ items
- Each successful **`find`** compares against _____ items

More rigorous chaining analysis

Definition: The load factor, λ, of a hash table is

$$\lambda = \frac{\text{N}}{\text{TableSize}} \quad \leftarrow \textbf{number of elements}$$

Under chaining, the average number of elements per bucket is λ

So if some inserts are followed by *random* finds, then on average:

- Each unsuccessful **`find`** compares against λ items
- Each successful **`find`** compares against $\lambda / 2$ items

So we like to keep λ fairly low (e.g., 1 or 1.5 or 2) for chaining

Alternative: Use empty space in the table

- Another simple idea: If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10

0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	38
9	/

Alternative: Use empty space in the table

- Another simple idea: If `h(key)` is already full,
  - try `(h(key) + 1) % TableSize`. If full,
  - try `(h(key) + 2) % TableSize`. If full,
  - try `(h(key) + 3) % TableSize`. If full…

- Example: insert 38, 19, 8, 109, 10

0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	38
9	19

Alternative: Use empty space in the table

- Another simple idea: If `h(key)` is already full,
  - try `(h(key) + 1) % TableSize`. If full,
  - try `(h(key) + 2) % TableSize`. If full,
  - try `(h(key) + 3) % TableSize`. If full…

- Example: insert 38, 19, 8, 109, 10

0	8
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	38
9	19

Alternative: Use empty space in the table

- Another simple idea: If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10

0	8
1	109
2	/
3	/
4	/
5	/
6	/
7	/
8	38
9	19

Alternative: Use empty space in the table

- Another simple idea: If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…

- Example: insert 38, 19, 8, 109, 10

0	8
1	109
2	10
3	/
4	/
5	/
6	/
7	/
8	38
9	19

Probing hash tables

Trying the next spot is called probing (also called open addressing)

- We just did linear probing
  - **`i`**th probe was **`(h(key) + i) % TableSize`**
- In general have some probe function **`f`** and use **`h(key) + f(i) % TableSize`**

Open addressing does poorly with high load factor λ

- So want larger tables
- Too many probes means no more $O(1)$

Other operations

`insert` finds an open table position using a probe function

What about `find`?
- Must use same probe function to “retrace the trail” for the data
- Unsuccessful search when reach empty position

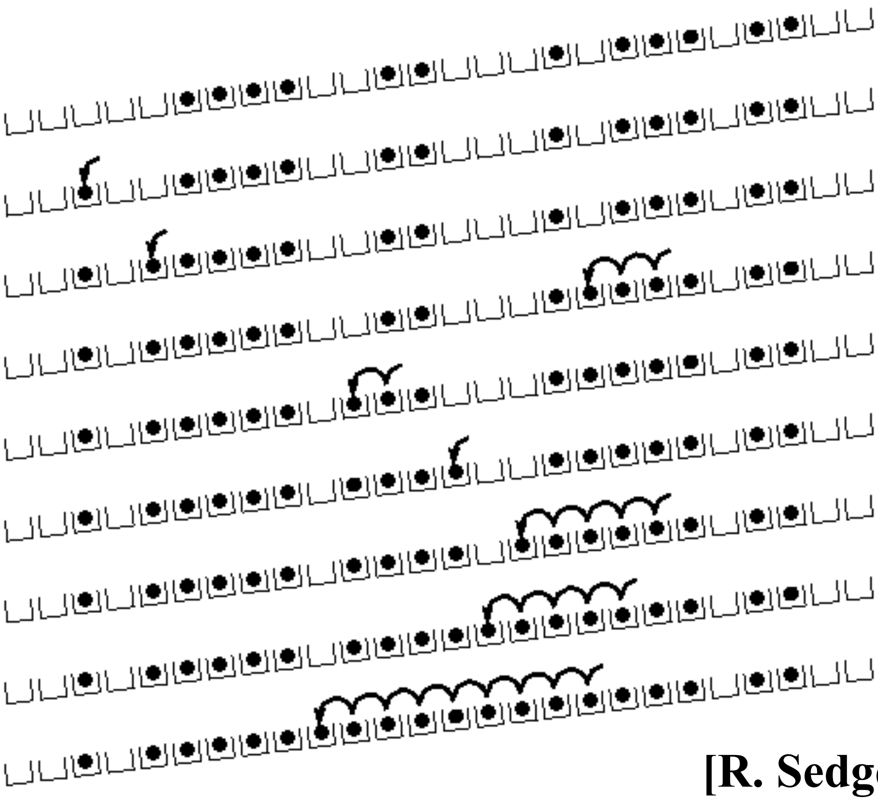
What about `delete`?
- ***Must*** use “lazy” deletion. Why?
  - Marker indicates “no data here, but don’t stop probing”
- Note: `delete` with chaining is plain-old list-remove

(Primary) Clustering

It turns out linear probing is a *bad idea*, even though the probe function is quick to compute (which is a good thing)

Tends to produce *clusters*, which lead to long probing sequences

- Called primary clustering
- Saw this starting in our example



[R. Sedgewick]

Analysis of Linear Probing

- Trivial fact: For any $\lambda < 1$, linear probing will find an empty slot
  - It is "safe" in this sense: no infinite loop unless table is full

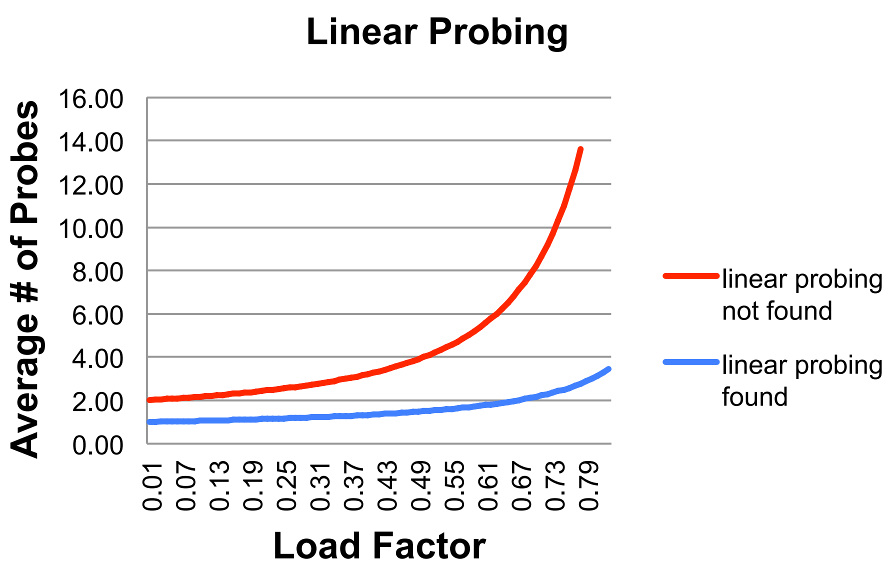
- Non-trivial facts we won't prove:

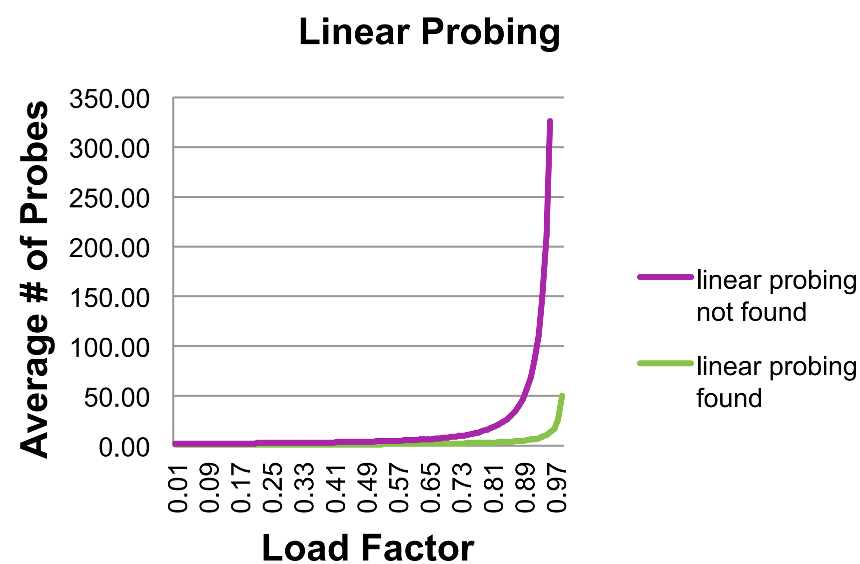
  Average # of probes given λ (in the limit as **`TableSize`** $\rightarrow \infty$)
  - Unsuccessful search: $$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$
  - Successful search: $$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$$

- This is pretty bad: need to leave sufficient empty space in the table to get decent performance (see chart)

In a chart

- Linear-probing performance degrades rapidly as table gets full
  - (Formula assumes "large table" but point remains)





- By comparison, chaining performance is linear in λ and has no trouble with $\lambda > 1$

Quadratic probing

- We can avoid primary clustering by changing the probe function

  `(h(key) + f(i)) % TableSize`

- A common technique is quadratic probing:

  `f(i) =` i^2

  – So probe sequence is:
    - 0^{th} probe: `h(key) % TableSize`
    - 1^{st} probe: `(h(key) + 1) % TableSize`
    - 2^{nd} probe: `(h(key) + 4) % TableSize`
    - 3^{rd} probe: `(h(key) + 9) % TableSize`
    - …
    - i^{th} probe: `(h(key) + i`^{2}`) % TableSize`

- Intuition: Probes quickly "leave the neighborhood"

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

TableSize=10
Insert:
89
18
49
58
79

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

TableSize=10
Insert:
89
18
49
58
79

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

TableSize=10
Insert:
89
18
49
58
79

Quadratic Probing Example

0	49
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

TableSize=10
Insert:
89
18
49
58
79

Quadratic Probing Example

0	49
1	
2	58
3	
4	
5	
6	
7	
8	18
9	89

TableSize=10
Insert:
89
18
49
58
79

Quadratic Probing Example

0	49
1	
2	58
3	79
4	
5	
6	
7	
8	18
9	89

TableSize=10
Insert:
89
18
49
58
79

Another Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	

TableSize = 7

Insert:

76	**(76 % 7 = 6)**
40	**(40 % 7 = 5)**
48	**(48 % 7 = 6)**
5	**(5 % 7 = 5)**
55	**(55 % 7 = 6)**
47	**(47 % 7 = 5)**

Another Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	76

TableSize = 7

Insert:

76	**(76 % 7 = 6)**
40	**(40 % 7 = 5)**
48	**(48 % 7 = 6)**
5	**(5 % 7 = 5)**
55	**(55 % 7 = 6)**
47	**(47 % 7 = 5)**

Another Quadratic Probing Example

0	
1	
2	
3	
4	
5	40
6	76

TableSize = 7

Insert:

76	**(76 % 7 = 6)**
40	**(40 % 7 = 5)**
48	**(48 % 7 = 6)**
5	**(5 % 7 = 5)**
55	**(55 % 7 = 6)**
47	**(47 % 7 = 5)**

Another Quadratic Probing Example

0	48
1	
2	
3	
4	
5	40
6	76

TableSize = 7

Insert:

76	**(76 % 7 = 6)**
40	**(40 % 7 = 5)**
48	**(48 % 7 = 6)**
5	**(5 % 7 = 5)**
55	**(55 % 7 = 6)**
47	**(47 % 7 = 5)**

Another Quadratic Probing Example

0	48
1	
2	5
3	
4	
5	40
6	76

TableSize = 7

Insert:

76	**(76 % 7 = 6)**
40	**(40 % 7 = 5)**
48	**(48 % 7 = 6)**
5	**(5 % 7 = 5)**
55	**(55 % 7 = 6)**
47	**(47 % 7 = 5)**

Another Quadratic Probing Example

0	48
1	
2	5
3	55
4	
5	40
6	76

TableSize = 7

Insert:

76	**(76 % 7 = 6)**
40	**(40 % 7 = 5)**
48	**(48 % 7 = 6)**
5	**(5 % 7 = 5)**
55	**(55 % 7 = 6)**
47	**(47 % 7 = 5)**

Another Quadratic Probing Example

0	48
1	
2	5
3	55
4	
5	40
6	76

TableSize = 7

Insert:

76	**(76 % 7 = 6)**
40	**(40 % 7 = 5)**
48	**(48 % 7 = 6)**
5	**(5 % 7 = 5)**
55	**(55 % 7 = 6)**
47	**(47 % 7 = 5)**

Doh!: For all *n*, `((n*n) +5) % 7 is 0, 2, 5, or 6`

- Excel shows takes "at least" 50 probes and a pattern
- Proof (like induction) using $(n^2+5) \% 7 = ((n-7)^2+5) \% 7$
    - In fact, for all *c* and *k*, $(n^2+c) \% k = ((n-k)^2+c) \% k$

From Bad News to Good News

- Bad news:
  - Quadratic probing can cycle through the same full indices, never terminating despite table not being full
- Good news:
  - If **`TableSize`** is *prime* and $\lambda < ½$, then quadratic probing will find an empty slot in at most **`TableSize/2`** probes
  - So: If you keep $\lambda < ½$ and **`TableSize`** is *prime*, no need to detect cycles
  - Optional: Proof is posted in **`lecture13.txt`**
    - Also, slightly less detailed proof in textbook
    - Key fact: For prime $\mathtt{T}$ and $\mathtt{0 < i,j < T/2}$ where $\mathtt{i} \neq \mathtt{j}$, $\mathtt{(k + i^2) \% T} \neq \mathtt{(k + j^2) \% T}$ (i.e., no index repeat)

Clustering reconsidered

- Quadratic probing does not suffer from primary clustering: no problem with keys initially hashing to the same neighborhood

- But it's no help if keys initially hash to the same index
  - Called secondary clustering

- Can avoid secondary clustering with a probe function that depends on the key: double hashing…

Double hashing

Idea:

- Given two good hash functions *h* and *g*, it is very unlikely that for some *key*, **`h(key) == g(key)`**
- So make the probe function **`f(i) = i*g(key)`**

Probe sequence:

- 0^{th} probe: **`h(key) % TableSize`**
- 1^{st} probe: **`(h(key) + g(key)) % TableSize`**
- 2^{nd} probe: **`(h(key) + 2*g(key)) % TableSize`**
- 3^{rd} probe: **`(h(key) + 3*g(key)) % TableSize`**
- …
- i^{th} probe: **`(h(key) + i*g(key)) % TableSize`**

Detail: Make sure **`g(key)`** cannot be **`0`**

Double-hashing analysis

- Intuition: Because each probe is "jumping" by **`g(key)`** each time, we "leave the neighborhood" *and* "go different places from other initial collisions"

- But we could still have a problem like in quadratic probing where we are not "safe" (infinite loop despite room in table)
  - It is known that this cannot happen in at least one case:
    - **`h(key) = key % p`**
    - **`g(key) = q - (key % q)`**
    - **`2 < q < p`**
    - **`p`** and **`q`** are prime

More double-hashing facts

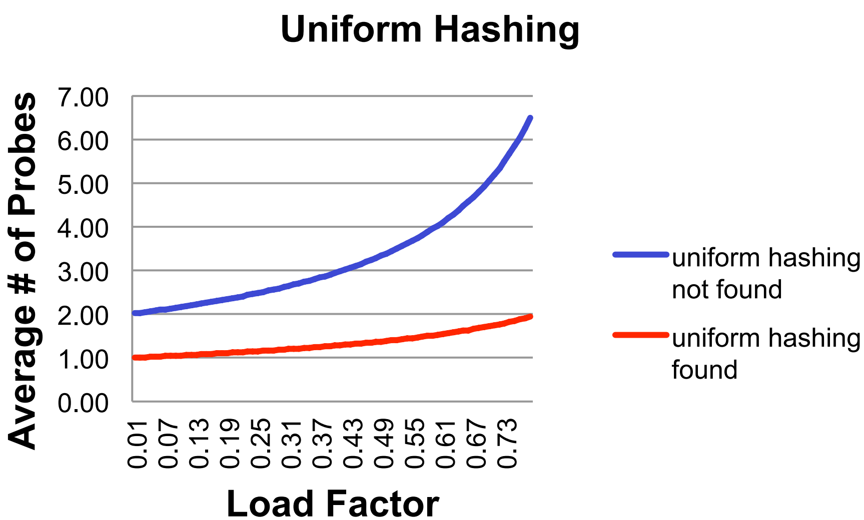
- Assume "uniform hashing"
  - Means probability of `g(key1) % p == g(key2) % p` is `1/p`

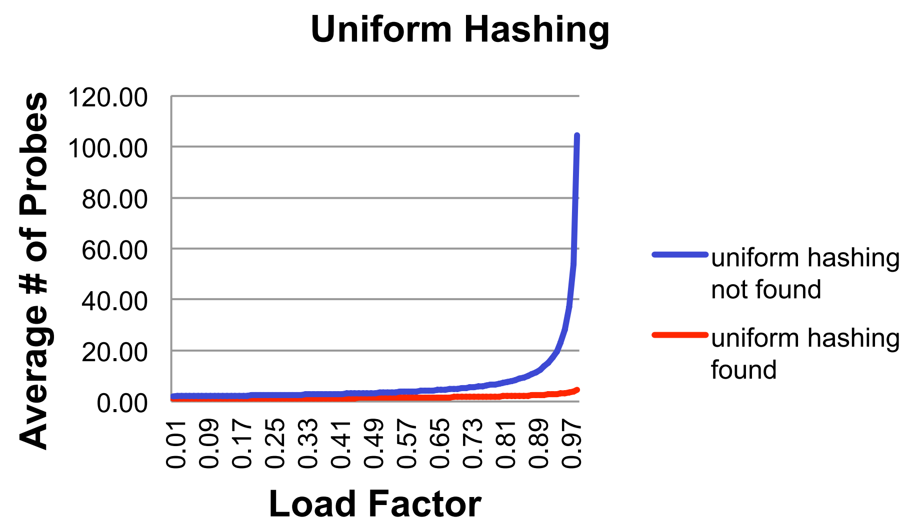
- Non-trivial facts we won't prove:

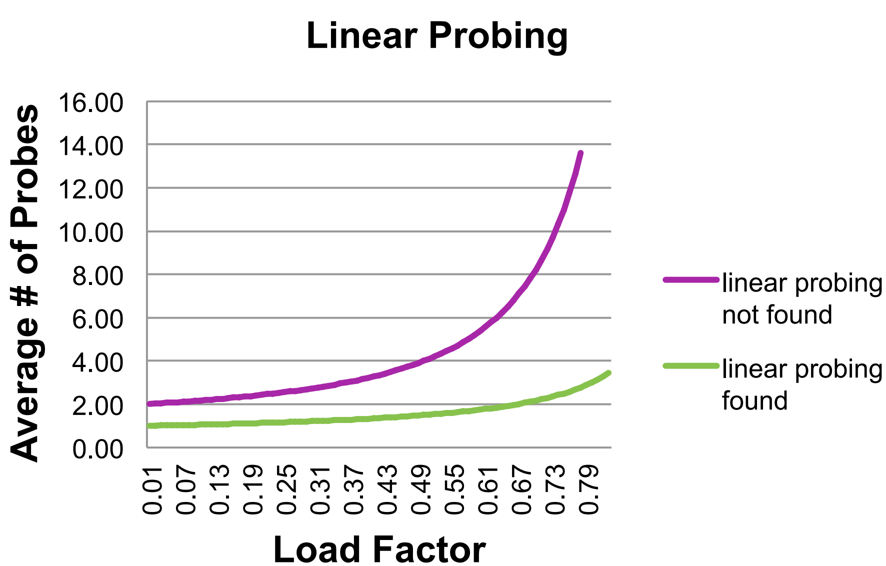
  Average # of probes given λ (in the limit as `TableSize` $\rightarrow \infty$)
  - Unsuccessful search (intuitive): $$\frac{1}{1-\lambda}$$
  - Successful search (less intuitive): $$\frac{1}{\lambda}\log_e\left(\frac{1}{1-\lambda}\right)$$

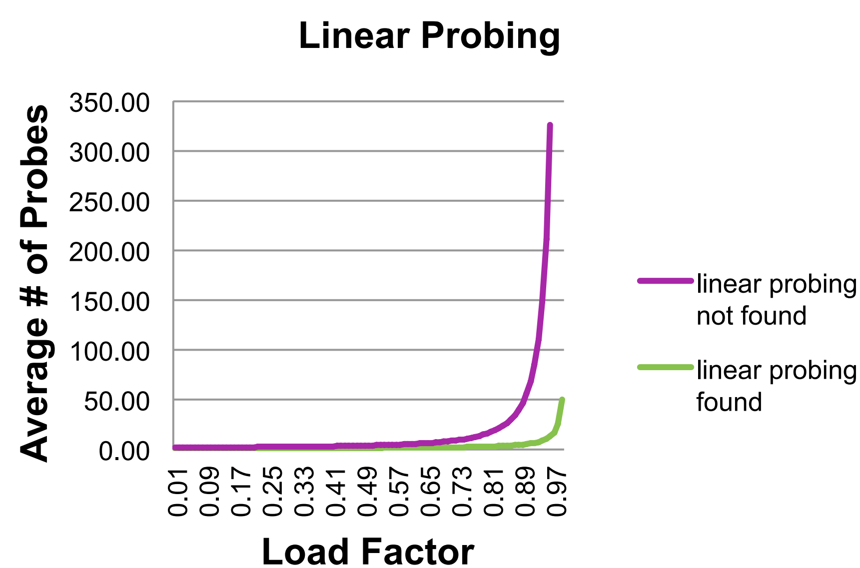
- Bottom line: unsuccessful bad (but not as bad as linear probing), but successful is not nearly as bad

Charts


Uniform Hashing
Average # of Probes
7.00
6.00
5.00
4.00
3.00
2.00
1.00
0.00
0.01 0.07 0.13 0.19 0.25 0.31 0.37 0.43 0.49 0.55 0.61 0.67 0.73
Load Factor
uniform hashing not found
uniform hashing found


Uniform Hashing
Average # of Probes
120.00
100.00
80.00
60.00
40.00
20.00
0.00
0.01 0.09 0.17 0.25 0.33 0.41 0.49 0.57 0.65 0.73 0.81 0.89 0.97
Load Factor
uniform hashing not found
uniform hashing found


Linear Probing
Average # of Probes
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
0.01 0.07 0.13 0.19 0.25 0.31 0.37 0.43 0.49 0.55 0.61 0.67 0.73 0.79
Load Factor
linear probing not found
linear probing found


Linear Probing
Average # of Probes
350.00
300.00
250.00
200.00
150.00
100.00
50.00
0.00
0.01 0.09 0.17 0.25 0.33 0.41 0.49 0.57 0.65 0.73 0.81 0.89 0.97
Load Factor
linear probing not found
linear probing found

Rehashing

- As with array-based stacks/queues/lists, if table gets too full, create a bigger table and copy everything
- With chaining, we get to decide what "too full" means
  - Keep load factor reasonable (e.g., < 1)?
  - Consider average or max size of non-empty chains?
- For probing, half-full is a good rule of thumb
- New table size
  - Twice-as-big is a good idea, except that won't be prime!
  - So go *about* twice-as-big
  - Can have a list of prime numbers in your code since you won't grow more than 20-30 times

Graphs

- A graph is a formalism for representing relationships among items
  - Very general definition because very general concept

- A graph is a pair

  $\mathbf{G = (V,E)}$

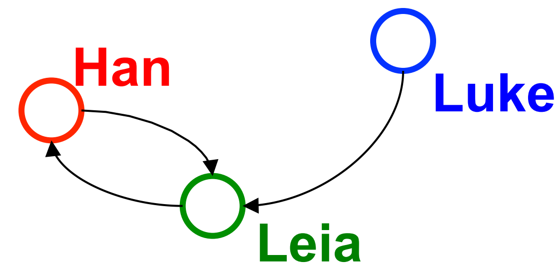
  - A set of vertices, also known as nodes

    $\mathbf{V = \{v_1, v_2, \ldots, v_n\}}$

  - A set of edges

    $\mathbf{E = \{e_1, e_2, \ldots, e_m\}}$

    - Each edge $\mathbf{e_i}$ is a pair of vertices $\mathbf{(v_j, v_k)}$
    - An edge "connects" the vertices



V = {Han,Leia,Luke}

E = {(Luke,Leia),
(Han,Leia),
(Leia,Han)}

- Graphs can be directed or undirected

An ADT?

- Can think of graphs as an ADT with operations like **`isEdge((`v_j`,`v_k`))`**
- But it is unclear what the "standard operations" are
- Instead we tend to develop algorithms over graphs and then use data structures that are efficient for those algorithms
- Many important problems can be solved by:
  1. Formulating them in terms of graphs
  2. Applying a standard graph algorithm
- To make the formulation easy and standard, we have a lot of *standard terminology* about graphs

Some Graphs

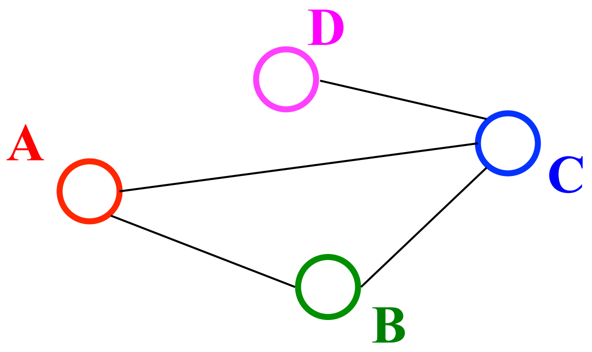
For each, what are the vertices and what are the edges?

- Web pages with links
- Facebook friends
- "Input data" for the Kevin Bacon game
- Methods in a program that call each other
- Road maps (e.g., Google maps)
- Airline routes
- Family trees
- Course pre-requisites
- …

Using the same algorithms for problems across so many domains sounds like "core computer science and engineering"

Undirected Graphs

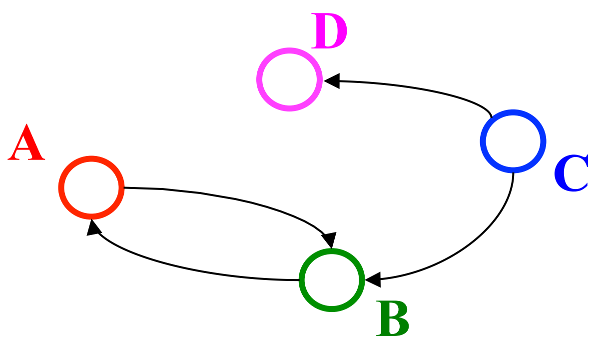
- In undirected graphs, edges have no specific direction
  - Edges are always “two-way”



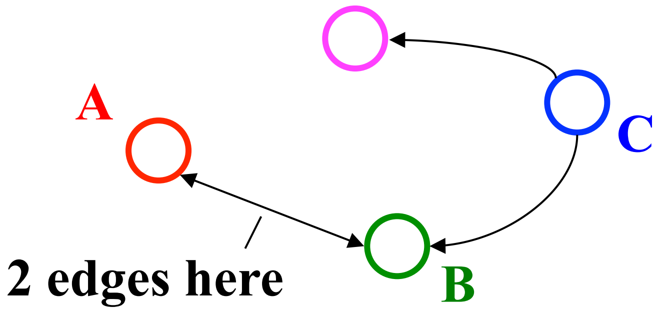
- Thus, **`(u,v)`** $\in$ **`E`** implies **`(v,u)`** $\in$ **`E`**
  - Only one of these edges needs to be in the set
  - The other is implicit, so normalize how you check for it
- Degree of a vertex: number of edges containing that vertex
  - Put another way: the number of adjacent vertices

Directed Graphs

- In directed graphs (sometimes called digraphs), edges have a direction



or

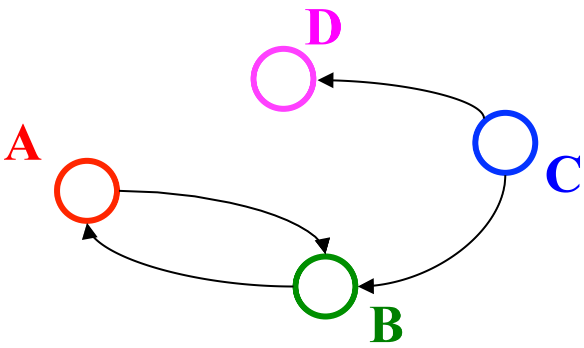


- Thus, **(u,v)** $\in$ **E** does *not* imply **(v,u)** $\in$ **E**.
  - Let **(u,v)** $\in$ **E** mean u → v
  - Call **u** the source and **v** the destination
- In-degree of a vertex: number of in-bound edges, i.e., edges where the vertex is the destination
- Out-degree of a vertex: number of out-bound edges i.e., edges where the vertex is the source

Self-Edges, Connectedness

- A self-edge a.k.a. a loop is an edge of the form **`(u,u)`**
  - Depending on the use/algorithm, a graph may have:
    - No self edges
    - Some self edges
    - All self edges (often therefore implicit, but we will be explicit)
- A node can have a degree / in-degree / out-degree of zero
- A graph does not have to be connected
  - Even if every node has non-zero degree

More notation

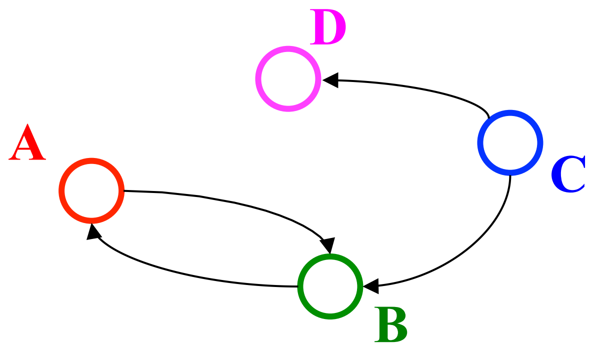


For a graph **`G = (V,E)`**

- **`|V|`** is the number of vertices
- **`|E|`** is the number of edges
  - Minimum?
  - Maximum for undirected?
  - Maximum for directed?

```
V = {A, B, C, D}
E = {(C, B),
     (A, B),
     (B, A)
     (C, D)}
```

More notation

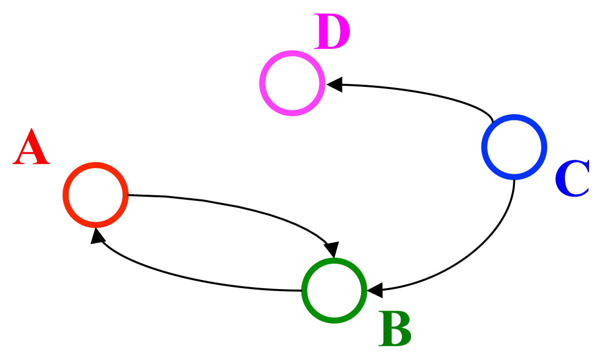


For a graph **`G = (V,E)`**

- **`|V|`** is the number of vertices
- **`|E|`** is the number of edges
  - Minimum? 0
  - Maximum for undirected?
  - Maximum for directed?

```
V = {A, B, C, D}
E = {(C, B),
     (A, B),
     (B, A)
     (C, D)}
```

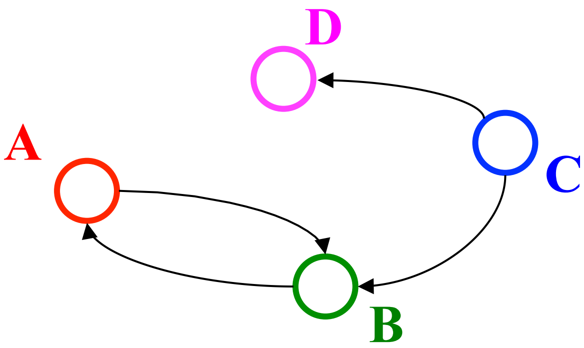
More notation



For a graph **G = (V,E)**

- **|V|** is the number of vertices
- **|E|** is the number of edges
  - Minimum? **0**
  - Maximum for undirected? **|V||V+1|/2** $\in O(|V|^2)$
  - Maximum for directed?

More notation

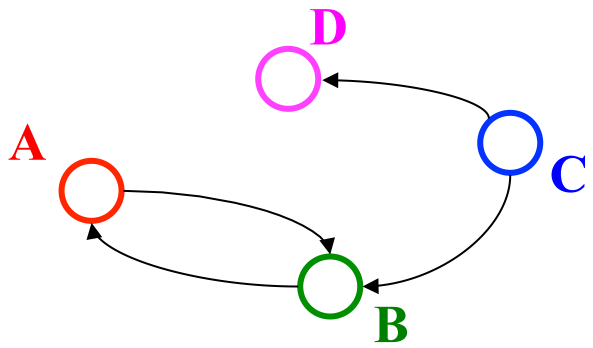


For a graph **G = (V,E)**

- **|V|** is the number of vertices
- **|E|** is the number of edges
  - Minimum? **0**
  - Maximum for undirected? **|V||V+1|/2** $\in$ ***O*(|V|²)**
  - Maximum for directed? **|V|²** $\in$ ***O*(|V|²)**

(assuming self-edges allowed, else subtract **|V|**)

More notation



For a graph `G = (V,E)`:

- `|V|` is the number of vertices
- `|E|` is the number of edges
  - Minimum? `0`
  - Maximum for undirected? `|V||V+1|/2` $\in$ $O(|V|^2)$
  - Maximum for directed? $|V|^2 \in O(|V|^2)$

    (assuming self-edges allowed, else subtract `|V|`)

- If `(u,v)` $\in$ `E`
  - Then **v** is a neighbor of **u**, i.e., **v** is adjacent to **u**
  - Order matters for directed edges
    - **u** is not adjacent to **v** unless `(v,u)` $\in$ `E`