

CSC263 Fall 2015

Data Structures and Analysis

Toniann Pitassi

Email: toni@cs.toronto.edu

BIG THANKS TO: Larry Zhang
(for his amazing slides)

The teaching team

Professor:

→ Toniann Pitassi

Tas:

Daniel Hidru

Dhyey Sejpal

Lalla Mouatadid

Noah Fleming

Majid Komeili

Pan Zhang

Ladislav Rampasek

Robert Robere

Outline for today

Why take CSC263?

What is in CSC263?

How to do well in CSC263?

Why take CSC263?

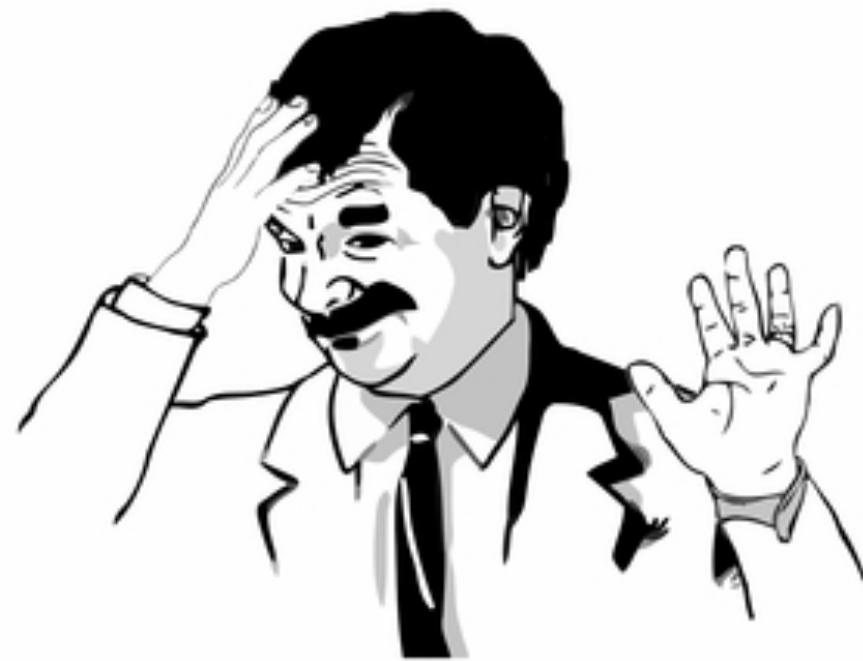
To land a job!

Scenario: The interview

Interviewer: You are given a set of courses, like CSC165, STA247, CSC263, CSC373, where each course stores a list of prerequisites. Devise an algorithm that returns a valid ordering of taking these courses.

You: (think for 1 minute...) Here is my algorithm:

- For a valid ordering to exist, there must be a course **X** that has no prerequisite.
- I choose **X** first, remove **X** from the set of courses, then remove **X** from all other courses' prerequisite list.
- Find the next course in the set that has no prerequisite.
- Repeat this until all courses are removed from the set.



SHUT UP AND GO HOME!

Scenario: The interview, Take 2

Interviewer: You are given a set of courses, like CSC165, STA247, CSC263, CSC373, where each course stores a list of prerequisites. Devise an algorithm that produces a valid ordering of taking these courses.

You: This is a **topological sort** problem which can be solved using **DFS**.



YOU GOT IT

Data structures are smart ways of organizing data, based on which we can develop efficient algorithms easily, in ways that people who don't take CSC263 can't even imagine.

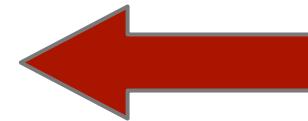
Design algorithms **like a pro.**

*“Bad programmers worry about the code.
Good programmers worry about the data
structures and their relationships.”*

-- Linus Torvalds

What's in CSC263?

(1) Data structures



and

(2) Analysis

What data structures

- Heaps
- Binary search trees
- Balanced search trees
- Hash tables
- Disjoint set forest
- Graphs (matrix, lists)
- ...

What data structures

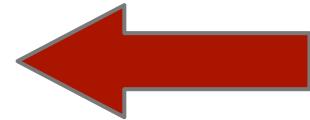
Ways of storing and organizing data to facilitate access and modifications.

We learn the strength and limitations of each data structure, so that we know which one to use.

(1) Data structures

and

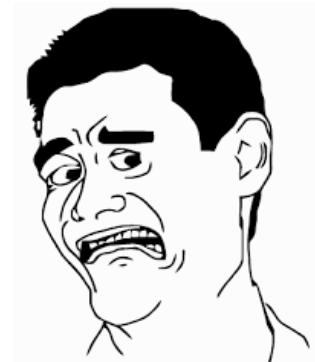
(2) Analysis



What analyses

- Worst-case analysis
- Average-case analysis
- Amortized analysis
- Expected worst-case analysis for randomized algorithms
- ...

Math and proofs



Secret Truth

Data structures are fun to learn,

but **analyses** are more important.

Cooking



A data structure is like a dish.

The analysis is to know the effect of each ingredient.

**Analyses enable you to
invent your own dish.**

Background (Required)

→ Theory of computation

- ◆ Inductions
- ◆ Recursive functions, Master Theorem
- ◆ Asymptotic notations, “big-Oh”

→ Probability theory

- ◆ Probabilities and counting
- ◆ Random variables
- ◆ Distribution
- ◆ Expectations

How to do well in CSC263?

First of all...

Be interested.

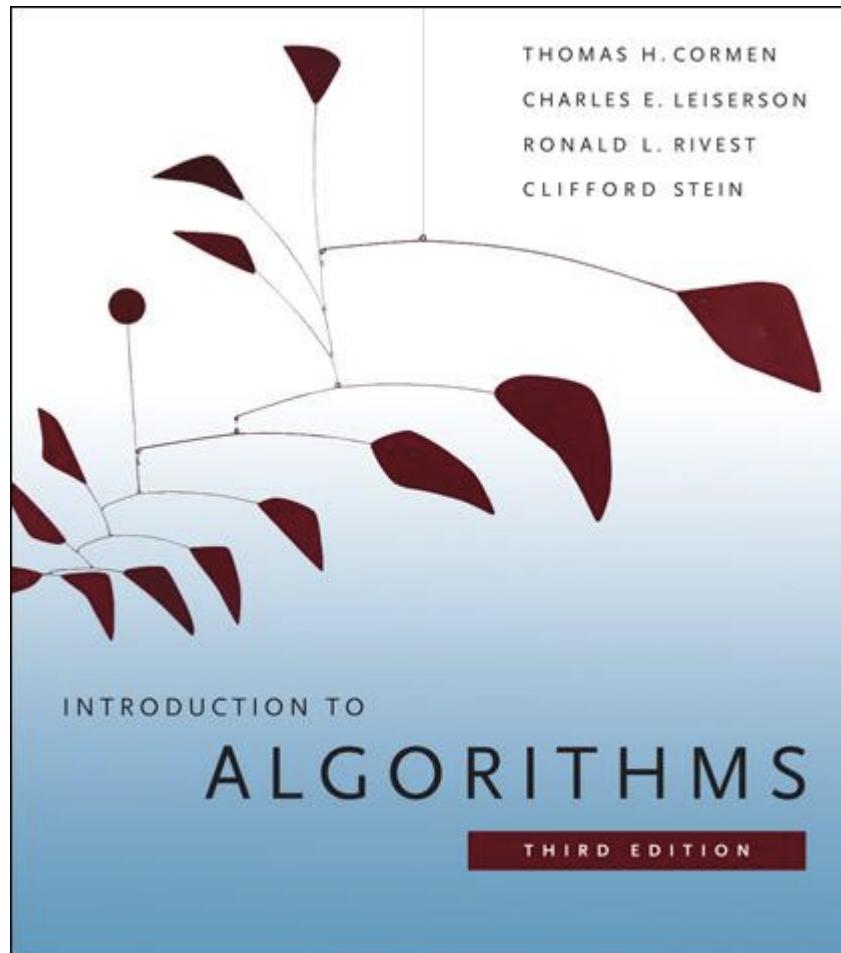
Course Web Page



www.cs.toronto.edu/~toni/Courses/263-2015

Lecture notes / slides (and everything else)
will be posted at the course web page.

Textbook: “CLRS”



Second and third editions are both fine.

Available online at UofT library

Reading for each week is on course info sheet.

Lectures

Exception: No tutorial on
Oct 23, Dec 2

- Wed 10-12 in RW 117
- Wed 2-4 in WI 1016
- Learn stuff

Tutorials

Starting this week!

Practices for homeworks and exams.

Tutorials are as important as lectures.

A tip for lectures and tutorials...

Get involved in classroom **interactions**

- answering a question
- making a guess / bet / vote
- back-of-the envelope calculations

**Emotional involvement makes
the brain remember better!**

I DON'T INTERACT
IN CLASS

I INTERACT
IN CLASS



Course Forum

piazza.com/utoronto.ca/fall2015/csc263h1

Use UToronto email to sign-up.

For discussions among students, instructors will be there answering questions, too. Very helpful.

Communicate intelligently!

Don't discuss homework solutions before due dates.

Office Hours

Wednesdays 12:15pm – 1:45pm

Location: SF2305A

Fridays 2-3pm, Mondays 6-7pm

Location Pratt 266

- They are very helpful and NOT scary at all!
- Statistically, students who go to office hours get higher grades.
- Additional office hours Friday and Mondays before assignments are due (CHECK WEBPAGE!)

Marking Scheme

→5 assignments:	40% = 8% × 5
→1 midterm:	20%
→1 final exam:	40%
→TOTAL	100%

Must get at least 40% on the final to pass.

Assignments (5 of them)

- You may work **individually or in groups of up to 4 students**
- Submissions need to be **typed** into a **PDF** file and submitted to **MarkUS** (link at course web page).
- Due dates are Tuesdays 5:59pm
- Late assignments: You will receive 4 tokens, each worth 6 hours of lateness. If your group submits late, everyone in the group will use up tokens.
- **Collaborate intelligently!** (so that everyone can pass exam)

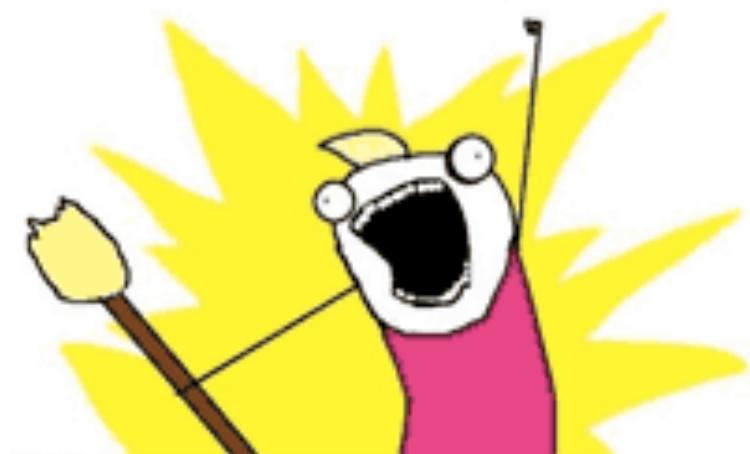
about typing

LaTeX is beautiful and strongly recommended

- We will post our TeX source files, which you can use as templates.
- Many tutorials online. e.g.,
<http://www.maths.tcd.ie/~dwilkins/LaTeXPrimer/>
- Handy tools that do everything in the browser
 - ◆ www.sharelatex.com
 - ◆ www.writelatex.com

Problem Set 1 is out today!

Due Tuesday (Sept 29)



Exams

Midterm:

Thursday, October 22, 8-9pm.

Fill out the form (see course webpage, under Tests) if you have a time conflict, by Sept 25.

Final exam:

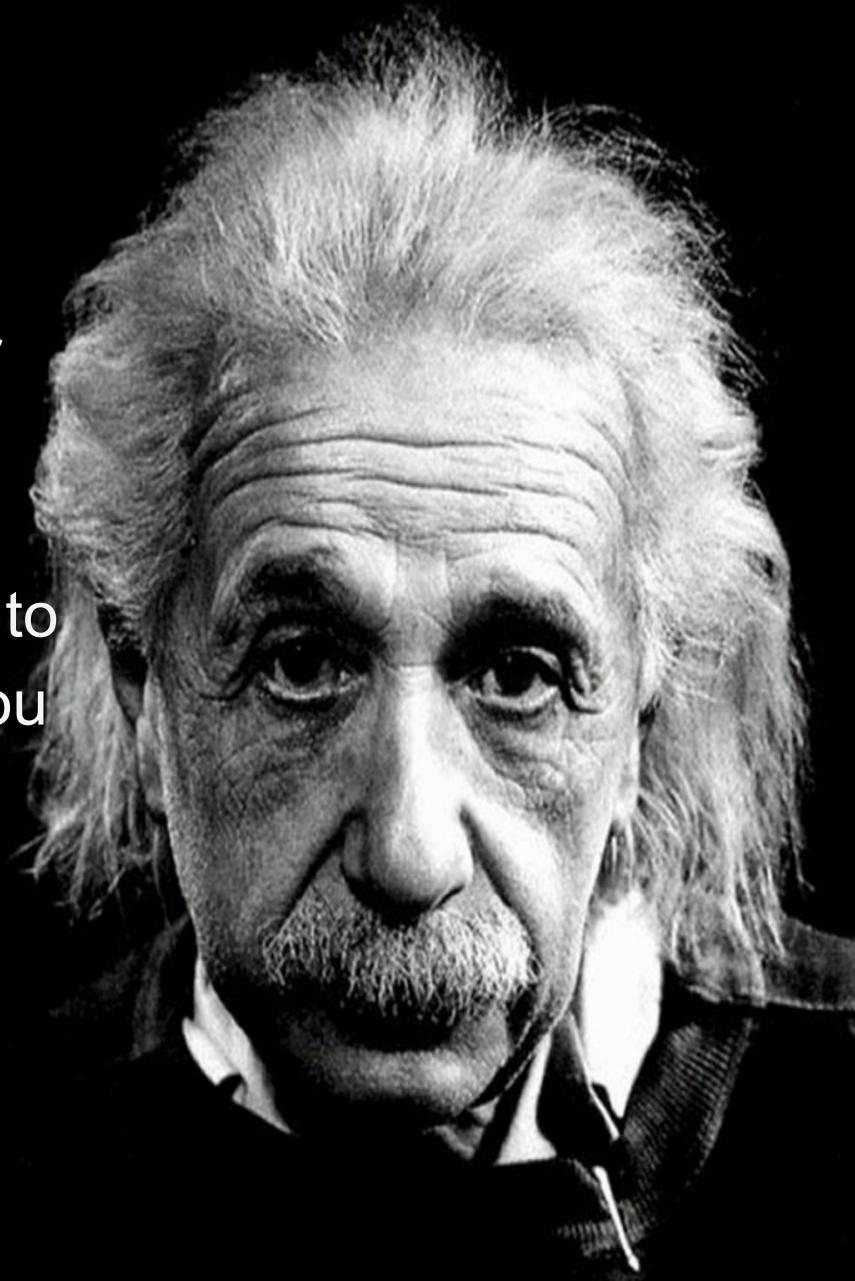
Date to be announced.

Feedback

Feedback at any time is encouraged and appreciated.

- Things you like to have more of.
- Things you want to have less of.
- A topic that you feel difficult to understand.
- Anything else related to the course.

Learn from
yesterday, live for
today, hope for
tomorrow. The
important thing is to
tell people how you
feel, once every
week.



Checklist: How to do well

- Be interested.
- Check course web page regularly.
- Go to lectures.
- Go to tutorials.
- Read textbook and notes.
- Discuss on Piazza.
- Feel free to go to office hours.
- Work on homeworks, and submit on time.
- Do well in exams.

I'M NOT TELLING
YOU IT'S GOING
TO BE EASY,
I'M TELLING YOU
IT'S GOING TO BE
WORTH IT.

Abstract Data Types (ADT) and Data Structures

Two related but different concepts

In short,

→ ADT describes **what** (the **interface**)

- ◆ what data is stored
- ◆ what operation is supported

→ Data structure describes **how** (the **implementation**)

- ◆ how the data is stored
- ◆ how to perform the operations

Real-life example

ADT

- It stores ice cream.
- Supported operations:
 - ◆ start getting ice cream
 - ◆ stop getting ice cream



Data structures

- How ice cream is stored.
- How are the start and stop operations implemented.
- It's the inside of the machine

A CS example

Stack is an **ADT**

- It stores a list of elements
- supports PUSH(S, v), POP(S), IS_EMPTY(S)

Data structures that can used to implement **Stack**

- Linked list
 - ◆ PUSH: insert at head of the list
 - ◆ POP: remove at head of the list (if not empty)
 - ◆ IS_EMPTY: return “head == None”
- Array with a counter (for size of stack) also works

In CSC263, we will learn many ADTs and many data structures for implementing these ADTs.

Review:

Algorithm Complexity

Complexity

Amount of **resource required** by an algorithm, measured as a function of the **input size**.

Time Complexity

- Number of **steps** (“**running time**”) executed by an algorithm

Space Complexity

- Number of units of space required by an algorithm
 - ◆ e.g., number of elements in a list
 - ◆ number of nodes in a tree / graph

In CSC263 we will be dealing with time complexity most of the time.

Example: search a linked list

```
SearchFortyTwo(L):  
1.   z = L.head  
2.   while z != None and z.key != 42:  
3.       z = z.next  
4.   return z
```

Let input **L = 41 -> 51 -> 12 -> 42 -> 20 -> 88**

How many times Line #2 will be executed?

4

Now let **L = 41 -> 51 -> 12 -> 24 -> 20 -> 88**

How many times Line #2 will be run?

7 (the last one is $z == \text{None}$)

Note

Running time can be measured by counting the number of times **all lines** are executed, or the number of times **some lines** (such as Line #2 in LinkedSearch) are executed.

It's up to the problem, or what the question asks.

best-case
worst-case
average-case

Worst-case running time

- $t_A(x)$: the running time of algorithm A with input x
- If it is clear what A is, we can simply write $t(x)$
- The **worst-case running time $T(n)$** is defined as

$$T(n) = \max\{ t(x) : x \text{ is an input of size } n \}$$

“worst-case” is the case with the **longest** running time.

Slow is bad!

Best-case running time

Similarly to worst-case, **best-case** is the case with the **shortest** running time.

$$\min\{ t(x) : x \text{ is an input of size } n \}$$

Best case is not very interesting, and is rarely studied.

**Because we should prepare for the worst,
not the best!**

Example: Search a linked list, again

```
SearchFortyTwo(L):
1.   z = L.head
2.   while z != None and z.key != 42:
3.       z = z.next
4.   return z
```

What is the worst-case running time among all possible L with length n , i.e., $T(n)$?

$$T(n) = n + 1$$

the case where 42 is not in L (compare all n nodes plus a final *None*)

Average-case running time



In reality, the running time is NOT always the best case, and is NOT always the worst case.

The running time is “**distributed**” between the best and the worst.

For our SearchFortyTwo(L) algorithm the running time is distributed between ...

1 and $n+1$, inclusive

Average-case running time



So, the average-case running time is the **expectation** of the running time which is distributed between 1 and $n+1$, i.e.,...

Let t_n be a random variable whose possible values are between 1 and $n+1$

$$E[t_n] = \sum_{t=1}^{n+1} t \cdot \Pr(t_n = t)$$

We need to know this!

$$E[t_n] = \sum_{t=1}^{n+1} t \cdot \Pr(t_n = t)$$

Average-case running time



To know $\Pr(t_n = t)$, we need to be **given** the **probability distribution** of the inputs, i.e., how inputs are **generated** (following what **distribution**).

Now I give you one:

For each key in the linked list, we pick an integer between 1 and 100 (inclusive), **uniformly at random**.

Figure out $\Pr(t_n = t)$

For each key in the linked list, we pick an integer between 1 and 100 (inclusive), uniformly at random.

What is

$$\Pr(t_n = 1) = 0.01$$

when head is 42

$$\Pr(t_n = 2) = 0.99 \times 0.01$$

head is not 42 and
the second one is

$$\Pr(t_n = 3) = (0.99)^2 \times 0.01$$

...

$$\Pr(t_n = n) = (0.99)^{n-1} \times 0.01$$

None of the n keys is 42.

$$\Pr(t_n = n+1) = (0.99)^n$$

the first t keys are not 42 and the t -th is

$$\Pr(t_n = t) = \begin{cases} (0.99)^{t-1} \times 0.01 & 1 \leq t \leq n \\ (0.99)^n & t = n+1 \end{cases}$$

Now we are ready to compute the average-case running time -- $E[t_n]$

$$\begin{aligned} E[t_n] &= \sum_{t=1}^{n+1} t \cdot \Pr(t_n = t) \\ &= \sum_{t=1}^n t \cdot (0.99)^{t-1} \times 0.01 + (n+1) \cdot (0.99)^n \\ &= (n+1) \cdot (0.99)^n + 0.01 \cdot \sum_{t=1}^n t \cdot (0.99)^{t-1} \\ &= 100 - 99 \times (0.99)^n \end{aligned}$$

$$\sum_{t=1}^n t \cdot (0.99)^{t-1}$$

This sum needs a bit of a trick, but can be done!

Calculate the sum (after-class reading)

$$S = \sum_{t=1}^n t \cdot 0.99^{t-1} = 1 + 2 \cdot 0.99 + 3 \cdot 0.99^2 + \cdots + n \cdot 0.99^{n-1}$$

$$0.99S = \sum_{t=1}^n t \cdot 0.99^t = 1 \cdot 0.99 + 2 \cdot 0.99^2 + \cdots + (n-1) \cdot 0.99^{n-1} + n \cdot 0.99^n$$

take the difference of the above two equations

$$0.01S = \sum_{i=0}^{n-1} 0.99^i - n \cdot 0.99^n = \frac{1 - 0.99^n}{1 - 0.99} - n \cdot 0.99^n$$

$$= 100 - (100 + n) \cdot 0.99^n$$

sum of geometric series

You **should** be comfortable with this type of calculations.

The final result

Input distribution: for each key in the linked list, we pick an integer between **1** and **100** (inclusive), **uniformly at random**.

The average-case running time of **SearchFortyTwo(L)** (measured by counting Line #2) is:

$$E[t_n] = 100 - 99 \cdot (0.99)^n$$

If $n = 0$, then $E[t_n] = 1$, since it's always 1 comparison

If n is very large (e.g., 1000000), $E[t_n]$ is close to 100, i.e., the algorithm is expected to finish within **100** comparisons, even if the worse-case is 1000000 comps.

ONE DOES NOT SIMPLY

**TALK ABOUT AVERAGE-CASE RUNNING TIME
WITHOUT TALKING ABOUT INPUT DISTRIBUTION**

asymptotic

upper bound

tight bound

lower bound

Asymptotic notations

$O(f(n))$: the set of functions that grow
no faster than $f(n)$

→ if $g \in O(f)$, then we say g is asymptotically
upper bounded by f

$\Omega(f(n))$: the set of functions that grow
no slower than $f(n)$

→ if $g \in \Omega(f)$, then we say g is asymptotically
lower bounded by f

Asymptotic notations

if $g \in O(f)$ **and** $g \in \Omega(f)$,

then we say $g \in \Theta(f)$

$\Theta(f(n))$: the set of functions that grow
no slower and no faster than $f(n)$

we call it the **tight** bound.

The ideas behind asymptotic notations

→ We only care about the **rate** of growth, so **constant factors** don't matter.

- ◆ $100n^2$ and n^2 have the same rate of growth (both are quadrupled when n is doubled)

→ We only care about **large inputs**, so only the **highest-degree** term matters

- ◆ n^2 and $n^2 + 263n + 3202$ are nearly the same when n is very large

growth rate ranking of typical functions

$$f(n) = n^n$$

$$f(n) = 2^n$$

$$f(n) = n^3$$

$$f(n) = n^2$$

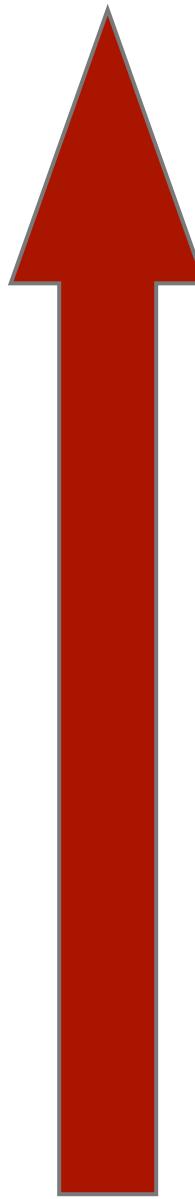
$$f(n) = n \log n$$

$$f(n) = n$$

$$f(n) = \sqrt{n}$$

$$f(n) = \log n$$

$$f(n) = 1$$



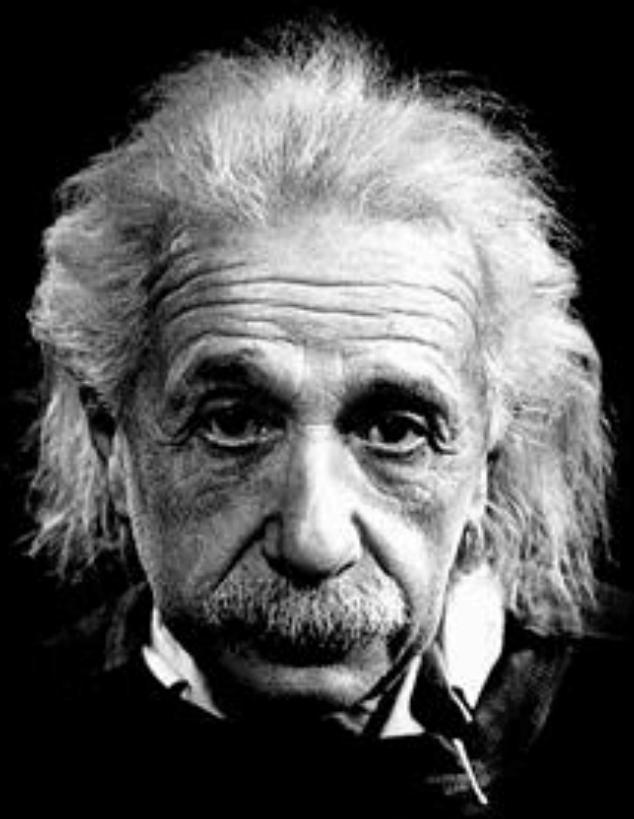
grow fast

grow slowly

a high-level look at asymptotic notations

*It is a **simplification** of the “real” running time*

- *it does not tell the whole story about how fast a program runs in real life.*
 - ◆ *in real-world applications, constant factor matters! hardware matters! implementation matters!*
- *this simplification makes possible the development of the whole **theory of computational complexity**.*
 - ◆ **HUGE idea!**

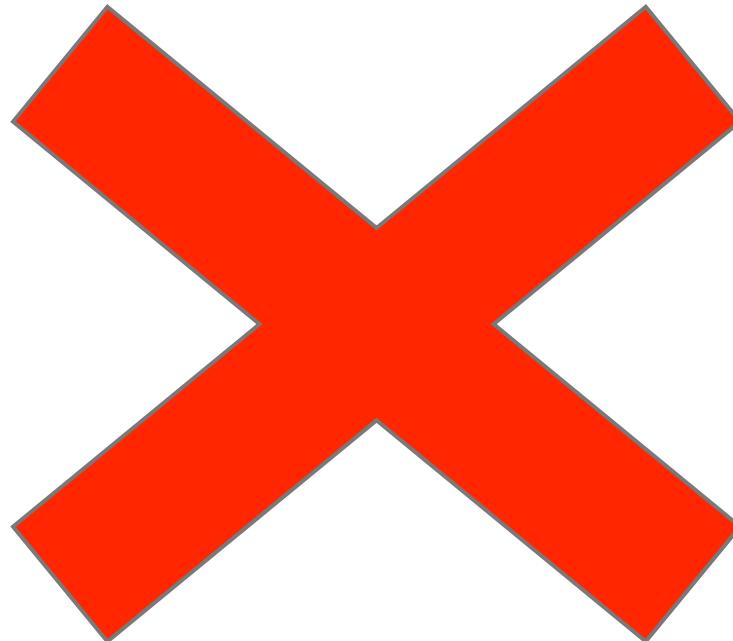


**“Make everything as
simple as possible,
but not simpler.”**

—Albert Einstein

O is for describing **worst-case** running time

Ω is for describing **best-case** running time



O and Ω can **both** be used to upper-bound and lower-bound the **worst-case** running time

O and Ω can **both** be used to upper-bound and lower-bound the **best-case** running time

O and Ω can **both** be used to upper-bound and lower-bound the **average-case** running time

How to argue algorithm A(x)'s **worst-case** running time is in $\mathbf{O}(n^2)$

We need to argue that, _____ **for every** _____ input x of size n , the running time of A with input x , i.e., $t(x)$ is **no larger** than cn^2 , where $c > 0$ is a constant.

- A. for every
- B. there exists an
- C. no larger
- D. no smaller



**think about the commuting time from school
to home every day**

“even the worst day is less than 2 hours”

that means every day is less than 2 hours



@视觉古都

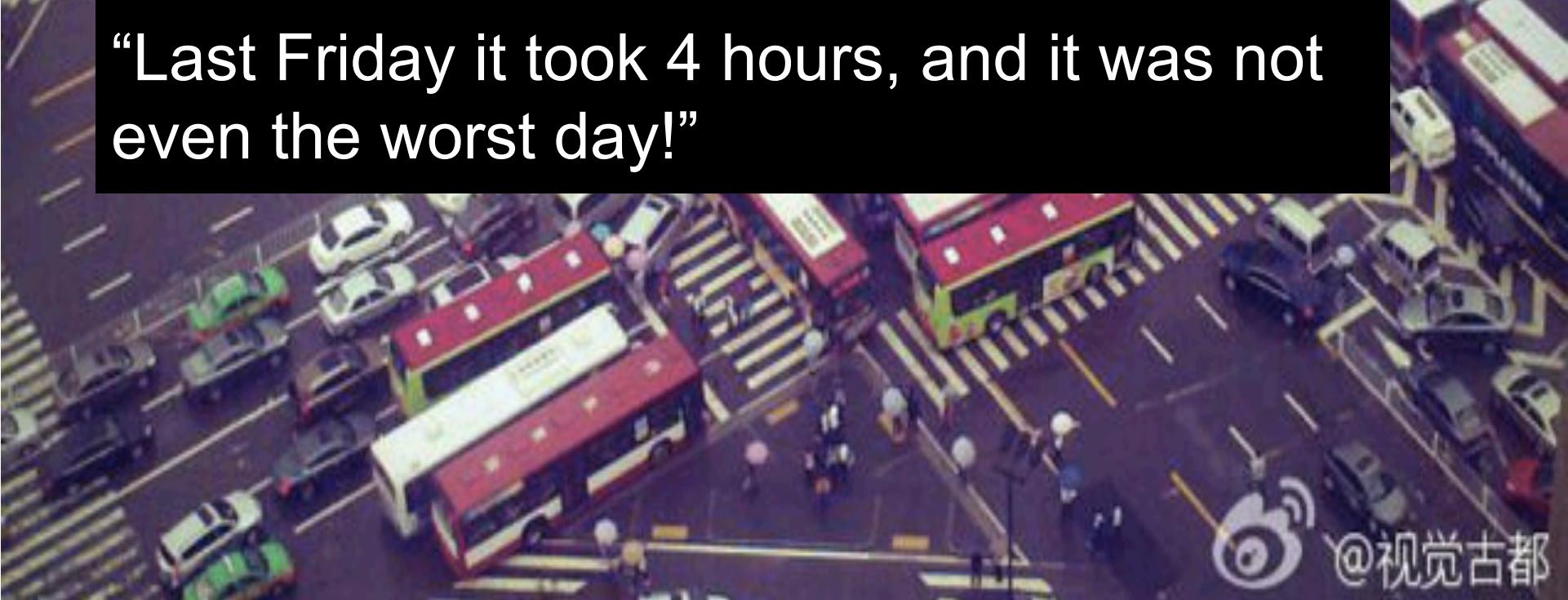
How to argue algorithm A(x)'s **worst-case** running time is in $\Omega(n^2)$

We need to argue that, there exists an input x of size n , the running time of A with input x , i.e., $t(x)$ is no smaller than cn^2 , where $c > 0$ is a constant.

- A. for every
- B. there exists an
- C. no larger
- D. no smaller



“the worst day is more than 2 hours”



“Last Friday it took 4 hours, and it was not even the worst day!”



@视觉古都

How to argue algorithm A(x)'s **best-case** running time is in $\mathbf{O}(n^2)$

We need to argue that, there exists an input x of size n , the running time of A with input x , i.e., $t(x)$ is no larger than cn^2 , where $c > 0$ is a constant.

- A. for every
- B. there exists an
- C. no larger
- D. no smaller

How to argue algorithm A(x)'s **best-case** running time is in $\Omega(n^2)$

We need to argue that, _____ **for every** _____ input x of size n , the running time of A with input x , i.e., $t(x)$ is **no smaller** than cn^2 , where $c > 0$ is a constant.

- A. for every
- B. there exists an
- C. no larger
- D. no smaller

In CSC263

- Most of the time, we study the upper-bound on worst-case running time.
- Sometimes we try to get a tight bound Θ if we can
- Sometimes we study the upper bound on average-case running time.

Note: exact form & asymptotic notations

In CSC263 homework and exam questions, sometimes we ask you to express running time in **exact forms**, and sometime we ask you to express running time in **asymptotic notations**, so always read the question carefully.

If you feel rusty with probabilities, please read the Appendix C of the textbook. It is only about 20 pages, and is highly relevant to what we need for CSC263.

Appendix A and B are also worth reading.

next week

ADT: Priority queue

Data structure: Heap

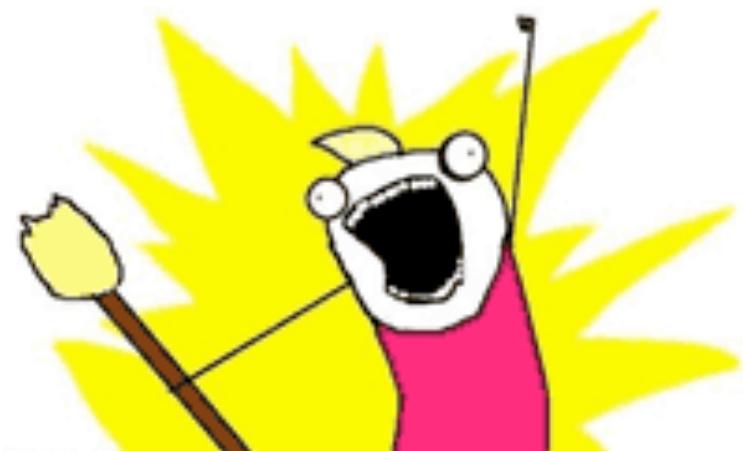
CSC263 Week 2

If you feel rusty with probabilities, please read the Appendix C of the textbook. It is only about 20 pages, and is highly relevant to what we need for CSC263.

Appendix A and B are also worth reading.

Problem Set 1 is due this Tuesday!

(Sept 29)



This week topic

→ADT: **Priority Queue**

→Data structure: **Heap**

An ADT we already know

First in first serve



Queue:

- a collection of elements
- supported operations
 - ◆ Enqueue(Q, x)
 - ◆ Dequeue(Q)
 - ◆ PeekFront(Q)

The new ADT

Max-Priority Queue:

- a collection of elements **with priorities**, i.e., each element x has $x.\text{priority}$
- supported operations
 - ◆ **Insert**(Q, x)
 - like enqueue(Q, x)
 - ◆ **ExtractMax**(Q)
 - like dequeue(Q)
 - ◆ **Max**(Q)
 - like PeekFront(Q)
 - ◆ **IncreasePriority**(Q, x, k)
 - increase $x.\text{priority}$ to k

Oldest person first



Applications of Priority Queues

- Job scheduling in an operating system
 - ◆ Processes have different priorities (Normal, high...)
- Bandwidth management in a router
 - ◆ Delay sensitive traffic has higher priority
- Find minimum spanning tree of a graph
- etc.

**Now, let's implement
a (Max)-Priority Queue**

40 -> 33 -> 18 -> 65 -> 24 -> 25

Use an **unsorted linked list**

→**INSERT(Q, x)** # x is a node

- ◆ Just insert x at the head, which takes $\Theta(1)$

→**IncreasePriority(Q, x, k)**

- ◆ Just change x.priority to k, which takes $\Theta(1)$

→**Max(Q)**

- ◆ Have to go through the whole list, takes $\Theta(n)$

→**ExtractMax(Q)**

- ◆ Go through the whole list to find x with max priority ($O(n)$), then delete it ($O(1)$ if doubly linked) and return it, so overall $\Theta(n)$.

65 → 40 → 33 → 25 → 24 → 18

Use a **reversely sorted linked list**

→**Max(Q)**

- ◆ Just return the head of the list, $\Theta(1)$

→**ExtractMax(Q)**

- ◆ Just delete and return the head, $\Theta(1)$

→**INSERT(Q, x)**

- ◆ Have to linearly search the correct location of insertion which takes $\Theta(n)$ in worst case.

→**IncreasePriority(Q, x, k)**

- ◆ After increase, need to move element to a new location in the list, takes $\Theta(n)$ in worst case.

$\Theta(1)$ is fine, but $\Theta(n)$ is kind-of bad...

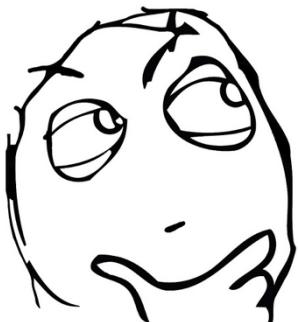
unsorted linked list

sorted linked list

...

Can we link these elements in a *smarter* way, so that we never need to do $\Theta(n)$?

Why does a sorted array also not work?

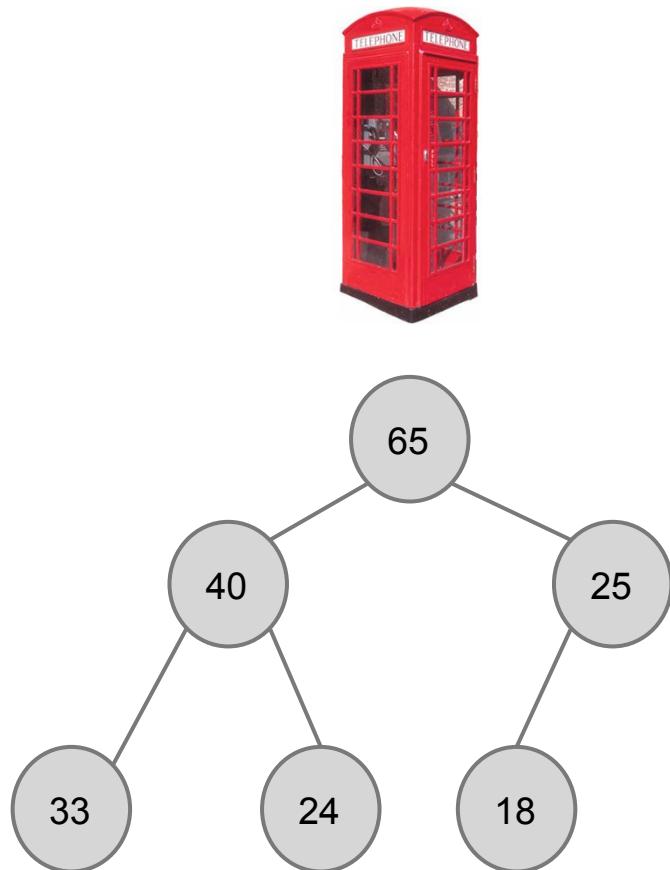


Yes, we can!

Worst case running times

	unsorted list	sorted list	Heap
Insert(Q, x)	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
Max(Q)	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
ExtractMax(Q)	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$
IncreasePriority (Q, x, k)	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$

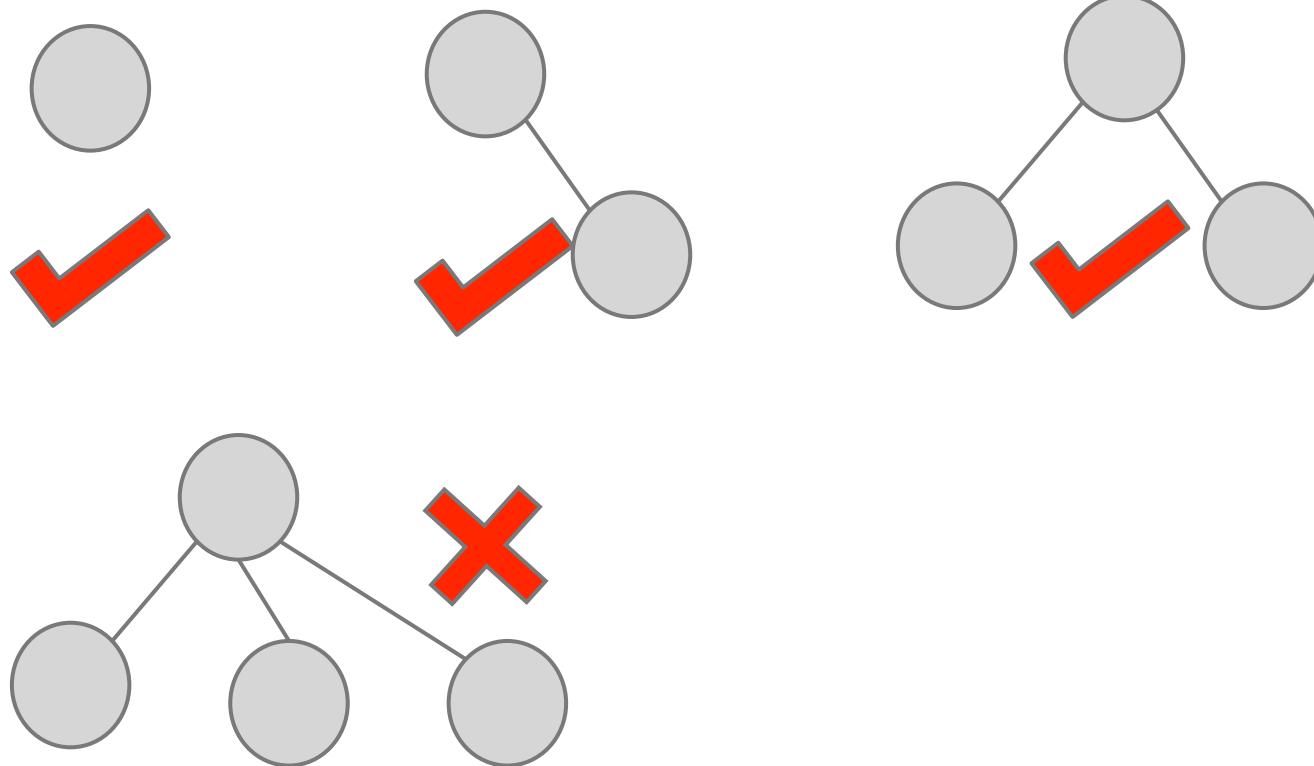
Binary Max-Heap



A **binary max-heap** is a **nearly-complete binary** tree that has the **max-heap property**.

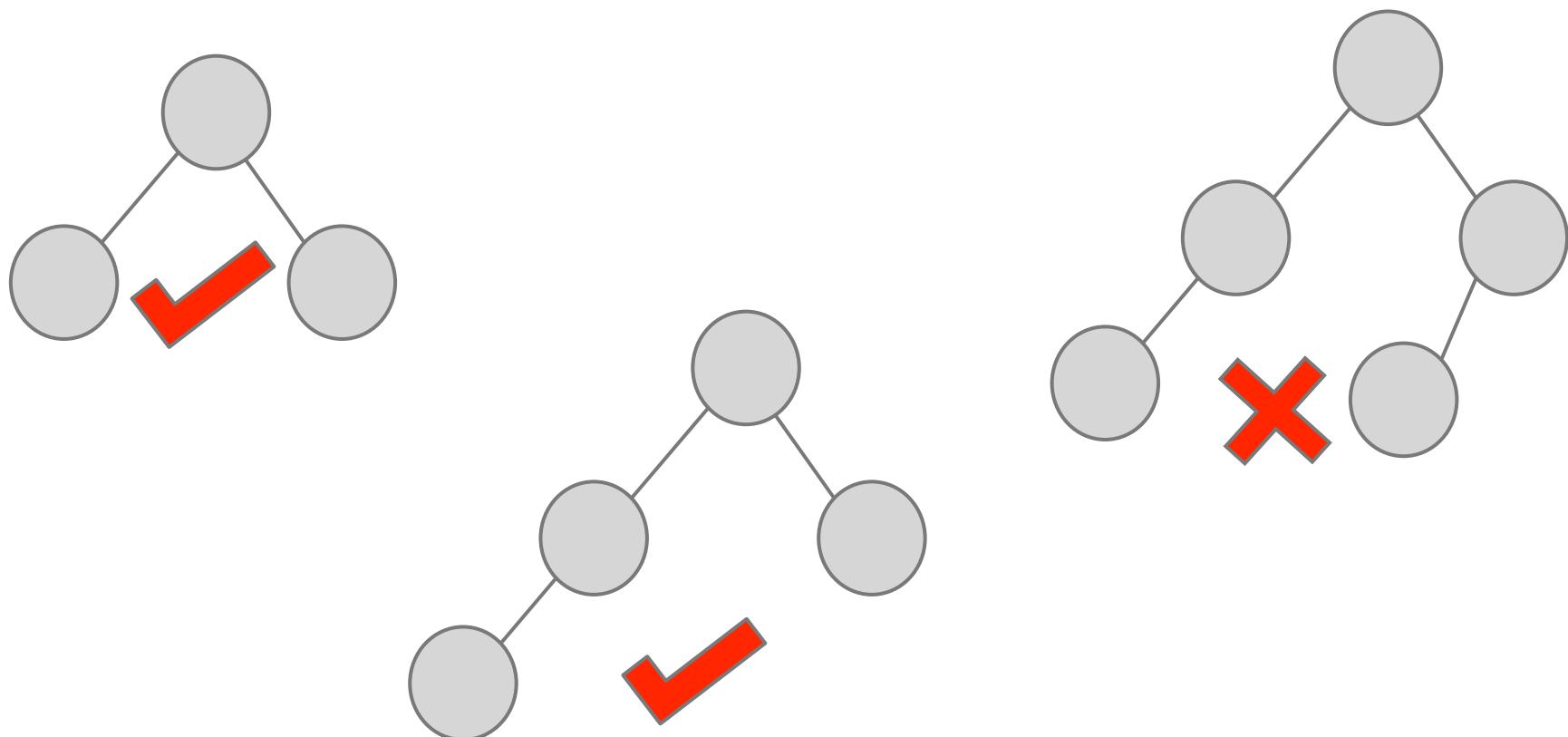
It's a **binary** tree

Each node has at **most** 2 children



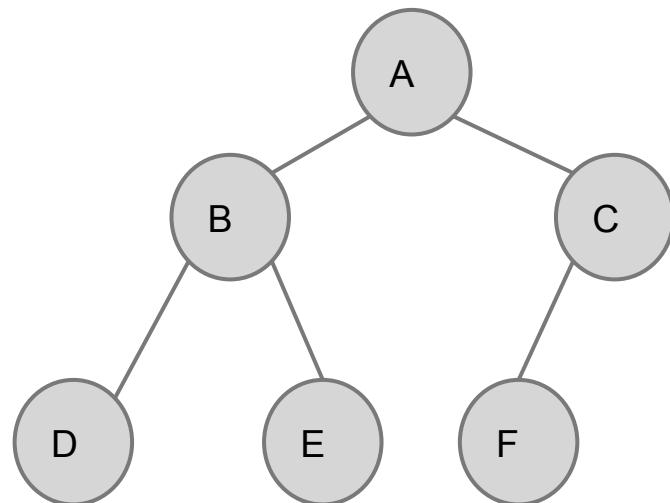
It's a **nearly-complete** binary tree

Each level is **completely filled**, except the bottom level
where nodes are filled to as **far left** as possible



Why is it important to be a nearly-complete binary tree?

Because then we can **store** the tree in an **array**, and each node knows which **index** has its parent and its left/right child.



A	B	C	D	E	F
index: 1	2	3	4	5	6

$$\text{Left}(i) = 2i$$

$$\text{Right}(i) = 2i + 1$$

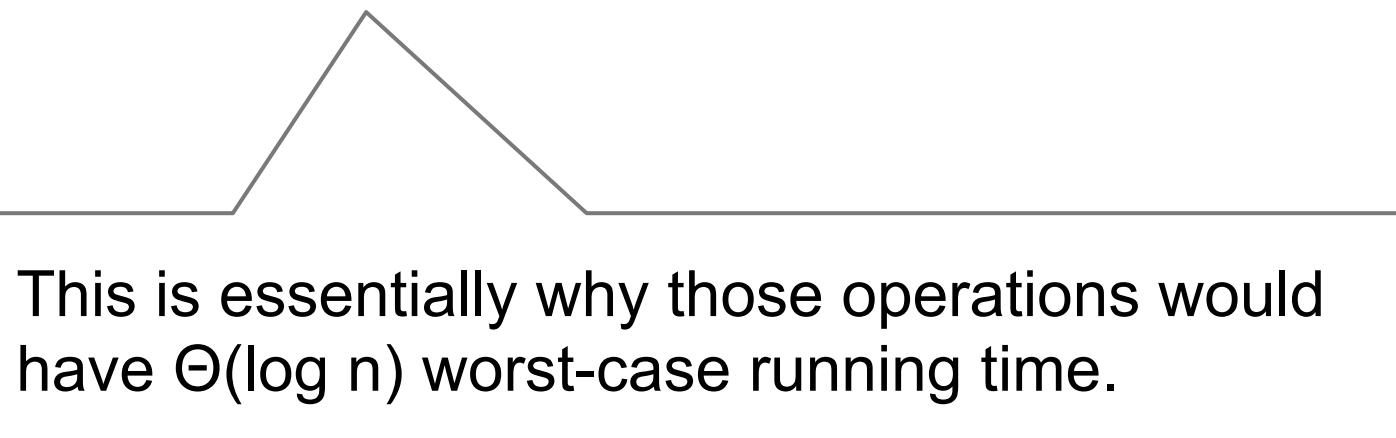
$$\text{Parent}(i) = \text{floor}(i/2)$$

Assume index starts from 1

Why is it important to be a **nearly-complete binary tree**?

Another reason:

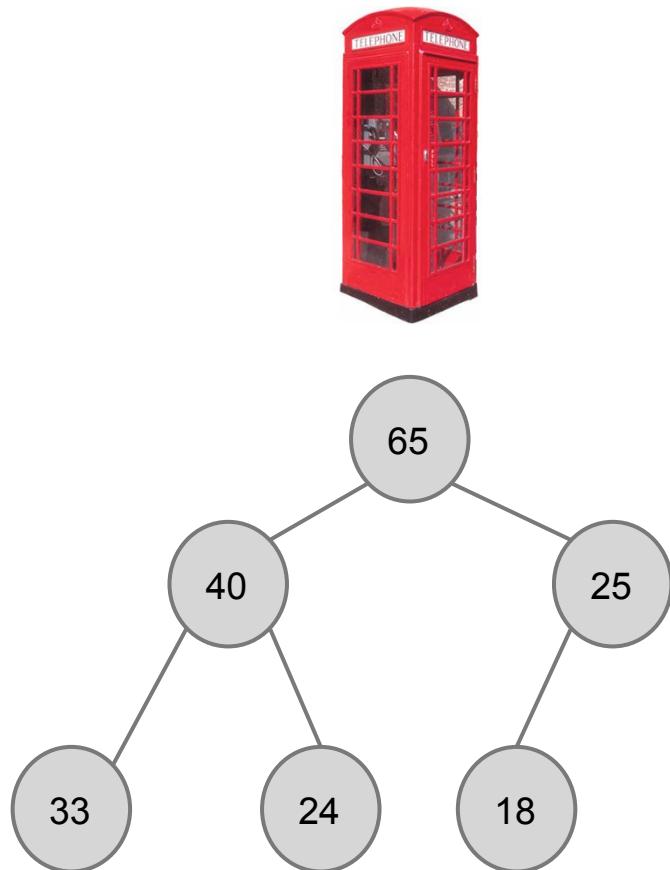
The **height** of a complete binary tree with **n** nodes is **$\Theta(\log n)$** .



A thing to remember...

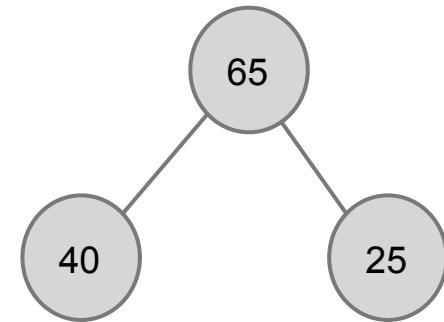
A heap is stored in an array.

Binary Max-Heap

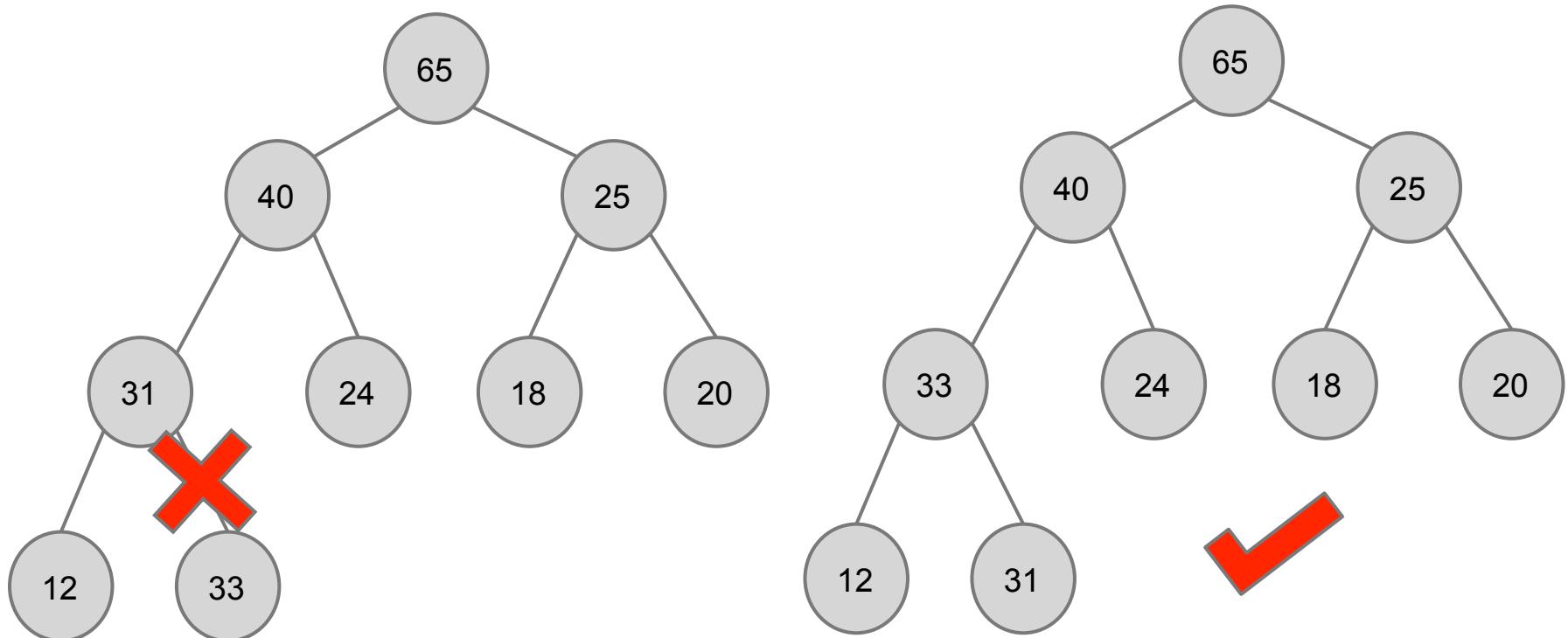


A **binary max-heap** is a **nearly-complete binary** tree that has the **max-heap property**.

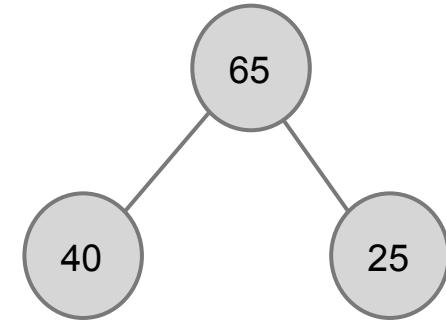
The max-heap property



Every node has key (priority) greater than or equal to keys of its **immediate** children.

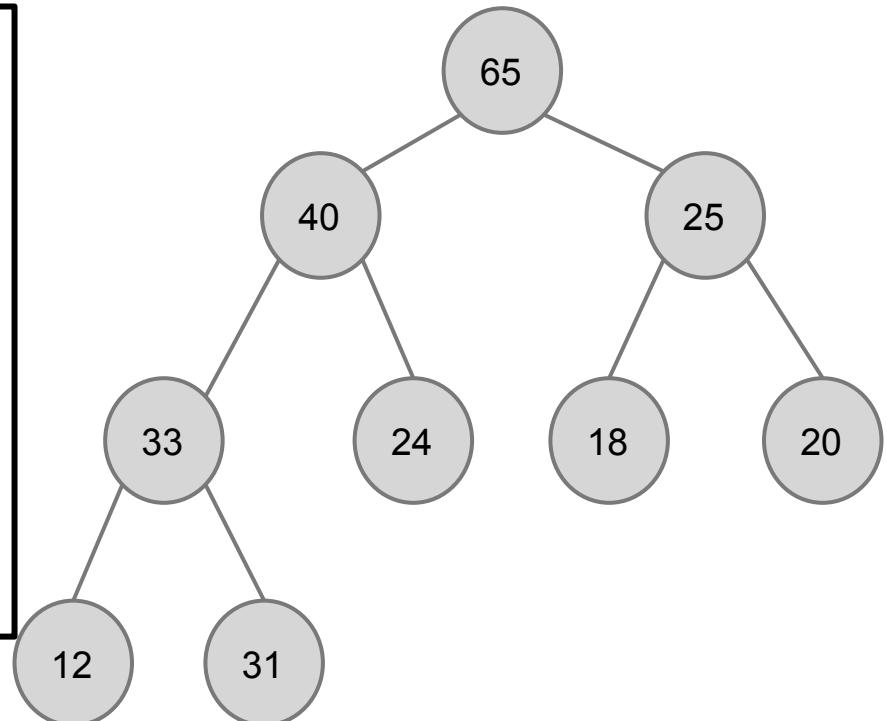


The **max-heap** property



Every node has key (priority) greater than or equal to keys of its **immediate** children.

Implication: every node is larger than or equal to **all its descendants**, i.e., every **subtree** of a heap is also a heap.



We have a binary max-heap defined,
now let's do operations on it.

- Max(Q)
- Insert(Q, x)
- ExtractMax(Q)
- IncreasePriority(Q, x, k)

Max(Q)

Return the largest key in Q,
in $O(1)$ time

Max(Q): return the maximum element

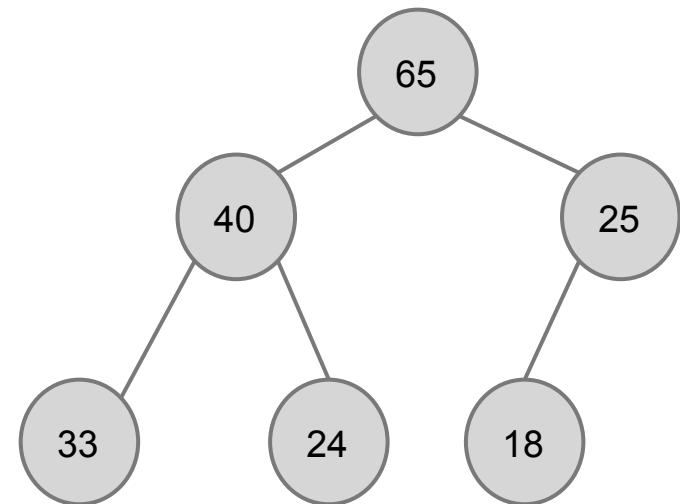
Return the **root** of the heap, i.e.,

just **return Q[1]**

(index starts from 1)

worst case **$\Theta(1)$**

Q	65	40	25	33	24	18
---	----	----	----	----	----	----



Insert(Q, x)

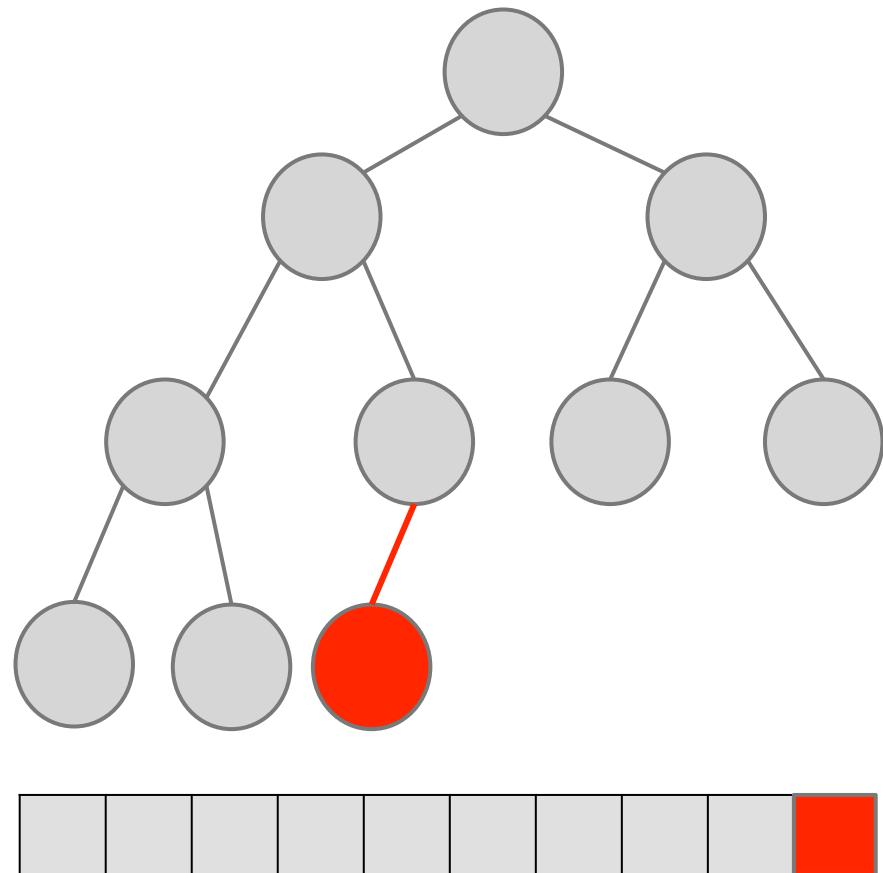
Insert node x into heap Q,
in $O(\log n)$ time

Insert(Q, x): insert a node to a heap

First thing to note:

Which spot to add
the new node?

The only spot that
keeps it a **complete**
binary tree.



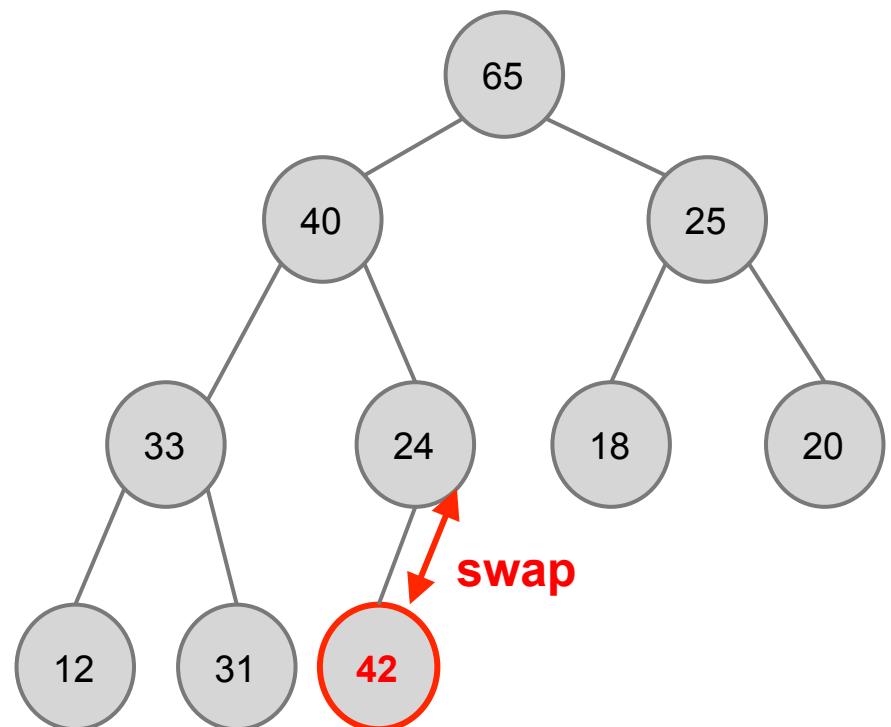
Increment heap size

Insert(Q, x): insert a node to a heap

Second thing to note:

Heap property might be broken, how to fix it and **maintain** the heap property?

“Bubble-up” the new node to a proper position, by **swapping with parent**.

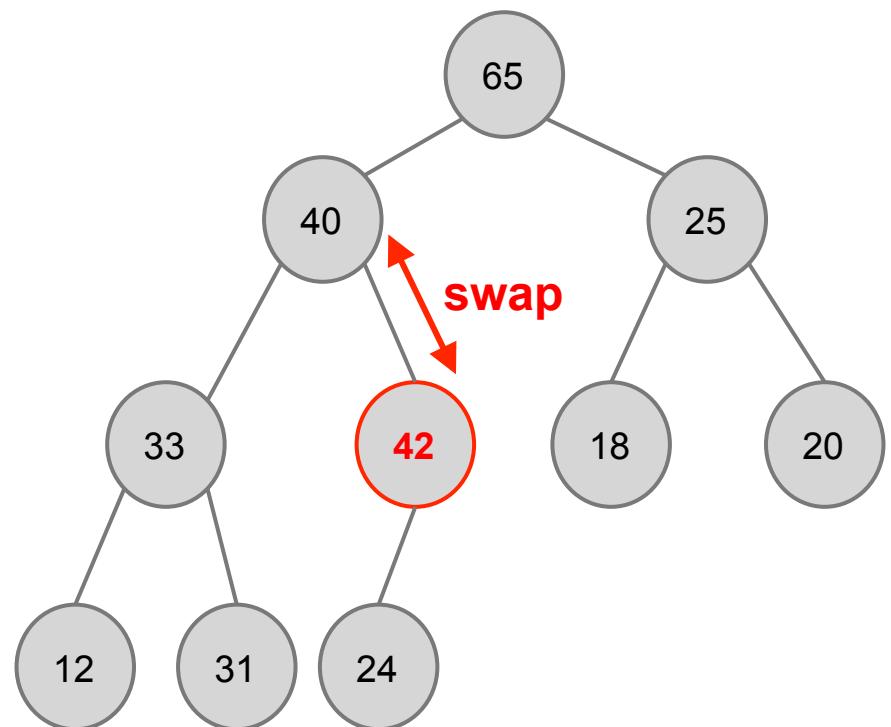


Insert(Q, x): insert a node to a heap

Second thing to note:

Heap property might be broken, how to fix it and **Maintain** the heap property.

“Bubble-up” the new node to a proper position, by **swapping with parent**.

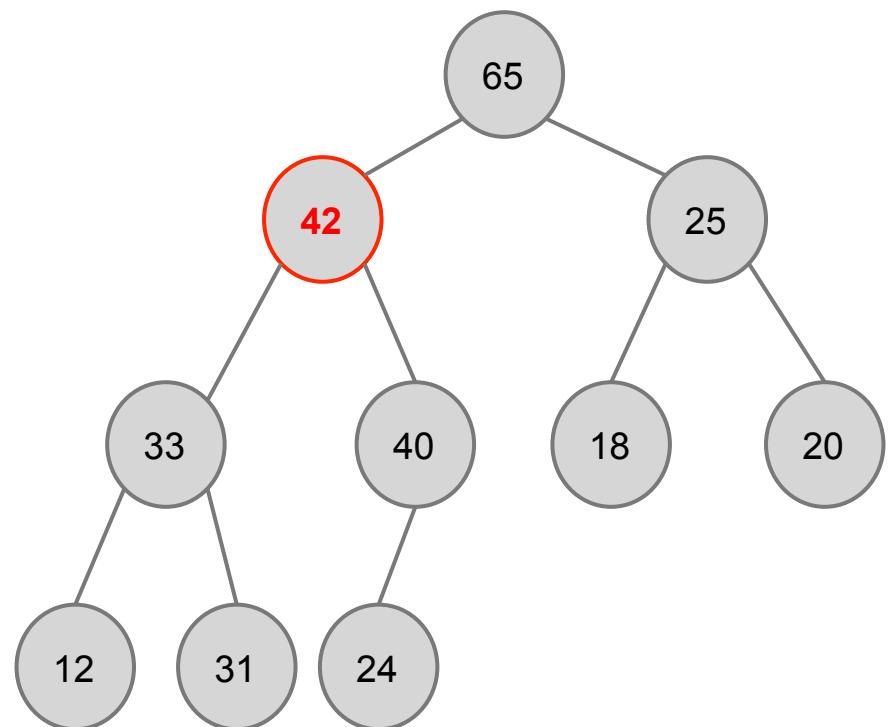


Insert(Q, x): insert a node to a heap

Second thing to note:
Heap property might be broken, how to fix it and
Maintain the heap
property.

“Bubble-up” the new
node to a proper
position, by **swapping**
with parent.

Worst-case:
 $\Theta(\text{height}) = \Theta(\log n)$



ExtractMax(Q)

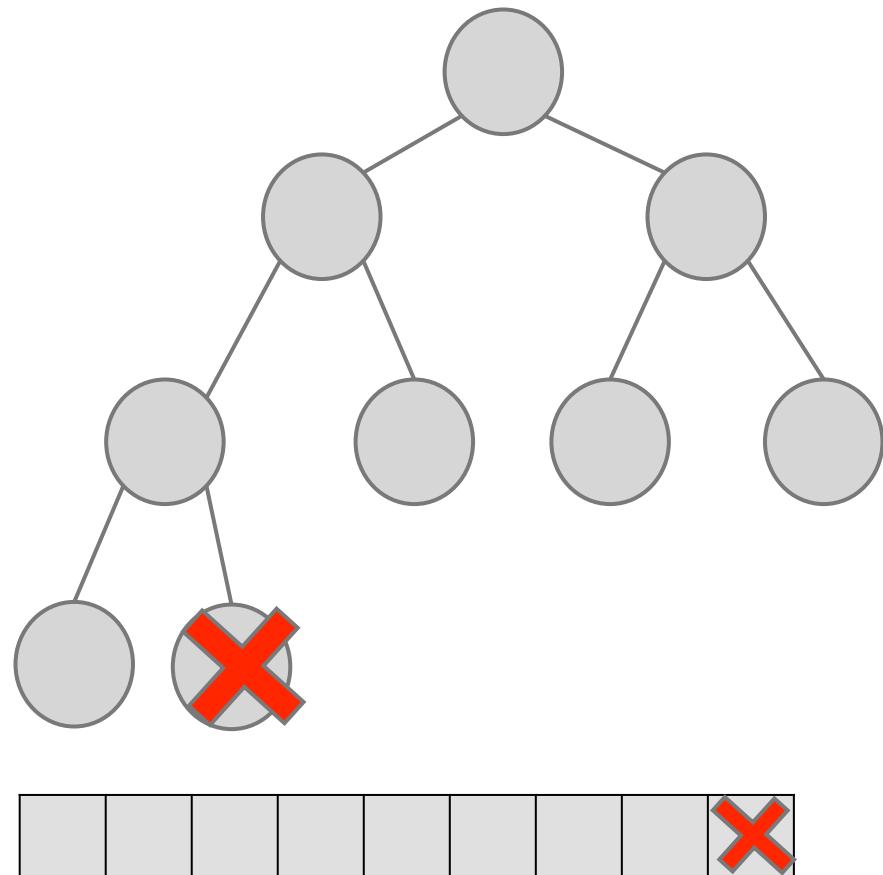
Delete and return the largest key in Q,
in $O(\log n)$ time

ExtractMax(Q): delete and return the maximum element

First thing to note:

Which **spot** to remove?

The only **spot** that keeps it a **complete** binary tree.



Decrement heap size

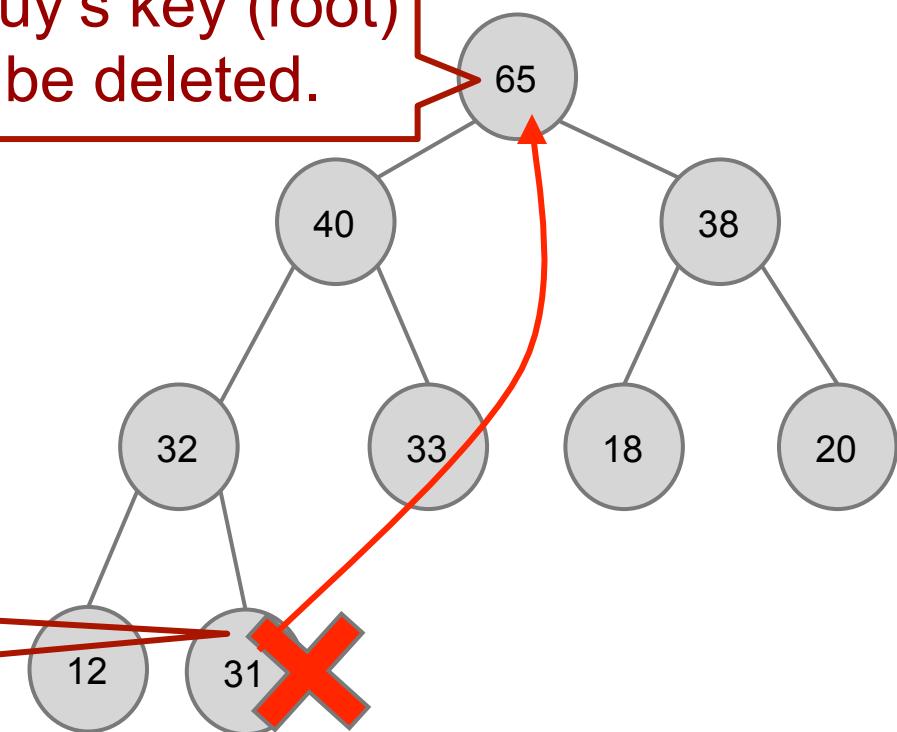
ExtractMax(Q): delete and return the maximum element

First thing to note:

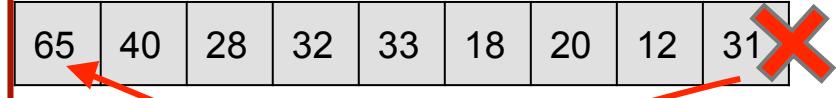
Which **spot** to remove?

The only **spot** that keeps it a **complete** binary tree.

THIS guy's key (root)
should be deleted.



Overwrite root with the **last** guy's key, then **delete** the last guy (decrement heap size).

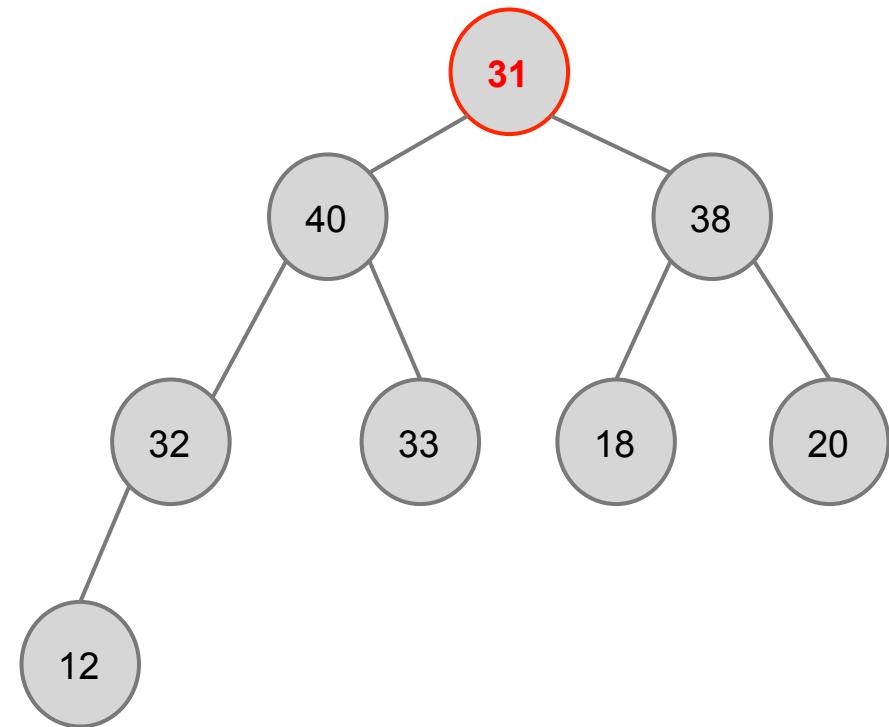


Decrement heap size

ExtractMax(Q): delete and return the maximum element

Now the **heap property** is broken again..., need to fix it.

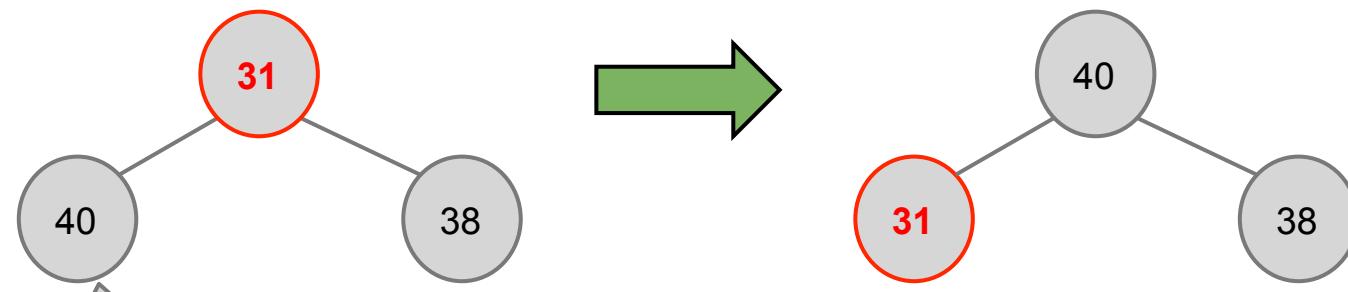
“Bubble-down” by swapping with...
a child...



Which child to swap with?



so that, after the swap, **max-heap property** is satisfied

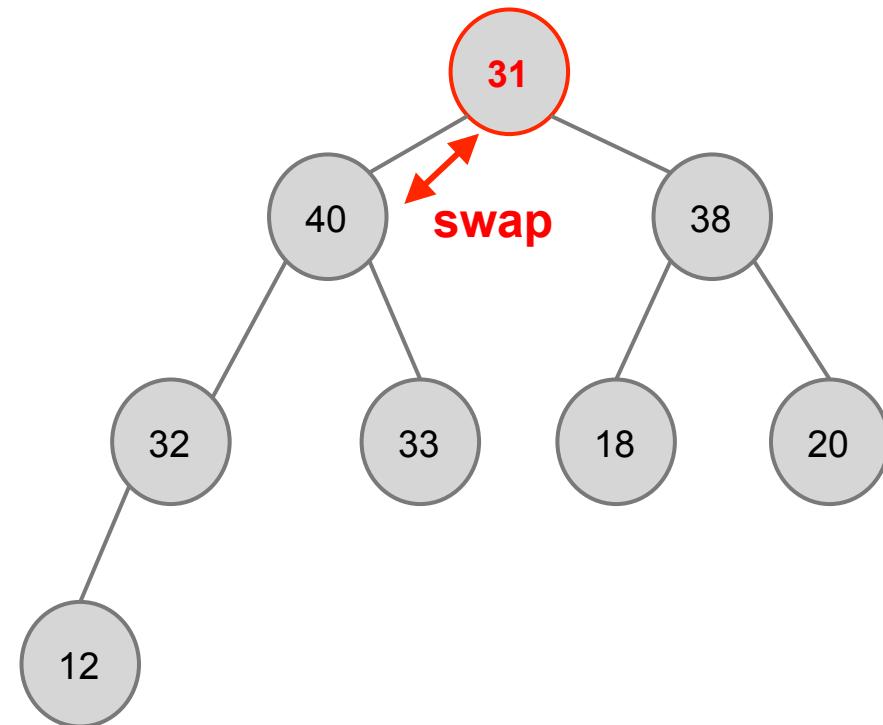


The “elder” child!
because it is the **largest** among the three

ExtractMax(Q): delete and return the maximum element

Now the **heap property** is broken again..., need to fix it.

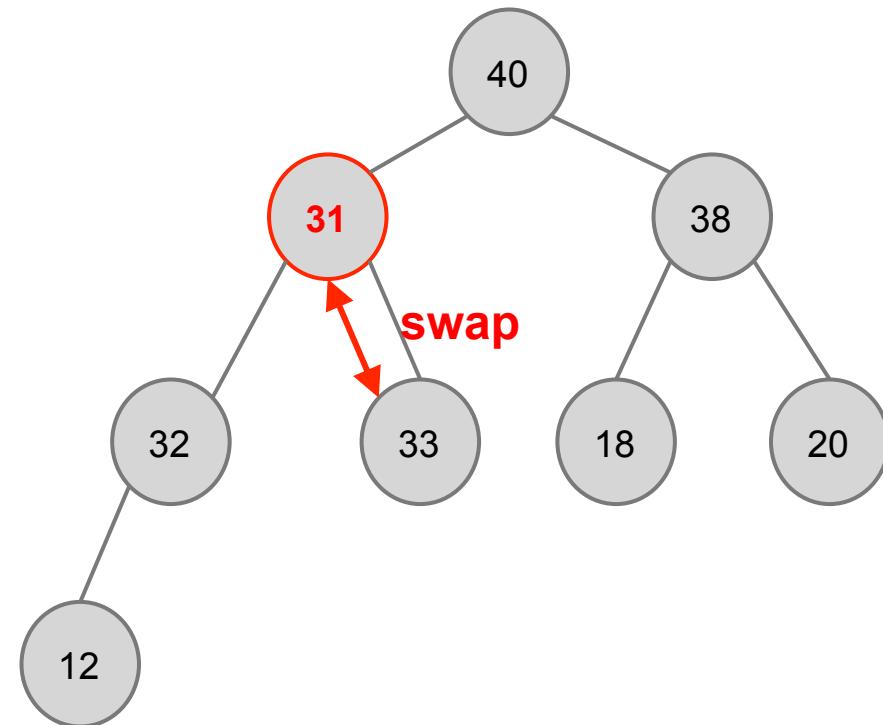
“Bubble-down” by swapping with **the elder child**



ExtractMax(Q): delete and return the maximum element

Now the **heap property** is broken again..., need to fix it.

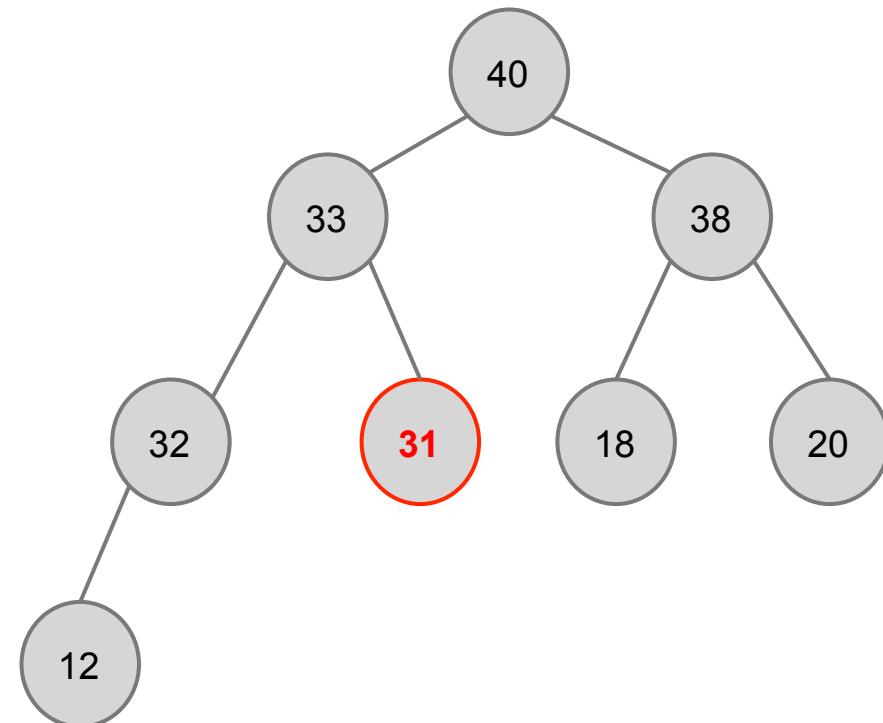
“Bubble-down” by swapping with...
the elder child



ExtractMax(Q): delete and return the maximum element

Now the **heap property** is broken again..., need to fix it.

“Bubble-down” by swapping with **the elder child**



Worst case running time: $\Theta(\text{height}) + \text{some constant work}$
 $\Theta(\log n)$

Quick summary

Insert(Q, x):

→Bubble-up, swapping with parent

ExtractMax(Q)

→Bubble-down, swapping elder child

Bubble up/down is also called **percolate** up/down, or **sift** up down, or **tickle** up/down, or **heapify** up/down, or **cascade** up/down.

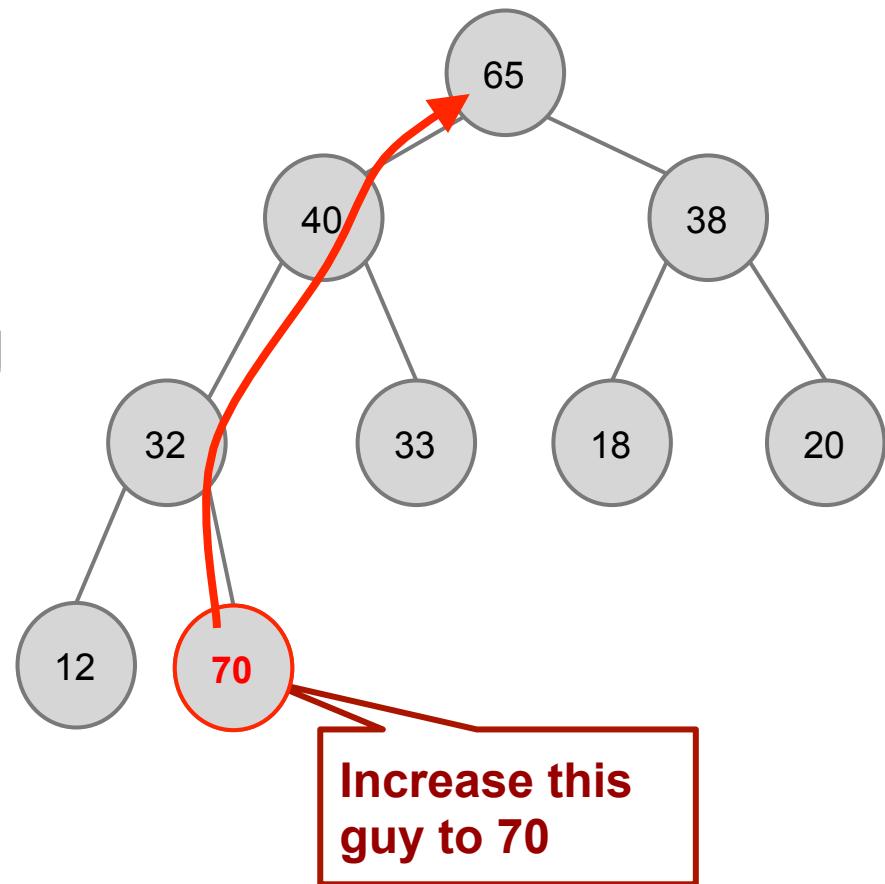
IncreasePriority(Q, x, k)

Increases the key of node x to k,
in $O(\log n)$ time

IncreasePriority(Q, x, k):
increase the key of node x to k

Just increase the key,
then...

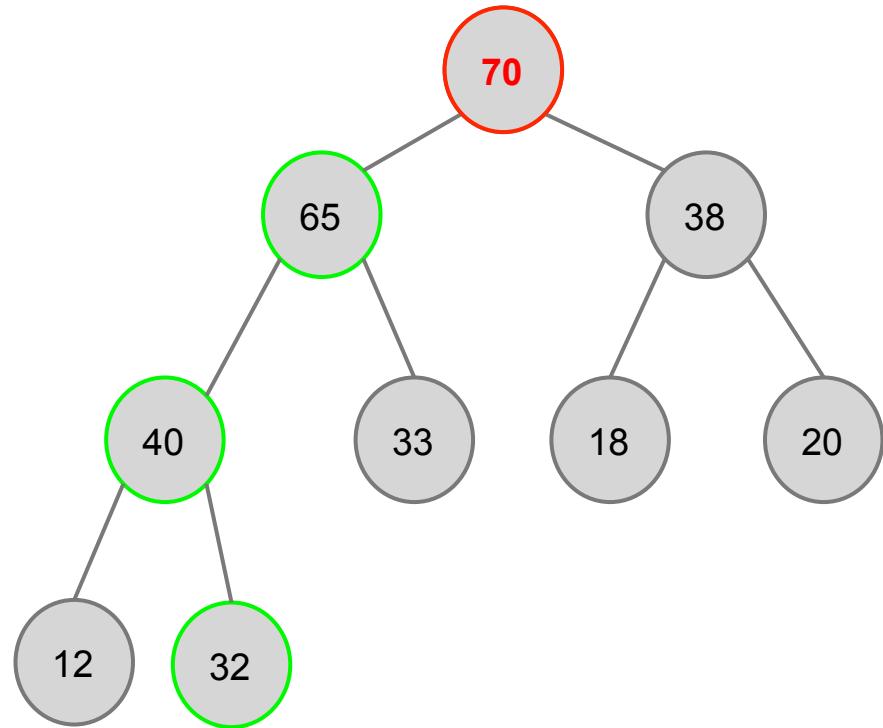
Bubble-up by swapping
with parents, to proper
location.



IncreasePriority(Q, x, k):
increase the key of node x to k

Just increase the key,
then...

Bubble-up by swapping
with parents, to proper
location.



Worst case running time: $\Theta(\text{height}) + \text{some constant work}$
 $\Theta(\log n)$

Now we have learned how implement a priority queue using a heap

- Max(Q)
- Insert(Q, x)
- ExtractMax(Q)
- IncreasePriority(Q, x, k)

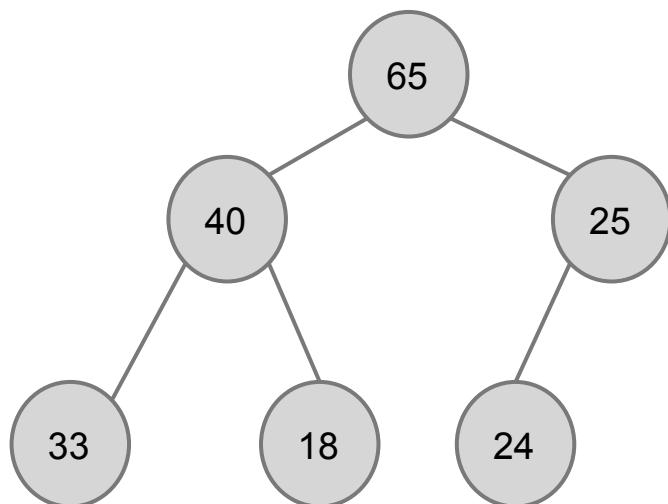
Next:

- How to use heap for **sorting**
- How to **build a heap** from an unsorted array

HeapSort

Sorts an array, in $O(n \log n)$ time

The idea



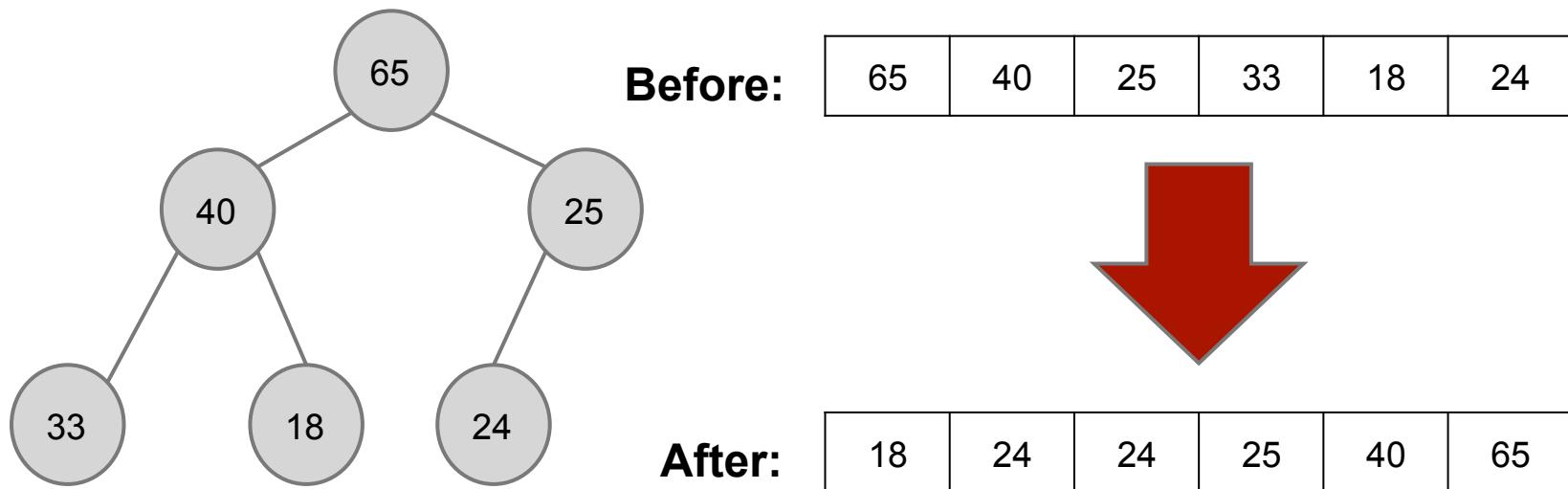
Worst-case running time: each ExtractMax is **O(log n)**, we do it **n** times, so overall it's...
O(n log n)

How to get a sorted list out of a heap with n nodes?

Keep extracting max for n times, the keys extracted will be sorted in non-ascending order.

Now let's be more precise

What's needed: modify a max-heap-ordered array into a **non-descendingly** sorted array



We want to do this "**in-place**" without using any extra array space, i.e., just by **swapping** things around.

Valid heaps are green rectangled

65	40	25	33	18	24
----	----	----	----	----	----

Step 1: swap first (65) and last (24), since the tail is where 65 (max) belongs to.

24	40	25	33	18	65
----	----	----	----	----	----

Step 2: decrement heap size

24	40	25	33	18	65
----	----	----	----	----	----

This node is like deleted from the tree, not touched any more.

40	33	25	24	18	65
----	----	----	----	----	----

18	33	25	24	40	65
----	----	----	----	----	----

Step 3: fix the heap by **bubbling down 24**

33	25	18	24	40	65
----	----	----	----	----	----

25	24	18	33	40	65
----	----	----	----	----	----

24	18	25	33	40	65
----	----	----	----	----	----

18	24	25	33	40	65
----	----	----	----	----	----

18	24	25	33	40	65
----	----	----	----	----	----

Repeat Step 1-3 until the array is fully sorted (at most n iterations).

HeapSort, the pseudo-code

HeapSort(A)

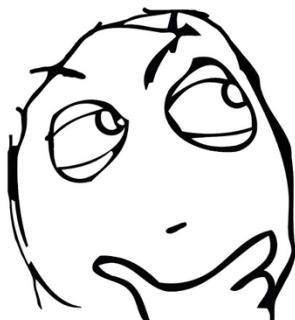
“sort any array A into non-descending order” Missing!

BuildMaxHeap(A) # convert any array A into a heap-ordered one

swap $A[1]$ and $A[i]$ # Step 1: swap the first and the last

$A.size \leftarrow A.size - 1$ # Step 2: decrement size of heap

BubbleDown($A, 1$) # Step 3: bubble down the 1st element in A



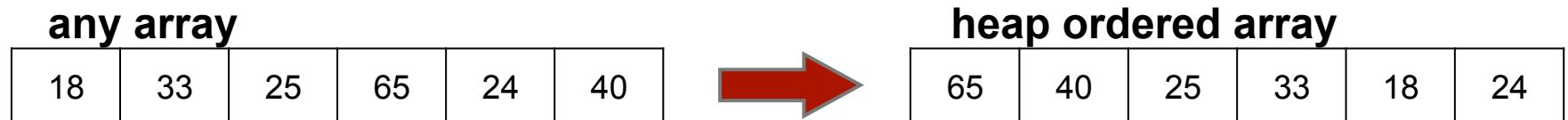
Does it work?

It works for an array A that is initially *heap-ordered*, it does work NOT for *any array*!

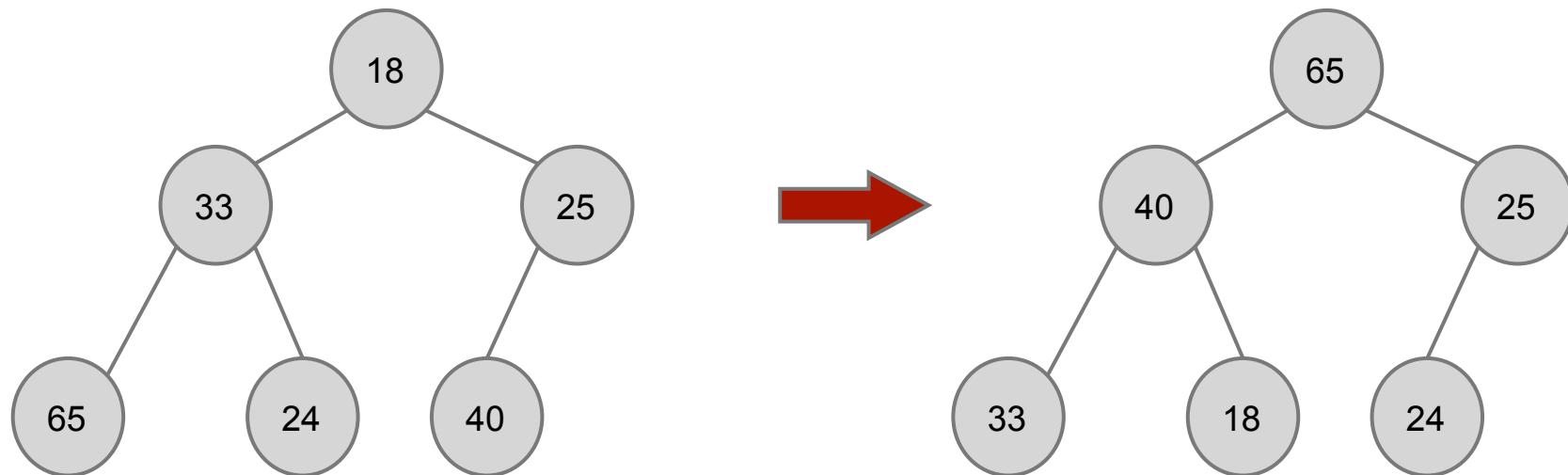
BuildMaxHeap(A)

Converts an array into a max-heap ordered array, in $O(n)$ time

Convert any array into a heap ordered one



In other words...



Idea #1

```
BuildMaxHeap(A):
```

```
    B ← empty array # empty heap
    for x in A:
        Insert(B, x) # heap insert
    A ← B      # overwrite A with B
```

Running time:

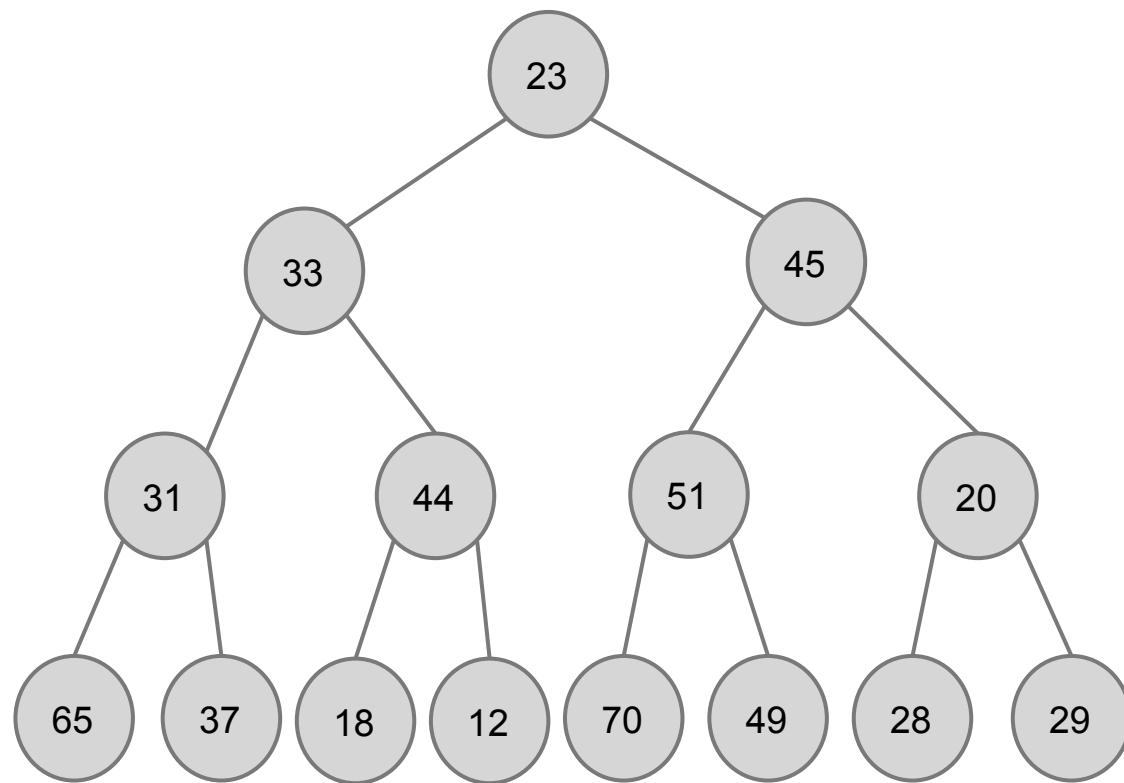
Each Insert takes **O(log n)**, there are **n** inserts...
so it's **O(n log n)**, not very exciting.
Not **in-place**, needs a second array.

WHAT IF I TOLD YOU

YOU CAN DO BETTER THAN THIS

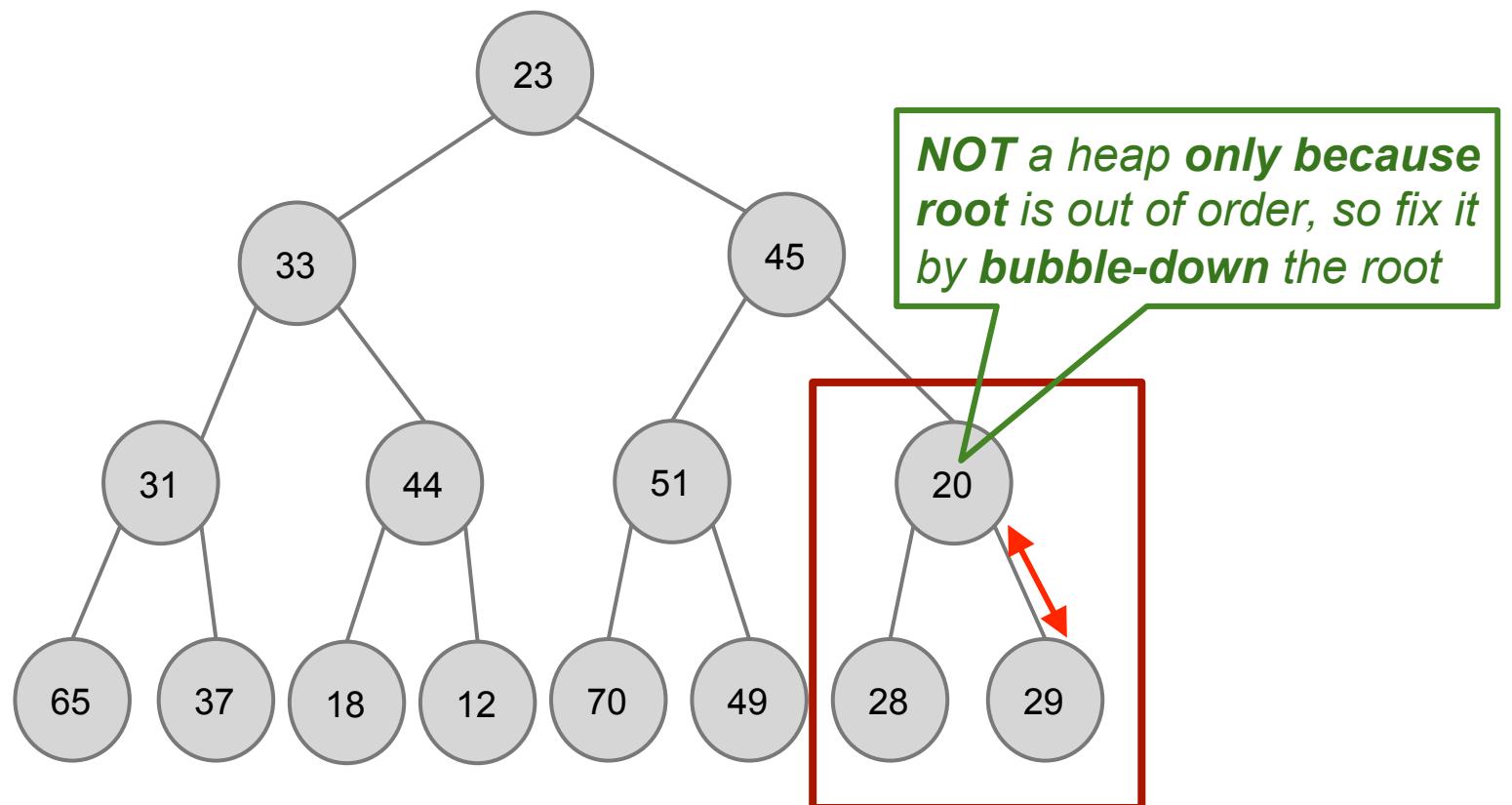
Idea #2

Fix heap order, from bottom up.



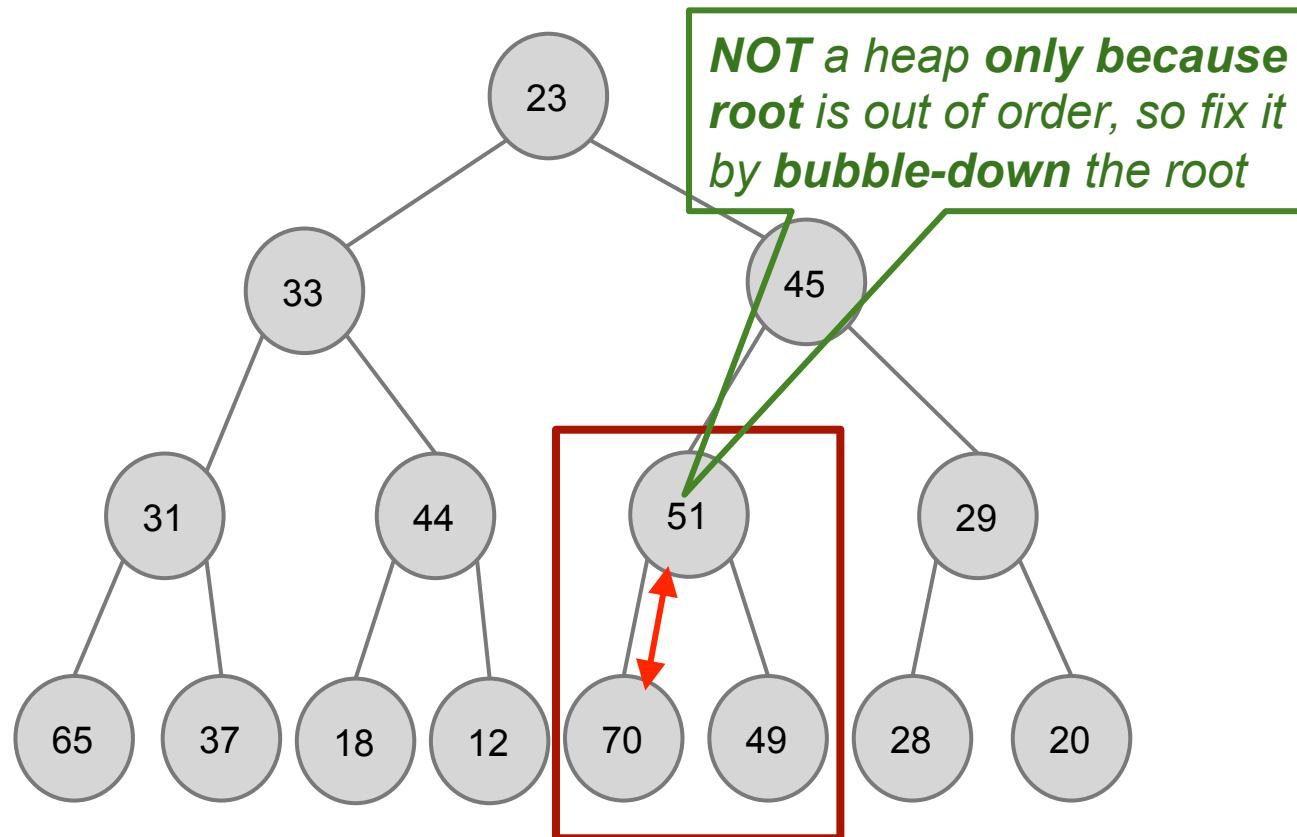
Idea #2

Adjust heap order, from bottom up.



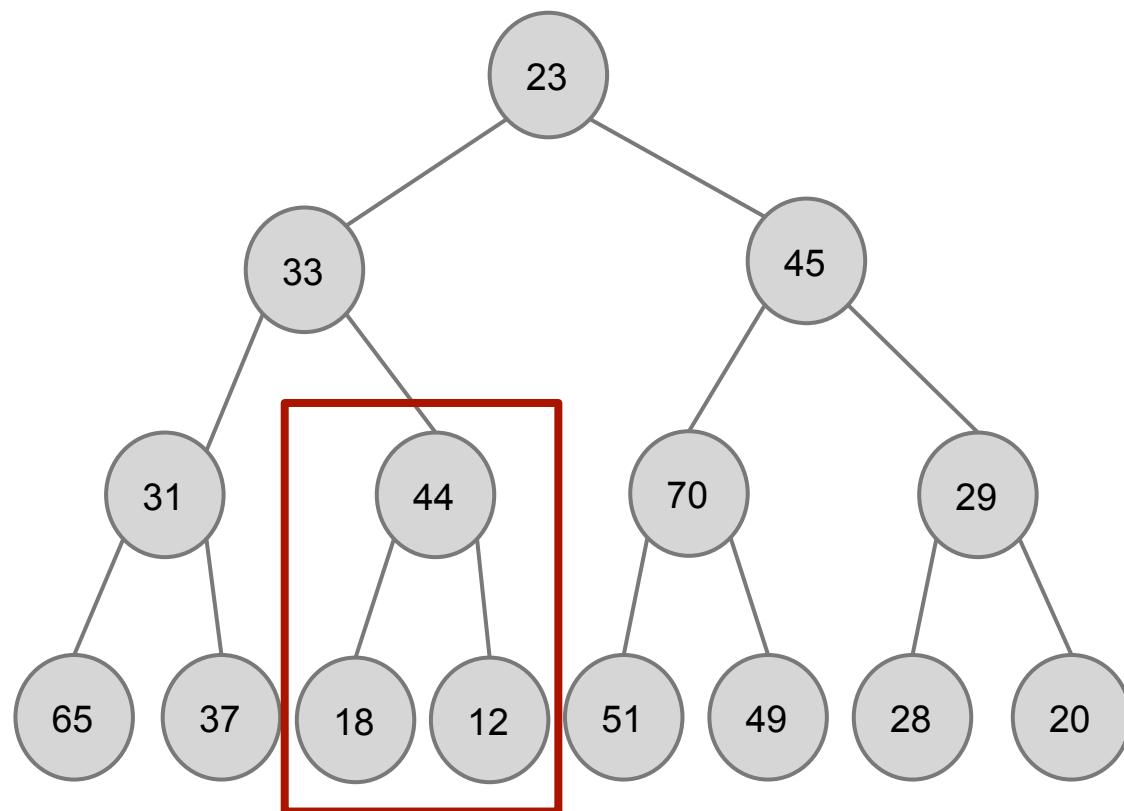
Idea #2

Adjust heap order, from bottom up.



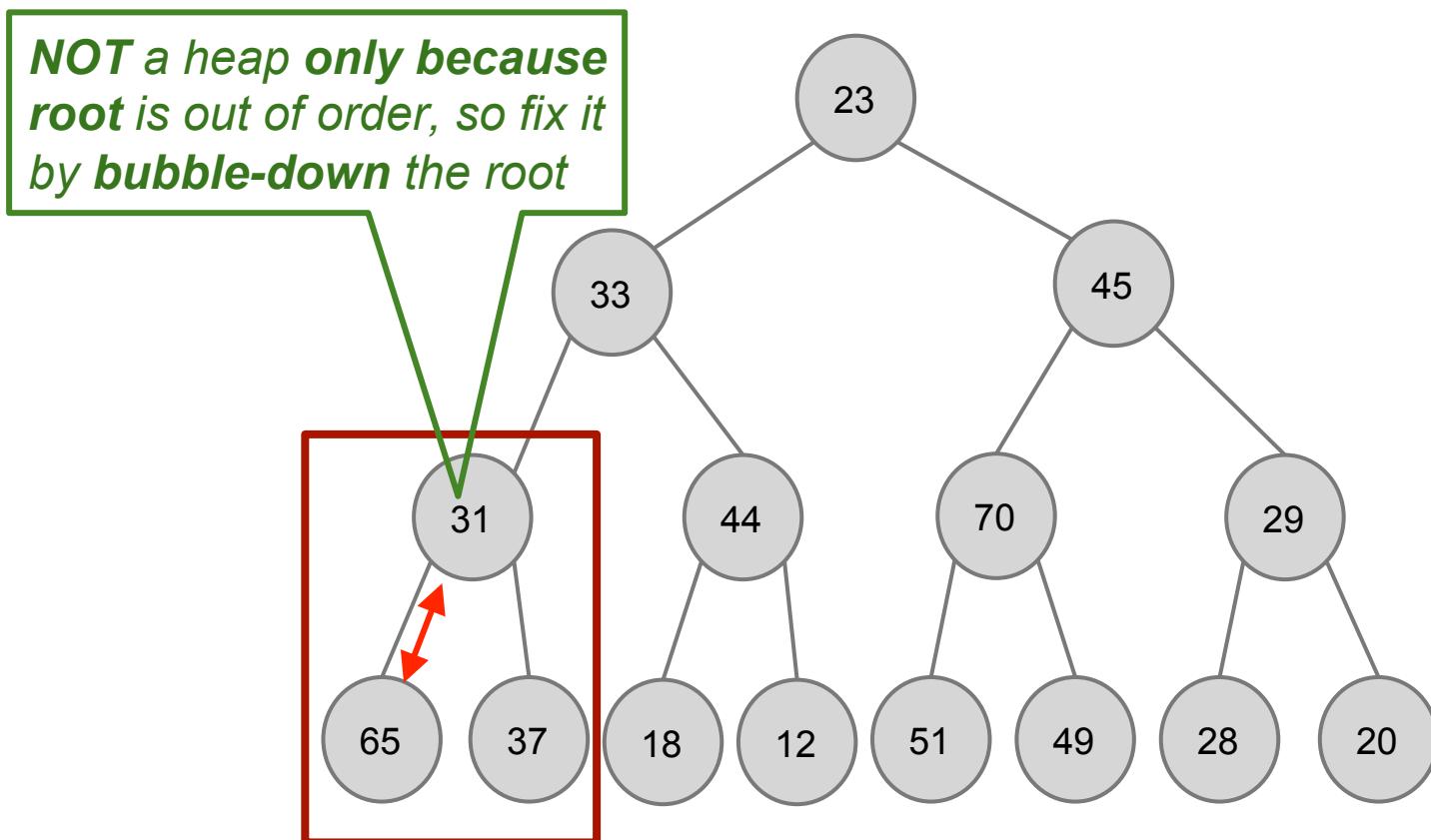
Idea #2

Adjust heap order, from bottom up.



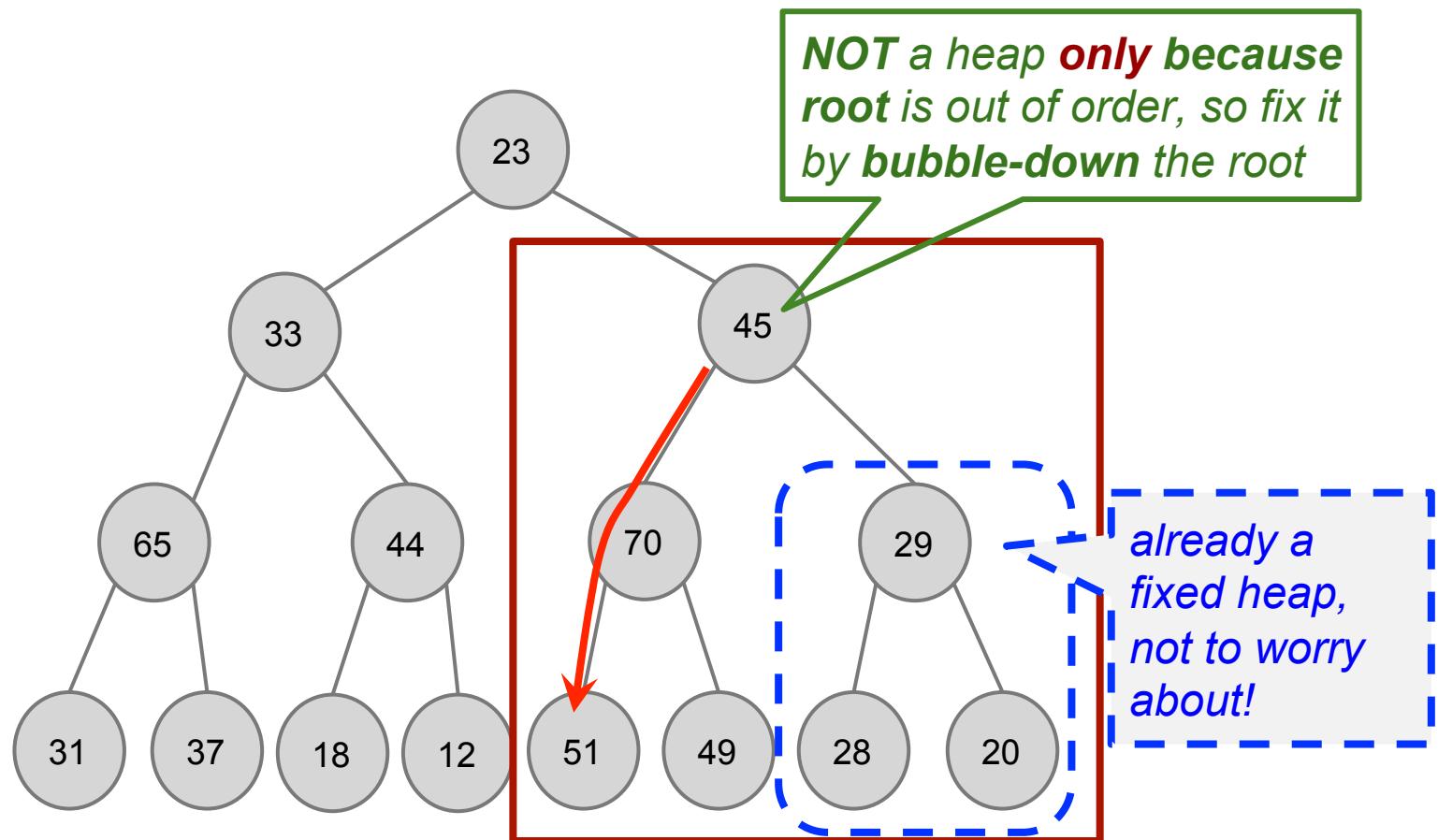
Idea #2

Adjust heap order, from bottom up.



Idea #2

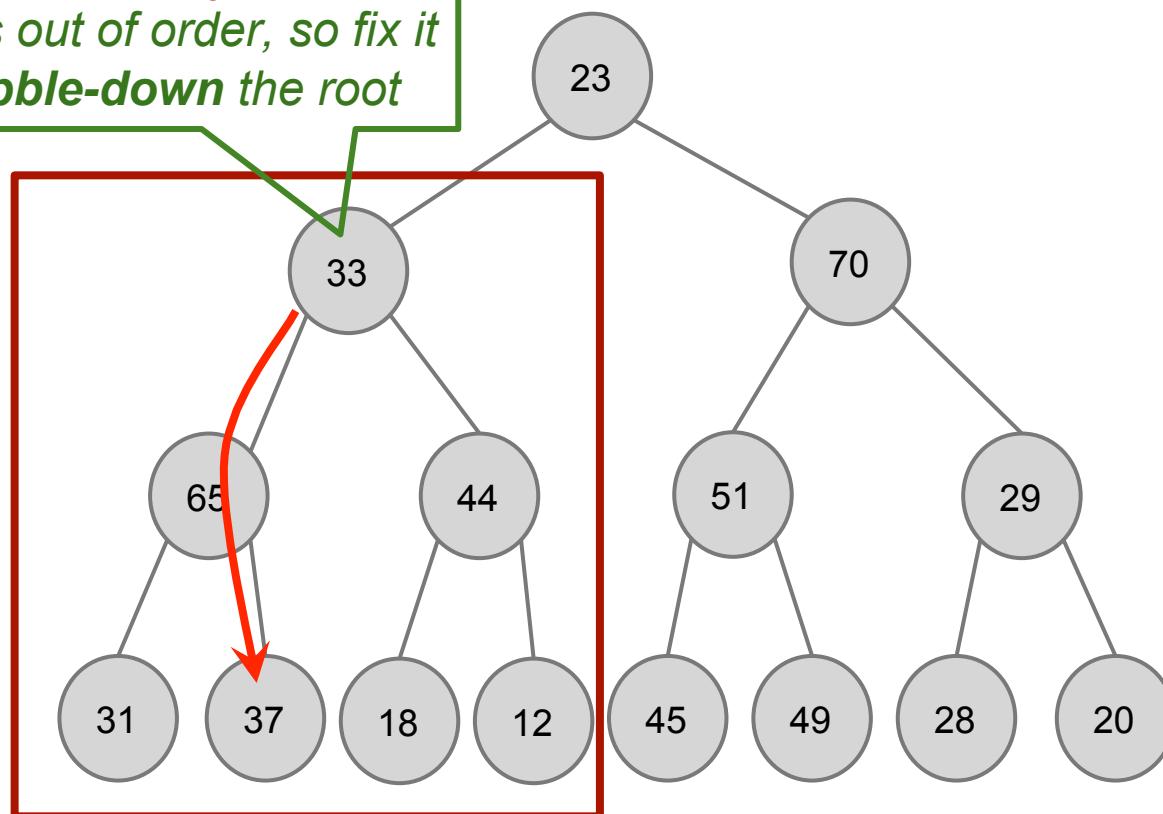
Adjust heap order, from bottom up.



Idea #2

Adjust heap order, from bottom up.

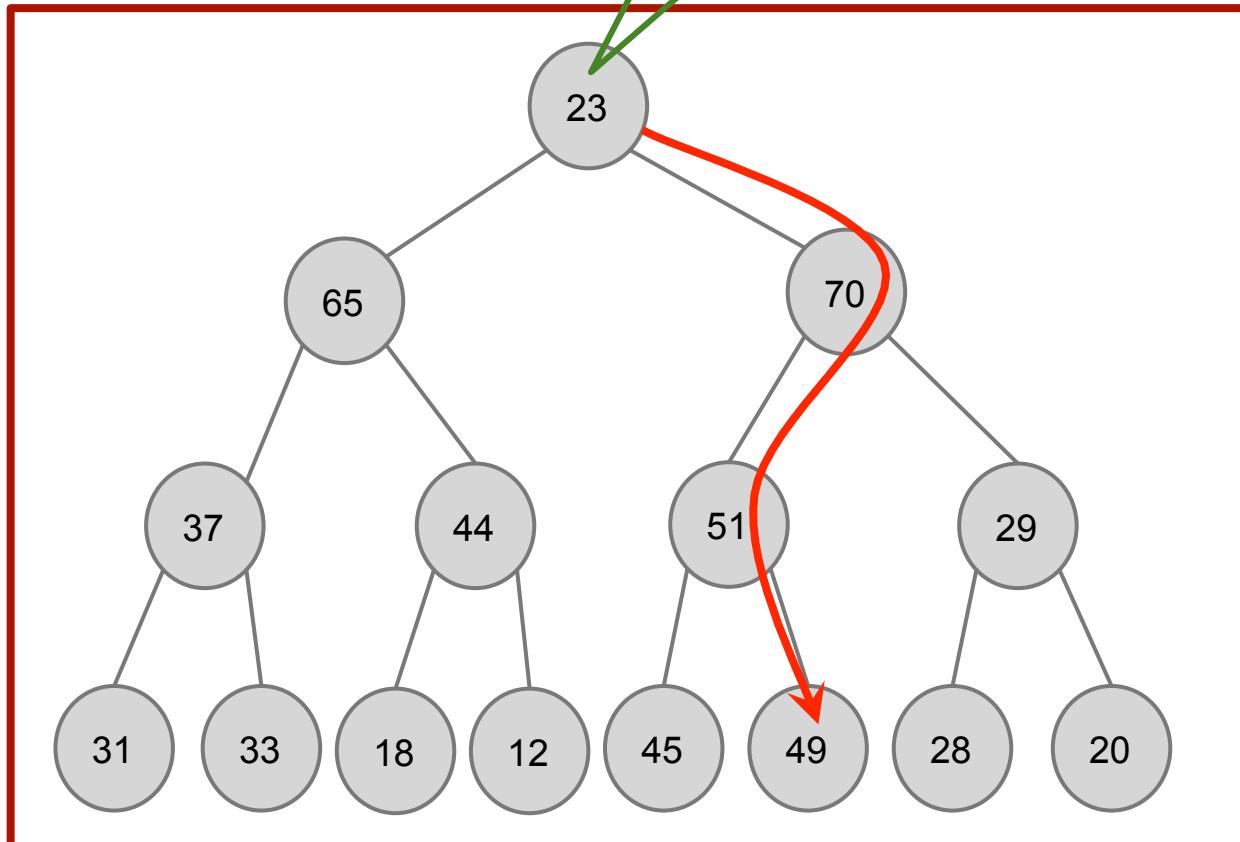
*NOT a heap **only because**
root is out of order, so fix it
by bubble-down the root*



Idea #2

*NOT a heap **only because**
root is out of order, so fix it
by bubble-down the root*

Adjust heap order, from bottom up.

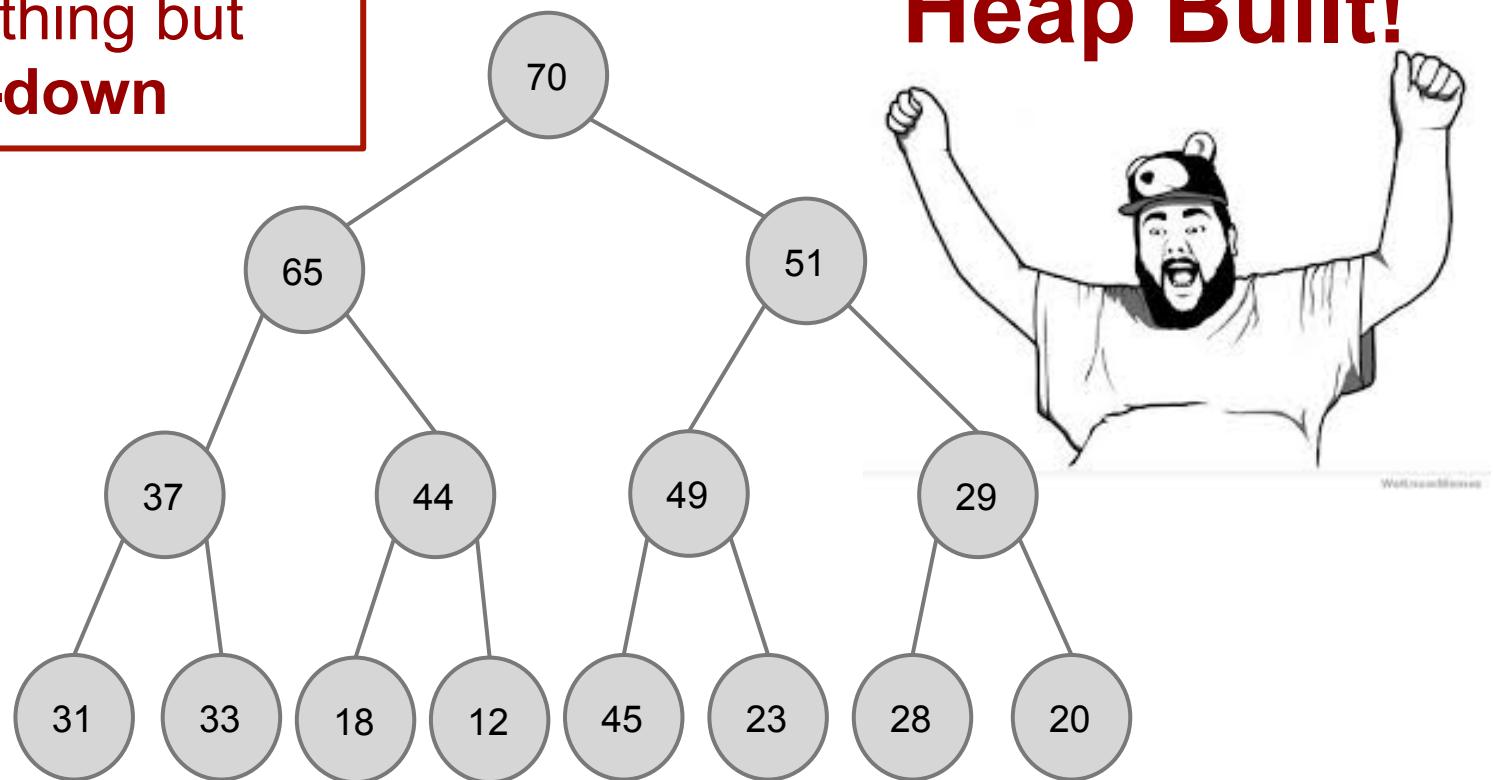


Idea #2

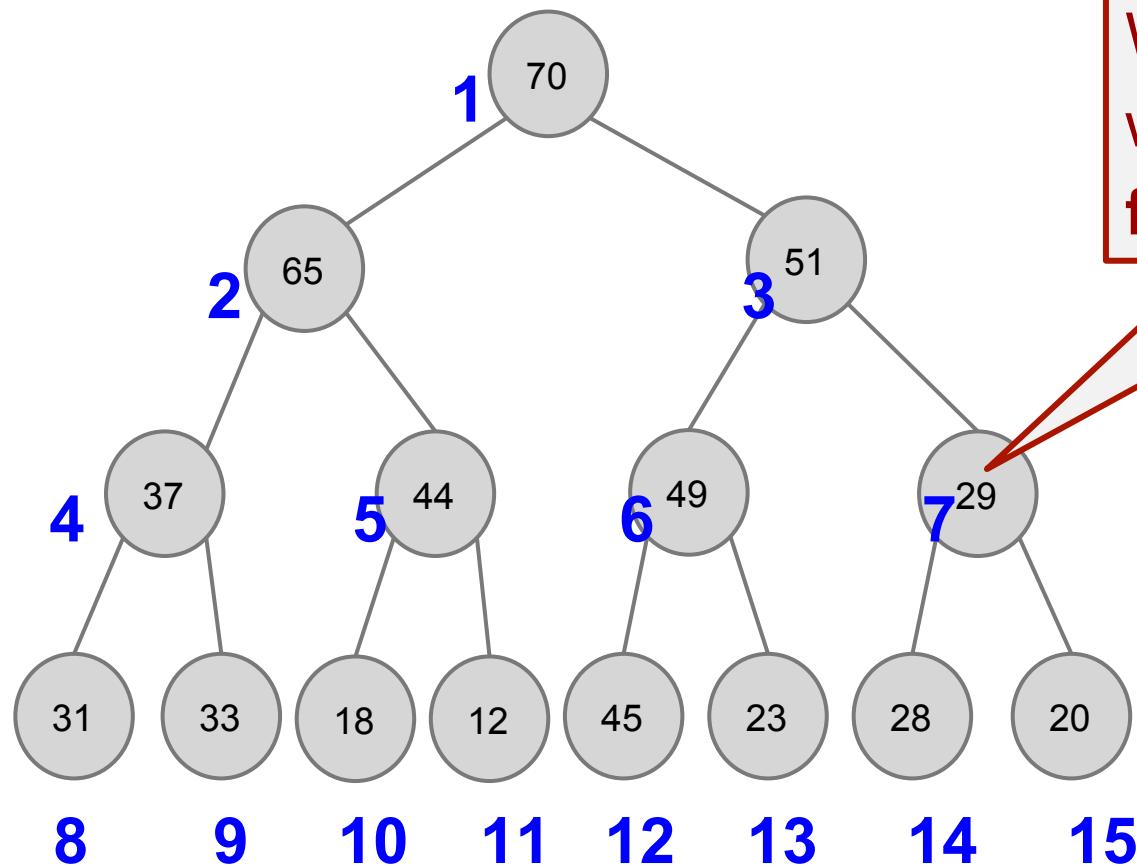
Adjust heap order, from bottom up.

We did nothing but
bubbling-down

Heap Built!

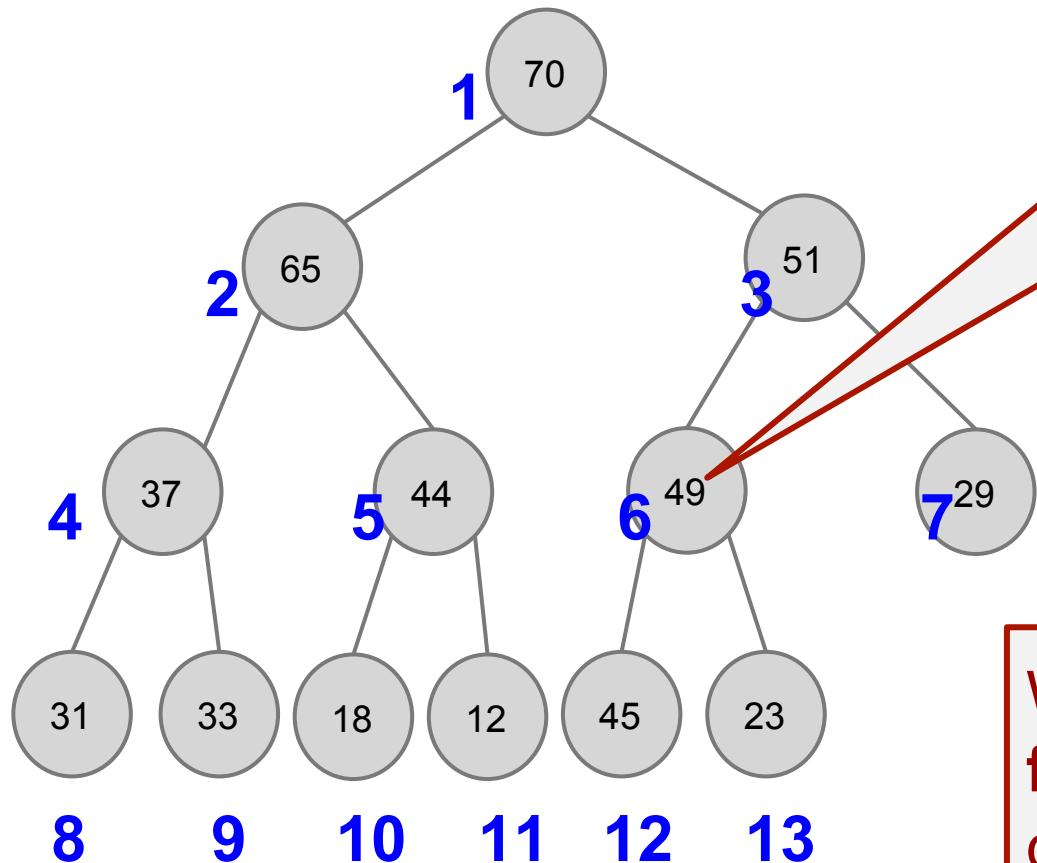


Idea #2: The starting index



We started here,
where the index is
 $\text{floor}(n/2)$

Idea #2: The starting index



Even the bottom level is not fully filled, we still start from **$\text{floor}(n/2)$**

We **always** start from **$\text{floor}(n/2)$** , and go down to 1.

Idea #2: Pseudo-code!

```
BuildMaxHeap(A):
```

```
    for i ← floor(n/2) downto 1:  
        BubbleDown(A, i)
```



Advantages of Idea #2:

- It's in-place, no need for extra array (we did nothing but bubble-down, which is basically swappings).
- It's worst-case running time is **O(n)**, instead of **O(n log n)** of Idea #1.

Why?

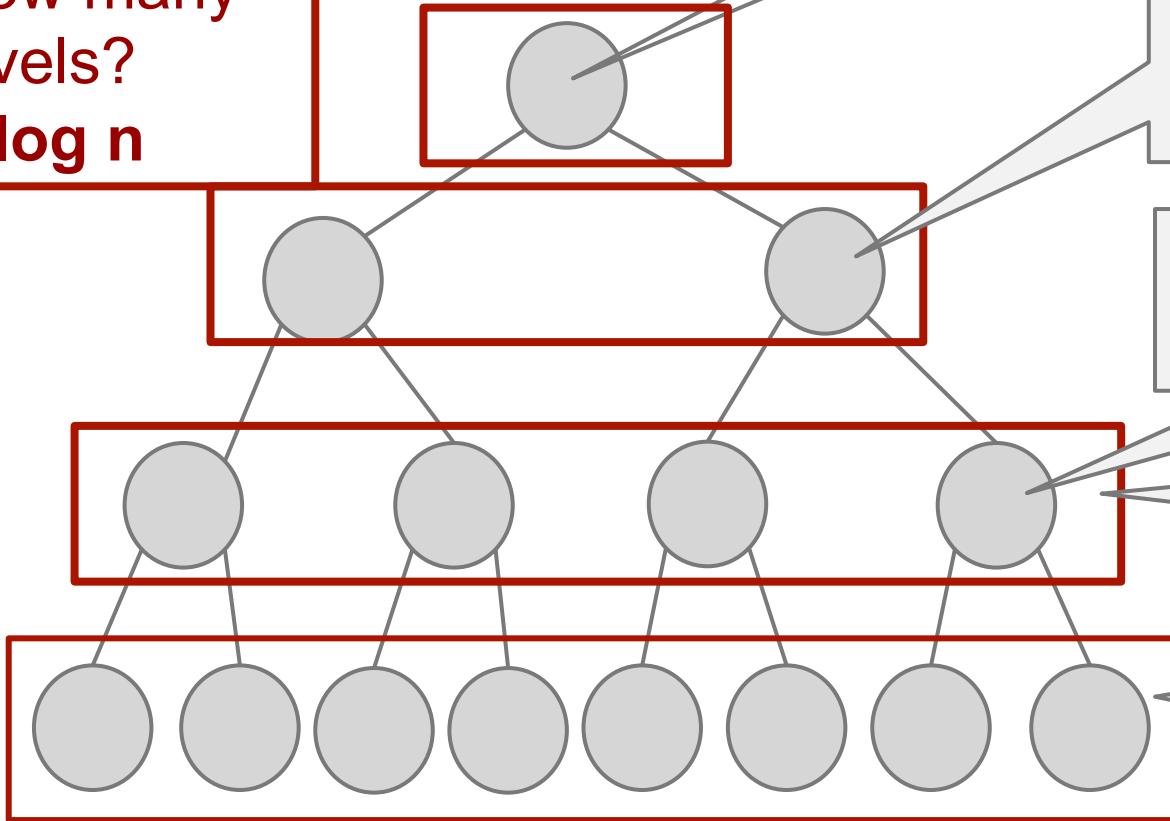
Analysis: Worst-case running time of **BuildMaxHeap(A)**



Intuition

A complete binary tree with n nodes...

How many levels?
 $\sim \log n$



$n/16$ nodes, and # of swaps per bubble-down: ≤ 3

$n/8$ nodes, and # of swaps per bubble-down: ≤ 2

of swaps per bubble-down: ≤ 1

$\sim n/4$ nodes

$\sim n/2$ nodes, and no work done at this level.

So, total number of swaps

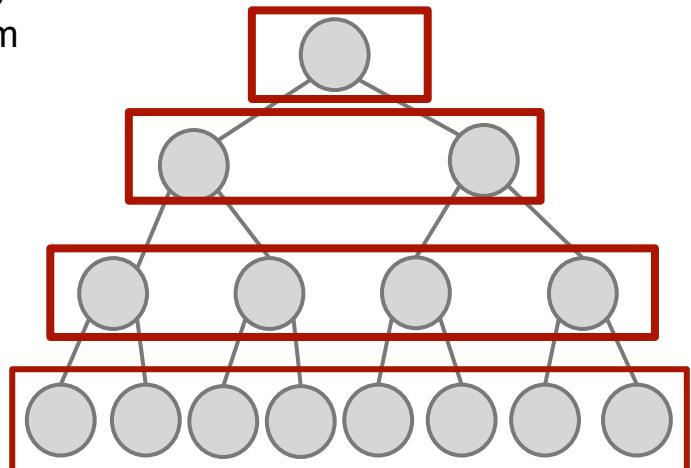
$$T(n) = 1 \cdot \frac{n}{4} + 2 \cdot \frac{n}{8} + 3 \cdot \frac{n}{16} + \dots$$

$$= \sum_{i=1}^{\log n} i \cdot \frac{n}{2^{i+1}} \leq \sum_{i=1}^{+\infty} i \cdot \frac{n}{2^{i+1}}$$

$$= n \sum_{i=1}^{+\infty} \frac{i}{2^{i+1}}$$

same trick as
Week 1's sum

$$= n$$



$$\sum_{i=0,1,\dots} i/2^i = \sum_{k=0,1,\dots} k x^k ,$$

when $x=1/2$

$$\sum_{k=0,1,\dots} k x^k = x/(1-x)^2$$

$$\text{So } \sum_{i=0,1,\dots} i/2^i = 1/2/(1-1/2)^2 = 2$$

A close-up portrait of Agent Smith, a bald man with a serious expression, wearing dark sunglasses. The background is dark and out of focus.

BUILD MAX HEAP

YOU CAN DO IN LINEAR TIME

Summary

HeapSort(A):

- Sort a heap-ordered array in-place
- $O(n \log n)$ worst-case running time

BuildMaxHeap(A):

- Convert an unsorted array into a heap, in-place
- Fix heap property from bottom up, do bubbling down on each sub-root
- $O(n)$ worst-case running time

Algorithm visualizer

<http://visualgo.net/heap.html>

Next week

→ ADT: Dictionary

→ Data structure: Binary Search Tree

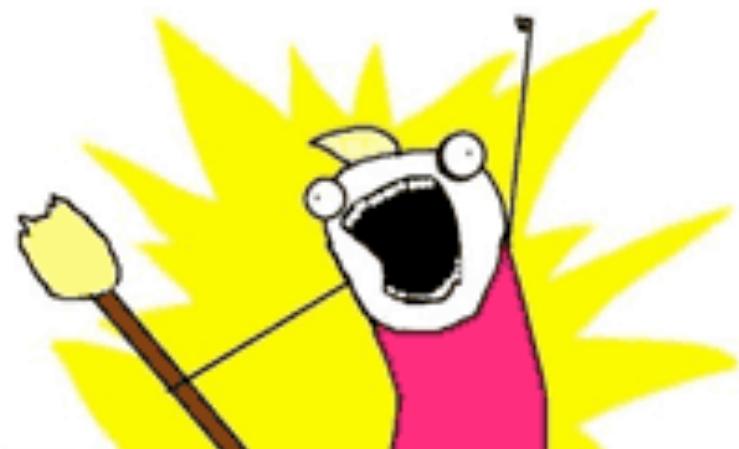
CSC263 Week 3

Announcements

Problem Set 2 is out today!

Due Tuesday (Oct 13)

More challenging so start early!



NOT EVERY GROUP PROJECT



IN SCHOOL YOU HAVE EVER DONE

This week

→ ADT: Dictionary

→ Data structure:

- ◆ Binary search tree (BST)
- ◆ Balanced BST - AVL tree

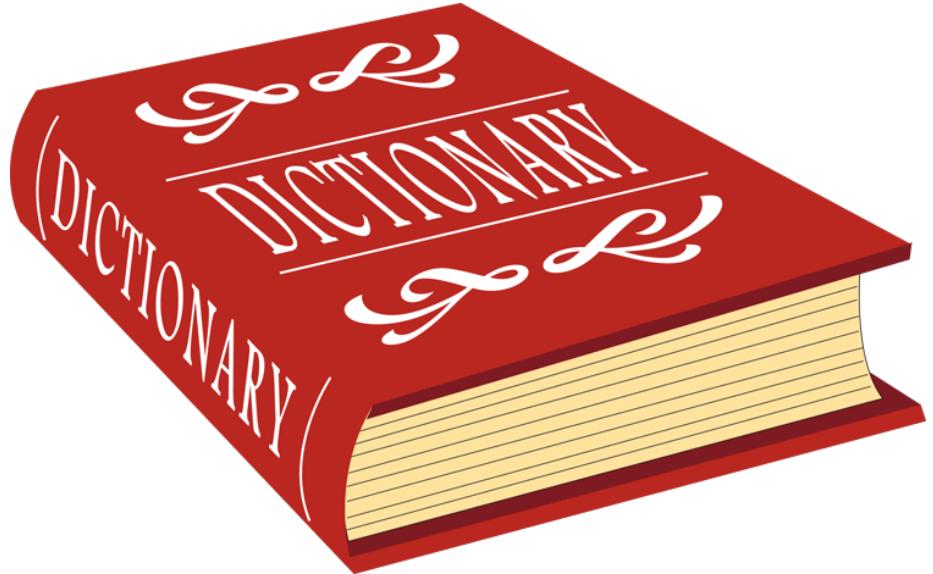
Dictionary

What's stored:

→words

Supported operations

- Search for a word
- Insert a word
- Delete a word



Dictionary, more precisely

What's stored

- A set **S** where each node **x** has a field **x.key**
(assumption: keys are distinct, unless o.w. specified)

Supported operations

- **Search(S, k)**: return **x** in **S**, s.t., **x.key = k**
 - ◆ return NIL if no such **x**
- **Insert(S, x)**: insert node **x** into **S**
 - ◆ if already exists node **y** with same key , replace **y** with **x**
- **Delete(S, x)**: delete a given **node x** from **S**

A thing to note: **k** is a key, **x** is a node.

More on Delete

Why $\text{Delete}(S, \mathbf{x})$ instead of $\text{Delete}(S, \mathbf{k})$?

$\text{Delete}(S, \mathbf{k})$ can be implemented by:

1. $x = \text{Search}(S, k)$
2. $\text{Delete}(S, x)$

We want separate different operations, i.e., each operation focuses on only one job.

**Implement a Dictionary using
simple data structures**

40 -> 33 -> 18 -> 65 -> 24 -> 25

Unsorted (doubly) linked list

→**Search(S, k)**

- ◆ **O(n)** worst case
- ◆ go through the list to find the key

→**Insert(S, x)**

- ◆ **O(n)** worst case
- ◆ need to check if **x.key** is already in the list

→**Delete(S, x)**

- ◆ **O(1)** worst case
- ◆ Just delete, **O(1)** in a doubly linked list

Sorted array [18 , 24 , 25 , 33 , 40 , 65]

→ **Search(S, k)**

- ◆ **O(log n)** worst case
- ◆ binary search!

→ **Insert(S, x)**

- ◆ **O(n)** worst case
- ◆ insert at front, everything has to shift to back

→ **Delete(S, x)**

- ◆ **O(n)** worst case
- ◆ Delete at front, everything has to shift to front

We can do better using smarter data structures, of course

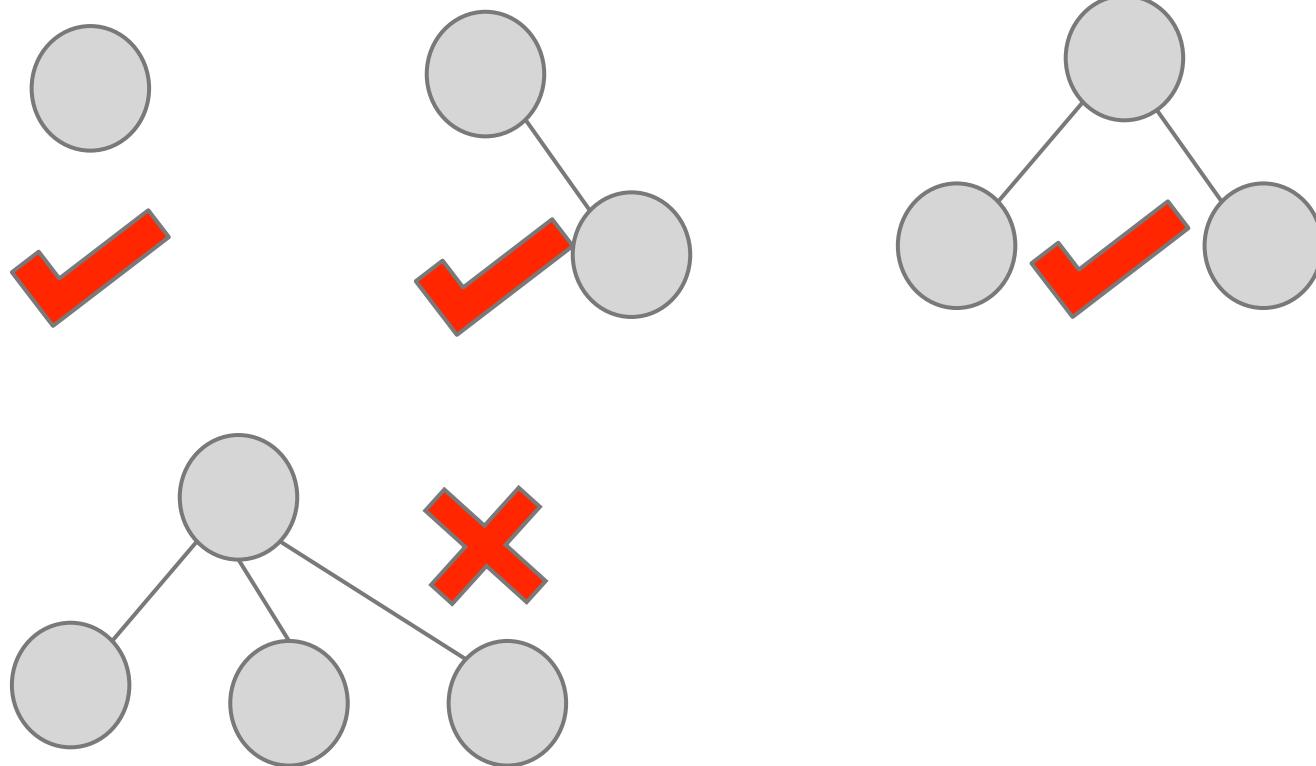
	unsorted list	sorted array	BST	Balanced BST
Search(S , k)	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$
Insert(S , x)	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
Delete(S , x)	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$



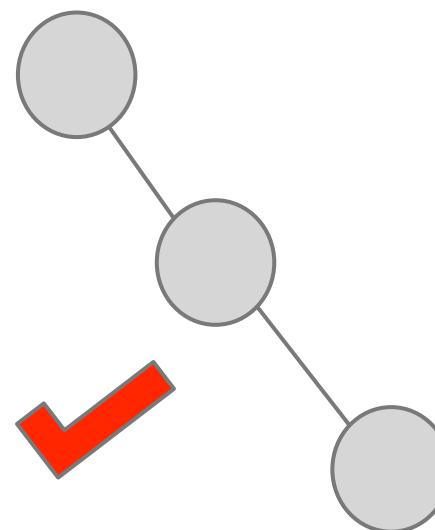
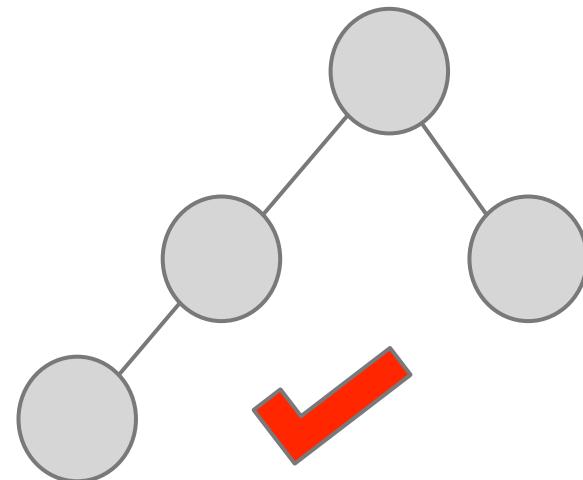
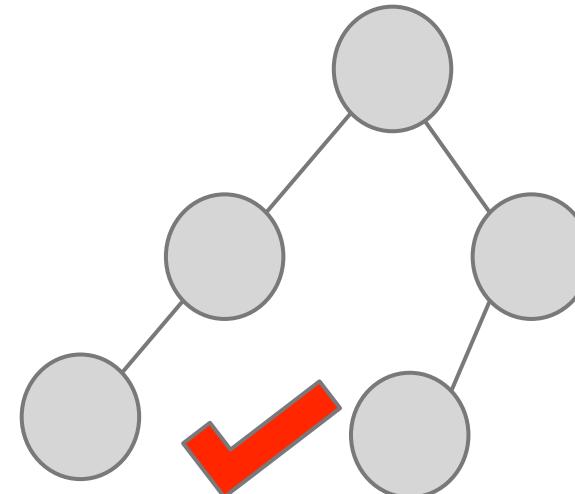
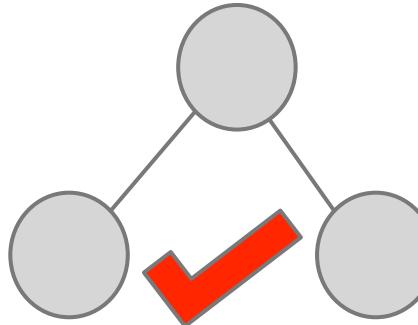
Binary Search Tree

It's a **binary tree, like binary heap**

Each node has at **most** 2 children

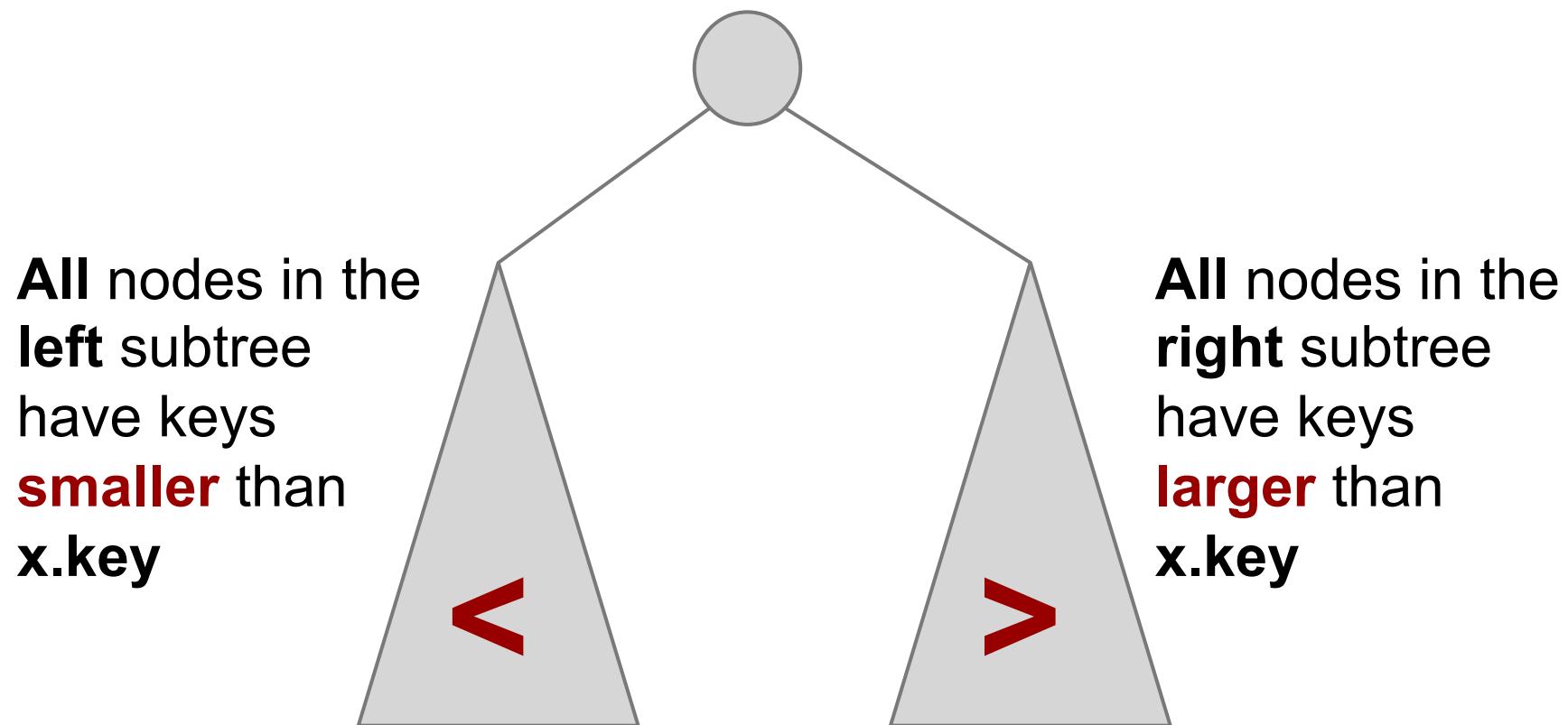


**need NOT be nearly-complete, unlike
binary heap**

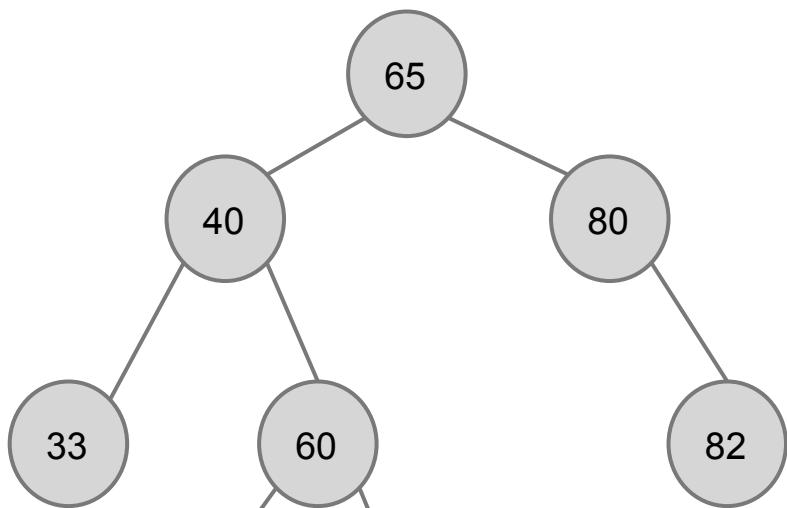
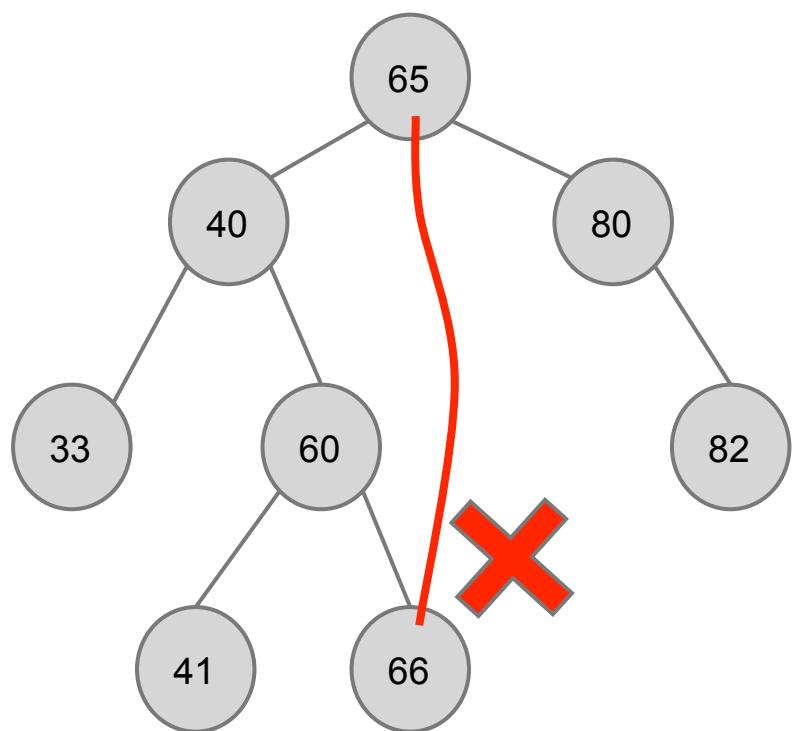


It has the **BST** property

For **every** node **x** in the tree



BST or NOT?



**Because of BST property, we can say that
the keys in a BST are sorted.**

**CSC148 Quiz: How to obtain a sorted list
from a BST?**

Perform an inorder traversal.

We pass a BST to a function by passing its **root** node.

```
InorderTraversal(x):  
# print all keys in BST rooted at x in ascending order  
  
if x ≠ NIL:  
    InorderTraversal(x.left)  
    print x.key  
    InorderTraversal(x.right)
```

Worst case running time of InorderTraversal:
O(n), because visit each node exactly once.

Operations on a BST

First, information at each node x

→`x.key`: the key

→`x.left`: the left child (node)

→`x.right`: the right child (node)

→`x.p`: the parent (node)

Operations on a BST

read-only operations

- TreeSearch(root, k)
- TreeMinimum(x) / TreeMaximum(x)
- Successor(x) / Predecessor(x)

modifying operations

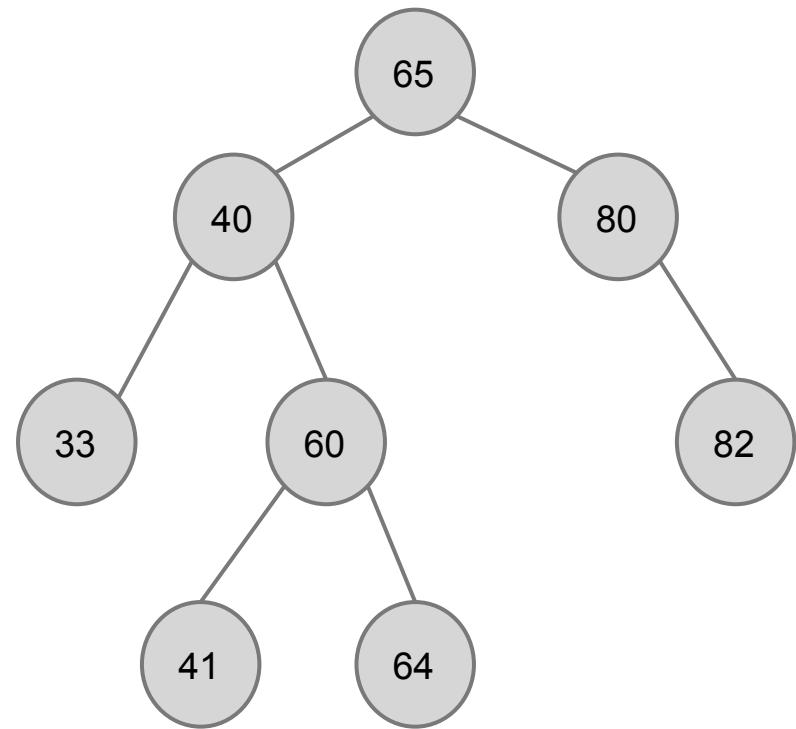
- TreeInsert(root, x)
- TreeDelete(root, x)

TreeSearch(root, k)

Search the BST rooted at root, return the node with key k; return NIL if not exist.

TreeSearch(root, k)

- start from root
- if k is **smaller** than the key of the current node, go **left**
- if k is **larger** than the key of the current node, go **right**
- if equal, **found**
- if going to **NIL**, **not found**



TreeSearch(root, k): Pseudo-code

```
TreeSearch(root, k):  
  
    if root = NIL or k = root.key:  
        return root  
    if k < root.key:  
        return TreeSearch(root.left, k)  
    else:  
        return TreeSearch(root.right, k)
```

Worst case running time:

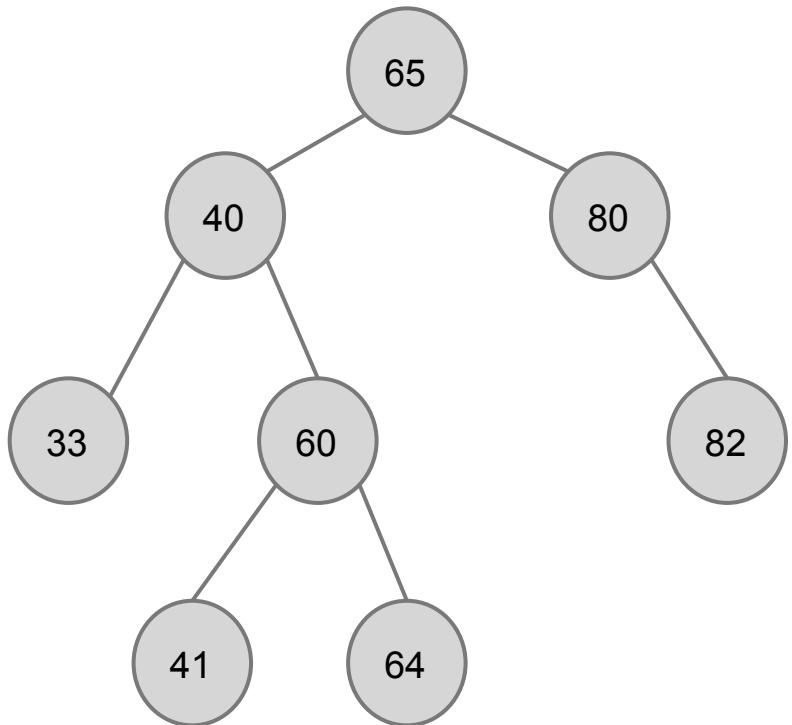
O(h), height of tree, going at most from root to leaf

TreeMinimum(x)

Return the node with the minimum key of the tree rooted at x

TreeMinimum(x)

- start from root
- keep going to the left, until cannot go anymore
- return that final node



TreeMinimum(x): pseudo-code

```
TreeMinimum(x):  
  
    while x.left ≠ NIL:  
        x ← x.left  
    return x
```

Worst case running time:

O(h), height of tree, going at most from root to leaf

TreeMaximum(x) is exactly the same, except that it goes to the right instead of to the left.

Successor(x)

Find the node which is the successor of x in the sorted list obtained by inorder traversal

or, node with the smallest key larger than x

Successor(x)

→ The successor of 33 is...

- ◆ 40

→ The successor of 40 is...

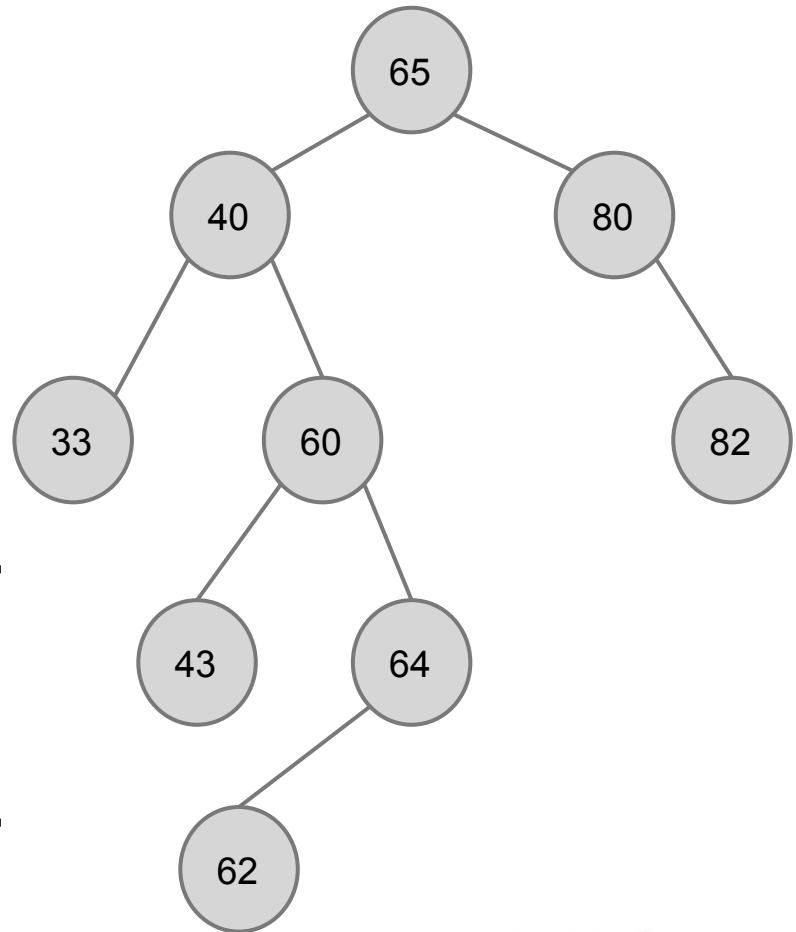
- ◆ 43

→ The successor of 64 is...

- ◆ 65

→ The successor of 65 is ...

- ◆ 80



Successor(x): Organize into two cases

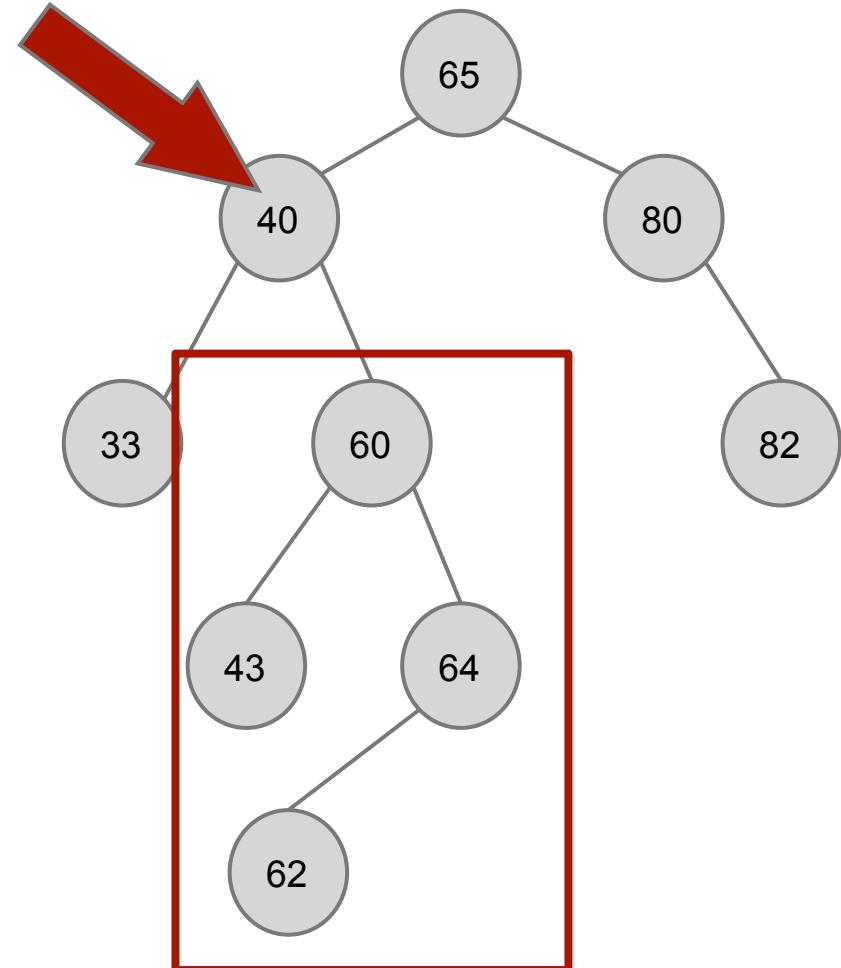
- x has a right child
- x does not have a right child

x has a right child

Successor(x) must be in x 's **right subtree** (the nodes **right after x** in the inorder traversal)

It's the **minimum** of x 's right subtree, i.e.,
`TreeMinimum($x.right$)`

The first (smallest) node among what's right after x .



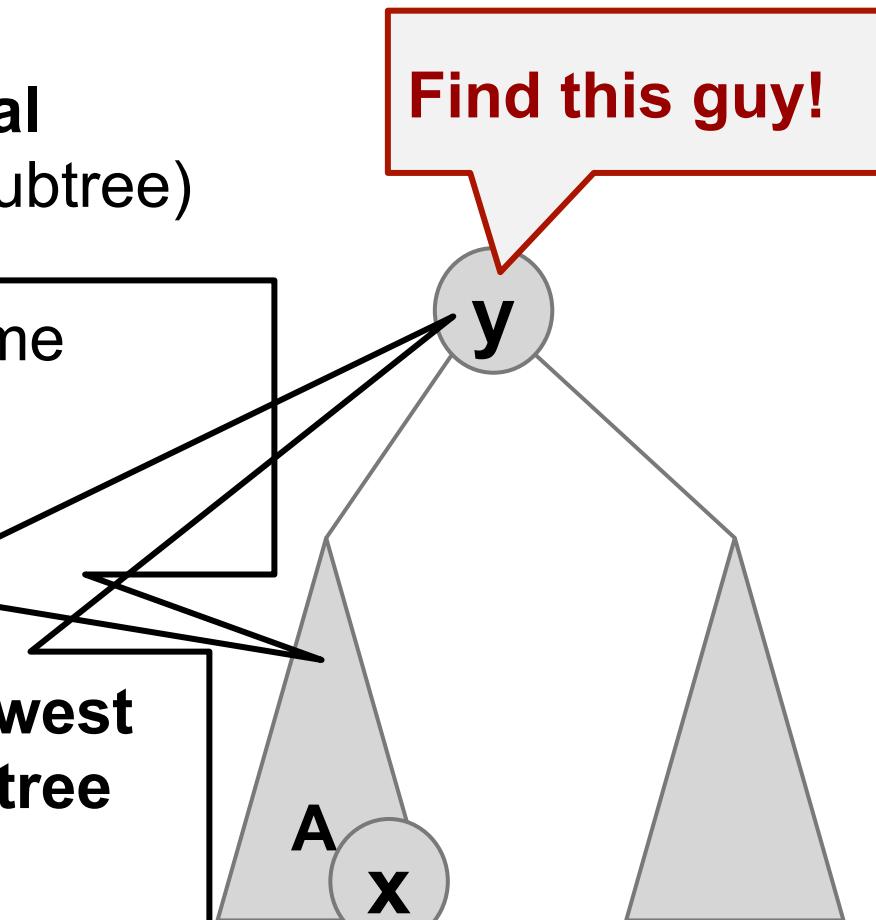
x does not have a right child

Consider the **inorder traversal**
(left subtree -> root -> right subtree)

x is the **last one** visited in some
left subtree A
(because no right child)

The successor **y** of **x** is the **lowest ancestor** of **x** whose **left subtree** contains **x**
(**y** is visited right after finishing
subtree **A** in inorder traversal)

Find this guy!



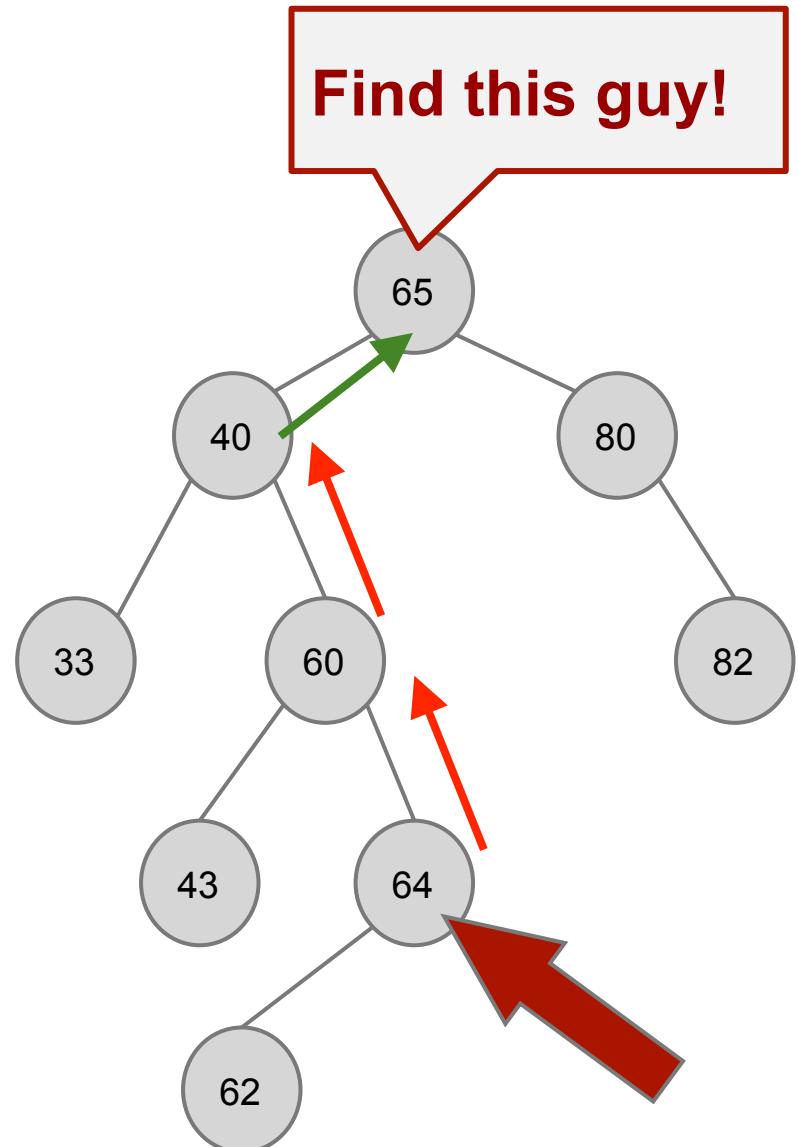
x does not have a right child

How to find:

→ go up to x.p

→ if x is a **right** child of
x.p, keep going up

→ if x is a **left** child of
x.p, stop, x.p is the
guy!



Summarize the two cases of Successor(x)

→ If x has a right child

- ◆ return TreeMinimum(x.right)

→ If x does not have a right child

- ◆ keep going up to x.p while x is a right child, stop when x is a left child, then return x.p
- ◆ if already gone up to the root and still not finding it, return NIL.

Successor(x): pseudo-code

```
Successor(x):
    if x.right ≠ NIL:
        return TreeMinimum(x.right)
    y ← x.p
    while y ≠ NIL and x = y.right: #x is right child
        x = y
        y = y.p # keep going up
    return y
```

Worst case running time

O(h), Case 1: TreeMin is O(log n); Case 2: at most leaf to root

$\text{Predecessor}(x)$ works symmetrically the same way as $\text{Successor}(x)$

TreeInsert(root, x)

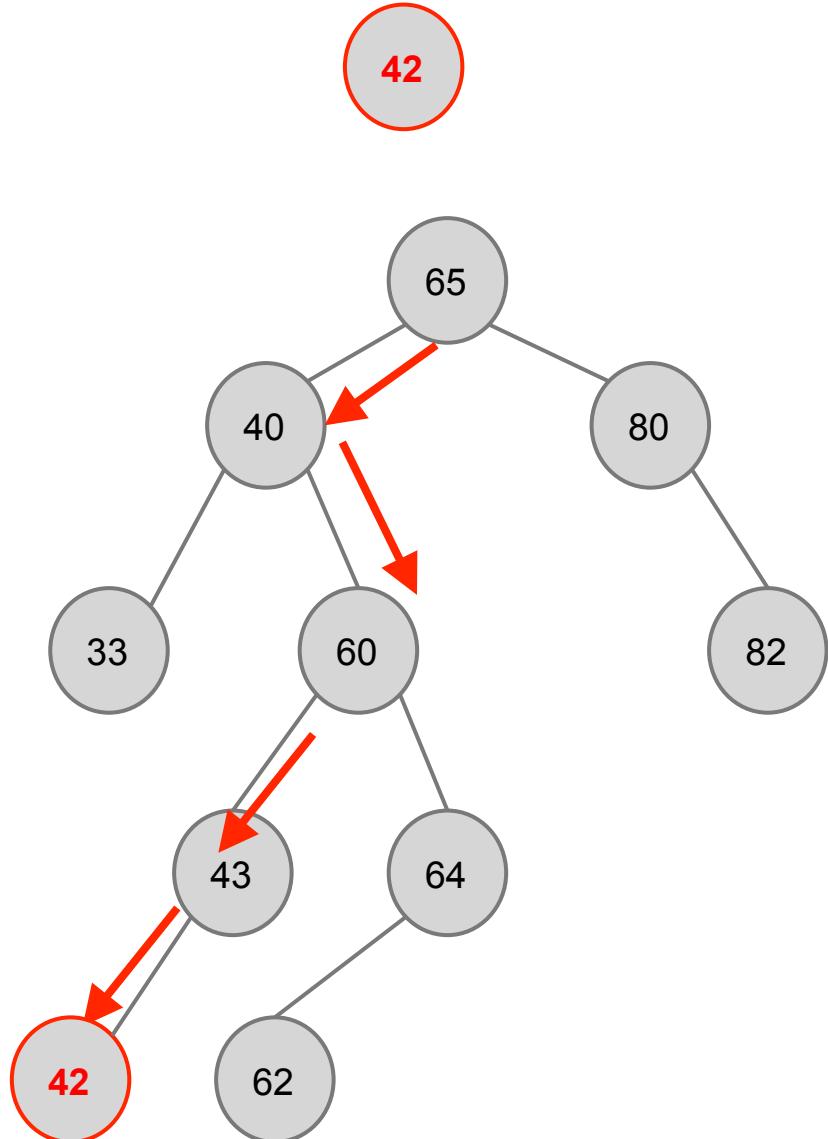
Insert node x into the BST rooted at root
return the new root of the modified tree
if exists y, s.t. y.key = x.key, replace y with x

TreeInsert(root, x)

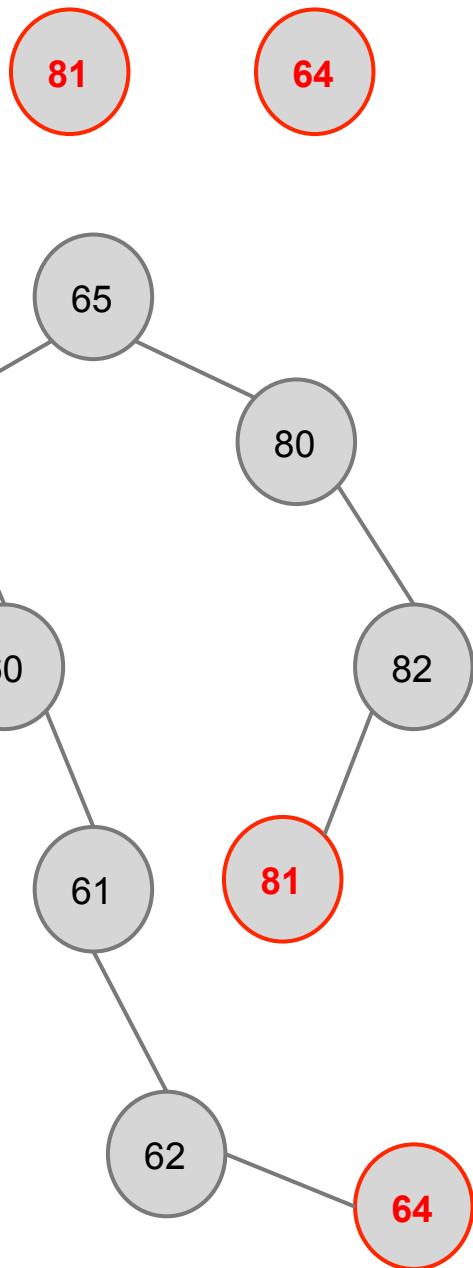
Go down, left and right
like what we do in
TreeSearch

When next position is
NIL, insert there

If find equal key, replace
the node



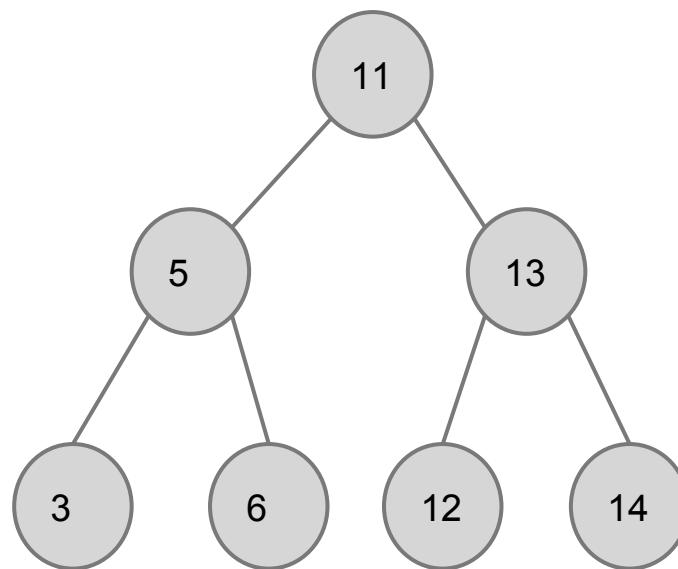
Exercise



Ex 2: Insert sequence into an empty tree

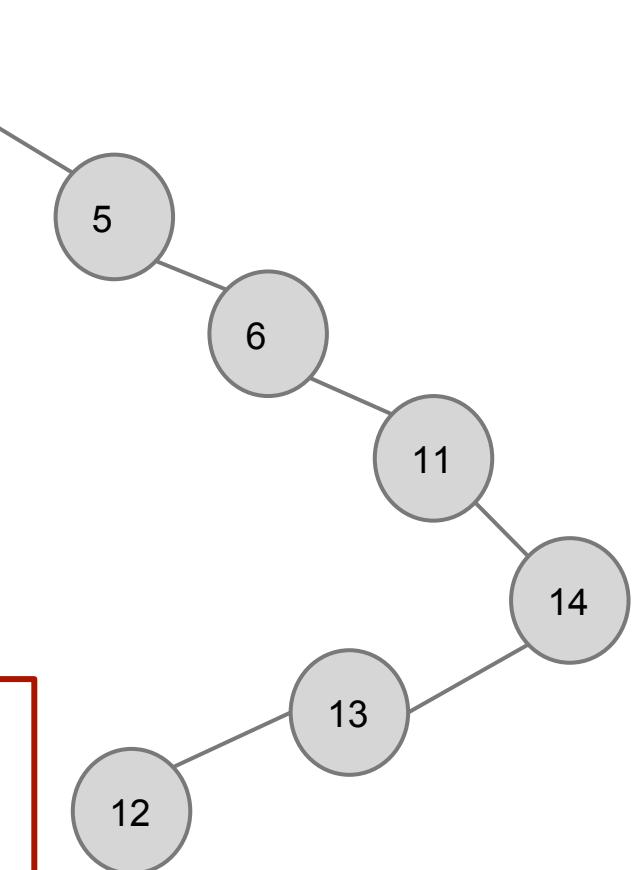
Insert sequence 1:

11, 5, 13, 12, 6, 3, 14



Insert sequence 2:

3, 5, 6, 11, 14, 13, 12



Different insert sequences result in different “shapes” (heights) of the BST.

TreeInsert(root, x): Pseudo-code

```
TreeInsert(root, x):
# insert and return the new root
    if root = NIL:
        root ← x
    elif x.key < root.key:
        root.left ← TreeInsert(root.left, x)
    elif x.key > root.key:
        root.right ← TreeInsert(root.right, x)
    else # x.key = root.key:
        replace root with x # update x.left, x.right
    return root
```

**Worst case
running time:
 $O(h)$**

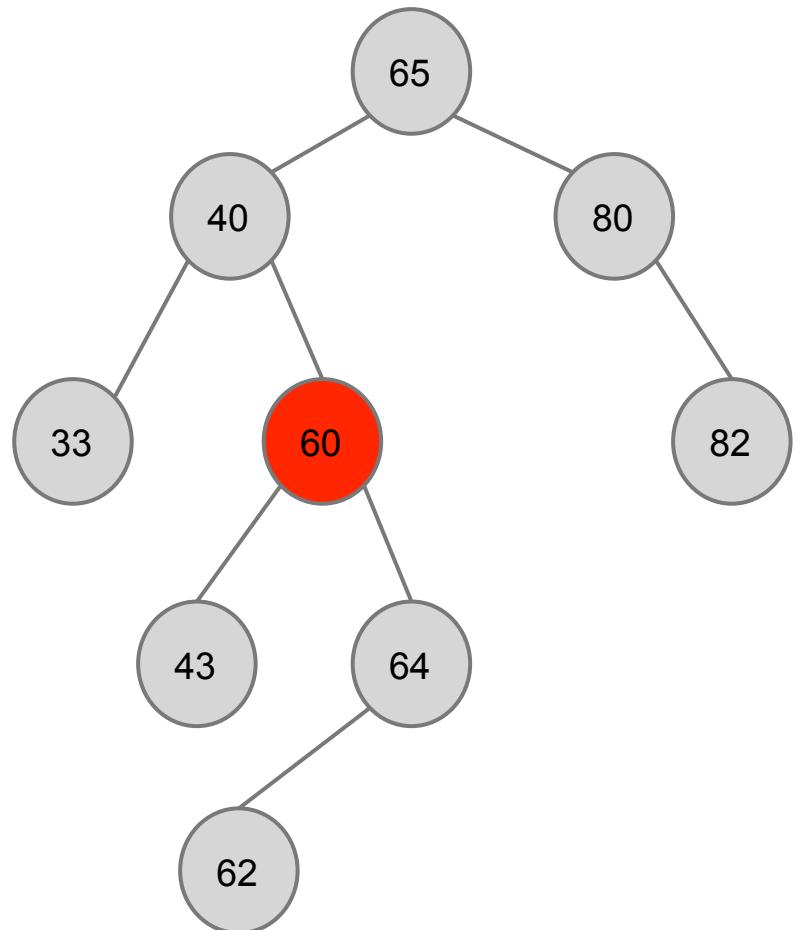


TreeDelete(root, x)

Delete node x from BST rooted at root while
maintaining BST property,
return the new root of the modified tree

What's tricky about deletion?

Tree might be **disconnected** after deleting a node, need to **connect** them back together, while maintaining the **BST property**.



Delete(root, x): Organize into 3 cases

Case 1: x has **no** child

Easy

Case 2: x has **one** child

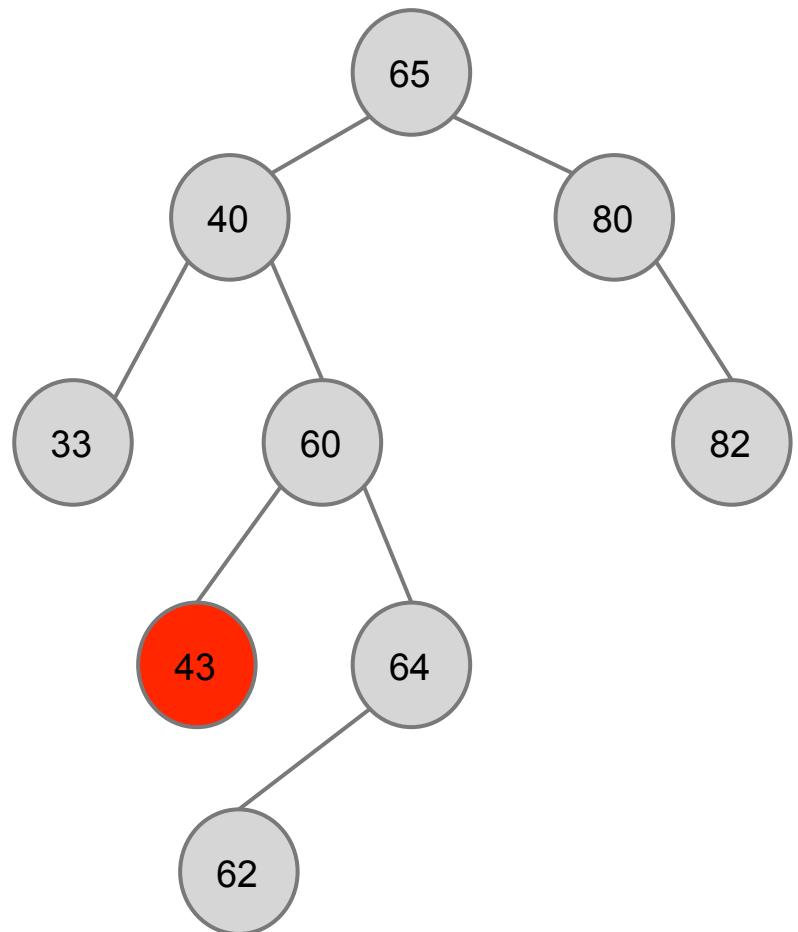
Easy

Case 3: x has **two** children

less easy

Case 1: x has no child

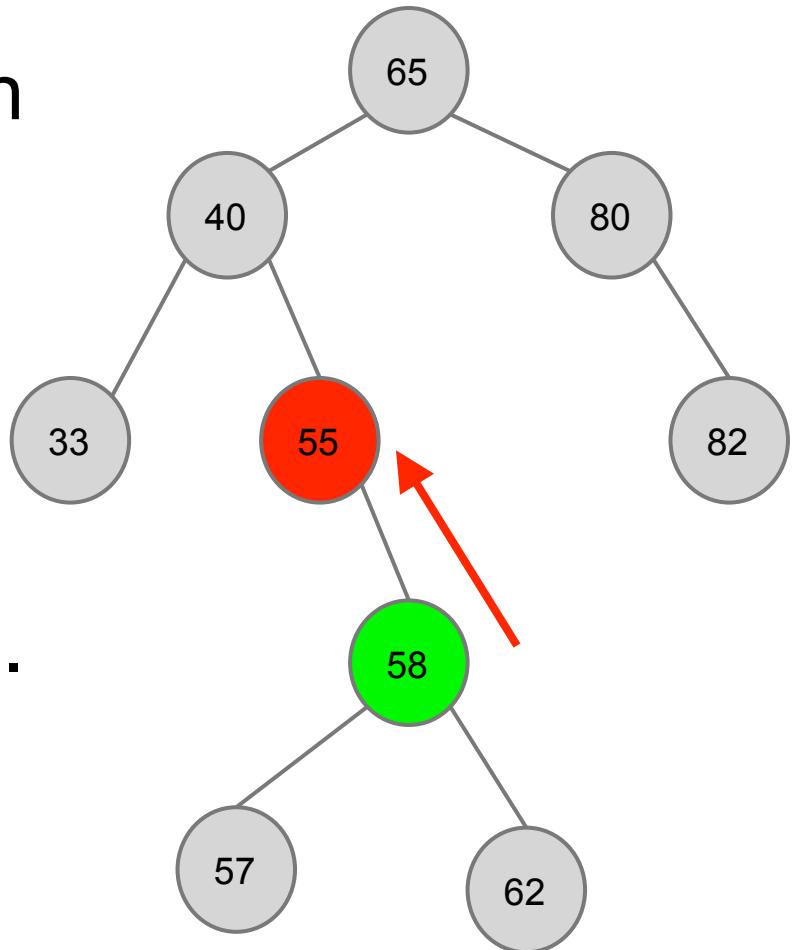
Just delete it,
nothing else need
to be changed.



Case 2: x has one child

First delete that node, which makes an **open spot**.

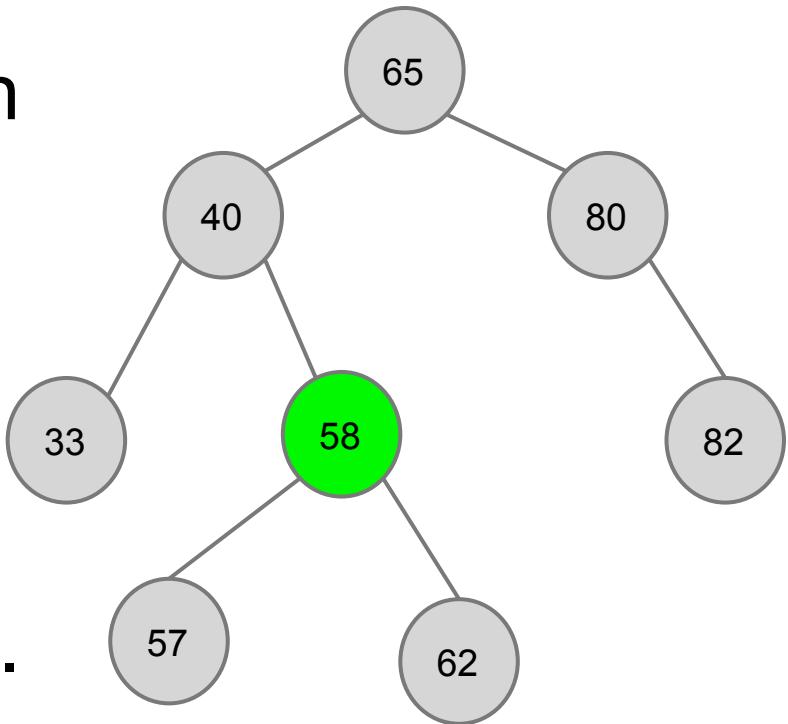
Then **promote x's only child** to the spot, together with the only child's subtree.



Case 2: x has one child

First delete that node, which makes an **open spot**.

Then **promote x's only child** to the spot, together with the only child's subtree.

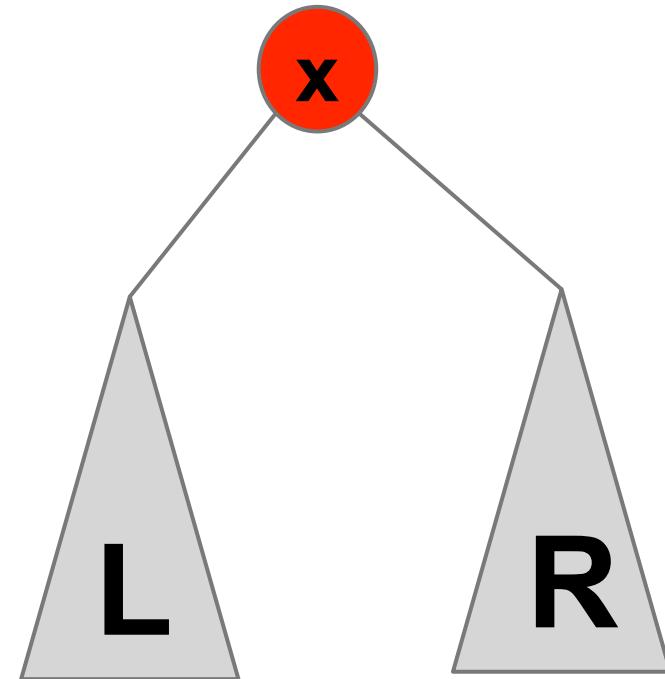


Case 3: x has two children

Delete x , which makes an open spot.

A node y should fill this spot,
such that $L < y < R$,

Who should be y ?



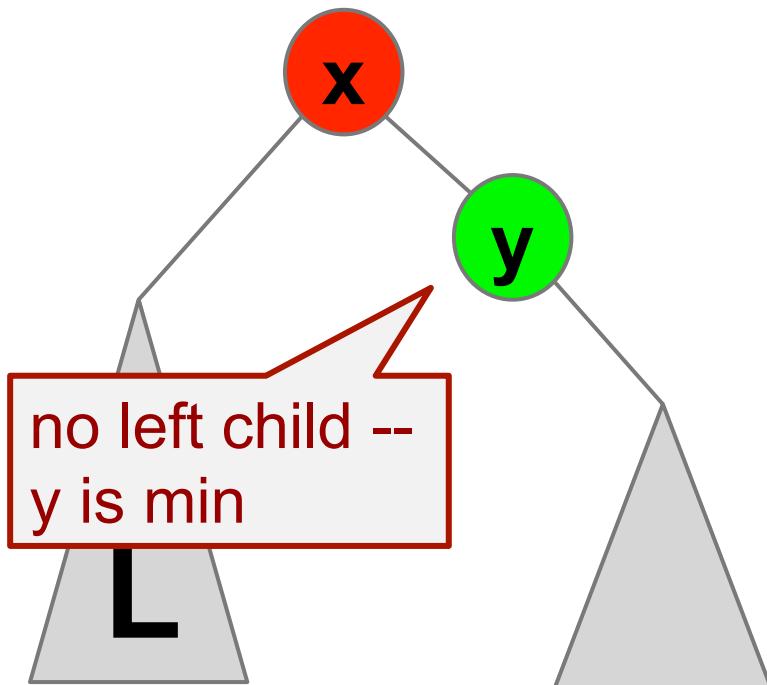
$y \leftarrow$ the minimum of R , i.e., Successor(x)

$L < y$ because y is in R , $y < R$ because it's minimum

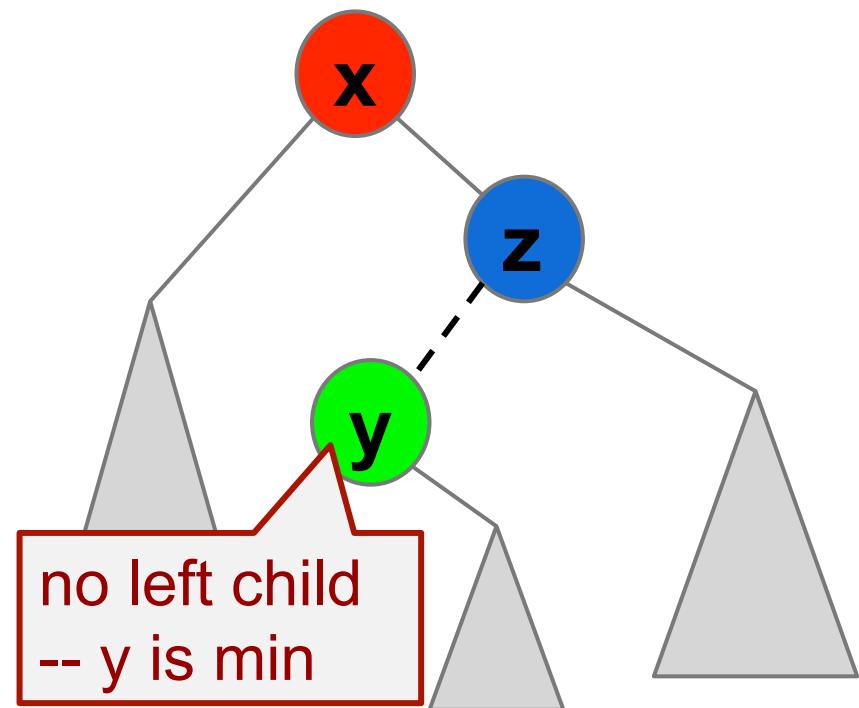
Further divide into two cases

Case 3.1:

y happens to be the right child of x

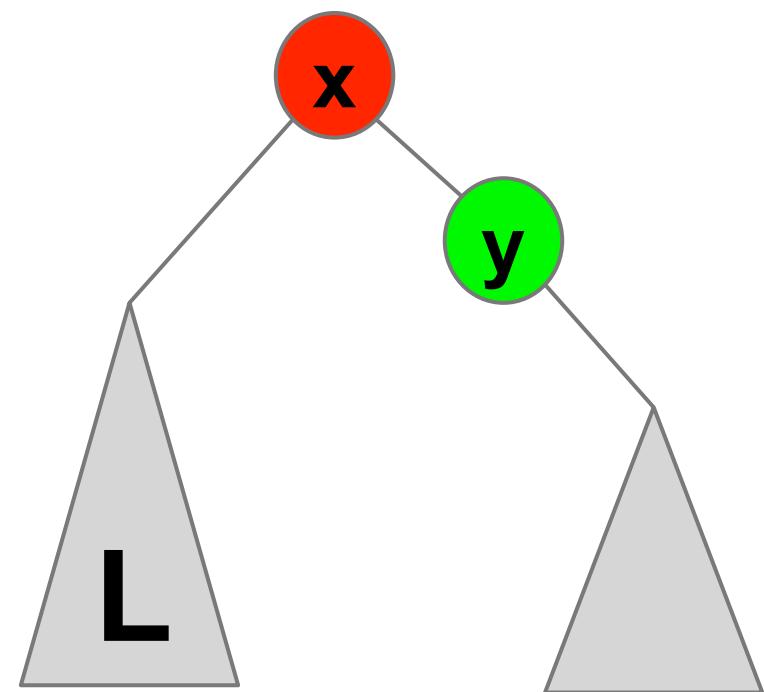


Case 3.2: y is not the right child of x



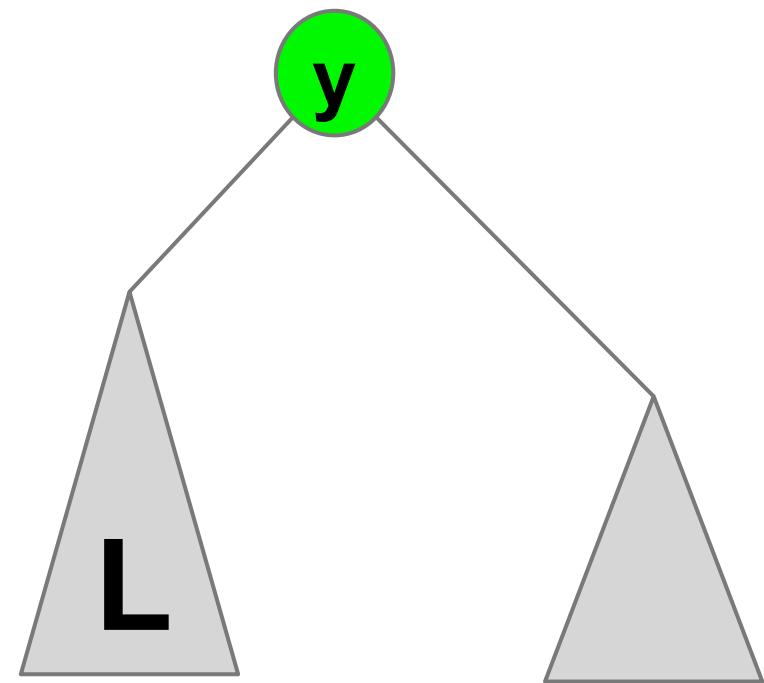
Case 3.1: y is x's right child

Easy, just **promote**
y to x's spot



Case 3.1: y is x's right child

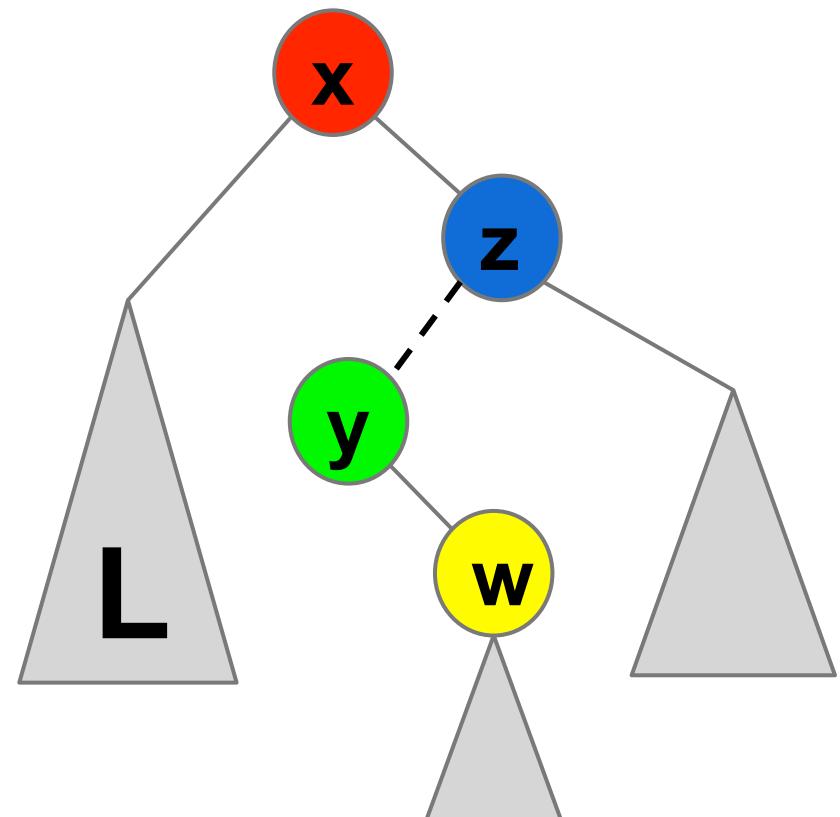
Easy, just **promote**
y to x's spot



Case 3.2: y is NOT x's right child

1. Promote w to y's spot, y becomes free.

Order: $y < w < z$

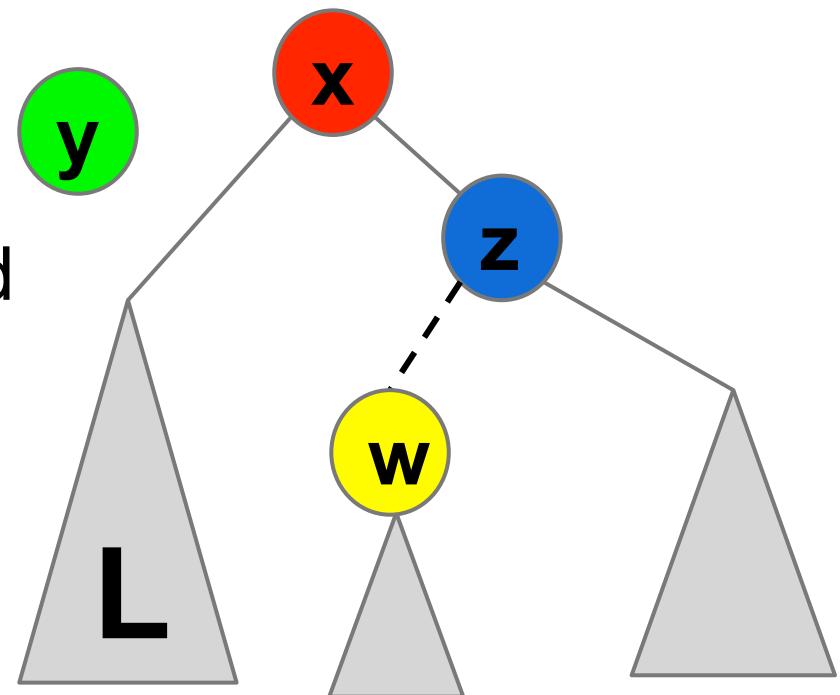


Case 3.2: y is NOT x's right child

1. Promote w to y's spot, y becomes free.

2. Make z be y's right child
(y adopts z)

Order: $y < w < z$



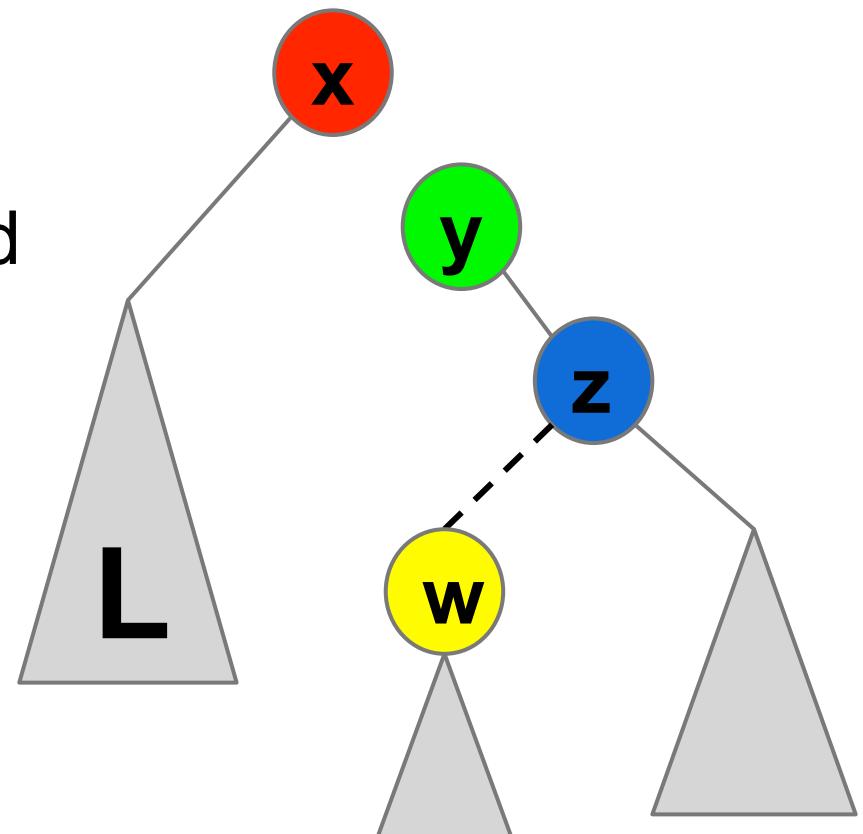
Case 3.2: y is NOT x's right child

1. Promote w to y's spot, y becomes free.

2. Make z be y's right child
(y adopts z)

3. Promote y to x's spot

Order: $y < w < z$



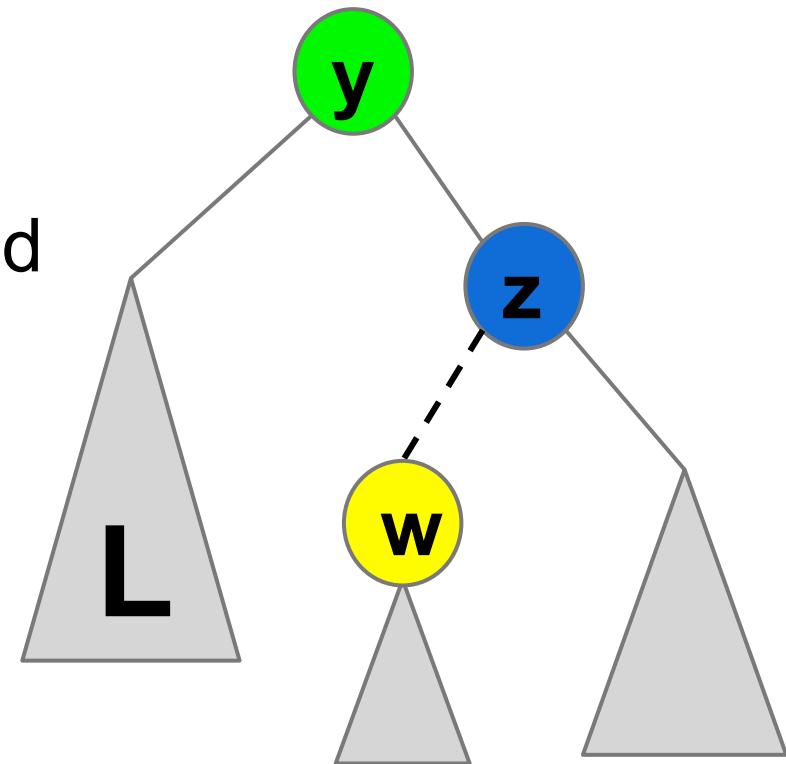
Case 3.2: y is NOT x's right child

1. Promote w to y's spot, y becomes free.

2. Make z be y's right child (y adopts z)

3. Promote y to x's spot

Order: $y < w < z$



x deleted, BST order maintained, all is good.

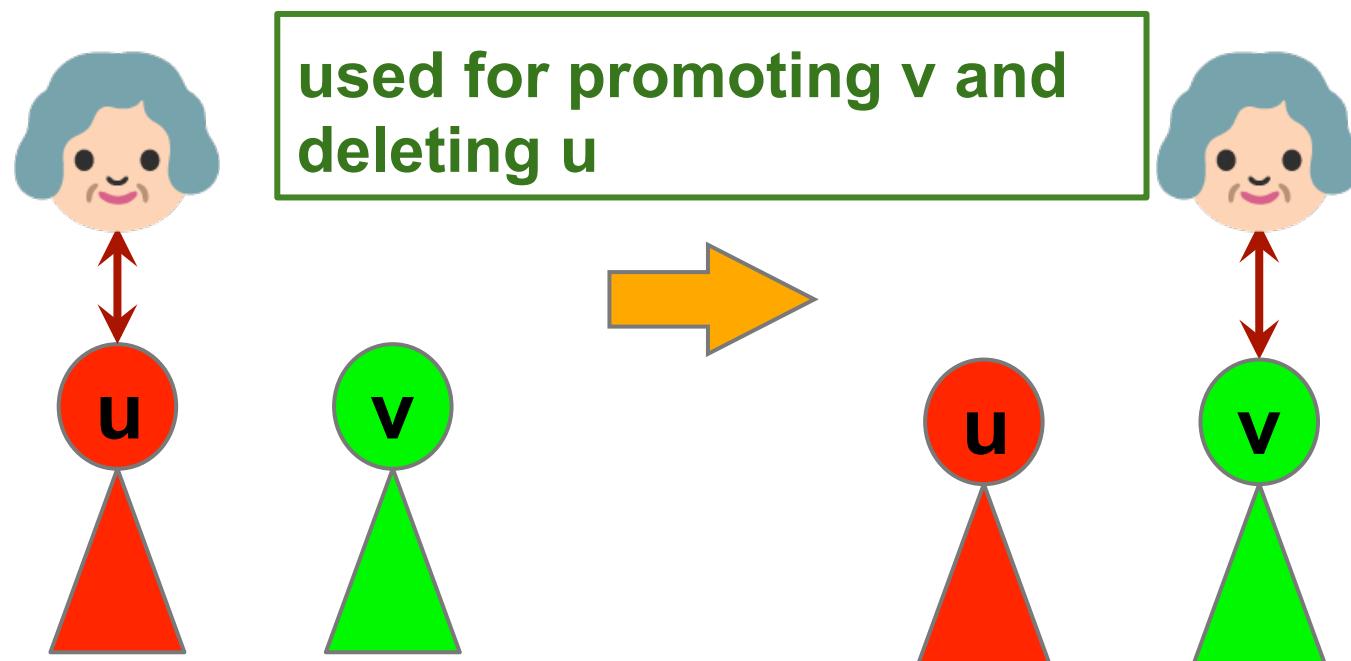
Summarize TreeDelete(root, x)

- Case 1: x has no child, **just delete**
- Case 2: x has one child, **promote**
- Case 3: x has two children, $y = \text{successor}(x)$
 - ◆ Case 3.1: y is x's right child, **promote**
 - ◆ Case 3.2: y is NOT x's right child
 - **promote** y's right child
 - y **adopt** x's right child
 - **promote** y

TreeDelete(root, x): pseudo-code

Textbook Chapter 12.3

Key: Understand **Transplant(root, u, v)**
v takes away u's parent



```
Transplant(root, u, v):
# v takes away u's parent
    if u.p = NIL: # if u is root
        root ← v # v replaces u as root
    elif u = u.p.left:# if u is mom's left child
        u.p.left ← v #mom accepts v as left child
    else: # if u is mom's right child
        u.p.right ← v #mom accept v as right child
    if v ≠ NIL:
        v.p ← u.p # v accepts new mom
                    # u can cry now...
```



```

TreeDelete(root, x):
    if x.left = NIL:
        Transplant(root, x, x.right)
    elif x.right = NIL:
        Transplant(root, x, x.left)
    else:
        y ← TreeMinimum(x.right)
        if y.p ≠ x:
            Transplant(root, y, y.right)
            y.right ← x.right
            y.right.p ← y
        Transplant(root, x, y)
        y.left ← x.left
        y.left.p ← y
    return root

```

Case 1 & 2

Promote right child

Promote left child

get successor(x)

y is not right child of x

Case 3

Case 3.2

promote w

y adopts z

promote y

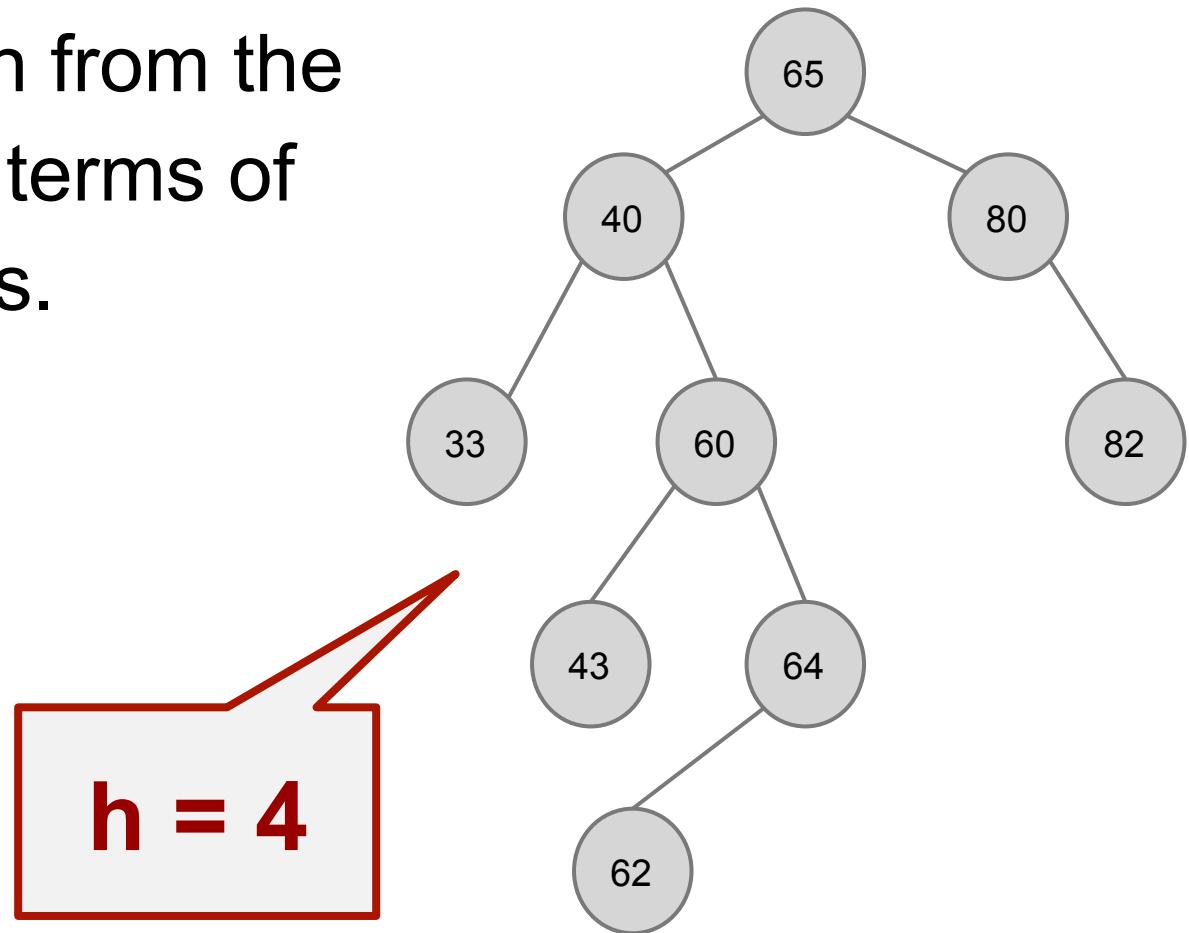
update pointers

TreeDelete(root, x) worst case running time
O(h) (time spent on TreeMinimum)

**Now, about that h
(height of tree)**

Definition: height of a tree

The longest path from the root to a leaf, in terms of number of edges.



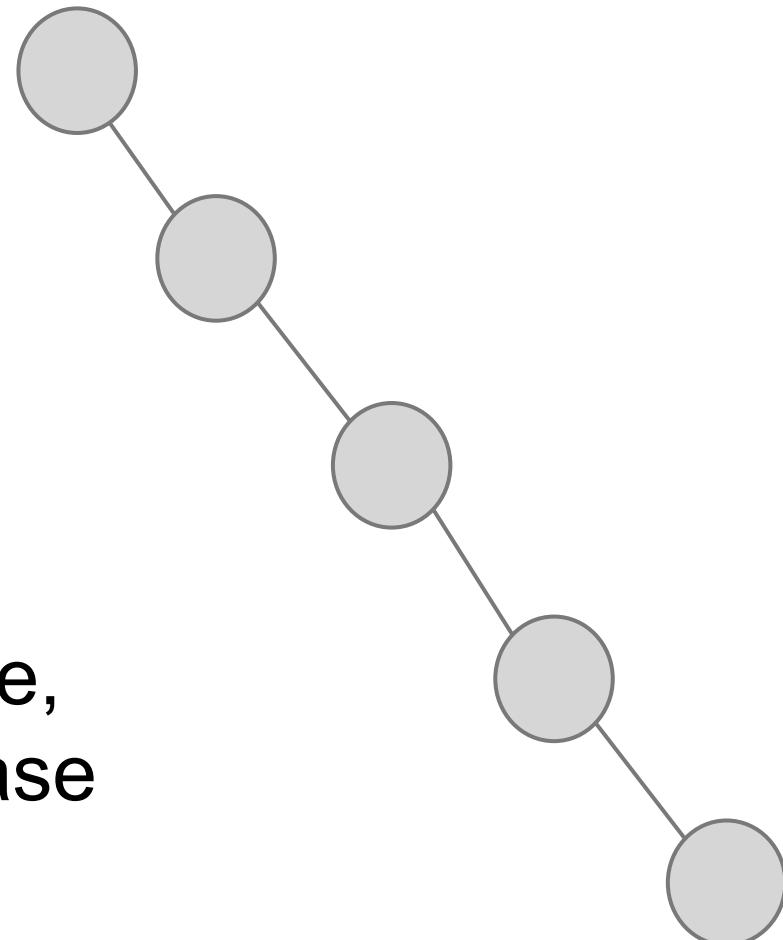
**Consider a BST with n nodes,
what's the highest it can be?**

$$h = n-1$$

i.e, in worst case

$$h \in \Theta(n)$$

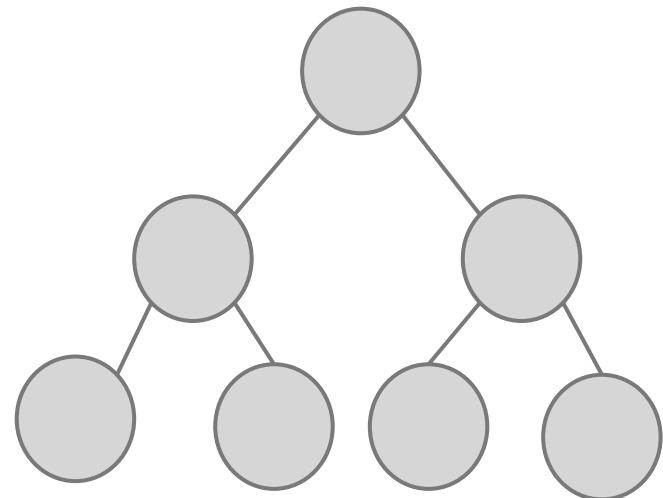
so all the operations we learned with $O(h)$ runtime, they are $O(n)$ in worst case



So, what's the best case for h ?

In best case, $h \in \Theta(\log n)$

A **Balanced BST**
guarantees to have height
 $\in \Theta(\log n)$



Therefore, all the $O(h)$ become $O(\log n)$

Next week

A Balanced BST called an **AVL tree**

CSC263 Week 4

Problem Set 2 is due this Tuesday!

Due Tuesday (Oct 13)



Other Announcements

Ass1 marks available on MarkUS

→ Re-marking requests accepted until October 14

****YOUR MARK MAY GO UP OR DOWN AS THE
RESULT OF A REMARK REQUEST**

Recap

ADT: Dictionary

→ Search, Insert, Delete

Binary Search Tree

→ TreeSearch, TreeInsert, TreeDelete, ...

→ Worst case running time: **O(h)**

→ Worst case height h: **O(n)**

Balanced BST: h is **O(log n)**

Balanced BSTs

AVL tree, Red-Black tree, 2-3 tree, AA tree,
Scapegoat tree, Splay tree, Treap, ...

AVL tree

Invented by **Georgy Adelson-Velsky** and **E. M. Landis** in 1962.

First self-balancing BST to be invented.

We use BFs to check the balance of a tree.

An extra attribute to each node in a BST -- balance factor

$h_R(x)$: height of x's **right** subtree

$h_L(x)$: height of x's **left** subtree

$$BF(x) = h_R(x) - h_L(x)$$

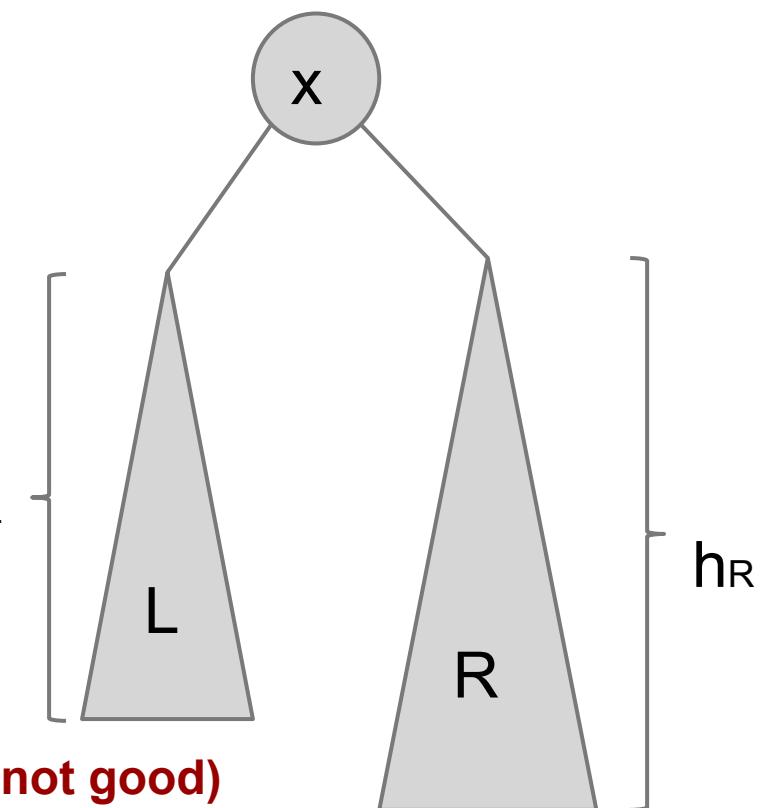
$BF(x) = 0$: x is **balanced**

$BF(x) = 1$: x is **right-heavy**

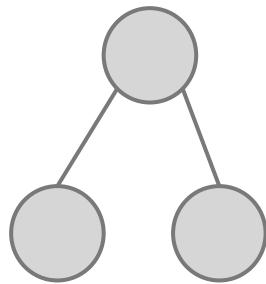
$BF(x) = -1$: x is **left-heavy**

above 3 cases are considered as "good" h_L

$BF(x) > 1$ or < -1 : x is **imbalanced (not good)**



heights of some special trees



$h = 1$



$h = 0$

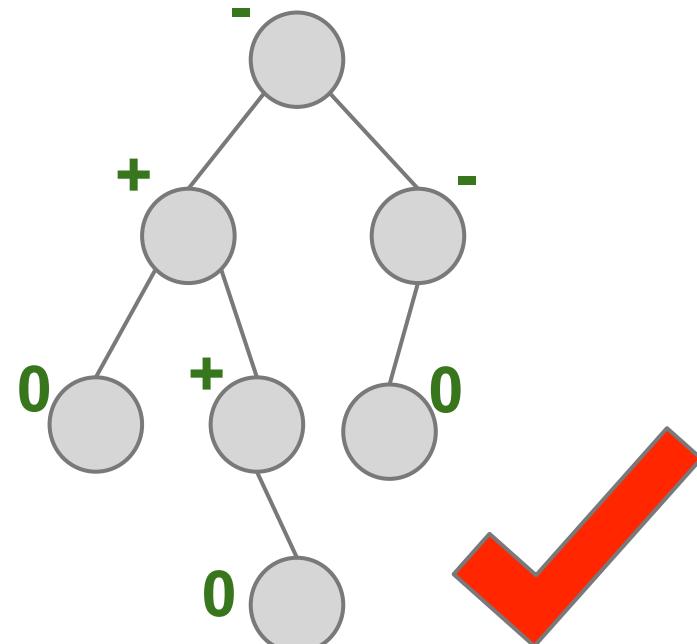
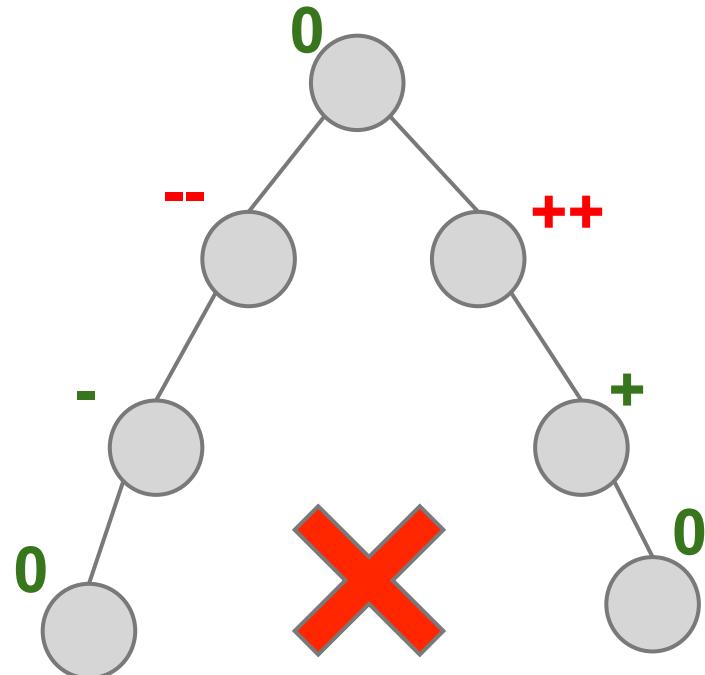
NIL

$h = -1$

Note: height is measured by the number of edges.

AVL tree: definition

An AVL tree is a BST in which **every** node is balanced, right-heavy or left-heavy.
i.e., the **BF** of **every** node must be 0, 1 or -1.



It can be **proven** that the height of an AVL tree with n nodes satisfies

$$h \leq 1.44 \log_2(n + 2)$$

i.e., h is in **$O(\log n)$**

WHY?

Operations on AVL trees

AVL-Search(root, k)

AVL-Insert(root, x)

AVL-Delete(root, x)

Things to worry about

- Before the operation, the BST is a **valid AVL tree** (precondition)
- After the operation, the BST must **still be a valid AVL tree** (so re-balancing may be needed)
- The **balance factor** attributes of some nodes need to be **updated**.

AVL-Search(root, k)

Search for key k in the AVL tree rooted at root

First, do a **TreeSearch(root, k)** as in **BST**.

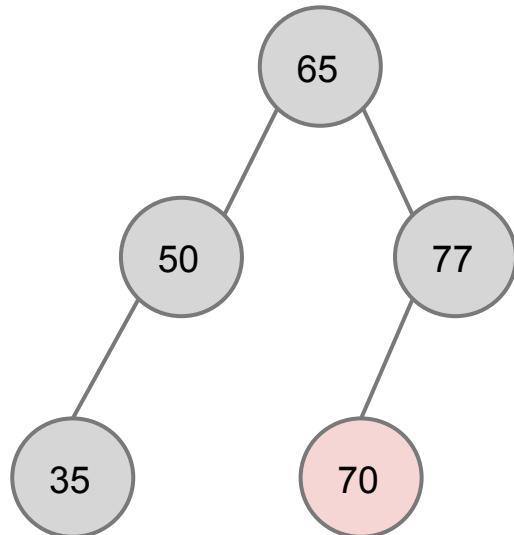
Then, nothing else!

(No worry about balance being broken because
we didn't change the tree)

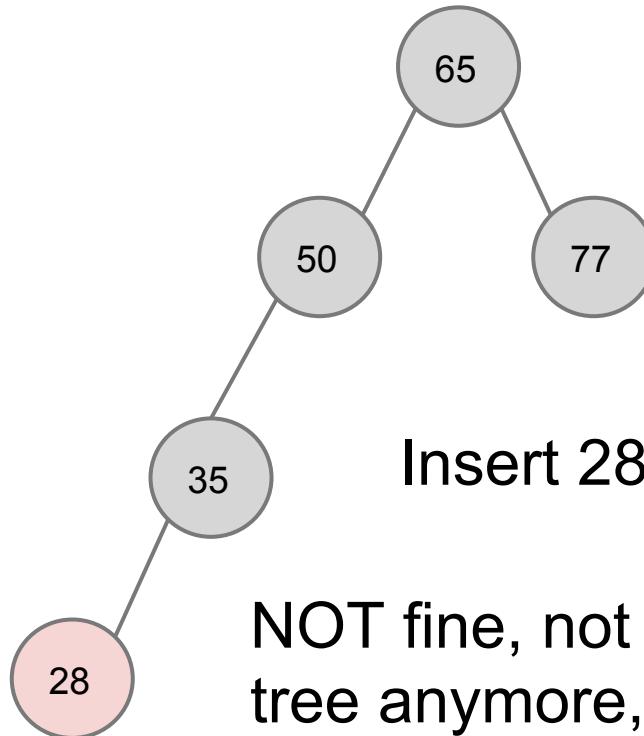


AVL-Insert(root, x)

First, do a **TreeInsert(root, x)** as in **BST**



Insert 70
everything is fine



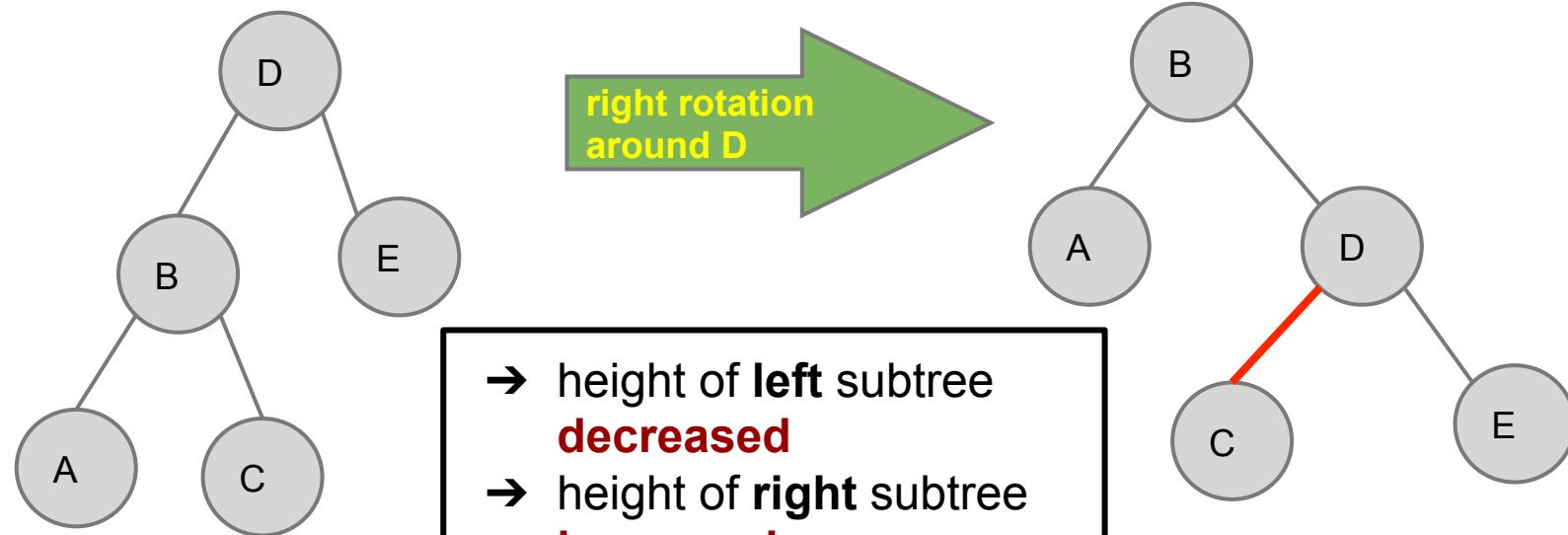
Insert 28
NOT fine, not an AVL
tree anymore,
need **rebalancing**.

Basic move for rebalancing -- Rotation

Objective:

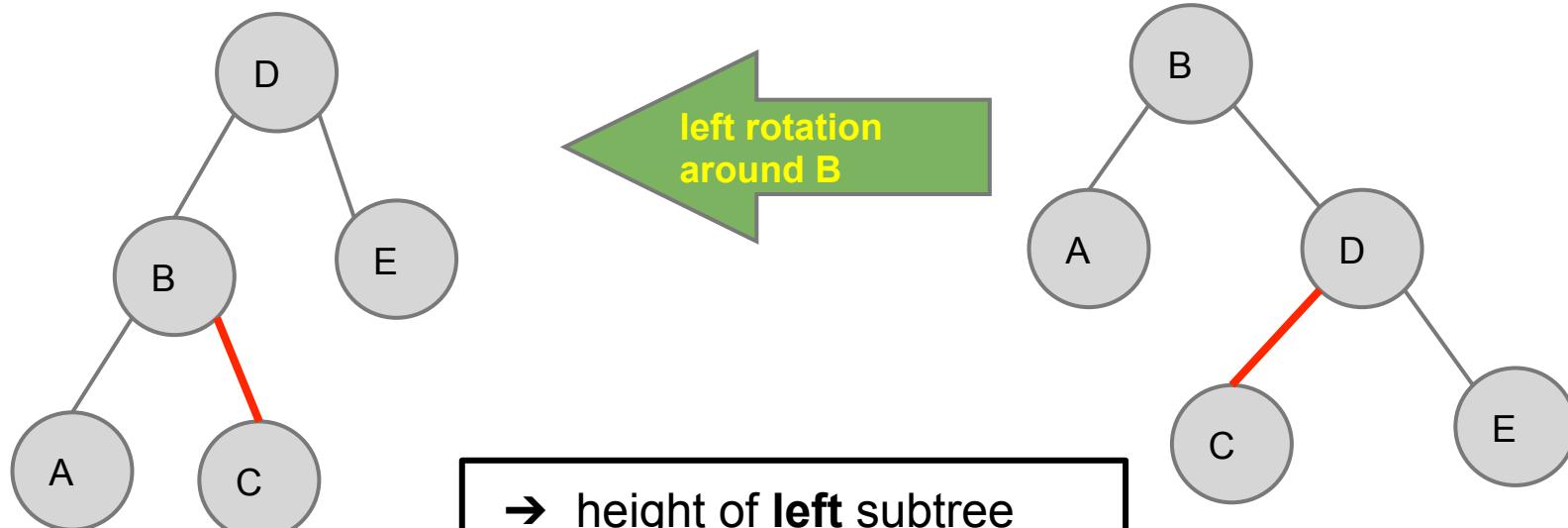
1. change heights of a node's left and right subtrees
2. maintain the BST property

BST order to be maintained: **ABCDE**



Similarly, left rotation

BST order to be maintained: **ABCDE**



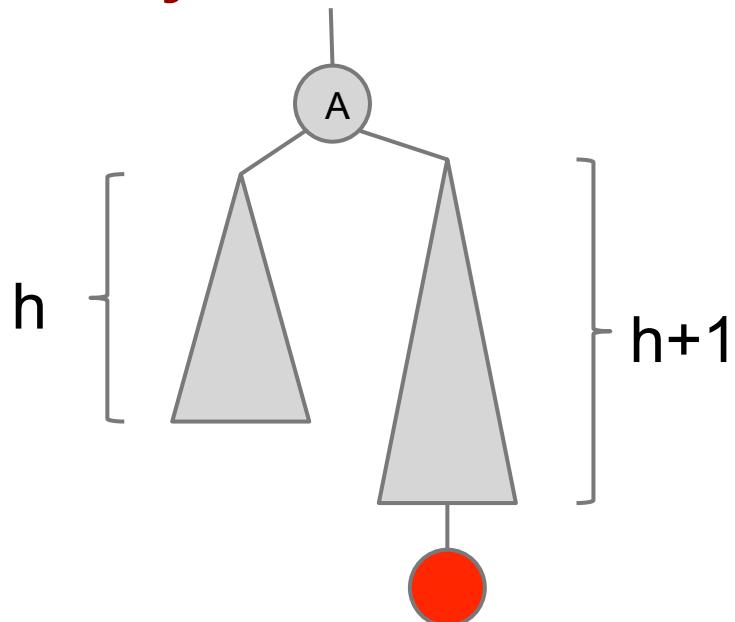
- height of **left** subtree **increased**
- height of **right** subtree **decreased**
- BST order **maintained**

Now, we are ready to use rotations to rebalance an AVL tree after insertion

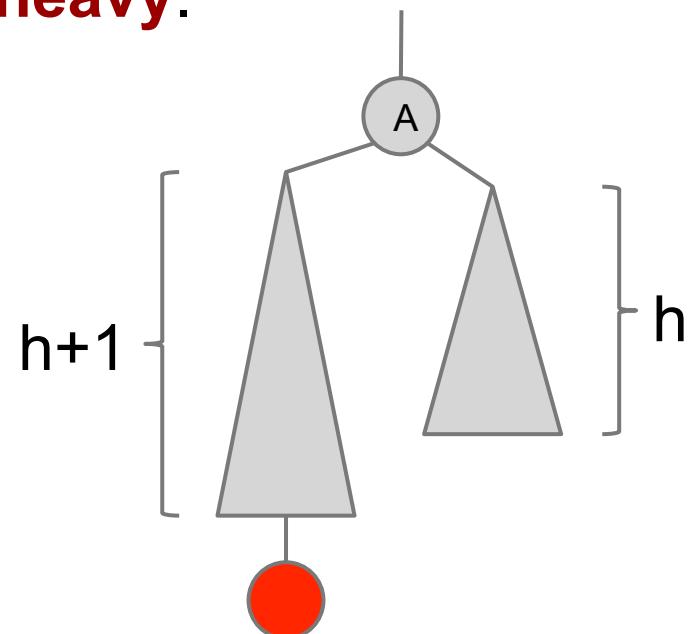
A is the lowest ancestor of the new node who became imbalanced.

When do we need to rebalance?

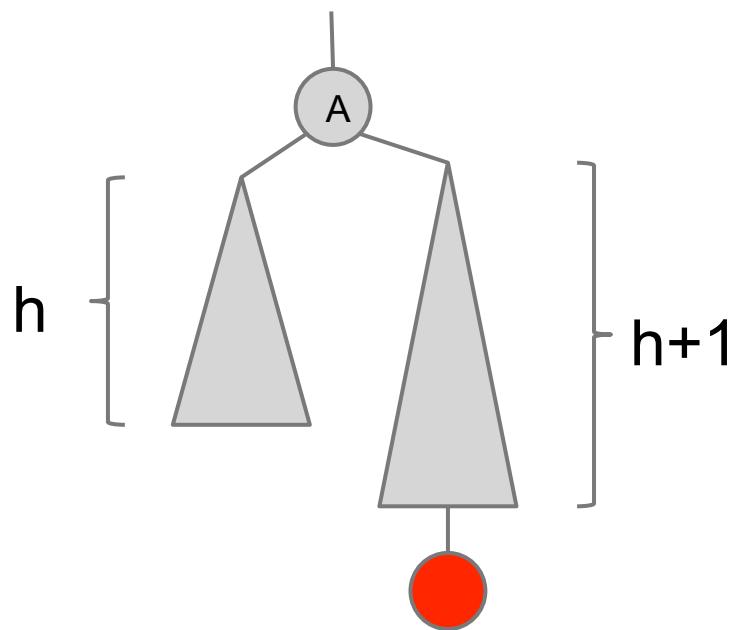
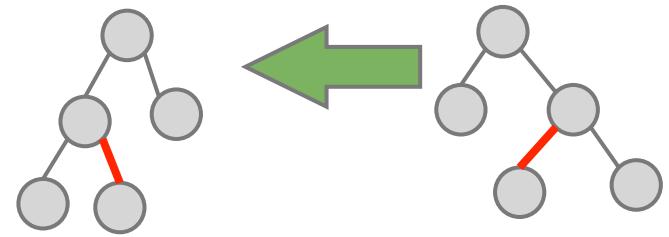
Case 1: the insertion increases the height of a node's **right subtree**, and that node was already **right heavy**.



Case 2: the insertion increases the height of a node's **left subtree**, and that node was already **left heavy**.



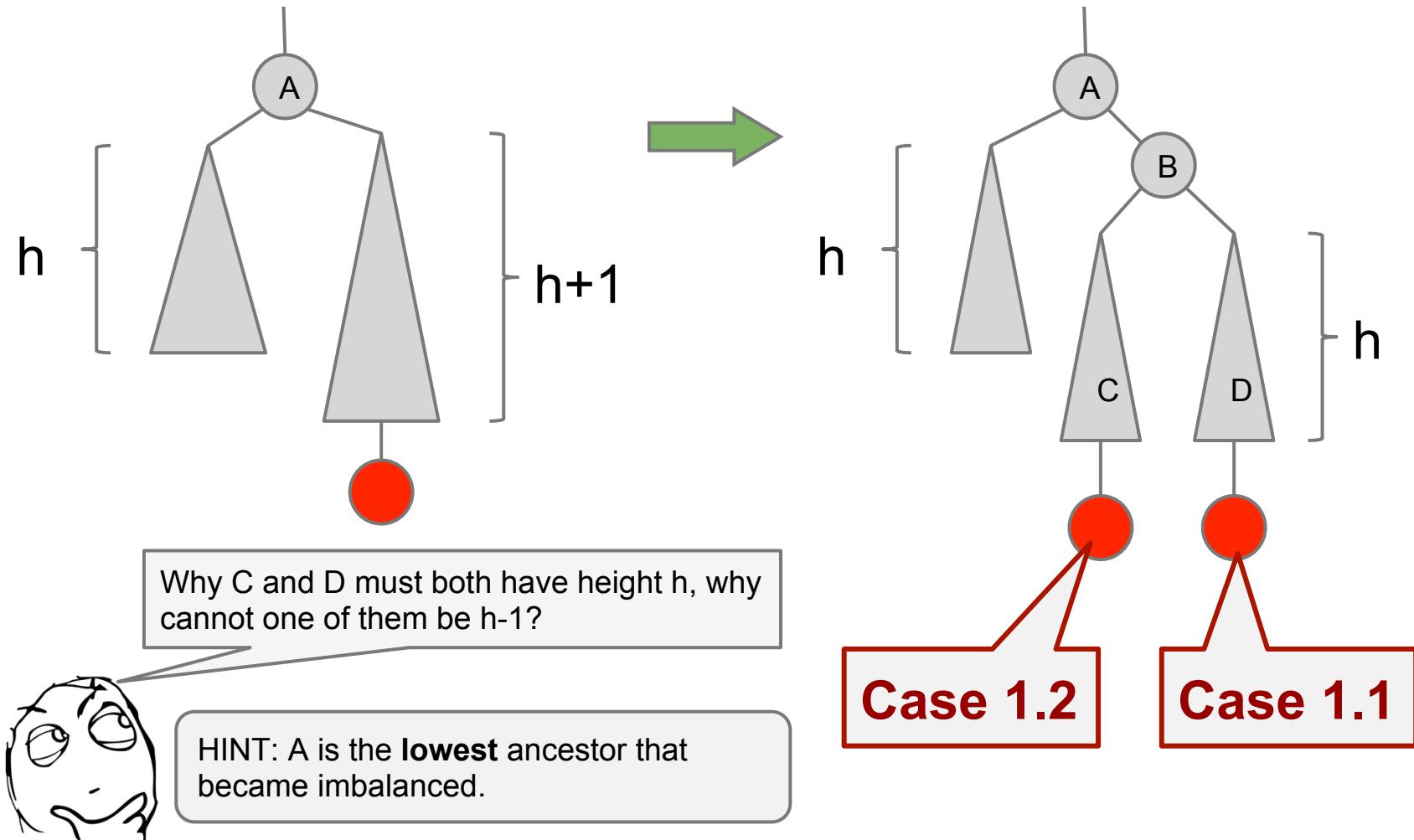
Let's deal with Case 1



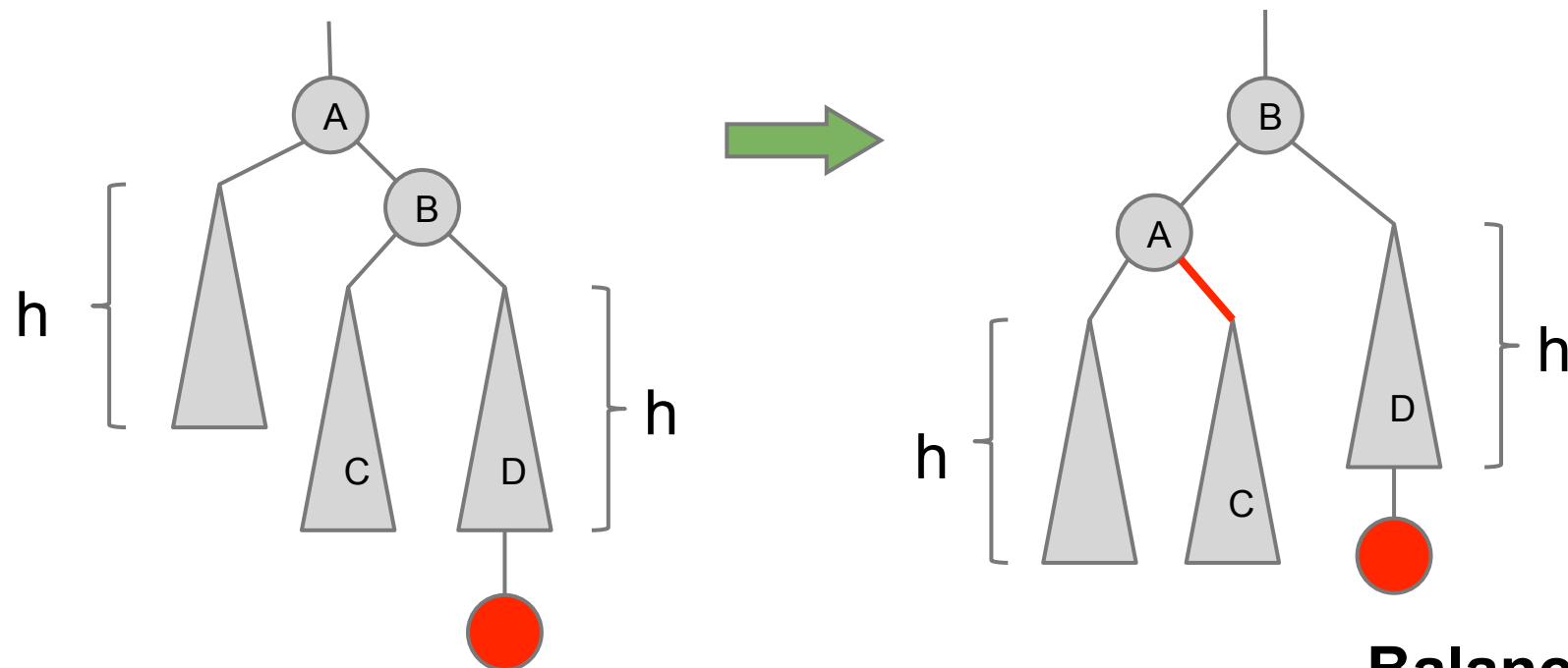
In order to rebalance, we need to **increase** the height of the **left** subtree and **decrease** the height of the **right** subtree, so....

We want to do a **left rotation** around A, but in order to do that, we need a more refined picture.

Case 1, more refined picture



Case 1.1, let's left-rotate around A!

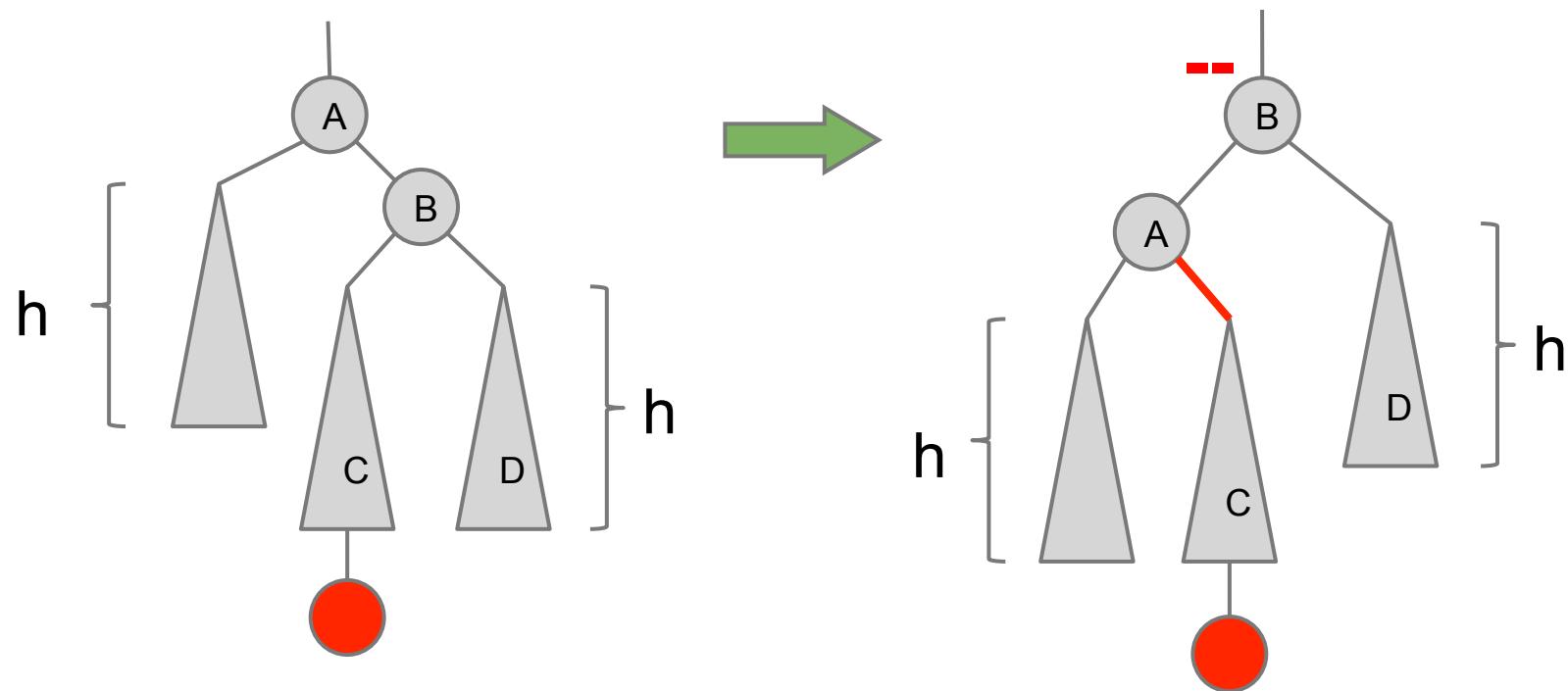


Balanced!



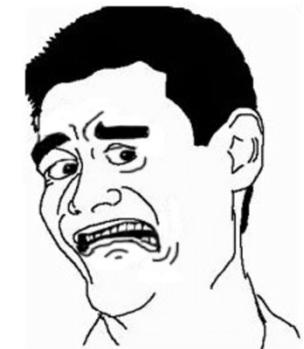
Another important thing to note:
After the rotation, the **height** of the whole subtree in the picture **does not change ($h+2$) before and after** the insertion , i.e., everything happens in this picture stays in this picture, nobody above would notice.

Case 1.2, let's left-rotate around A!

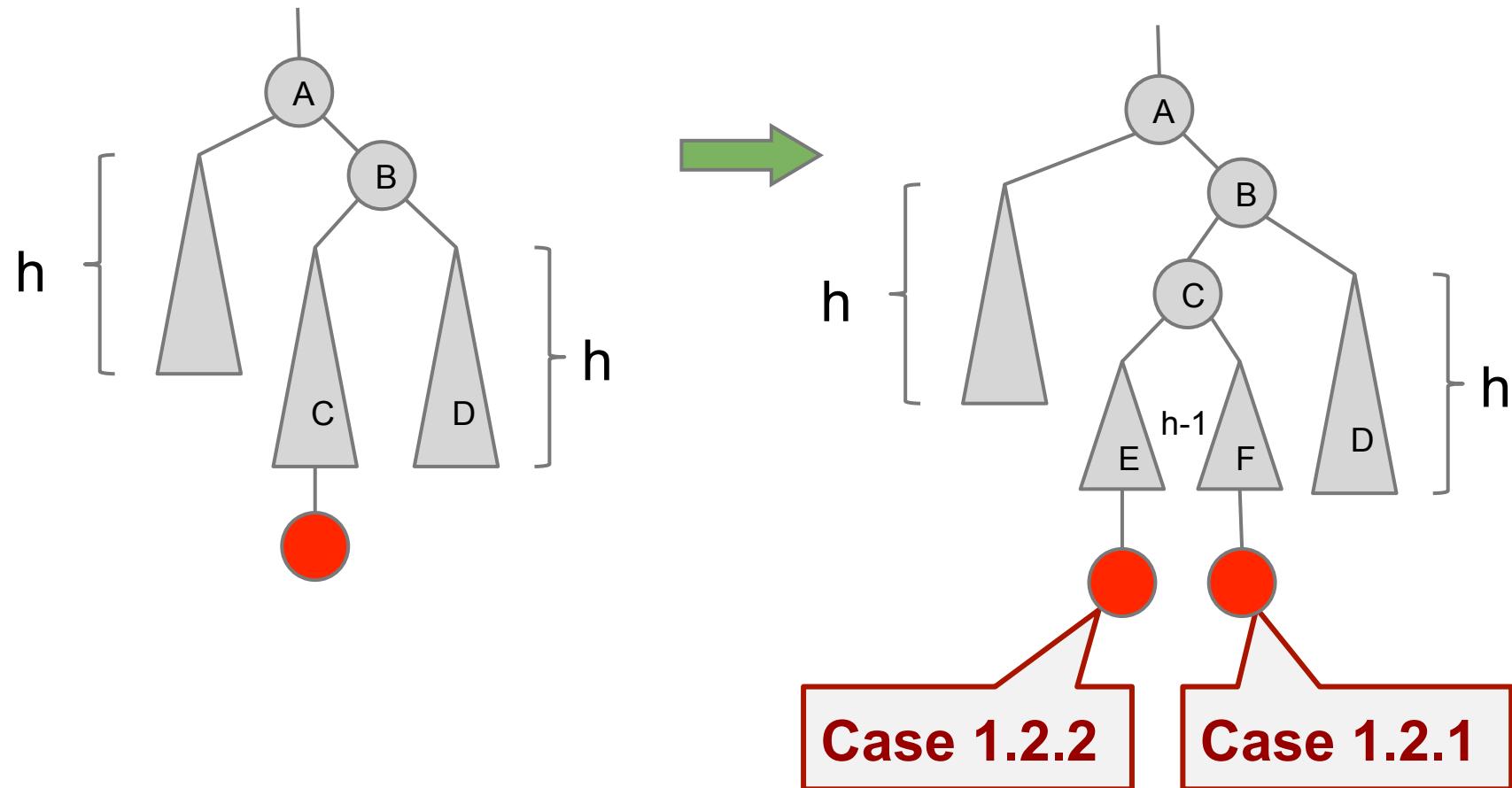


Still not balanced.

To deal with this, we need an even more refined picture.

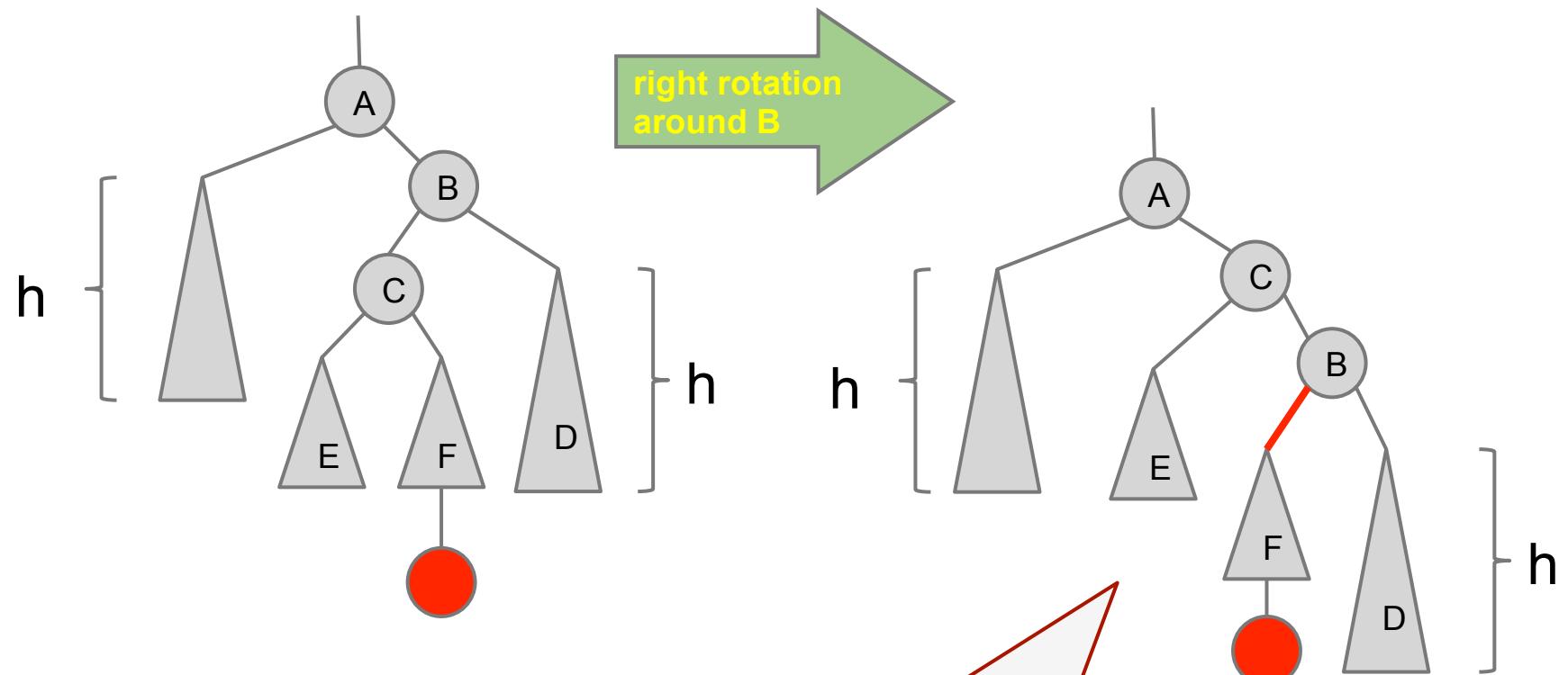


Case 1.2, an even more refined picture



These two cases are actually not that different.

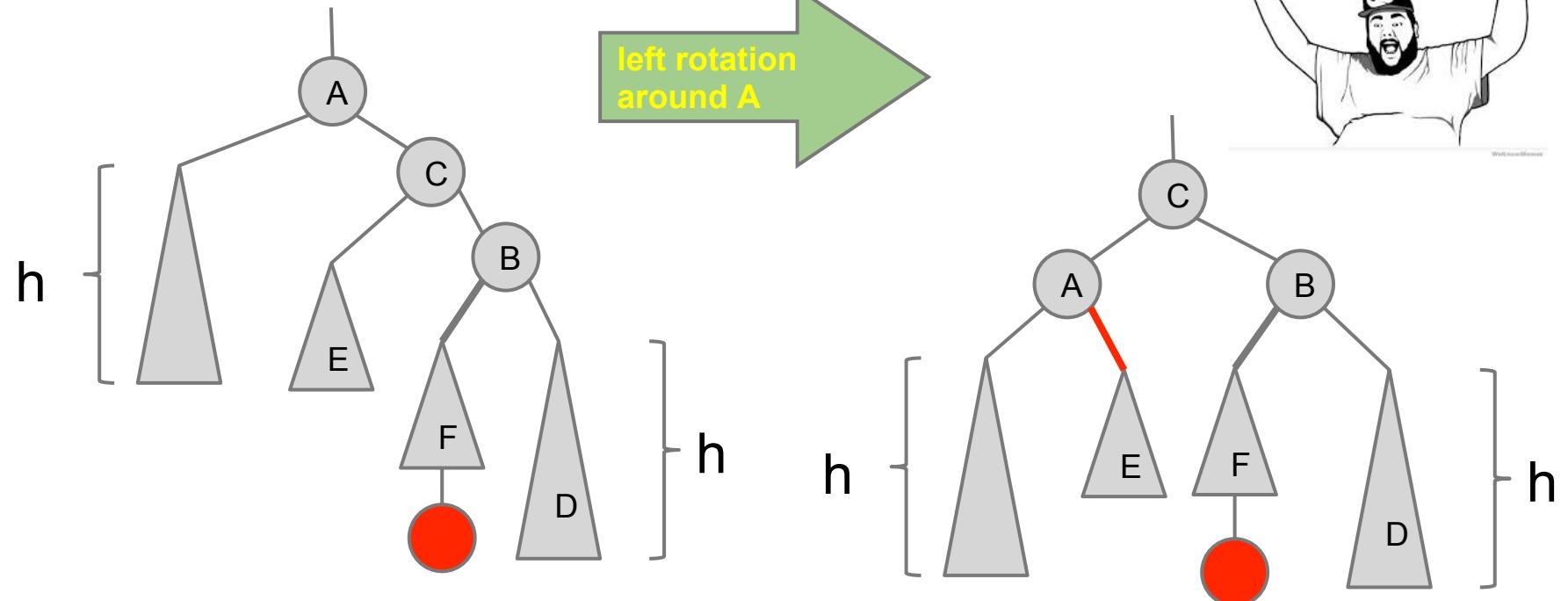
Case 1.2.1, ready to rotate



Now the right side looks “**heavy**” enough for a **left rotation around A**.

Case 1.2.1, second rotation

Balanced!

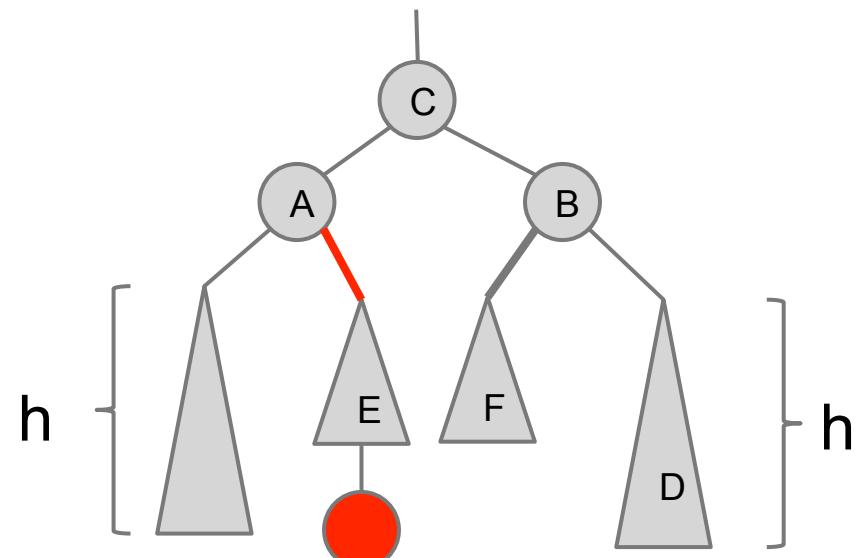


Same note as before: After the rotations, the **height** of the whole subtree in the picture **does not change ($h+2$) before and after** the insertion , i.e., everything happens in this picture stays in this picture, nobody above would notice.

What did we just do for Case 1.2.1?

We did a **double rotation, first a right and then a left rotation.**

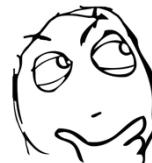
For **Case 1.2.2**, we do exactly the same thing, and get this...



Practice for home

AVL-Insert -- outline

- First, insert like a BST
- If still balanced, return.
- Else: (need re-balancing)
 - ◆ Case 1:
 - Case 1.1: single left rotation
 - Case 1.2: double right-left rotation
 - ◆ Case 2: (symmetric to Case 1)
 - Case 2.1: single right rotation
 - Case 2.2: double left-right rotation



Something missing?

Things to worry about

- Before the operation, the BST is a **valid AVL tree** (precondition)
- After the operation, the BST must **still be a valid AVL tree**
- The **balance factor** attributes of some nodes need to be **updated**.

Updating balance factors

Just update accordingly as rotations happen.

And nobody outside the picture needs to be updated, because the height is the same as before and nobody above would notice a difference.

“Everything happens in Vegas stays in Vegas”.

So, only need $O(1)$ time for updating BFs.

Note: this nice property is only for Insert. Delete will be different.

Running time of AVL-Insert

Just Tree-Insert plus some constant time for rotations and BF updating.

Overall, worst case $O(h)$

since it's balanced, **$O(\log n)$**

Recap

- AVL tree: a self-balancing BST
 - ◆ each node keeps a balance factor

- Operations on AVL tree
 - ◆ AVL-Search: same as BST
 - ◆ AVL-Insert:
 - First do a BST TreeInsert
 - Then rebalance if necessary
 - Single rotations, double rotations.
 - ◆ AVL-Delete



AVL-Delete(root, x)

Delete node x from the AVL tree rooted at root

AVL-Delete: General idea

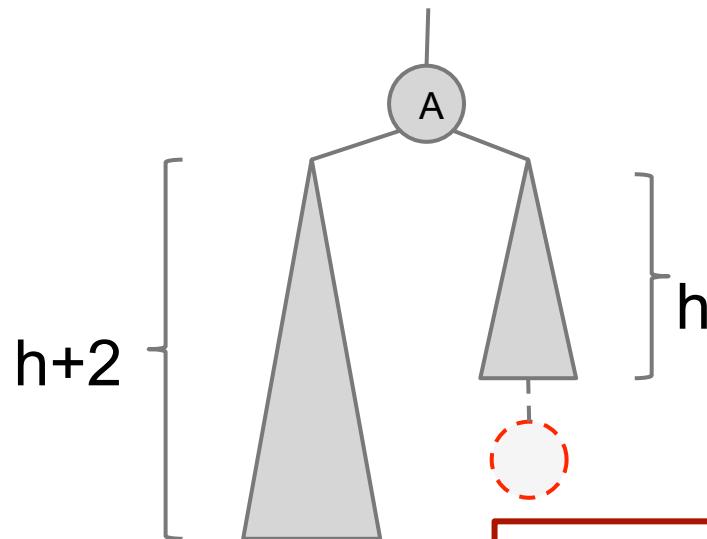
- First do a normal BST **Tree-Delete**
- The deletion may cause changes of subtree heights, and may cause certain nodes to **lose AVL-ness** ($BF(x)$ is 0, 1 or -1)
- Then **rebalance** by single or double **rotations**, similar to what we did for AVL-Insert.
- Then **update BFs** of affected nodes.

Note : node A is the **lowest ancestor** that becomes imbalanced.

Note 2: height of the “whole subtree” rooted at A before deletion is **$h + 3$**

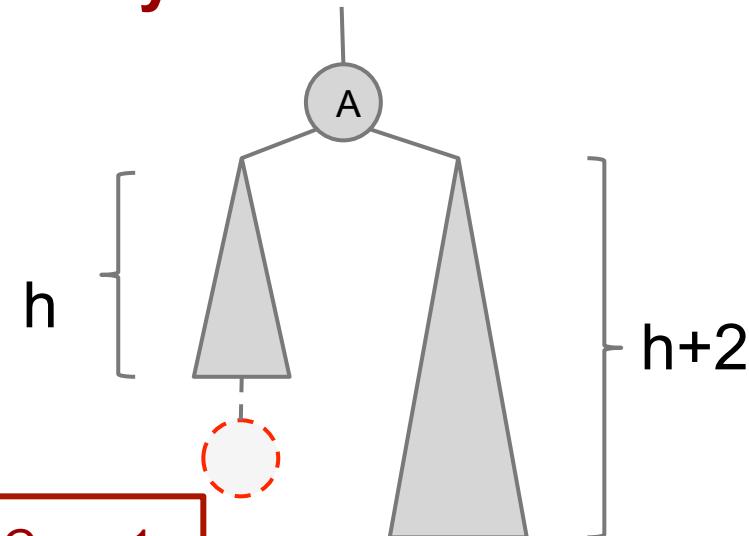
Cases that need rebalancing.

Case 1: the deletion **reduces the height** of a node’s **right subtree**, and that node was **left heavy**.



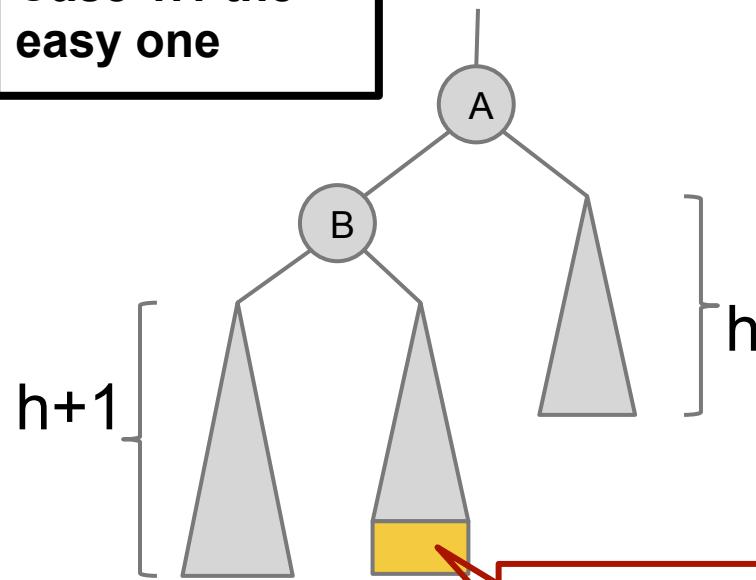
Just need to handle Case 1,
Case 2 is symmetric.

Case 2: the insertion **increases the height** of a node’s **left subtree**, and that node was already **left heavy**.



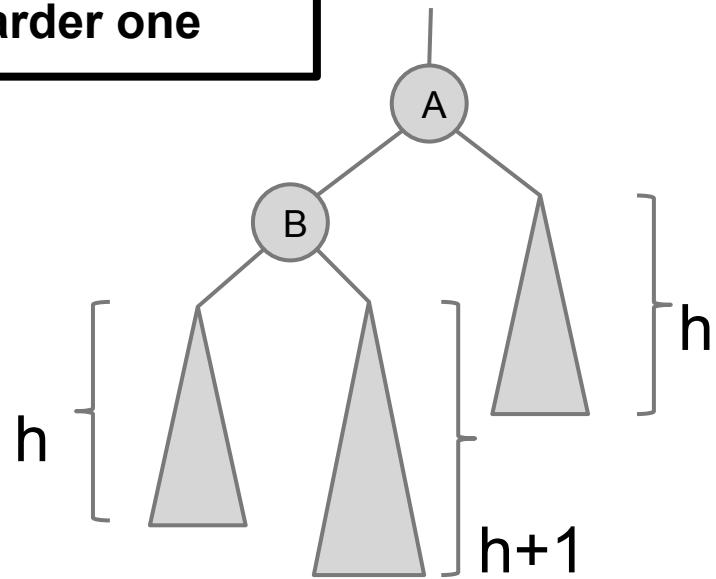
Case 1.1 and Case 1.2 in a refined picture

Case 1.1 the
easy one



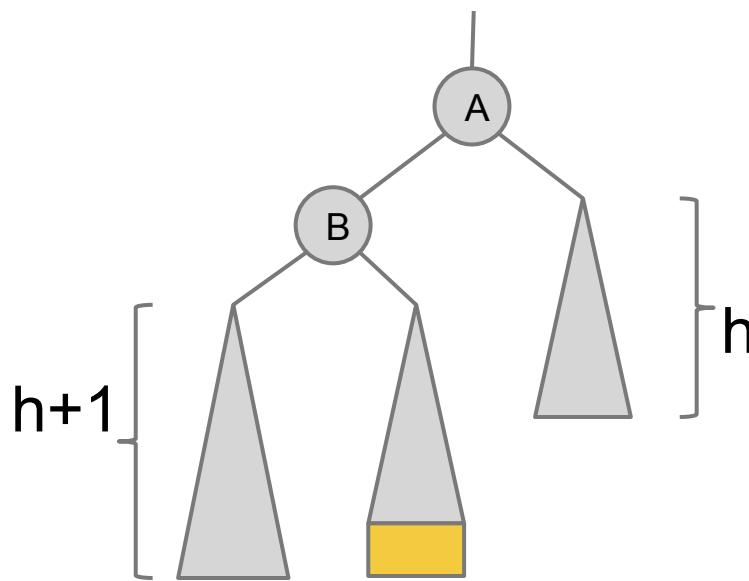
A single right rotation
around A would fix it

Case 1.2 the
harder one

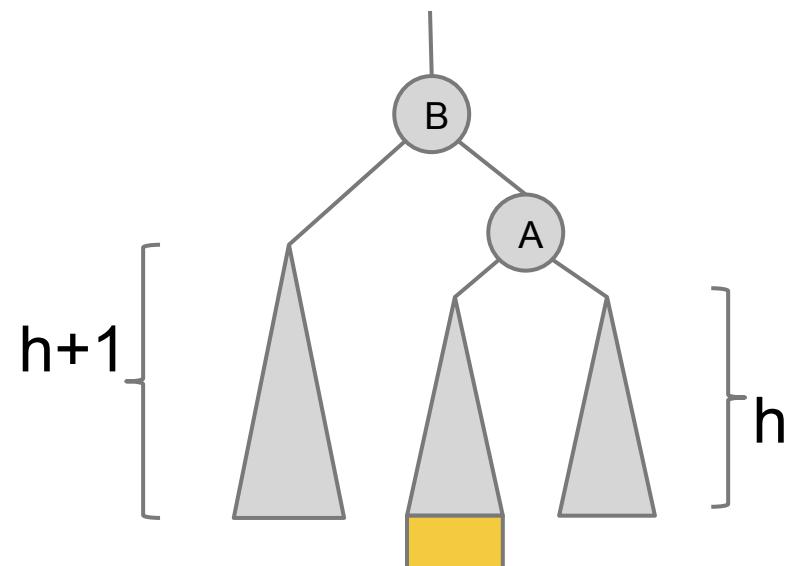


Need double left-right
rotations

Case 1.1: single right rotation



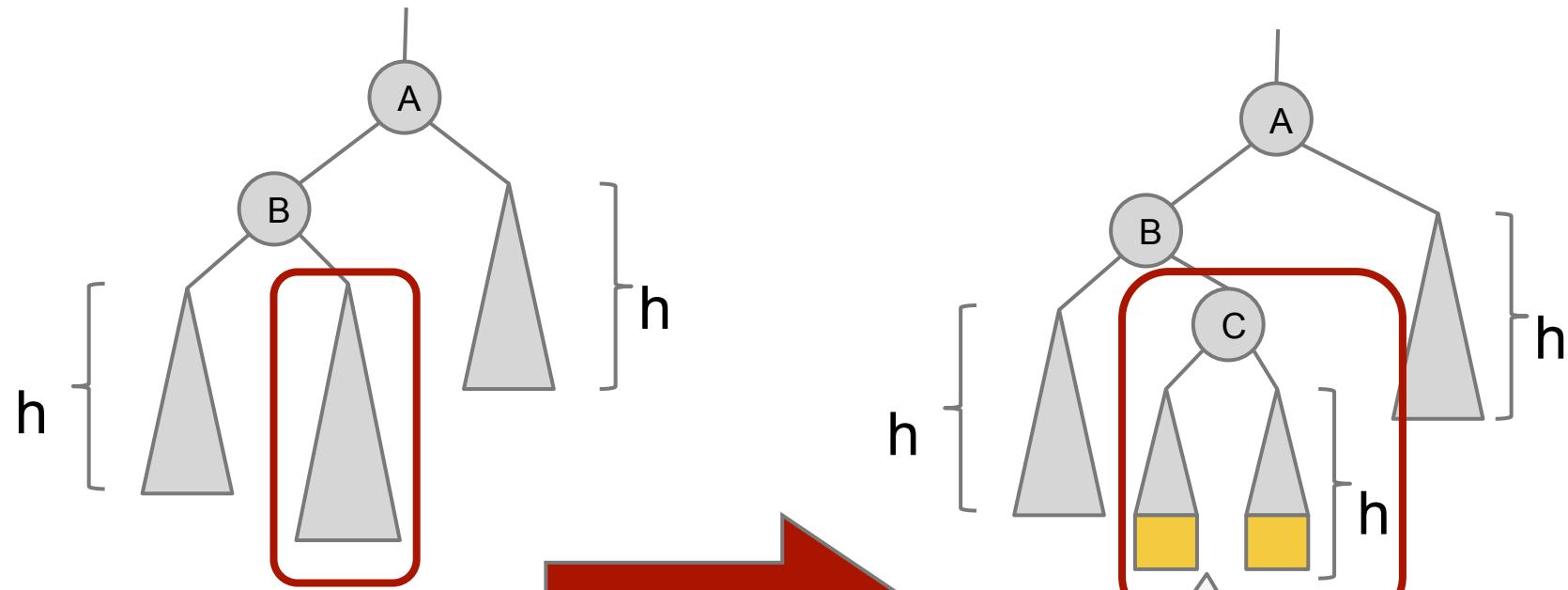
right rotation
around A



Note: after deletion, the height of the whole subtree could be $h+3$ (same as before) or $h+2$ (**different** from before) depending on whether the **yellow** box exists or not.

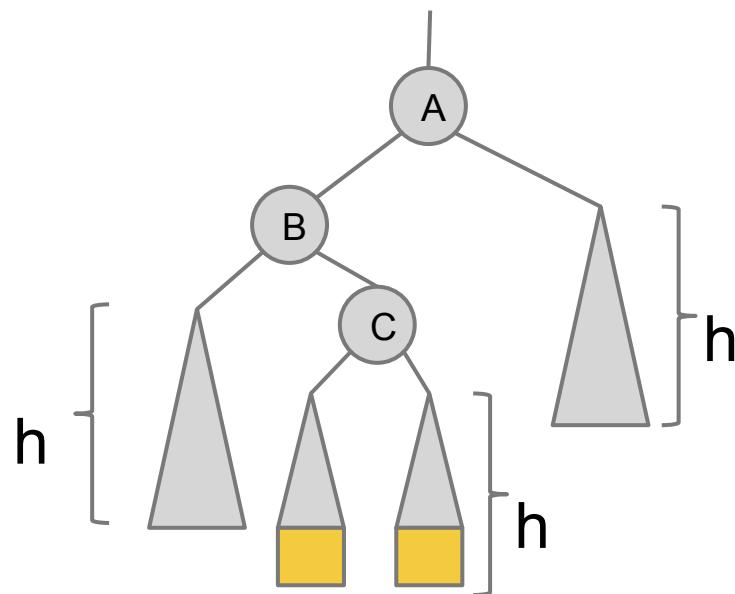
Balanced!

Case 2: first refine the picture

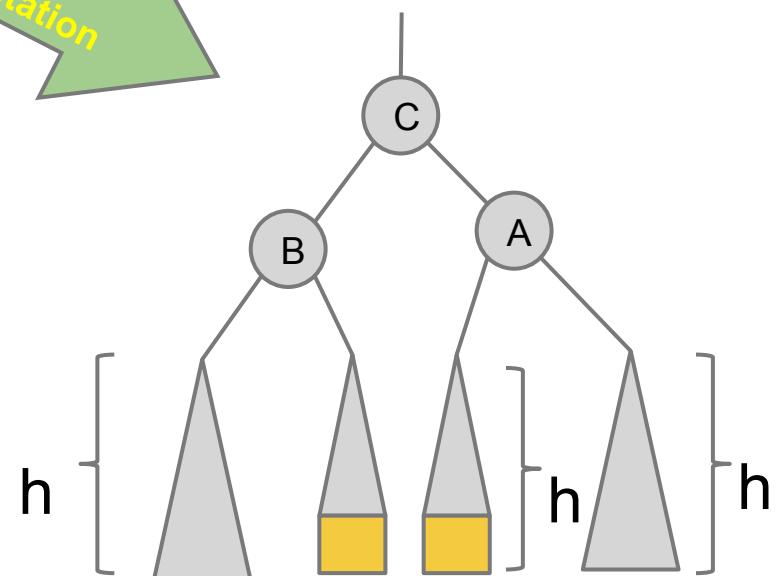


Only one of the two yellow boxes needs to exist.

Case 2: double left-right rotation



double left
right rotation



Note: In this case, the height of the whole subtree after deletion must be $h+2$ (guaranteed to be **different** from before).

No Vegas any more!

Beautifully balanced!

Updating the balance factors

Since the **height** of the “whole subtree” may change, then the **BFs** of **some nodes** outside the “whole subtree” need to be updated.

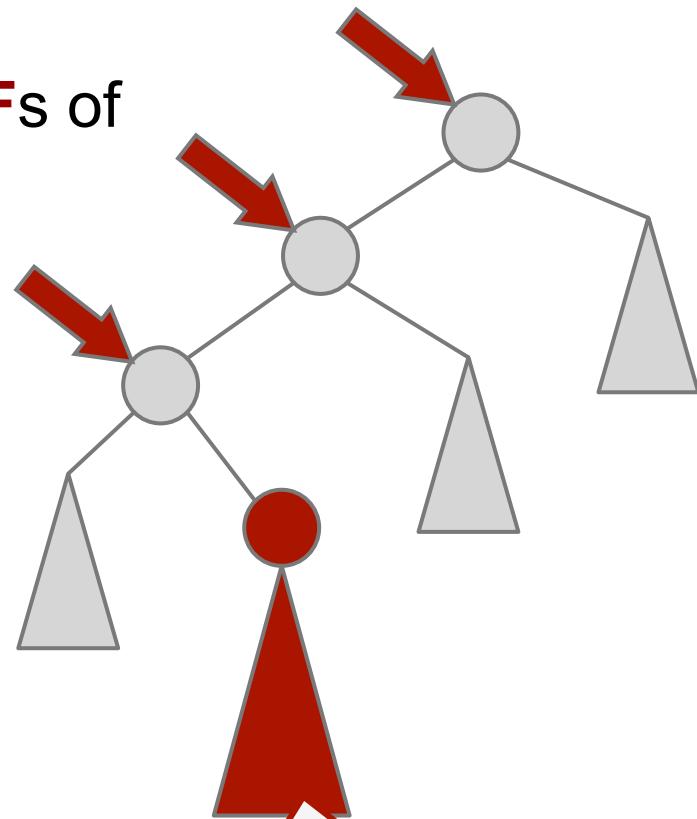
Which nodes?

All **ancestors** of the subtree

How many of them?

$O(\log n)$

**Updating BFs take
 $O(\log n)$ time.**



The “whole subtree”,
where things happened.

For home thinking



In an AVL tree, each node does NOT really store the ***height*** attribute. They only store the ***balance factor***.

But a node can always **infer** the change of height from the change of BF of its child.

For example, “After an insertion, my left child’s BF changed from 0 to +1, then my left subtree’s height must have increased by 1. I gotta update my BF...”

Think it through by enumerating all possible cases.

Alternative implementation of AVL tree

Instead of storing the balance factor at each node x , we can also store the height of the subtree rooted at x . All information about the balance factor can be calculated from the height information.

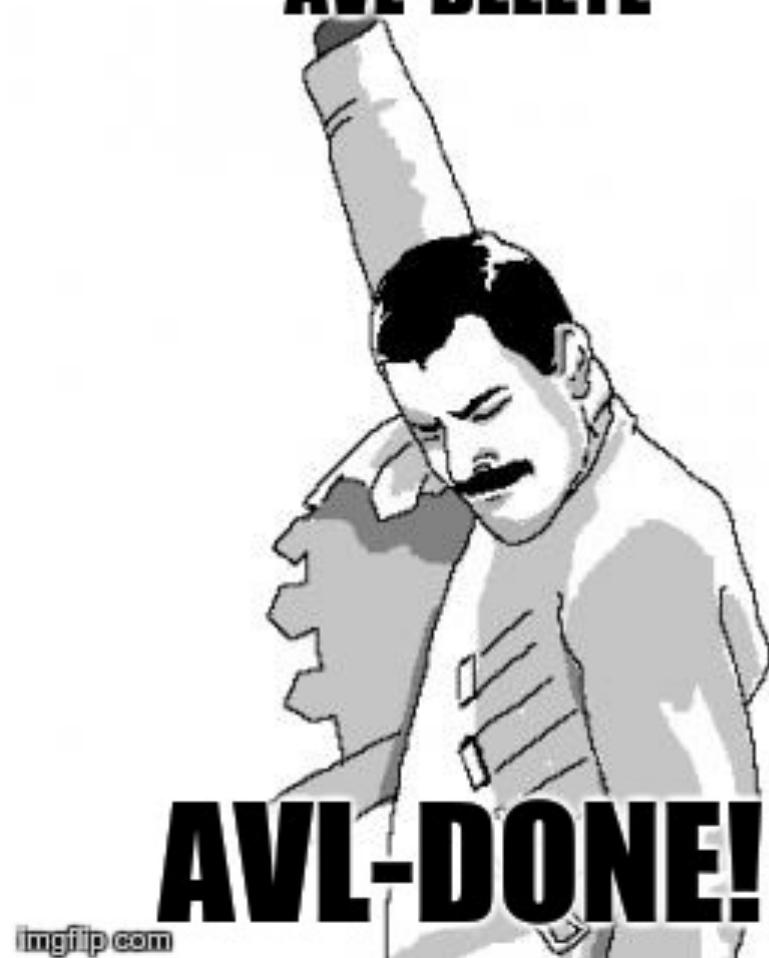
AVL-Deletion: Outline

- First, Delete like a BST
- If still balanced, return.
- Else: (need re-balancing)
 - ◆ Case 1:
 - Case 1.1: single right rotation
 - Case 1.2: double left-right rotation
 - ◆ Case 2: (symmetric to Case 1)
 - Case 2.1: single left rotation
 - Case 2.2: double right-left rotation
 - ◆ Update balance factor as rotation happens, and propagates up to root.

AVL-Delete: Running time

- BST Tree-Delete: $O(\log n)$
- Update balance factors: $O(\log n)$
- Rotations: $O(\log n)$ (*not $O(1)$ because more rotations at higher level may be caused as a result of updating ancestors' balance factors*)
- Overall: $O(\log n)$ worst-case

**AVL-SEARCH, AVL-INSERT,
AVL-DELETE**



AVL-DONE!

Augmenting Data Structures

This is not about a particular dish,
this is about **how to cook.**

Reflect on AVL tree

- We “**augmented**” BST by storing **additional information** (the balance factor) at each node.
- The additional information enabled us to do **additional cool things** with the BST (keep the tree balanced).
- And we can **maintain** this additional information **efficiently** in modifying operations (within $O(\log n)$ time, without affecting the running time of Insert or Delete).

Augmentation is an important methodology for data structure and algorithm design.

It's widely used in practice, because

- On one hand, textbook data structures rarely satisfy what's needed for solving real interesting problems.
- On the other hand, people also rarely need to invent something completely new.
- Augmenting known data structures to serve specific needs is the sensible middle-ground.

Augmentation: General Procedure

1. Choose data structure to augment
2. Determine **additional information**
3. Check **additional information** can be maintained, during each original operation, hopefully efficiently.
4. Implement new operations.

Example: Ordered Set

An ADT with the following operations

→ Search(S, k) in $O(\log n)$

→ Insert(S, x) in $O(\log n)$

→ Delete(S, x) in $O(\log n)$

→ Rank(k): return the rank of key k

→ Select(r): return the key with rank r



AVL tree
would work



E.g., $S = \{ 27, 56, 30, 3, 15 \}$

$\text{Rank}(15) = 2$ because 15 is the second smallest key

$\text{Select}(4) = 30$ because 30 is the 4th smallest key

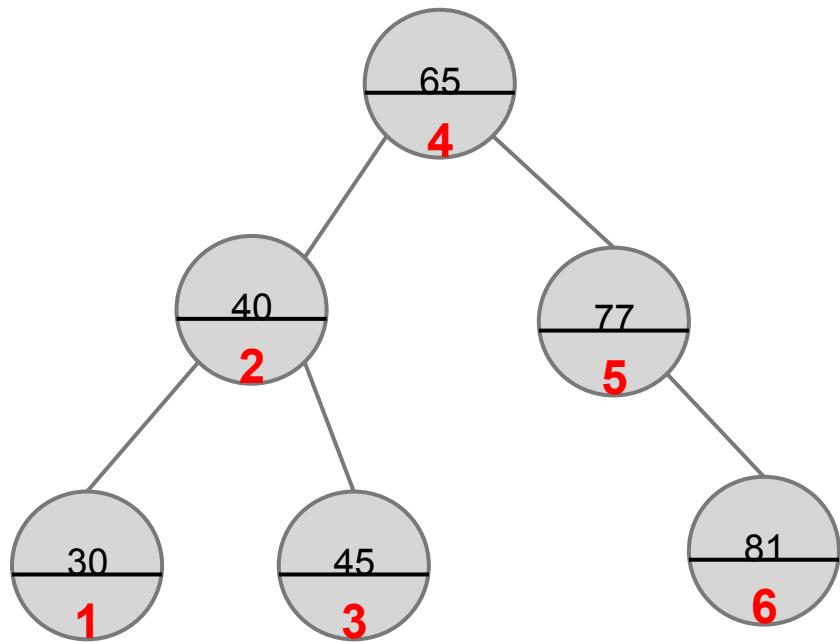
Augmentation
needed

Ideas will be explored in this week's **tutorial**

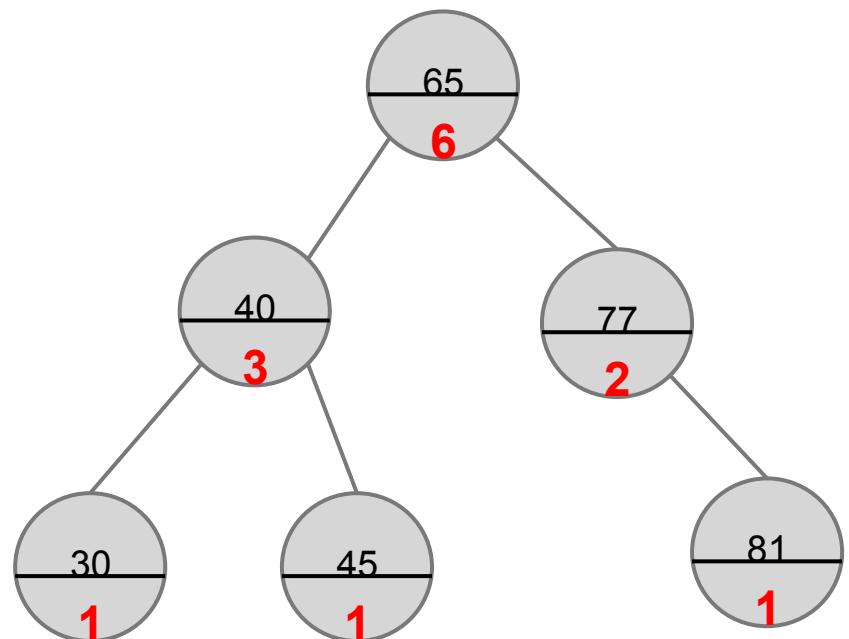
- Use unmodified AVL tree
- AVL tree with additional **node.rank** attribute for each node
- AVL tree with additional **node.size** (size of subtree) attribute for each node

Only one of these works really well, go to the tutorial and find out why!

node.rank



node.size



Which one is better?

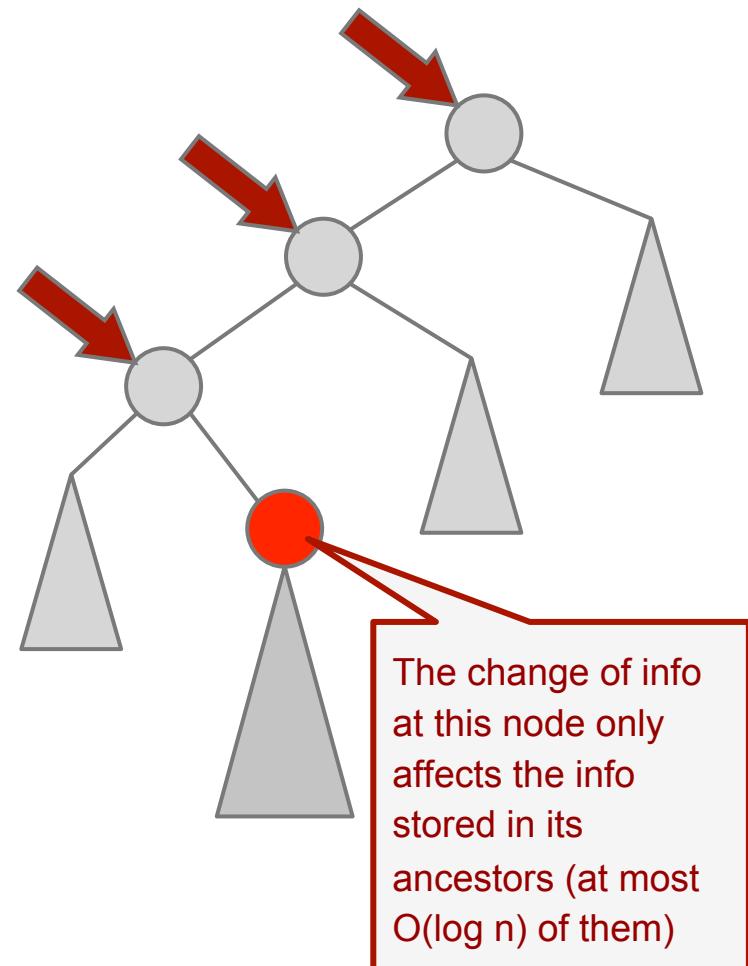
faster Rank(k) ?
easier to maintain?

A useful theorem about AVL tree (or red-black tree) augmentation

Theorem 14.1 of Textbook

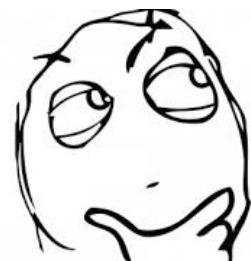
If the additional information of a node **only depends** on the information stored in **its children and itself**...

then this information can be **maintained efficiently** during Insert() and Delete() without affecting their **O(log n)** worst-case runtime.



Next week

→Hash tables



CSC263 Week 5

Announcements

Assignment 1 marks out
class average 70.5%
median 75%

MIDTERM next week!!

MIDTERM:

- Asymptotic analysis (O , Ω)
- Runtime analysis
(worst-case, best-case, average-case)
- Heaps
- Binary Search Trees
- AVL Trees
- Hashing

Short-answer questions, multiple choice

Example: insert/delete x into this heap/avl tree

Data Structure of the Week

Hash Tables

Hash table is for implementing Dictionary

	unsorted list	sorted array	Balanced BST	Hash table
Search(S, k)	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$
Insert(S, x)	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$
Delete(S, x)	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$

average-case, and if
we do it right

Direct address table

a fancy name for “array”...

Problem

Read a grade file, keep track of number of occurrences of each grade (integer 0~100).

The fastest way: create an array $T[0, \dots, 100]$, where $T[i]$ stores the number of occurrences of grade i .

Everything can be done in $O(1)$ time, worst-case.

values:	33	20	35	65	771	332	21	125	...	2
keys:	0	1	2	3	4	5	6	7	100

Direct-address table: directly using the key as the index of the table

The **drawbacks** of direct-address table?

values:	33	20	35	65	771	332	21	125	...	2
keys:	0	1	2	3	4	5	6	7	100

Drawback #1: What if the keys are **not integers**? Cannot use keys as indices anymore!

We need to be able to **convert** any type of key to an **integer**.

Drawback #2: What if the grade 1,000,000,000 is allowed? Then we need an array of size 1,000,000,001! Most space is **wasted**.

We need to map the **universe** of keys into a small number of **slots**.

A **hash function** does both!

An unfortunate naming confusion

Python has a built-in “**hash()**” function

```
>>>  
>>> hash("sdfsadfdsdf")  
-3455985408728624747  
>>>  
>>> hash(3.1415926)  
2319024436  
>>>  
>>> hash(42)  
42  
>>>
```

By our definition, this “**hash()**” function is not really a **hash function** because it only does the first thing (convert to integer) but **not** the second thing (map to a small number of slots).

Definitions

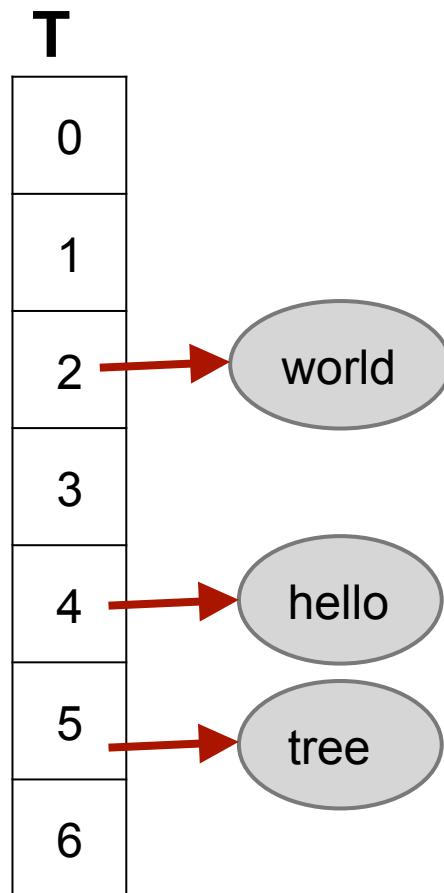
Universe of keys U , the set of all possible keys.

Hash Table T : an array with M positions, each position is called a “slot” or a “bucket”.

Hash function h : a functions maps U to $\{0, 1, \dots, M-1\}$ in other words, $h(k)$ maps any key k to one of the M buckets in table T .

in yet other words, $h(k)$ is the index in T where the key k is stored.

Example: A hash table with $M = 7$



Insert("hello")
assume $h("hello") = 4$

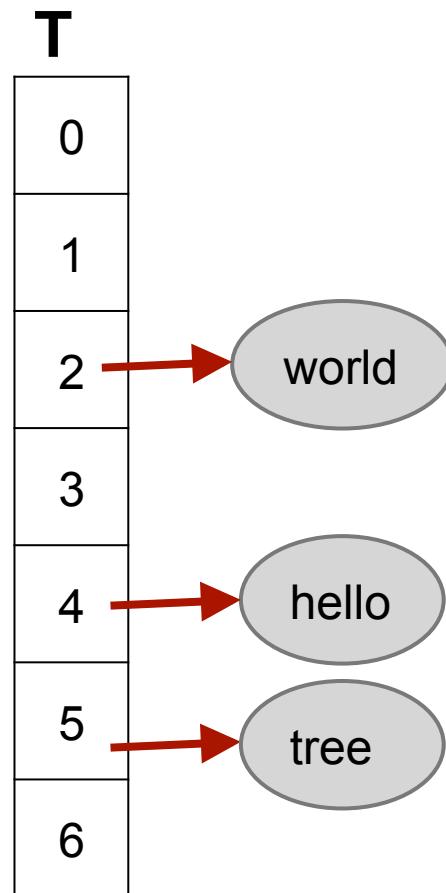
Insert("world")
assume $h("world") = 2$

Insert("tree")
assume $h("tree") = 5$

Search("hello")
return $T[h("hello")]$

What's new potential issue?

Example: A hash table with $M = 7$

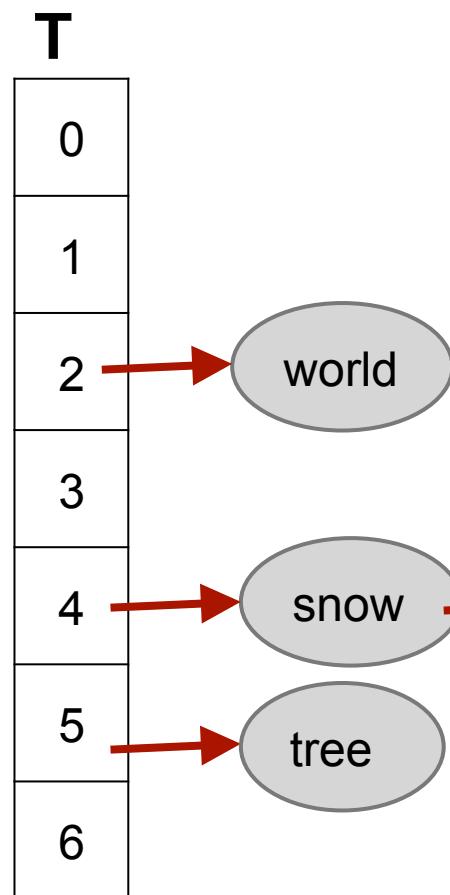


What if we Insert("snow"),
and $h("snow") = 4$?

Then we have a **collision**.

One way to resolve collision is
Chaining

Example: A hash table with $M = 7$



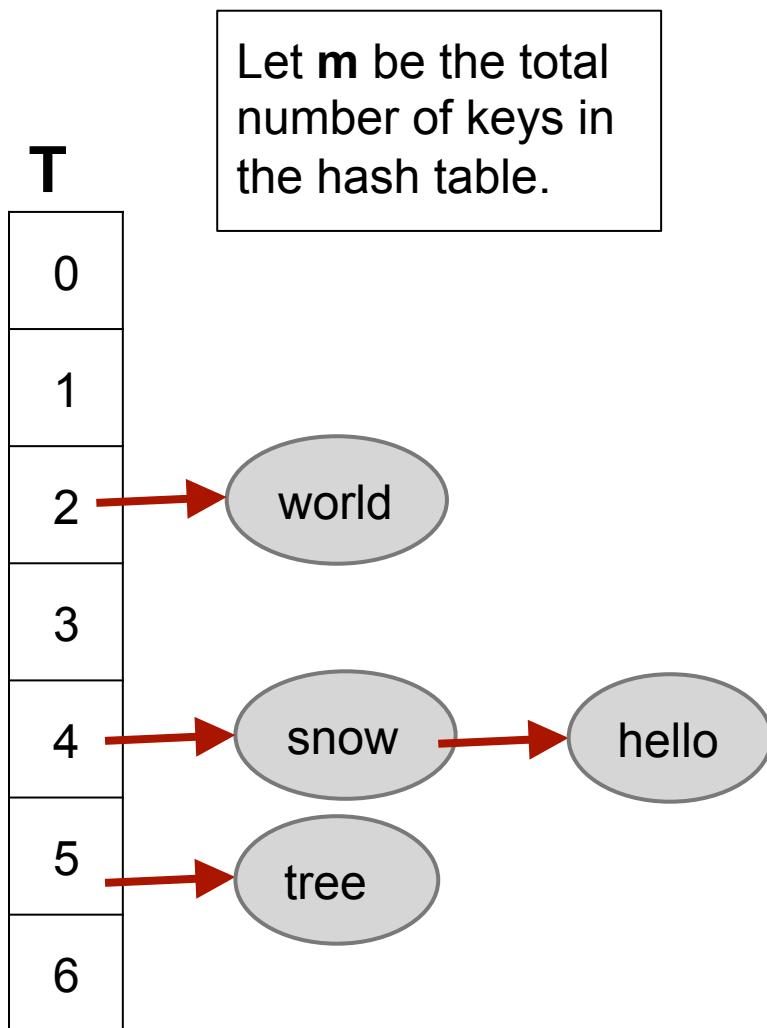
What if we **Insert**("snow"),
and h ("snow") = 4?

Then we have a **collision**.

Store a **linked list** at
each bucket, and insert
new ones at the **head**

One way to resolve collision is
Chaining

Hashing with chaining: Operations



→ Search(k):

- ◆ Search k in the linked list stored at $T[h(k)]$
- ◆ Worst-case $O(\text{length of chain})$,
- ◆ Worst length of chain: $O(m)$ (e.g., all keys hashed to the same slot)

→ Insert(k):

- ◆ Insert into the linked list stored at $T[h(k)]$
- ◆ Need to check whether key already exists, still takes $O(\text{length of chain})$

→ Delete(k)

- ◆ Search k in the linked list stored at $T[h(k)]$, then delete, $O(\text{length of chain})$

Hashing with chaining operations, worst-case running times are $O(m)$ in general. Doesn't sound too good.

However, in practice, hash tables work really well, that is because

- The worst case almost never happens.
- **Average case** performance is **really** good.
(More on this soon!)

In fact, Python “**dict**” is implemented using hash table.

So what can we do?

We use some **heuristics**.

Heuristic

(noun)

A method that works in practice but
you don't really know why.

First of all

Every object stored in a computer can be represented by a **bit-string** (string of 1's and 0's), which corresponds to a (large) **integer**, i.e., any type of key can be converted to an **integer** easily.

So the only thing a **hash function** really needs to worry about is how to **map** these large integers to a small set of integers **{0, 1, ..., M-1}**, i.e., the buckets.

What do we want to have in a hash function?

Want-to-have #1

$h(k)$ depends on **every bit** of k ,
so that the differences between different k 's are
fully considered.

$h(k) = \text{lowest 3-bits of } k$
e.g.,
 $h(101001010001\textcolor{red}{010}) = 2$

bad

$h(k) = \text{sum of all bits}$
e.g.,
 $h(101001010001010) = 6$

a little
better

Want-to-have #2

$h(k)$ “spreads out” values, so all buckets get something.

Assume there are $M = 263$ buckets in the hash table.

$$h(k) = k \bmod 2$$

bad

because all keys
hash to either
bucket 0 or bucket
1

$$h(k) = k \bmod 263$$

better

because all
buckets could get
something

Want-to-have #3

$h(k)$ should be **efficient to compute**

$h(k) = \text{solution to the PDE } * \$^{\%}$ with parameter k

Yuck!

$h(k) = k \bmod 263$

better

1. $h(k)$ depends on every bit of k
2. $h(k)$ “spreads out” values
3. $h(k)$ is efficient to compute

In practice, it is difficult to get all three of them, ...

but there are some **heuristics** that work well

Summary: hash functions

Hash

(noun)

a dish of cooked meat cut into small pieces and cooked again, usually with potatoes.

(verb)

make (meat or other food) into a hash



The spirit of hashing

The division method

The division method

$$h(k) = k \bmod M$$

$h(k)$ is between 0 and $M-1$

Pitfall: sensitive to the value of M

- If $M = 8, \dots$
 - ◆ $h(k)$ just returns the lowest 3-bits of k
- So M better be a **prime number**

A variation of the division method

$$h(k) = (ak + b) \bmod M$$

where a and b are constants that can be picked

Used in “**Universal hashing**” (coming up next!)

→ Achieve constant sized chains in expectation by choosing randomly from this set of hash functions.
(choose a,b randomly)

On Heuristics

- These methods can be good in practice but there is NO GUARANTEE.
- If the hash function h is chosen **in advance**, there will be sets S that hash very badly (and thus all operations will be inefficient.)

So what **else** can we do?

Use **randomness!!**

Randomness

(noun)

A random sequence of events, symbols or steps has no order and does not follow an intelligible pattern or combination

Randomness

- Randomness is a wonderful resource!
- It allows us to fool adversaries.
- It can give faster and simpler algorithms
- MANY applications including: cryptography, data privacy, statistics

Universal Hashing

- Use randomization to achieve good expected behavior of hashing with chaining for **any** subset S of U
- **Idea:** Pick a hash function h from \mathcal{H} at random, where \mathcal{H} is a “nice” family of hash functions H so that:
for any S , the expected number of collisions is constant.

Definitions

Universe of keys U , the set of all possible keys.

$U = [0, 1, \dots, N-1]$

Range = $[0, 1, \dots, M-1]$

Hash functions h : maps $[0, 1, \dots, N-1]$ to $[0, 1, \dots, M-1]$

Let **S** be the (unknown) subset of **U** that is getting mapped to **$[0, 1, \dots, M-1]$**

Let $|S| = m$, and let $m/M = \alpha$ be the load factor
(typically we want to choose $M \sim m$)

Universal Hashing

A family \mathcal{H} of hash functions from $[0, 1, \dots, N-1]$ to $[0, 1, \dots, M-1]$ is **d-universal** if for all j, k in $[0, 1, \dots, N-1]$, $j \neq k$

$$\Pr_{h \text{ in } \mathcal{H}}[h(j) = h(k)] \leq d/M$$

****Think of $d = 1$ (or maybe 2)**

Equivalently let $X_{jk} = 1$ if $h(j)=h(k)$ and 0 otherwise.
Then \mathcal{H} is universal if for all $j \neq k$: $E_h [X_{jk}] \leq d/M$

Universal Hashing: why is d-universal good enough?

Theorem

Let $0 \leq j \leq N-1$, and let \mathbf{S} be a subset of $[0,..,N-1]$, $|S|=m$. Then

$$E_{\mathbf{h} \text{ in } \mathcal{H}}[\# \text{ collisions between } j \text{ and } \mathbf{S}] \leq dm/M$$

[The number of collisions between j and \mathbf{S} is the number of items in \mathbf{S} that map to $h(j)$]

Theorem tells us that each j , the expected chain length of bucket containing j is at most $1+dm/M$
If we pick M so that $m/M = O(1)$, this is constant!

Universal hashing

Proof of Theorem.

Let $C_h(j, S) = \# \text{ collisions between } j \text{ and } S$

Let X_{jk} be the indicator random variable that is
1 if $h(j)=h(k)$ and 0 otherwise

Let $C_h(j, S) = \sum_{k \in S} X_{jk}$

Then:

$$E_h[C_h(j, S)] = \sum_{k \in S} E_h[X_{jk}] \leq \sum_{k \in S} 1/M = md/M$$

Designing a universal family \mathcal{H}

Example 1. The set of all functions from
 $U=[0,\dots,N-1]$ to $[0,\dots,M-1]$ is a universal family.

What is the problem??

Designing a universal family \mathcal{H}

Example 1. The set of all functions from
 $U=[0,\dots,N-1]$ to $[0,\dots,M-1]$ is a universal family.

What is the problem??

This universal family is WAY too large.
Just to write down one h in \mathcal{H} takes
time $>> N > M > m$
BUT we want to run in time $O(\log m)$ or $O(1)$

Designing a universal family \mathcal{H}

Example 2. Let $U=[0, \dots, N-1]$, where N is a prime p , and M divides $p-1$

Let $\mathcal{H} = \{ h_a \mid a=1,2,\dots,p-1 \}$
where $h_a(x) = (ax \bmod p) \bmod M$

Theorem.

$\Pr_{\mathbf{h}} [h(j) = h(k)] \leq 2/M$ (so 2-universal)

Designing a universal family \mathcal{H}

Let $U=[0, \dots, N-1]$, where N is a prime p , and M divides $p-1$

Let $\mathcal{H} = \{ h_a \mid a=1,2,\dots,p-1 \}$

where $h_a(x) = (ax \bmod p) \bmod M$

Theorem.

$$\Pr_h [h(j) = h(k)] \leq 2/M$$

Fact 1. For p prime, $ax \bmod p$ is 1-1, onto for all a

Fact 2. For all $z \neq 0$, for all i : $\Pr_a [az \bmod p = i] = 1/p$

Designing a universal family \mathcal{H}

Fact 1. For p prime, for all a , $(ax \bmod p)$ is bijective

Fact 2. For all $z \neq 0$, for all i : $\Pr_a [az \bmod p = i] = 1/p$

Example: $p=11$

x	0	1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	---	---	----

a=1	0	1	2	3	4	5	6	7	8	9	10
a=2	0	2	4	6	8	10	1	3	5	7	9
a=3	0	3	6	9	1	4	7	10	2	5	8
a=4	0	4	8	1	5	9	2	6	10	4	7
a=5	0	5	10	4	9	3	8	2	7	1	6
a=6	0	6	1	7	2	8	3	9	4	10	5
a=7	0	7	3	10	6	2	9	5	1	8	4
a=8	0	8	5	2	10	7	4	1	9	6	3
a=0	0	9	7	5	3	1	10	8	6	4	2
a=10	0	10	9	8	7	6	5	4	3	2	1

Designing a universal family \mathcal{H}

The facts tells us that all columns (except the first) of the matrix below are different permutations of $[1 \dots p-1]$.

Example: $p=11$

x	0	1	2	3	4	5	6	7	8	9	10
a=1	0	1	2	3	4	5	6	7	8	9	10
a=2	0	2	4	6	8	10	1	3	5	7	9
a=3	0	3	6	9	1	4	7	10	2	5	8
a=4	0	4	8	1	5	9	2	6	10	4	7
a=5	0	5	10	4	9	3	8	2	7	1	6
a=6	0	6	1	7	2	8	3	9	4	10	5
a=7	0	7	3	10	6	2	9	5	1	8	4
a=8	0	8	5	2	10	7	4	1	9	6	3
a=0	0	9	7	5	3	1	10	8	6	4	2
a=10	0	10	9	8	7	6	5	4	3	2	1

Designing a universal family \mathcal{H}

- Fact 1. For p prime, for all a , $(ax \bmod p)$ is bijective
Fact 2. For all $z \neq 0$, for all i : $\Pr_a [az \bmod p = i] = 1/p$

Say $p-1 = cM$ (M divides $p-1$). Then:

$$\begin{aligned} & \Pr_h [h(x)=h(y)] \quad (y < x < p, \text{ maps to something } < M-1) \\ &= \Pr_a [(ax \bmod p) \bmod M = (ay \bmod p) \bmod M] \\ &= \Pr_a [(a(x-y) \bmod p) \bmod M = 0] \\ &= \Pr_a [a(z) \bmod p = 0 \text{ or } M \text{ or } 2M \text{ or } \dots \text{ or } cM] \quad (z \neq 0) \\ &\leq 1/p (c+1) \quad [\text{by Fact 2}] \\ &\leq 1/p (2p/M) \\ &= 2/M \end{aligned}$$

Summary: Universal Hashing

- Start with unknown S
- Randomly pick one hash function, h , from a small, efficient universal hash family and use h to map S
- The expected chain length will be constant!
- **Very important:** this expectation is for all S , over the random choice of h

Open addressing

another way of resolving **collisions**
other than chaining

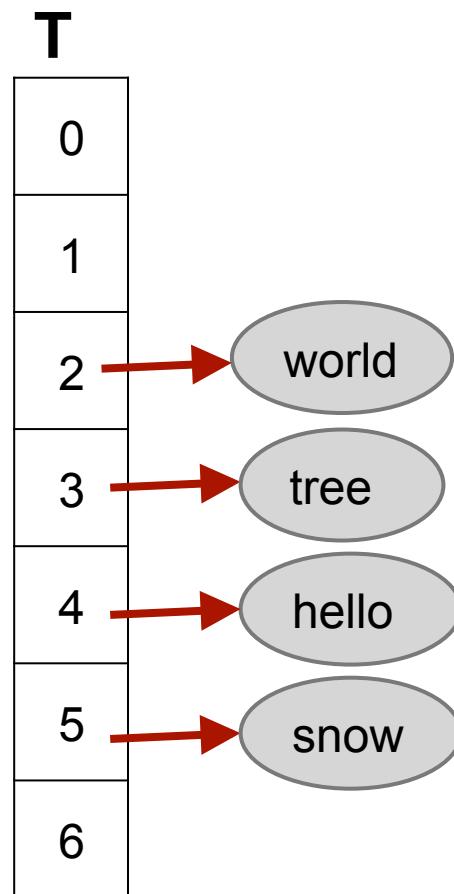
Open addressing

- There is no chain
- Then what to do when having a **collision**?
 - ◆ Find **another bucket** that is **free**
- How to find another bucket that is free?
 - ◆ We **probe**.
- How to probe?
 - ◆ **linear** probing
 - ◆ **quadratic** probing
 - ◆ **double hashing**

Linear probing

Probe sequence:

$$(h(k) + i) \bmod M, \text{ for } i=0,1,2, \dots$$



Insert("hello")

assume $h("hello") = 4$

Insert("world")

assume $h("world") = 2$

Insert("tree")

assume $h("tree") = 2$

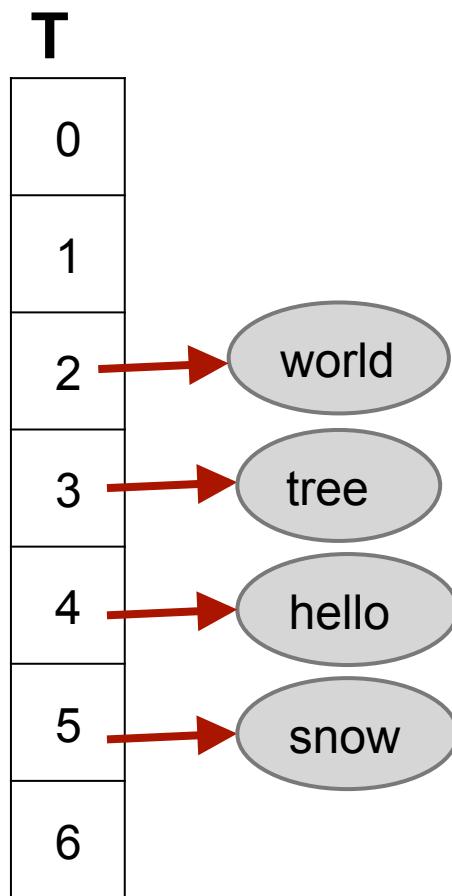
probe 2, 3 ok

Insert("snow")

assume $h("snow") = 3$

probe 3, 4, 5 ok

Problem with linear probing



Keys tend to **cluster**, which causes **long runs** of probing.

Solutions: Jump **farther** in each probe.

before: $h(k), h(k)+1, h(k)+2, h(k)+3, \dots$

after: $h(k), h(k)+1, h(k)+4, h(k)+9, \dots$

This is called **quadratic probing**.

Quadratic probing

Probe sequence

$$(h(k) + c_1 i + c_2 i^2) \bmod M, \text{ for } i=0,1,2,\dots$$

Pitfalls:

- Collisions still cause a milder form of **clustering**, which still cause **long runs** (*keys that collide jump to the same places and form crowd*).
- Need to be careful with the values of c_1 and c_2 , it could jump in such a way that some of the buckets are never **reachable**.

Double hashing

Probe sequence:

$$(\mathbf{h}_1(k) + i\mathbf{h}_2(k)) \bmod M, \text{ for } i=0,1,2,\dots$$

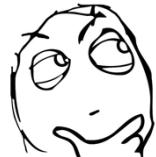
Now the jumps almost look like random, the jump-step ($\mathbf{h}_2(k)$) is different for different k , which helps avoiding clustering upon collisions, therefore avoids long runs (*each one has their own way of jumping, so no crowd*).

Performance of open addressing

Assuming simple uniform hashing, the average-case **number of probes** in an **unsuccessful** search is $1/(1-\alpha)$.

For a **successful** search it is $\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$

In both cases, assume $\alpha < 1$



Open addressing cannot have $\alpha > 1$. Why?

How exactly to do Search, Insert and Delete
work in an open-addressing hash table?

Will see in this week's **tutorial.**

Hashing is one of the most important ideas in Computer Science!!

What you have seen today is just the tip of the iceberg!!

- Perfect hashing
- Cuckoo hashing
- Bloom filter
- Fast string search algorithm
- Geometric hashing
- Cryptography: authentication, message fingerprinting, digital signatures
- Complexity: approximate counting, recycling random bits, interactive proofs



Recap

- **Hash table**: a data structure used to implement the Dictionary ADT.
- **Hash function $h(k)$** : maps any key k to $\{0, 1, \dots, m-1\}$
- Hashing with **chaining**: expected time $O(1+\alpha)$ for search, insert and delete when h chosen randomly from a universal hash family

Next week

→ Randomized algorithms



CSC263 Week 6

Announcements

PS2 marks out today.

Class average 85% !

Midterm tomorrow evening, 8-9pm EX100

Don't forget to bring your ID!

This week

- QuickSort and analysis
- Randomized QuickSort
- Randomized algorithms in general

QuickSort

Background

Invented by **Tony Hoare** in
1960

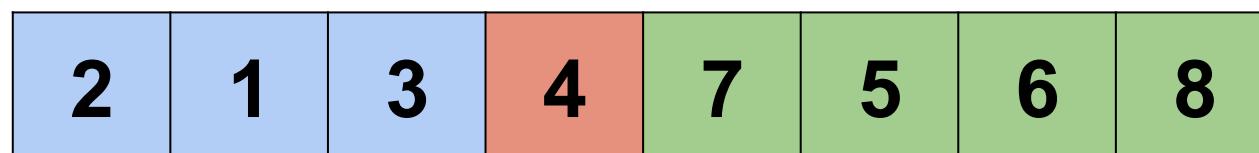
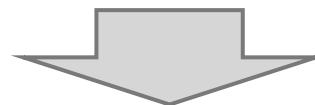
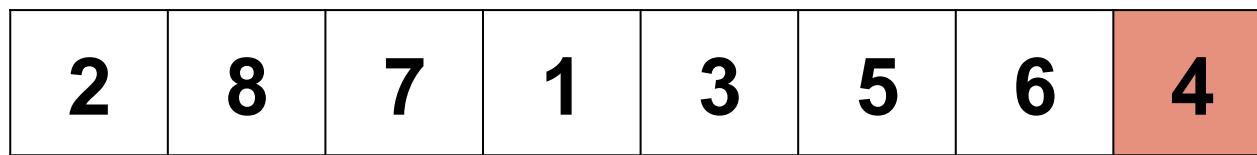
Very commonly used
sorting algorithm. When
implemented well, can be
about 2-3 times faster than
merge sort and **heapsort**.



QuickSort: the idea

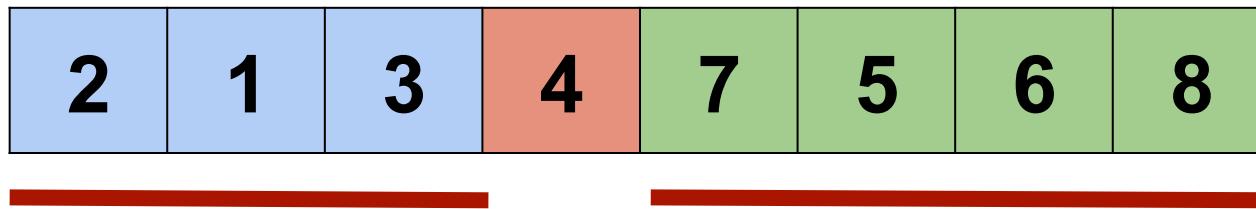
→ Partition an array

pick a **pivot**
(the last one)



smaller than pivot

larger than pivot



Recursively partition the sub-arrays
before and after the pivot.

Base case:

1

sorted

Read textbook Chapter 7
for details of the Partition
operation

Worst-case Analysis of QuickSort

$T(n)$: the total number of **comparisons** made

For simplicity, assume all elements are distinct



Claim 1. Each element in **A** can be chosen as **pivot at most once**.

A pivot never goes into a sub-array on which a recursive call is made.

Claim 2. Elements are **only** compared to **pivots**.

That's what partition is all about -- comparing with pivot.

A



Claim 3. Every **pair** (a, b) in A are compared with each other **at most once**.

The only possible one happens when **a or b** is chosen as a **pivot** and the other is compared to it; after being the pivot, the pivot one will be out of the market and never compare with anyone anymore.

So, the total number of **comparisons** is no more than the **total number of pairs**.

So, the total number of **comparisons** is no more than the **total number of pairs**.

$$T(n) \leq \binom{n}{2} = \frac{n(n - 1)}{2}$$

$$T(n) \in \mathcal{O}(n^2)$$

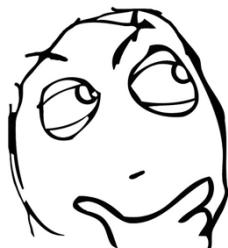
Next, show $T(n) \in \Omega(n^2)$

Show $T(n) \in \Omega(n^2)$

i.e., the **worst-case** running time is
lower-bounded by some cn^2

Just find **one input** for which the
running time is at least cn^2

so, just find **one input** for which the running time is some cn^2



i.e., find one input that results in **awful partitions** (everything on one side).

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

IRONY:
**The worst input for QuickSort
is an already sorted array.**

Remember that we always pick the last one as pivot.

Calculate the number of comparisons

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

Choose pivot **A[n]**, then **n-1** comparisons

Recurse to subarray, pivot **A[n-1]**, then **n-2** comps

Recursive to subarray, pivot **A[n-2]**, then **n-3** comps

...

Total # of comps:

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

So, the worst-case runtime

$$T(n) \geq \frac{n(n - 1)}{2}$$

$$T(n) \in \Omega(n^2)$$

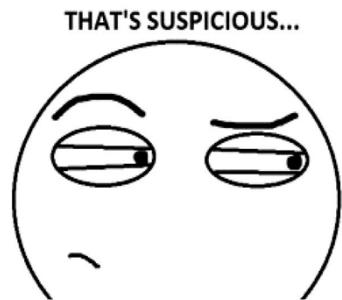
already shown $T(n) \in \mathcal{O}(n^2)$

so, $T(n) \in \Theta(n^2)$

$$T(n) \in \Theta(n^2)$$

What other sorting algorithms have n^2 worst-case running time?

(The stupidest) Bubble Sort!



Is QuickSort really “quick” ?

Yes, in **average-case**.

Average-case Analysis of QuickSort

$O(n \log n)$



Average over what?

Sample space and input distribution

All **permutations** of array [1, 2, ..., n], and each permutation appears **equally likely**.

Not the only choice of sample space, but it is a representative one.

What to compute?

Let **X** be the random variable representing the **number of comparisons** performed on a sample array drawn from the sample space.

We want to compute **E[X]**.

An indicator random variable!

array is a permutation of [1, 2, ..., n]

$$X_{ij} = \begin{cases} 1 & \text{if the values } i \text{ and } j \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$$

So the total number of comparisons:

$$X = \sum_{i=1}^n \sum_{j=i+1}^n X_{ij}$$

sum over all
possible pairs

$$X = \sum_{i=1}^n \sum_{j=i+1}^n X_{ij}$$

$$E[X] = E \left[\sum_{i=1}^n \sum_{j=i+1}^n X_{ij} \right]$$

$$= \sum_{i=1}^n \sum_{j=i+1}^n E[X_{ij}]$$

↑
because
IRV

$$= \sum_{i=1}^n \sum_{j=i+1}^n \Pr(i \text{ and } j \text{ are compared})$$

Just need to figure
this out!

$\Pr(i \text{ and } j \text{ are compared})$

Note: $i < j$

Think about the sorted sub-sequence

$Z_{ij} : i, i + 1, \dots, j$

A Clever Claim: i and j are compared **if and only if**, among all elements in Z_{ij} , the first element to be picked as a **pivot** is **either i or j** .

$$Z_{ij} : i, i + 1, \dots, j$$

Claim: i and j are compared **if and only if**, among all elements in Z_{ij} , the first element to be picked as a **pivot** is **either i or j** .

Proof:

The “**only if**”: suppose the first one picked as pivot as some k that is between i and j ,...
then i and j will be separated into **different partitions** and will never meet each other.

The “**if**”: if i is chosen as pivot (the **first one** among Z_{ij}), then j will be compared to pivot i for sure, because nobody could have possibly separated them yet!

Similar argument for first choosing j

$$Z_{ij} : i, i+1, \dots, j$$

Claim: i and j are compared **if and only if**, among all elements in Z_{ij} , the first element to be picked as a **pivot** is **either i or j** .

$$\begin{aligned} & \Pr(i \text{ and } j \text{ are compared}) \\ &= \Pr(i \text{ or } j \text{ is the first among } Z_{ij} \text{ chosen as pivot}) \end{aligned}$$

$$= \frac{2}{j - i + 1}$$

There are **$j-i+1$** numbers in Z_{ij} , and each of them is **equally likely** to be chosen as the first pivot.

$$X = \sum_{i=1}^n \sum_{j=i+1}^n X_{ij}$$

$$E[X] = E \left[\sum_{i=1}^n \sum_{j=i+1}^n X_{ij} \right]$$

$$= \sum_{i=1}^n \sum_{j=i+1}^n E[X_{ij}]$$

We have figured
this out!

$$= \sum_{i=1}^n \sum_{j=i+1}^n \Pr(i \text{ and } j \text{ are compared})$$

$$E[X] = \sum_{i=1}^n \sum_{j=i+1}^n \Pr(i \text{ and } j \text{ are compared})$$

$$= \sum_{i=1}^n \sum_{j=i+1}^n \frac{2}{j - i + 1}$$

$$\in \mathcal{O}(n \log n)$$

Analysis Over!

Something
close to

$$n \sum_{k=1}^n \frac{1}{k}$$

$$= \sum_{i=1}^n \sum_{j=i+1}^n \frac{2}{j-i+1}$$

$$\leq 2n (1 + 1/2 + 1/3 + 1/4 + 1/5 + \dots + 1/n)$$

$$\in \mathcal{O}(n \log n)$$

Why is $(1 + 1/2 + 1/3 + 1/4 + .1/5 + \dots + 1/n) \leq \log n$?

Divide sum into $(\log n)$ groups:

$$S_1 = 1$$

$$S_2 = 1/2 + 1/3$$

$$S_3 = 1/4 + 1/5 + 1/6 + 1/7$$

$$S_4 = 1/8 + 1/9 + 1/10 + 1/11 + 1/12 + 1/13 + 1/14 + 1/15$$

Each group sums to a number ≤ 1 , so total sum of all groups is $\leq \log n$!

Summary

The worst-case runtime of Quicksort is $\Theta(n^2)$.

The average-case runtime is $O(n \log n)$.

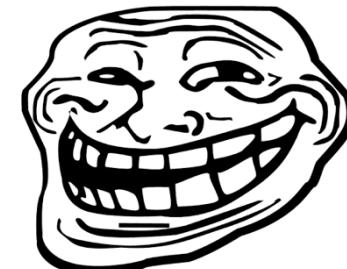
(over all permutations of $[1, \dots, n]$)

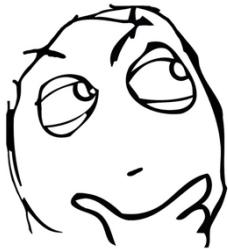
However, in real life....

Average case analysis tells us that for most inputs the runtime is $O(n \log n)$, but this is a small consolation if our input is one of the bad ones!

QuickSort(A)

The theoretical $O(n \log n)$ performance is in no way guaranteed in real life.





Let's try to get around this problem by adding randomization into the algorithm itself:

```
Randomize-QuickSort(A):  
    run QuickSort(A) as above  
    but each time picking a random  
    element in the array as a pivot
```



Let's try to get around this problem by adding randomization into the algorithm itself:

```
Randomize-QuickSort(A):  
    run QuickSort(A) as above  
    but each time picking a random  
    element in the array as a pivot
```

- We will prove that for **any** input array of n elements, the expected time is $O(n \log n)$
- This is called a **worst-case expected time bound**
- We no longer assume any special properties of the input

Worst-case Expected Runtime of Randomized QuickSort

$O(n \log n)$



What to compute?

Let **X** be the random variable representing the **number of comparisons** performed on a sample array drawn from the sample space.

We want to compute **E[X]**.

Now the expectation is over the random choices for the pivot, and the input is fixed.

An indicator random variable!

array is a permutation of [1, 2, ..., n]

$$X_{ij} = \begin{cases} 1 & \text{if the values } i \text{ and } j \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$$

So the total number of comparisons:

$$X = \sum_{i=1}^n \sum_{j=i+1}^n X_{ij}$$

sum over all
possible pairs

$$X = \sum_{i=1}^n \sum_{j=i+1}^n X_{ij}$$

$$E[X] = E \left[\sum_{i=1}^n \sum_{j=i+1}^n X_{ij} \right]$$

$$= \sum_{i=1}^n \sum_{j=i+1}^n E[X_{ij}]$$

↑
because
IRV

$$= \sum_{i=1}^n \sum_{j=i+1}^n \Pr(i \text{ and } j \text{ are compared})$$

Just need to figure
this out!

$$Z_{ij} : i, i+1, \dots, j$$

Claim: i and j are compared **if and only if**, among all elements in Z_{ij} , the first element to be picked as a **pivot** is **either i or j** .

$\Pr(i \text{ and } j \text{ are compared})$

$= \Pr(i \text{ or } j \text{ is the first among } Z_{ij} \text{ chosen as pivot})$

$$= \frac{2}{j - i + 1}$$

There are **$j-i+1$** numbers in Z_{ij} , and each of them is **equally likely** to be chosen as the first pivot.

A Different Analysis (less clever)

$T(n)$ is expected time to sort n elements. First pivot chooses i^{th} smallest element, all equally likely. Then:

$$T(n) = (n - 1) + \frac{1}{n} \sum_{i=0}^{n-1} (T(i) + T(n - i - 1)).$$

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

Solving this recurrence gives $T(n) \leq O(n \log n)$

Randomized Algorithms

Use randomization to guarantee expected performance

We do it everyday.



Two types of randomized algorithms

“Las Vegas” algorithm

→ Deterministic **answer**, random **runtime**

“Monte Carlo” algorithm

→ Deterministic **runtime**, random **answer**

Randomized-QuickSort is a ...

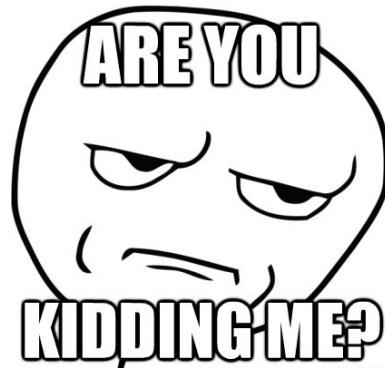
Las Vegas algorithm

An Example of Monte Carlo Algorithm

“Equality Testing”

The problem

Alice holds a binary number x and Bob holds y , decide whether $\mathbf{x = y}$.



No kidding, what if the **size** of x and y are **10TB** each?
Alice and Bob would need to transmit $\sim 10^{14}$ bits.

Can we do better?



Why assuming x and y are of the same length?

Let $n = \text{len}(x) = \text{len}(y)$ be the length of x and y .

Randomly choose a **prime number** $p \leq n^2$,
then $\text{len}(p) \leq \log_2(n^2) = 2\log_2(n)$
then compare $(x \bmod p)$ and $(y \bmod p)$
i.e., return $(x \bmod p) == (y \bmod p)$

Need to compare **at most $2\log(n)$** bits.

But, does it give the correct answer?

$$\log_2(10^{14}) \approx 46.5$$

Huge improvement on runtime!

Does it give the correct answer?

If $(x \bmod p) \neq (y \bmod p)$, then...

Must be **$x \neq y$** , our answer is correct **for sure**.

If $(x \bmod p) = (y \bmod p)$, then...

Could be **$x = y$ or $x \neq y$** , so our answer **might be correct**.

Correct with what probability?

What's the probability of a wrong answer?

Prime number theorem

In range $[1, m]$, there are roughly $m/\ln(m)$ prime numbers.

So in range $[1, n^2]$, there are

$$n^2/\ln(n^2) = n^2/2\ln(n) \text{ prime numbers.}$$

How many (**bad**) primes in $[1, n^2]$ satisfy
 $(x \bmod p) = (y \bmod p)$ even if $x \neq y$?

At most n

$(x \bmod p) = (y \bmod p) \Leftrightarrow |x - y| \text{ is a multiple of } p, \text{ i.e., } p \text{ is a divisor of } |x - y|.$
 $|x - y| < 2^n$ (**n -bit binary #**) so it has no more than n prime divisors (**otherwise it will be larger than 2^n**).

So...

Out of the $n^2/2\ln(n)$ prime numbers we choose from, at most n of them are **bad**.

If we choose a **good** prime, the algorithm gives correct answer for sure.

If we choose a **bad** prime, the algorithm may give a wrong answer.

So the prob of wrong answer is less than

$$\frac{n}{n^2/(2 \ln n)} = \frac{2 \ln n}{n}$$

Error probability of our Monte Carlo algorithm

$$\Pr(\text{error}) \leq \frac{2 \ln n}{n}$$

When $n = 10^{14}$ (10TB)

$\Pr(\text{error}) \leq 0.00000000000644$

Performance comparison ($n = 10\text{TB}$)

The **regular** algorithm $x == y$

- Perform 10^4 comparisons
- Error probability: 0

The **Monte Carlo** algorithm $(x \bmod p) == (y \bmod p)$

- Perform < 100 comparisons
- Error probability: 0.000000000000644

If your boss says: “This error probability is too high!”

Run it **twice**: Perform < 200 comparisons

- Error prob squared: 0.0000000000000000000000000000215

Summary

Randomized algorithms

- Guarantees worst-case expected performance
- Make algorithm less vulnerable to malicious inputs

Monte Carlo algorithms

- Gain time efficiency by sacrificing some correctness.

For more details:

**Notes on Randomized Algorithms and
Quicksort**

posted on course webpage, lecture 6

- Also gives a good review of probability theory and computing expectations!

CSC263 Week 7

Announcements

Problem Set 3 is due this Tuesday!

**Midterm graded and will be returned on Friday
during tutorial (average 60%)**



Amortized Analysis

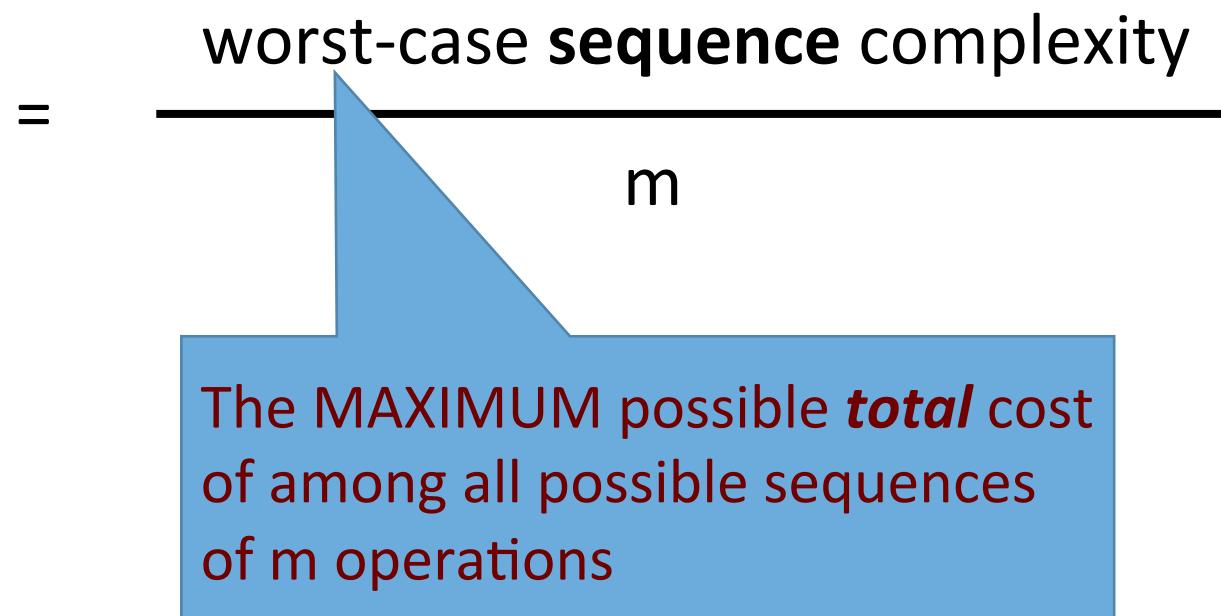
- Often, we perform **sequences** of operations on data structures
- Define "**worst-case sequence complexity**" of a sequence of m operations as:
maximum total time over all sequences of m operationsSimilar to worst-case time for one operation
- Worst-case sequence complexity is at most:
 $m(\text{worst-case complexity of any operation})$
- But is it really always that bad?

Amortized analysis

- We do amortized analysis when we are interested in the total complexity of a **sequence** of operations.
 - Unlike in average-case analysis where we are interested in a **single** operation.
- The ***amortized sequence complexity*** is the “average” cost per operation **over the sequence**.
 - But unlike average-case analysis, there is **NO** probability or expectation involved.

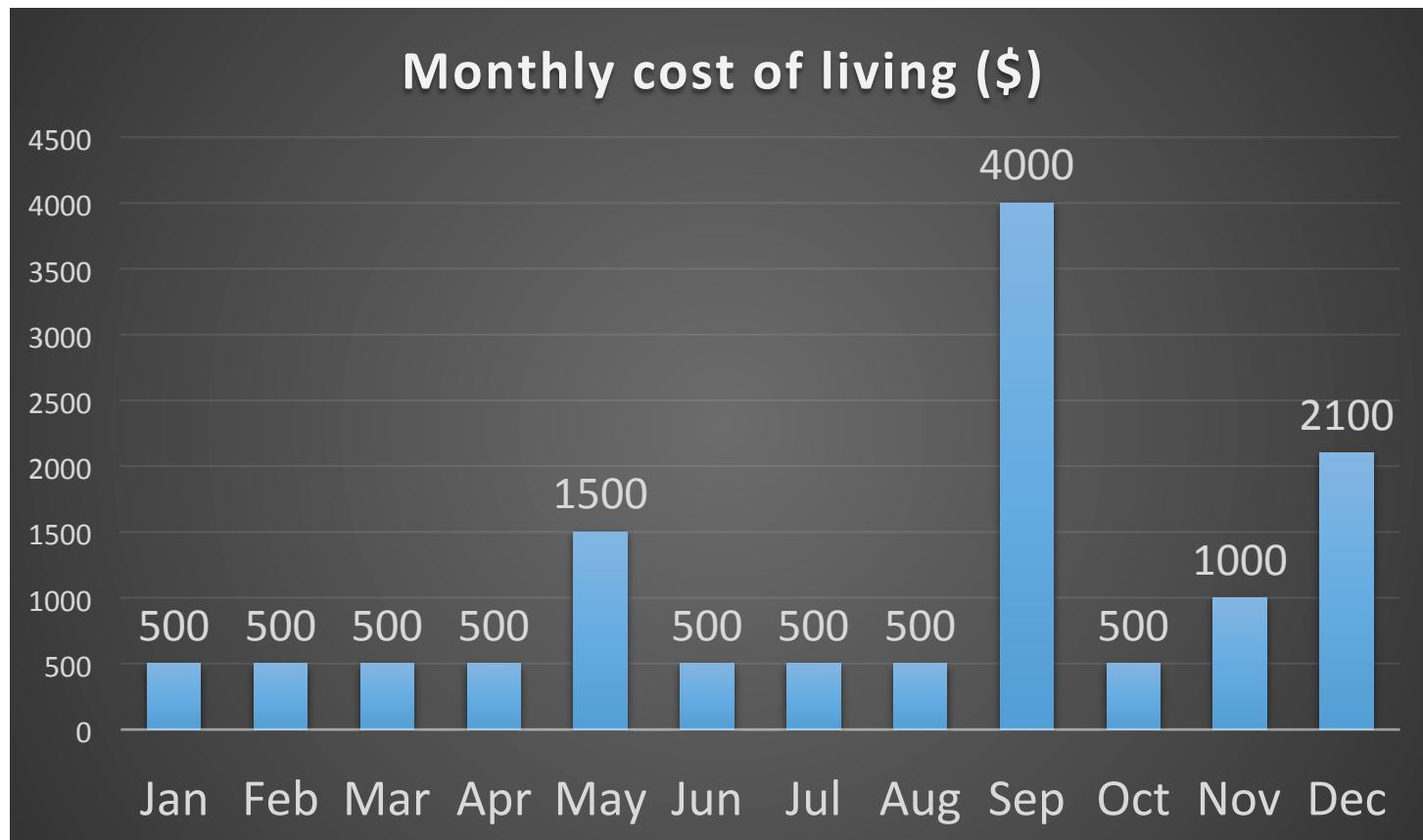
For a sequence of m operations:

Amortized sequence complexity



Amortized analysis

- Real-life intuition: Monthly cost of living, a sequence of 12 operations



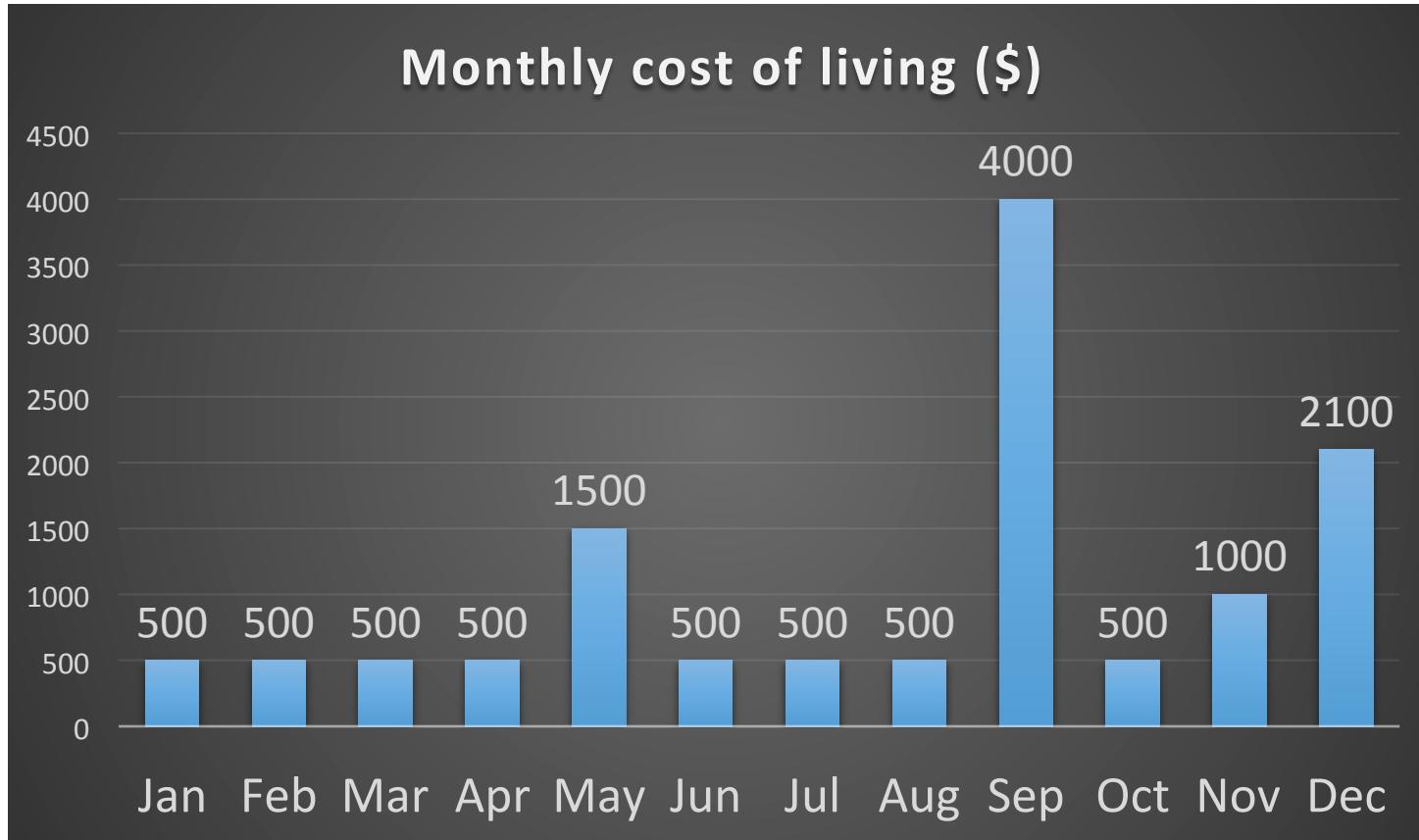
Methods for amortized analysis

- Aggregate method
- Accounting method
- Potential method (skipped, read Chapter 17 if interested)

Aggregate method

What is the amortized cost per month (operation)?

Just **sum up** the costs of all months (operations) and **divide** by the number of months (operations).



Aggregate method: sum of all months' spending is \$126,00, divided by 12 months
– the amortized cost is \$1,050 per month.

Binary Counter

- Sequence of k bits (k fixed)
- Single operation INCREMENT: add 1 (in binary)
- Cost of one INCREMENT: number of bits that need to change

Binary Counter

	0000
Add 1	0001
Add 1	0010
Add 1	0011
Add 1	0100
Add 1	0101
Add 1	0110
Add 1	0111
Add 1	1000

Binary Counter

Initially	0000	Cost
Add 1	0001	1
Add 1	0010	2
Add 1	0011	1
Add 1	0100	3
Add 1	0101	1
Add 1	0110	2
Add 1	0111	1
Add 1	1000	4

Binary Counter

- If we do n increments, the worst case complexity of any increment is $\log n$.
- A naïve analysis would then say that the worst case complexity of n increments is $(n \log n)$
- But it is never this bad since the worst case only happens once!

Aggregate Method for Binary Counter

Amortized cost of sequence of n INCREMENT operations:

bit	changes	total number of changes
0	every operation	n
1	every 2 operations	$n/2$
2	every 4 operations	$n/4$
...
i	every 2^i operations	$n/2^i$

Total number of bit flips during sequence = \sum_i (flips of bit i)

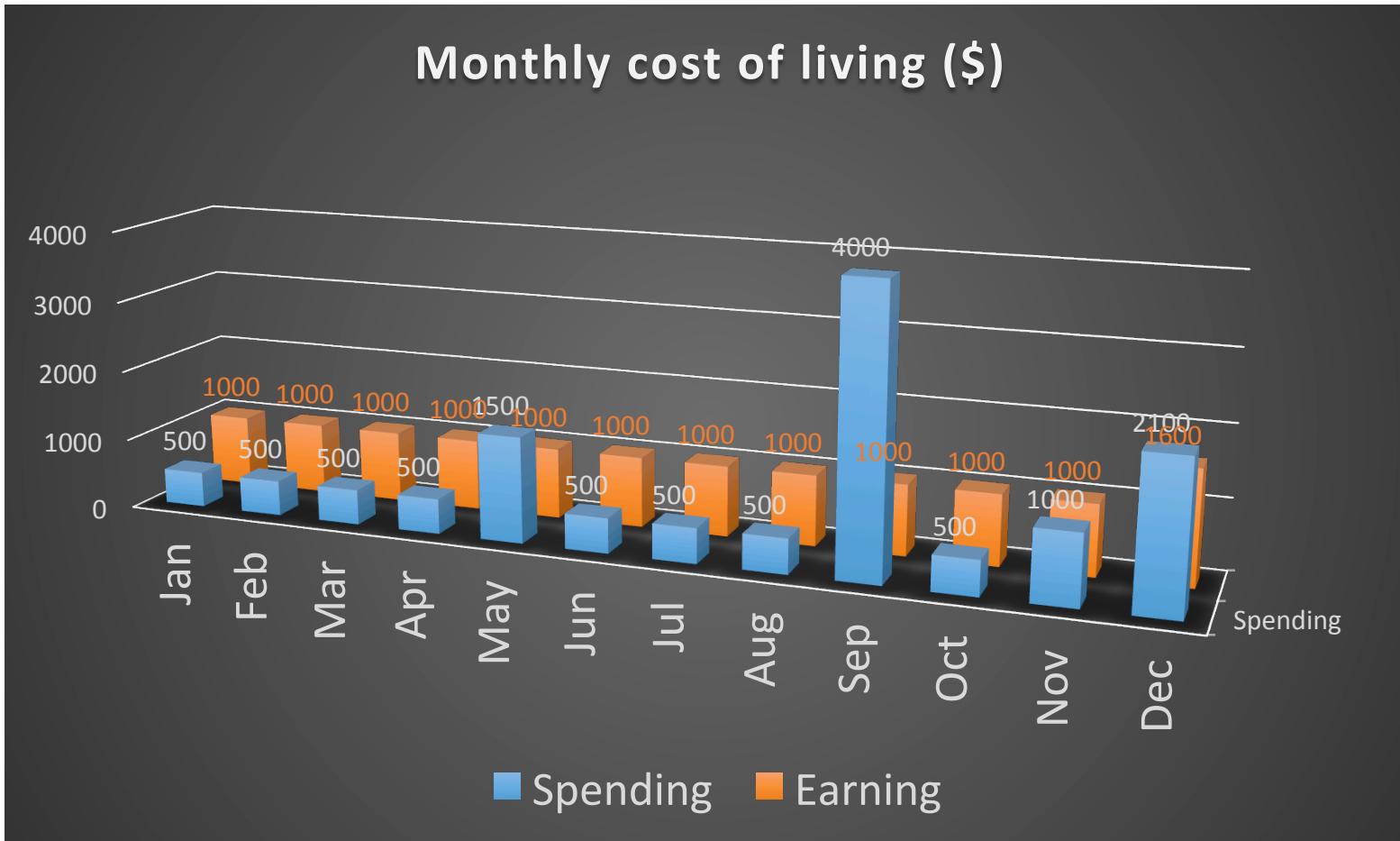
$$= n + n/2 + \dots + n/2^{\log n} = n(1 + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{2^{\log n}}) \leq 2n$$

So amortized cost $\leq 2n/n = 2$ for each operation!

Accounting method

Instead of calculating the average spending, we think about the cost from a **different angle**, i.e.,

How much money do I need to **earn** each month in order to **keep living**? That is, be able to pay for the spending every month and **never become broke**.



Accounting method: if I **earn** \$1,000 per month from Jan to Nov and earn \$1,600 in December, I will never become broke (assuming earnings are paid at the beginning of month).

So the **amortized cost**: \$1,000 from Jan to Nov and \$1,600 in Dec.

Aggregate vs Accounting

- Aggregate method is easy to do when the cost of each operation in the sequence is concretely defined.
- Accounting method is more interesting
 - It works even when the sequence of operation is not concretely defined
 - It can obtain more refined amortized cost than aggregate method (different operations can have different amortized cost)

Accounting Method

- Find a charge (some number of time units charged per operation) such that:
the sum of the charges is an upper bound on the total actual cost
- Like maintaining a bank account
 - Low cost operations charged a bit more than their actual amount
the surplus is deposited in the account for later use
- Analogy: Rahul earns 2K per month. Typically he spends 2K. On good months, he spends < 2K, and surplus goes in the bank to pay for the bad (expensive) months.
- Charges must be set high enough so that the balance is always positive.
- But if set too high: upper bound will be $>>$ the (worst case) total actual cost.
- Goal: just scrape by -- Set charges as low as possible so that bank account is always positive

Accounting Method

- We want to show amortized cost is, say \$5
- Assign a **charge** for each operation
- When **charge** > actual cost, leftover amount is assigned as credit (usually to specific elements in data structure)
- When an operation's **charge** < actual cost, use some stored credit to “pay” for excess cost.
- For this to work, need to argue that credit is never negative

If we have more than one operation, we can assign different charges to each one

Accounting Method for Binary Counter

- Charge each operation \$2
 - \$1 to flip $0 \rightarrow 1$ (only one bit flips from 0 to 1)
used stored credits to pay for flips $1 \rightarrow 0$
 - \$1 credit -- store with the bit just changed to 1
- **Credit Invariant:** At any step each bit of the counter that is equal to 1 will have \$1 credit

Accounting Method for Binary Counter

Credit Invariant: At any step each bit of the counter that is equal to 1 will have \$1 credit

Proof by induction:

- Initially counter is 0 and no credit
- Induction step: assume true up to some value of x and now consider next increment

Case 1: $x = b \dots b b 0 1 \dots 1 \rightarrow b \dots b b 1 0 \dots 0$

(i least significant bits are 1, $i+1^{\text{st}}$ bit is 0)

$i+1$ = actual cost: use i credits to pay for i flips $1 \rightarrow 0$

use 1 out of 2 to pay for $0 \rightarrow 1$,

use 1 out of 2 for credit on the new “1”

Accounting Method for Binary Counter

Credit Invariant: At any step each bit of the counter that is equal to 1 will have \$1 credit

Proof by induction:

- Initially counter is 0 and no credit
- Induction step: assume true up to some value of x and now consider next increment

Case 2: $x = 11 \dots 1 \rightarrow 00 \dots 0$

(all bits are 1)

actual cost is k

use k credits to pay for k flips $1 \rightarrow 0$

extra \$2 isn't needed.

Accounting Method for Binary Counter

Credit Invariant: At any step each bit of the counter that is equal to 1 will have \$1 credit

- Thus invariant is always true
- So total charge for sequence is upper bound on total cost.
- Total charge = $2n$ so amortized cost per operation = 2

NOTE: you need the invariant in order to show that the credit is always positive

Amortized Analysis on Dynamic Arrays

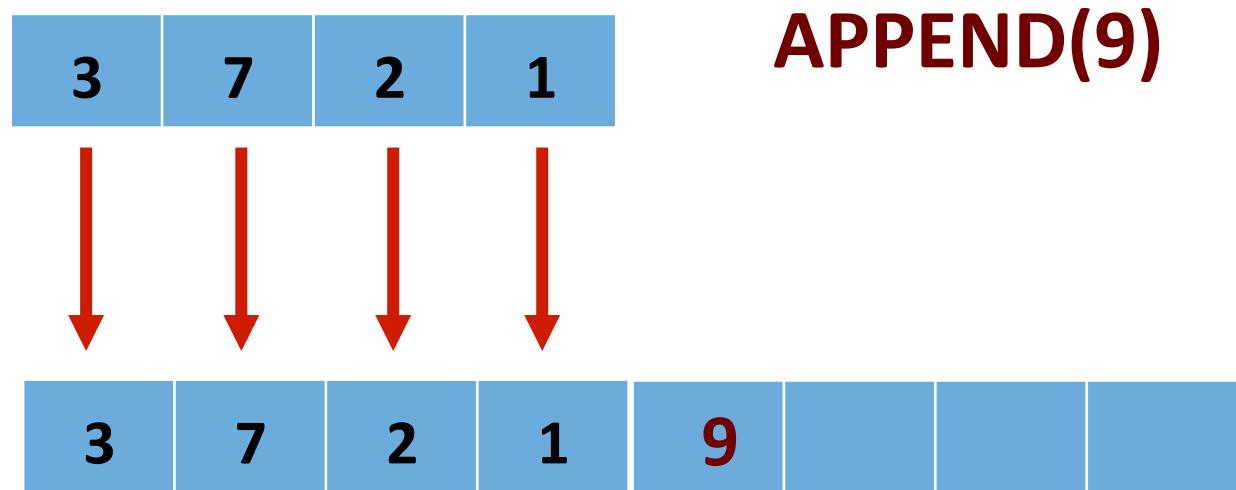
Problem description

- Think of an **array** initialized with a **fixed** number of slots, and supports **APPEND** and **DELETE** operations.
- When we APPEND too many elements, the array would be **full** and we need to **expand** the array (make the size larger).
- When we DELETE too many elements, we want to **shrink** the array (make the size smaller).
- Requirement: the array must be using **one contiguous block** of memory all the time.

How do we do the **expanding** and **shrinking**?

One way to **expand**

- If the array is full when APPEND is called
 - Create a new array of **twice** the size
 - Copy all the elements from old array to new array
 - Append the element



Amortized analysis of **expand**

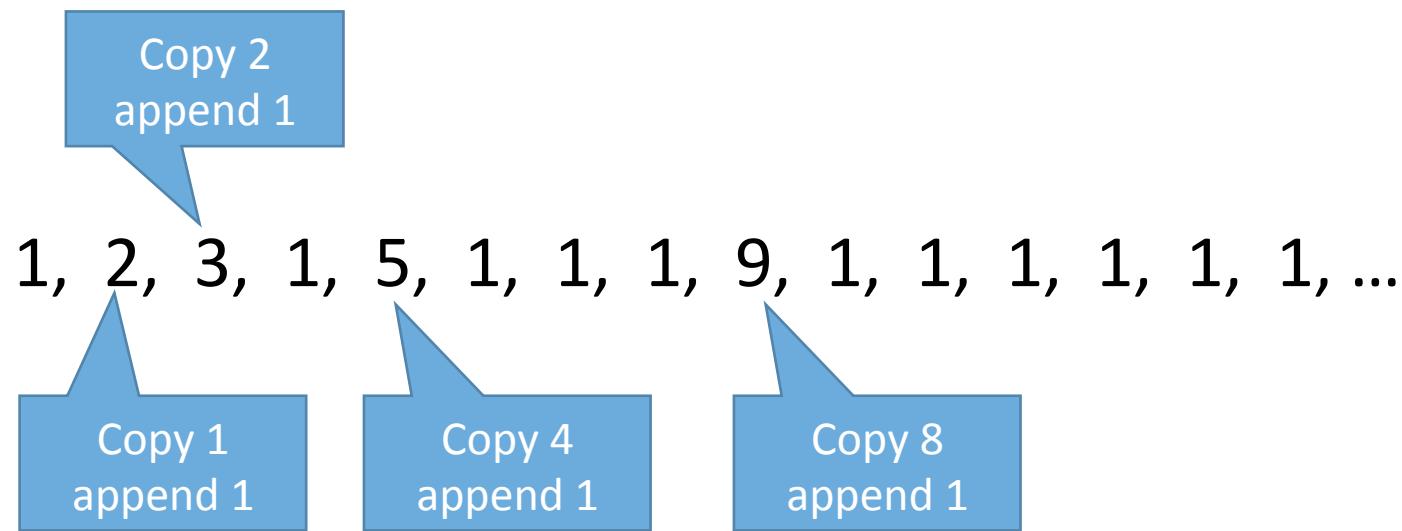
Now consider a dynamic array initialized with size 1 and a sequence of m APPEND operations on it.

Analyze the amortized cost per operation

*Assumption: only count array assignments, i.e.,
append an element and **copy** an element*

Use the aggregate method

The cost sequence would be like:



Cost sequence concretely defined, sum-and-divide can be done, but we want to do something more interesting...

Use the **accounting** method!

*How much money do we need to **earn** at each operation,
so that all future costs can be paid for?*

*How much money to earn for **each APPEND'ed element** ?*

\$1 ?

\$2 ?

\$3 ?

\$log m ?

\$m ?

First try: charge \$1 for each append

This \$1 (the “append-dollar”) is spent when appending the element.

But, when we need to copy this element to a new array (when expanding the array), we don’t have any money to pay for it --

BROKE!

This makes sense since the total cost of n appends is greater than n



Next try: charge \$2 for each append

\$1 (the “append-dollar”) will be spent when appending the element

\$1 (the “copy-dollar”) will be spent when copying the element to a new array

What if the element is copied for a **second** time (when expanding the array for a second time)?

BROKE!



Third try: charge \$3 for each append

\$1 (the “append-dollar”) will be spent when appending the element

\$1 (the “copy-dollar”) will be spent when copying the element to a new array

\$1 (the “recharge-dollar”) is used to **recharge** the old elements that have spent their “copy-dollars”.

So one dollar stored to pay for my copy, and one for a friend

NEVER BROKE!



\$1 (the “recharge-dollar”) is used to **recharge** the old elements that have used their “copy-dollar”.



Old elements who have used their “copy-dollars”

New elements each of whom **spares** \$1 for recharging one old element’s “copy-dollar”.

There will be enough new elements who will spare **enough money** for all the old elements, because the way we expand – **TWICE the size**

Third try: charge \$3 for each append

\$1 (the “append-dollar”) to pay for append

\$1 (the “copy-dollar”) as credit to pay for copy

\$1 (the “recharge-dollar”) as credit to pay for friends’ copy

Credit invariant:

Each element in 2nd half of array has \$2 credit

Third try: charge \$3 for each append

\$1 (the “append-dollar”) to pay for append

\$1 (the “copy-dollar”) as credit to pay for copy

\$1 (the “recharge-dollar”) as credit to pay for friends’ copy

Credit invariant:

Each element in 2nd half of array has \$2 credit

Base case: no elements in array so true

Third try: charge \$3 for each append

\$1 (the “append-dollar”) to pay for append

\$1 (the “copy-dollar”) as credit to pay for copy

\$1 (the “recharge-dollar”) as credit to pay for friends’ copy

Credit invariant:

Each element in 2nd half of array has \$2 credit

Inductive step.

Case 1: array not full

\$1 to append, \$2 stored on new item

Third try: charge \$3 for each append

\$1 (the “append-dollar”) to pay for append

\$1 (the “copy-dollar”) as credit to pay for copy

\$1 (the “recharge-dollar”) as credit to pay for friends’ copy

Credit invariant:

Each element in 2nd half of array has \$2 credit

Inductive step.

Case 2: Array full; make new array

Copy all items using stored credit

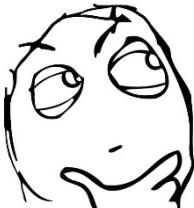
Add new item (\$1) plus \$2 credit

So in all cases credit invariant is maintained

If we charge \$3 for each APPEND it is enough units to pay for all costs in any sequence of APPEND operations (starting with an array of size 1)

In other words, for a sequence of m APPEND operations, the amortized cost per operations is **3**, which is in $O(1)$.

In a regular worst-case analysis (non-amortized), what is the worst-case runtime of an APPEND operation on an array with m elements?



By performing the amortized analysis, we showed that “**double the size when full**” is a good strategy for expanding a dynamic array, since its amortized cost per operation is in $O(1)$.

In contrast, “**increase size by 100 when full**” would not be a good strategy. Why?



Takeaway

Amortized analysis provides us valuable insights into what is the proper strategy of expanding dynamic arrays.

Expanding and Shrinking dynamic arrays

A bit trickier...

First thing that comes to mind...

When the array is $\frac{1}{2}$ full after DELETE, create a new array of half of the size, and copy all the elements.

Consider the following sequence of operations performed on a **full** array with n element...

APPEND, DELETE, APPEND, DELETE, APPEND, ...

$\Theta(n)$ amortized cost per operation since every APPEND or DELETE causes allocation of new array.

NO GOOD!

The right way of shrinking

When the array is $\frac{1}{4}$ full after DELETE, create a new array of $\frac{1}{2}$ of the size, and copy all the elements.

Charge \$3 per APPEND

\$2 per DELETE

- 1 append/delete-dollar
- 1 copy-dollar
- 1 recharge-dollar

The array, after shrinking...



Elements who just spent
their copy-dollars

Array is half-empty

Before the **next expansion**, we need to **fill** the empty half, which will spare enough money for copying the **green** part.

Before the **next shrinking**, we need to **empty** half of the **green** part, which will spare enough money for copying what's left.



Credit Invariant for Dynamic Arrays: Append and Delete

Credit Invariant:

In an array of size 2^k there are at least $2^k / 4$ elements.

Elements in rightmost half have \$2 stored.

Empty slots in leftmost half have \$1 stored

Proof

Base case: First operation is an insert

\$1 to pay for append, \$2 stored

Credit Invariant for Dynamic Arrays: Append and Delete

Credit Invariant:

In an array of size 2^k there are at least $2^k / 4$ elements.

Elements in rightmost half have \$2 stored.

Empty slots in bottom half have \$1 stored

Proof

Inductive Step: four cases

- (a) append without overflow
- (b) append with overflow
- (c) delete without shrinking
- (d) delete with shrinking

Credit Invariant for Dynamic Arrays: Append and Delete

Credit Invariant:

In an array of size 2^k there are at least $2^k / 4$ elements.

Elements in rightmost half have \$2 stored.

Empty slots in bottom half have \$1 stored

Inductive Step: four cases

(a) append without overflow:

2

a1: a b c d e X X X → a b c d e f X X

2 2

1 1

a2: a b X X X X X X → a b c X X X X X

1

Credit Invariant for Dynamic Arrays: Append and Delete

Credit Invariant:

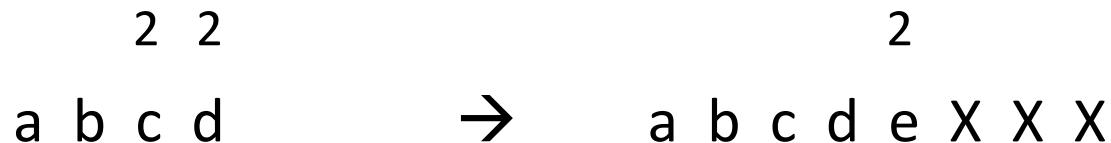
In an array of size 2^k there are at least $2^k / 4$ elements.

Elements in rightmost half have \$2 stored.

Empty slots in bottom half have \$1 stored

Inductive Step: four cases

(b) append with overflow:



credit pays to copy old stuff

\$3: store \$2, and use \$1 to pay for new copy

Credit Invariant for Dynamic Arrays: Append and Delete

Credit Invariant:

In an array of size 2^k there are at least $2^k / 4$ elements.

Elements in rightmost half have \$2 stored.

Empty slots in bottom half have \$1 stored

Inductive Step: four cases

(c) delete no shrinking:

2 2

c1: a b c d e f X X → a b c d e X X X

1

c2: a b c X X X X X → a b X X X X X X X

\$2 charge: \$1 for delete, \$1 for credit

Credit Invariant for Dynamic Arrays: Append and Delete

Credit Invariant:

In an array of size 2^k there are at least $2^k / 4$ elements.

Elements in rightmost half have \$2 stored.

Empty slots in bottom half have \$1 stored

Inductive Step: four cases

(d) delete with shrinking:

1 1 1
a b c X X X X X → a b X X X X X X → a b X X

First delete c, \$1 to delete, \$1 stored on new blank spot

Then shrink – dollars stored will pay for copy of all guys to left

So, overall credit invariant maintained

Summary: In a dynamic array, if we expand and shrink the array as discussed (double on full, halve on $\frac{1}{4}$ full) then:

For any sequence of APPEND or DELETE operations, \$3 per APPEND and \$2 per DELETE is enough money to pay for all costs in the sequence.

Therefore the amortized cost per operation of any sequence is upper-bounded by 3, i.e., O(1).

Next week

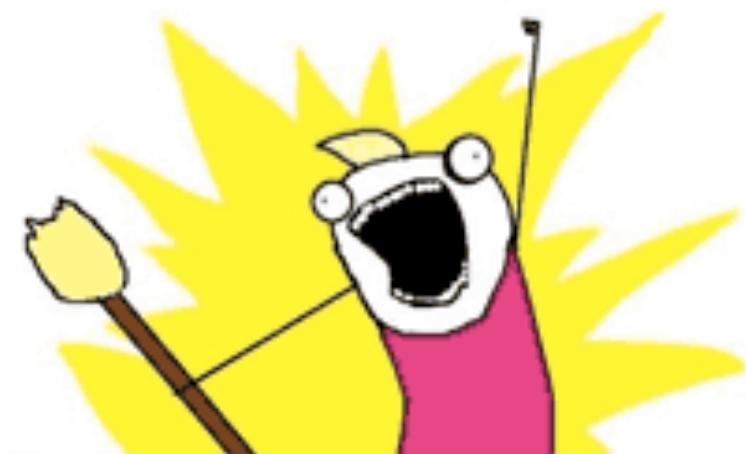
Graphs!

CSC263 Week 8

Announcements

Problem Set 4 is out!

Due Tuesday (Nov 17)



Other Announcements

- Drop date Nov 8
- Final exam schedule is posted
 - CSC263 exam Dec 11, 2-5pm

This week's outline

→Graphs

→BFS

Graph

A really, really important ADT that is used to model **relationships** between objects.

Get that job at Google

Whenever someone gives you a problem, *think graphs*. They are the most fundamental and flexible way of representing any kind of a relationship, so it's about a 50-50 shot that any interesting design problem has a graph involved in it. Make absolutely sure you can't think of a way to solve it using graphs before moving on to other solution types. This tip is important!

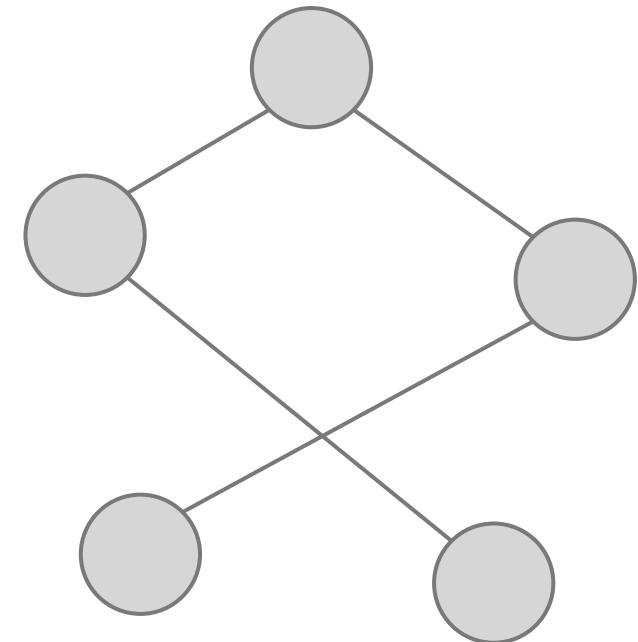
Reference: <http://steve-yegge.blogspot.ca/2008/03/get-that-job-at-google.html>

Things that can be modelled using graphs

- Web
- Facebook
- Task scheduling
- Maps & GPS
- Compiler (garbage collection)
- OCR (computer vision)
- Database
- Rubik's cube
- (many many other things)

Definition

$$G = (V, E)$$

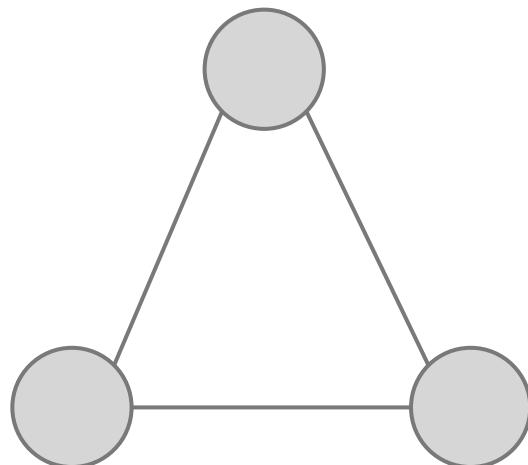


Set of **vertices**
e.g., {a, b, c}

Set of **edges**
e.g., { (a, b), (c, a) }

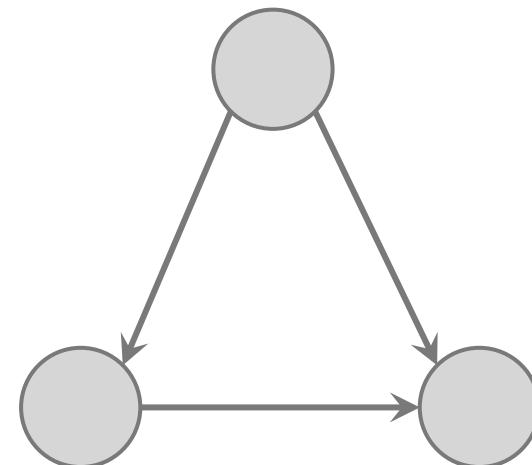
Flavours of graphs

each edge is an
unordered pair
 $(u, v) = (v, u)$

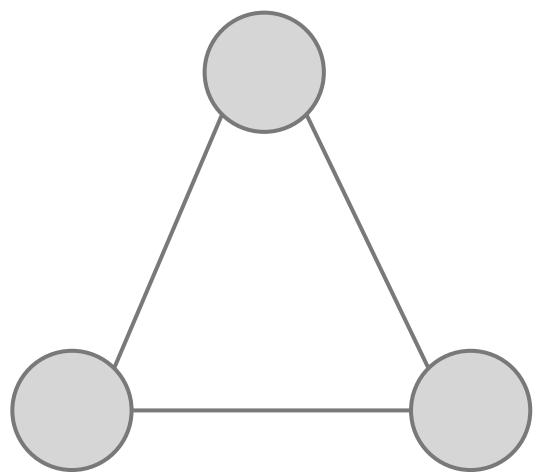


Undirected

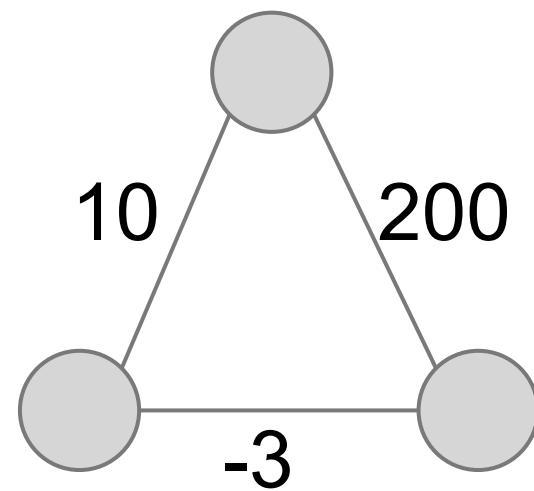
each edge is an
ordered pair
 $(u, v) \neq (v, u)$



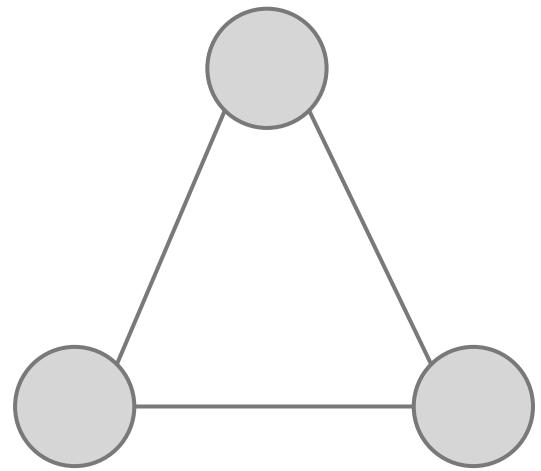
Directed



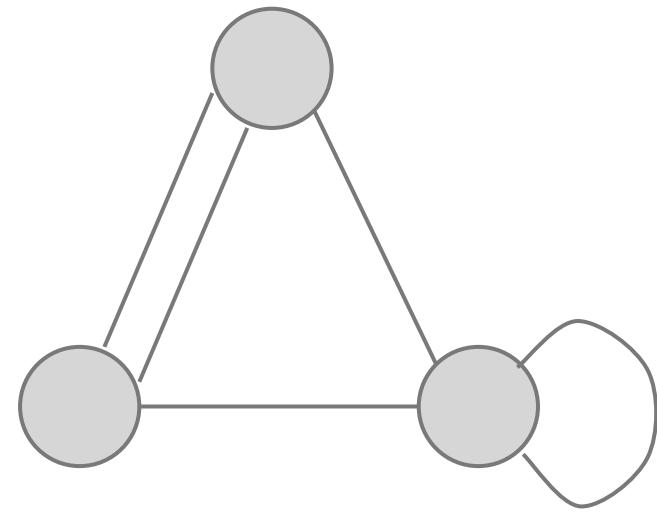
Unweighted



Weighted

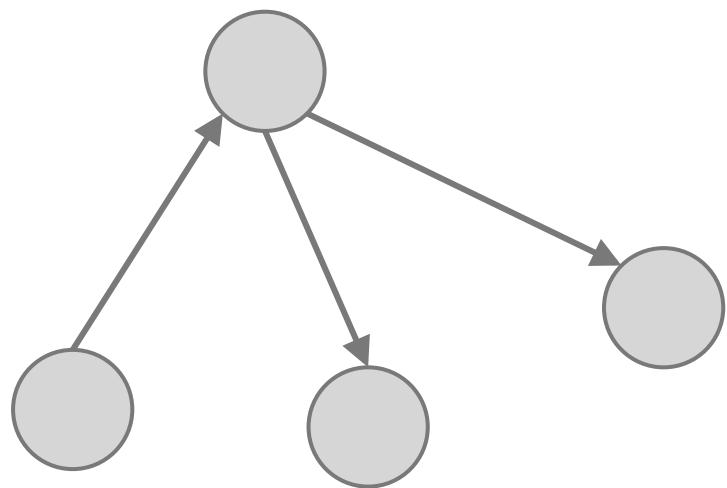


Simple

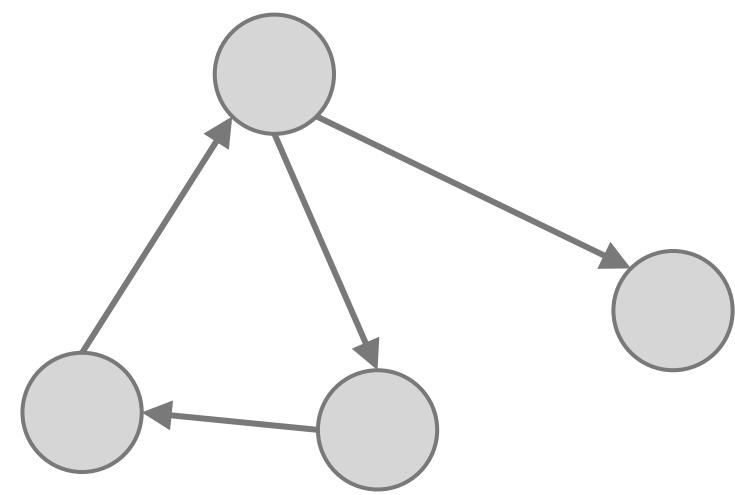


Non-simple

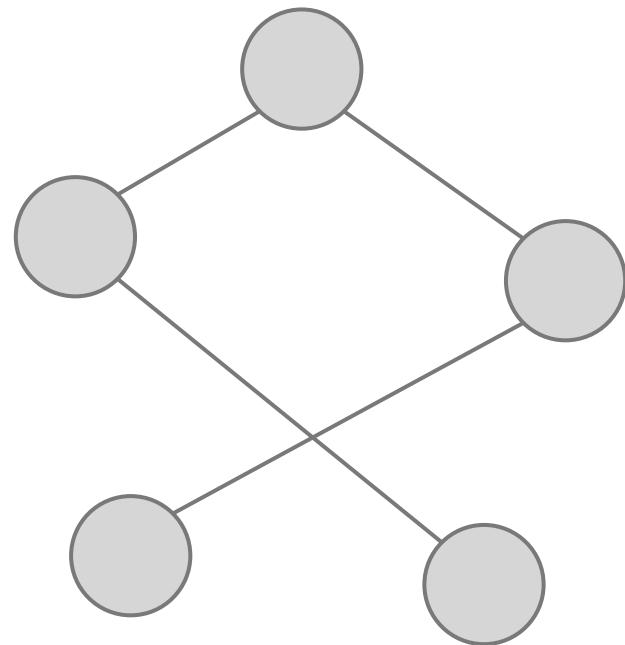
No multiple edge, no self-loop



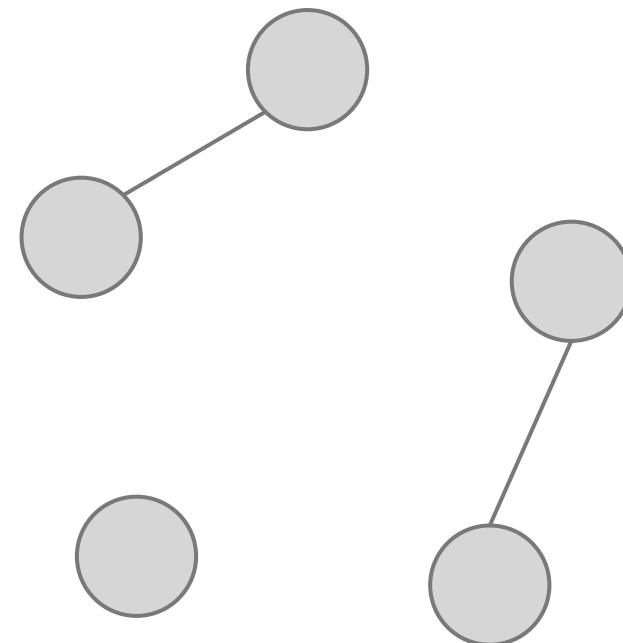
Acyclic



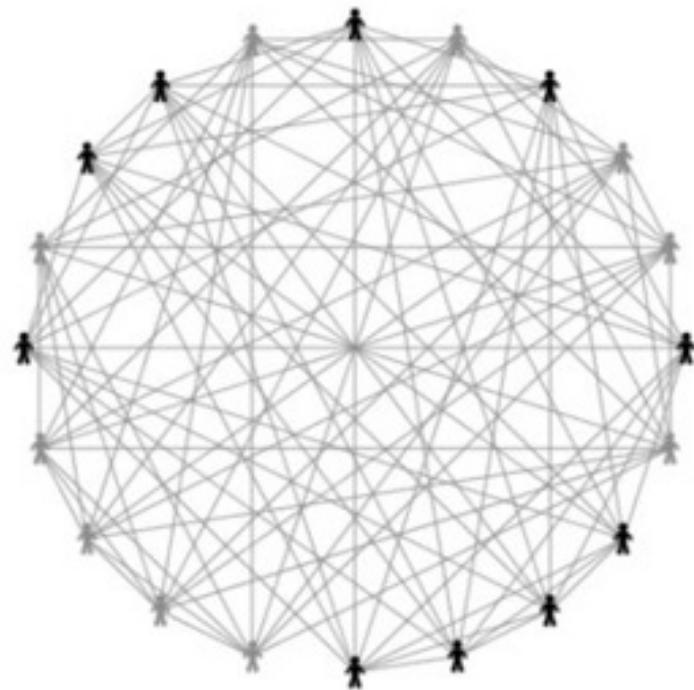
Cyclic



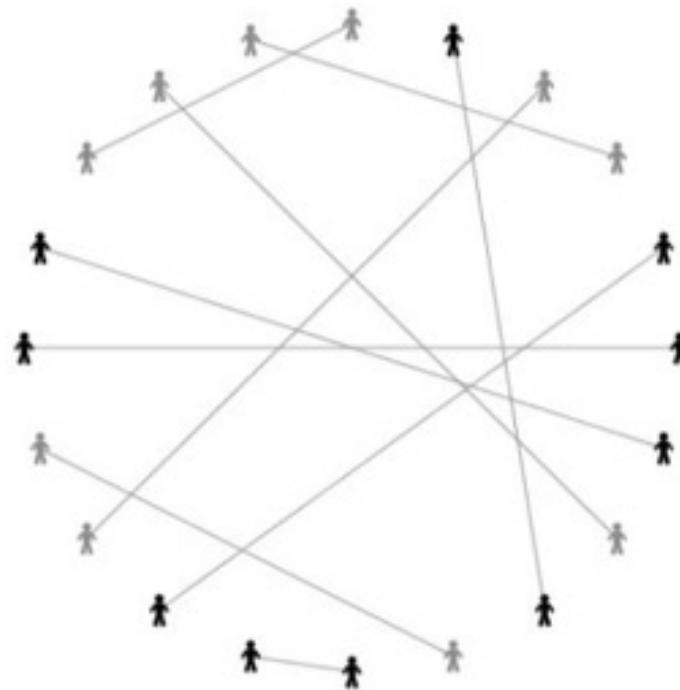
Connected



Disconnected



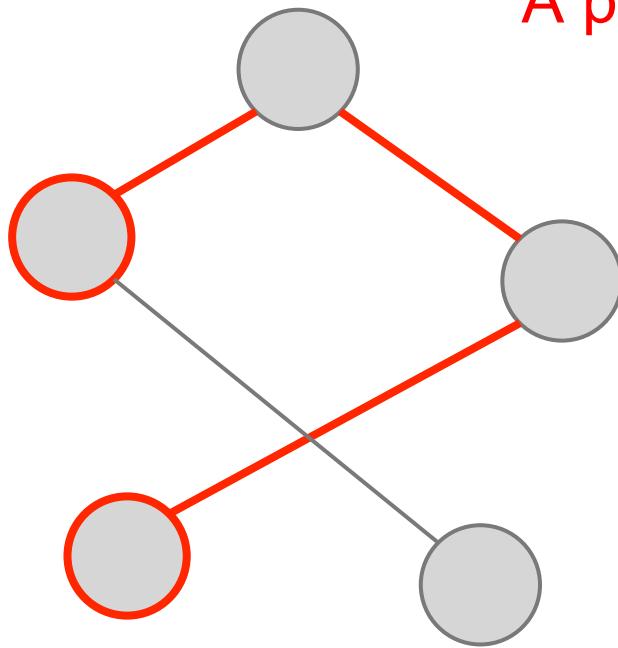
Dense



Sparse

Path

A path of length 3



Length of path = number of edges

Read Appendix B.4 for more background on graphs.

Operations on a graph

- **Add** a vertex; **remove** a vertex
- **Add** an edge; **remove** an edge
- **Get neighbours** (undirected graph)
 - ◆ Neighbourhood(u): all $v \in V$ such that $(u, v) \in E$
- **Get in-neighbours / out-neighbours** (directed graph)
- **Traversal**: visit every vertex in the graph

Many other operations:

→Traversal:

BFS (breadth first search)

DFS (depth first search)

**→Given s,t find a (minimum length) path
from s to t**

**→Given a connected graph G, output a
spanning tree of G**

→Is G connected?

Data structures for the graph ADT

- Adjacency matrix
- Adjacency list

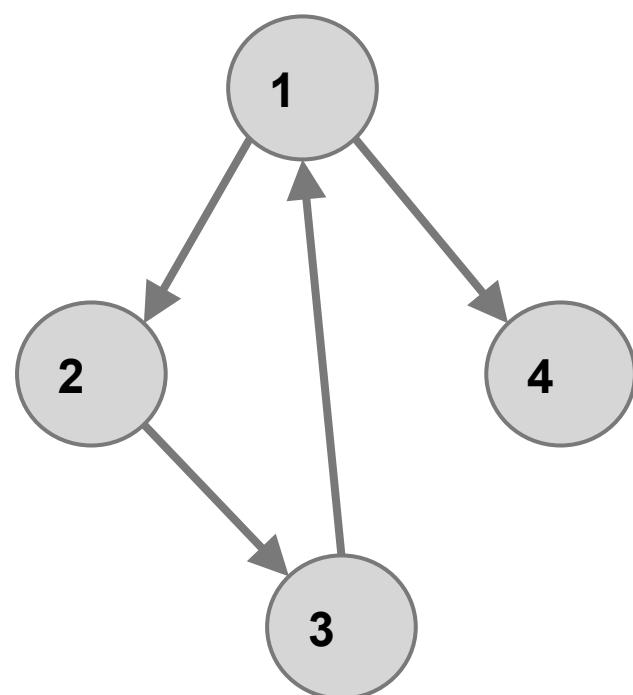
Adjacency matrix

A $|V| \times |V|$ matrix \mathbf{A}

Let $V = \{v_1, v_2, \dots, v_n\}$

$$A[i, j] = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Adjacency matrix



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

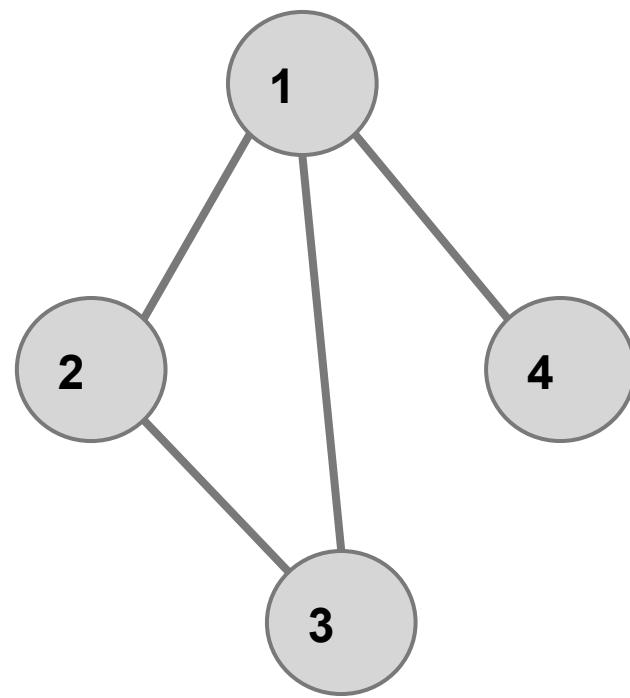
Adjacency matrix

How much space
does it take?

$|V|^2$

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

Adjacency matrix (undirected graph)



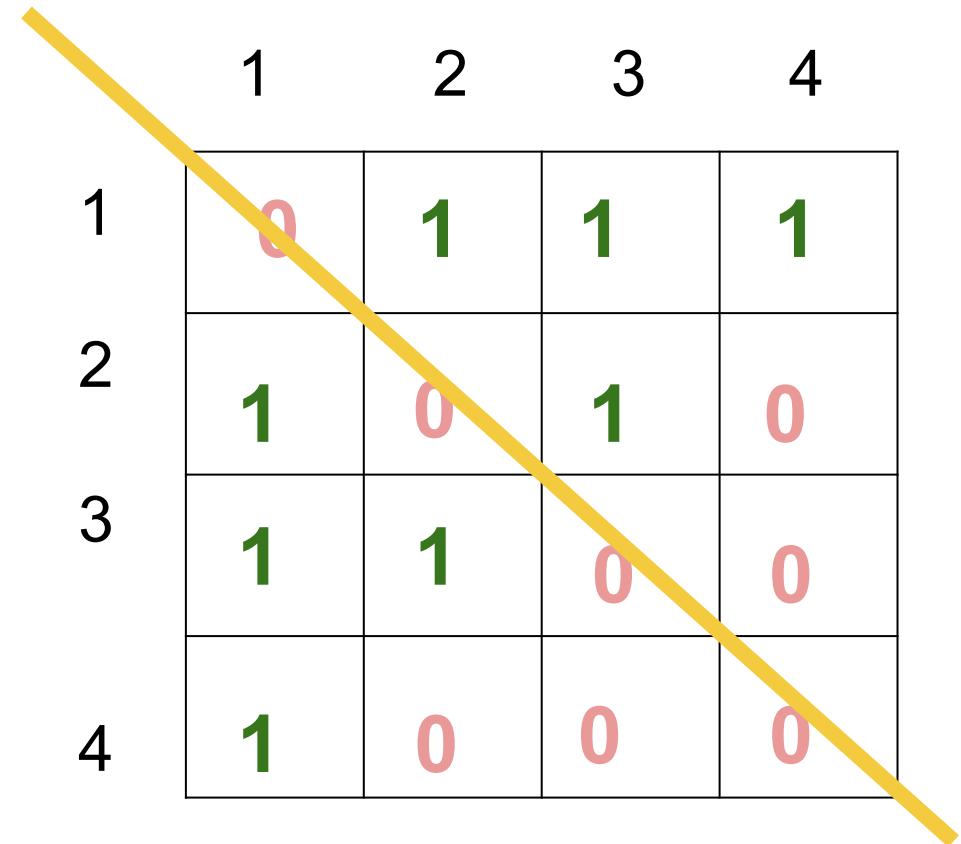
	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	0
4	1	0	0	0

The adjacency matrix of an **undirected** graph is **symmetric**.

Adjacency matrix (undirected graph)

How much space
does it take?

$|V|^2$

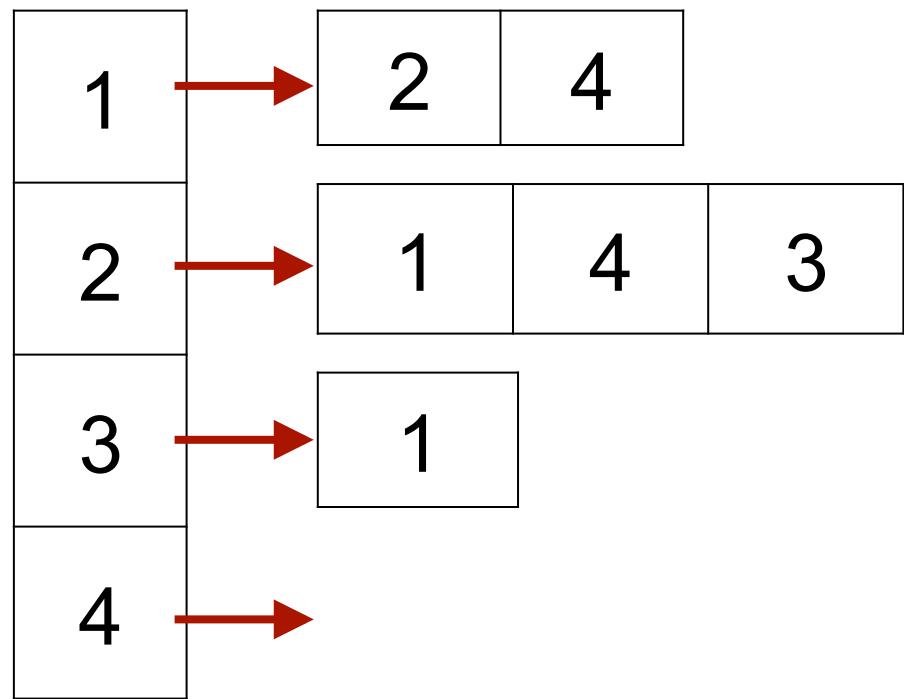
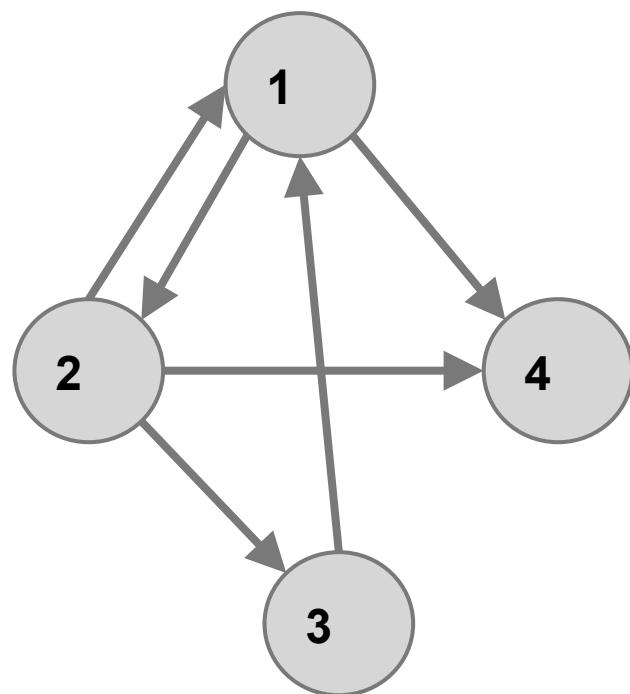


	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	0
4	1	0	0	0

Adjacency list

Adjacency list (directed graph)

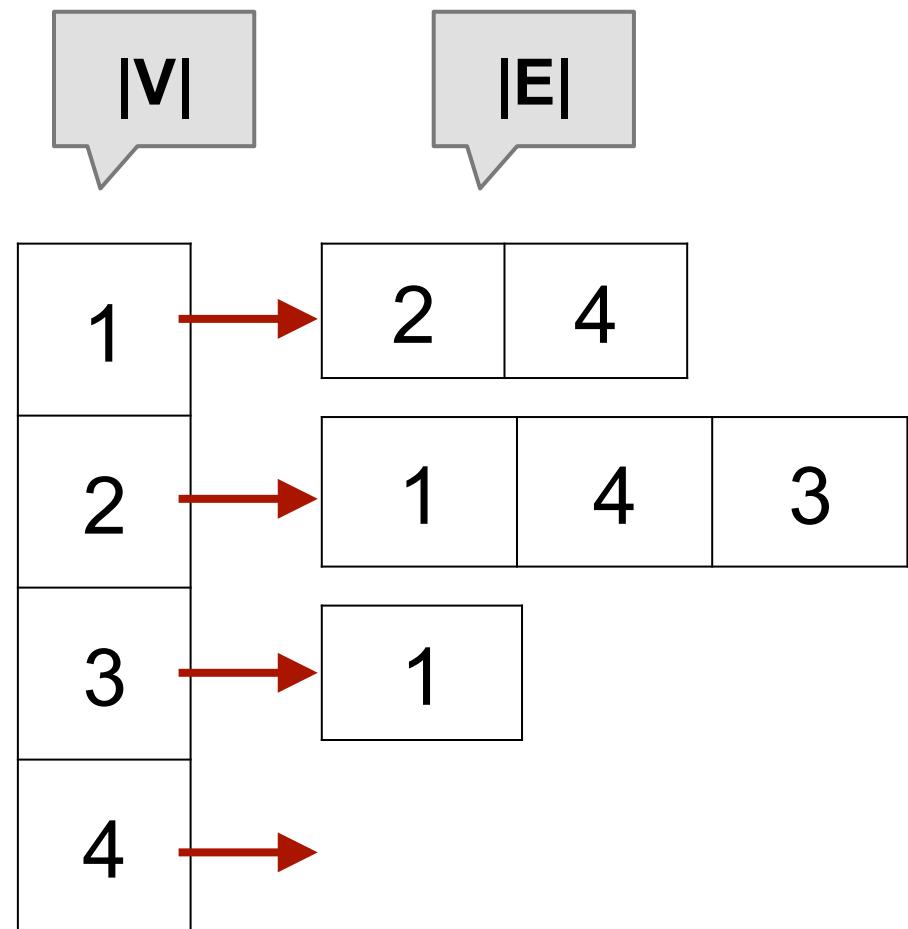
Each vertex v_i stores a list $A[i]$ of v_j that satisfies $(v_i, v_j) \in E$



Adjacency list (directed graph)

How much space
does it take?

$|V| + |E|$

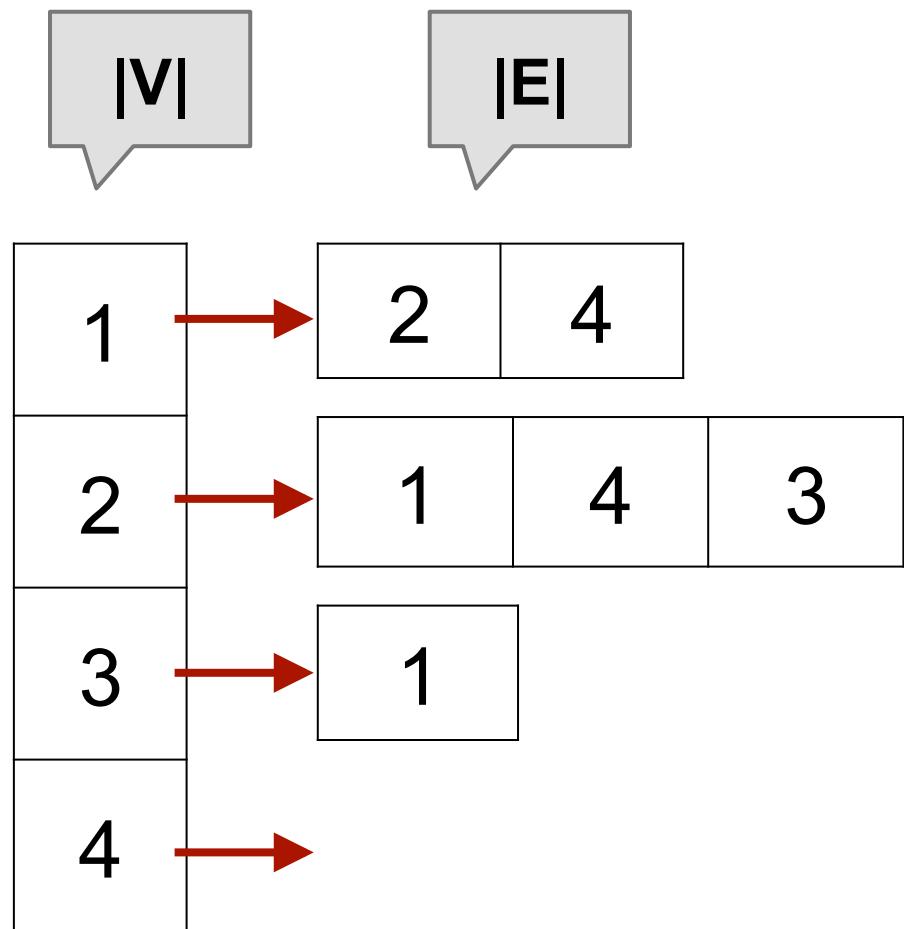


Adjacency list (directed graph)

How much space does it take?

$$|V| + |E|$$

This assumes we can store the name of a vertex in one cell of the linked list. In terms of bits, the size would be more like $|V| + |E| (\log |V|)$

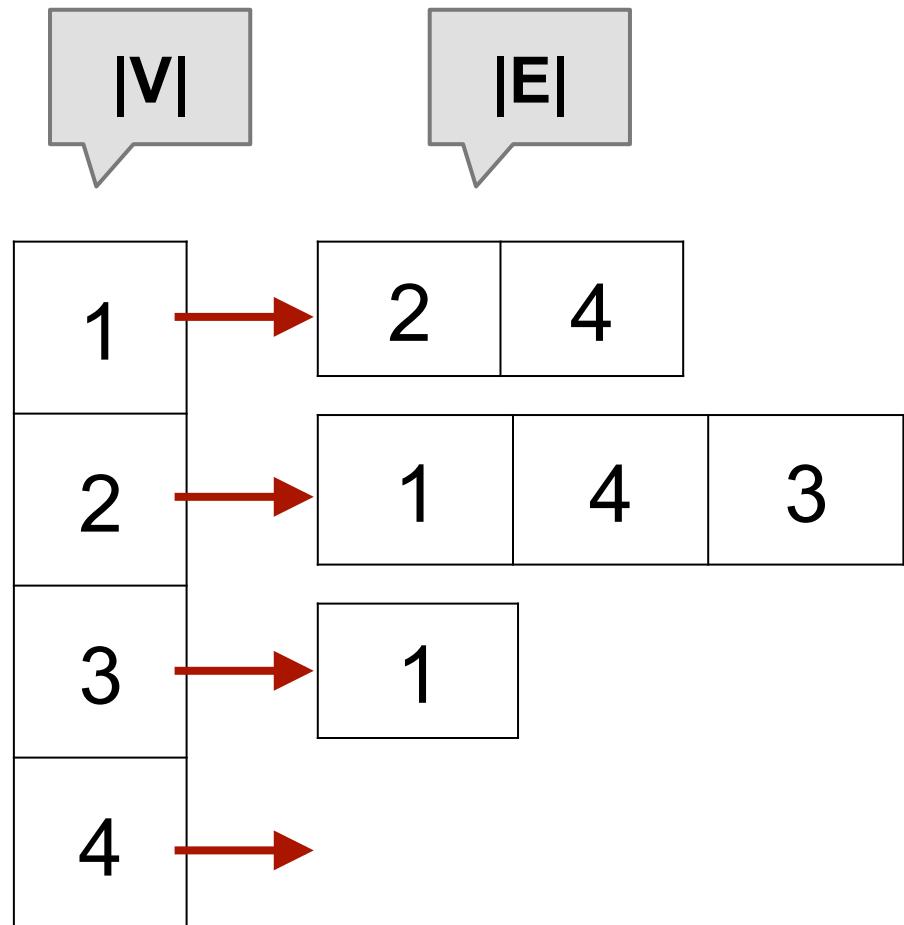


Adjacency list (directed graph)

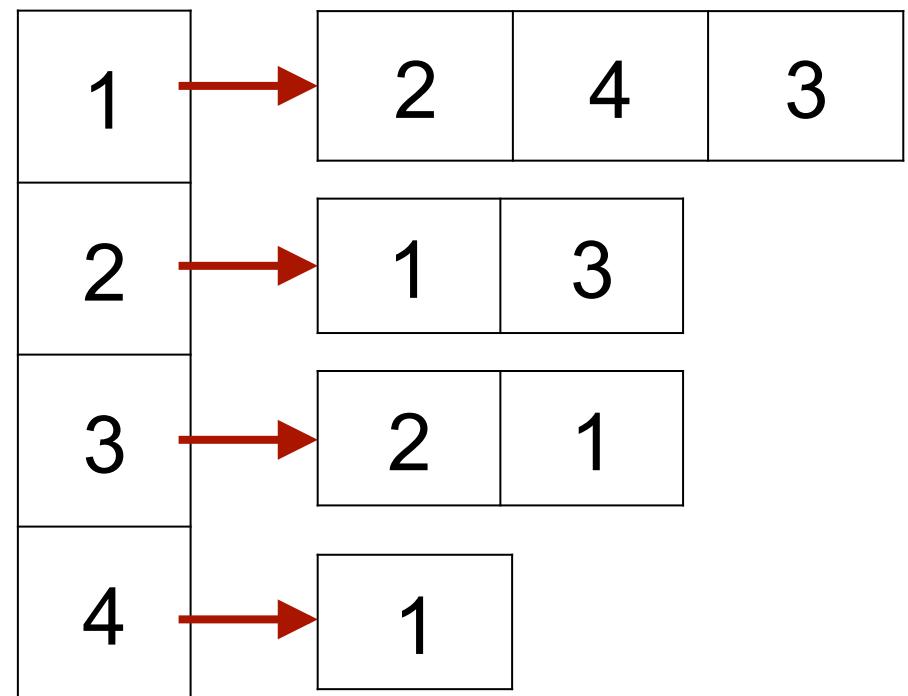
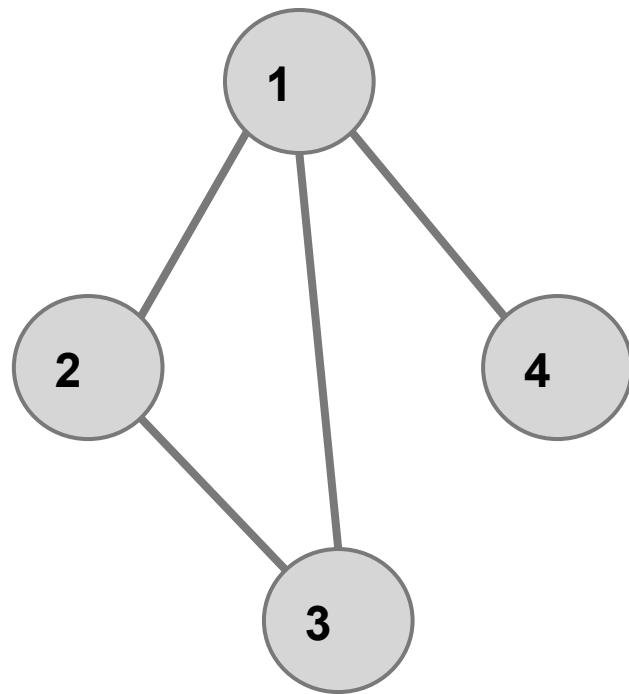
How much space does it take?

$|V| + |E|$

One often ignores these lower order factors of $\log n$.
(Recall for hashing, we assume $h(x)$ in constant time.)



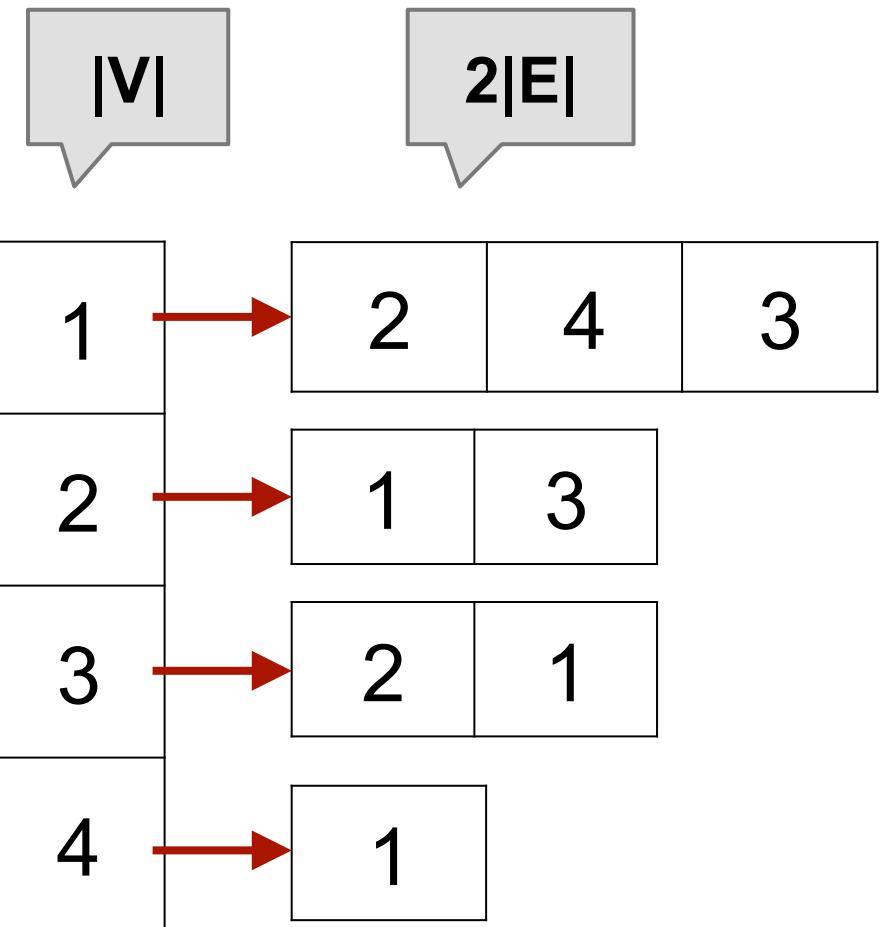
Adjacency list (undirected graph)



Adjacency list (undirected graph)

How much space
does it take?

$|V| + 2|E|$



Matrix *versus* List

In term of space complexity

- adjacency matrix is $\Theta(|V|^2)$
- adjacency list is $\Theta(|V|+|E|)$

Which one is more space-efficient?

Adjacency list, if $|E| \ll |V|^2$, i.e., the graph is not very **dense**.

Matrix **versus** List

Anything that **Matrix** does better than **List**?



Check whether edge (v_i, v_j) is in E

→ **Matrix**: just check if $A[i, j] = 1$, **O(1)**

→ **List**: go through list $A[i]$ see if j is in there,
O(length of list)

Takeaway

Adjacency matrix or adjacency list?

Choose the more appropriate one depending on the problem.

Graph Traversals

BFS and DFS



Graph traversals

Visiting **every vertex once**, starting from a given vertex.

The visits can follow different **orders**, we will learn about the following two ways

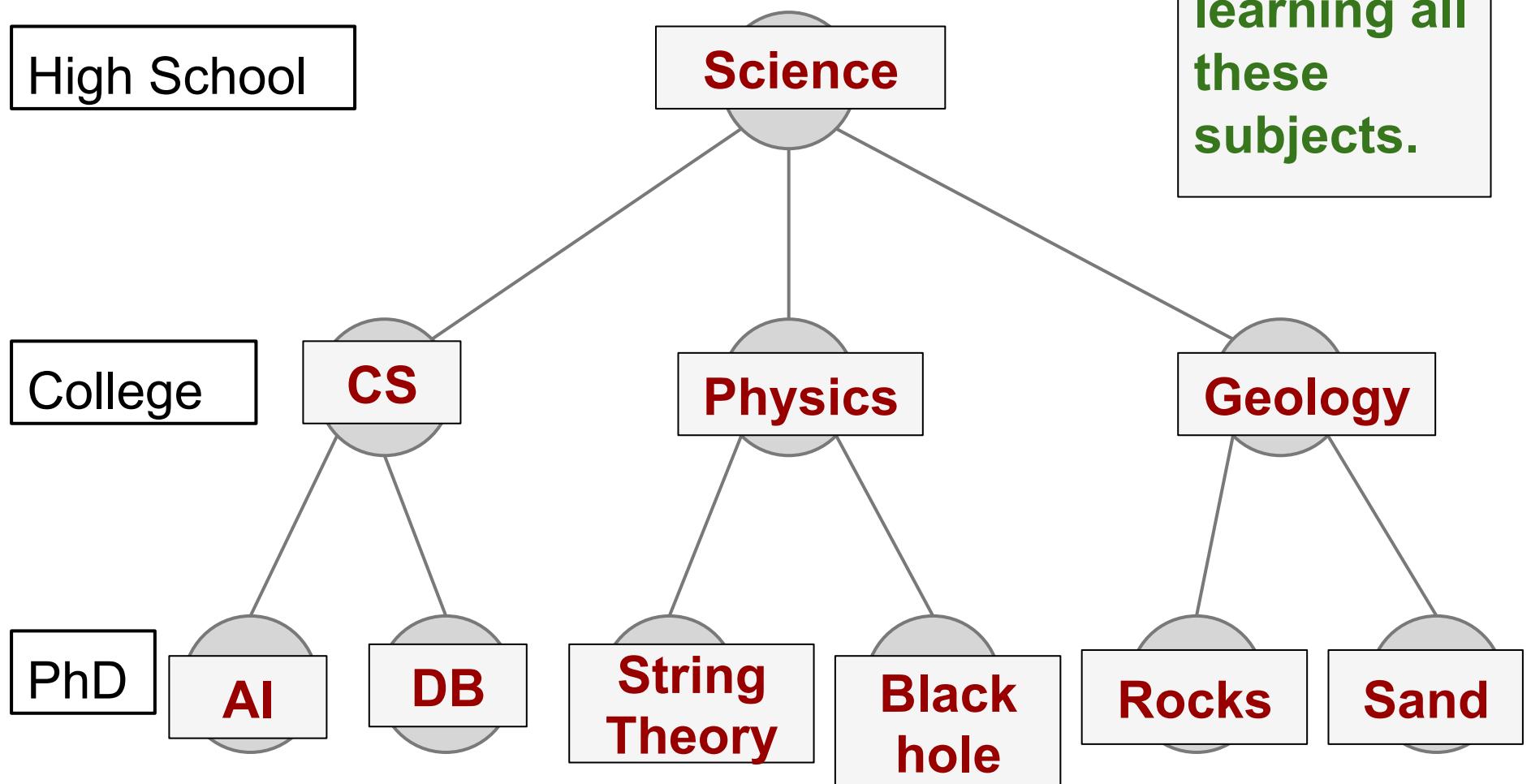
→**Breadth First Search (BFS)**

→**Depth First Search (DFS)**

Intuitions of BFS and DFS

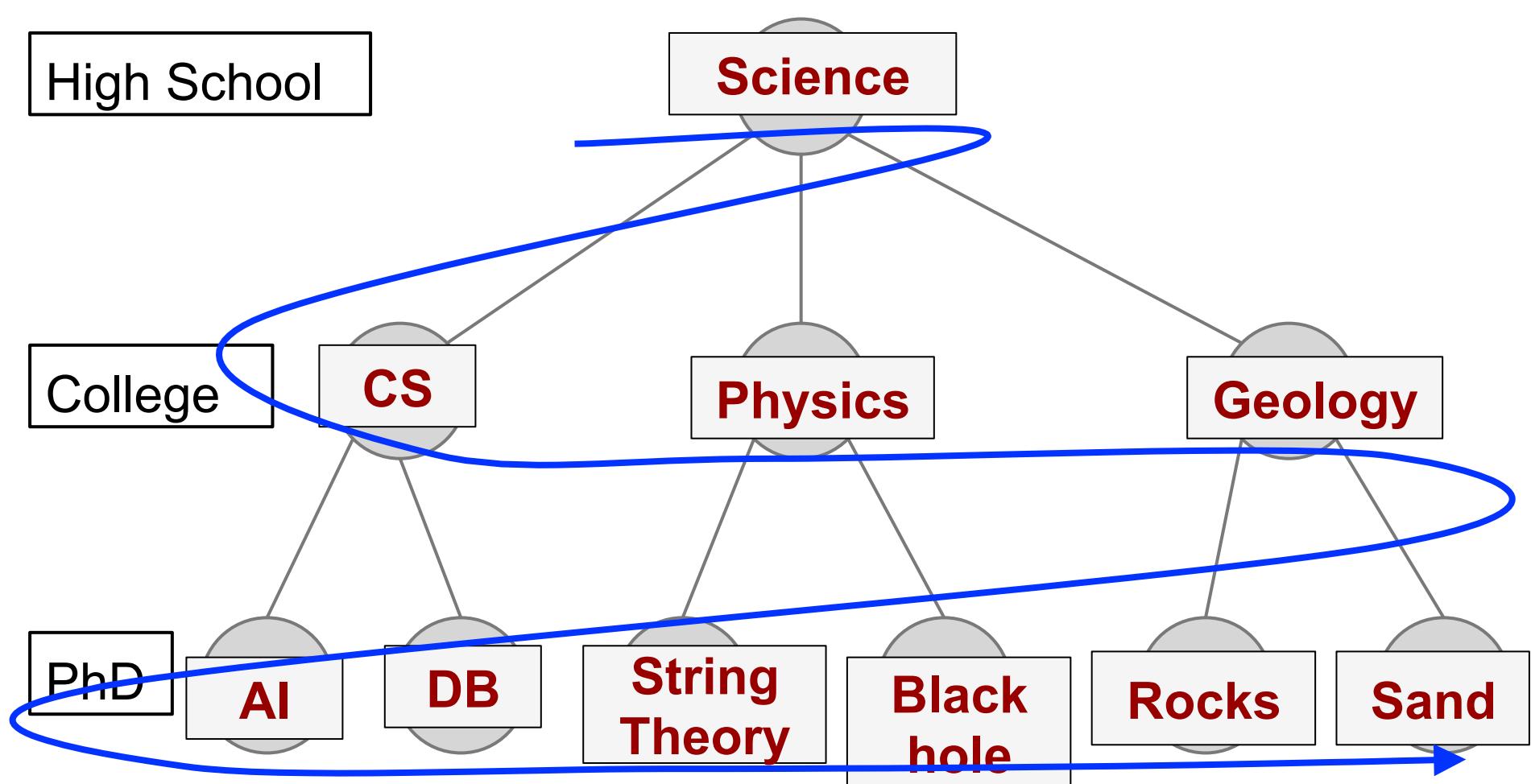
Consider a special graph -- a **tree**

“The knowledge learning tree”



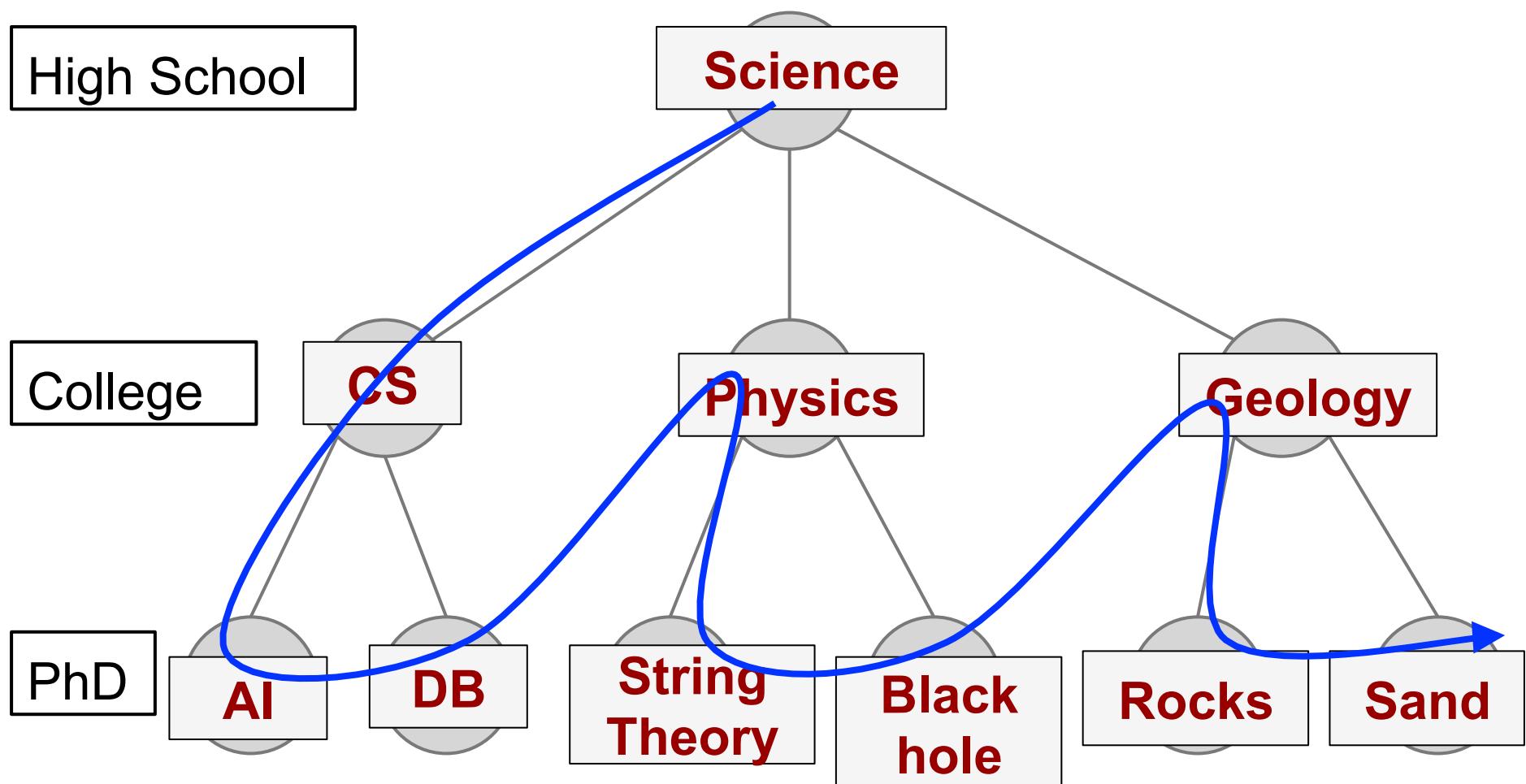
The Breadth-First ways of learning these subjects

→ Level by level, finish high school, then all subjects at College level, then finish all subjects in PhD level.



The Depth-First way of learning these subjects

→ Go towards PhD whenever possible; only start learning physics after finishing everything in CS.



Now let's seriously start
studying **BFS**



Special case: BFS in a tree

Review CSC148:

BFS in a tree (starting from root) is a
level-by-level traversal.

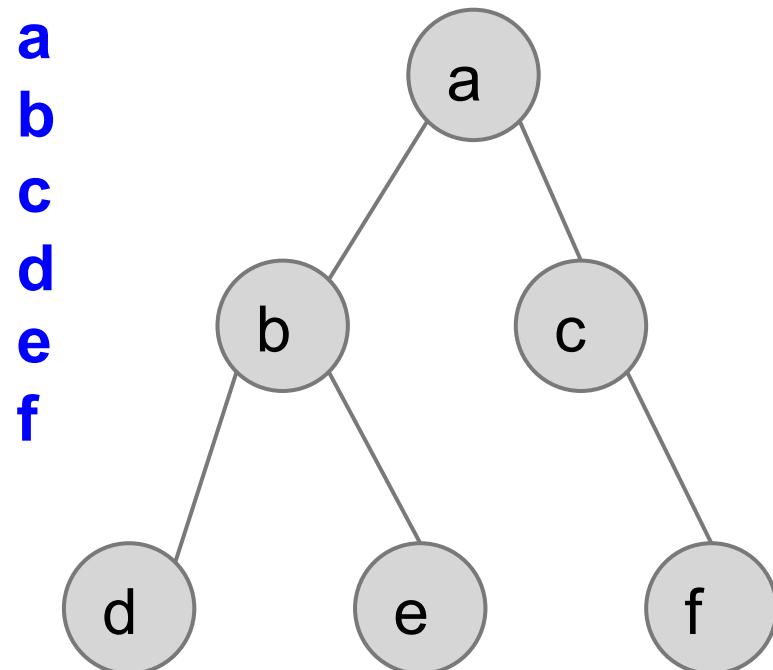
(NOT preorder!)

What ADT did we use for implementing the **level-by-level** traversal?

Queue!

Special case: BFS in a tree

Output:



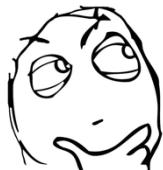
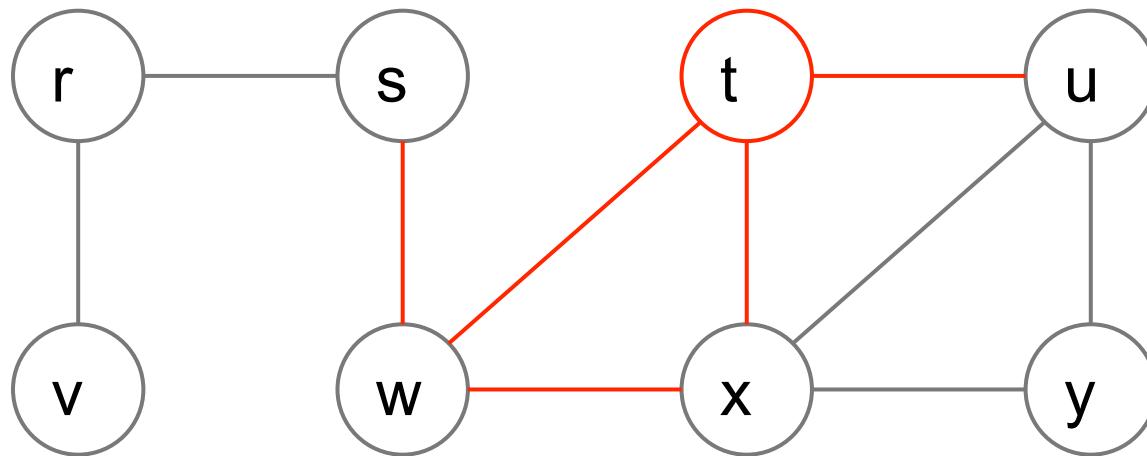
NOT_YET_BFS(root):

```
Q ← Queue()
Enqueue(Q, root)
while Q not empty:
    x ← Dequeue(Q)
    print x
    for each child c of x:
        Enqueue(Q, c)
```

Queue: **EMPTY!**

DQ DQ DQ DQ DQ DQ

The real deal: BFS in a Graph



If we just run **NOT_YET_BFS(t)** on the above graph. What problem would we have?



It would want to visit some vertices more than once (e.g., **x**)

```
NOT_YET_BFS(root):
    Q ← Queue()
    Enqueue(Q, root)
    while Q not empty:
        x ← Dequeue(Q)
        print x
        for each neighbr c of x:
            Enqueue(Q, c)
```

How avoid visiting a vertex twice

Remember you visited it by
labelling it using **colours**.

→ **White**: “unvisited”

→ **Gray**: “encountered”

→ **Black**: “explored”



- Initially all vertices are **white**
- Colour a vertex **gray** the **first** time visiting it
- Colour a vertex **black** when **all** its **neighbours** have been encountered
- Avoid visiting **gray** or **black** vertices
- In the end, all vertices are **black** (sort-of)

Some other values we want to remember during the traversal...

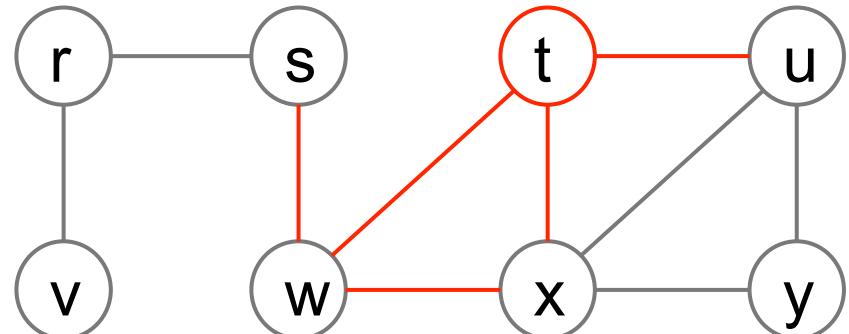
→ **pi[v]**: the vertex from which v is encountered

- ◆ “I was introduced as **whose** neighbour?”

→ **d[v]**: the distance value

- ◆ the distance from v to the source vertex of the BFS

This **d[v]** is going to be **really** useful!



```

BFS(G=(V, E), s):
1   for all v in V:
2       colour[v] ← white
3       d[v] ← ∞
4       pi[v] ← NIL
5   Q ← Queue()
6   colour[s] ← gray
7   d[s] ← 0
8   Enqueue(Q, s)
9   while Q not empty:
10      u ← Dequeue(Q)
11      for each neighbour v of u:
12          if colour[v] = white
13              colour[v] ← gray
14              d[v] ← d[u] + 1
15              pi[v] ← u
16              Enqueue(Q, v)
17      colour[u] ← black

```

Pseudocode: the **real** BFS

Initialize vertices

source s is encountered

distance from s to s is 0

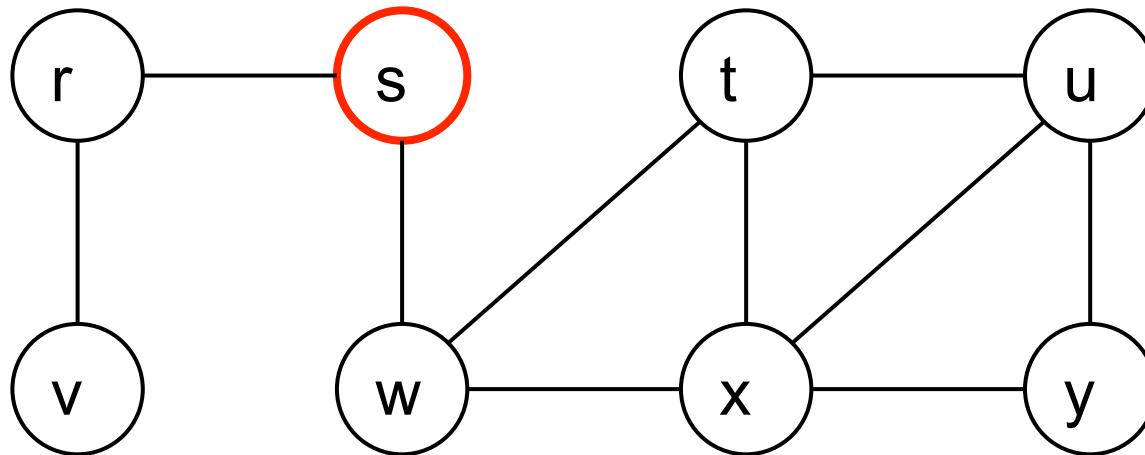
only visit unvisited vertices

v is “1-level” farther from s than u
v is introduced as u’s neighbour

all neighbours of u have been
encountered, therefore u is explored

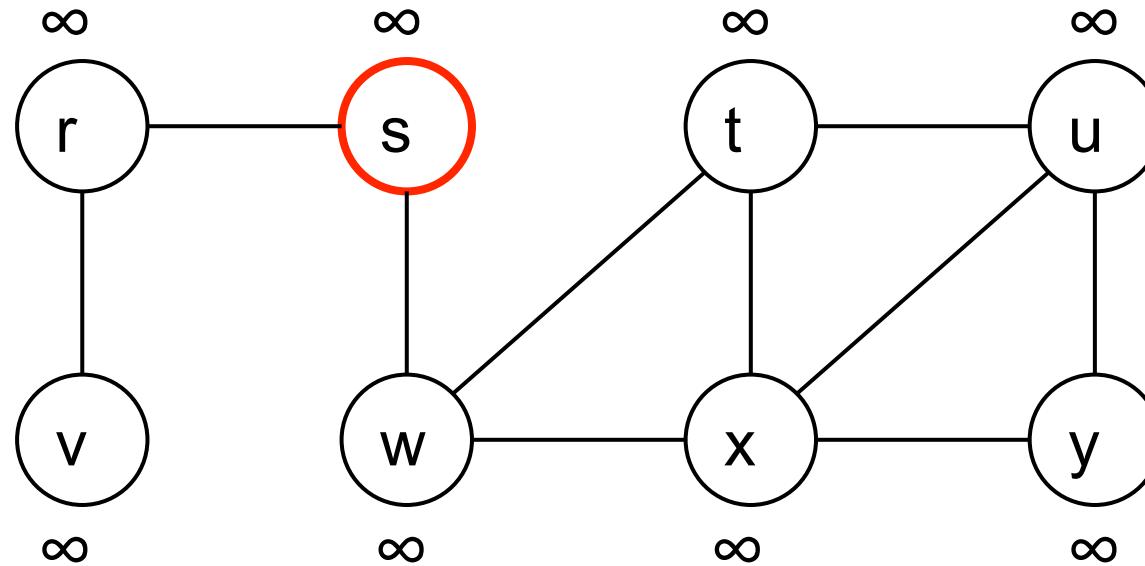
The blue lines are
the same as
NOT_YET_BFS

Let's run an example!



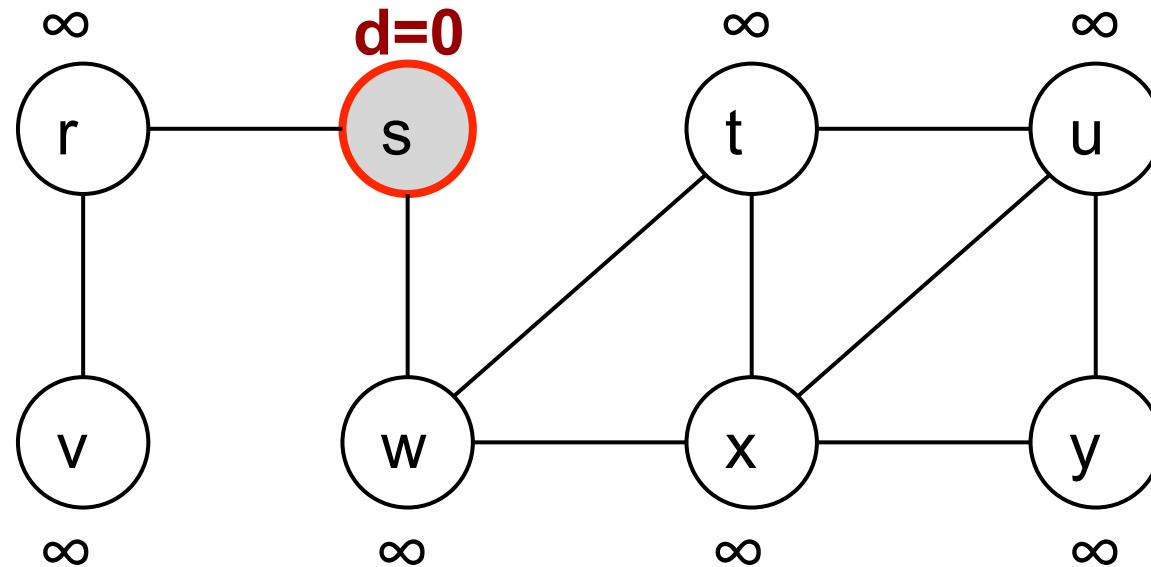
BFS(G, $\textcolor{red}{s}$)

After initialization



All vertices are **white** and have $d = \infty$

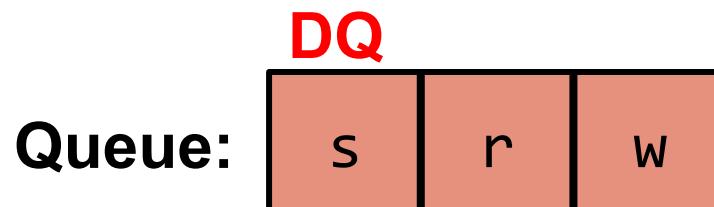
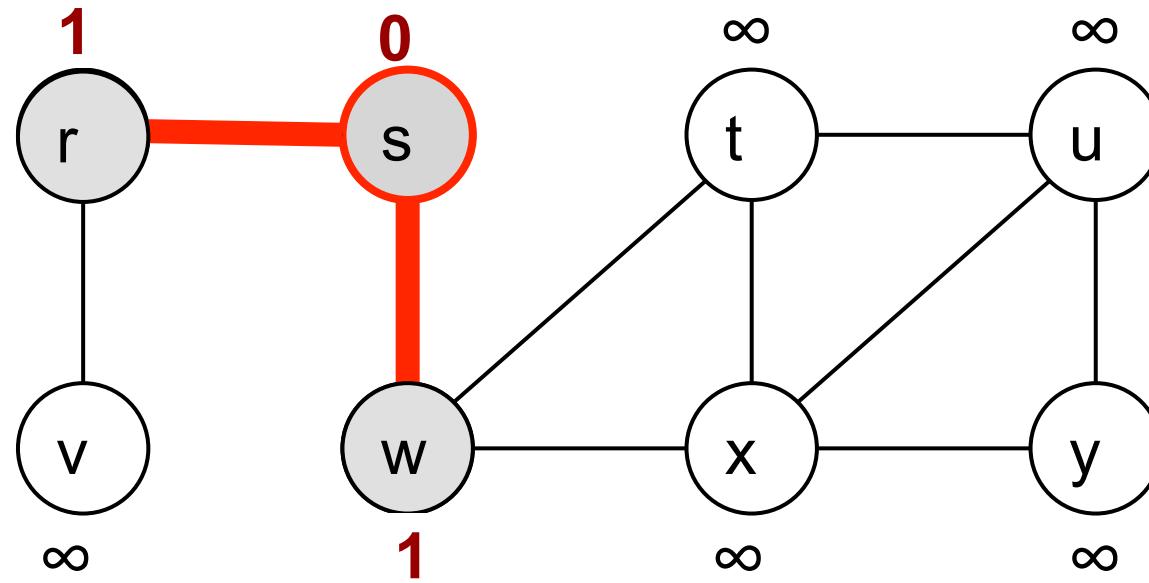
Start by “encountering” the source



Queue: s

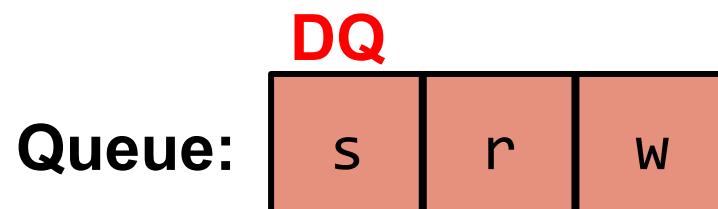
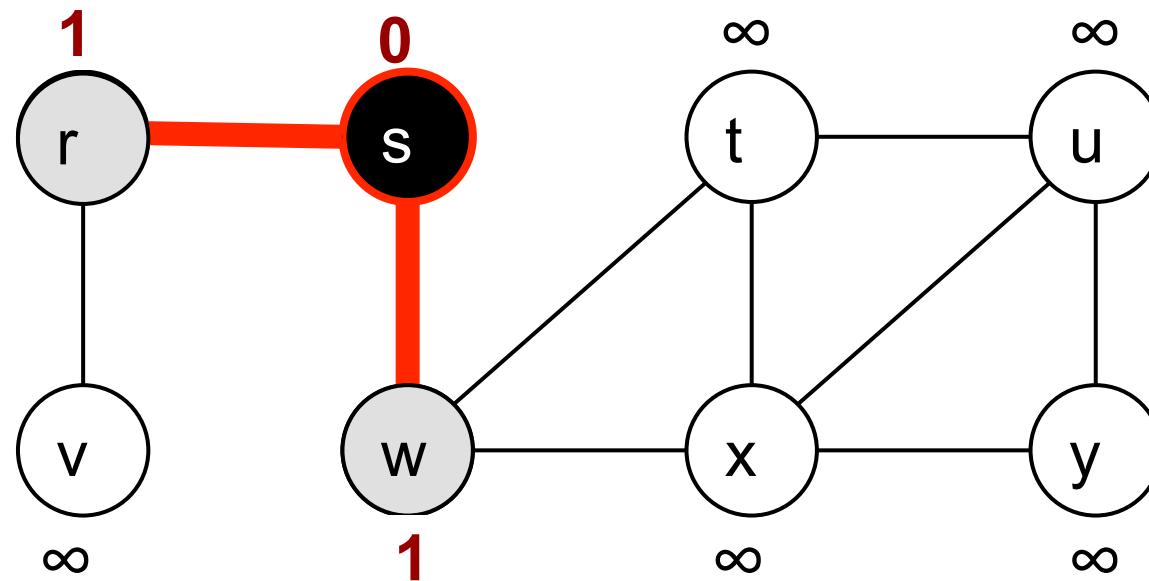
Colour the source **gray** and set its $d = 0$, and Enqueue it

Dequeue, explore neighbours

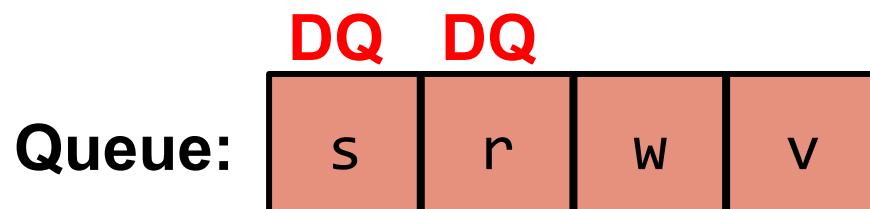
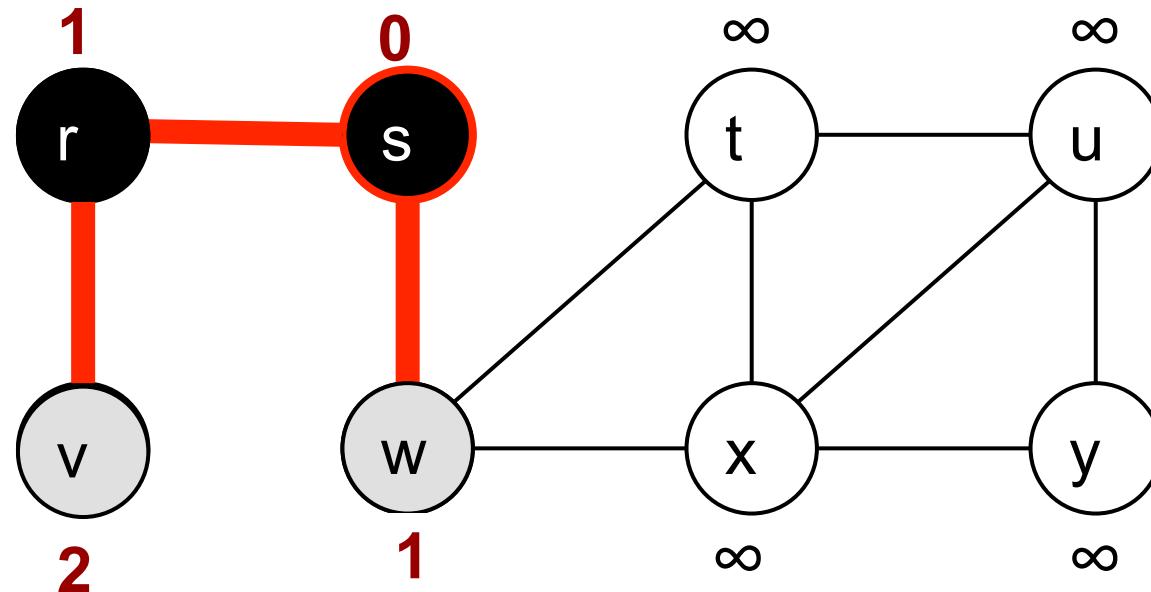


The red edge indicates the $\text{pi}[v]$ that got remembered

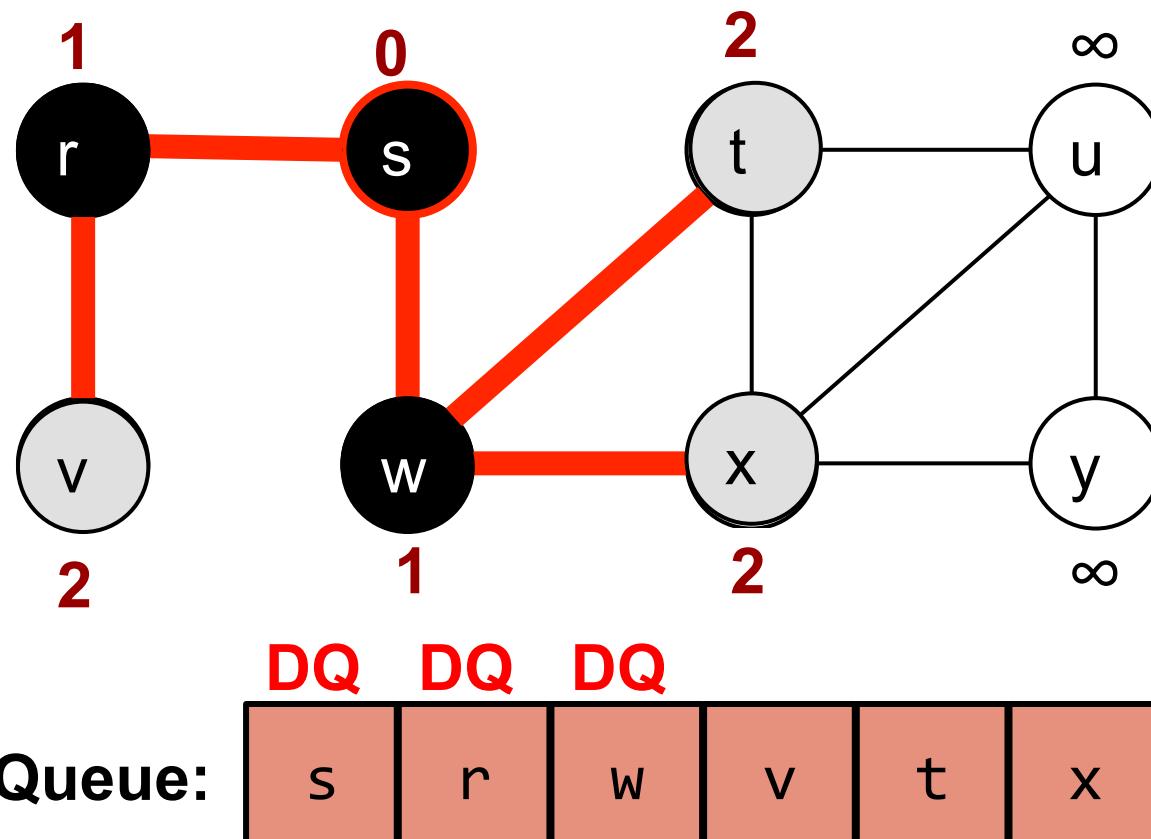
Colour black after exploring all neighbours



Dequeue, explore neighbours (2)



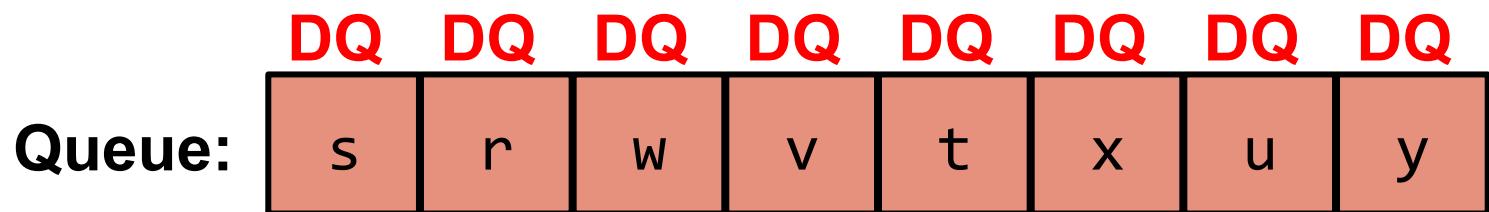
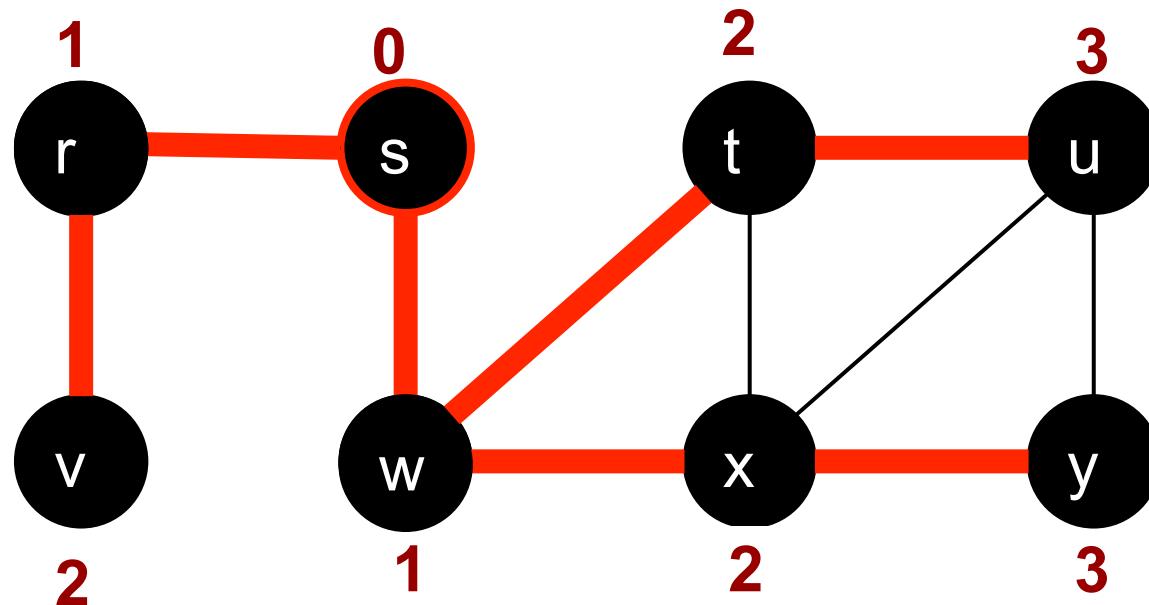
Dequeue, explore neighbours (3)



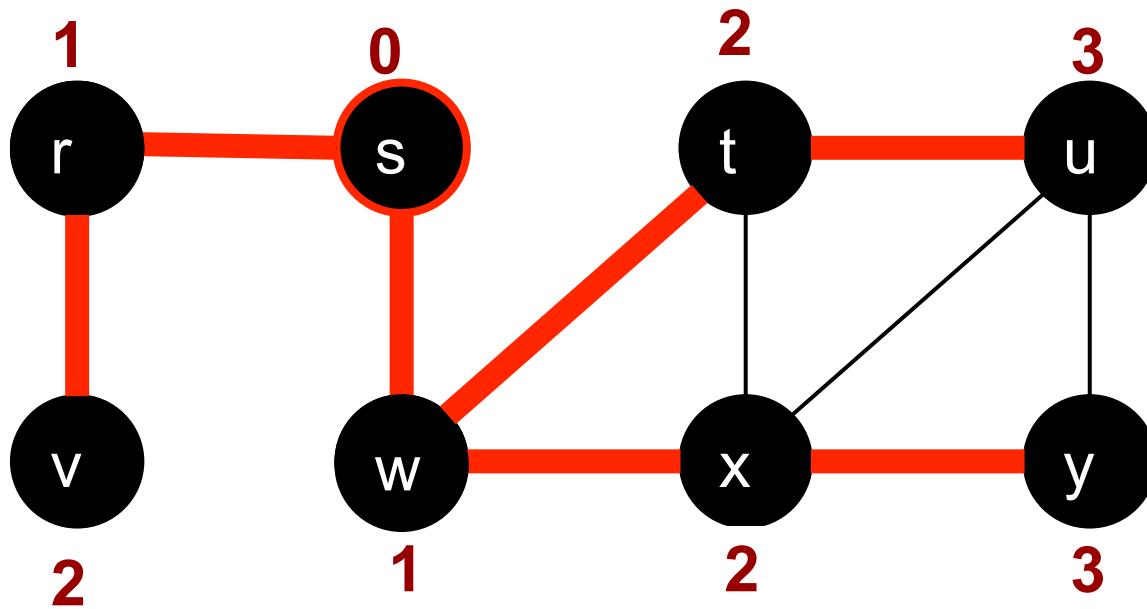
after a few more steps...



BFS done!

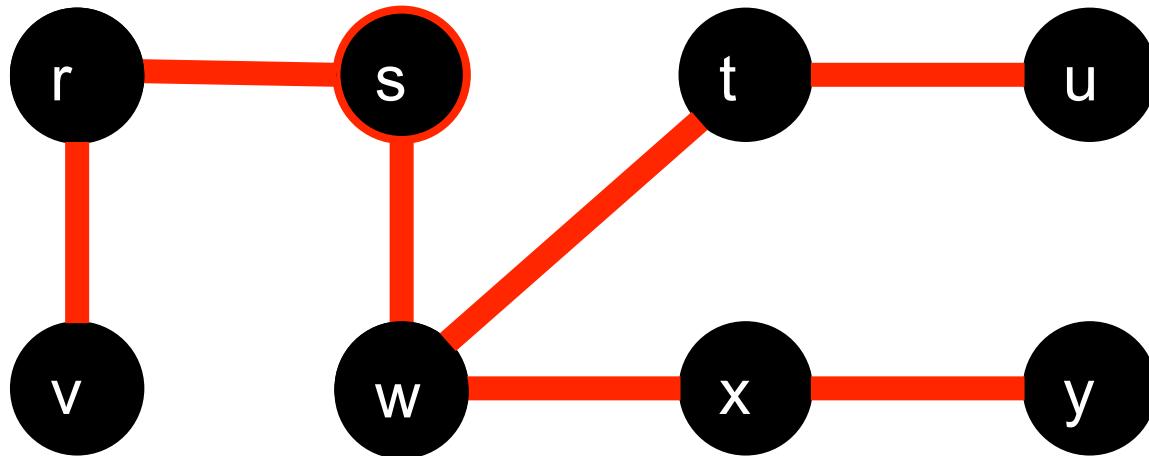


**What do we get after
doing all this?**



First of all, we get to visit **every** vertex **once**.

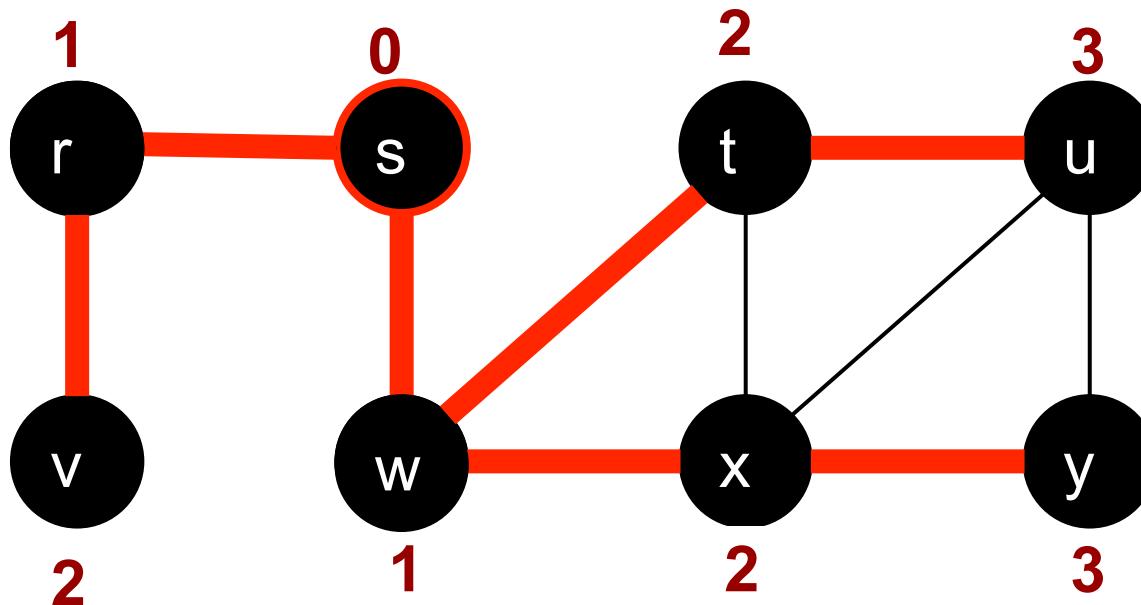
Did you know? The official name of the red edges are called “**tree edges**”.



This is called the **BFS-tree**, it's a **tree** that connects all vertices, if the graph is **connected**.

These $d[v]$ values, we said they were going to be really useful.

Short path from u to s :
 $u \rightarrow \pi[u] \rightarrow \pi[\pi[u]] \rightarrow \pi[\pi[\pi[u]]] \rightarrow \dots \rightarrow s$

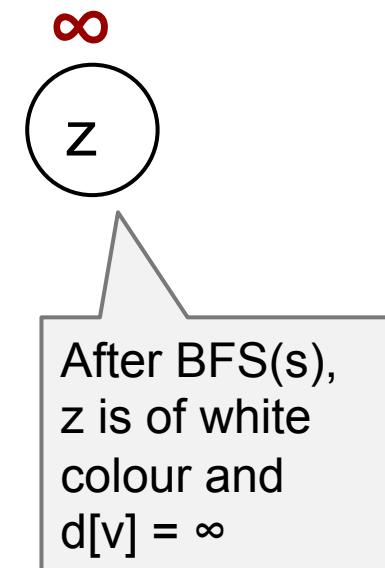
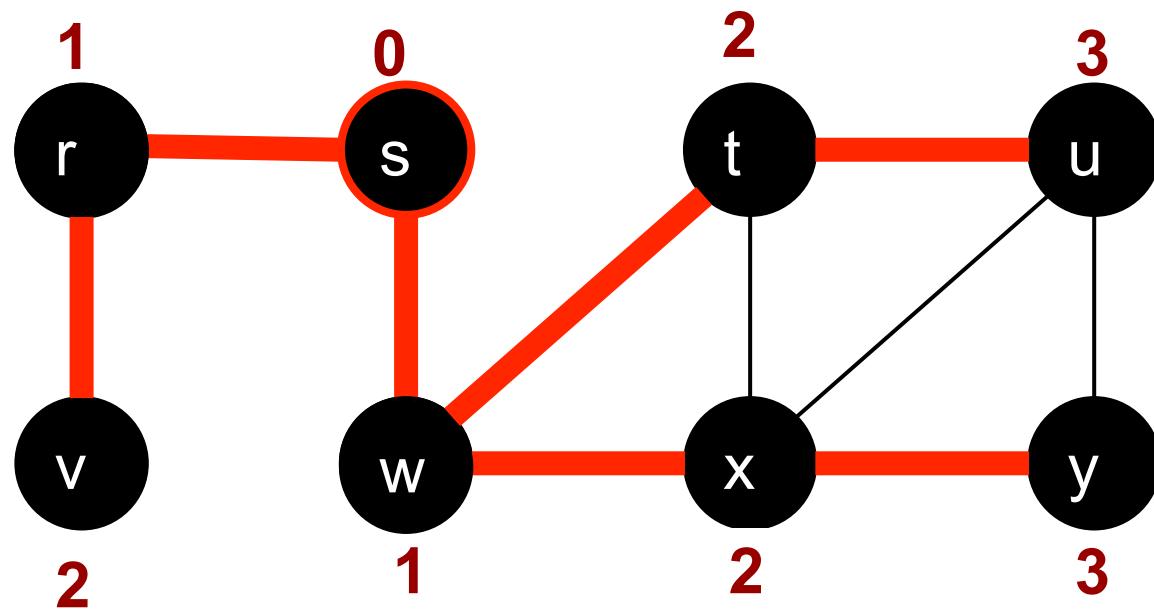


The value indicates the vertex's **distance** from the source vertex.

Actually more than that, it's the **shortest-path distance**, we can prove it.

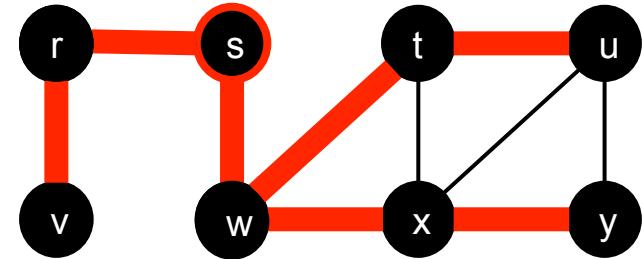
How about finding **short path** itself?
Follow the red edges, $\pi[v]$ comes in handy for this.

What if G is disconnected?



The infinite distance value of z indicates that it is **unreachable** from the source vertex.

Runtime analysis!



The total amount of work (use **adjacency list**):

- Visit each vertex once
 - ◆ Enqueue, Dequeue, change colours, assign $d[v]$, ..., constant work per vertex
 - ◆ in total: $O(|V|)$
- At each vertex, check all its neighbours (all its **incident edges**)
 - ◆ Each edge is checked **twice** (by the two end vertices)
 - ◆ in total: $O(|E|)$

Total runtime:
 $O(|V|+|E|)$

Summary of BFS

- Explores **breadth** rather than **depth**
- Useful for getting **single-source shortest paths** on **unweighted** graphs
- Useful for testing reachability
- Runtime $O(|V|+|E|)$ with adjacency **list** (with adjacency **matrix** it'll be different)

Next week

DFS



BFS



CSC263 Week 9

Announcements

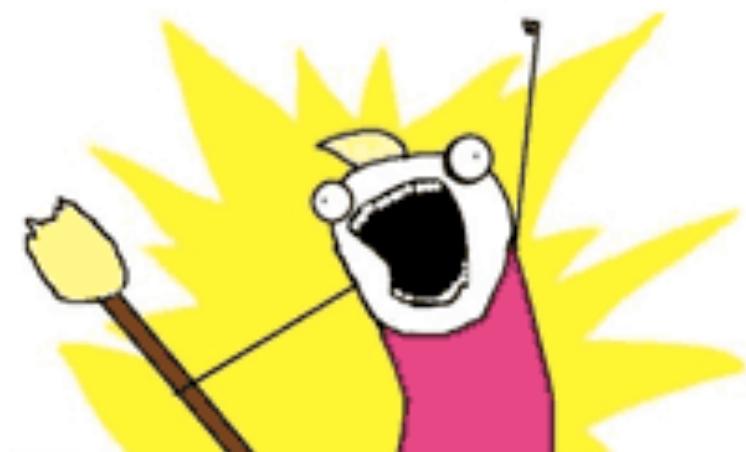
HW3 is graded. Average is 81%



Announcements

Problem Set 4 is due this Tuesday!

Due Tuesday (Nov 17)



Recap

→ The Graph ADT

- ◆ definition and data structures

→ BFS

- ◆ gives us single-source **shortest** path
- ◆ Let $\delta(s, v)$ denote the length of shortest path from s to v ...
- ◆ then after performing a BFS starting from s , we have, for all vertices v

$$d[v] = \delta(s, v)$$

We can prove it.

Idea of the proof

There is no way $d[v] < \delta(s, v)$, according to Lemma 22.2

Use contradiction: suppose there exist v s.t. $d[v] > \delta(s, v)$, let v be the one with the **minimum** $\delta(s, v)$.

Then on a shortest path between s and v , pick vertex u which is immediately before v ...

then we have $d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1$

Must be equal because u is on the shortest path from s to v .

Must be equal because v is the minimum $\delta(s, v)$ that violates $d[v] > \delta(s, v)$, so u must not be violating.

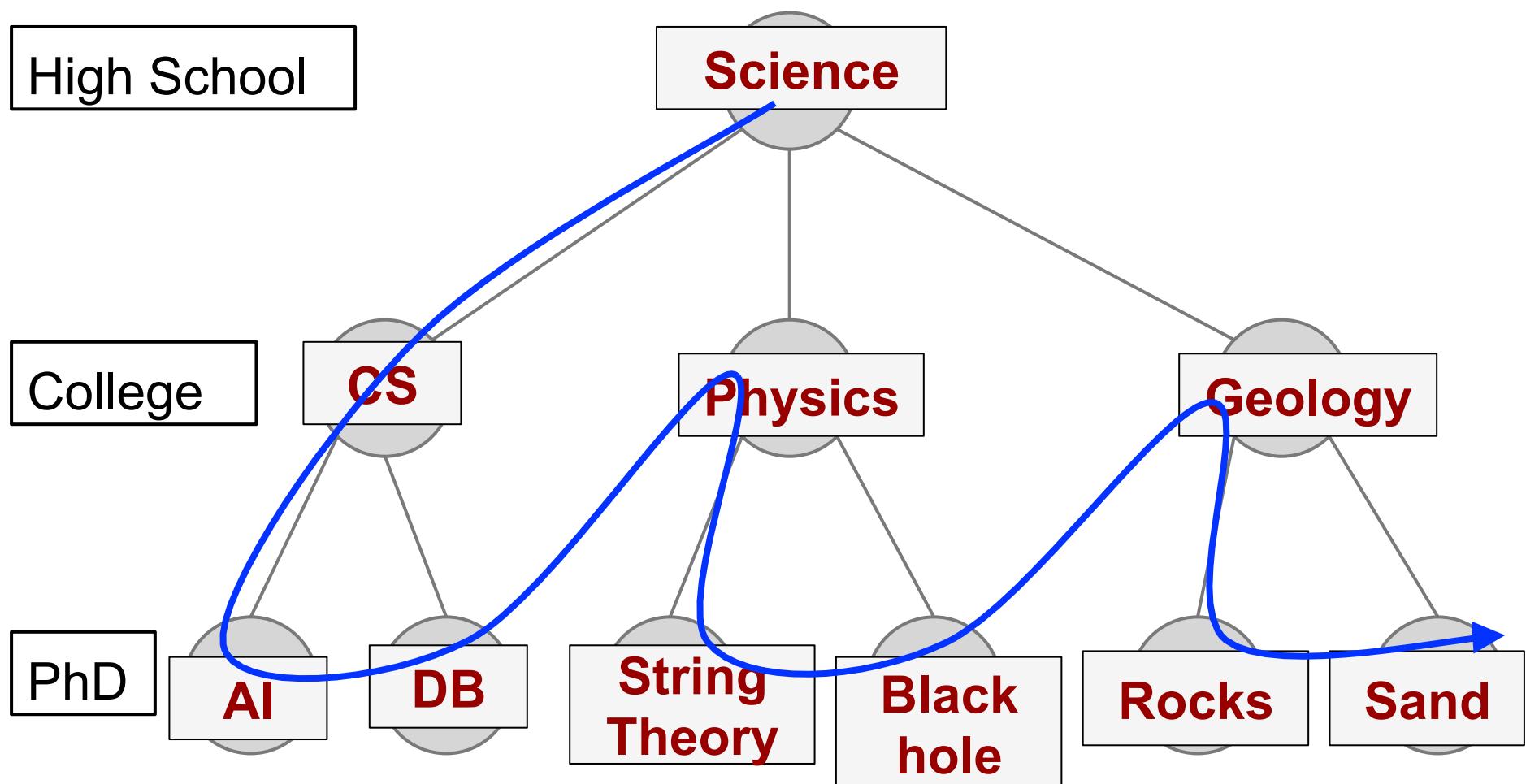
Think about the moment after dequeue u (checking u 's neighbours)

- if v is white, $d[v] = d[u] + 1$ (how BFS works), **contradiction!**
- if v is black, $d[v] \leq d[u]$ (coz v is dequeued before u), **contradiction!**
- if v is gray, then it is coloured gray by some other vertex w , then $d[v] = d[w] + 1$ and $d[w] \leq d[u]$, therefore $d[v] \leq d[u] + 1$, **contradiction!**

Depth-First Search

The Depth-First way of learning these subjects

→ Go towards PhD whenever possible; only start learning physics after finishing everything in CS.



DFS



BFS



```
NOT_YET_DFS(root):
```

```
Q ← Stack()
```

```
Push(Q, root)
```

```
while Q not empty:
```

```
    x ← Pop(Q)
```

```
    print x
```

```
    for each child c of x:
```

```
        Push(Q, c)
```

```
NOT_YET_BFS(root):
```

```
Q ← Queue()
```

```
Enqueue(Q, root)
```

```
while Q not empty:
```

```
    x ← Dequeue(Q)
```

```
    print x
```

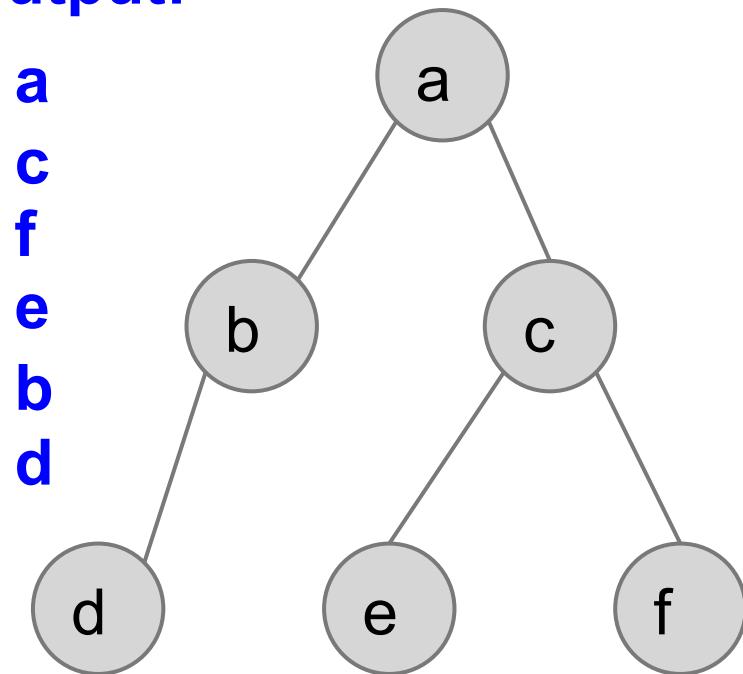
```
    for each child c of x:
```

```
        Enqueue(Q, c)
```

**Why they are
twins!**

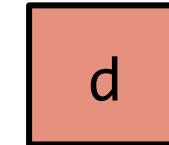
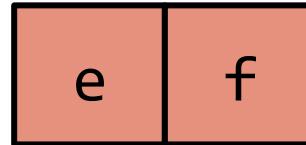
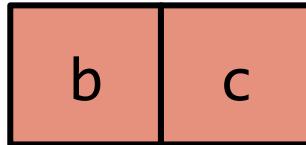
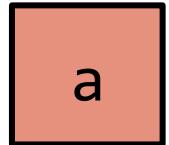
DFS in a tree

Output:



```
NOT_YET_DFS(root):  
    Q ← Stack()  
    Push(Q, root)  
    while Q not empty:  
        x ← Pop(Q)  
        print x  
        for each child c of x:  
            Push(Q, c)
```

Stack:



POP

POP POP

POP POP POP

A nicer way to write this code?

The use of stack is basically implementing **recursion**.

```
NOT_YET_DFS(root):  
    Q ← Stack()  
    Push(Q, root)  
    while Q not empty:  
        x ← Pop(Q)  
        print x  
        for each child c of x:  
            Push(Q, c)
```

```
NOT_YET_DFS(root):  
    print root  
    for each child c of x:  
        NOT_YET_DFS(c)
```

Exercise: Try this code on the tree in the previous slide.

Avoid visiting a vertex twice, **same as BFS**

Remember you visited it by **labelling** it using **colours**.

→ **White**: “unvisited”

→ **Gray**: “encountered”

→ **Black**: “explored”



- Initially all vertices are **white**
- Colour a vertex **gray** the **first** time visiting it
- Colour a vertex **black** when **all** its **neighbours** have been encountered
- Avoid visiting **gray** or **black** vertices
- In the end, all vertices are **black**

Other values to remember, some are same as BFS

- $\text{pi}[v]$: the vertex from which v is encountered
 - ◆ “I was introduced as **whose** neighbour?”

Other values to remember, different from BFS

- There is a **clock** ticking, incremented whenever someone's colour is changed
- For each vertex v , remember two **timesteps**
 - ◆ **$d[v]$** : “discovery time”, when the vertex is first encountered
 - ◆ **$f[v]$** : “finishing time”, when all the vertex's neighbours have been visited.

Note : this $d[v]$ is totally different from that distance value $d[v]$ in BFS!

The pseudo-code (incomplete)

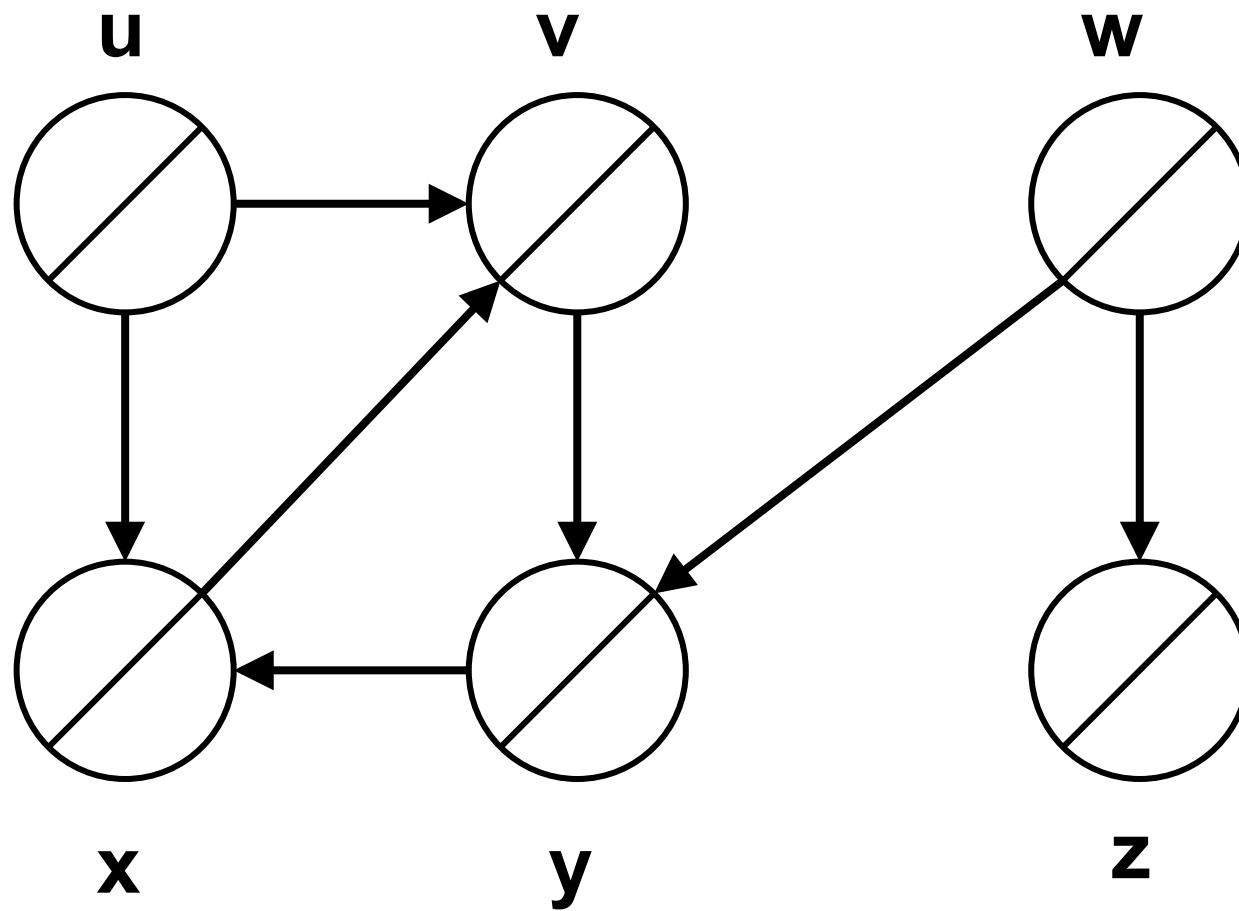
```
DFS_VISIT(G, u):
    colour[u] ← gray
    time ← time + 1
    d[u] ← time      # keep discovery time
                      on first encounter
    for each neighbour v of u:
        if colour[v] = white:
            pi[v] ← u
            DFS_VISIT(G, v)
    colour[u] ← black
    time ← time + 1
    f[u] ← time      # keep finishing time after
                      exploring all neighbours
```

The red part is
the same as
`NOT_YET_DFS`

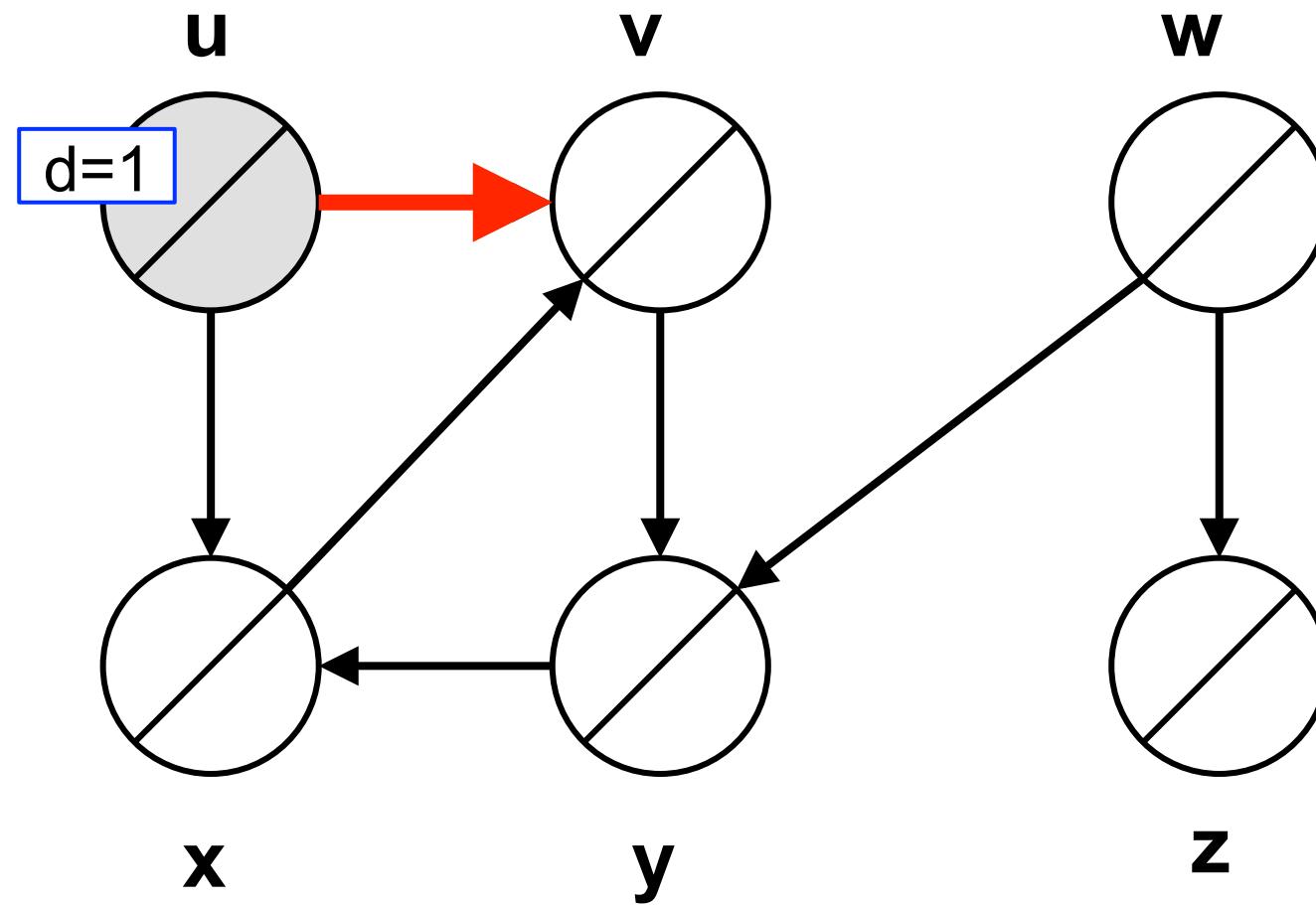
Why **DFS_VISIT**
instead of **DFS**?
We will see...

Let's run an example!

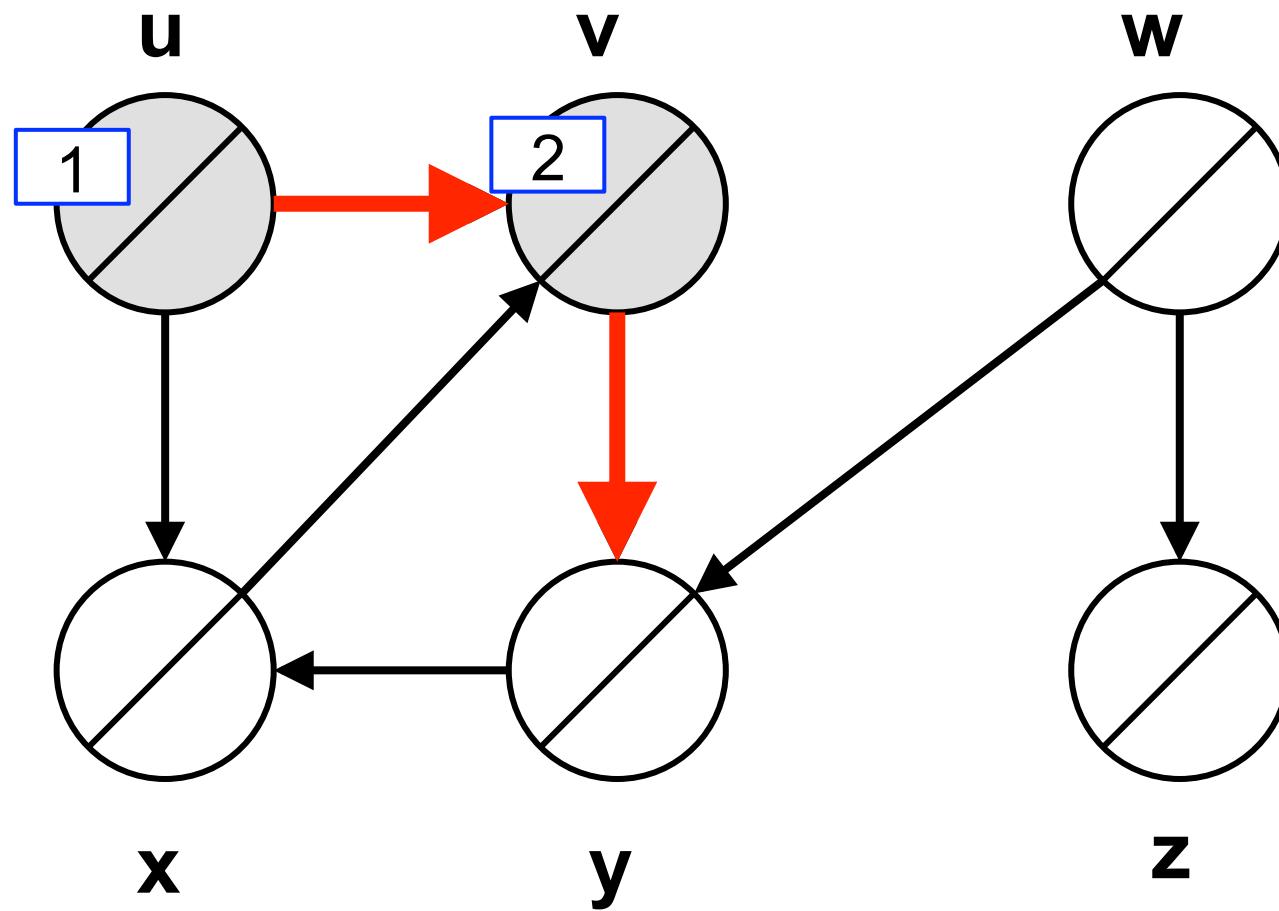
DFS_VISIT(G, u)



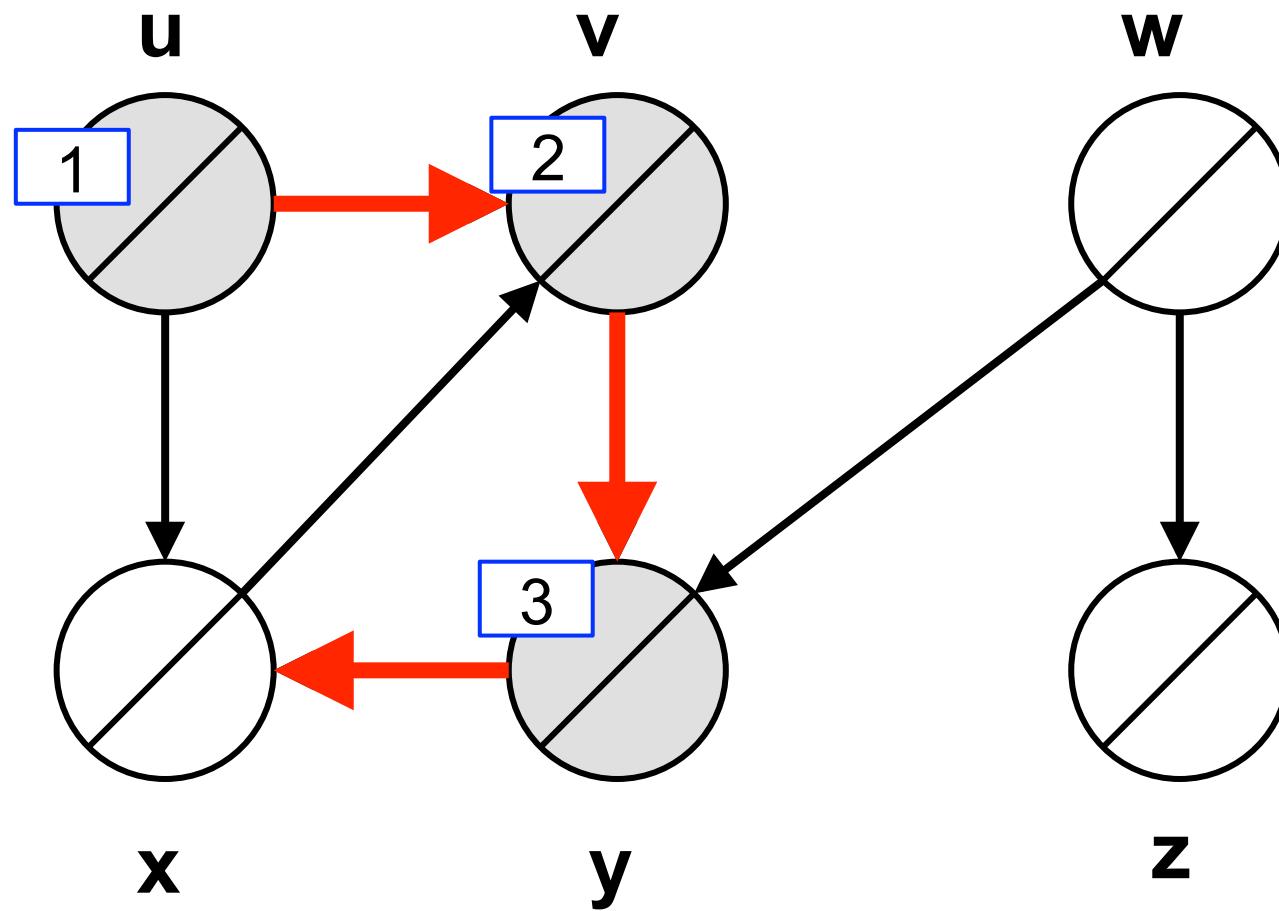
time = 1, encounter the source vertex



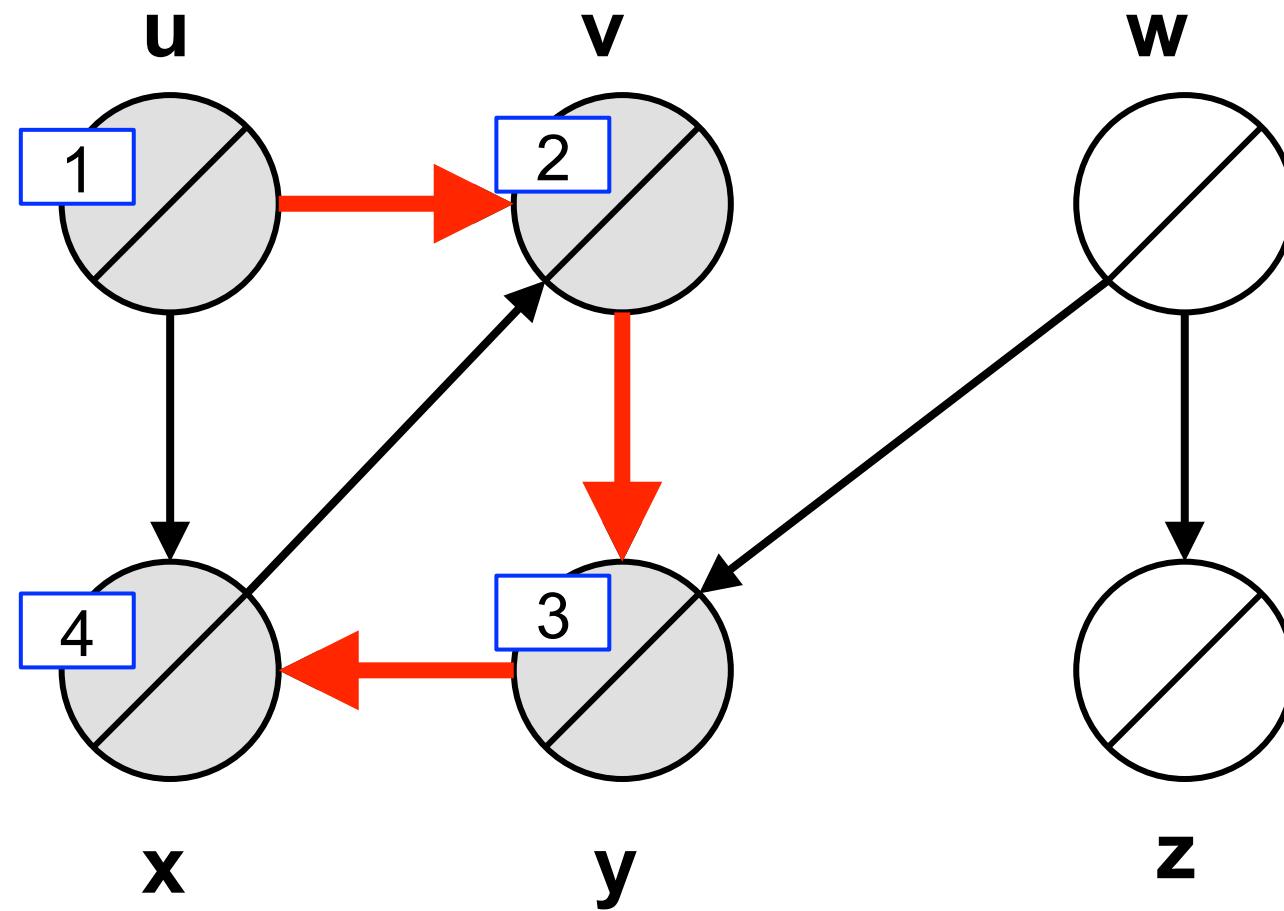
time = 2, recursive call, level 2



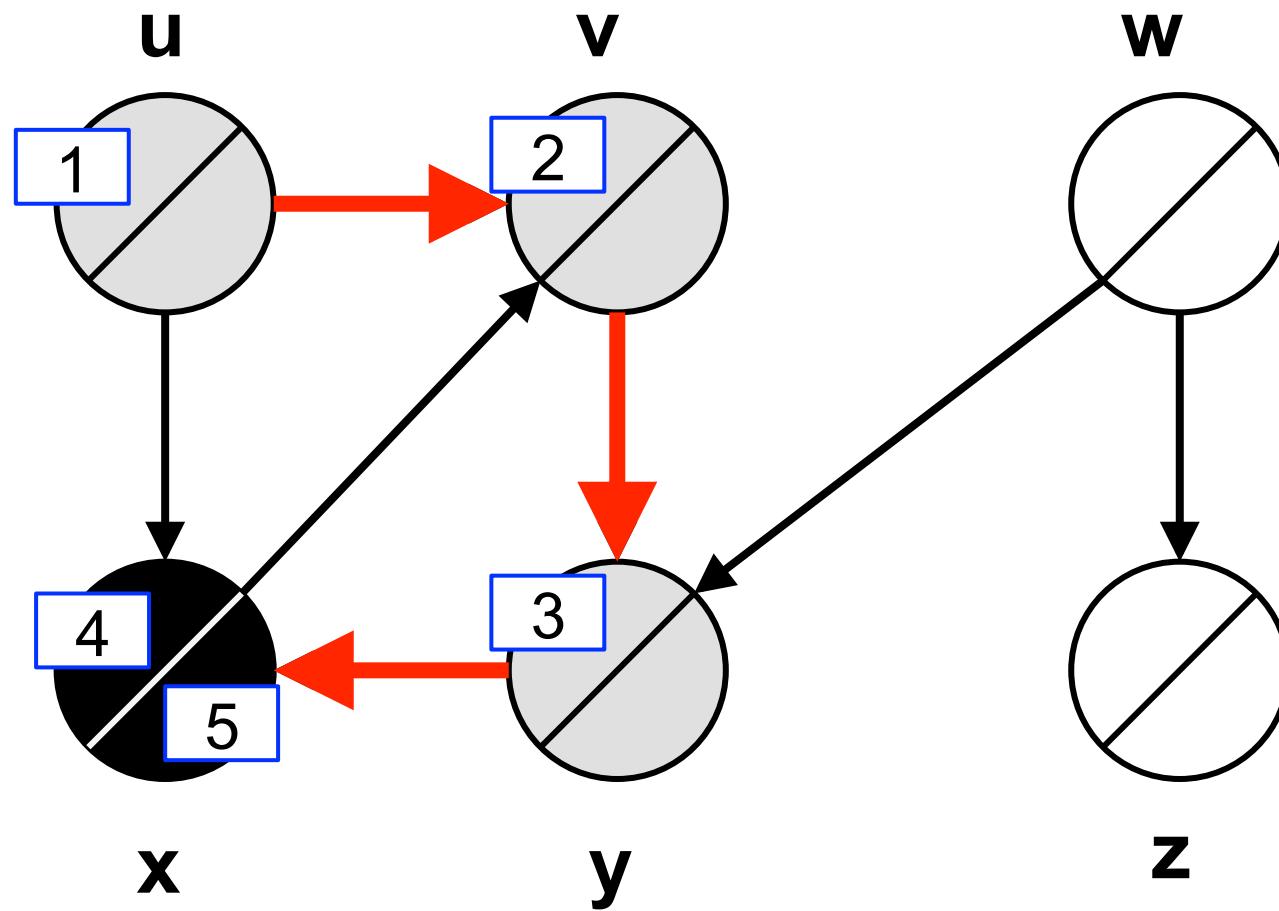
time = 3, recursive call, level 3



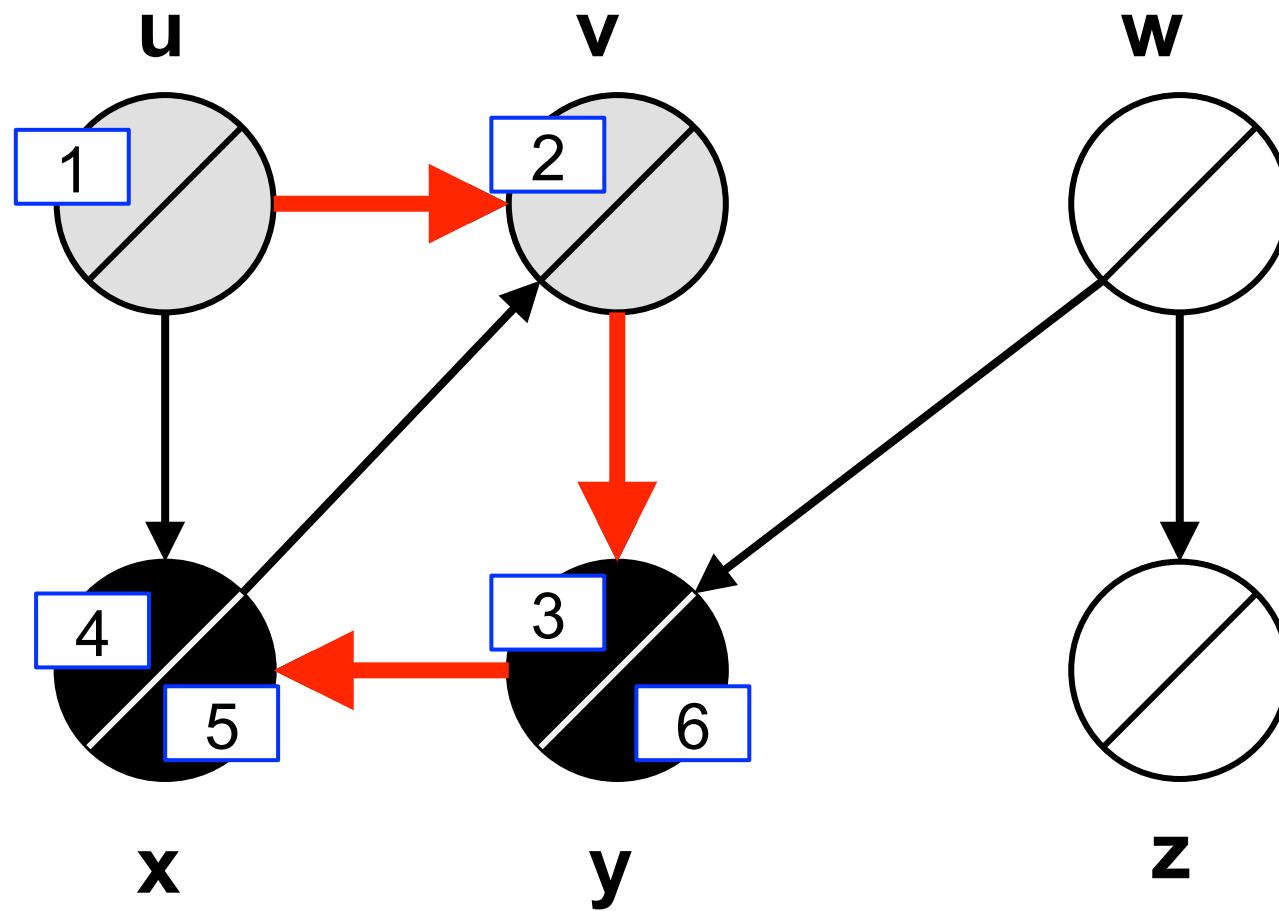
time = 4, recursive call, level 4



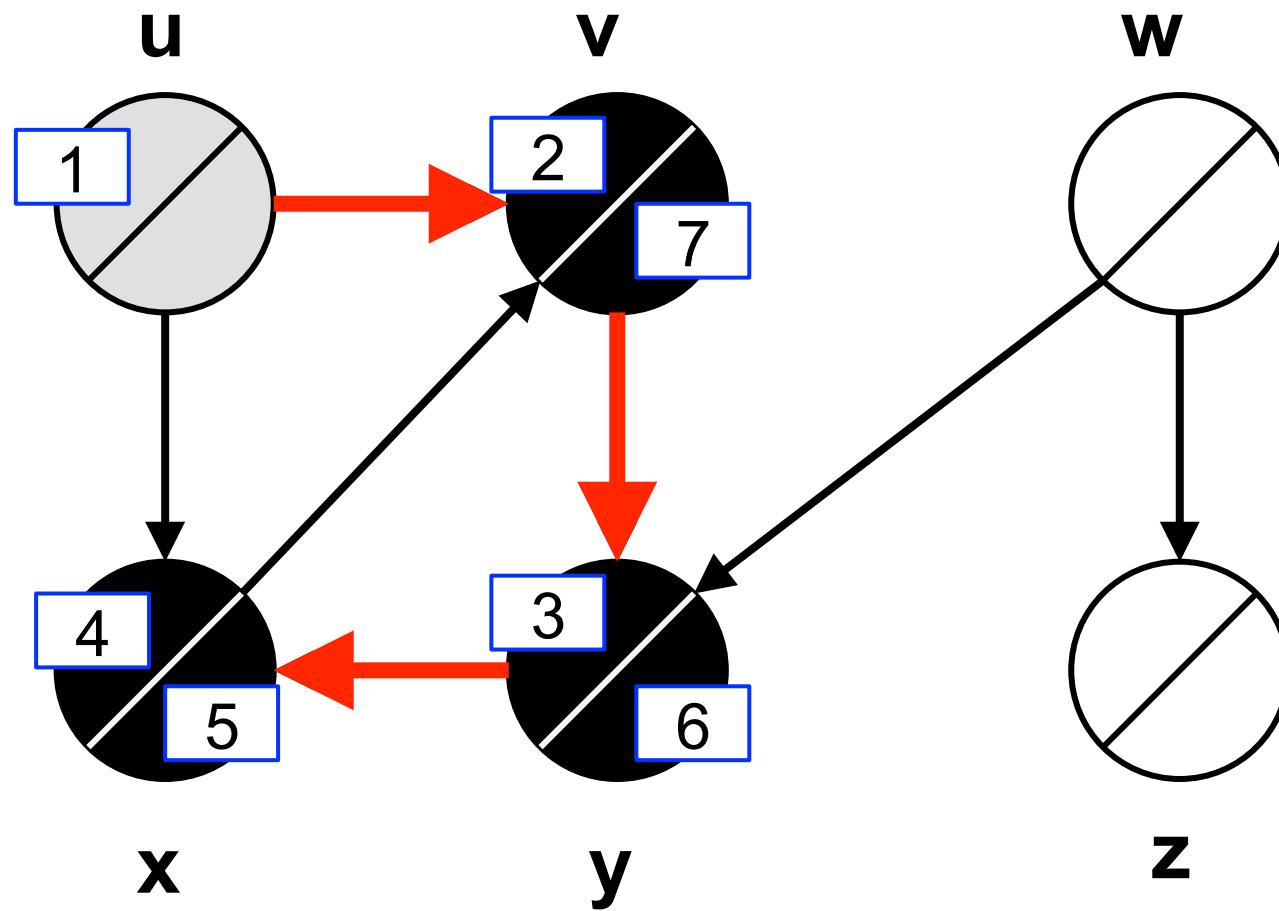
time = 5, vertex x finished



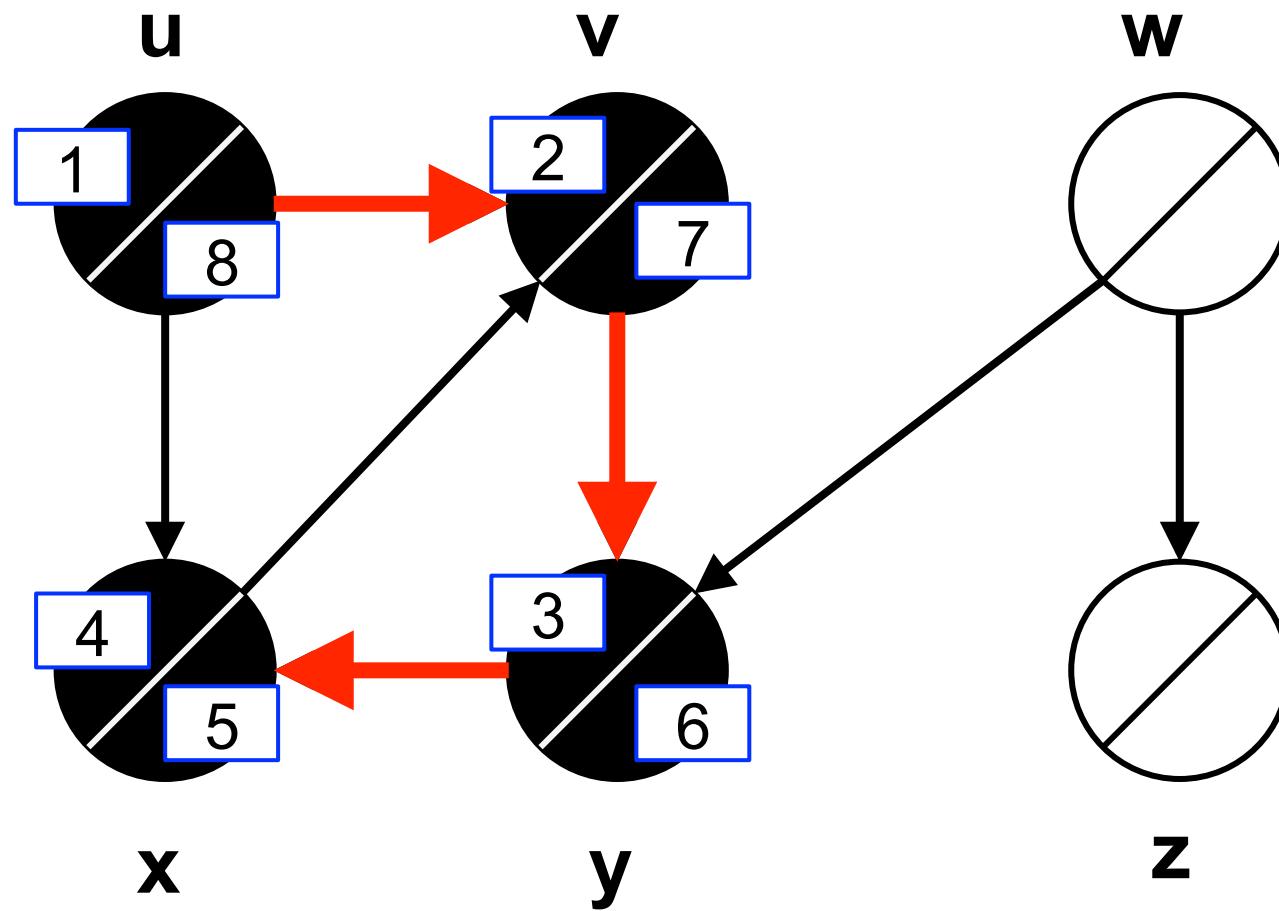
time = 6, recursion back to level 3, finish y



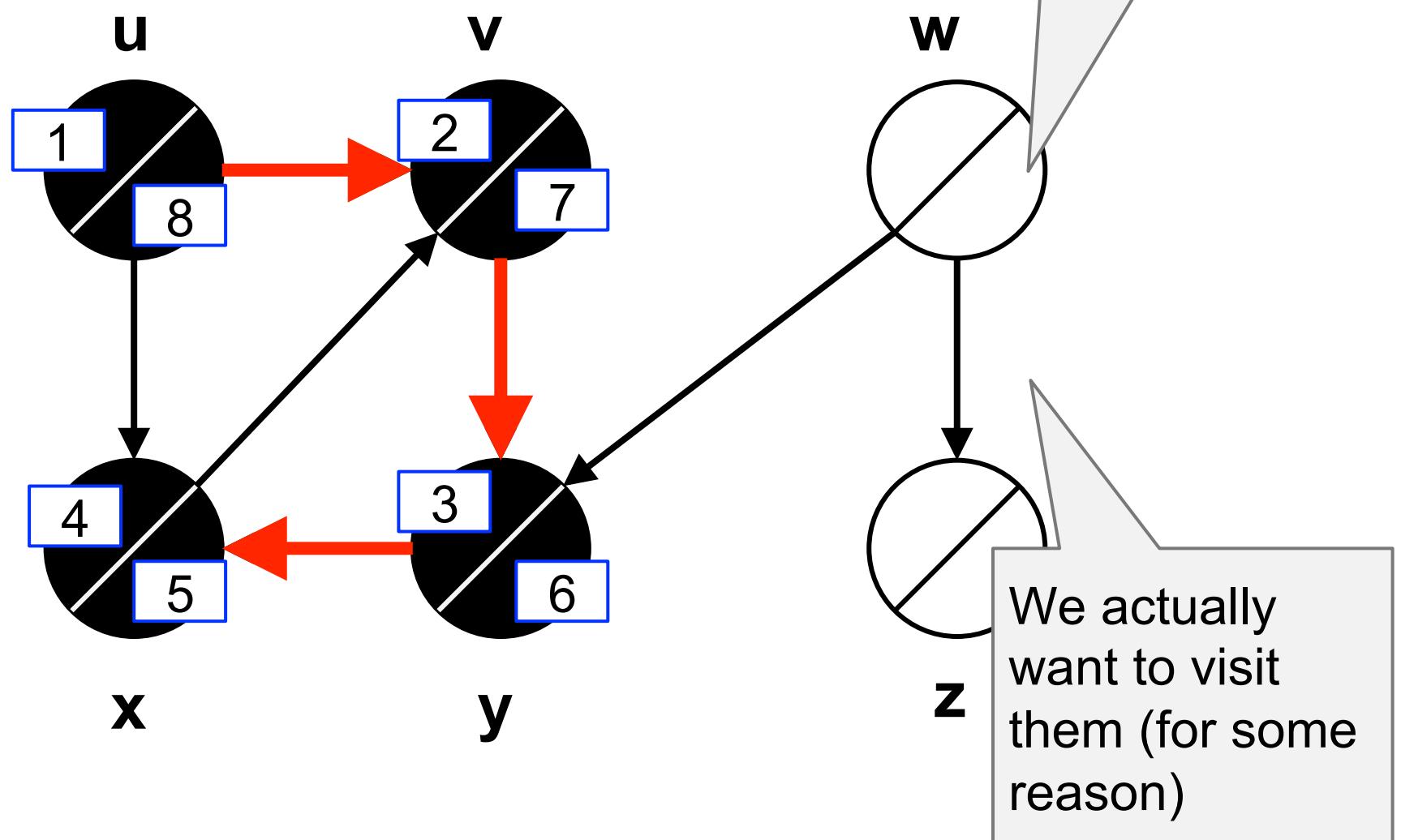
time = 7, recursive back to level 2, finish v



time = 8, recursion back to level 1, finish u



DFS_VISIT(G, u) done!



The pseudo-code for visiting everyone

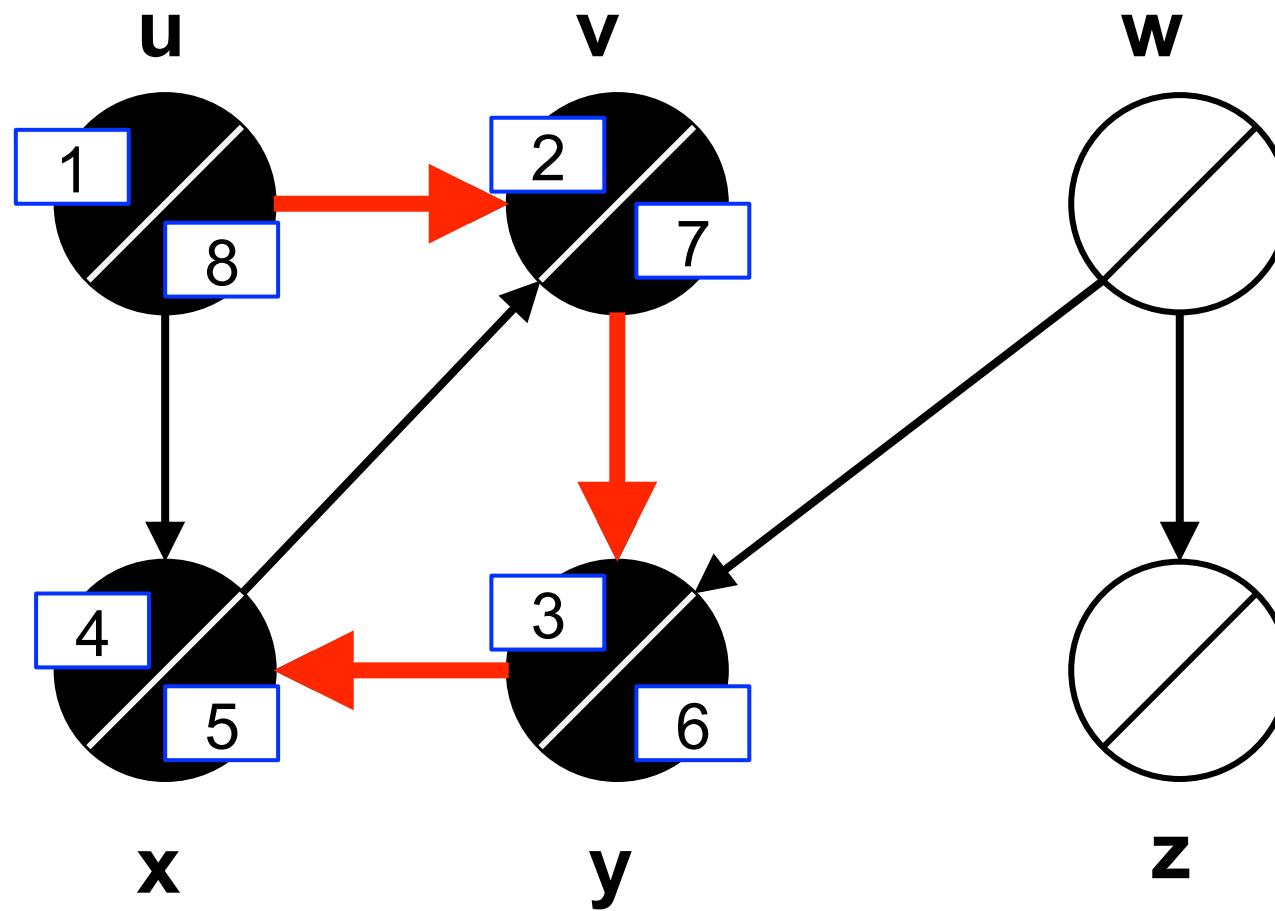
```
DFS(G):  
    for each v in G.V:  
        colour[v] ← white  
        f[v] ← d[v] ← ∞  
        pi[v] ← NIL  
    time ← 0  
    for each v in G.V:  
        if colour[v] = white:  
            DFS_VISIT(G, v)
```

Initialization

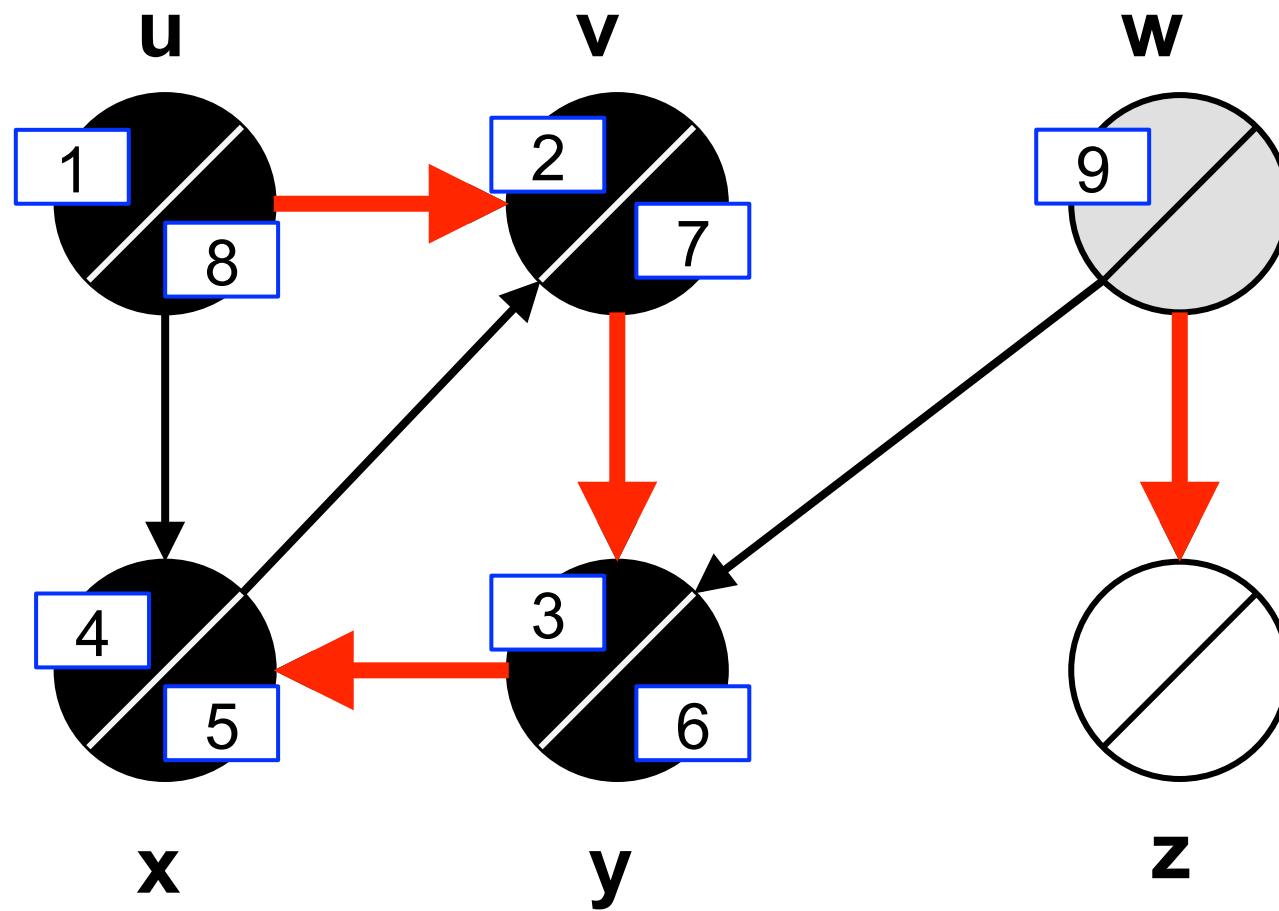
```
DFS_VISIT(G, u):  
    colour[u] ← gray  
    time ← time + 1  
    d[u] ← time  
    for each neighbour v of u:  
        if colour[v] = white:  
            pi[v] ← u  
            DFS_VISIT(G, v)  
    colour[u] ← black  
    time ← time + 1  
    f[u] ← time
```

Make sure NO vertex is left with white colour.

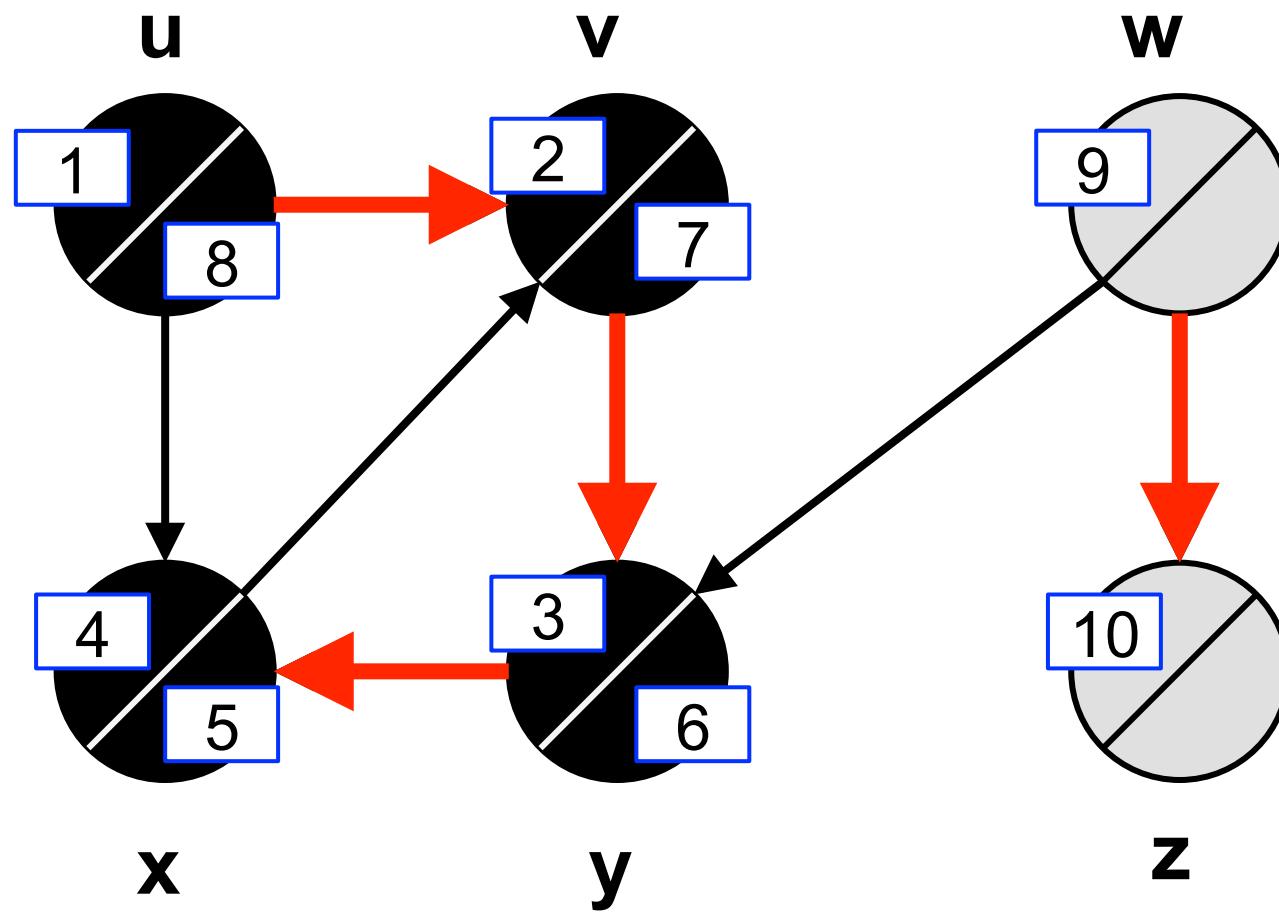
So, let's finish this DFS



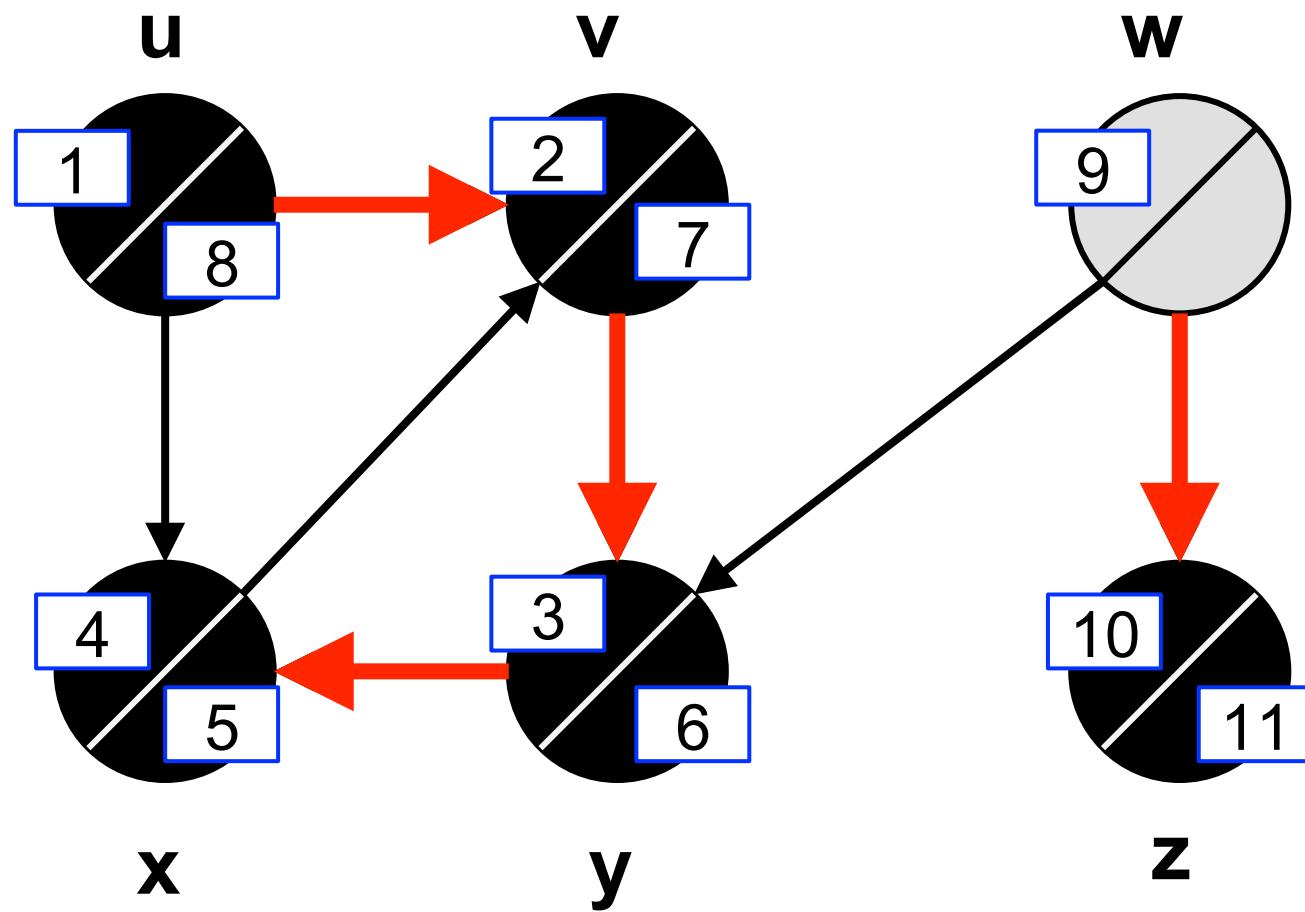
time = 9, DFS_VISIT(G, w)



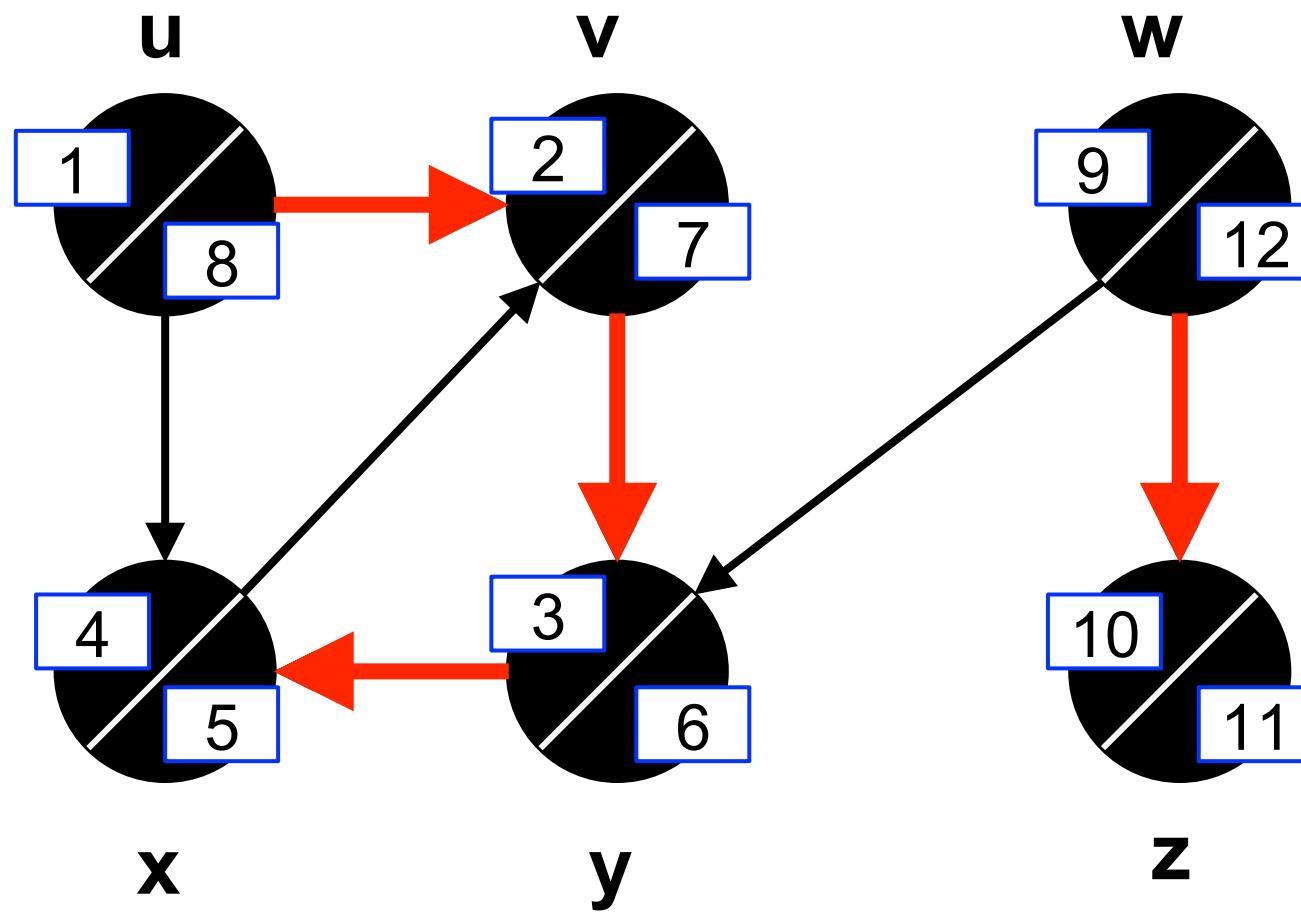
time = 10



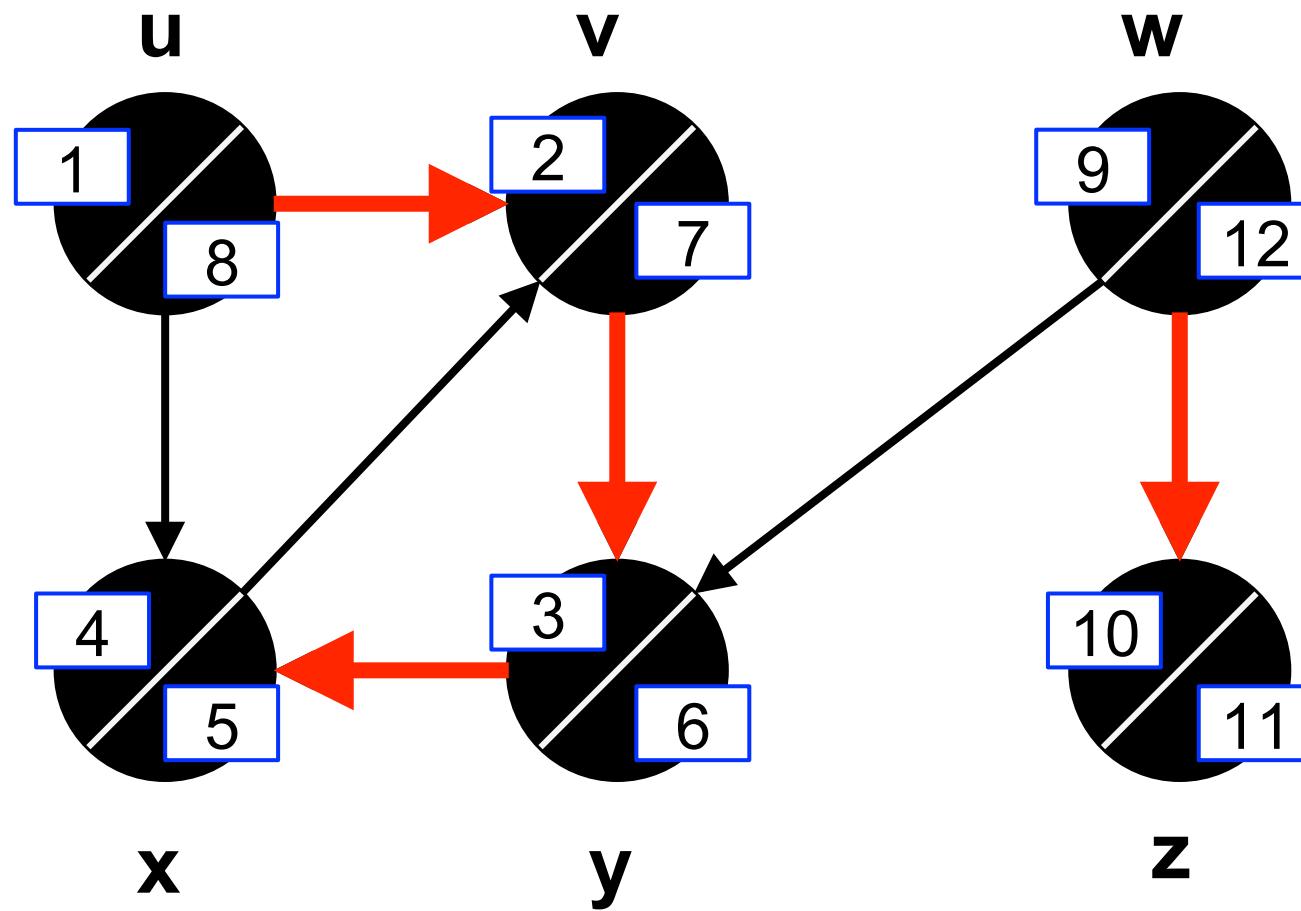
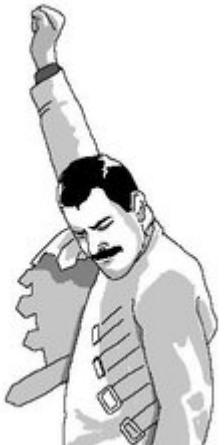
time = 11



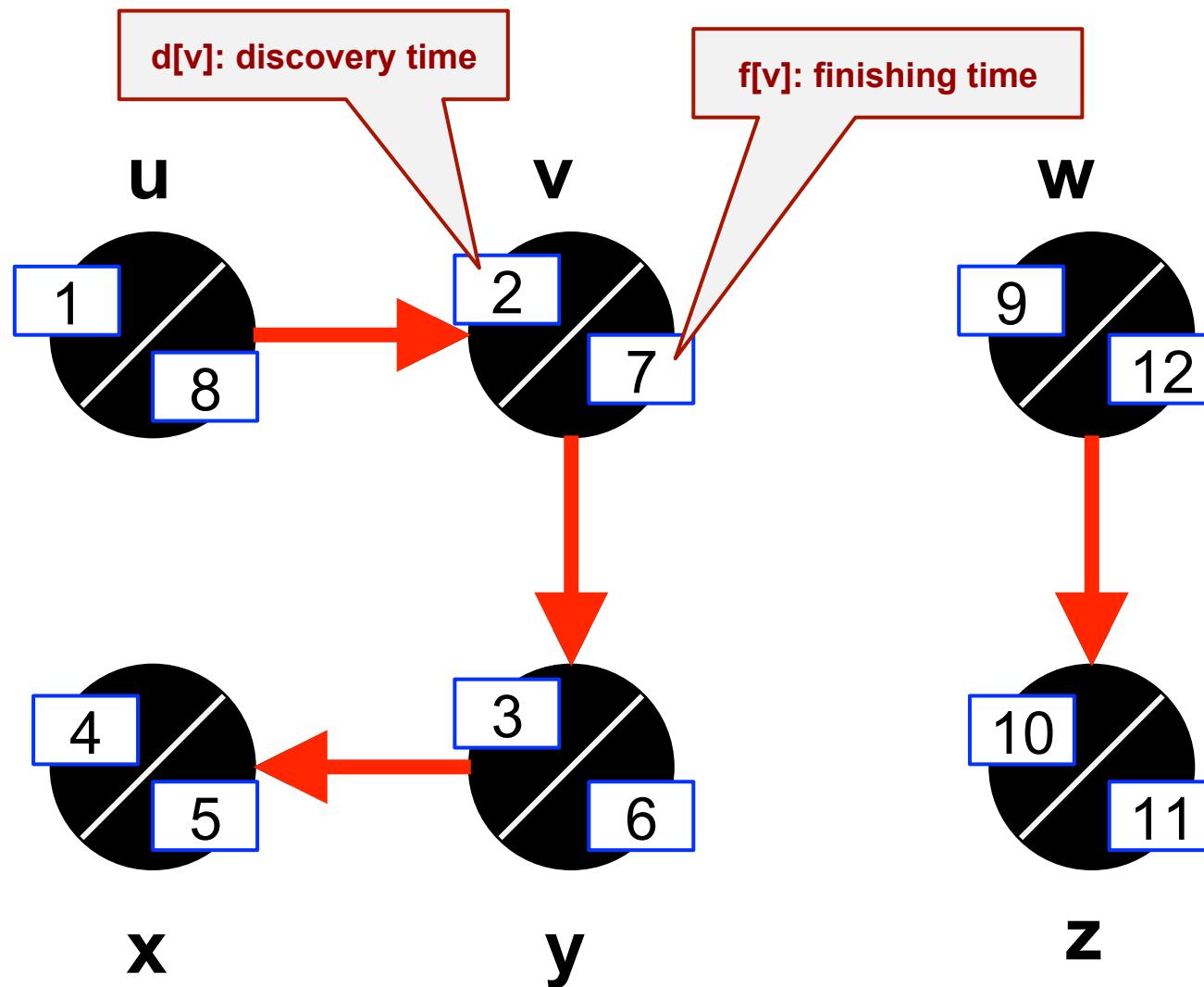
time = 12



DFS(G) done!



Recap



We get a
DFS forest
(a set of
disjoint
trees)

Runtime analysis!

The total amount of work (use **adjacency list**):

- Visit each vertex once
 - ◆ constant work per vertex
 - ◆ in total: $O(|V|)$
- At each vertex, check all its neighbours (all its **incident edges**)
 - ◆ Each edge is checked **once** (in a directed graph)
 - ◆ in total: $O(|E|)$

Same as BFS

Total runtime:
 $O(|V|+|E|)$

What do we get from DFS?

→ Detect whether a graph has a cycle.

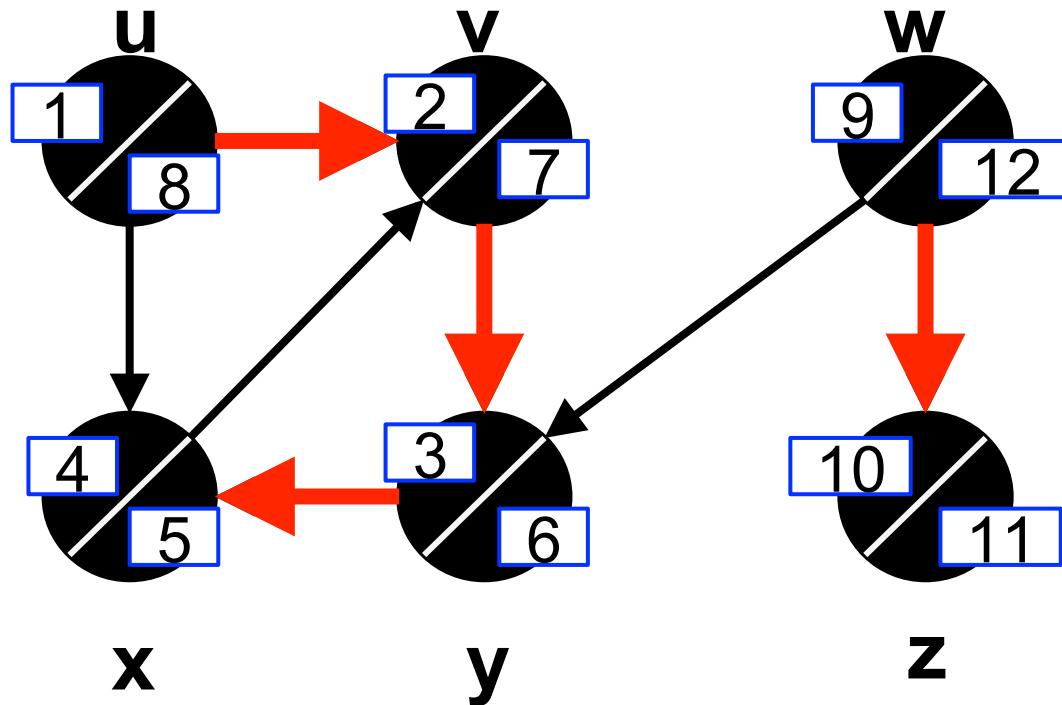
- ◆ That's why we wanted to visit all vertices -- if you want to be sure whether a graph has a cycle or not, you'd better check **everywhere**.
- ◆ Why didn't we do the similar thing for BFS?

→ How exactly do we detect a cycle?

**determine descendant / ancestor
relationship in the DFS forest**

How to decide whether y is a **descendant** of u in the DFS forest?

Idea #1: trace back the $\text{pi}[v]$ pointers
(the red edges) starting from y , see
whether you can get to u .
Worst-case takes $O(n)$ steps.



the “parenthesis structure”

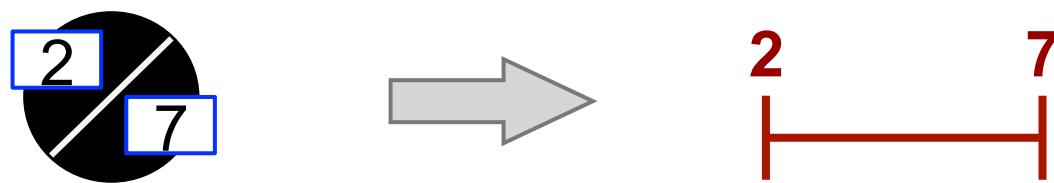
((())) () (())

- Either one pair **contains** the another pair.
- Or one pair is **disjoint** from another

(())

This (overlapping)
never happens!

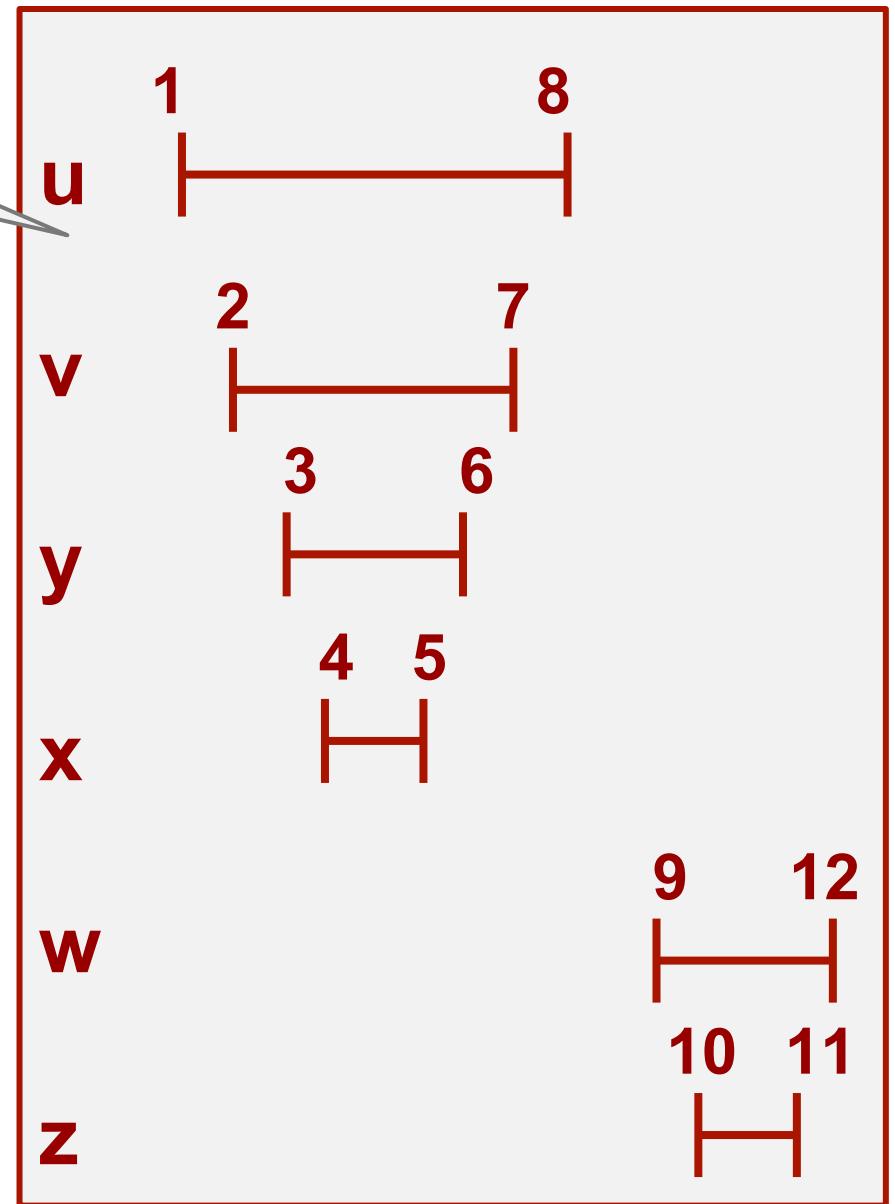
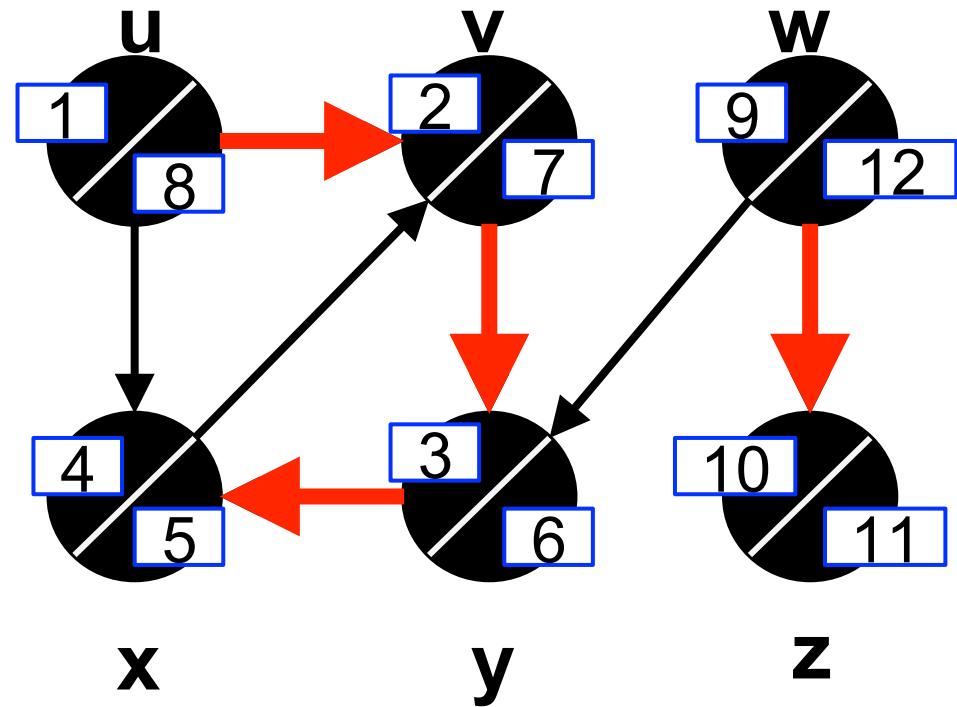
Visualize $d[v]$, $f[v]$ as interval $[d[v], f[v]]$



Now, visualize all the intervals!

What do you see in this?

Parenthesis structure!



The $[d[v], f[v]]$ intervals that we got from DFS follow the parenthesis structure, i.e.,

- Either one interval **contains** another
- Or one is **disjoint** from another

Moreover,

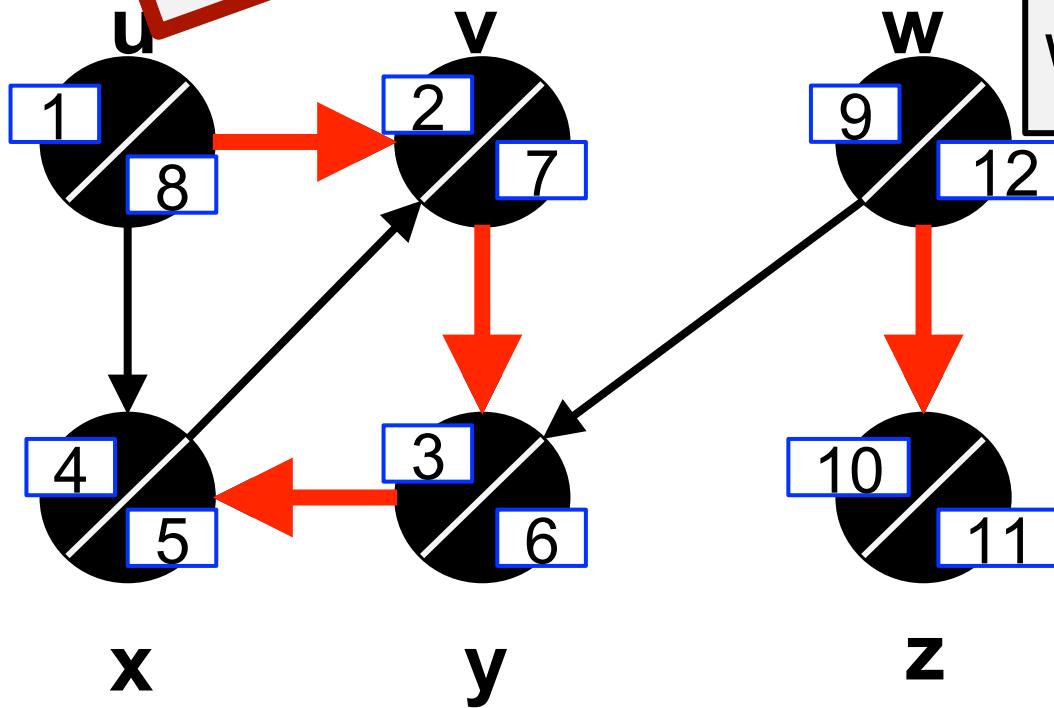
- **Iff** interval of u contains interval of v , then u is an **ancestor** of v in the DFS forest.
- If interval of u is disjoint from interval of v , then they are **not** ancestors of each other.

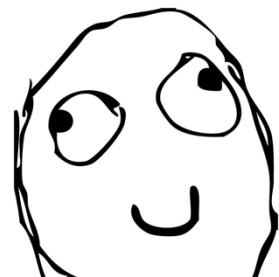
How to decide whether y is a descendant of u in the DFS forest?

Idea #1: trace back from y to u .
(the root of the tree)
Worst-case: $O(n)$ steps.
Very slow.

FORGET ABOUT IT

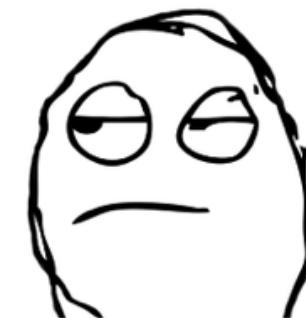
Idea #2: see if $[d[u], f[u]]$ contains $[d[y], f[y]]$.
Worst-case: 1 step!





We can efficiently check whether a vertex is an ancestor of another vertex in the DFS forest.

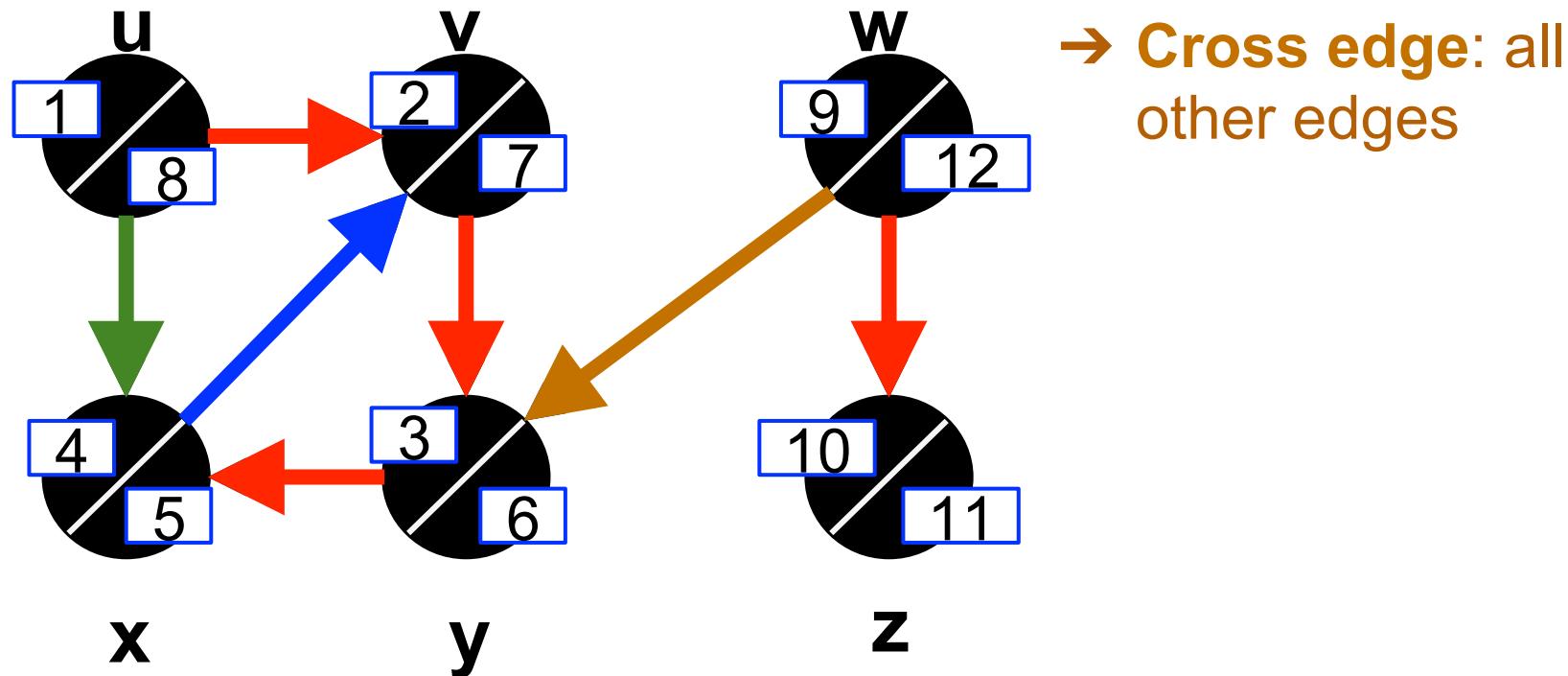
so what...



Classifying Edges

4 types of edges in a graph after a DFS

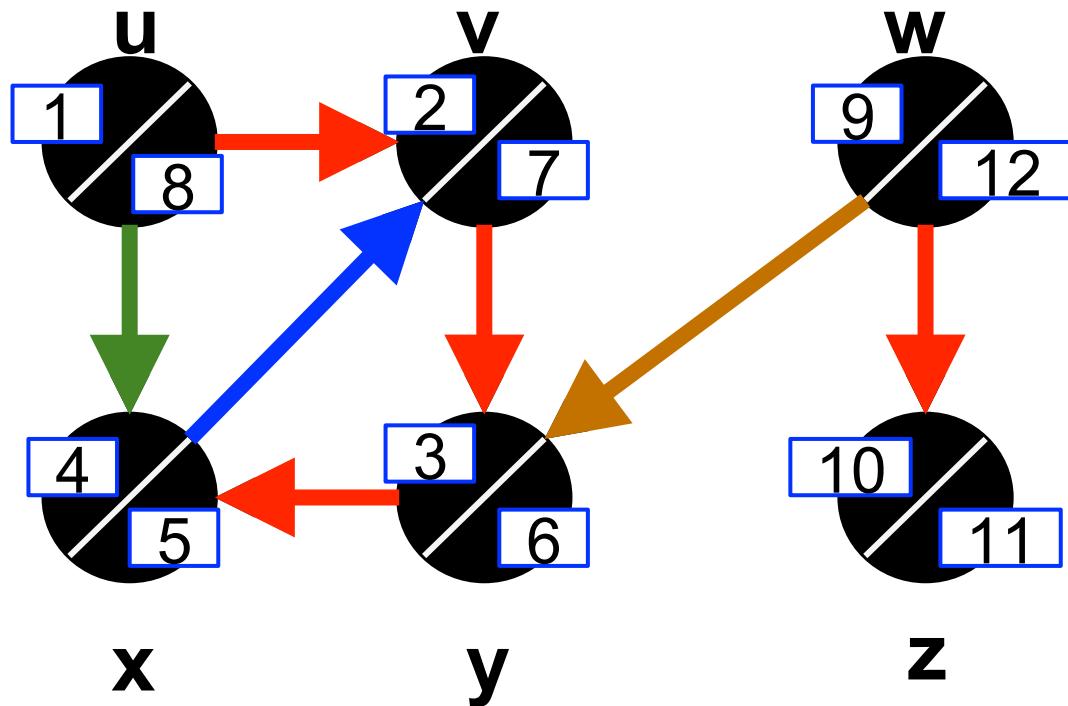
- **Tree edge:** an edge in the DFS-forest
- **Back edge:** a non-tree edge pointing from a vertex to its **ancestor** in the DFS forest.
- **Forward edge:** a non-tree edge pointing from a vertex to its **descendant** in the DFS forest

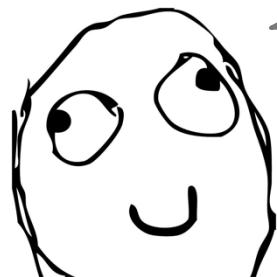


Checking edge types

We can efficiently check edge types, because...

we can efficiently check whether a vertex is an **ancestor / descendant** of another vertex using...
the **parenthesis structure** of **[$d[v]$, $f[v]$]** intervals!





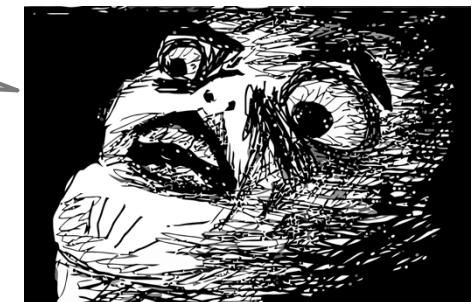
We can efficiently check edge types after a DFS!

so what...

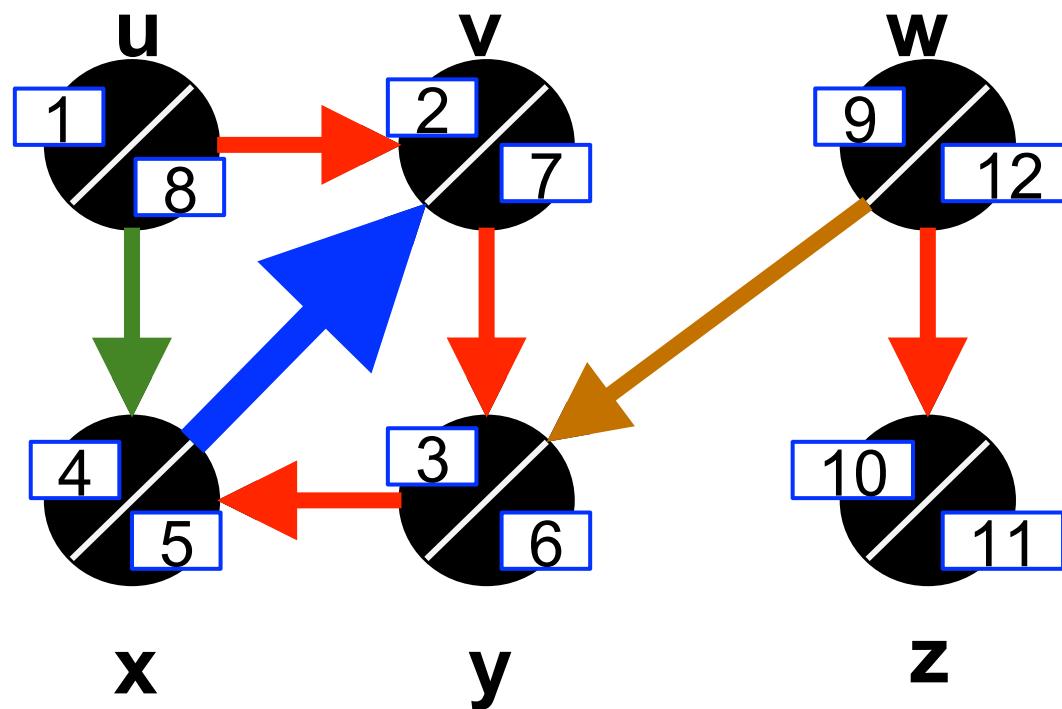


A graph is **cyclic** if and only if DFS yields a **back edge**.

That's useful!



A (directed) graph contains a **cycle** if and only if DFS yields a **back edge**



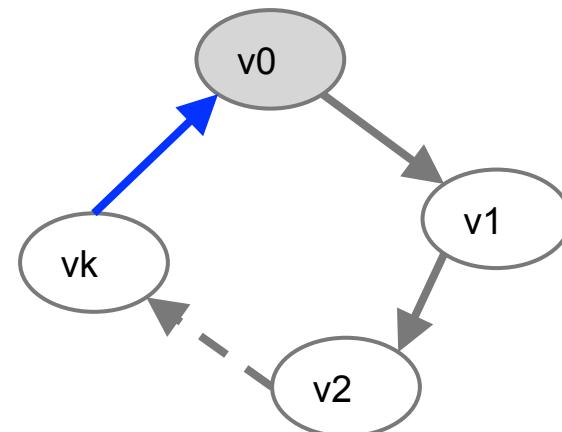
A (directed) graph contains a **cycle** if and only if DFS yields a **back edge**

Proof of “if”:

Let the edge be (u, v) ,
then by definition of back
edge, v is an ancestor of u
in the DFS tree,
then there is a path from v to
 u , i.e., $v \rightarrow \dots \rightarrow u$,
plus the back edge $u \rightarrow v$,
BOOM! Cycle.

Proof of “only if”:

Let the cycle be...,

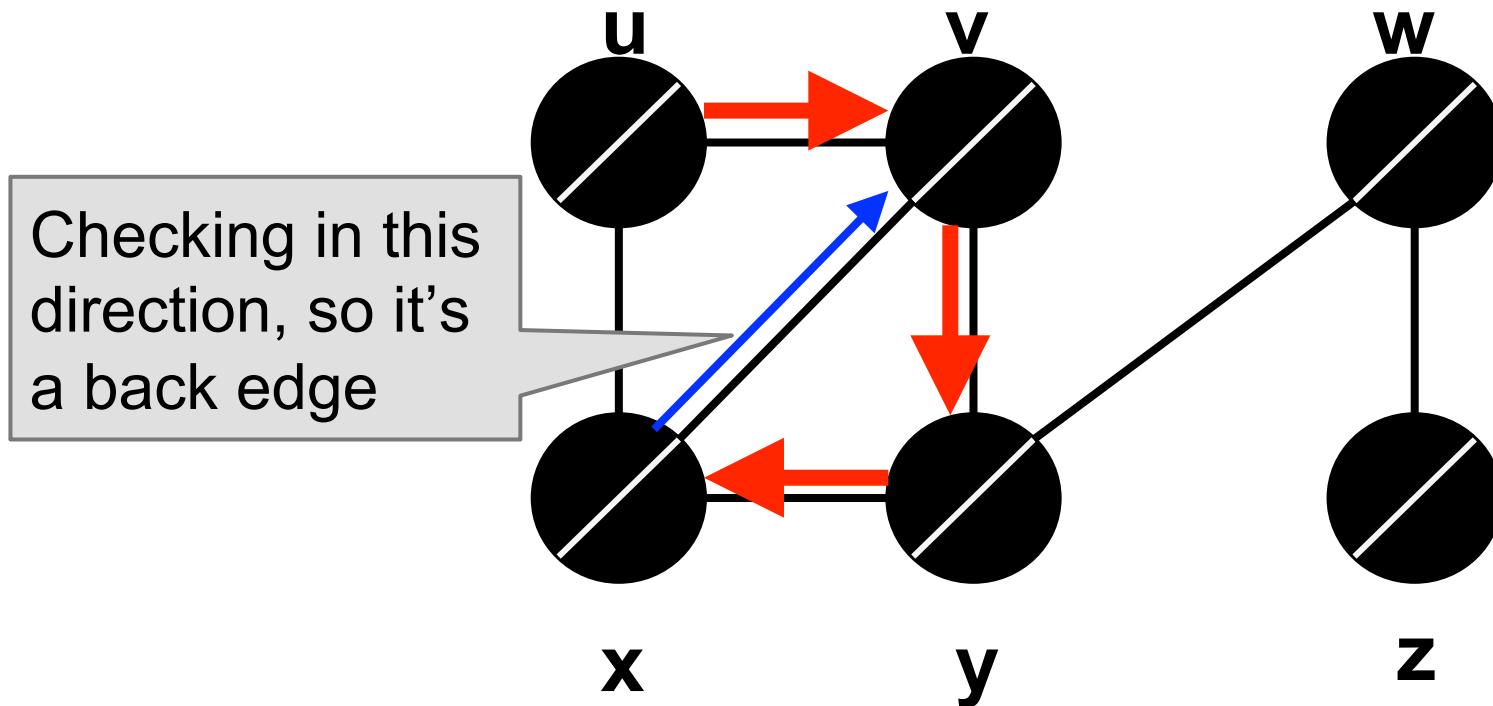


Let v_0 be the first one that turns gray, when all others in the cycle are white, then v_k must be a descendant of v_0 . (Read “White Path Theorem” in Text)

How about undirected graph?

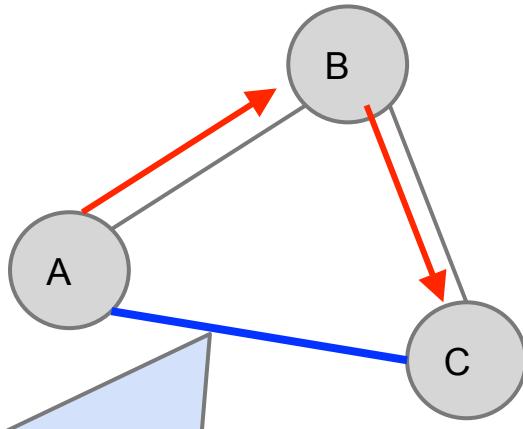
Should **back** and **forward** edges be the same thing?

→ No, because although the edges are undirected,
neighbour checking still has a “direction”.

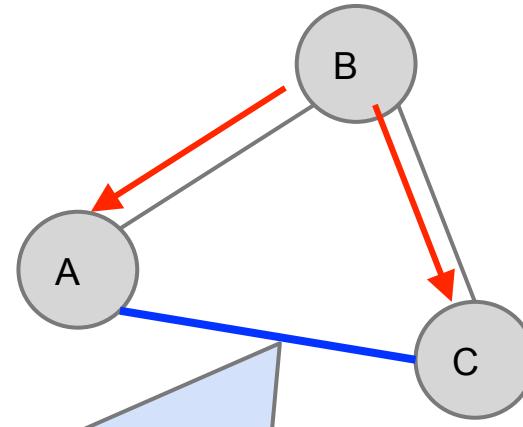


More about undirected graph

After a DFS on a undirected graph, **every** edge is either a **tree edge** or a **back edge**, i.e., **no** forward edge or cross edge.



If this were a forward edge, it would violate the DFS algorithm (not checking at C but tracing back and check at A)

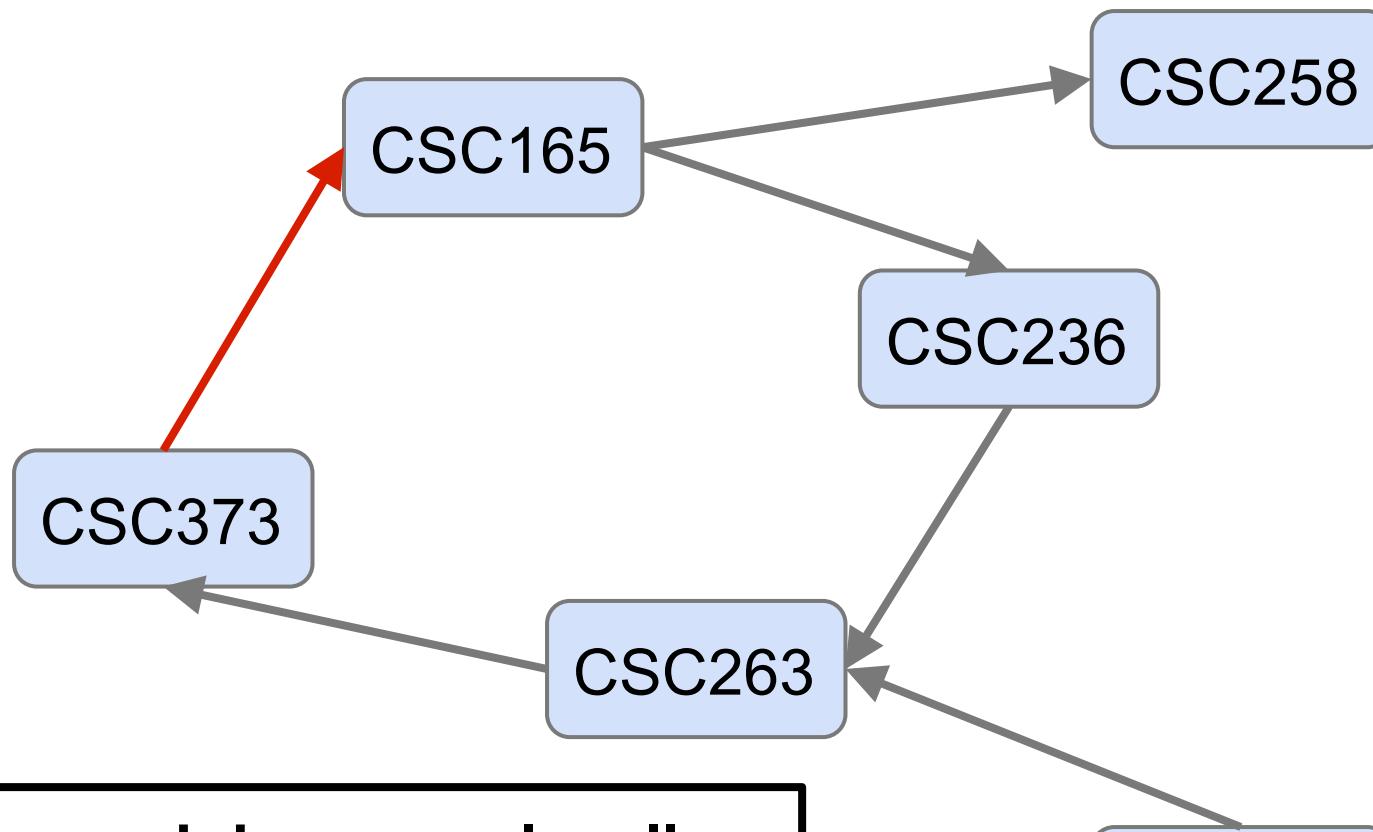


If this were a cross edge, it violates DFS again (should have checked (A, C) when reached A, but instead wait until C is visited.)

Why do we care about **cycles** in a graph?

Because cycles have meaningful implication
in real applications.

Example: a course prerequisite graph



If the graph has a cycle, all courses in the cycle become **impossible to take!**

Applications of DFS

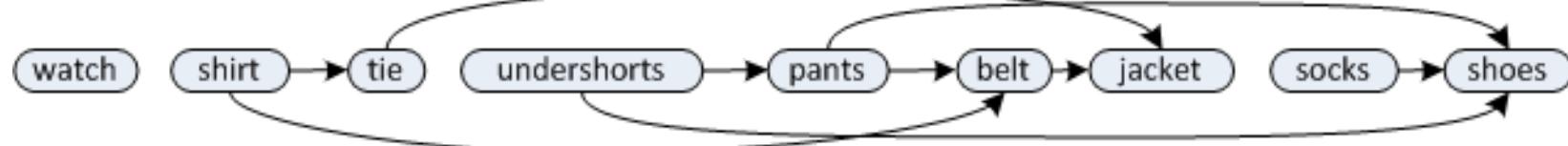
- Detect cycles in a graph
- Topological sort
- Strongly connected components

Topological Sort

→ Place the vertices in such an order that all edges are pointing to the right side.



A valid order of getting dressed.



Topological sort more formally

Suppose that in a **directed** graph $G = (V, E)$ vertices V represent tasks, and each edge $(u, v) \in E$ means that task u must be done before task v

What is an ordering of vertices $1, \dots, |V|$ such that for every edge (u, v) , u appears before v in the ordering?

Such an ordering is called a **topological sort of G**

Note: there can be multiple topological sorts of G

Topological sort more formally

Is it possible to execute all the tasks in **G** in an order that respects all the precedence requirements given by the graph edges?

The answer is "**yes**" *if and only if* the directed graph **G** has **no cycle**!

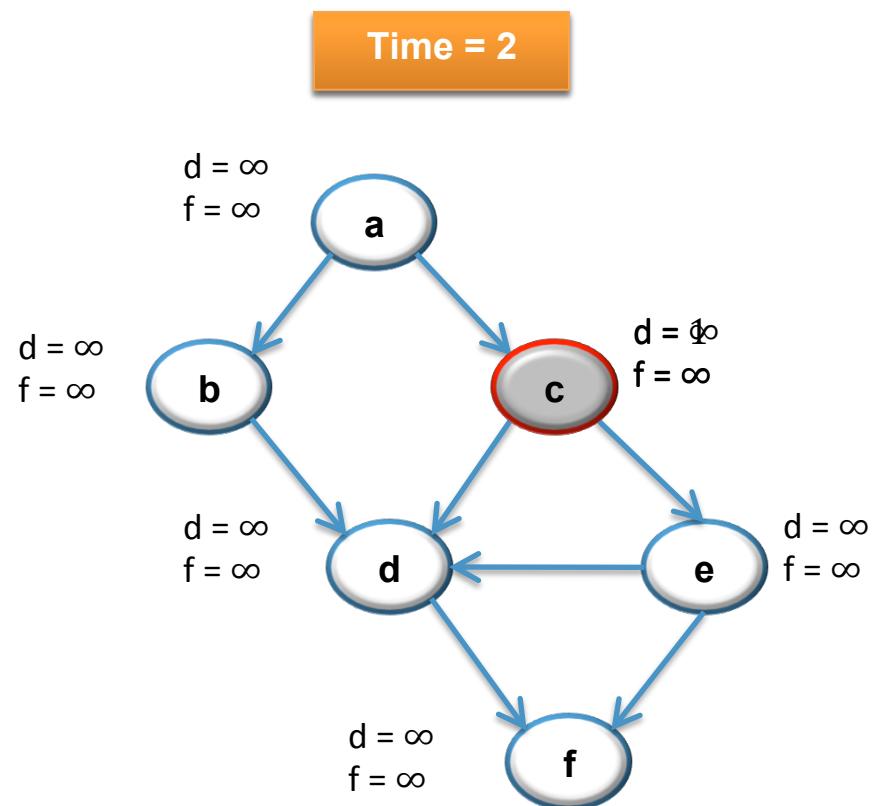
(otherwise we have a **deadlock**)

Such a **G** is called a Directed Acyclic Graph, or just a **DAG**

Algorithm for TS

- **TOPOLOGICAL-SORT(G):**
 - 1) call DFS(G) to compute **finishing** times $f[v]$ for each vertex v
 - 2) as each vertex is finished, insert it onto the **front** of a linked list
 - 3) return the linked list of vertices
- Note that the result is just a list of vertices in order of **decreasing** finish times $f[]$

Topological sort

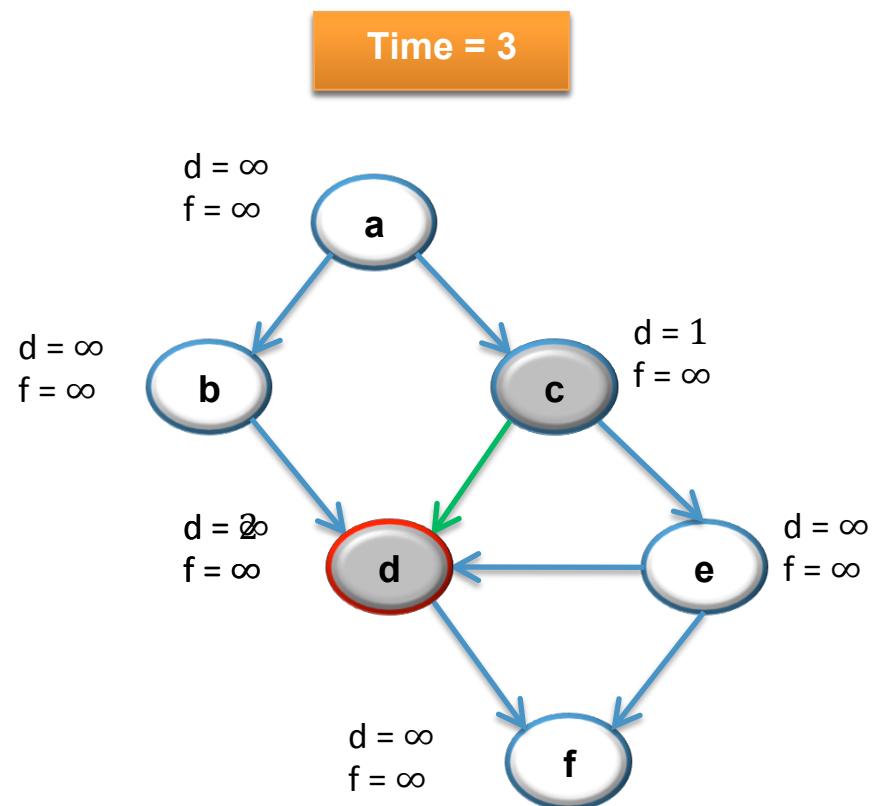


- 1) Call DFS(**G**) to compute the finishing times **f[v]**

Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Topological sort

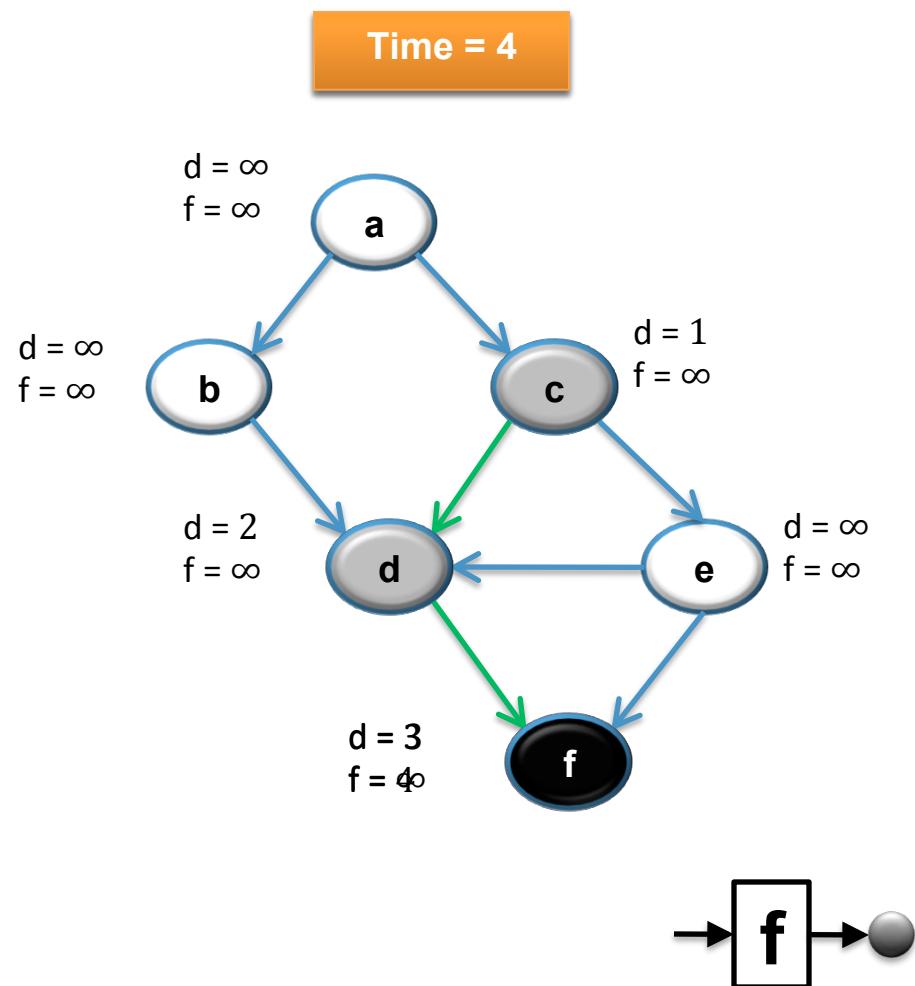


- 1) Call $\text{DFS}(\mathbf{G})$ to compute the finishing times $f[v]$

Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Topological sort



1) Call $\text{DFS}(\mathbf{G})$ to compute the finishing times $f[v]$

2) as each vertex is finished, insert it onto the **front** of a linked list

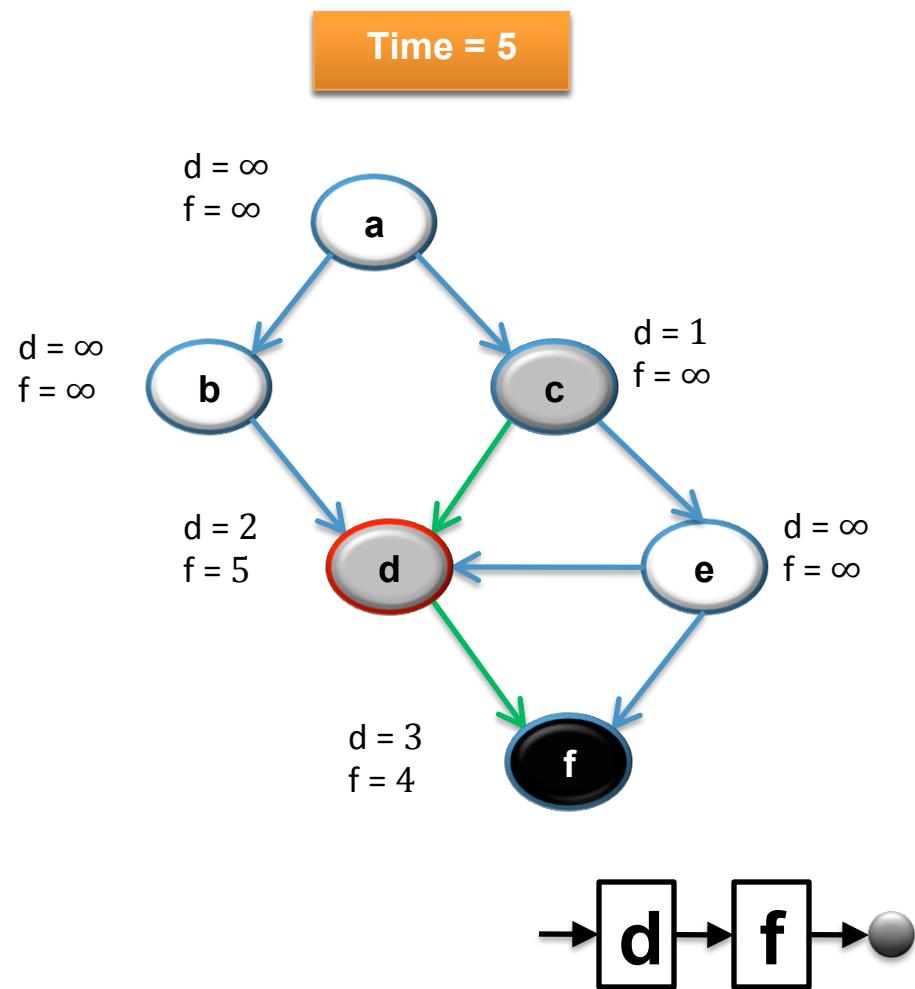
vertex d

Next we discover the

vertex f

f is done, move back to d

Topological sort



- 1) Call $\text{DFS}(G)$ to compute the finishing times $f[v]$

Let's say we start the DFS from the vertex **c**

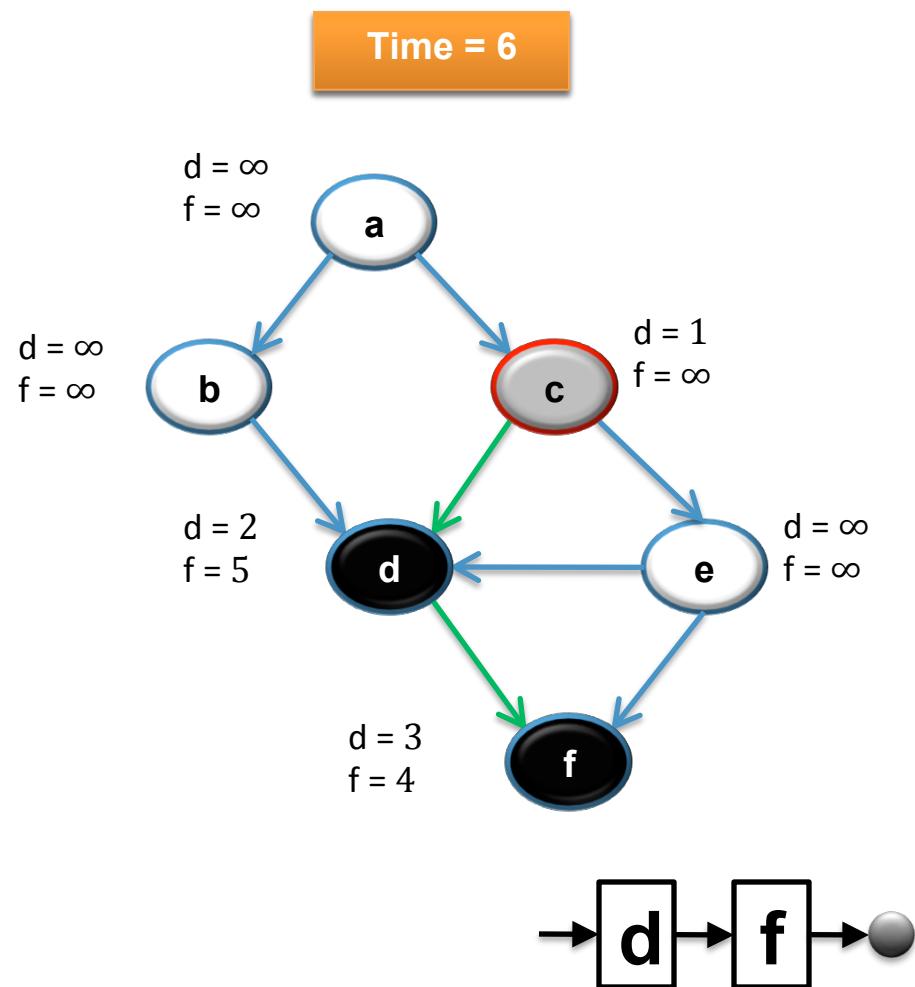
Next we discover the vertex **d**

Next we discover the vertex **f**

f is done, move back to **d**

d is done, move back to **c**

Topological sort



1) Call DFS(**G**) to compute the finishing times **f[v]**

Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

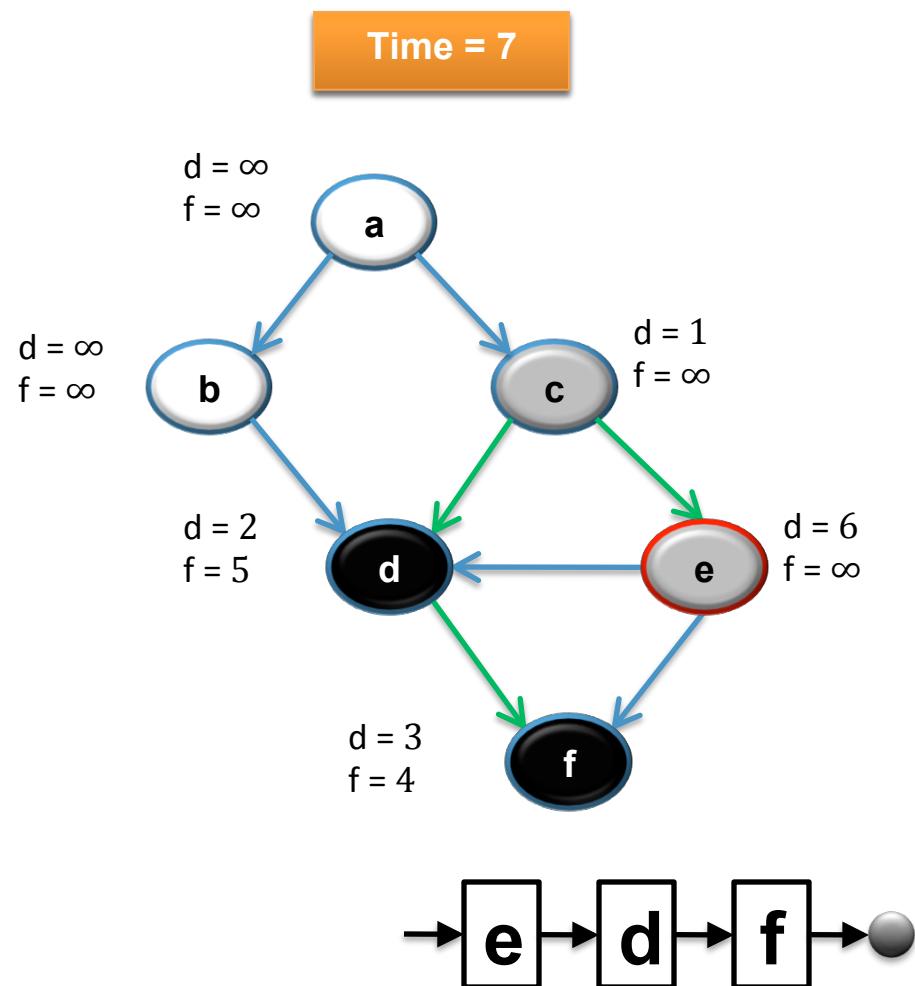
Next we discover the vertex **f**

f is done, move back to **d**

d is done, move back to **c**

Next we discover the vertex **e**

Topological sort



1) Call $\text{DFS}(\mathbf{G})$ to compute the finishing times $f[v]$

Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

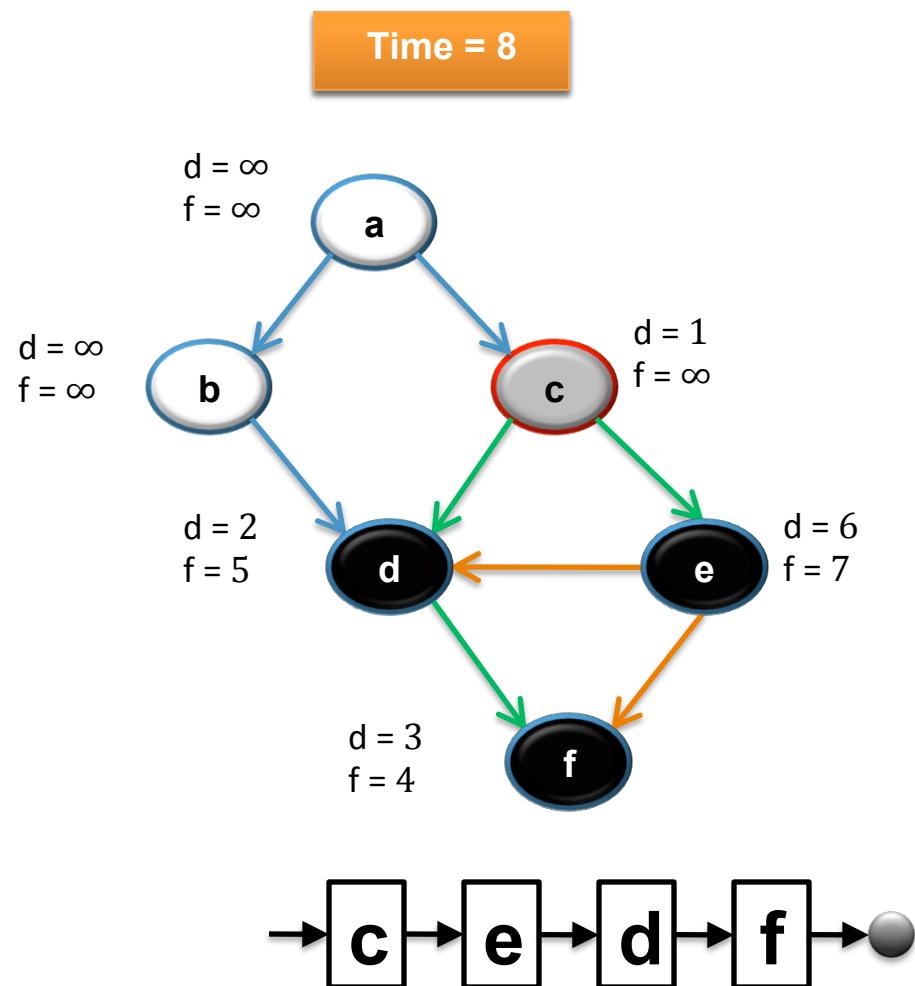
Both edges from **e** are **cross edges**

d is done, move back to **c**

Next we discover the vertex **e**

e is done, move back to **c**

Topological sort



1) Call $\text{DFS}(\mathbf{G})$ to compute the finishing times $f[v]$

Let's say we start the DFS from the vertex **c**

Just a note: If there was **(c,f)** edge in the graph, it would be classified as a **forward edge**

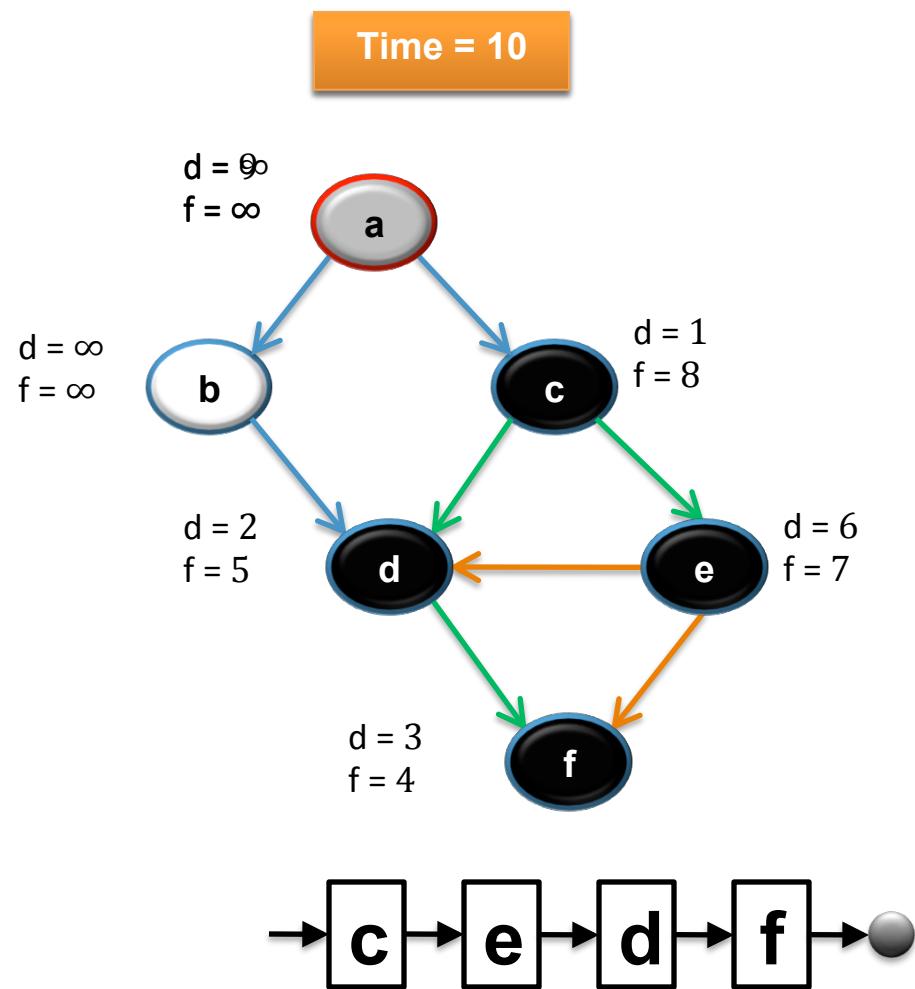
(in this particular DFS run, move back to c)

Next we discover the vertex **e**

e is done, move back to **c**

c is done as well

Topological sort



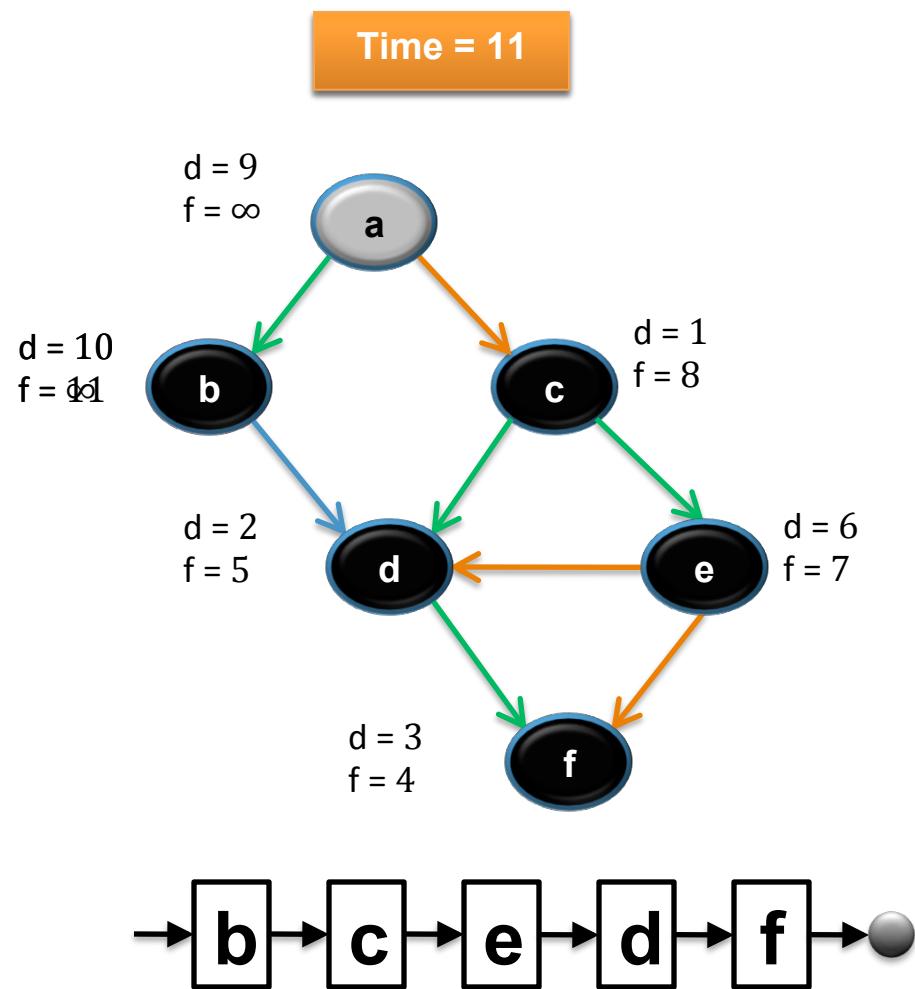
- 1) Call DFS(**G**) to compute the finishing times $f[v]$

Let's now call DFS visit from the vertex **a**

Next we discover the vertex **c**, but **c** was already processed

Next we discover the vertex **b**

Topological sort



- 1) Call DFS(**G**) to compute the finishing times $f[v]$

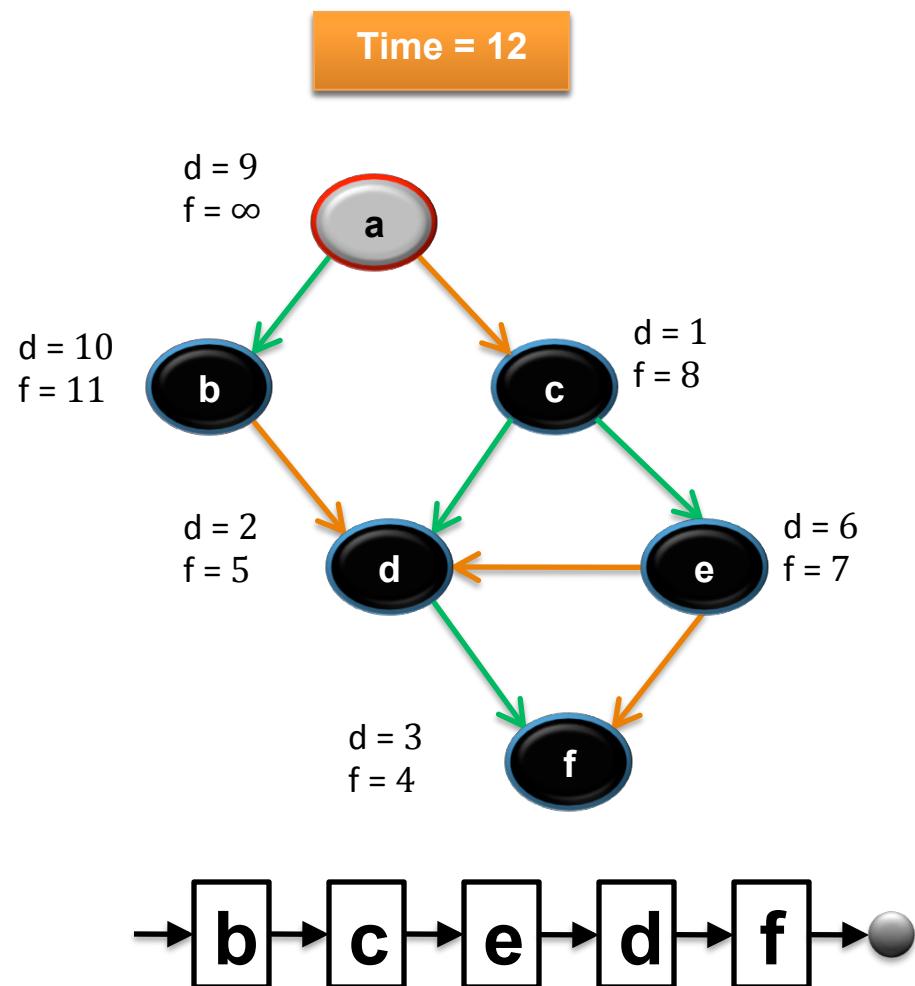
Let's now call DFS visit from the vertex **a**

Next we discover the vertex **c**, but **c** was already processed as (a,c) is a

Next we discover the vertex **b** as

b is done as (b,d) is a cross edge => now move back to **c**

Topological sort



- 1) Call DFS(**G**) to compute the finishing times $f[v]$

Let's now call DFS visit from the vertex **a**

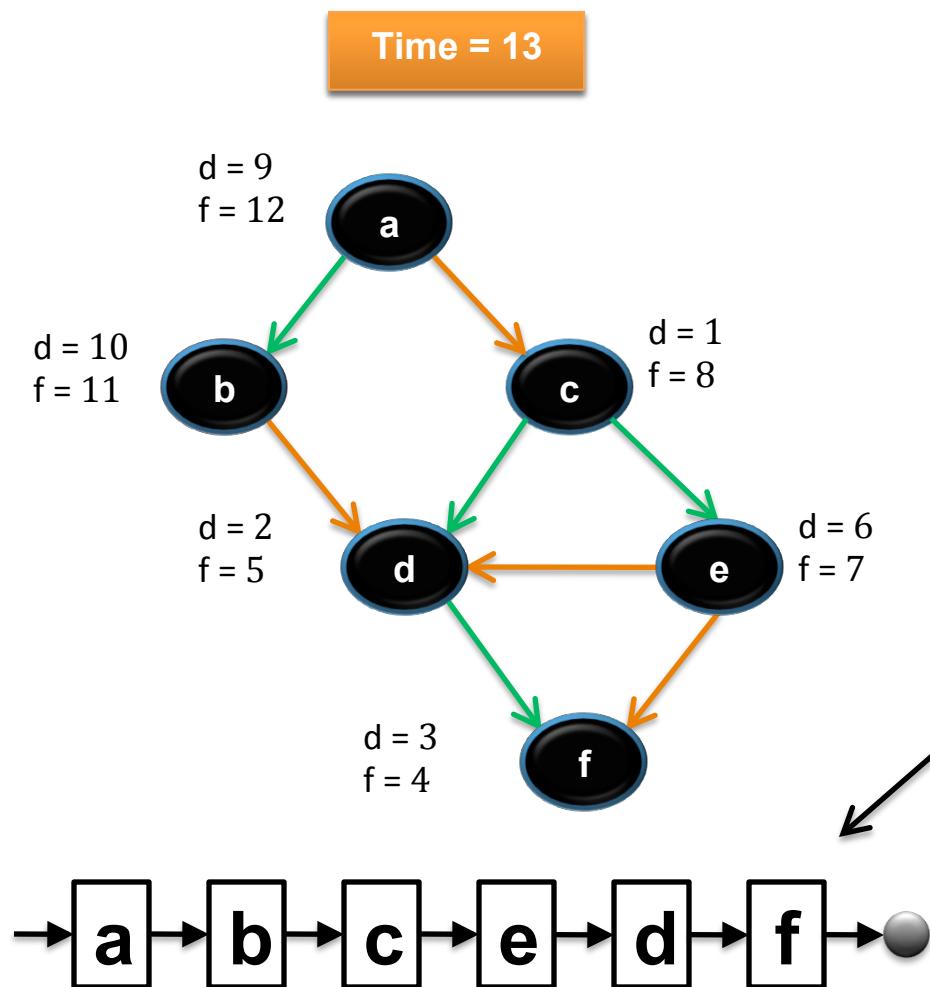
Next we discover the vertex **c**, but **c** was already processed as (a,c) is a

Next we discover the vertex **b**.

b is done as (b,d) is a cross edge => now move back to **c**

a is done as well

Topological sort



1) Call $\text{DFS}(\mathbf{G})$ to compute the finishing times $f[v]$

WE HAVE THE RESULT!

3) return the linked list of vertices

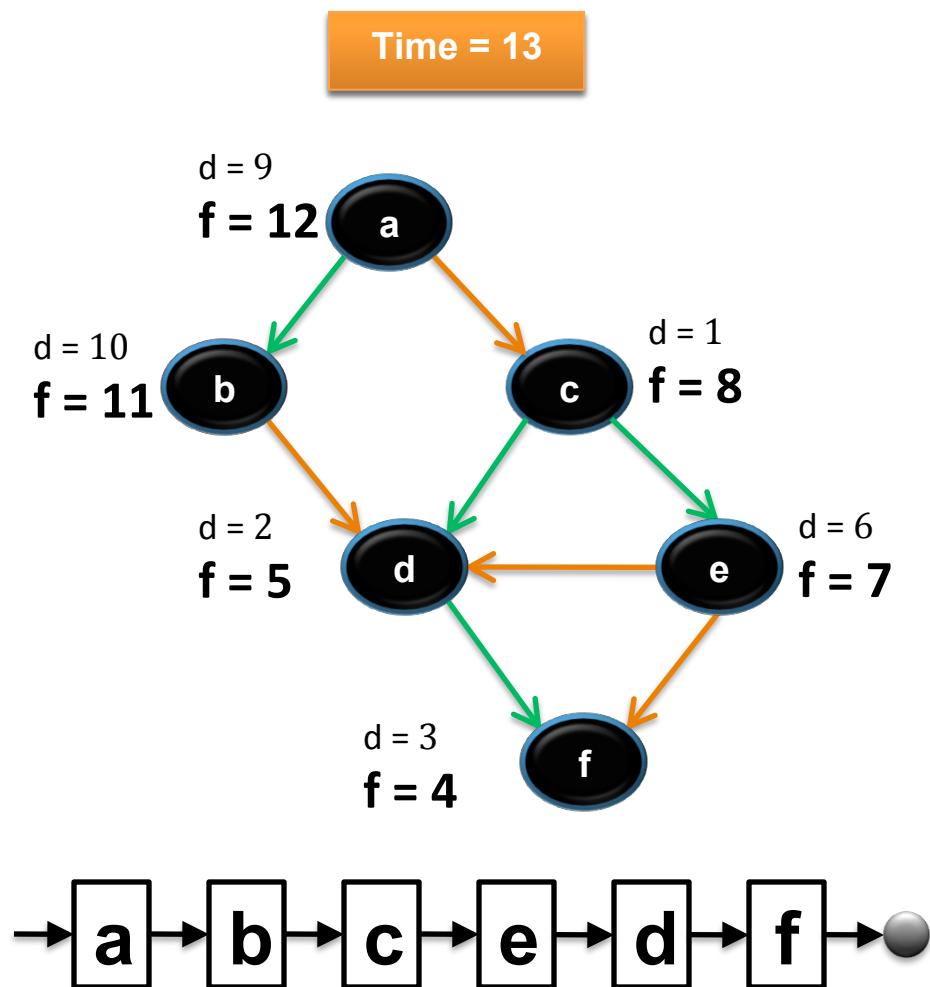
but **c** was already processed as (c,d) is a cross edge

Next we discover the vertex **b**

b is done as (b,d) is a cross edge => now move back to **c**

a is done as well

Topological sort



The linked list is sorted in **decreasing** order of finishing times $f[]$

Try yourself with different vertex order for DFS visit

Note: If you redraw the graph so that all vertices are in a line ordered by a valid topological sort, then all edges point „from left to right“

Time complexity of TS(G)

Running time of topological sort:

$$\Theta(n + m)$$

where $n=|V|$ and $m=|E|$

Why? Depth first search takes $\Theta(n + m)$ time in the worst case, and inserting into the front of a linked list takes $\Theta(1)$ time

Proof of correctness

- **Theorem:** $\text{TOPOLOGICAL-SORT}(\mathbf{G})$ produces a topological sort of a DAG \mathbf{G}
- The $\text{TOPOLOGICAL-SORT}(\mathbf{G})$ algorithm does a DFS on the DAG \mathbf{G} , and it lists the nodes of \mathbf{G} in order of decreasing finish times $f[]$
- We must show that this list satisfies the topological sort property, namely, that for every edge (u,v) of \mathbf{G} , u appears before v in the list
- **Claim:** For every edge (u,v) of \mathbf{G} : $f[v] < f[u]$ in DFS

Proof of correctness

“For every edge (u,v) of G , $f[v] < f[u]$ in this DFS”

The DFS classifies (u,v) as a **tree edge**, a **forward edge** or a **cross-edge** (it cannot be a back-edge since G has no cycles):

- i. If (u,v) is a **tree** or a **forward edge** $\Rightarrow v$ is a descendant of $u \Rightarrow f[v] < f[u]$
- ii. If (u,v) is a **cross-edge**

Proof of correctness

“For every edge (u,v) of G : $f[v] < f[u]$ in this DFS”

ii. If (u,v) is a **cross-edge**:

as (u,v) is a cross-edge, by definition, neither u is a descendant of v nor v is a descendant of u :

$$d[u] < f[u] < d[v] < f[v]$$

or

$$d[v] < f[v] < d[u] < f[u]$$

$$f[v] < f[u]$$

Q.E.D. of Claim

since (u,v) is an edge, v is surely discovered **before** u 's exploration completes

Proof of correctness

TOPOLOGICAL-SORT(G) lists the nodes of G from highest to lowest finishing times

By the **Claim**, for every edge (u, v) of G :

$$f[v] < f[u]$$

$\Rightarrow u$ will be before v in the algorithm's list

Q.E.D of Theorem

Recap: topological sorting

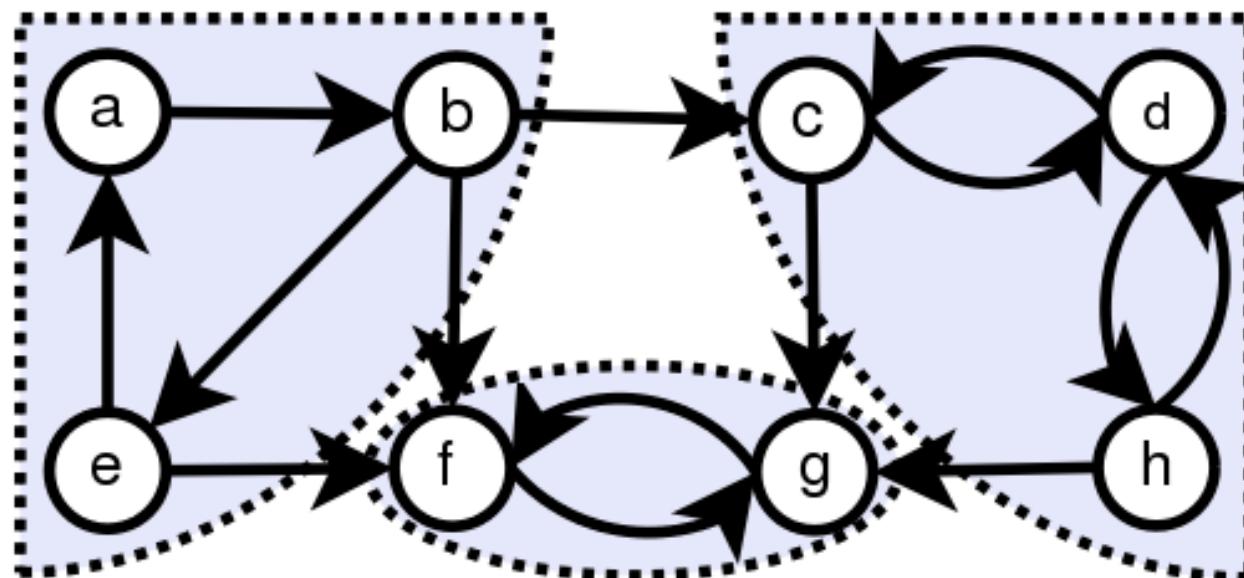
1.Do a DFS

2.Order vertices according to their finishing times $f[v]$

Strongly connected components

(covered in this week's tutorial)

→ Subgraphs with strong connectivity (any pair of vertices can reach each other)



Summary of DFS

- It's the twin of BFS (Queue vs Stack)
- Keeps two timestamps: $d[v]$ and $f[v]$
- Has same runtime as BFS
- Does NOT give us shortest-path
- Give us cycle detection (back edge)
- For real problems, choose BFS and DFS wisely.

Next week

→ Minimum Spanning Tree

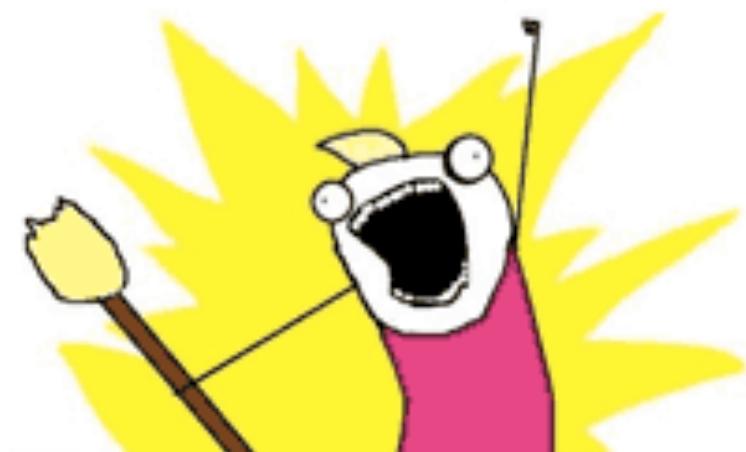


CSC263 Week 10

Announcements

Problem Set 5 is out (today)!

Due Tuesday (Dec 1)

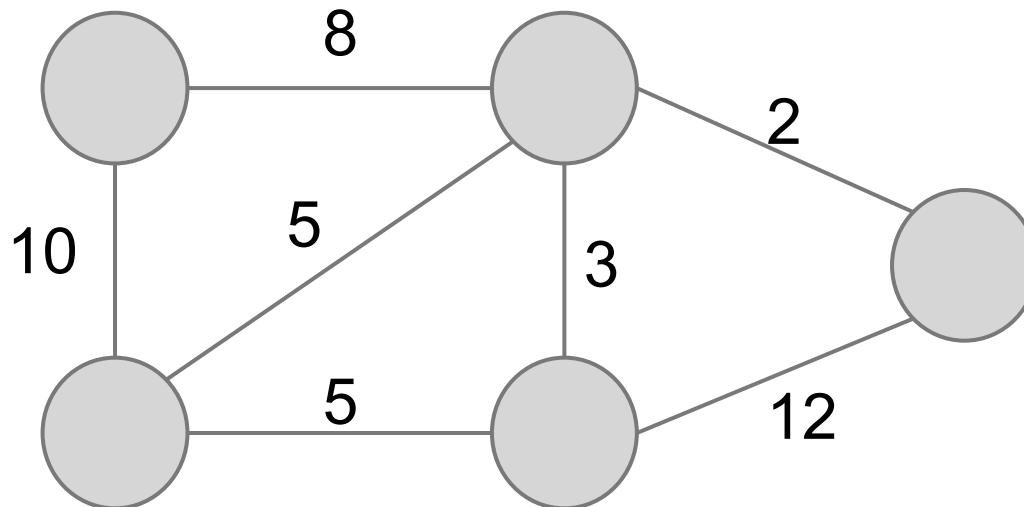


Minimum Spanning Trees

The Graph of interest today

A connected undirected **weighted** graph

$G = (V, E)$ with weights **w(e)** for each $e \in E$



It has the **smallest** total weight

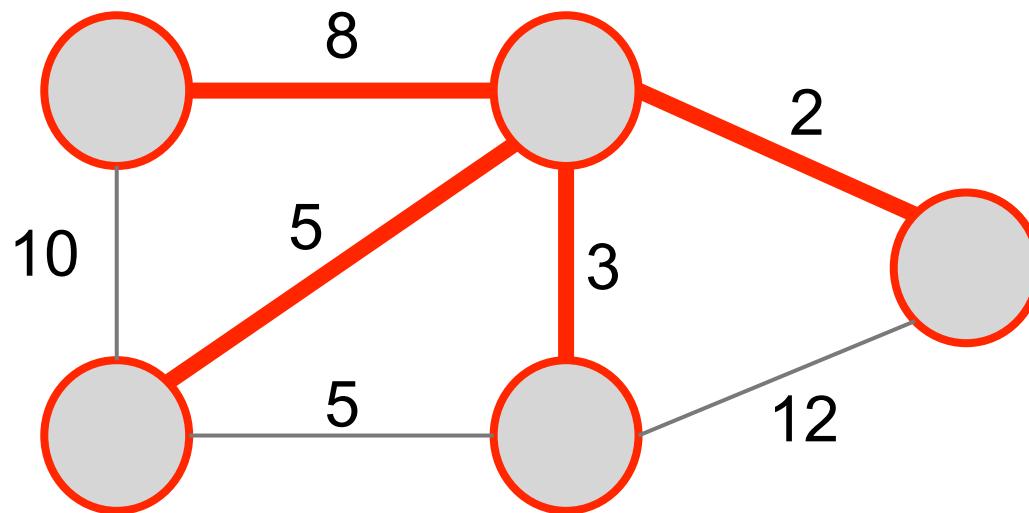
It covers **all** vertices in G

Minimum Spanning Tree

of graph G

It's a **connected**,
acyclic subgraph

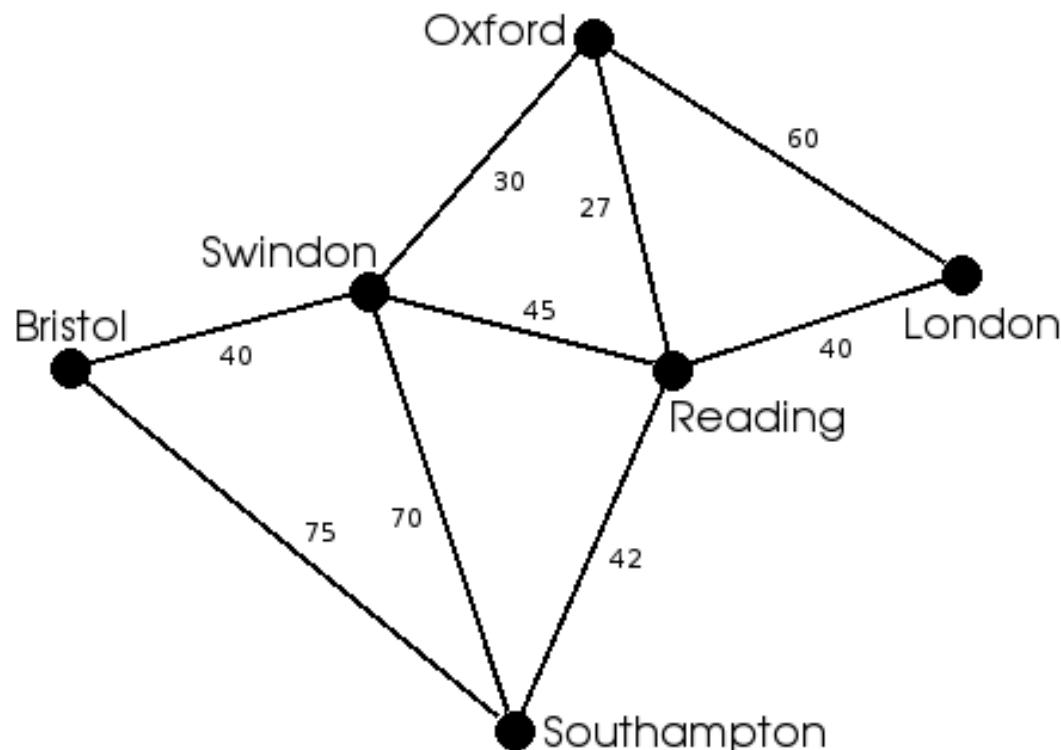
A Minimum Spanning Tree



May NOT be unique

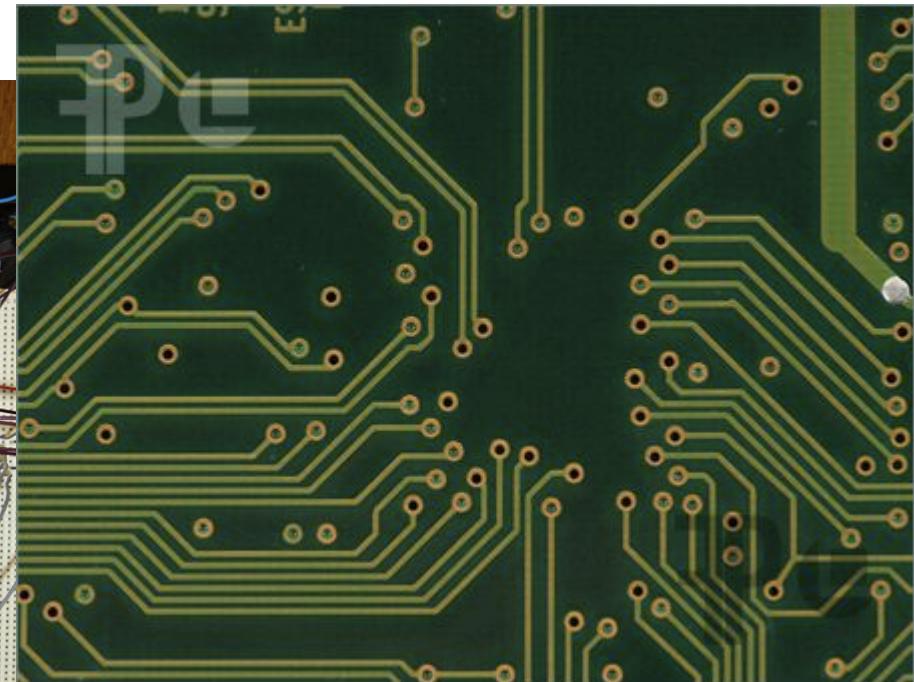
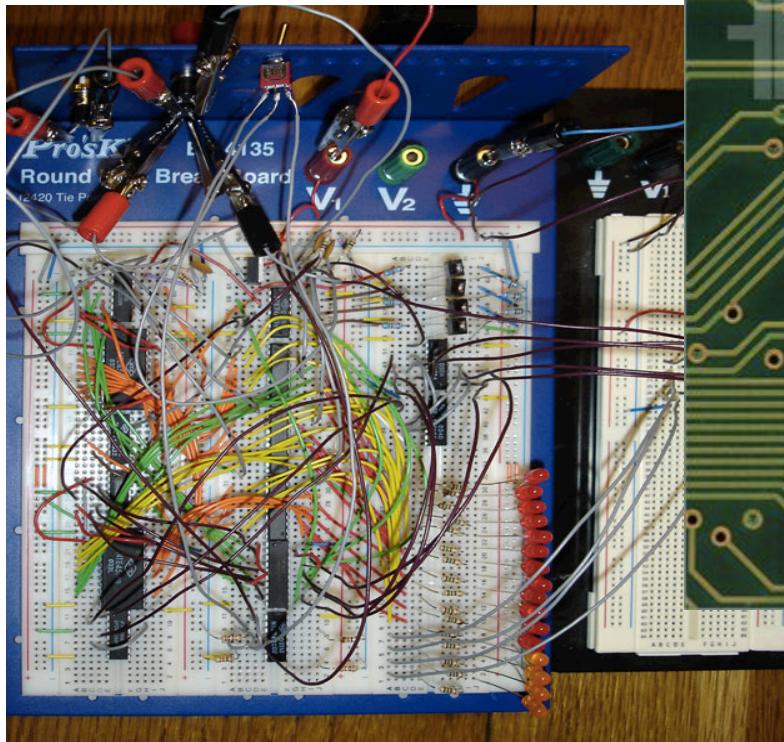
Applications of MST

Build a road network that connects all towns and with the minimum cost.



Applications of MST

Connect all components with the least amount of wiring.

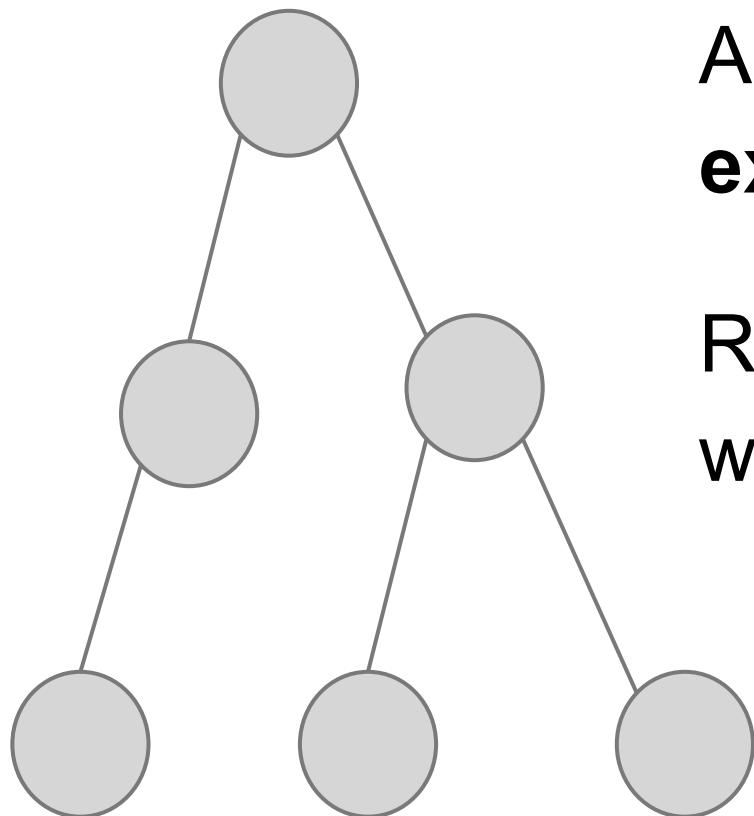


Other applications

- Cluster analysis
- Approximation algorithms for the “travelling salesman problem”
- ...

In order to understand
minimum spanning tree
we need to first understand
tree

Tree: undirected connected acyclic graph



A tree T with n vertices has
exactly $n-1$ edges.

Removing one edge from T
will **disconnect the tree**

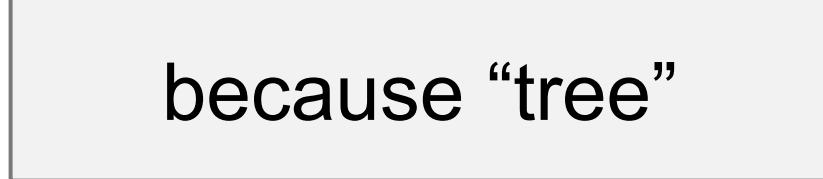
Adding one edge to T will
create a cycle.

The MST of a connected graph $G = (V, E)$
has $|V|$ vertices.



because “spanning”

The MST of a connected graph $G = (V, E)$
has $|V| - 1$ edges.



because “tree”

**Now we are ready to talk about
algorithms**

Idea #1

Start with $T = G.E$, then keep deleting edges until an MST remains.



**Which sounds more efficient
in terms of worst-case runtime?**

Idea #2

Start with **empty** T , then keep adding edges until an MST is built.

Hint

A undirected simple graph G with n vertices can have at most
_____ edges.

$$\binom{n}{2} = \frac{n(n - 1)}{2} \in \mathcal{O}(n^2)$$

Idea #1

Note: Here T is an edge set

Start with $T = G.E$, then keep deleting edges until an MST remains.

In worst-case, need to delete
 $O(|V|^2)$ edges $(n \text{ choose } 2) - (n-1)$

Idea #2

In worst-case, need to
add $O(|V|)$ edges

Start with **empty** T, then keep adding edges until an MST is built.

This is more efficient!

So, let's explore more of Idea #2,
i.e.,
building an MST by **adding** edges
one by one

i.e.,
we “grow” a tree



The generic growing algorithm

```
GENERIC-MST(G=(V, E, w)):  
    T ← ∅  
    while T is not a spanning tree:  
        find a “safe” edge e  
        T ← T ∪ {e}  
    return T
```

$|T| < |V|-1$

What is a “safe” edge?

“Safe” means it keeps the **hope** of T growing into an MST.

“Safe” edge e for T

Assuming **before** adding e , $T \subseteq \text{some MST}$,
edge **e** is safe if **after** adding **e**, still $T \subseteq \text{some MST}$

If we make sure T is always a subset of some MST while we grow it, then eventually T will become an MST!

```
GENERIC-MST( $G=(V, E, w)$ ):  
     $T \leftarrow \emptyset$   
    while  $T$  is not a spanning tree:  
        find a “safe” edge  $e$   
         $T \leftarrow T \cup \{e\}$   
    return  $T$ 
```



Intuition

If we make sure the pieces we put together is always a subset of the real picture while we grow it, then eventually it will become the real picture!

The generic growing algorithm

```
GENERIC-MST( $G = (V, E, w)$ ):
```

```
     $T \leftarrow \emptyset$ 
```

```
    while  $T$  is not a spanning tree:
```

```
        find a “safe” edge  $e$ 
```

```
         $T \leftarrow T \cup \{e\}$ 
```

```
    return  $T$ 
```

$|T| < |V|-1$

How to find a “safe” edge?

Two major algorithms we'll learn

→ Kruskal's algorithm



→ Prim's algorithm



**They are both based on
one theorem...**

How to find a safe edge: The cut property

Let $G=(V,E)$ be a connected, undirected graph.

X a subset of edges of G such that T contains X , where T is a minimum spanning tree of G . (So X is a forest and can be extended to a MST)

Let S be a connected component of (V,X) . (So no edge in X crosses the cut $S, V-S$)

Among all edges crossing between S and $V-S$, let e be an edge of minimum weight.

Then some MST T' contains $X+e$ (In other words, e is a safe edge.)

Basic outline of all MST algs:

Start with $G=(V,E,w)$

Let X be a set of edges, initially X is empty

Repeat until $|X| = |V|-1$:

1. Pick a connected component S of (V,X)
2. Let e be a lightest edge in E that crosses between S and $V-S$
3. Add e to X

Basic outline of all MST algs:

Start with $G=(V,E,w)$

Let X be a set of edges, initially E is empty

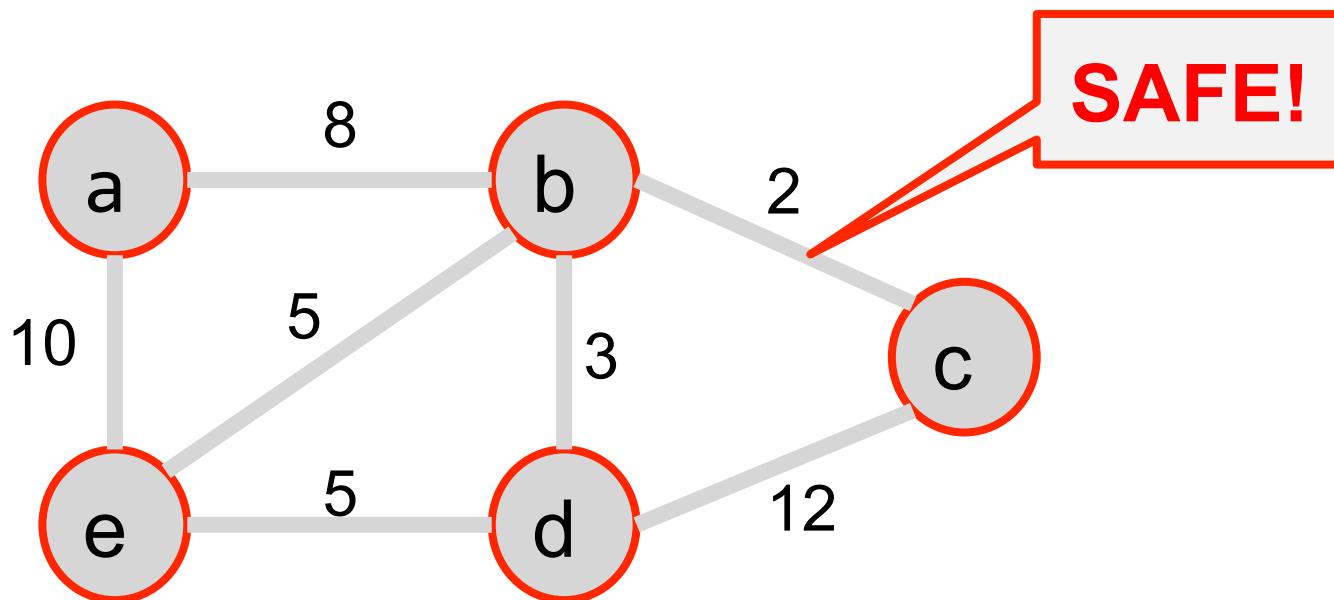
Repeat until $|X| = |V|-1$:

1. Pick a connected component S of (V,X)
2. Let e be a lightest edge in E that crosses between S and $V-S$
3. Add e to X

Prim: S starts off being a single vertex r , and in general S is the connected component containing r

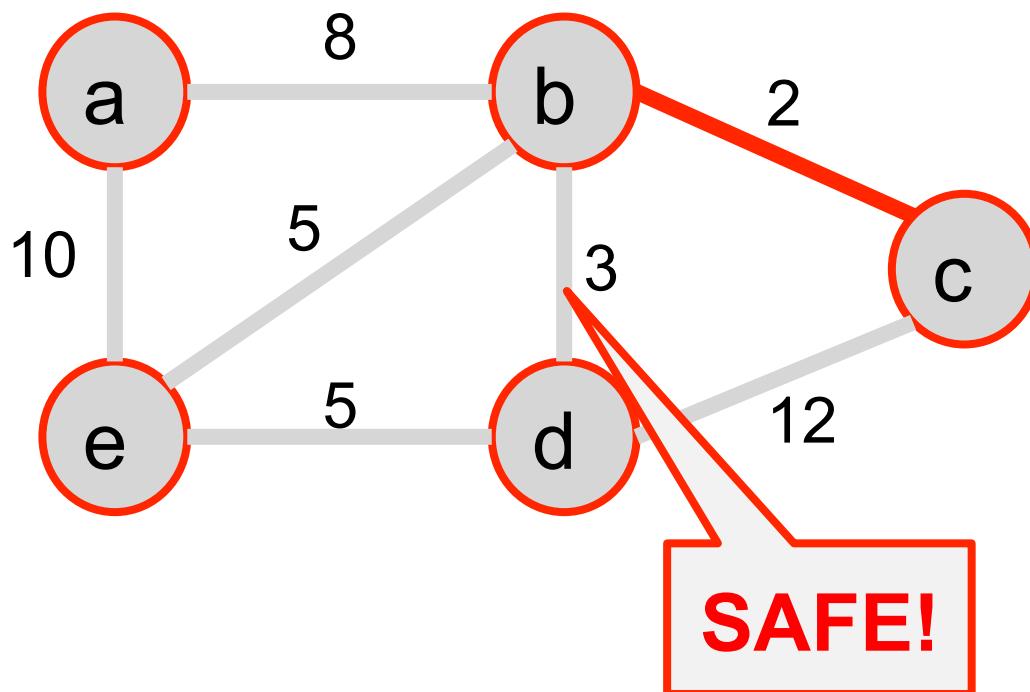
Kruskal: choose S so that the length of e is minimum

Initially, T (red) is a subgraph with no edge,
each vertex is a connected component,
all edges are **crossing** components,
and the minimum weighted one is ...

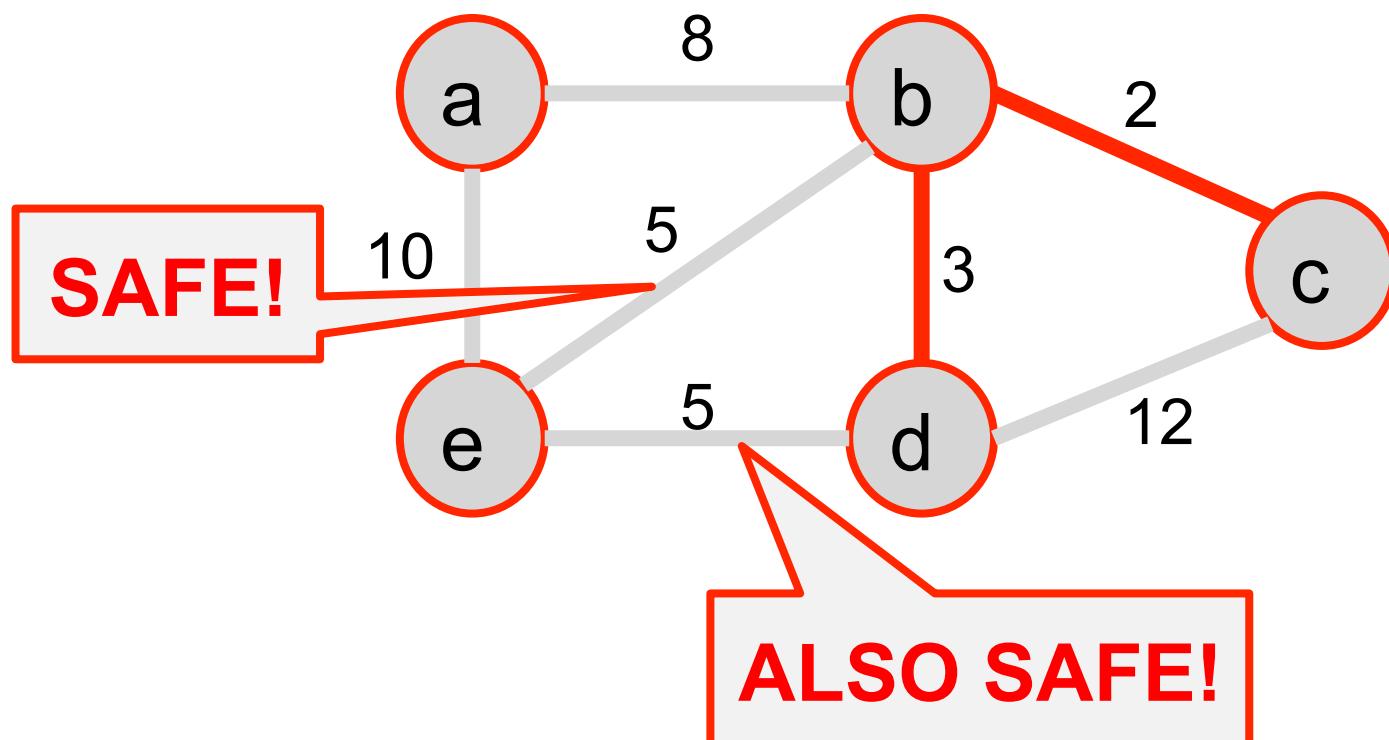


Now **b** and **c** in one connected component,
each of the other vertices is a component, i.e.,
4 components.

All gray edges are crossing components.

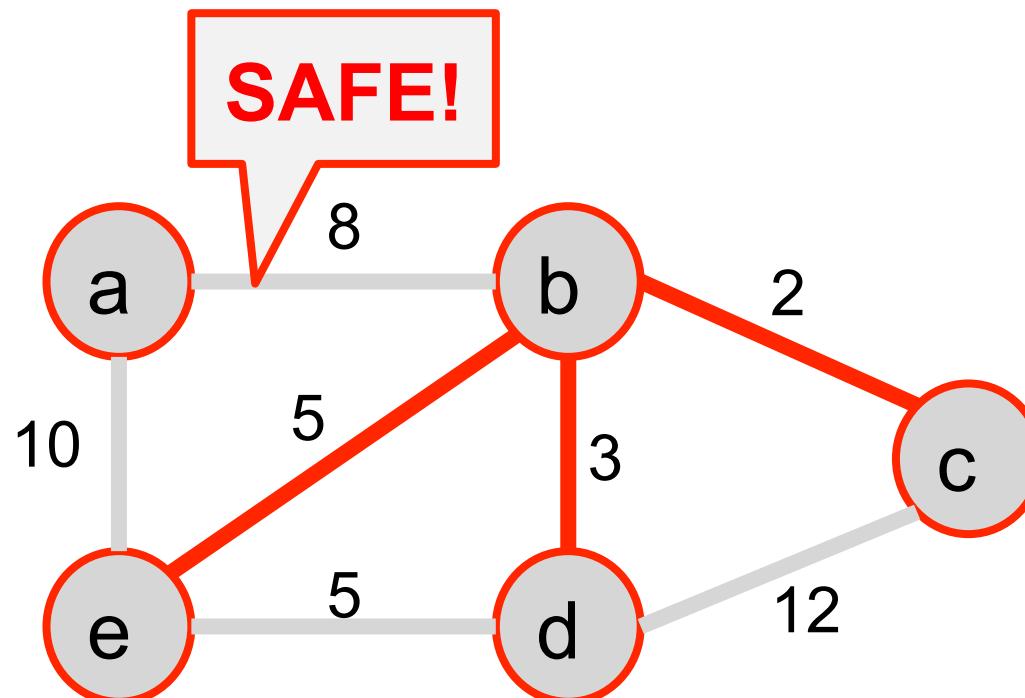


Now **b, c and d** are in one connected component, **a** and **e** each is a component.
(c, d) is NOT crossing components!

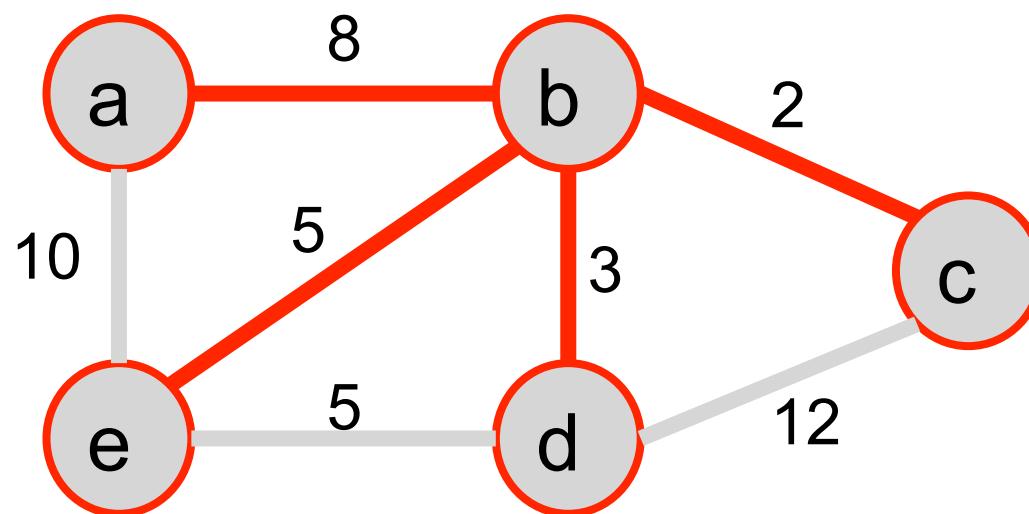


Now **b, c, d and e** are in one connected component, **a** is a component.

(a, e) and **(a, b)** are crossing components.



MST grown!



Two things that need to be worried about when actually implementing the algorithm

→ How to keep track of the **connected components?**

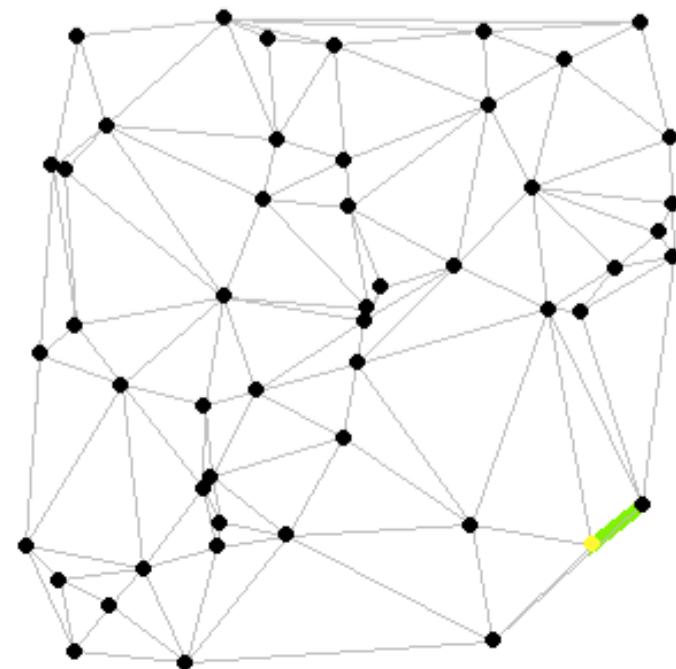
→ How to efficiently find the **minimum weighted edge?**

Kruskal's and Prim's basically use different **data structures** to do these two things.

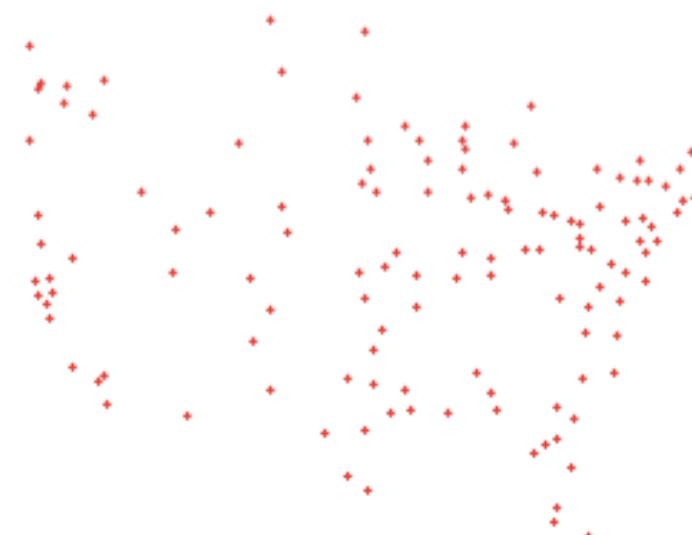
Overview: Prim's and Kruskal's

	Keep track of connected components	Find minimum weight edge
Prim's	Keep “one tree plus isolated vertices”	<i>use priority queue ADT</i>
Kruskal's	<i>use “disjoint set” ADT</i>	<i>Sort all edges according to weight</i>

Prim's



Kruskal's



<https://trendsofcode.files.wordpress.com/2014/09/dijkstra.gif>

https://www.projectrhea.org/rhea/images/4/4b/Kruskal_Old_Kiwi.gif

Prim's MST algorithm

Prim's algorithm: Idea

- Start from an arbitrary vertex as root
- Focus on growing **one** tree. This tree is one **component**; the cut is always $(T, V-T)$ where T is the tree so far.)
- Choose a minimum weight edge among all edges that are **incident to the current tree** (**edges crossing the cut**)
- How to get that minimum? Store all candidate vertices in a **Min-Priority Queue** whose key is the weight of the **crossing** edge (incident to tree).

PRIM-MST($G=(V, E, w)$):

```
1  T ← {}
2  for all  $v$  in  $V$ :
3      key[ $v$ ] ←  $\infty$ 
4      pi[ $v$ ] ← NIL
5  pick arbitrary vertex  $r$  as root;  $key[r]=0$ 
6  initialize priority queue  $Q$  with all  $v$  in  $V$ 
7   $T \leftarrow \{r\}$ 
8  while  $Q$  is not empty:
9       $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
10     if  $pi[u] \neq \text{NIL}$ :
11          $T \leftarrow T \cup \{(pi[u], u)\}$ 
12     for each neighbour  $v$  of  $u$ :
13         if  $v$  in  $Q$  and  $w(u, v) < key[v]$ :
14             DECREASE-KEY( $Q, v, w(u, v)$ )
15              $pi[v] \leftarrow u$ 
```

key[v] keeps the “shortest distance” between v and the current tree

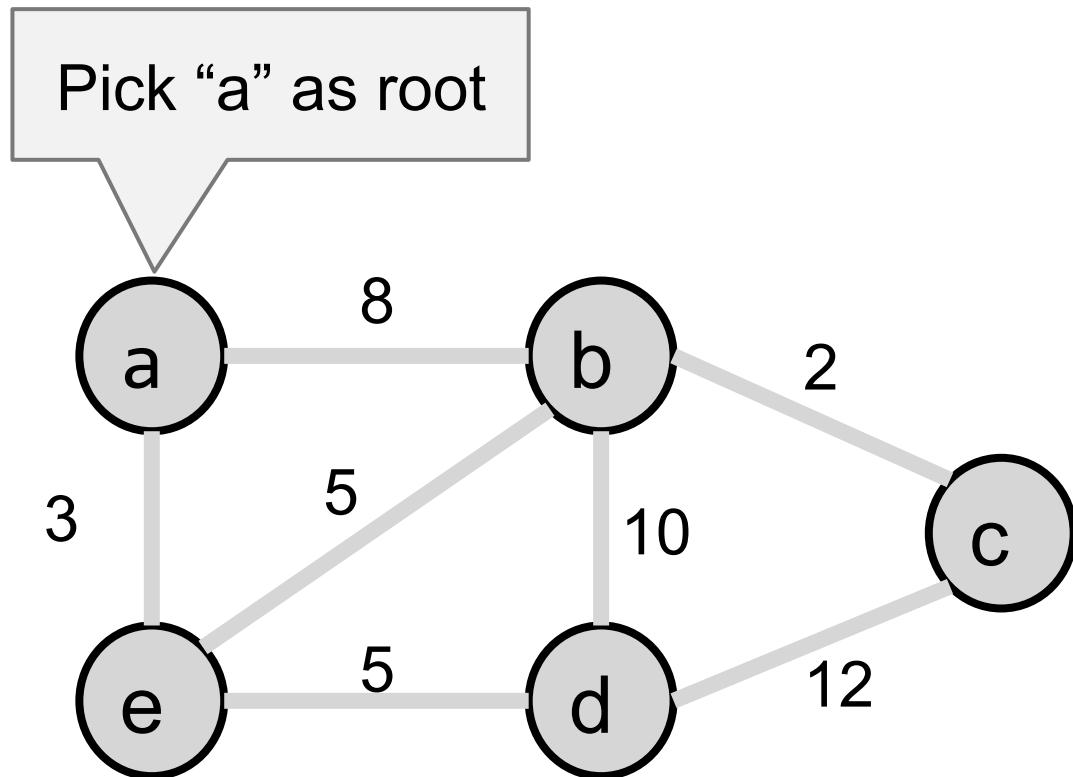
pi[v] keeps who, in the tree, is v connected to via lightest edge.

u is the next vertex to add to current tree

add edge, $pi[u]$ is lightest vertex to connect to, “safe”

all u 's neighbours' distances to the current tree need update

Trace an example!

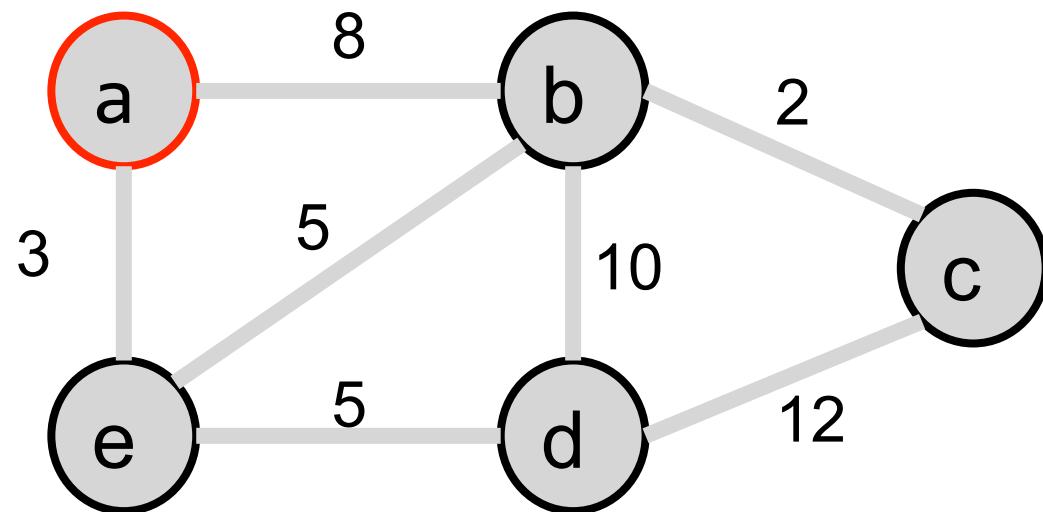


Next, ExtractMin !

Q	key	pi
a	0	NIL
b	∞	NIL
c	∞	NIL
d	∞	NIL
e	∞	NIL

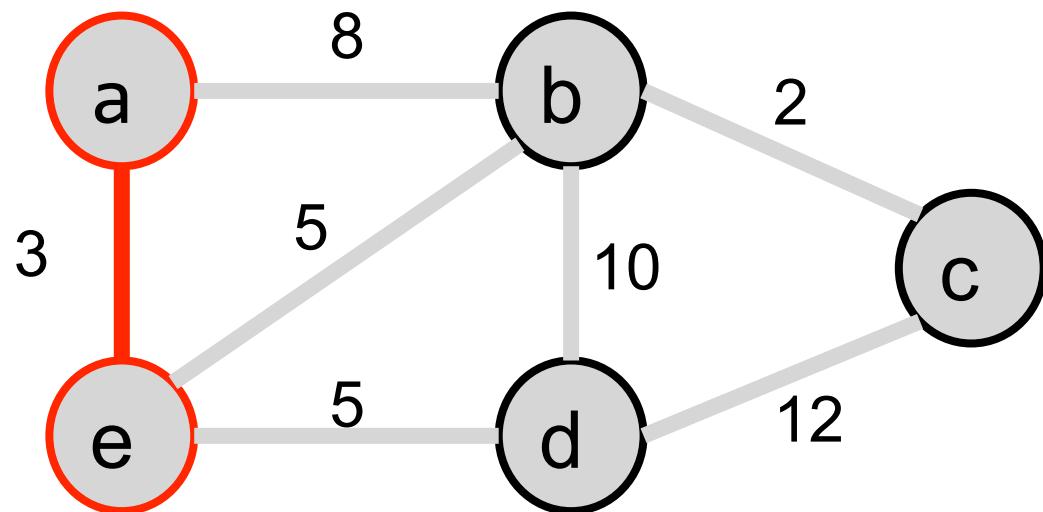
ExtractMin (#1)
then update neighbours' keys

a: 0, NIL



Q	key	pi
b	$\infty \rightarrow 8$	$\text{NIL} \rightarrow a$
c	∞	NIL
d	∞	NIL
e	$\infty \rightarrow 3$	$\text{NIL} \rightarrow a$

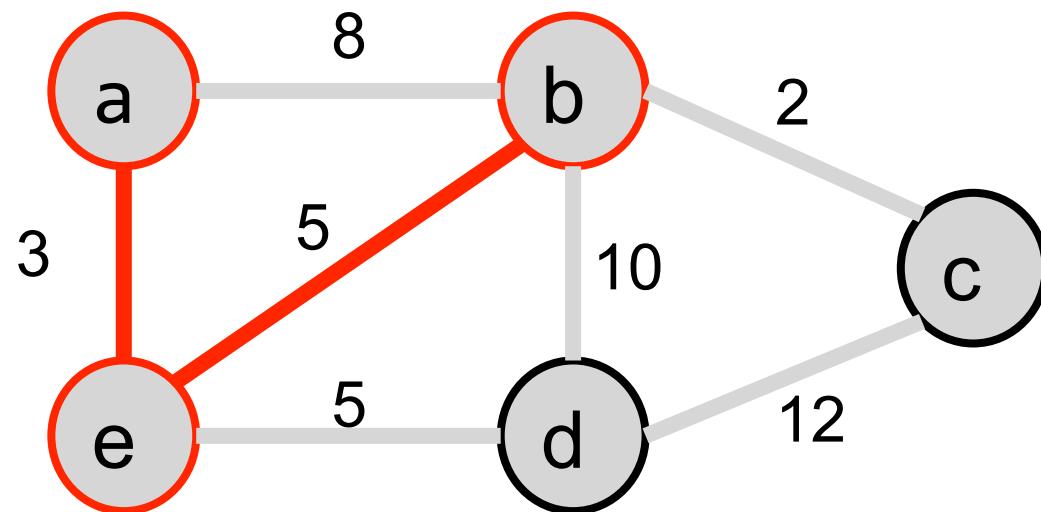
ExtractMin (#2)
then update neighbours' keys



e: 3, a

Q	key	pi
b	8→5	a →e
c	∞	NIL
d	∞ →5	NIL →e

ExtractMin (#3)
then update neighbours' keys



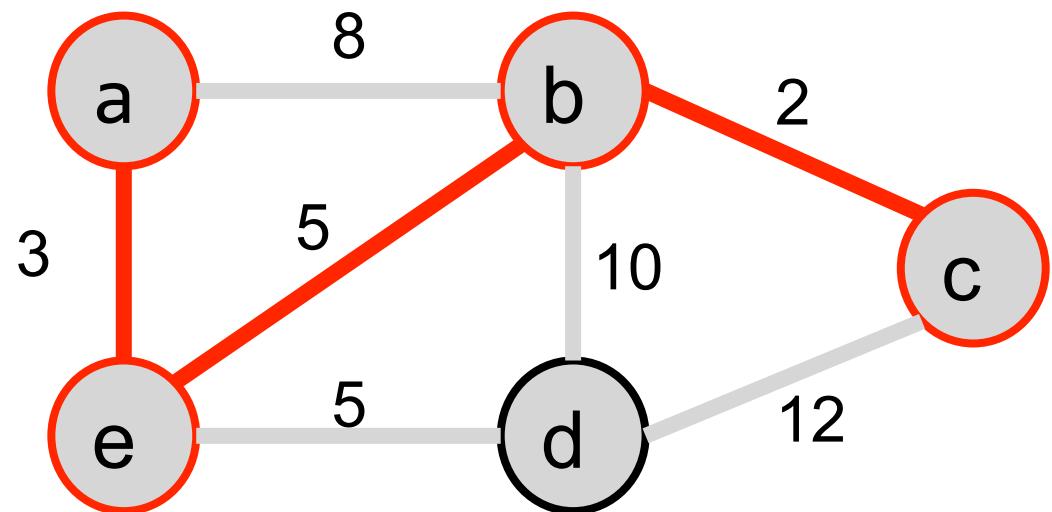
b: 5, e

Q	key	pi
c	$\infty \rightarrow 2$	NIL $\rightarrow b$
d	5	e

Could also have extracted d
since its key is also 5 (min)

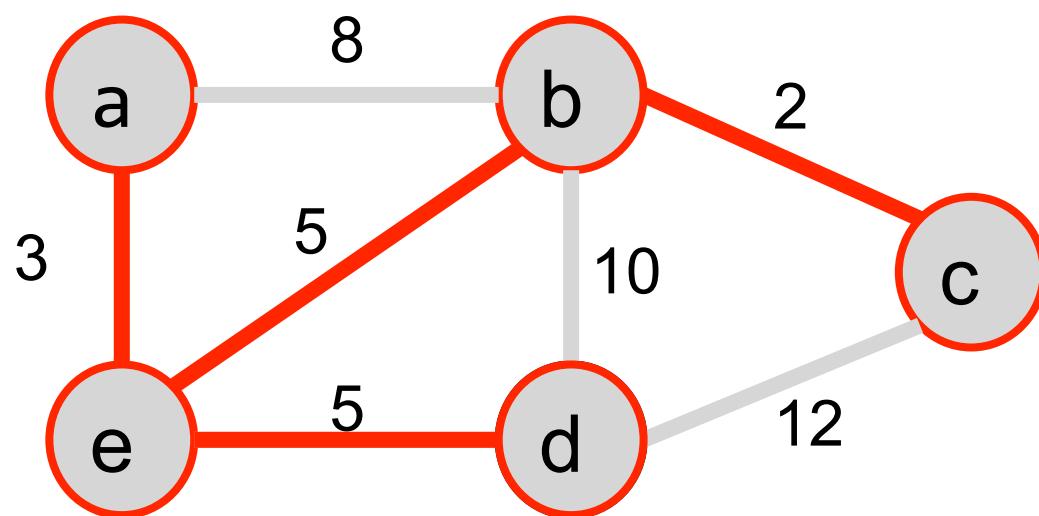
ExtractMin (#4)
then update neighbours' keys

c: 2, b



Q	key	pi
d	5	e

ExtractMin (#4)
then update neighbours' keys

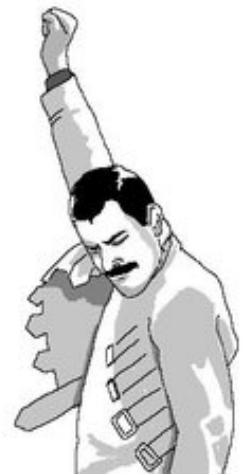


d: 5, e

Q	key	pi
---	-----	----

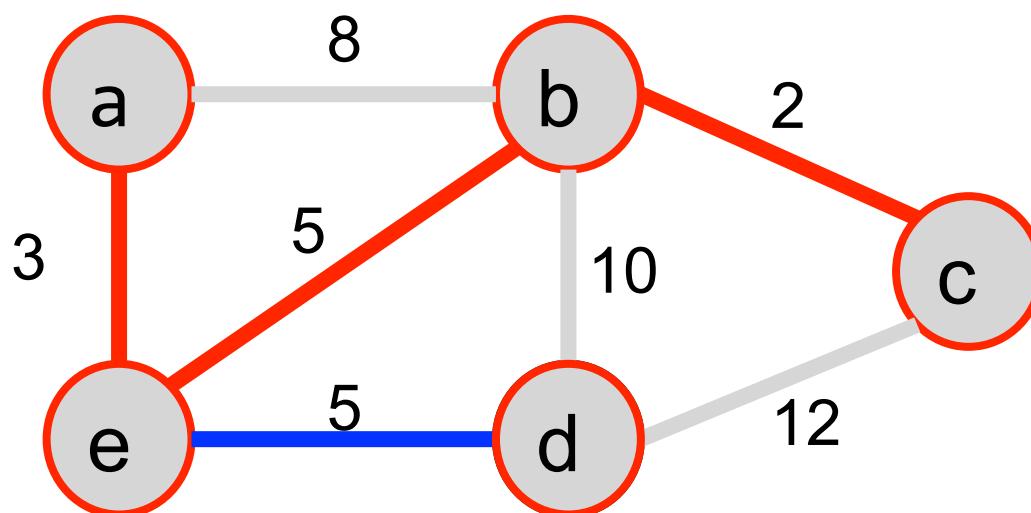
Q is empty now.

MST grown!



Correctness of Prim's

The added edge is always a “**safe**” edge, i.e., the **minimum** weight edge crossing the cut (because of **ExtractMin**).



Runtime analysis: Prim's

- Assume we use **binary min heap** to implement the priority queue.
- Each ExtractMin takes **$O(\log V)$**
- In total V ExtractMin's
- In total, check at most **$O(E)$** neighbours, each check neighbour could lead to a **DecreaseKey** which takes **$O(\log V)$**
- **TOTAL: $O((V+E)\log V) = O(E \log V)$**

In a connected graph $G = (V, E)$

$|V|$ is in $O(|E|)$ because...

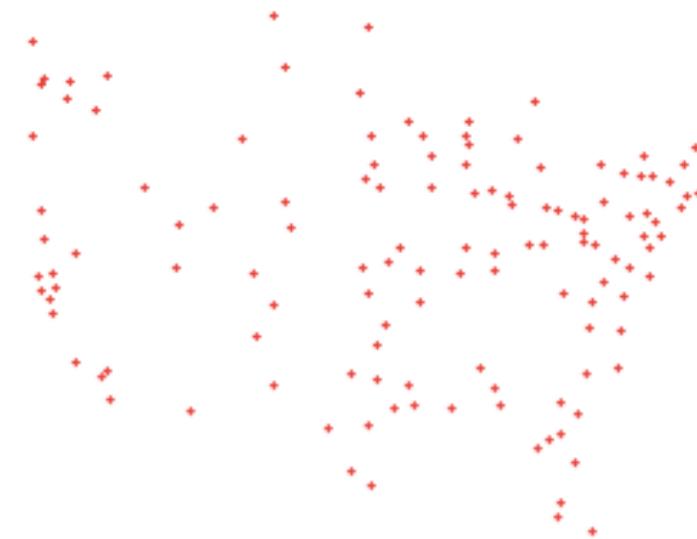
$|E|$ has to be at least $|V|-1$

Also, $\log |E|$ is in $O(\log |V|)$ because ...

E is at most V^2 ,

so $\log E$ is at most $\log V^2 = 2 \log V$, which is in $O(\log V)$

Kruskal's MST algorithm



Kruskal's algorithm: idea

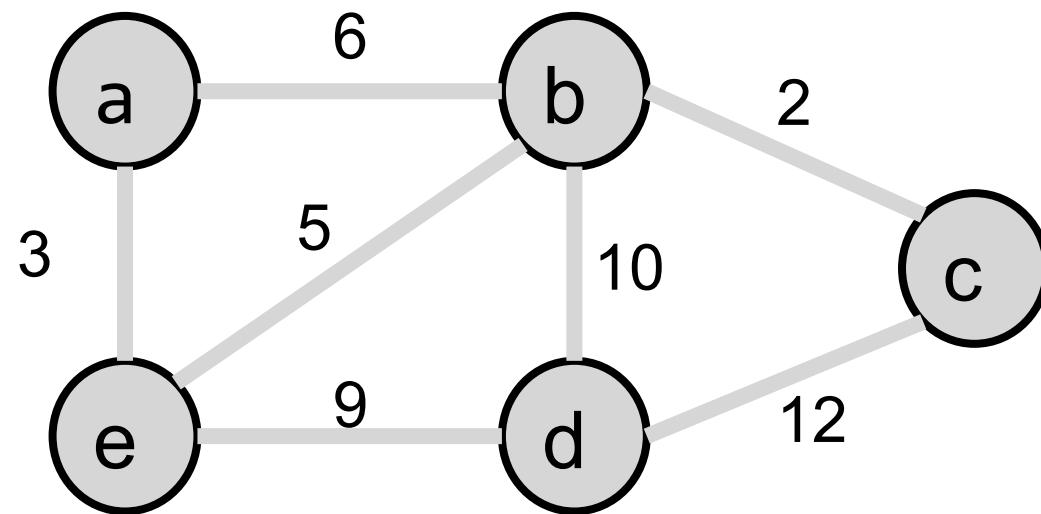
- Sort all edges according to **weight**, then start adding to MST from the **lightest** one.
 - ◆ This is “greedy”!
- Constraint: added edge must NOT cause a **cycle**
 - ◆ In other words, the two endpoints of the edge must belong to two **different** trees (components).
- The whole process is like unioning small trees into a big tree.

Pseudocode

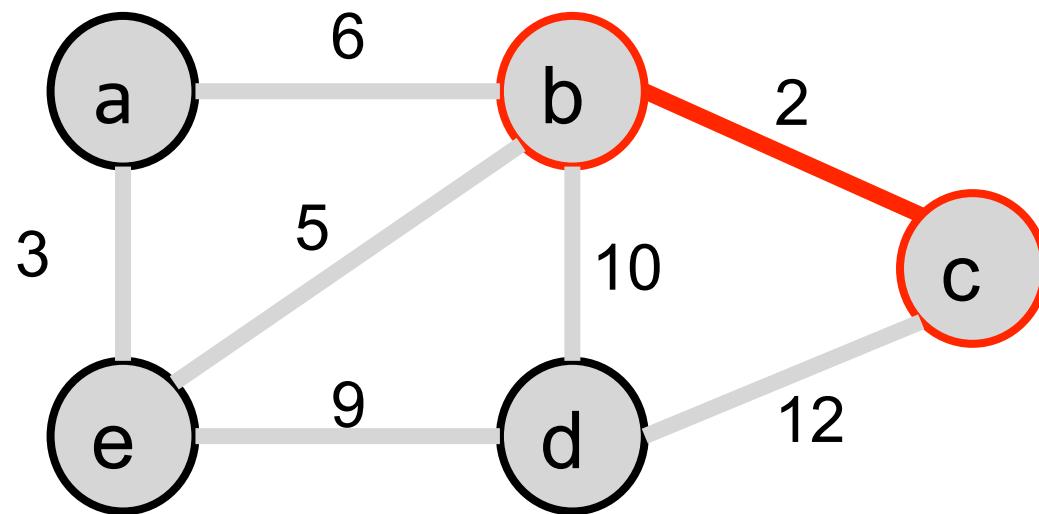
$$m = |E|$$

```
KRUSKAL-MST(G(V, E, w)):  
1   T ← {}  
2   sort edges so that w(e1) ≤ w(e2) ≤ ... ≤ w(em)  
3   for i ← 1 to m:  
4       # let (ui, vi) = ei  
5       if ui and vi in different components:  
6           T ← T ∪ {ei}
```

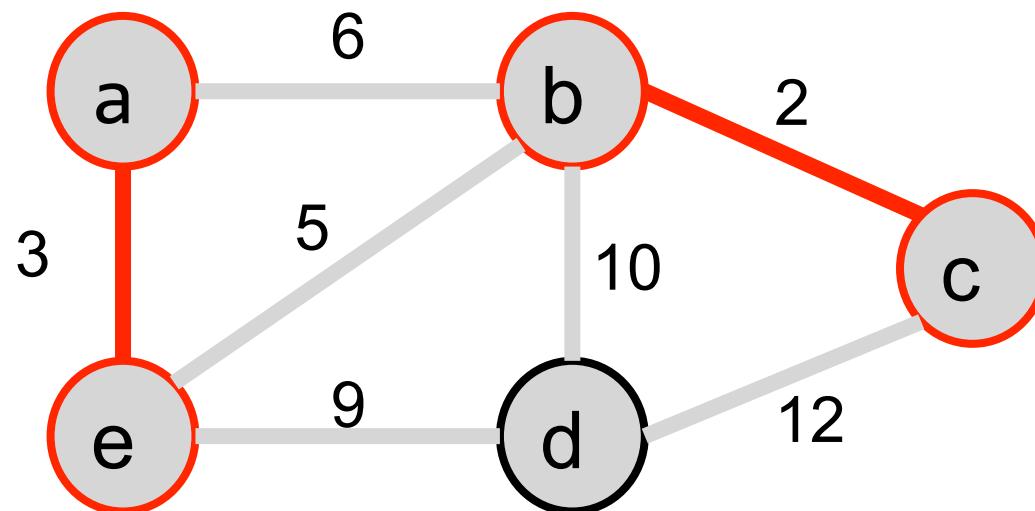
Example



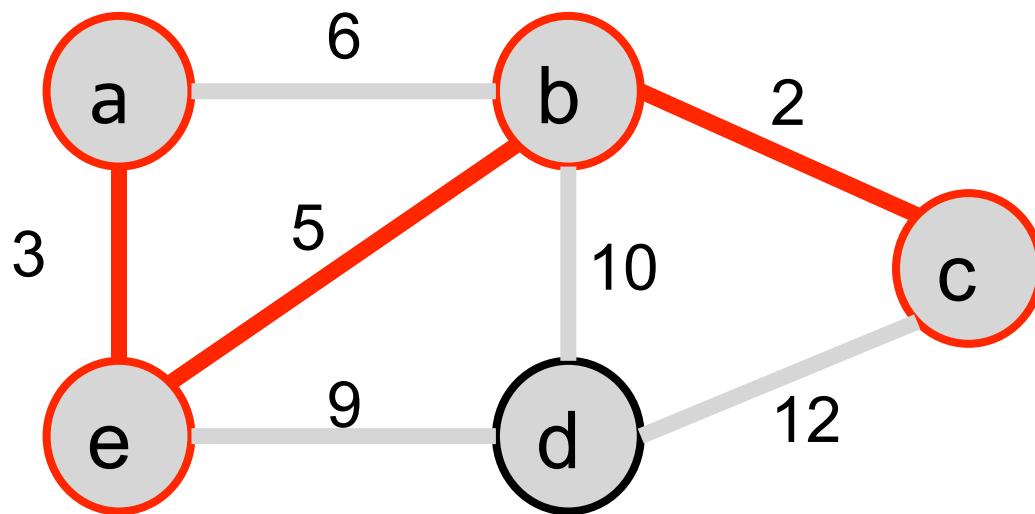
Add (b, c), the lightest edge



Add (a, e), the 2nd lightest

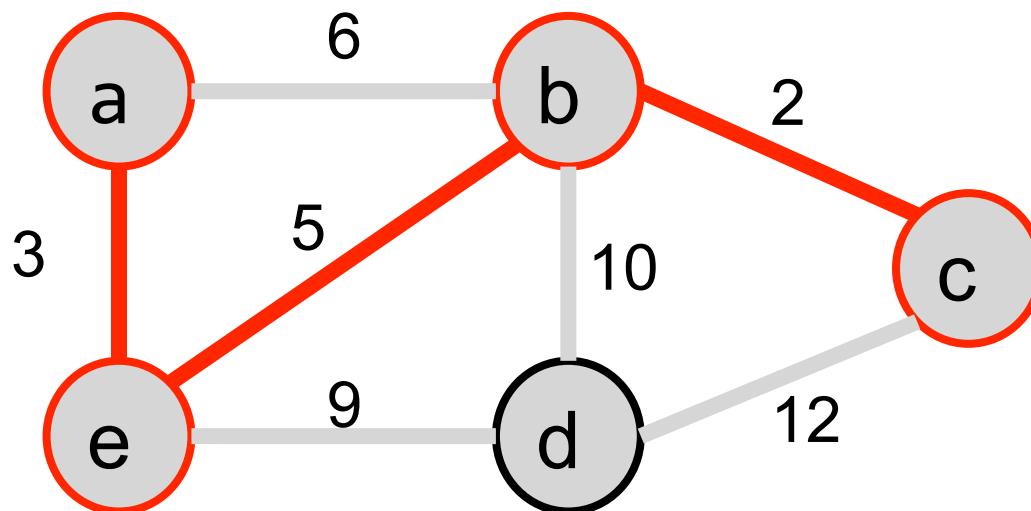


Add (b, e), the 3rd lightest



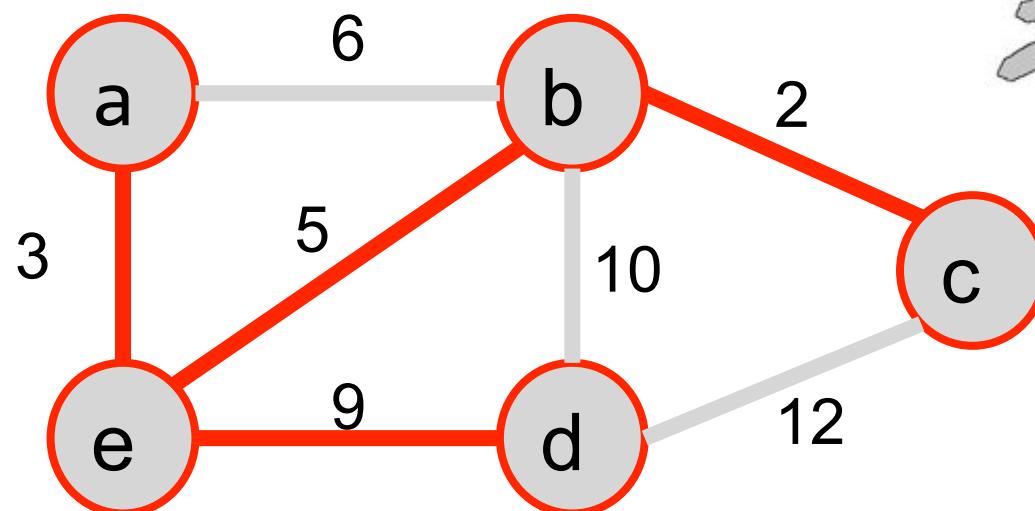
Add (a, b), the 4th lightest ...

**No! a, b are in the same component
Add (d, e) instead!**



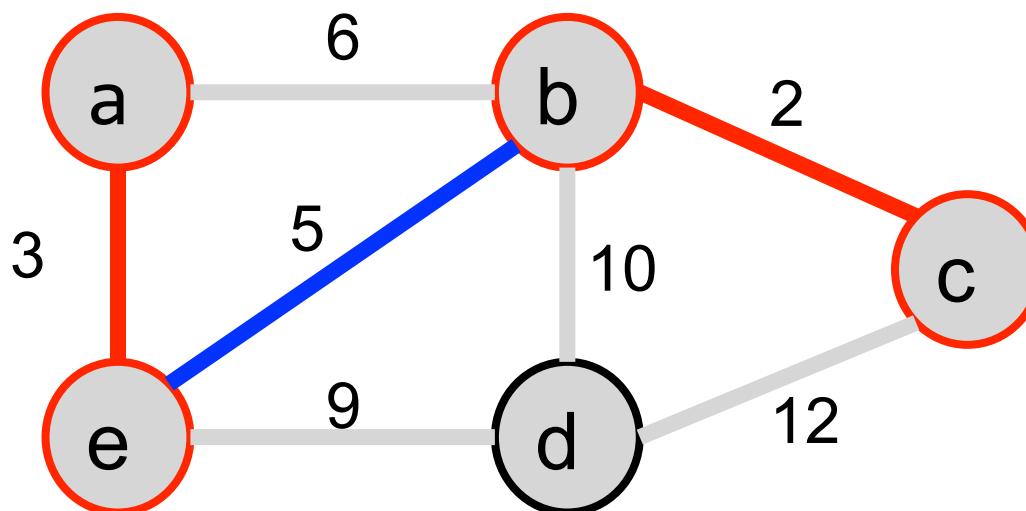
Add (d, e) ...

MST grown!



Correctness of Kruskal's

The added edge is always a “**safe**” edge, because it is the **minimum** weight edge among all edges that **cross** components



Runtime ...

$$m = |E|$$

sorting takes $O(E \log E)$

KRUSKAL-MST($G(V, E, w)$):

```
1   T ← {}
2   sort edges so that  $w(e_1) \leq w(e_2) \leq \dots \leq w(e_m)$ 
3   for  $i \leftarrow 1$  to  $m$ :
4       # let  $(u_i, v_i) = e_i$ 
5       if  $u_i$  and  $v_i$  in different components:
6            $T \leftarrow T \cup \{e_i\}$ 
```

How exactly do we do these two lines?

We need the **Disjoint Set ADT**

which stores a collections of nonempty disjoint sets **S₁, S₂, ..., S_k**, each has a “representative”.

and supports the following operations

→ **MakeSet(x)**: create a new set {x}

→ **FindSet(x)**: return the representative of the set that x belongs to

→ **Union(x, y)**: union the two sets that contain x and y, if different

$$m = |E|$$

Real Pseudocode

```
KRUSKAL-MST(G(V, E, w)):
```

```
1   T ← {}
2   sort edges so that w(e1)≤w(e2)≤...≤w(em)
3   for each v in V:
4       MakeSet(v)
5   for i ← 1 to m:
6       # let (ui, vi) = ei
7       if FindSet(ui) != FindSet(vi):
8           Union(ui, vi)
9       T ← T ∪ {ei}
```

Next week

→ More on Disjoint Set



CSC263 Week 11

Announcements

Problem Set 5 (the last one!!) is due this Tuesday (Dec 1)

Problem Set 4 is graded. Average=77%



ADT: Disjoint Sets

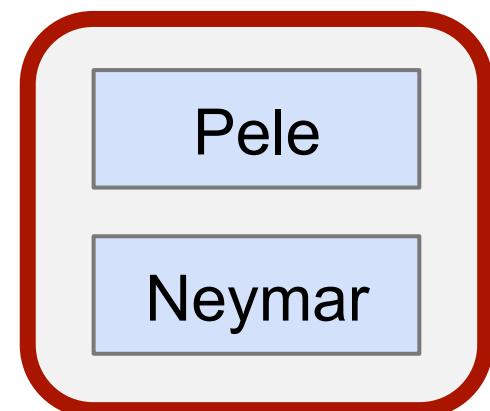
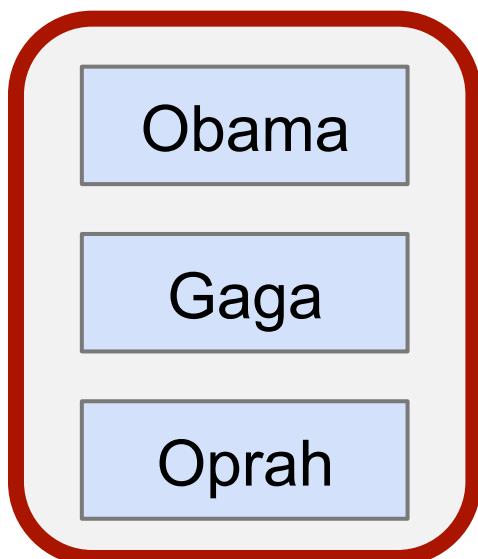
- What does it store?
- What operations are supported?

What does it store?

The elements in the sets can change dynamically.

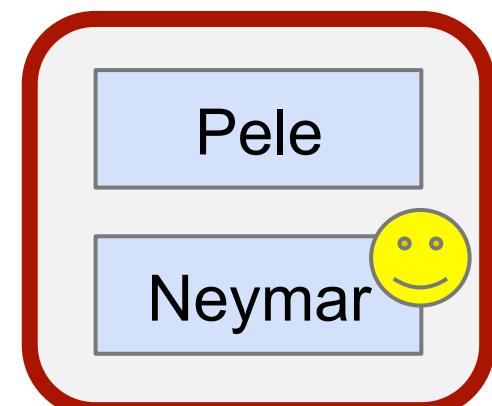
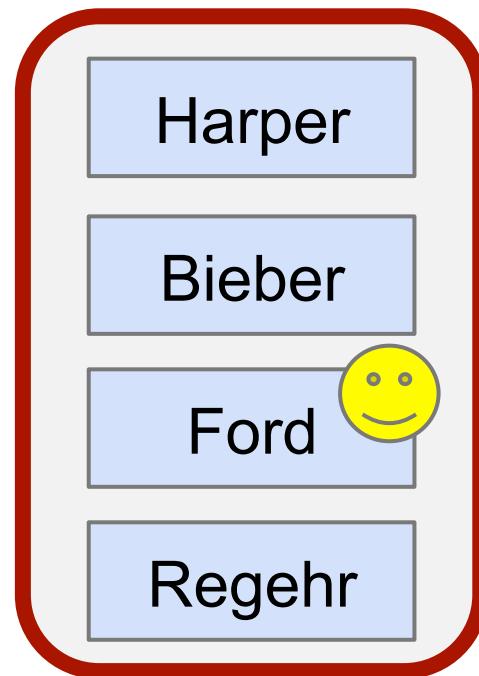
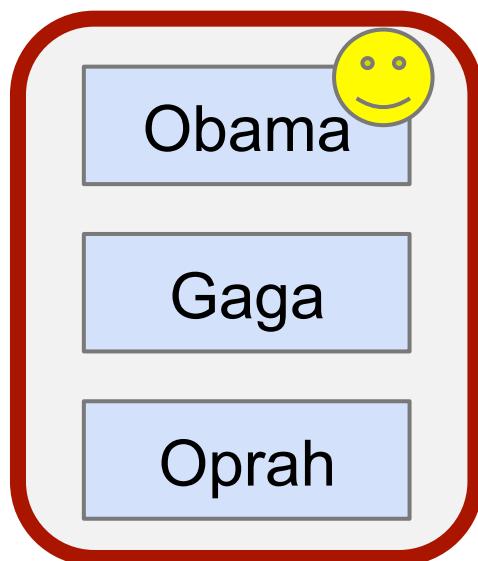
It stores a collection of (**dynamic**) **sets** of elements, which are **disjoint** from each other.

Each element belongs to **only one** set.



Each set has a **representative**

A set is **identified** by its representative.



Operations

MakeSet(x): Given an element x that does NOT belong to any set, create a new set $\{x\}$, that contains only x , and assign x as the representative.

MakeSet("Newton")



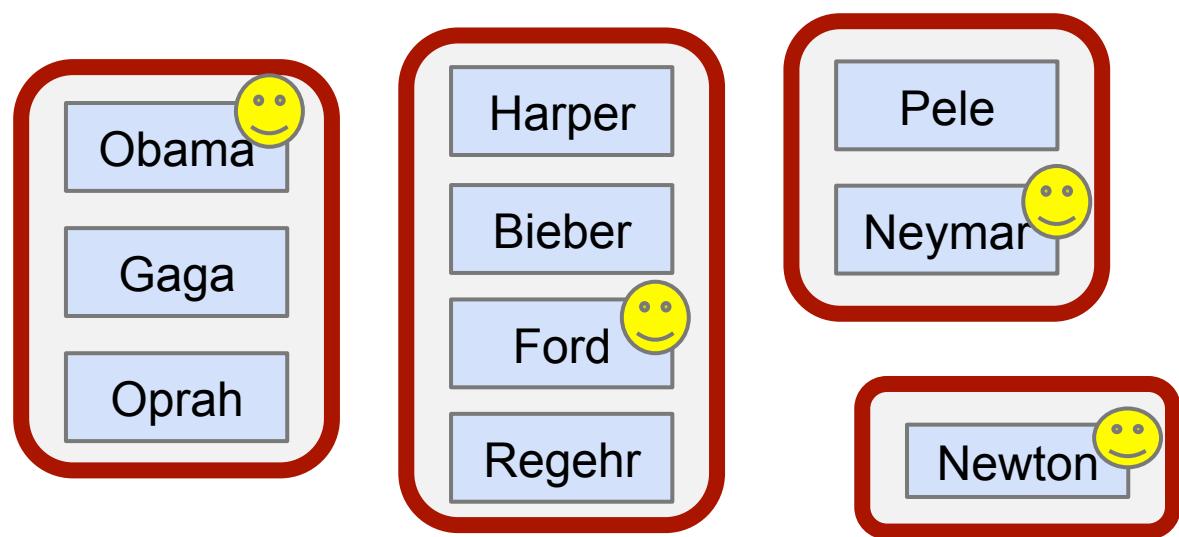
Operations

FindSet(x): return the representative of the set that contains x.

FindSet("Bieber") returns: **Ford**

FindSet("Oprah") returns: **Obama**

FindSet("Newton")
returns: **Newton**

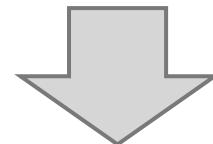
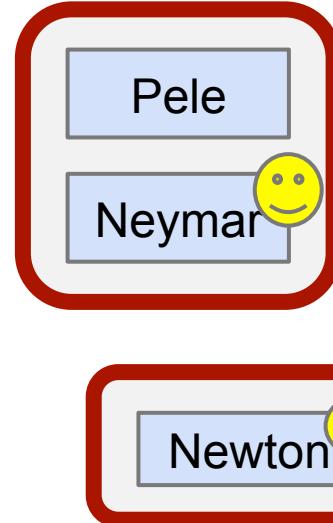
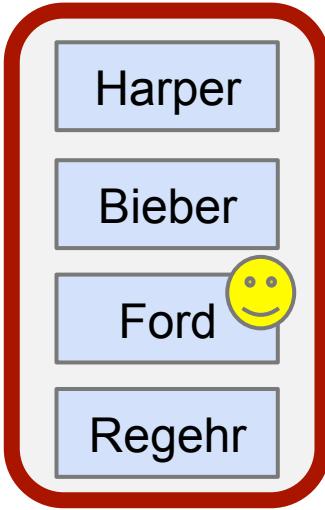
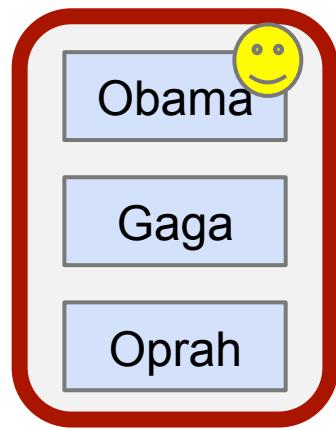


Operations

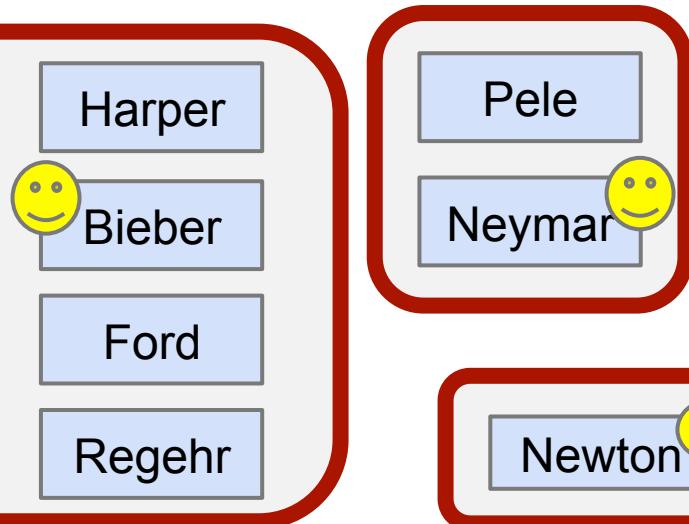
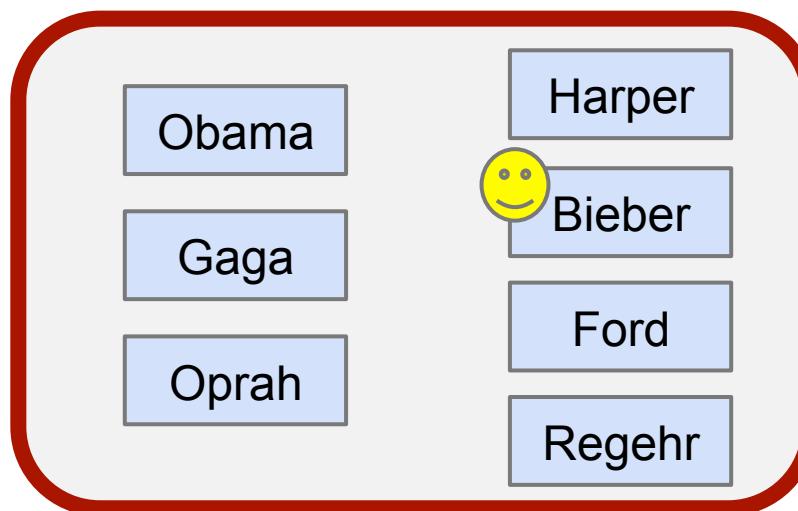
If x and y are already in the **same** set, then nothing happens.

Union(x, y): given two elements x and y , create a **new set** which is the **union** of the two sets that contain x and y , **delete** the original sets that contains x and y .

Pick a **representative** of the new set, usually (but not necessarily) one of the representatives of the two original sets.



Union("Gaga", "Harper")



Applications

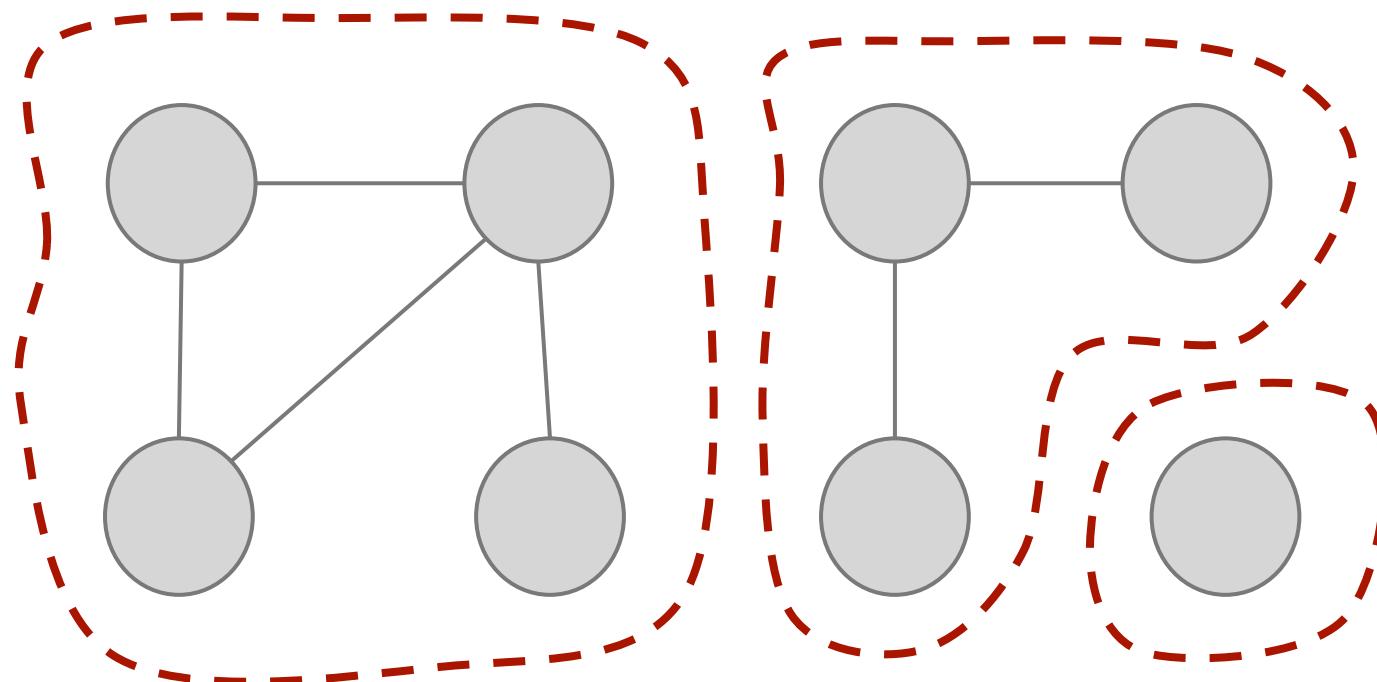
KRUSKAL-MST(G(V, E, w)):

```
1  T ← {}
2  sort edges so that w(e1)≤w(e2)≤...≤w(em)
3  for each v in V:
4      MakeSet(v)
5  for i ← 1 to m:
6      # let (ui, vi) = ei
7      if FindSet(ui) != FindSet(vi):
8          Union(ui, vi)
9      T ← T ∪ {ei}
```

Other applications

```
For each edge (u, v)  
if FindSet(u) != FindSet(v),  
then Union(u, v)
```

Finding connected components of a graph



Summary: the ADT

- Stores a collection of disjoint sets
- Supported operations
 - ◆ MakeSet(x)
 - ◆ FindSet(x)
 - ◆ Union(x, y)

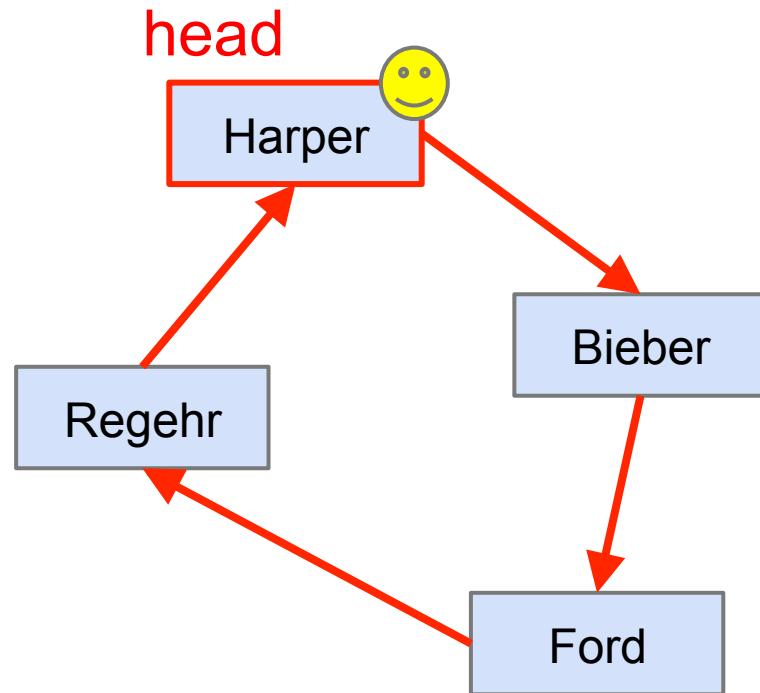
How to **implement the
Disjoint Sets ADT (efficiently) ?**

Ways of implementations

- 1.Circularly-linked lists
- 2.Linked lists with extra pointer
- 3.Linked lists with extra pointer and with union-by-weight
- 4.Trees
- 5.Trees with union-by-rank
- 6.Trees with path-compression
- 7.Trees with union-by-weight and path-compression

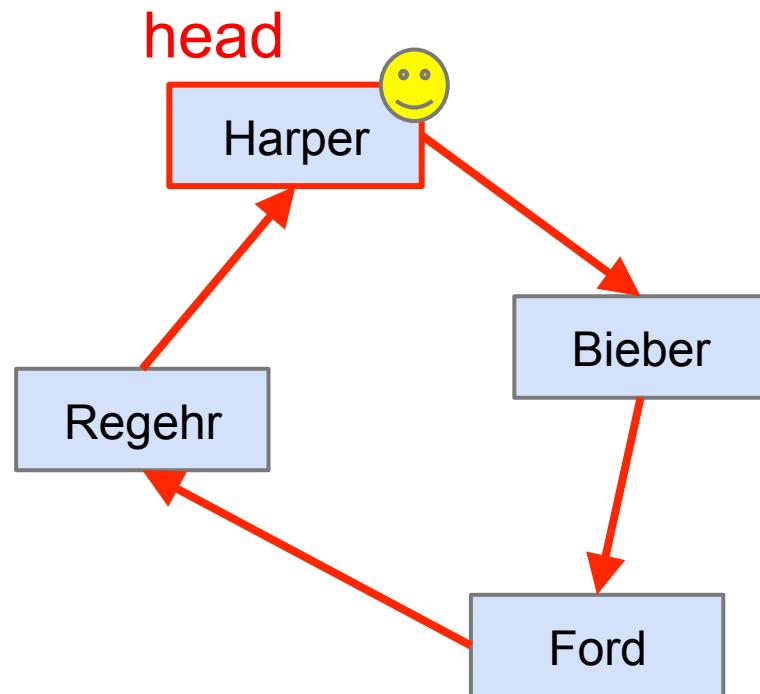
Circularly-linked list

Circularly-linked list



- One circularly-linked list per set
- Head of the linked list also serves as the representative.

Circularly-linked list



→ **MakeSet(x):** just a new linked list with a single element x

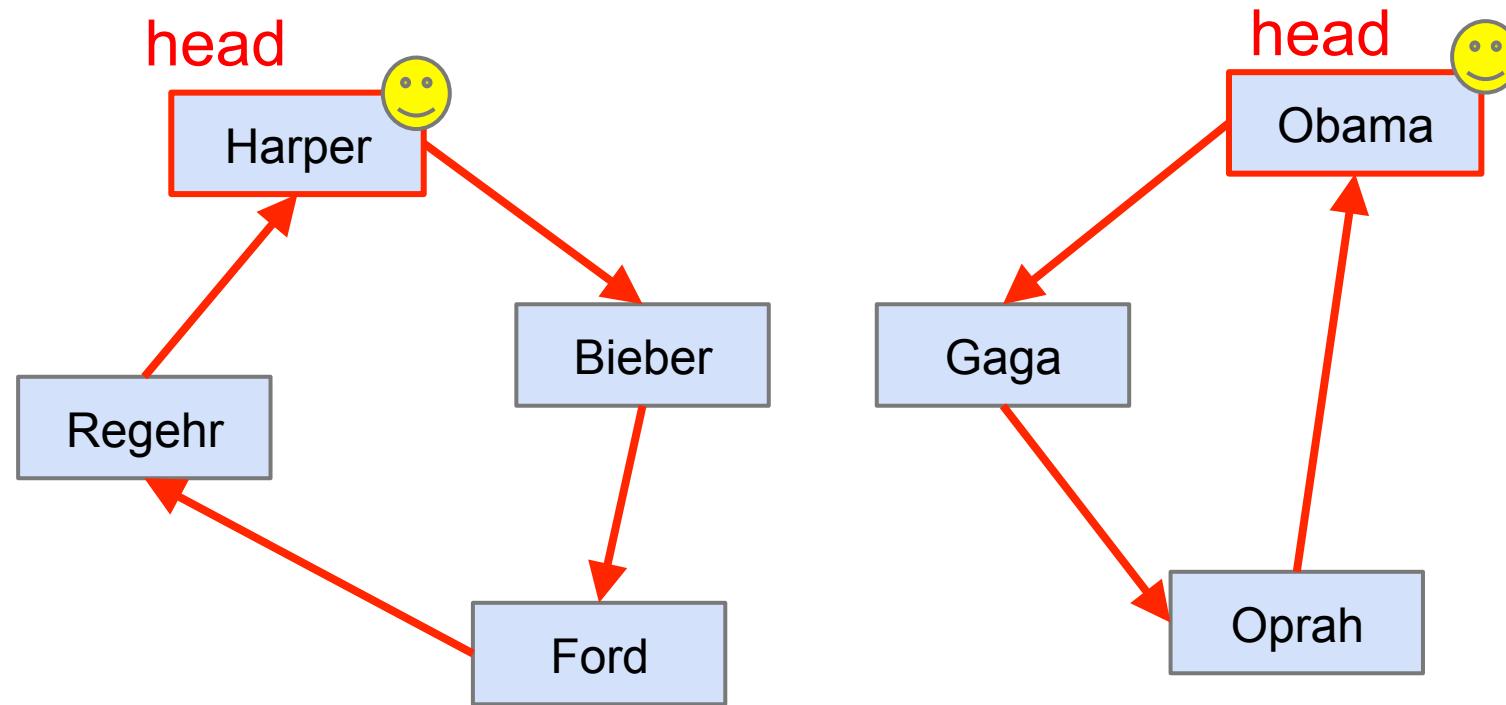
- ◆ worst-case: **O(1)**

→ **FindSet(x):** follow the links until reaching the head

- ◆ **$\Theta(\text{Length of list})$**

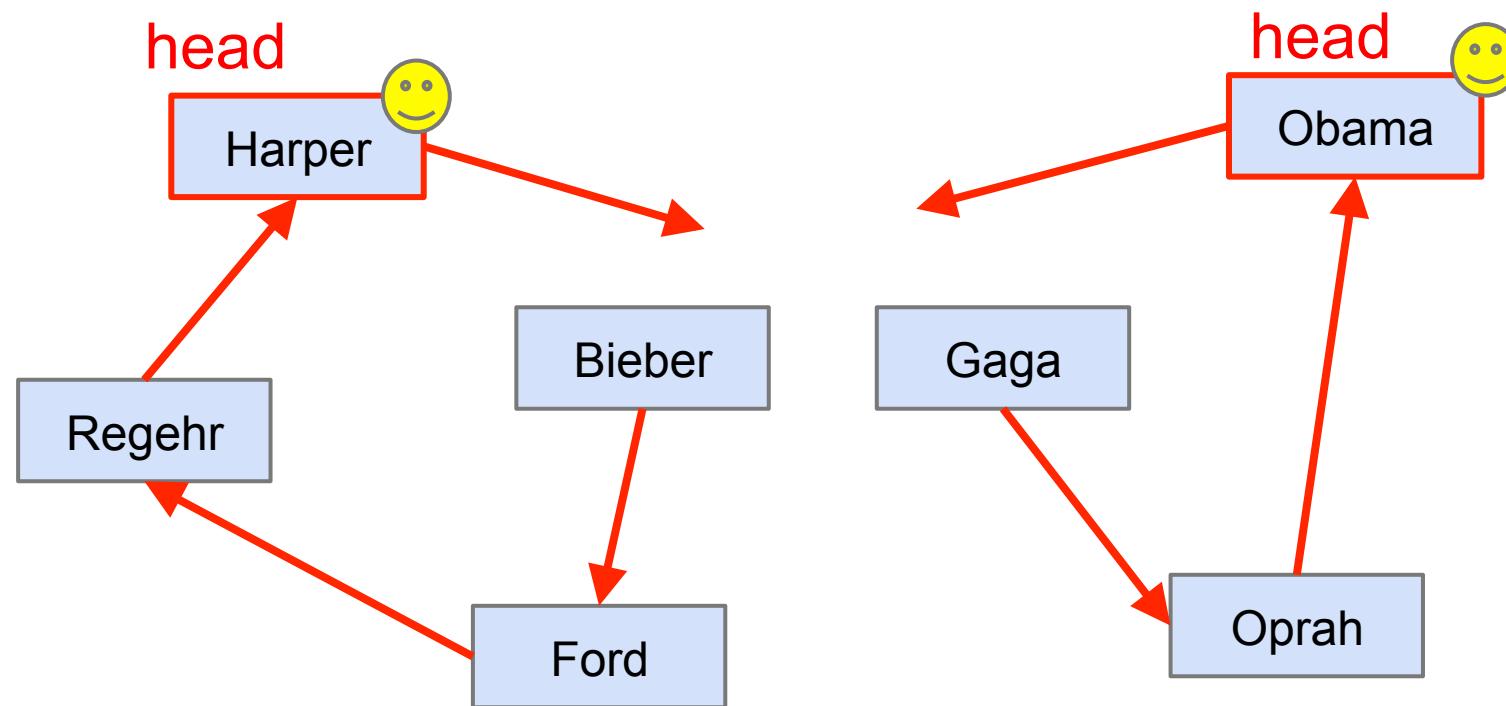
→ **Union(x, y): ...**

Circularly-linked list: Union(Bieber, Gaga)

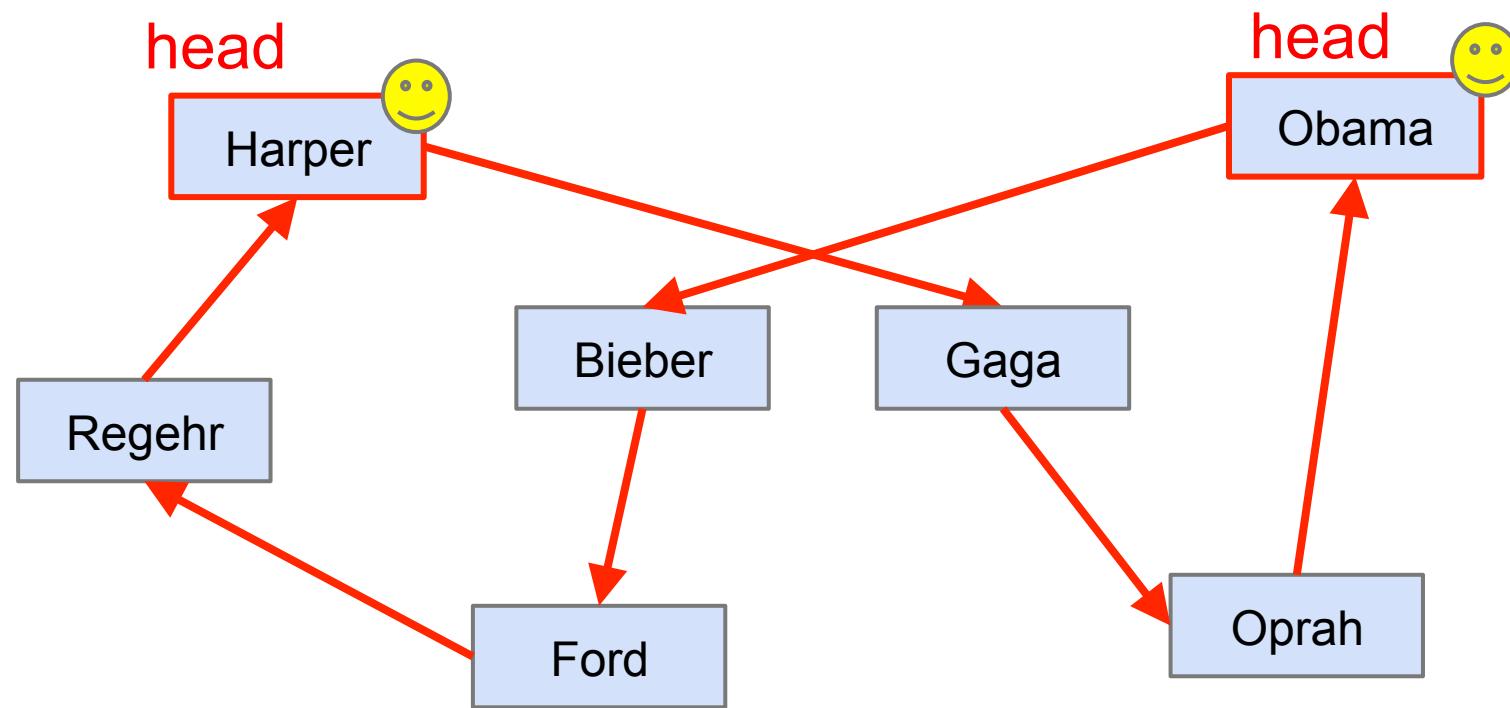


First, locate the head of each linked-list by calling FindSet, takes $\Theta(L)$

Circularly-linked list: Union... 1

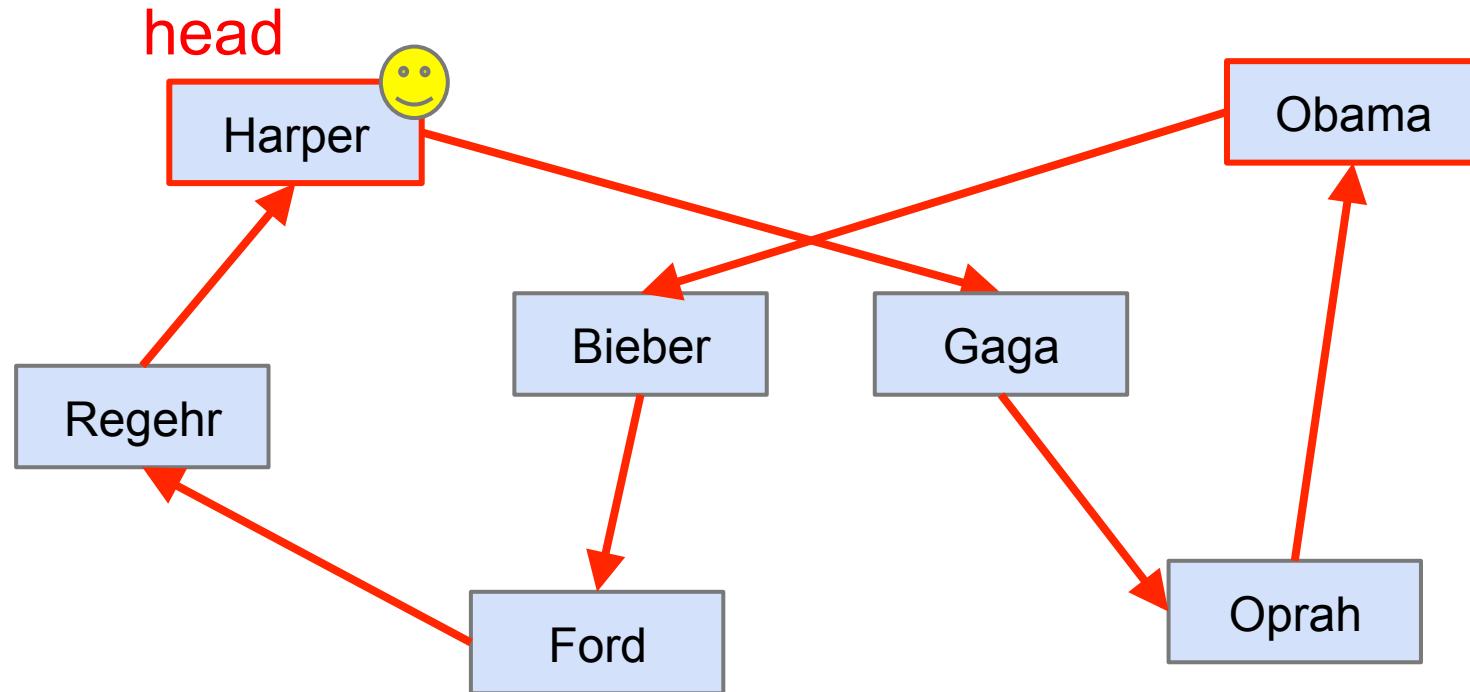


Circularly-linked list: Union... 2



Exchange the two heads' “next” pointers, **O(1)**

Circularly-linked list: Union... 3



Keep only one representative for the new set.

Circularly-linked list: runtime

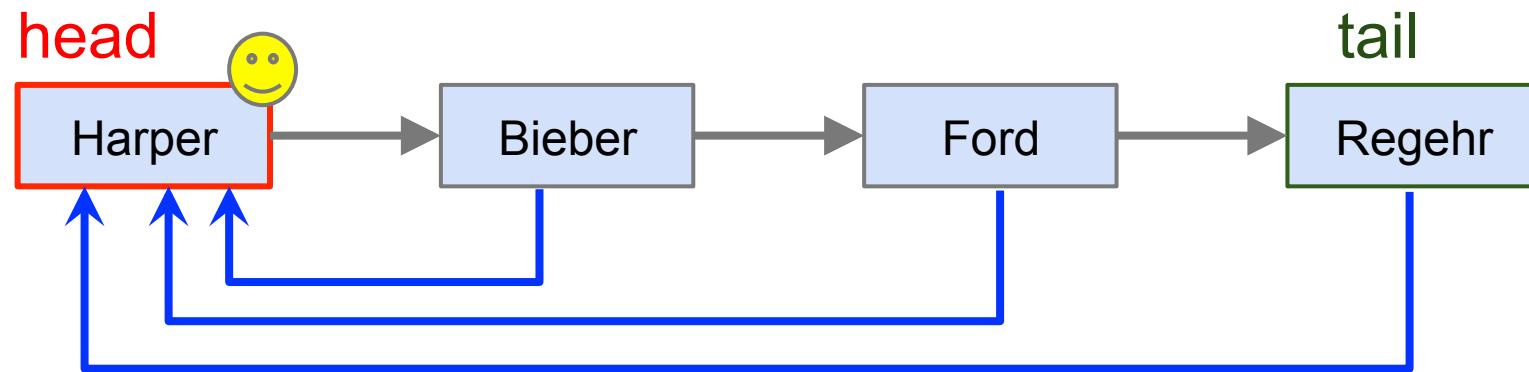
FindSet is the time consuming operation

Amortized analysis: How about the **total cost** of a sequence of m operations (MakeSet, FindSet, Union)?

- A bad sequence: $m/4$ MakeSet, then $m/4 - 1$ Union, then $m/2 + 1$ FindSet
 - ◆ why it's bad: because many FindSet on a large set (of size $m/4$)
- Total cost: **$\Theta(m^2)$**
 - ◆ each of the $m/2 + 1$ FindSet takes **$\Theta(m/4)$**

**Linked list
with extra pointer
(to head)**

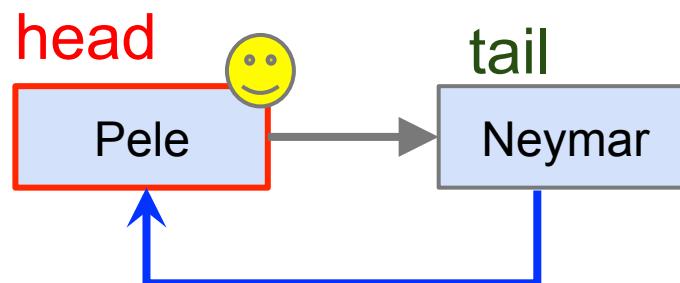
Linked list with pointer to head



- **MakeSet** takes **O(1)**
- **FindSet** now takes **O(1)**, since we can go to head in 1 step, better than circular linked list
- **Union...**

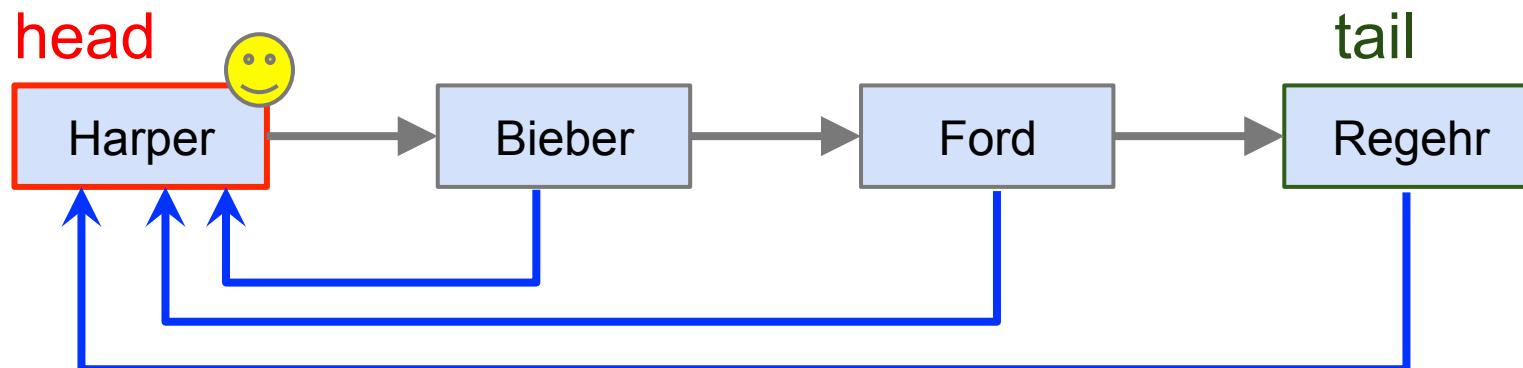
Linked list with pointer to head

Union(Bieber, Pele)

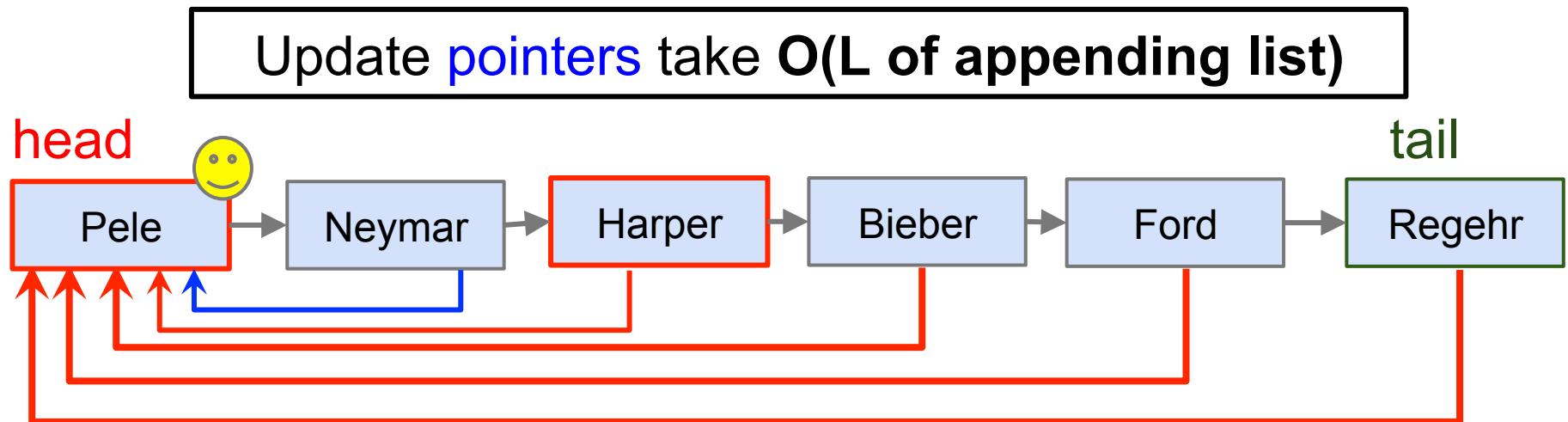
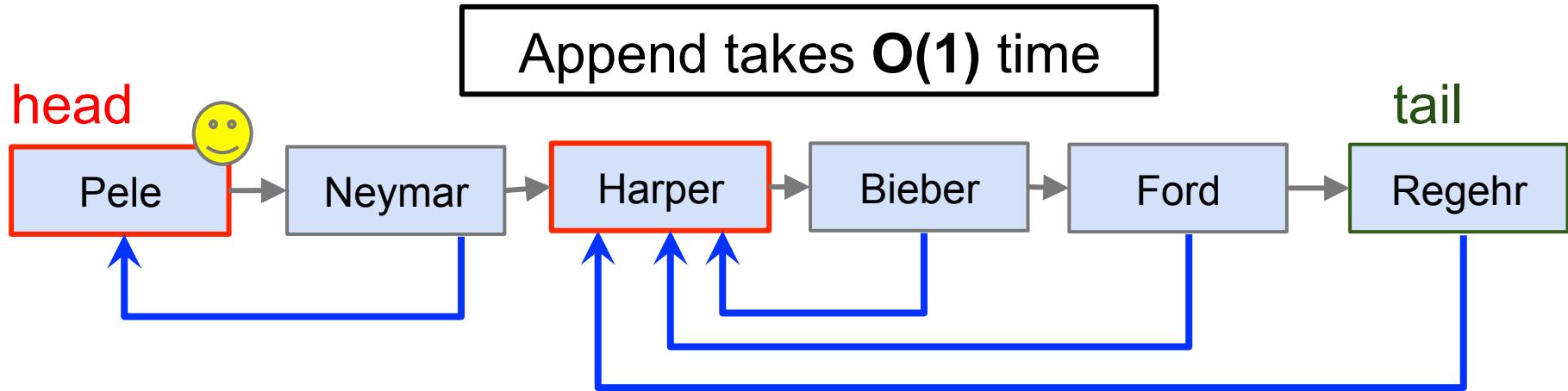


Idea:

Append one list to the other, then **update** the pointers to head



Linked list with pointer to head



Linked list with pointer to head

MakeSet and **FindSet** are fast, **Union** now becomes the time-consuming one, especially if appending a long list.

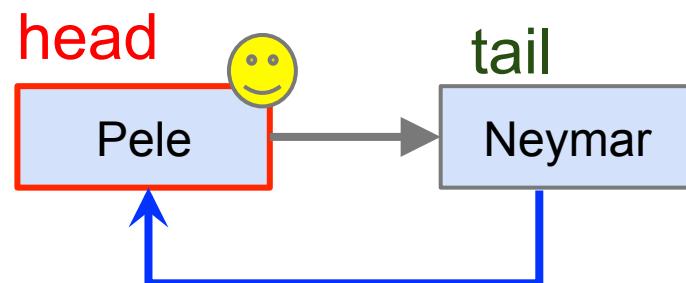
Amortized analysis: The total cost of a sequence of m operations.

- Bad sequence: $m/2$ MakeSet, then $m/2 - 1$ Union, then 1 whatever.
 - ◆ Always let the longer list append, like 1 appd 1, 2 appd 1, 3 appd 1,, $m/2 - 1$ appd 1.
- Total cost: $\Theta(1+2+3+\dots+m/2 - 1) = \Theta(m^2)$

**Linked list
with extra pointer to head
with union-by-weight**

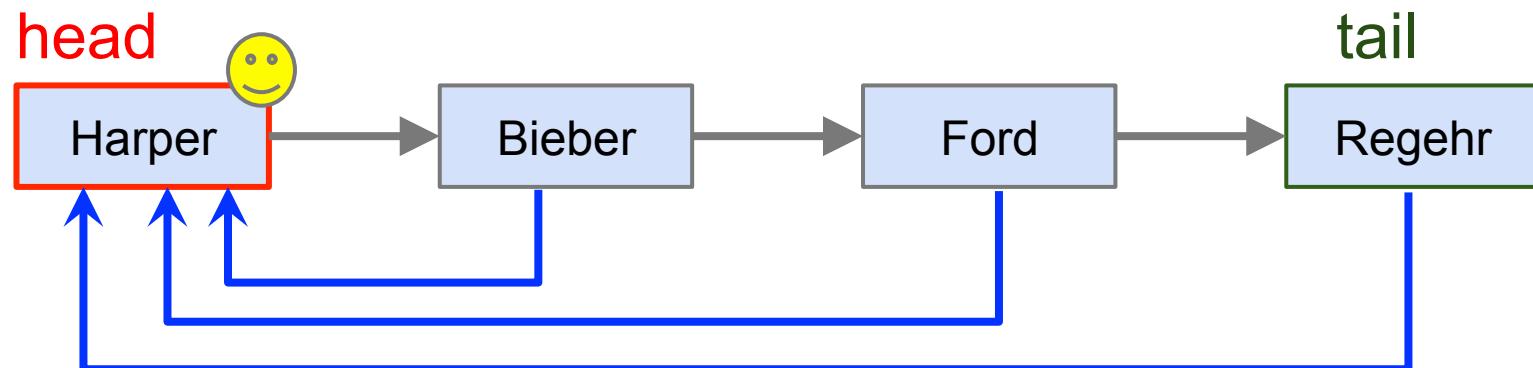
Linked list with union-by-weight

Union(Bieber, Pele)

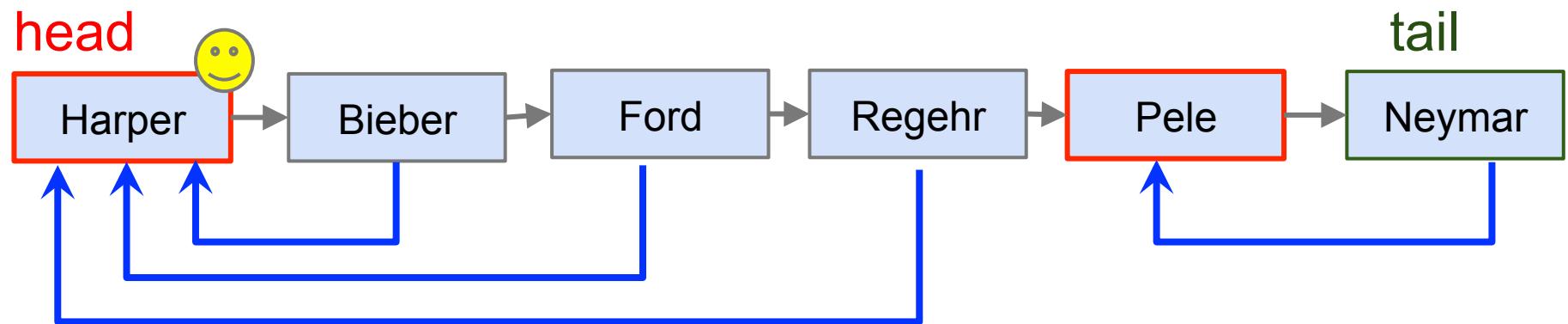


Here we have a choice, let's be a bit smart about it...

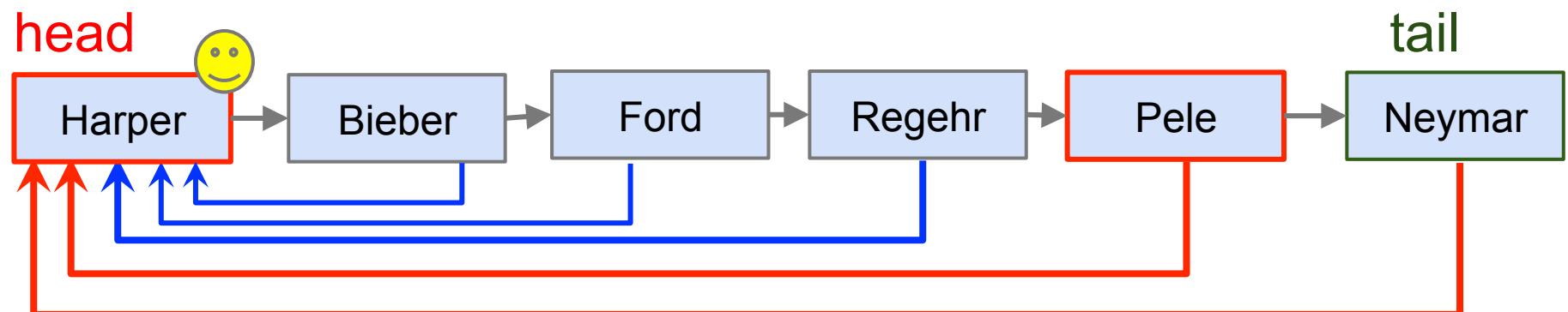
Append the shorter one to the longer one



Linked list with union-by-weight



Need to keep track of the **size (weight)** of each list, therefore called **union-by-weight**



Linked list with union-by-weight

Union-by-weight sounds like a simple heuristic, but it actually provides significant improvement.

For a sequence of m operations which includes n MakeSet operations, i.e., n elements in total, the total cost is $O(m + n \log n)$

i.e., for the previous sequence with $m/2$ MakeSet and $m/2 - 1$ Union, the total cost would be $O(m \log m)$, as opposed to $\Theta(m^2)$ when without union-by-weight.

Linked list with **union-by-weight**

Proof: (assume there are n elements in total)

- Consider an arbitrary element x , how many times does its head pointer need to be updated?
- Because **union-by-weight**, when x is updated, it must be in the smaller list of the two. In other words, after **union**, the size of list at least **doubles**.
- That is, every time x is **updated**, set size **doubles**.
- There are only n elements in total, so we can double at most **$O(\log n)$** times, i.e., x can be updated at most **$O(\log n)$** .
- Same for all n elements, so total updates **$O(n \log n)$**

Ways of implementing Disjoint Sets

-  1. Circularly-linked lists $\Theta(m^2)$
-  2. Linked lists with extra pointer $\Theta(m^2)$
-  3. Linked lists with extra pointer and
with union-by-weight $\Theta(m \log m)$
- 4. Trees
- 5. Trees with union-by-rank
- 6. Trees with path-compression
- 7. Trees with union-by-weight and
path-compression

Benchmark:

Worst-case
total cost of a
sequence of m
operations
(`MakeSet` or `FindSet`
or `Union`)

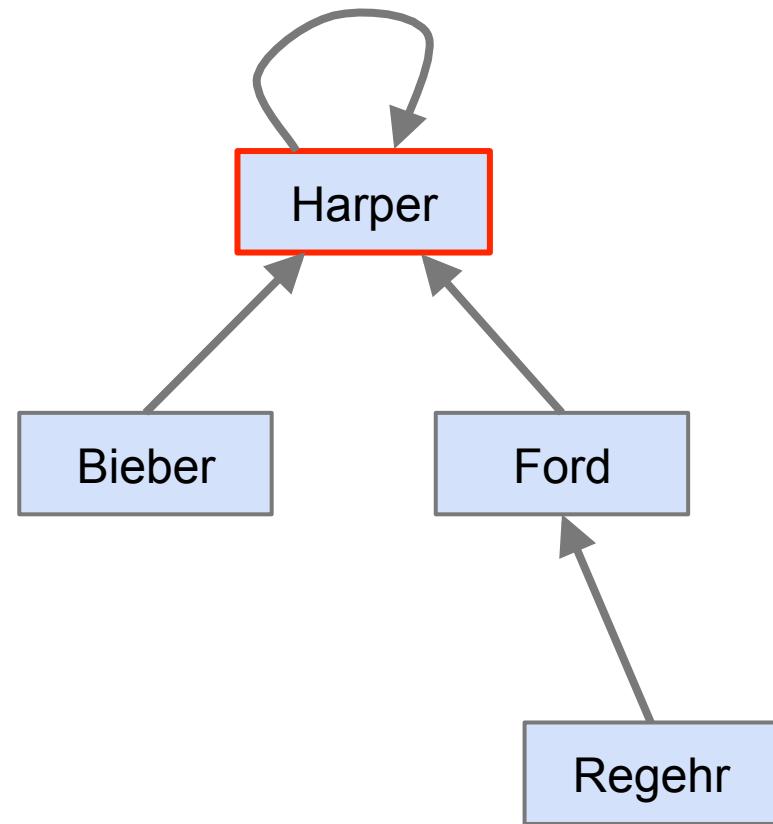
Trees

a.k.a. disjoint set forest



Each set is an “inverted” tree

- Each element keeps a pointer to its **parent** in the tree
- The root points to **itself** (test root by $x.p = x$)
- The **representative** is the root
- NOT necessarily a binary tree or balanced tree



Operations

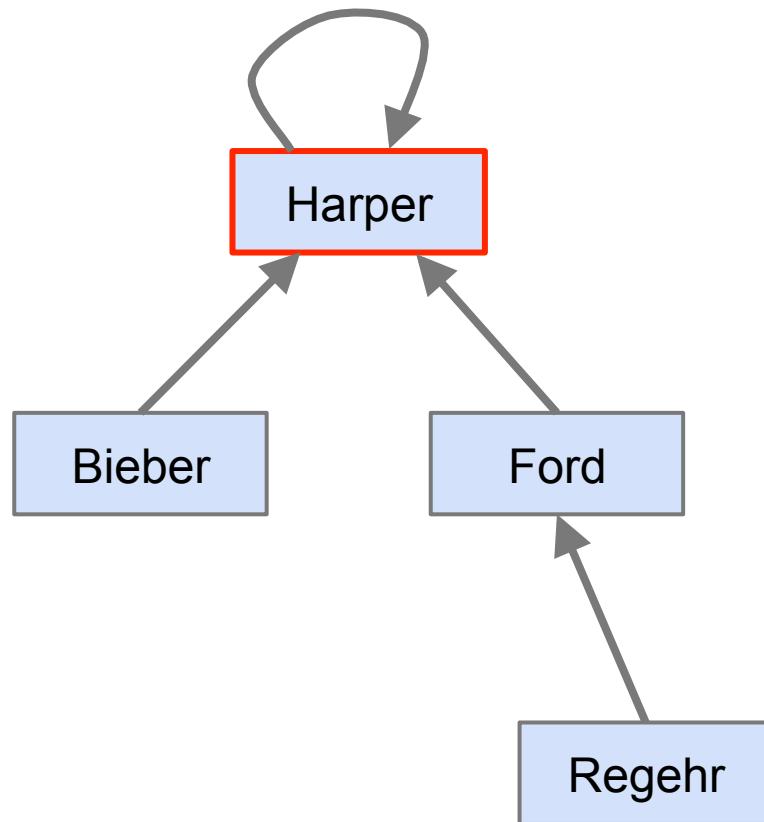
→ **MakeSet(x)**: create a single-node tree with root x

- ◆ $O(1)$

→ **FindSet(x)**: Trace up the parent pointer until the root is reached

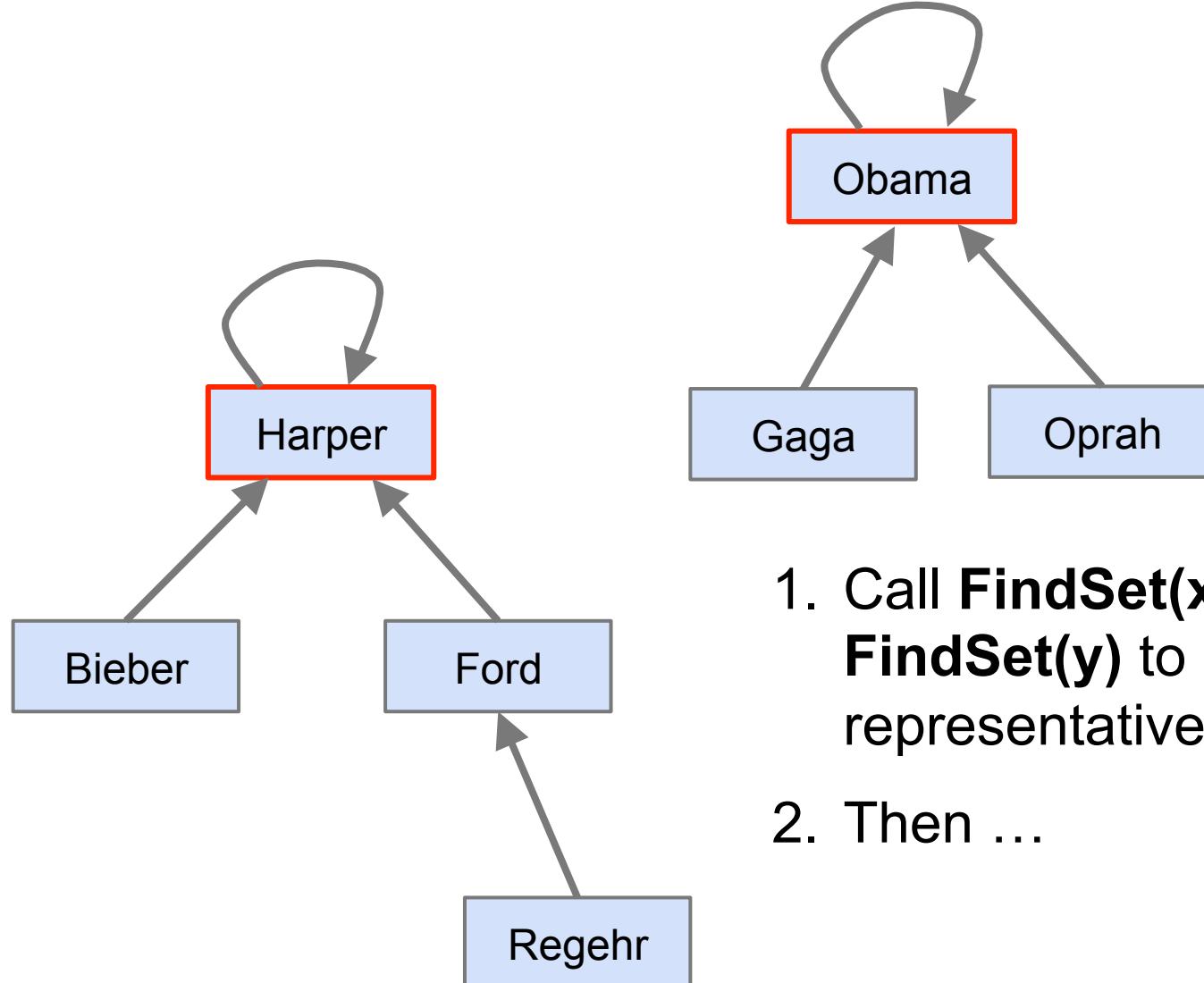
- ◆ $O(\text{height of tree})$

→ **Union(x, y)...**



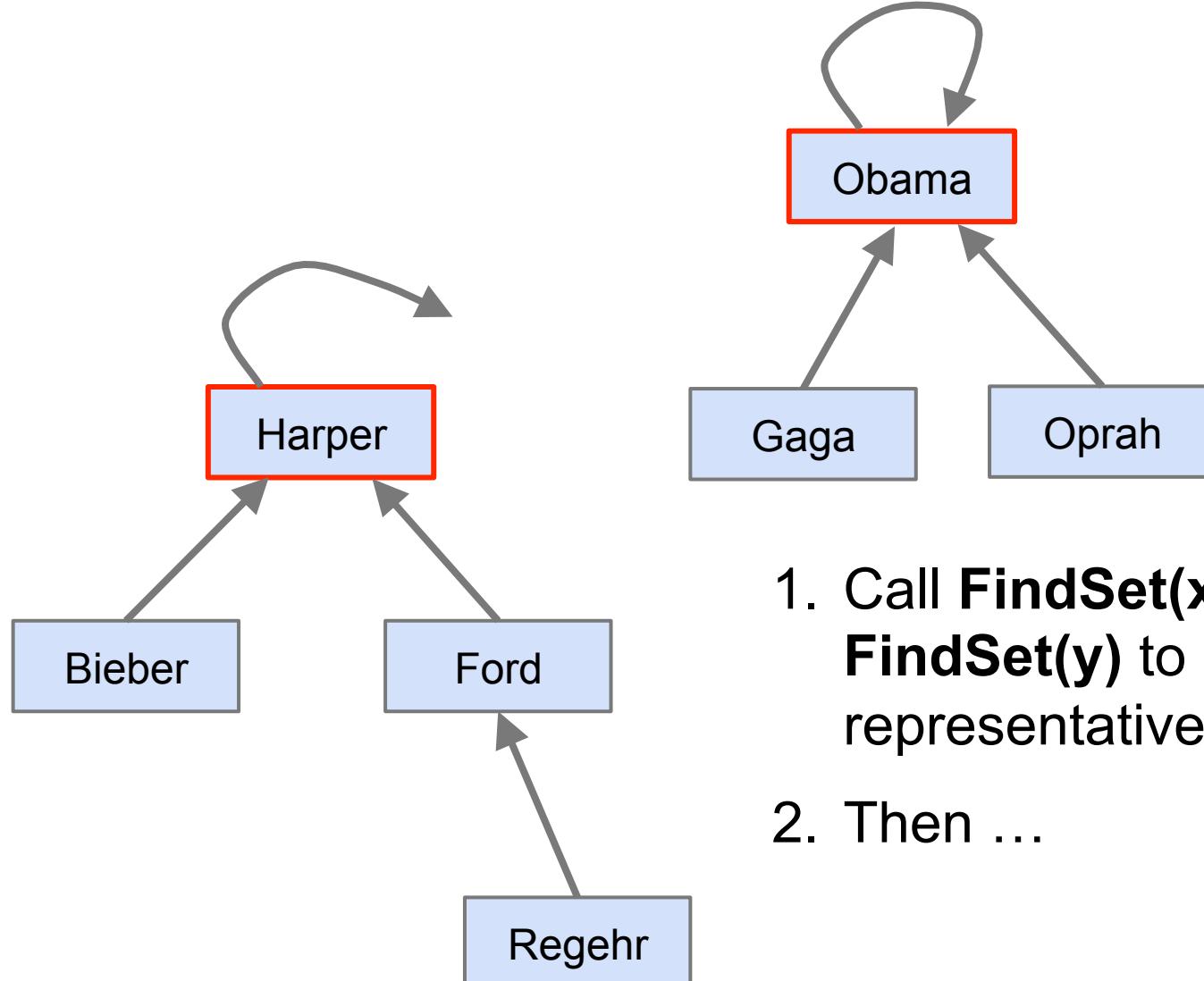
Trees with small heights would be nice.

Union(Bieber, Gaga)



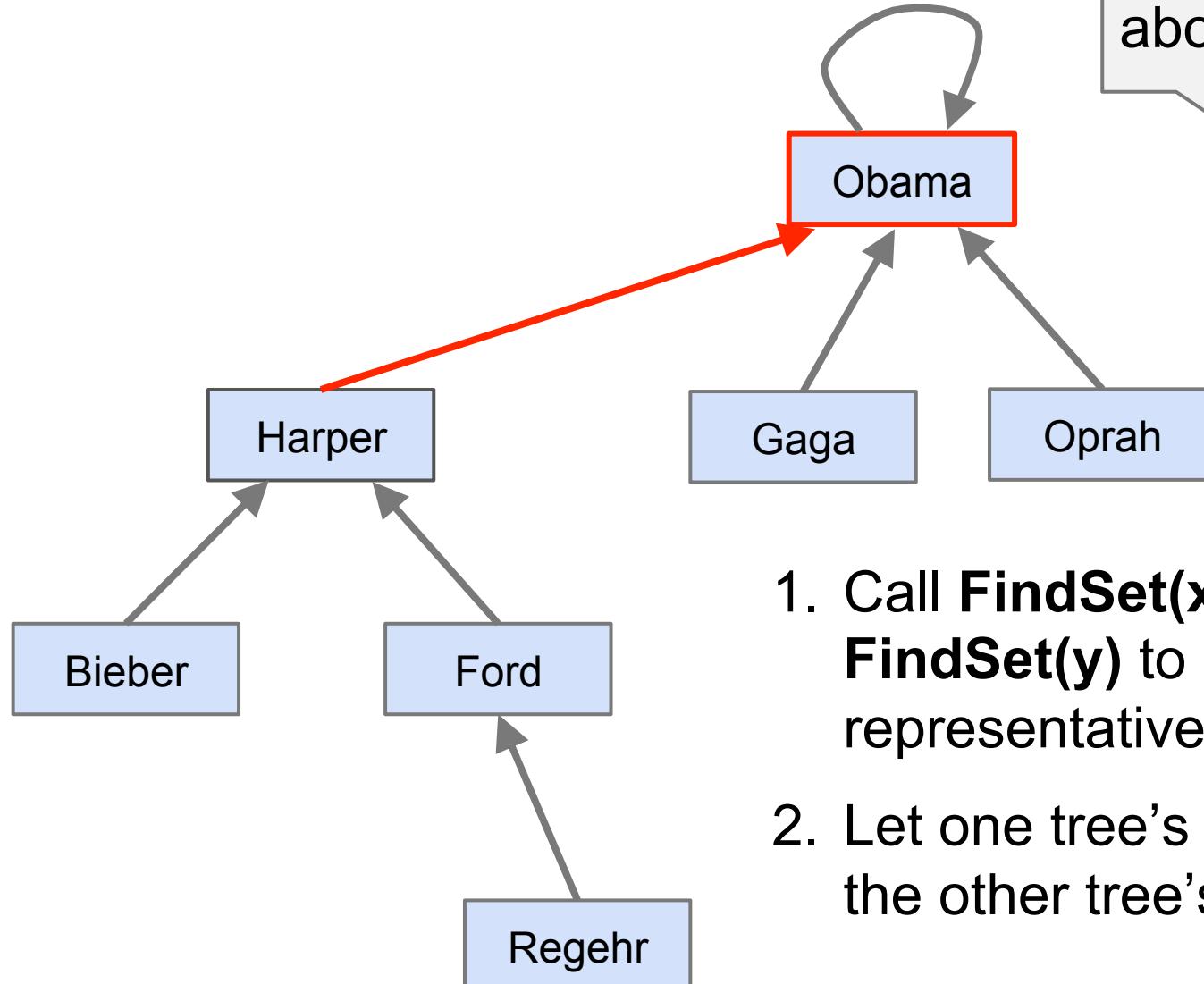
1. Call **FindSet(x)** and **FindSet(y)** to locate the representatives, **O(h)**
2. Then ...

Union(Bieber, Gaga)

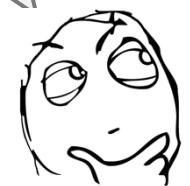


1. Call **FindSet(x)** and **FindSet(y)** to locate the representatives, **O(h)**
2. Then ...

Union(Bieber, Gaga)



Could we have been smarter about this?

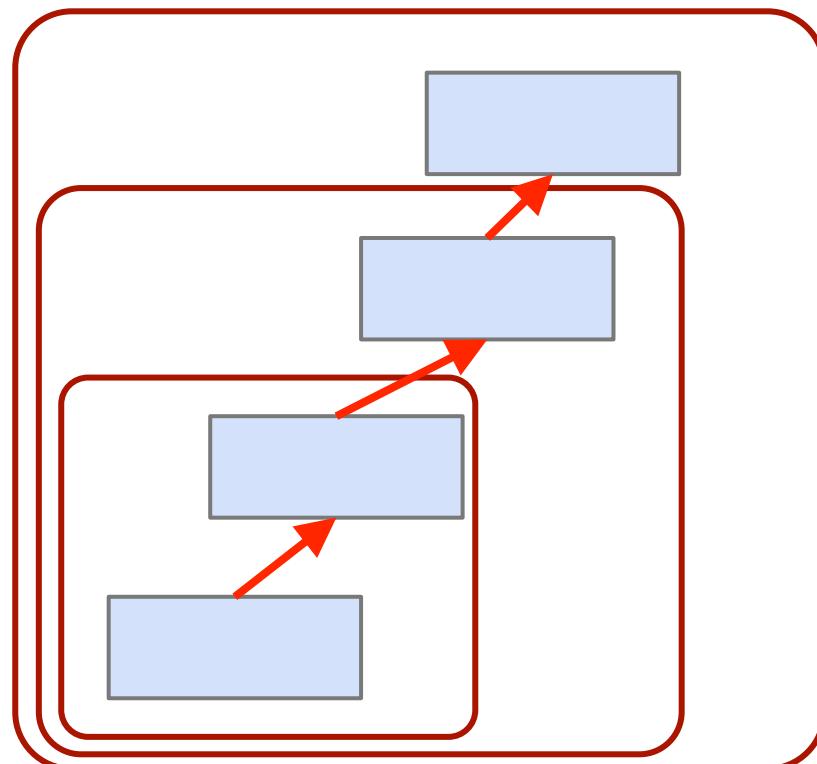


1. Call **FindSet(x)** and **FindSet(y)** to locate the representatives, **O(h)**
2. Let one tree's root point to the other tree's root, **O(1)**

Benchmarking: runtime

The worst-case sequence of m operations.
(with **FindSet** being the bottleneck)

$m/4$ MakeSets, $m/4 - 1$ Union, $m/2 + 1$ FindSet



Total cost in worst-case sequence :

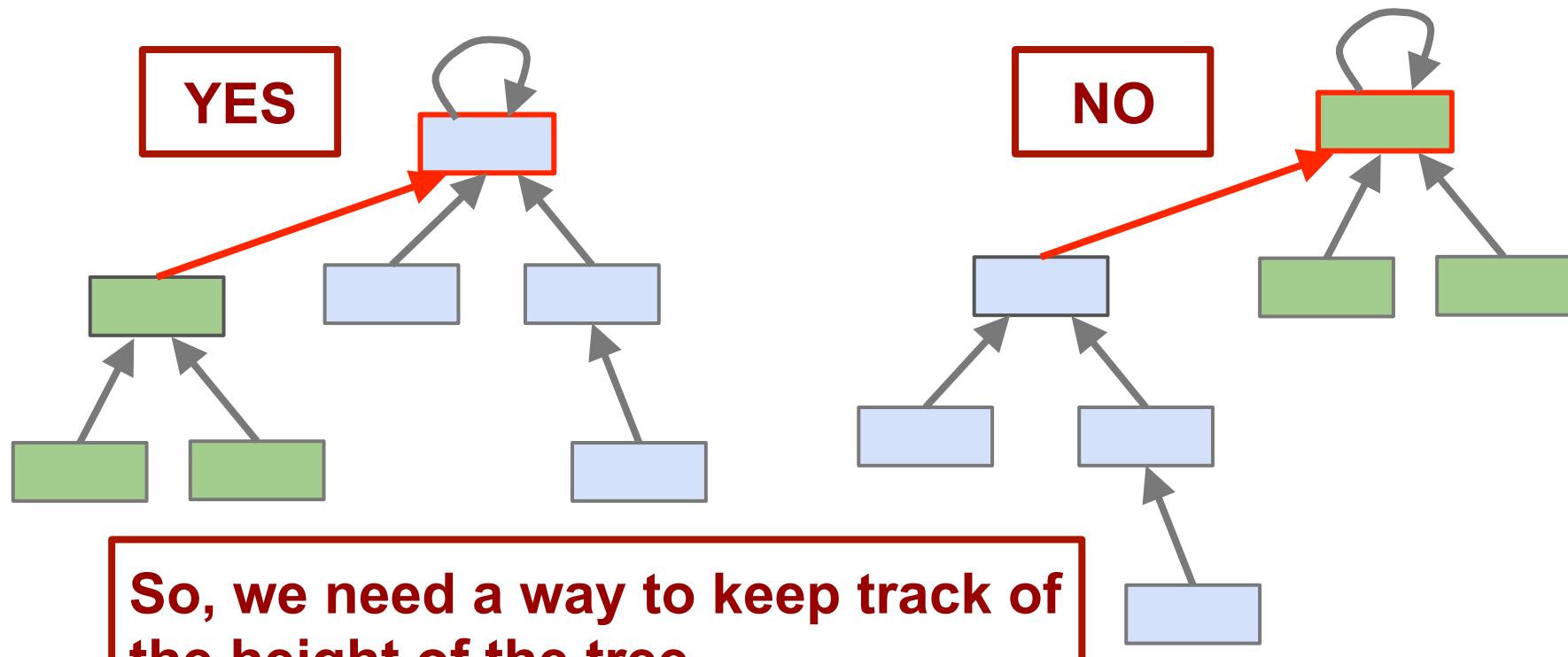
$$\Theta(m^2)$$

(each FindSet would take up to $m/4$ steps)

Trees with **union-by-rank**

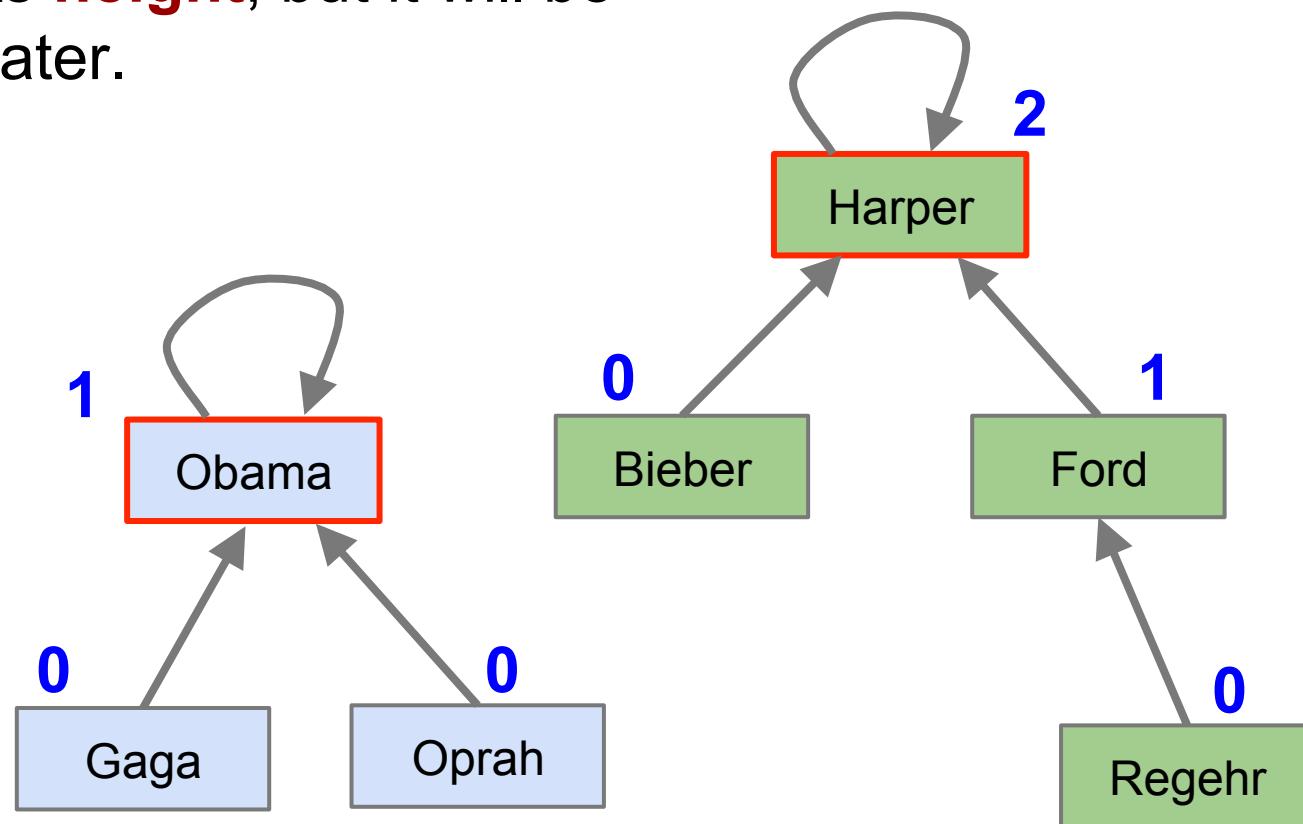
Intuition

- FindSet takes **O(h)**, so the **height** of tree matters
- To keep the unioned tree's height small, we should let the **taller** tree's root be the root of the unioned tree



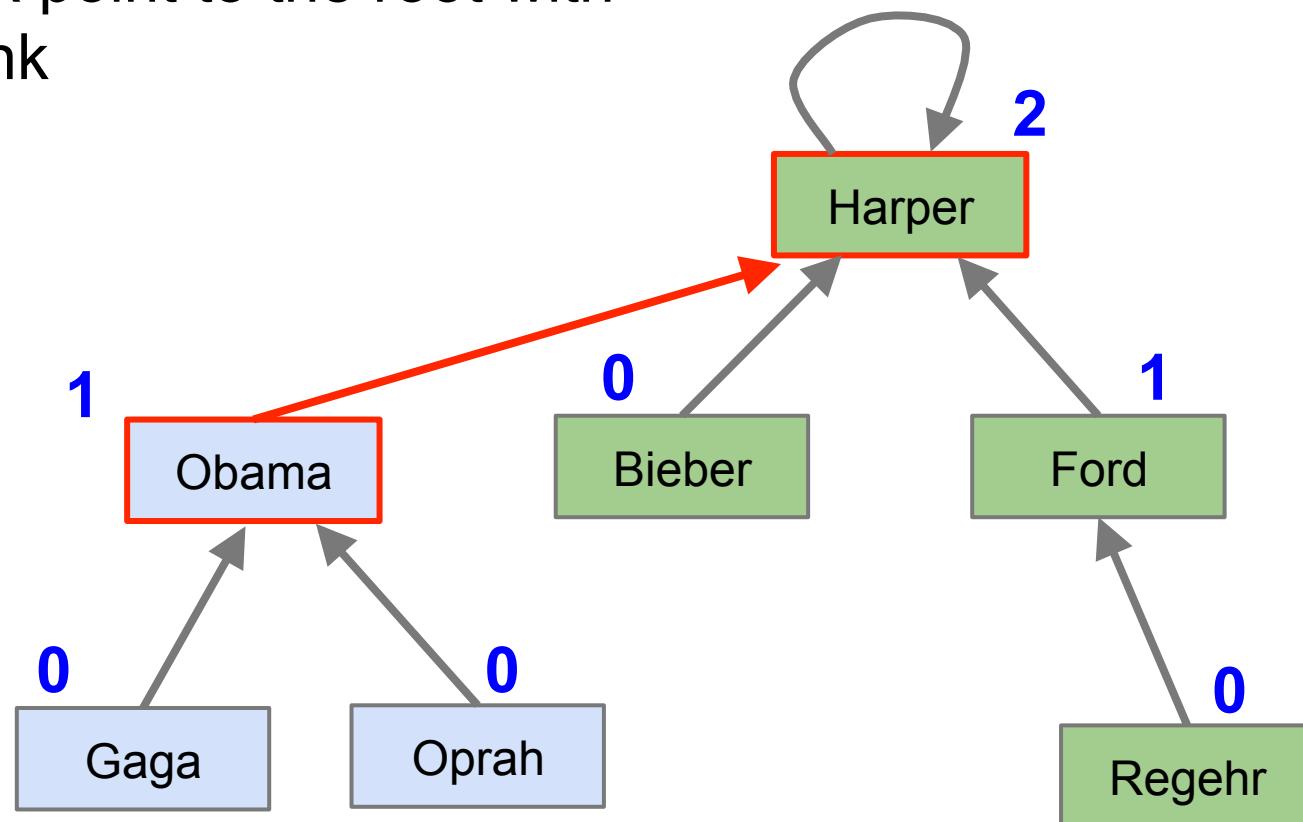
Each node keeps a **rank**

For now, a node's **rank** is the same as its **height**, but it will be **different** later.



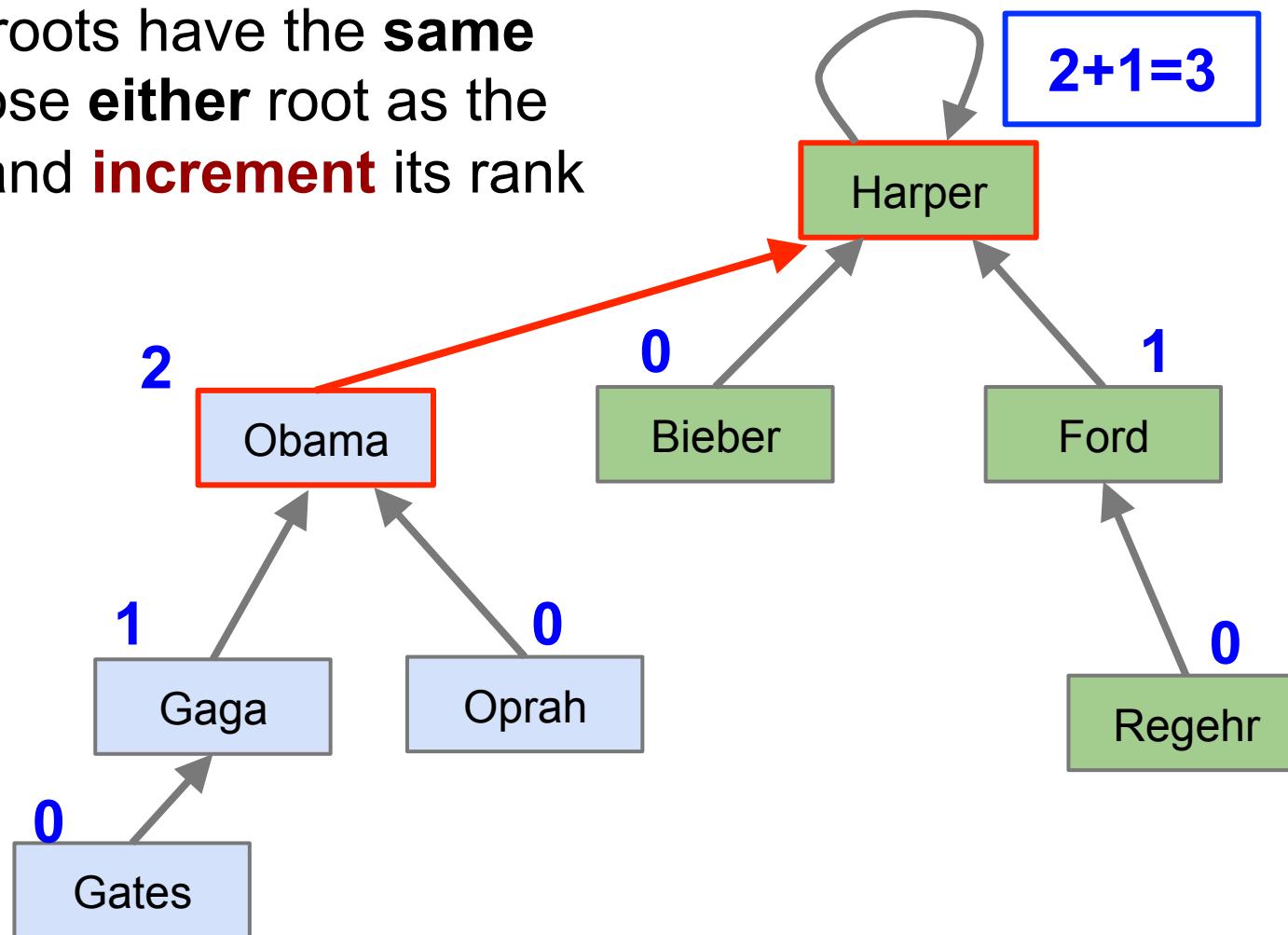
Each node keeps a **rank**

When **Union**, let the root with **lower rank** point to the root with **higher rank**



Each node keeps a rank

If the two roots have the **same** rank, choose **either** root as the new root and **increment** its rank



Benchmarking: runtime

It can be proven that, a tree of n nodes formed by **union-by-rank** has height at most **$\log n$** , which means **FindSet** takes **$O(\log n)$**

So for a sequence of **$m/4$** MakeSets, **$m/4 - 1$** Union, **$m/2 + 1$** FindSet operations, the total cost is **$O(m \log m)$**

Rank of a tree with n nodes is at most $\log n$,
i.e., $r(n) \leq \log n$

Proof:

Equivalently, prove $n(r) \geq 2^r$

Use **induction** on r

Base step: if $r = 0$ (single node), $n(0) = 1$, TRUE

Inductive step: assume $n(r) \geq 2^r$

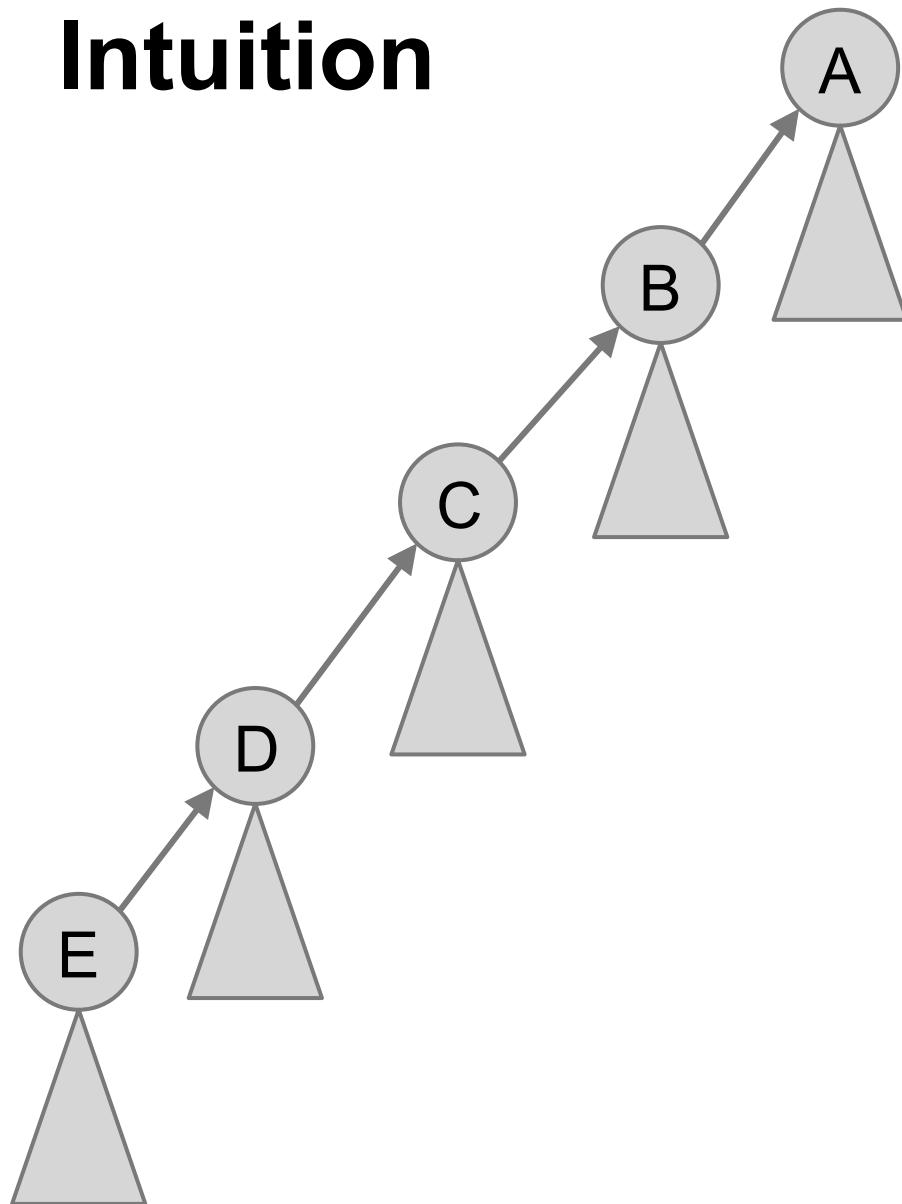
→ a tree with root rank $r+1$ is a result of unioning two trees
with root rank r , so

→ $n(r+1) = n(r) + n(r) \geq 2 \times 2^r = 2^{r+1}$

→ Done.

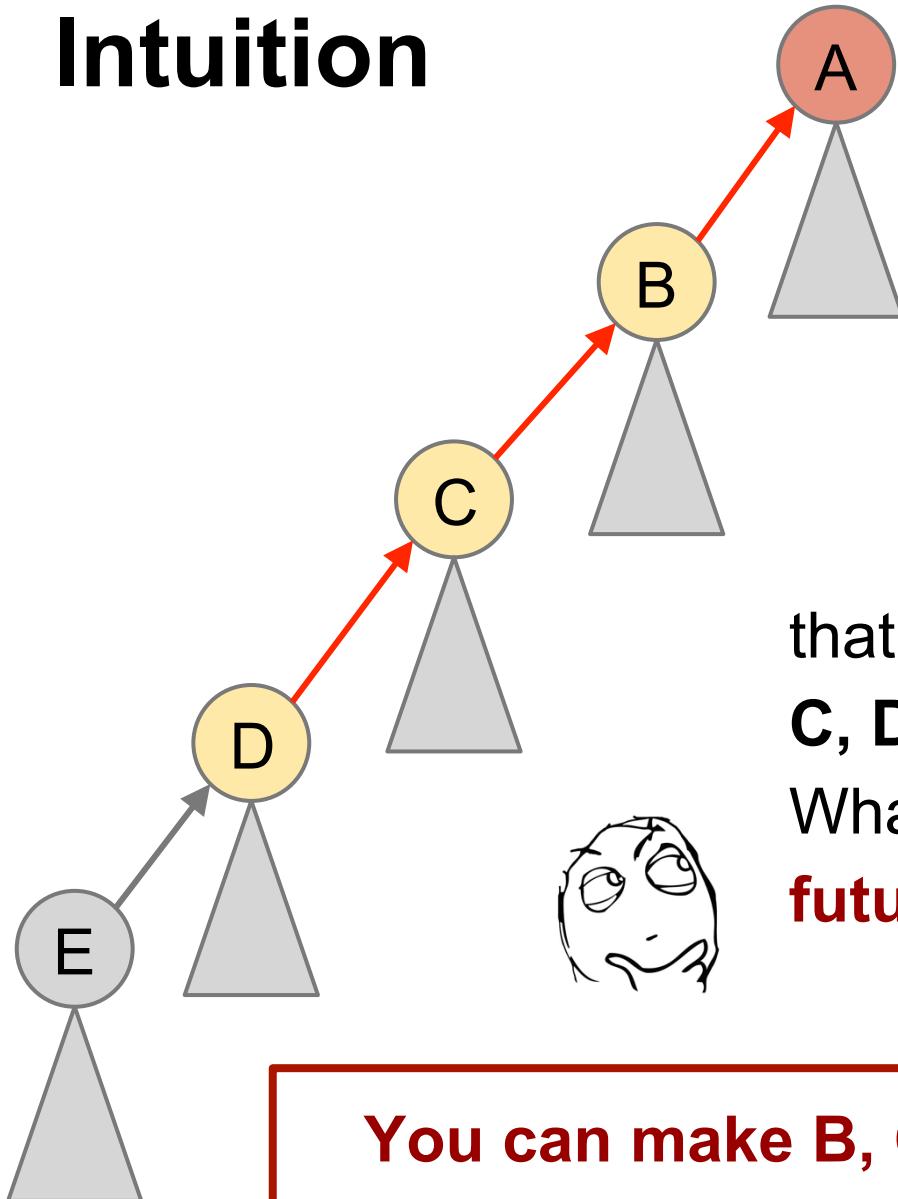
Trees with path compression

Intuition



Now I do a
FindSet(D)

Intuition



Now I do a
FindSet(D)

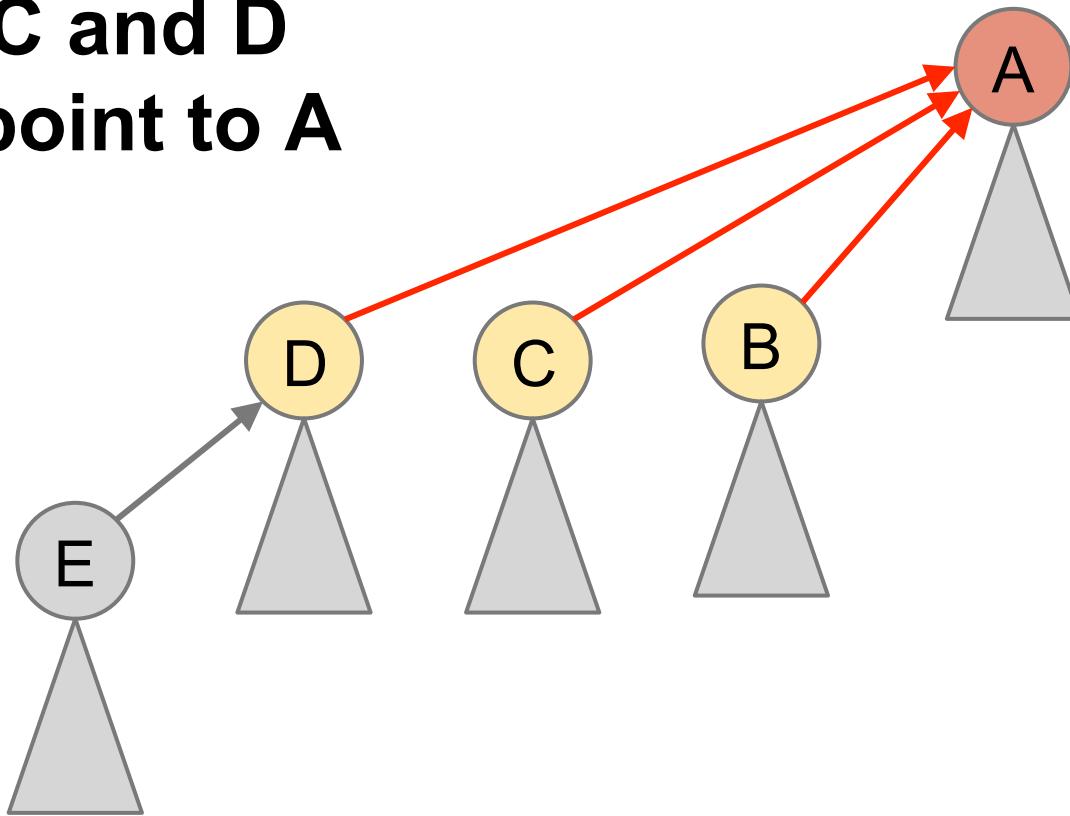
On the way of finding **A**, you visit **D**, **C**, **B** and **A**.

that is, now you have access to **B**, **C**, **D** and the root **A**.

What **nice** things can you do for
future FindSet operations?

You can make **B**, **C** and **D** super close to **A**!

**Make B, C and D
directly point to A**



In other words, the path $D \rightarrow C \rightarrow B \rightarrow A$ is “**compressed**”.

Extra cost to FindSet: at most **twice** the cost, so does not affect the order of complexity

Benchmark: runtime

Can be prove: for a sequence of operations with **n** MakeSet (so at most **n-1** Union), and **k** FindSet, the worst-case total cost of the sequence is in

$$\Theta\left(n + k \cdot \left(1 + \log_{2+\frac{k}{n}} n\right)\right)$$

So for a sequence of **m/4** MakeSets, **m/4 - 1** Union, **m/2 + 1** FindSet, the worst-case total cost is in **$\Theta(m \log m)$**

Ways of implementing Disjoint Sets

- | | |
|--|--------------------|
| 1. Circularly-linked lists | $\Theta(m^2)$ |
| 2. Linked lists with extra pointer | $\Theta(m^2)$ |
| 3. Linked lists with extra pointer and
with union-by-weight | $\Theta(m \log m)$ |
| 4. Trees | $\Theta(m^2)$ |
| 5. Trees with union-by-rank | $\Theta(m \log m)$ |
| 6. Trees with path-compression | $\Theta(m \log m)$ |

Benchmark:

Worst-case
total cost of a
sequence of m
operations
(`MakeSet` or `FindSet`
or `Union`)

Can we do better than $\Theta(m \log m)$?

U. B. R.

P. C.



**Trees with
union-by-rank
and
path compression**

How to **combine** union-by-rank and path compression?

- Path compression happens in the **FindSet** operation
- Union-by-rank happens in the **Union** operation (outside **FindSet**)
- So they don't really interfere with each other, simply use them both!

Pseudocodes

Complete code using both union-by-rank
and path compression

MakeSet(x):

```
x.p ← x  
x.rank ← 0
```

FindSet(x):

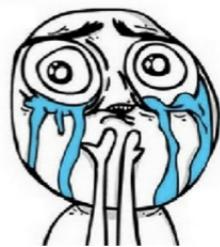
```
if x ≠ x.p: # if not root  
    x.p ← FindSet(x.p)  
return x.p
```

Union(x, y):

```
Link(FindSet(x), \  
     FindSet(y))
```

Link(x, y):

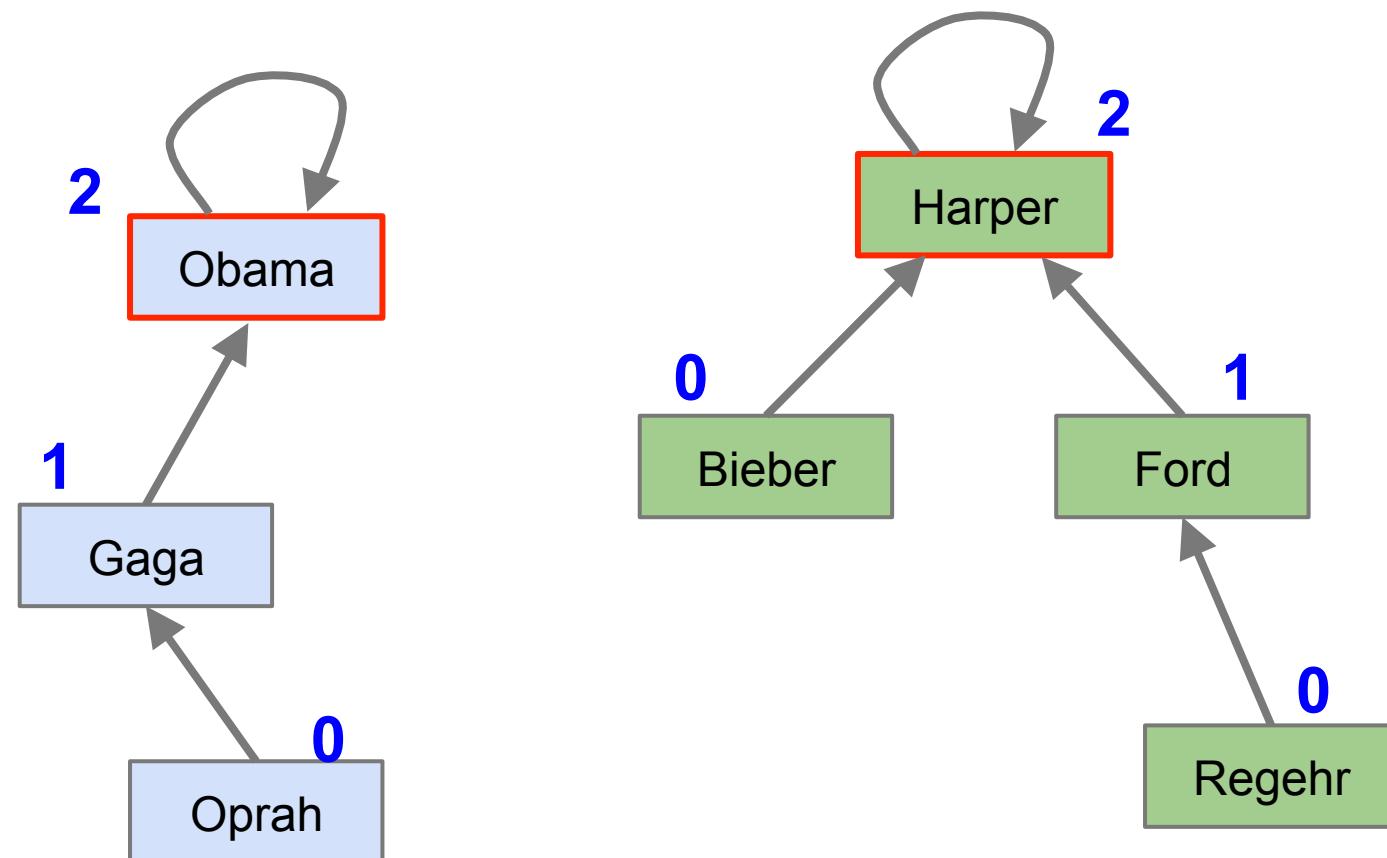
```
if x.rank > y.rank:  
    y.p ← x  
else:  
    x.p ← y  
    if x.rank = y.rank:  
        y.rank += 1
```



IT'S SO BEAUTIFUL
memegenerator.net

Exercise

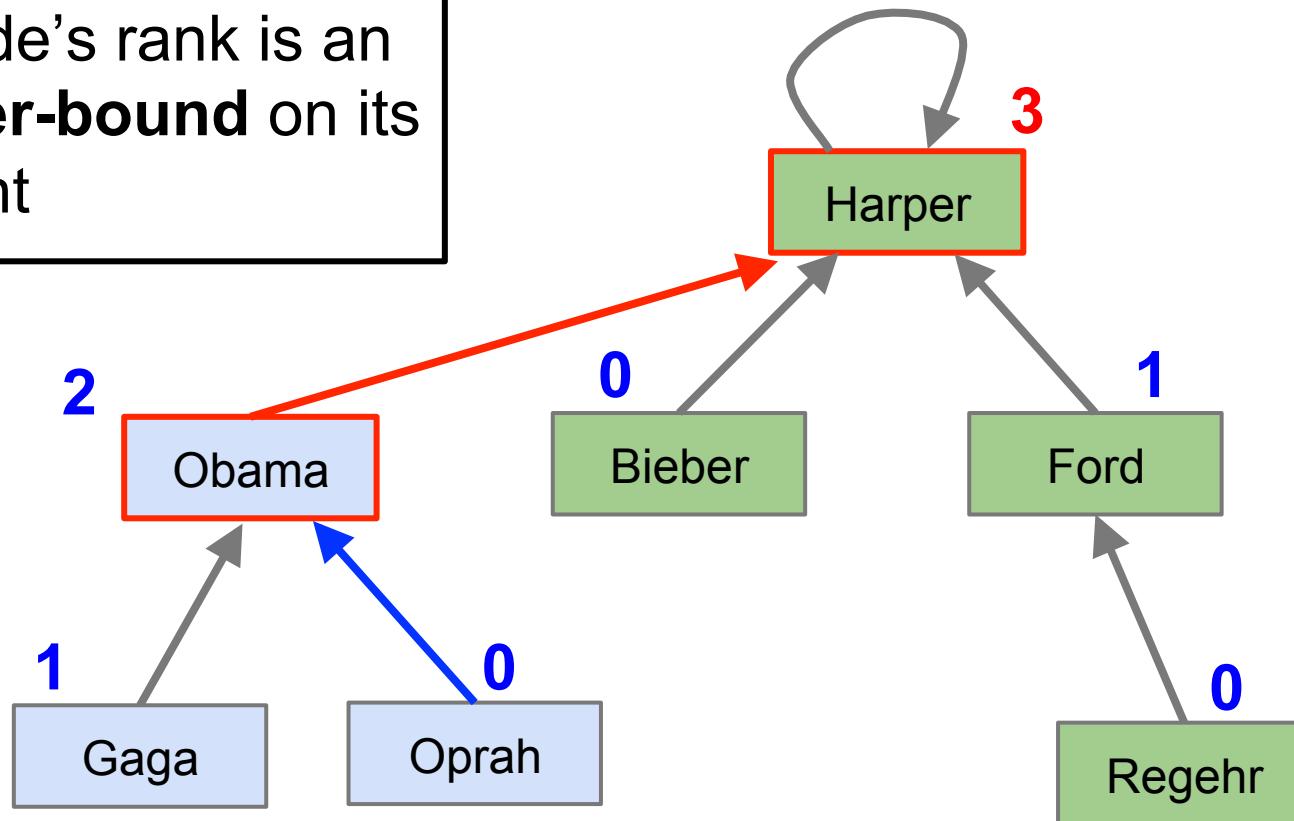
Draw the result after **Union(Oprah, Ford)**.
using both union-by-rank and path compression



Note: rank \neq height

because path compression does NOT maintain height info

a node's rank is an
upper-bound on its
height



Benchmark: runtime

Can be proven: for a sequence of m operations with n MakeSet (so at most $n-1$ Union), worst-case total cost of the sequence is $O(m \log^* n)$

Note: $\log^* n$ is equal to the number of times the log function must be iteratively applied so that the result is at most 1

$$\text{Example: } \log_2(2^{256}) = 256$$

$$\log_2(256) = 8$$

$$\log_2(8) = 3$$

$$\log_2(3) < 1.6$$

$$\log_2(1.6) < 1$$

So $\log^*(2^{256}) = 5$, and $\log^*(2^m) = 6$, where $m=2^{256}$

Since $\log^* n$ is so slowly growing it is like a constant.

Sketch of Analysis

Lemma: A node v which is the root of a subtree of rank r has at least 2^r nodes

(We already proved this.)

Lemma: If there are n nodes, the maximum number of nodes of rank r is $n/2^r$

Each node which is the root of a subtree with rank r has at least 2^r nodes. So maximum is $n/2^r$ rank r root notes, each with 2^r children

Sketch of Analysis

Group the nodes into at most $\log^* n$ buckets:

Bucket 0: nodes of rank 0

Bucket 1: nodes of rank 1

Bucket 2: nodes of rank 2-3

Bucket 3: nodes of rank 4-16

...

Bucket B: nodes of rank $[r, 2^r - 1] = [r, R-1]$

Bucket B+1: nodes of rank $[R, 2^R - 1]$

Note: the maximum number of elements in bucket containing nodes of rank $[R, 2^R - 1]$ is at most $n/2^R + n/2^{R+1} + \dots + n/2^{2^R-1} \leq 2n/2^R$

Sketch of Analysis

Let F be the list of all m FindSet operations performed

Then total cost of m finds is $T_1 + T_2 + T_3$

Where T_1 = links pointing to root that are traversed

T_2 = links traversed between nodes in different buckets

T_3 = links traversed between nodes in same bucket

- $T_1 \leq m$ since each FindSet traverses one link to root
- $T_2 \leq m \log^* n$ since there are only $\log^* n$ buckets
- It is left to bound T_3

Sketch of Analysis

It is left to bound T_3

Suppose we are traversing from u to v , where u, v are both in the bucket of nodes with rank $[B, 2^B - 1]$

Since the rank is always increasing as we follow a path to a root, the number of links going from u to v is at most $2^B - 1 - B \leq 2^B$

Thus $T_3 \leq \sum_B 2^B \frac{2n}{2^B} \leq 2n \log^* n$

Thus $T_1 + T_2 + T_3 = O(m \log^* n)$

Summary of worst case runtime for m operations, n elements)

- 1. Circularly-linked lists $\Theta(m^2)$
- 2. Linked lists with extra pointer $\Theta(m^2)$
- 3. Linked lists with extra pointer and
with union-by-weight $\Theta(m \log m)$
- 4. Trees $\Theta(m^2)$
- 5. Trees with union-by-rank $\Theta(m \log m)$
- 6. Trees with path compression $\Theta(m \log m)$
- 7. Trees with union-by-rank and
path compression $O(m \log^* n)$

Next week

- Lower bounds
- Review for final exam

CSC263 Week 12

Announcements

- No tutorial this week
- No office hours today (but the usual ones on Friday and Monday)
- Extra office hours for final exam

Lower Bounds

So far, we have mostly talked about **upper-bounds** on algorithm complexity, i.e., $O(n \log n)$ means the algorithm takes **at most** $cn \log n$ time for some c .

However, sometime it is also useful to talk about **lower-bounds** on algorithm complexity, i.e., how much time the algorithm **at least** needs to take.

Scenario #1



You, implement a sorting algorithm with worst-case runtime $O(n \log \log n)$ by next week.

Okay Boss, I will try to do that ~

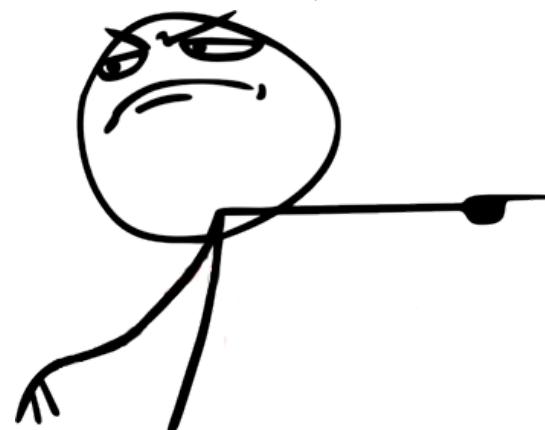
You try it for a week, cannot do it, then you are fired...



Scenario #2



You, implement a sorting algorithm with worst-case runtime $O(n \log \log n)$ by next week.



No, Boss. $O(n \log \log n)$ is below the **lower bound** on sorting algorithm complexity , I can't do it, **nobody** can do it!

Why learn about lower bounds

→ Know your limit

- ◆ we always try to make algorithms faster, but if there is a limit that you cannot exceed, you want to know

→ Approach the limit

- ◆ Once you have an understanding about of limit of the algorithm's performance, you get insights about how to approach that limit.

Lower bounds on sorting algorithms

Upper bounds: We know a few sorting algorithms with worst-case $O(n \log n)$ runtime.

Is $O(n \log n)$ the best we can do?

Actually, yes, because the lower bound on sorting algorithms is $\Omega(n \log n)$, i.e., a sorting algorithm needs **at least** $cn \log n$ time to finish in worst-case.

actually, more precisely ...

The lower bound $n \log n$ applies to only all **comparison based** sorting algorithms, with **no assumptions** on the values of the elements.

It is possible to do faster than $n \log n$ if we make **assumptions** on the values.

Example: sorting with assumptions

Sort an array of n elements which are either 1 or 2.

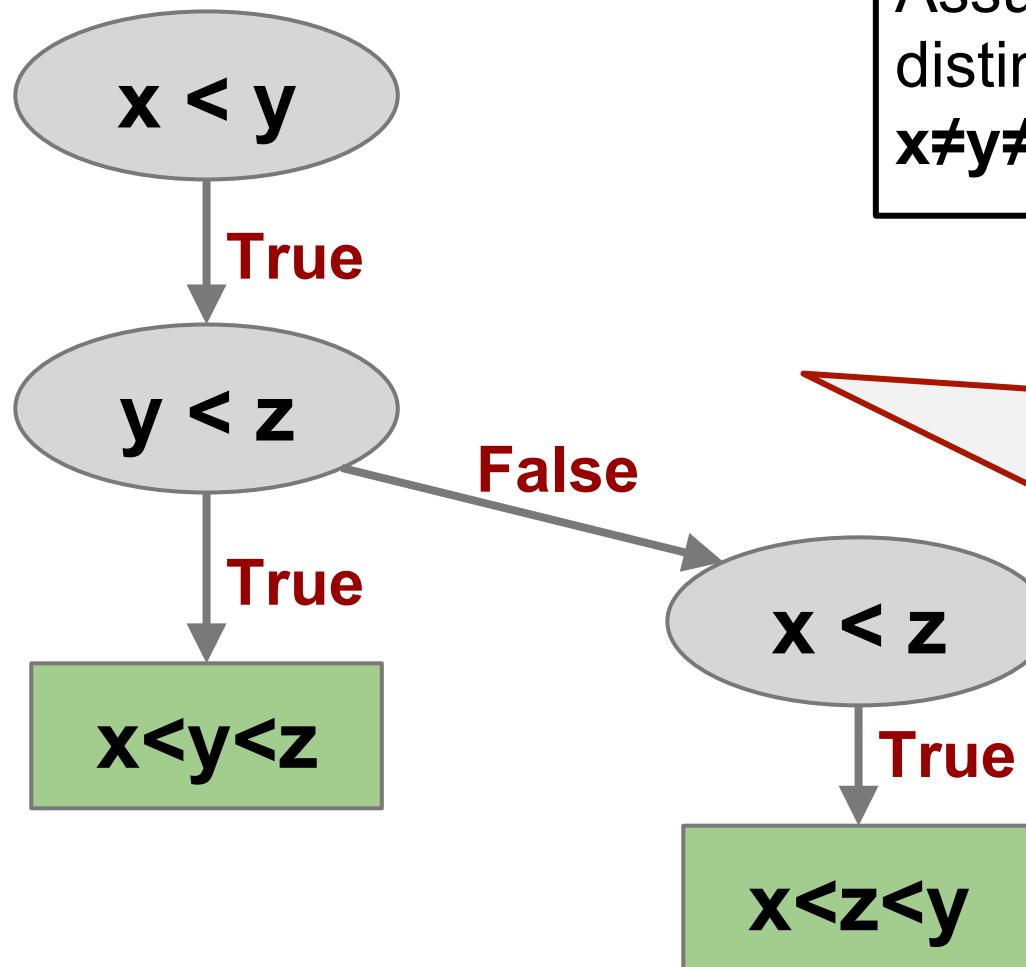
2	1	1	2	2	2	1
---	---	---	---	---	---	---

- Go through the array, count the number of 1's, namely, k
- Then output an array with k 1's followed by $n-k$ 2's
- This takes $O(n)$.

**Now prove it
the worst-case runtime of comparison
based sorting algorithms is in $\Omega(n \log n)$**



Sort $\{x, y, z\}$ via comparisons

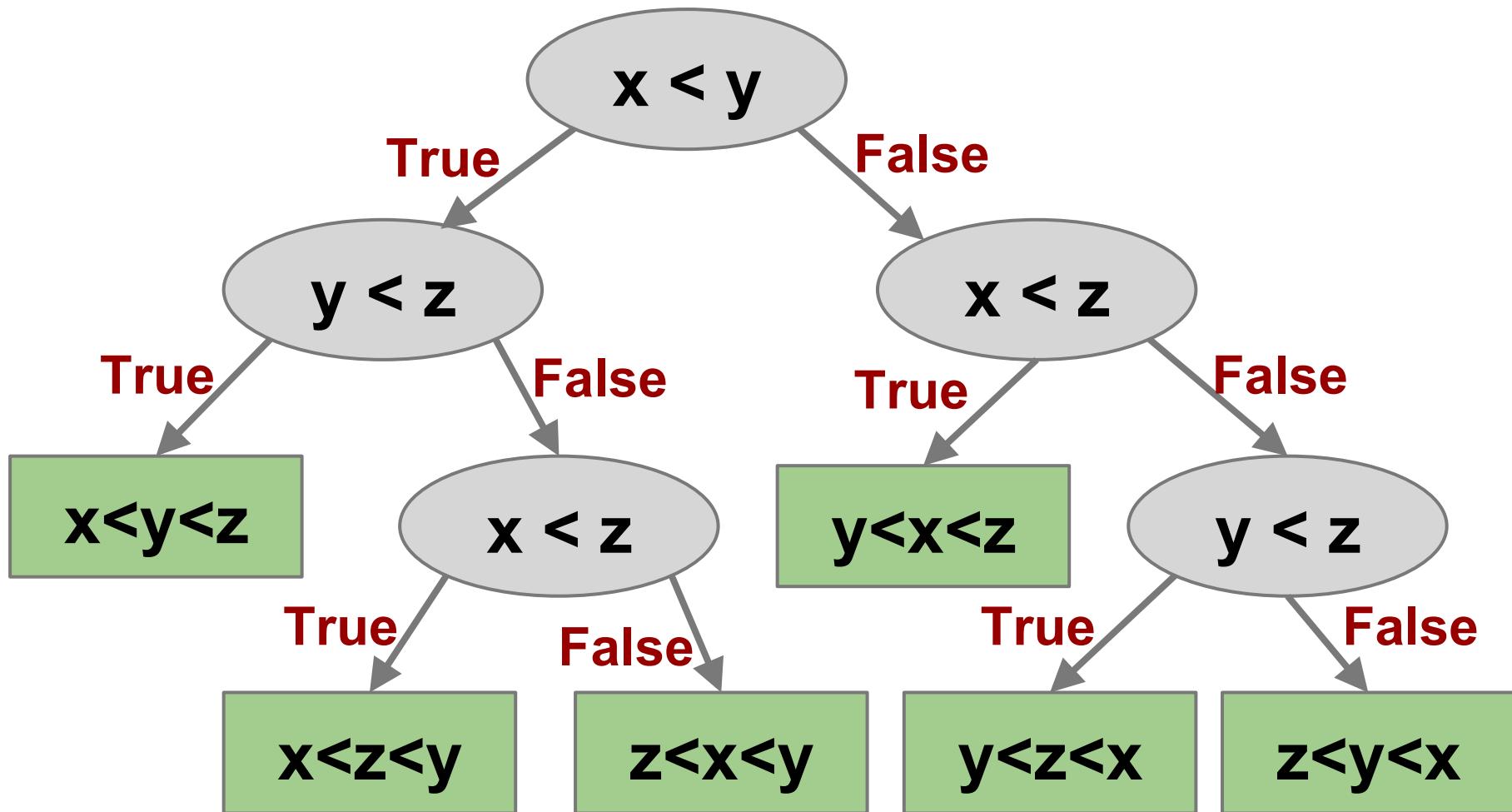


Assume x, y, z are distinct values, i.e.,
 $x \neq y \neq z$

A tree that is used to decide what the sorted order of x, y, z should be ...

The decision tree for sorting $\{x, y, z\}$

a tree that contains a complete set of decision sequences

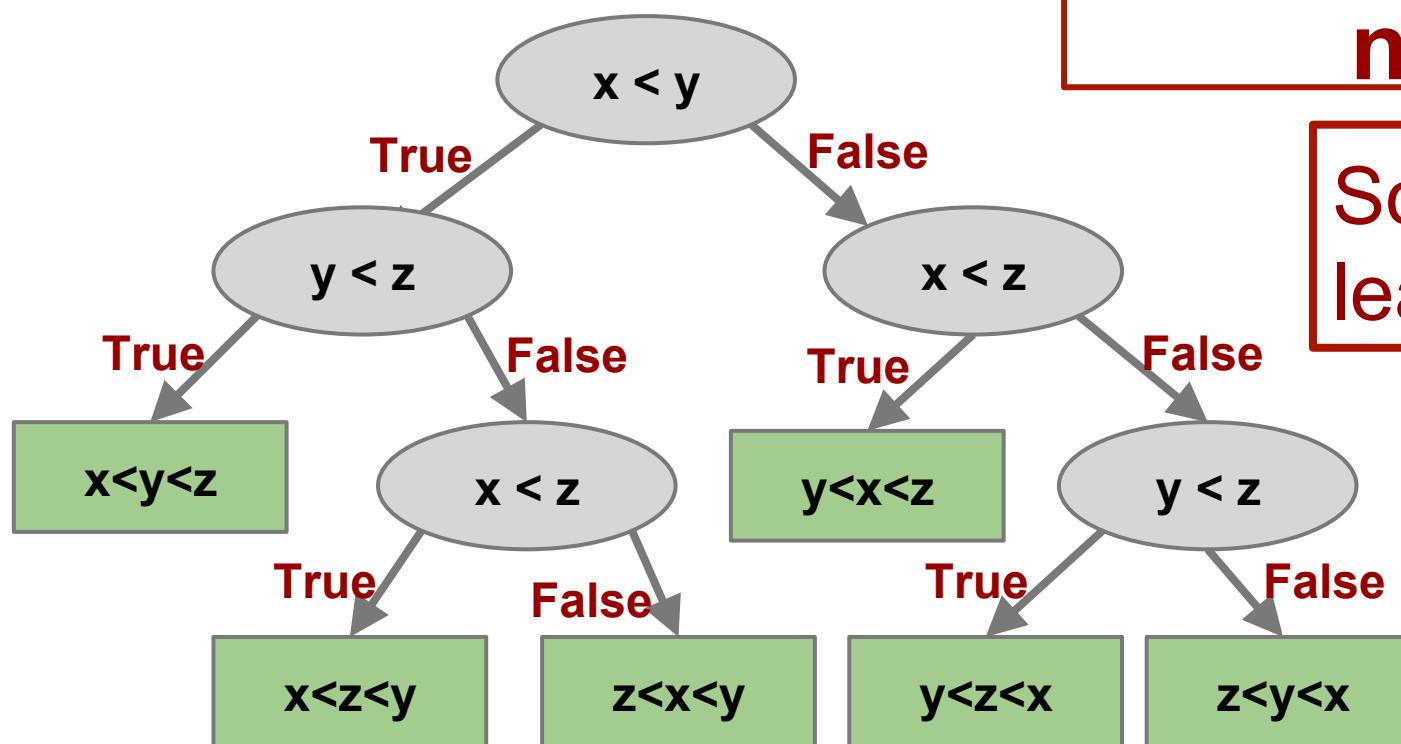


Each **leaf node** corresponds to a possible sorted order of $\{x, y, z\}$, a decision tree need to contain **all possible orders**.

How many possible orders for n elements?

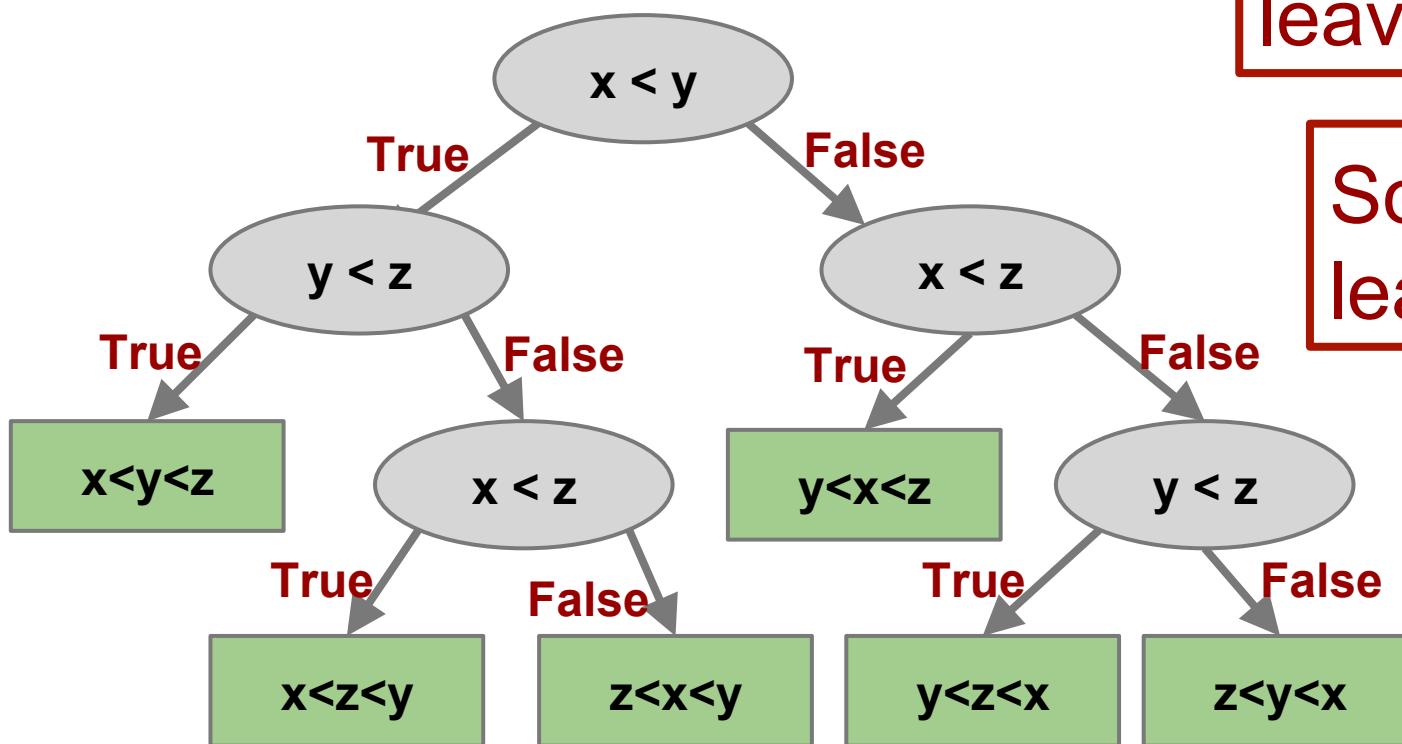
$n!$

So number of leaves $L \geq n!$



Now think about the **height** of the tree

A binary tree with height h has at most 2^h leaves



So number of leaves $L \leq 2^h$

So number of leaves $L \geq n!$

So,

$$2^h \geq n!$$

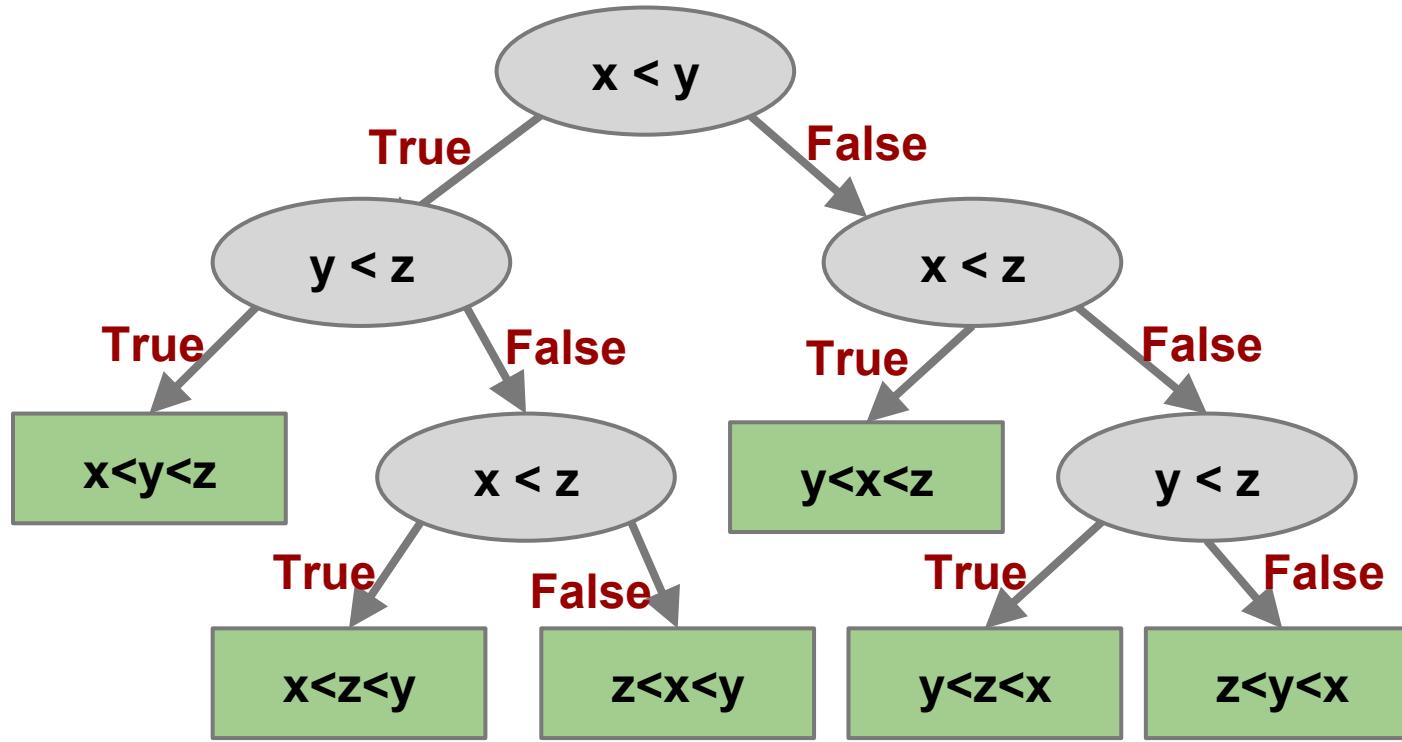
Not trivial, will
show it later

$$h \geq \log(n!) \in \Omega(n \log n)$$

So number of
leaves $L \leq 2^h$

So number of
leaves $L \geq n!$

$$h \in \Omega(n \log n)$$



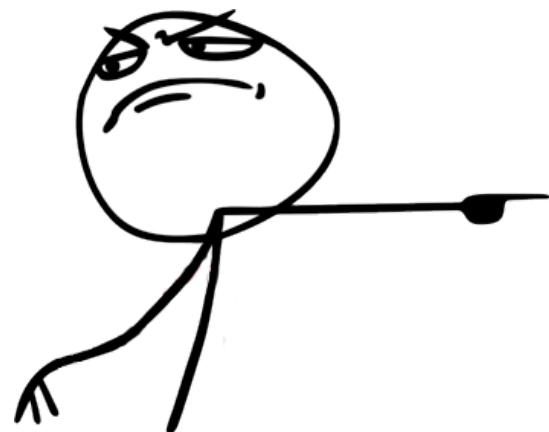
What does **h** represent, really?

The worst-case # of comparisons to sort!

$$h \in \Omega(n \log n)$$

What did we just show?

The worst-case number of comparisons needed to sort n elements is in $\Omega(n \log n)$



Lower bound proven!

Appendix: the missing piece

Show that $\log(n!)$ is in $\Omega(n \log n)$

$\log(n!)$

$$= \log 1 + \log 2 + \dots + \log n/2 + \dots + \log n$$

$$\geq \log n/2 + \dots + \log n \quad (n/2 + 1 \text{ of them})$$

$$\geq \log n/2 + \log n/2 + \dots + \log n/2 \quad (n/2 + 1 \text{ of them})$$

$$\geq n/2 \cdot \log n/2$$

$$\in \Omega(n \log n)$$

Often the number of possible solutions is small so we can't use the previous easy strategy.

**A more general lower bound tool:
The Adversary Method**



How does your opponent smartly **cheat** in this game?

- While you ask questions, the opponent alters their ships' positions so that they can “**miss**” whenever possible, i.e., construct the **worst possible input** (layout) **based on your questions**.
- They won’t get caught as long as their answers are **consistent** with one possible input.

If we can prove that, no matter what sequence of questions you ask, the opponent can always craft an input such that it takes at least **42 guesses** to sink a ship.

Then we can say the **lower bound** on the complexity of the “sink-a-ship” problem is **42 guesses**, no matter what “guessing algorithm” you use.

more formally ...

To prove a lower bound $L(n)$ on the complexity of problem P ,

we show that for every algorithm A and arbitrary input size n , there exists some input of size n (picked by an imaginary adversary) for which A takes at least $L(n)$ steps.

Example: search unsorted array

Problem:

Given an unsorted array of n elements, return the **index** at which the value is **42**.
(assume that **42** must be in the array)

3	5	2	42	7	9	8
---	---	---	----	---	---	---

Possible algorithms

- Check through indices 1, 2, 3, ..., n
- Check from n, n-1, n-2, ..., to 1
- Check all odd indices 1, 3, 5, ..., then check all even indices 2, 4, 6, ...
- Check in the order 3, 1, 4, 1, 5, 9, 2, 6, ...

Prove: the **lower bound** on this problem is **n-1**, no matter what algorithm we use.

3	5	2	42	7	9	8
---	---	---	----	---	---	---

Proof: (using adversarial argument)

- Let **A** be an **arbitrary** algorithm in which the first $n-1$ indices checked are **i₁, i₂, ..., i_{n-1}**
- Construct (adversarially) an input array **L** such that **L[i₁], L[i₂], ..., L[i_{n-1}]** are **not 42**, and **L[i_n]** is **42**.
- Because **A** is arbitrary, therefore the lower bound on the complexity of solving this problem is **n**, no matter what algorithm is used.

The problem

Given n elements, determine the **maximum** element.

How many comparisons are needed **at least**?

The problem

Given n elements, determine the **maximum** element.

How many comparisons are needed **at least?**

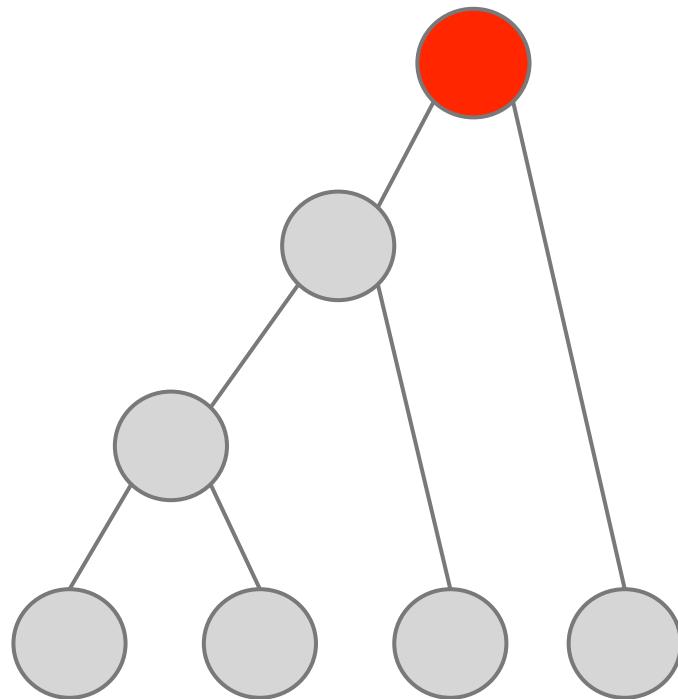
Answer: Need at least **$n-1$** comparisons

Insight: upper bound for max

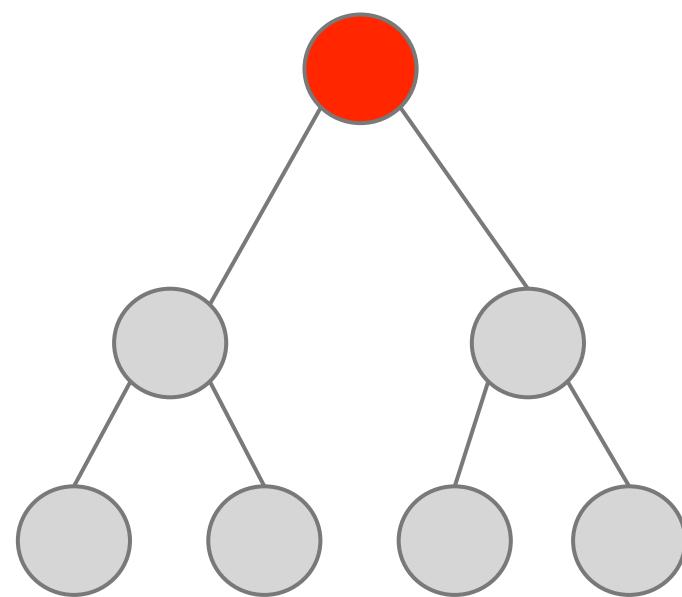
How to design a maximum-finding algorithm that reaches the lower bound $n-1$?

- Make every comparison **count**, i.e., every comparison should guarantee to **eliminate a possible candidate** for maximum/champion.
- No match between losers, because neither of them is a candidate for champion.
- No match between a candidate and a loser, because if the candidate wins, the match makes no contribution (not eliminating a candidate)

These algorithms reach the lower bound



Linear scanning



Tournament

Adversary strategy for Max

Suppose Algorithm A claims to find the max of n elements using $< n-1$ comparisons (on some path)

Construct a graph in which we join two elements by an edge if they are compared (along this path) by A.

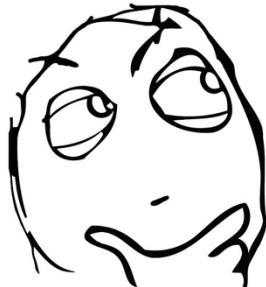
Since $< n-1$ comparisons on this path, the underlying graph has at least 2 components, C1 and C2

Suppose A outputs u in component C1 (as max)

Then we can fix values for elements in C1, C2 to be consistent with the comparisons, and where every element in C2 is larger than u . Contradiction!

Challenge question

Given n elements, what is the lower bound on the number of comparisons needed to determine both the **maximum** element and the **minimum** element?



Hint: it is smaller than $2(n-1)$

**proving lower bounds
using Reduction**

The idea

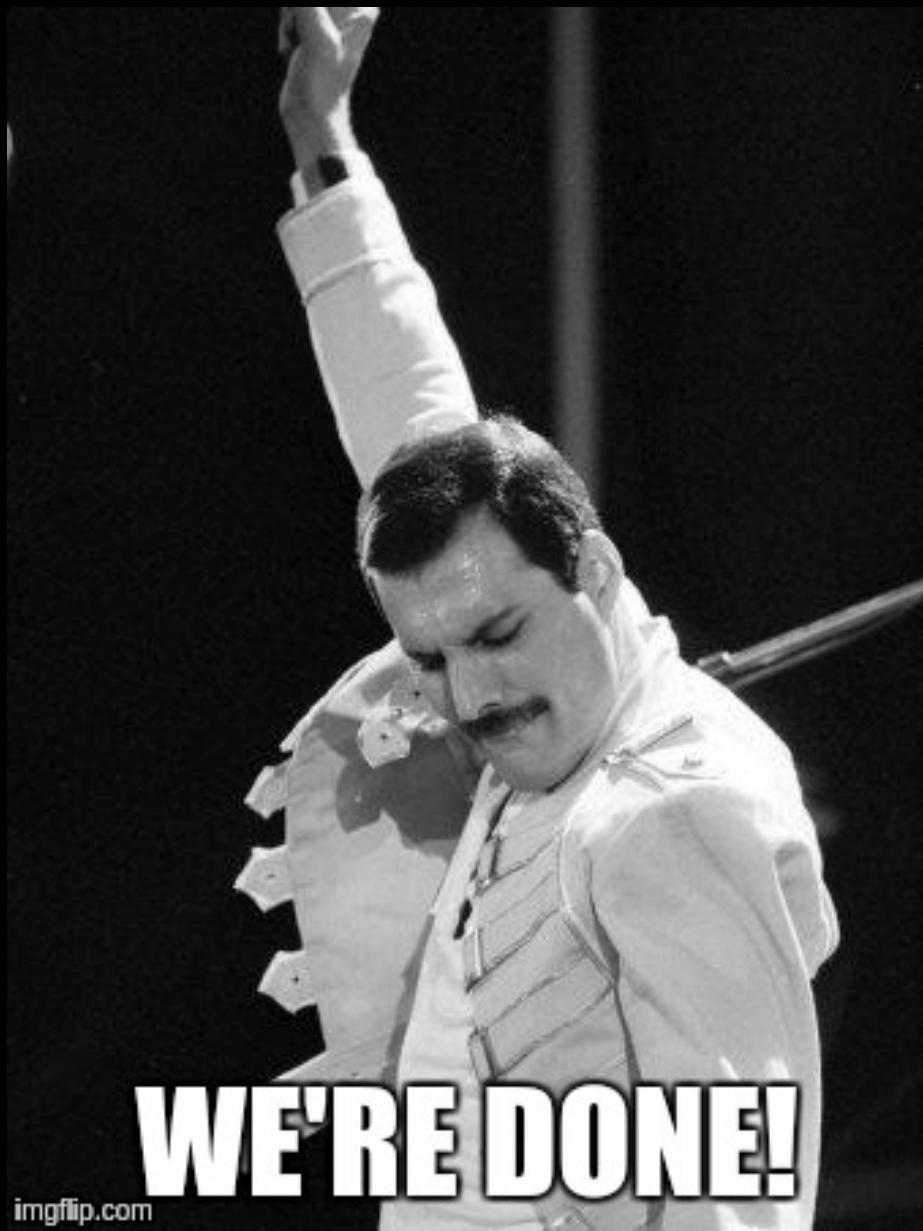
- Proving one problem's lower bound using another problem's known lower bound.
- If we know problem **B** can be solved by solving an instance of problem **A**, i.e., **A** is “harder” than **B**
- and we know that **B** has lower bound $L(n)$
- then **A** must also be lower-bounded by $L(n)$

Example:

Prove: ExtractMax on a binary heap is lower bounded by $\Omega(\log n)$.

Suppose ExtractMax can be done faster than $\log n$, then HeapSort can be done faster than $n \log n$, because HeapSort is basically ExtractMax n times

But HeapSort, as a comparison based sorting algorithm, has been proven to be lower bounded by $\Omega(n \log n)$. Contrdiction, so ExtractMax must be lower bounded by $\Omega(\log n)$



WE'RE DONE!

Final thoughts

**what did we learn in
CSC263**

Data structures are the underlying skeleton of a good computer system.

If you will get to design such a system yourself and make fundamental decisions, what you learned from CSC263 should give you some clues on what to do.

- Understand the nature of the system / problem, and model them into structured data
- Investigate the probability distribution of the input
- Investigate the real cost of operations
- Make reasonable assumptions and estimates where necessary
- Decide what you care about in terms of performance, and analyse it
 - ◆ “No user shall experience a delay more than 500 milliseconds” -- worst-case analysis
 - ◆ “It’s ok some rare operations take a long time” -- average-case analysis
 - ◆ “what matter is how fast we can finish the whole sequence of operations” -- amortized analysis

In CSC263, we learned to be
a computer scientist,
not just a programmer.

Original words from lecture notes of Michelle Craig

what we did NOT learn

but are now ready to learn

Other (even better!) kinds of heaps

- Sometimes we want to be able to **merge** two heaps into one heap, with binary heap we can do it in **$O(n)$** time worst-case.
- Using **binomial heap**, we can do merge in **$O(\log n)$** time worst-case
- Using **Fibonacci heap**, we can do merge (as well as Max/Insert/IncreaseKey) in **$O(1)$** time amortized.

Even better kinds of search trees

- We learned BST and AVL tree, and there are others called red-black tree, 2-3 tree, splay tree, AA tree, scapegoat tree, etc.
- There is **B-tree**, optimized for accessing big blocks of data (like in a hard drive)
- There is **B+ tree**, which is even better than B-tree (widely used in database systems).
- You'll learn about these in CSC443.

Amazing applications of hashing

- **Perfect hashing** guarantees **worst-case** $O(1)$ time for searching, instead of **average-case** $O(1)$ time
- **Cuckoo hashing** (coolest thing ever)

Shortest paths in a graph

- We learned how to get shortest paths using BFS on a graph
- We did NOT learn how to get **shortest (weighted) paths** on a weighted graph.
 - ◆ Dijkstra, Bellman-Ford, ...
- You'll learn about them in CSC 373

Greedy algorithms

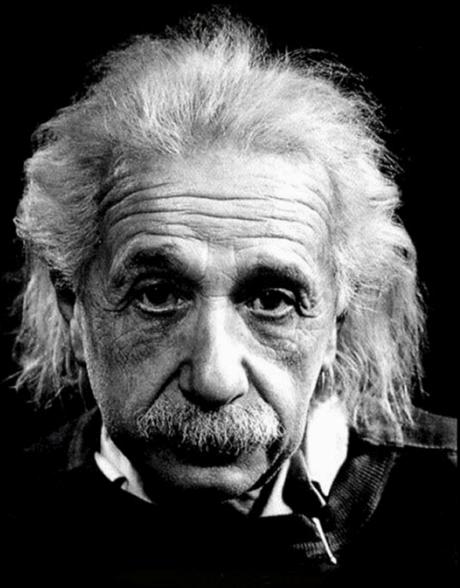
- We learned that Kruskal's and Prim's MST algorithms are greedy
- What property is satisfied by the problems that can be perfectly solved by greedy algorithms?
- Will learn in CSC373

Dynamic programming

- Pick an interesting algorithm design problem, very likely it involves dynamic programming
- Will learn in CSC373

P vs NP, approximation algorithms

- We learned a bit about lower bounds.
- There are some problems, we can prove they cannot be perfectly solved in polynomial time.
- For these problems, we have to design some **approximation algorithms**.
- Will learn in CSC373 / 463



As our circle of knowledge
expands, so does the
circumference of darkness
surrounding it.

Final Exam Prep

Topics covered: all of them

- Heaps
- BST, AVL tree, augmentation
- Hashing
- Randomized algorithms, Quicksort
- Graphs, BFS, DFS, MST
- Disjoint sets
- Lower bounds
- Analysis: worst-case, average-case, amortized.

Types of questions

- Short-answer questions testing basic understanding.
- Trace operations we learned on a data structure
- Implement an ADT using a data structure
- Analysis runtimes
 - ◆ best / worst-case
 - ◆ average-case
 - ◆ amortized cost
- Given a real-world problem, design data structures / algorithms to solve it.

Study for the exam

- Review lecture notes/slides
- Review tutorial problems
- Review all problem sets / assignments
- Practice with past exams (available at
exam repository)
- Come to **office hours** whenever
confused.

Toni's pre-exam office hours

- Monday Dec 7, 3-4pm
- Wednesday Dec 9, 1-2pm

Exam Time & Location

Friday, Dec 11, 2:00 - 5:00 pm

No aid sheet

Go to the right location.

All the best!