



# CSE332: Data Abstractions

## Lecture 1: Intro; ADTs; Stacks/Queues

James Fogarty

Winter 2012

# *Terminology*

- **Abstract Data Type (ADT)**
  - Mathematical description of a “thing” with set of operations
- **Algorithm**
  - A high level and language-independent description of a step-by-step process
- **Data Structure**
  - A specific family of algorithms for implementing an ADT
- **Implementation**
  - A specific instantiation in a specific language

# Example: Stacks

- The **Stack ADT** supports operations:
  - **isEmpty**: have there been same number of pops as pushes
  - **push**: takes an item
  - **pop**: raises an error if isEmpty, else returns most-recently pushed item not yet returned by a pop
  - Often some more operations
- A Stack **data structure** could use a linked-list or an array or something else, with associated **algorithms** for the operations
- One **implementation** is in the library `java.util.Stack`

# *Why is a Stack Useful*

The Stack ADT is a useful abstraction because:

- It arises **all the time** in programming (see Weiss 3.6.3)
  - Recursive function calls
  - Balancing symbols (parentheses)
  - Evaluating postfix notation:  $3\ 4\ +\ 5\ *$
  - Infix  $((3+4) * 5)$  to postfix conversion
- We can code up a **reusable library**
- We can **communicate** in high-level terms
  - “Use a stack and push numbers, popping for operators...”
  - Rather than, “create a linked list and add a node when...”

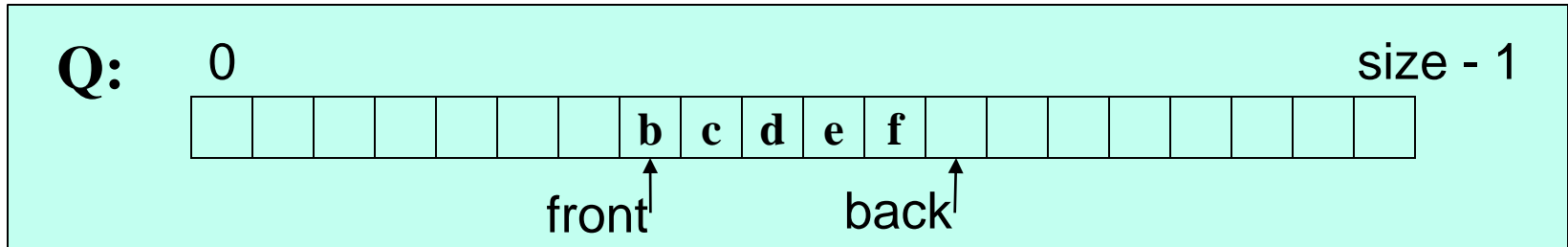
# The Queue ADT

- Operations  
`create`  
`destroy`  
`enqueue`  
`dequeue`  
`is_empty`



- Just like a stack except:
  - Stack: LIFO (last-in-first-out)
  - Queue: FIFO (first-in-first-out)
- Just as useful and ubiquitous

# Circular Array Queue Data Structure

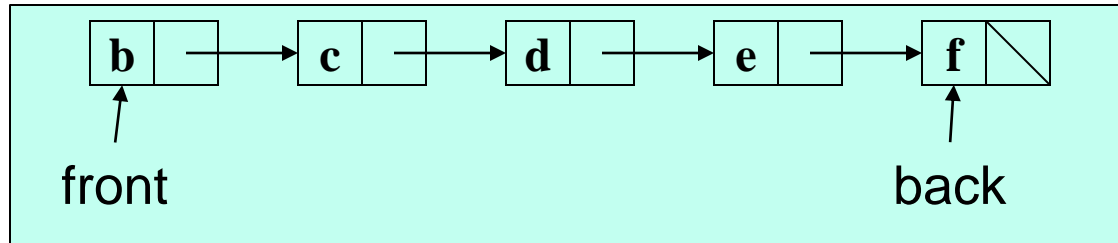


```
// Basic idea only!  
enqueue(x) {  
    Q[back] = x;  
    back = (back + 1) % size  
}
```

```
// Basic idea only!  
dequeue() {  
    x = Q[front];  
    front = (front + 1) % size;  
    return x;  
}
```

- What if **queue** is empty?
  - Enqueue?
  - Dequeue?
- What if **array** is full?
- How to *test* for empty?
- What is the *complexity* of the operations?
- Can you find the  $k^{\text{th}}$  element in the queue?

# Linked List Queue Data Structure



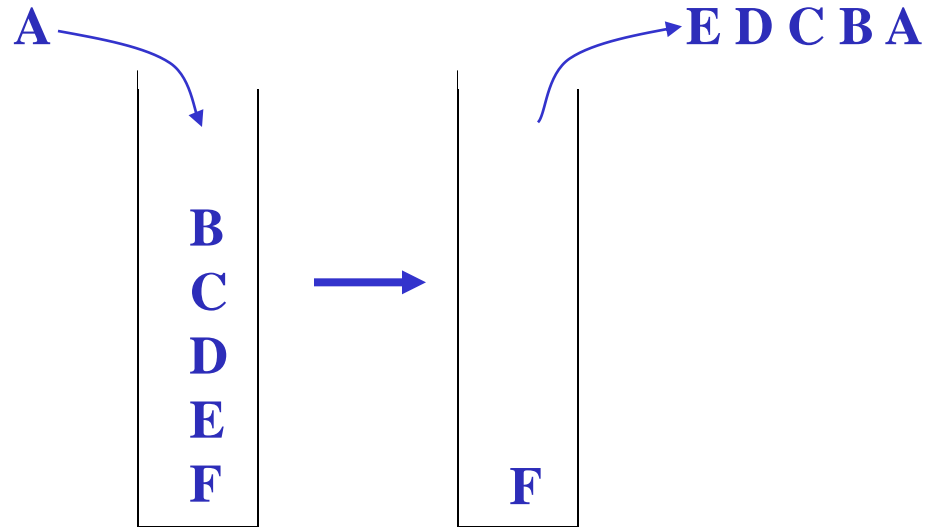
```
// Basic idea only!  
enqueue(x) {  
    back.next = new Node(x);  
    back = back.next;  
}
```

```
// Basic idea only!  
dequeue() {  
    x = front.item;  
    front = front.next;  
    return x;  
}
```

- What if **queue** is empty?
  - Enqueue?
  - Dequeue?
- Can **list** be full?
- How to *test* for empty?
- What is the *complexity* of the operations?
- Can you find the  $k^{\text{th}}$  element in the queue?

# The Stack ADT

- Operations
  - `create`
  - `destroy`
  - `push`
  - `pop`
  - `top`
  - `is_empty`



- Can also be implemented with an array or a linked list
  - This is Project 1!
  - As with queues, type of elements is irrelevant
    - Ideal for Java's generic types (Project 1B)



# *Array vs. Linked List Implementations*

## Array:

- May waste unneeded space or run out of space
- Space per element excellent
- Operations very simple / fast
- Constant-time access to  $k^{\text{th}}$  element
- For operation `insertAtPosition`, must shift elements
  - But not part of these ADTs

## List:

- Always just enough space
- But more space per element
- Operations very simple / fast
- No constant-time access to  $k^{\text{th}}$  element
- For operation `insertAtPosition` must traverse elements
  - But not part of these ADTs

This is something every trained computer scientist knows in their sleep. It's like knowing how to do arithmetic or ride a bike.



# CSE332: Data Abstractions

## Lecture 2: Math Review; Algorithm Analysis

Tyler Robison (covering for James Forgarty)

Winter 2012

# Proof via mathematical induction

Suppose  $P(n)$  is some rule involving  $n$

– Example:  $n \geq n/2 + 1$ , for all integers  $n \geq 2$

To prove  $P(n)$  for all integers  $n \geq c$ , it suffices to prove

1.  $P(c)$  – called the “basis” or “base case”
2. If  $P(k)$  then  $P(k+1)$  – called the “induction step” or “inductive case”

Why we will care:

Use to show that an algorithm is correct or has a certain running time *no matter how big a data structure or input value is* (Our “ $n$ ” will be the data structure or input size.)

# Example

$P(n)$  = “the sum of the first  $n$  powers of 2 (starting at  $2^0$ ) is the next power of 2 minus 1”

Theorem:  $P(n)$  holds for all integers  $n \geq 1$

$$1=2-1$$

$$1+2=4-1$$

$$1+2+4=8-1$$

So far so good...

# Example

Theorem:  $P(n)$  holds for all  $n \geq 1$

Proof: By induction on  $n$

- Base case,  $n=1$ :  $2^0 = 1 = 2^1 - 1$
- Inductive case: If it holds for  $k$ , then it holds for  $k+1$ 
  - Inductive hypothesis: Assume the sum of the first  $k$  powers of 2 is  $2^k - 1$
  - Show, given the hypothesis, that the sum of the first  $(k+1)$  powers of 2 is  $2^{k+1} - 1$

From our inductive hypothesis we know:

$$1 + 2 + 4 + \dots + 2^{k-1} = 2^k - 1$$

Add the next power of 2 to both sides...

$$1 + 2 + 4 + \dots + 2^{k-1} + 2^k = 2^k - 1 + 2^k$$

We have what we want on the left; massage the right a bit

$$1 + 2 + 4 + \dots + 2^{k-1} + 2^k = 2(2^k) - 1 = 2^{k+1} - 1$$

# Another Example

For all  $n \geq 1$

$$1+2+3+\dots+(n-1)+n = n(n+1)/2$$

$$\text{Ex: } 1+2+3+4+5+6 = 6*7/2 = 21$$

Proof: By induction on  $n$

- Base case,  $n=1$ :  $1=1*(1+1)/2$
- Inductive case:
  - Inductive hypothesis: Assume the sum of the first  $k$  integers (from 1 up) equals  $k(k+1)/2$
  - Show, given the hypothesis, that it holds true for the next integer ( $k+1$ )

From our inductive hypothesis we know:

$$1+2+3+\dots+k = k(k+1)/2$$

Add  $k+1$  to both sides...

$$1+2+3+\dots+k + (k+1) = k(k+1)/2 + (k+1)$$

We have what we want on the left; massage the right a bit

$$1+2+3+\dots+k + (k+1) = (k(k+1) + 2(k+1))/2 = (k^2+k+2k+2)/2 = (k+1)(k+2)/2$$

# Note for homework

Proofs by induction may come up in the homework

When doing them, be sure to state each part clearly:

- What you're trying to prove
- The base case
- The inductive case
- The inductive hypothesis

# Powers of 2

- A bit is 0 or 1
- A sequence of  $n$  bits can represent  $2^n$  distinct things
  - For example, the numbers 0 through  $2^n-1$
- $2^{10}$  is 1024 (“about a thousand”, kilo in CSE speak)
- $2^{20}$  is “about a million”, mega in CSE speak
- $2^{30}$  is “about a billion”, giga in CSE speak

Java: an **int** is 32 bits and signed, so “max int” is “about 2 billion”

a **long** is 64 bits and signed, so “max long” is  $2^{63}-1$



# Therefore...

We could give a unique id to...

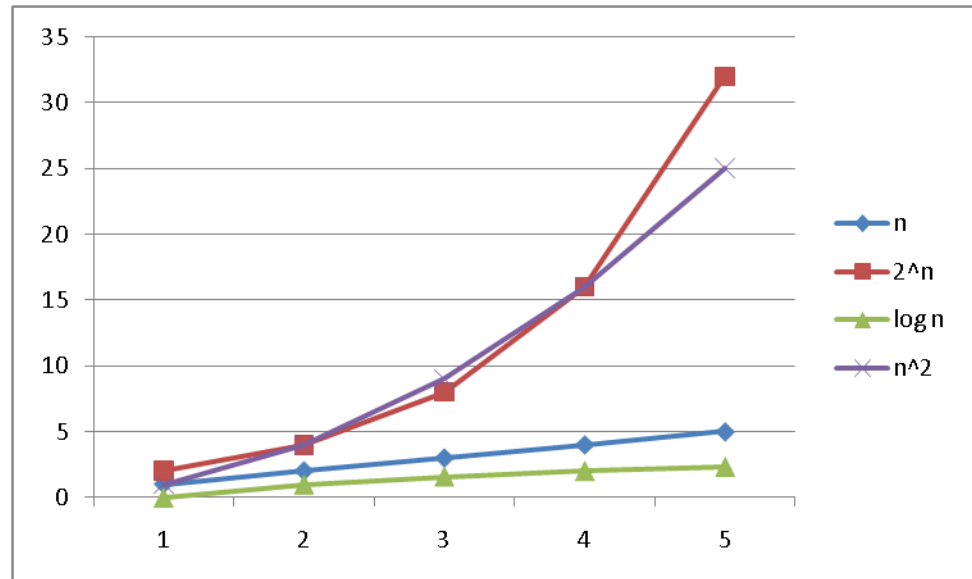
- Every person in this room with 7 bits
- Every person in the U.S. with 29 bits
- Every person in the world with 33 bits
- Every person to have ever lived with 38 bits (estimate)
- Every atom in the universe with 250-300 bits

So if a password is 128 bits long and randomly generated,  
do you think you could guess it?

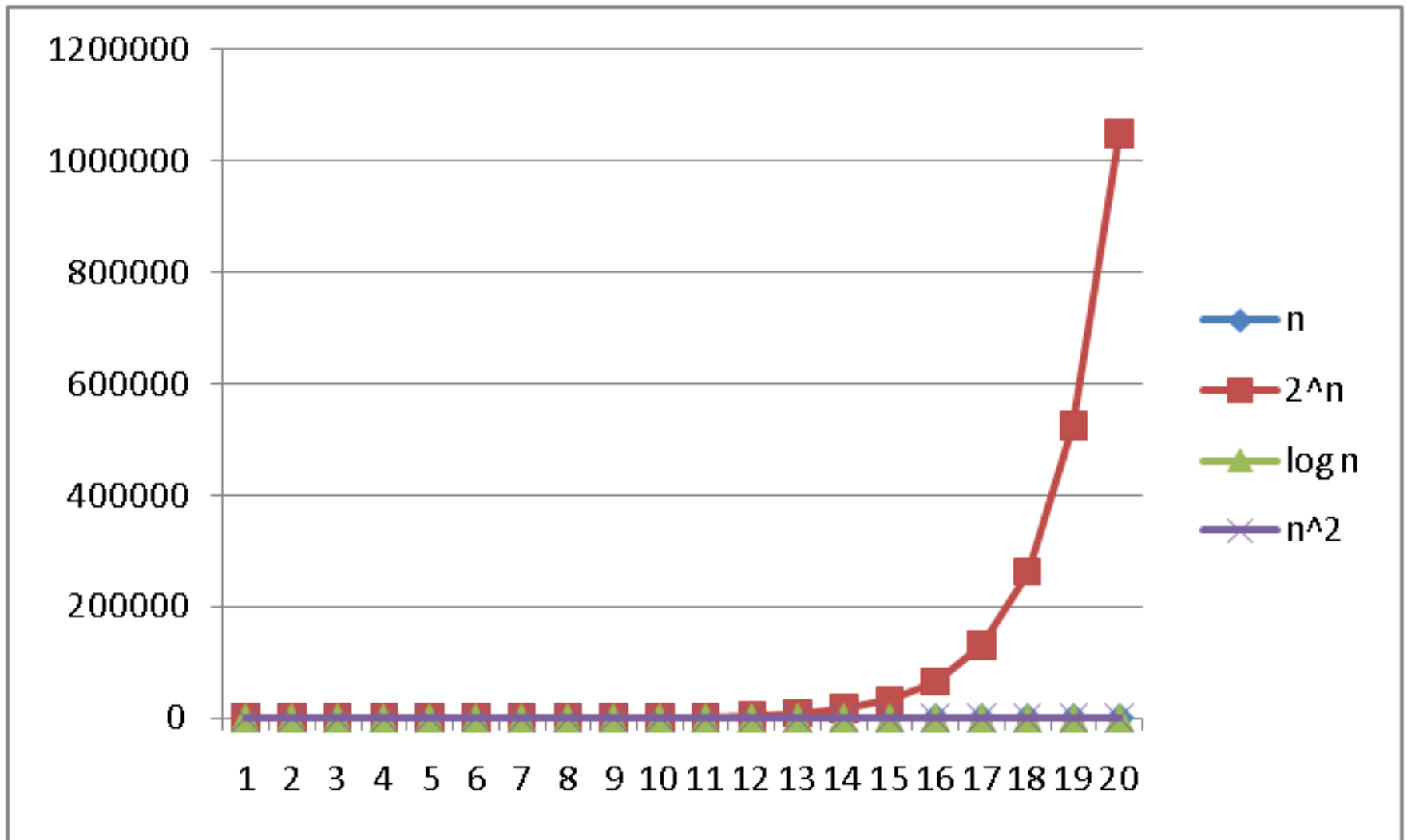
# Logarithms and Exponents

- Since so much is binary in CS, **log** almost always means **log<sub>2</sub>**
- Definition: **log<sub>2</sub> x = y** iff **x = 2<sup>y</sup>**
- So, **log<sub>2</sub> 1,000,000 = “a little under 20”**

Just as exponents grow *very* quickly, logarithms grow *very* slowly



# Logarithms and Exponents



# Properties of logarithms

- $\log(A*B) = \log A + \log B$ 
  - So  $\log(N^k) = k \log N$
- $\log(A/B) = \log A - \log B$
- $\log_2 2^x = x$
- $\log(\log x)$  is written  $\log \log x$ 
  - Grows as slowly as  $2^{2^x}$  grows fast
  - Ex:  $\log_2 \log_2 4\text{billion} \sim \log_2 \log_2 2^{32} = \log_2 32 = 5$
- $(\log x)(\log x)$  is written  $\log^2 x$ 
  - It is greater than  $\log x$  for all  $x > 2$

# Log base doesn't matter (much)

“Any base  $B$  log is equivalent to base 2 log within a constant factor”

- And we are about to stop worrying about constant factors!
- In particular,  $\log_2 x = 3.22 \log_{10} x$
- In general, we can convert log bases via a constant multiplier
- Say, to convert from base  $B$  to base  $A$ :

$$\log_B x = (\log_A x) / (\log_A B)$$

$$\log_{10} x = (\log_2 x) / (\log_2 10)$$

# Algorithm Analysis

As the “size” of an algorithm’s input grows

(length of array to sort, size of queue to search, etc.):

- How much longer does the algorithm take (time)
- How much more memory does the algorithm need (space)

We are generally concerned about approximate runtimes

- Whether  $T(n)=3n+2$  or  $T(n)=n/4+8$ , we say it runs in linear time
- Common categories:
  - Constant:  $T(n)=1$
  - Linear:  $T(n)=n$
  - Logarithmic:  $T(n)=\log n$
  - Exponential:  $T(n)=2^n$

# Example

- First, what does this pseudocode return?

```
x := 0;  
for i=1 to n do  
  for j=1 to i do  
    x := x + 3;  
return x;
```

- For any  $n \geq 0$ , it returns  $3n(n+1)/2$
- Why?
  - Consider, how many times does the inner loop run?
  - For  $i=1$ , it runs once
  - For  $i=2$ , it runs twice
  - Etc.
  - $1+2+3+\dots+n = n(n+1)/2$
  - $x$  gets raised by 3 each time

# Example

- How long does this pseudocode run?

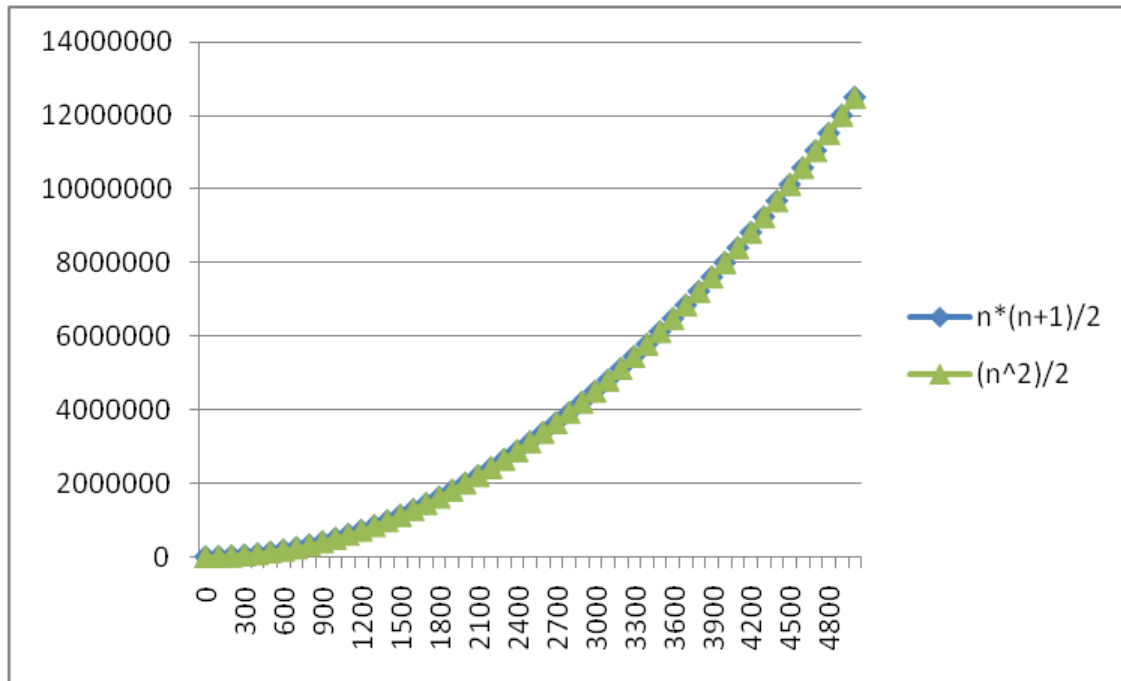
```
x := 0;  
for i=1 to n do  
  for j=1 to i do  
    x := x + 3;  
return x;
```

- Find running time in terms of  $n$ , for any  $n \geq 0$ 
  - Assignments, additions, simple comparisons, etc. take “1 unit time”
    - Constant time
  - Loops take the sum of the time for their iterations
- Say, (roughly)  $2+5*(\text{number of times inner loop runs})$ 
  - Inner loop runs  $n(n+1)/2$  times
  - So  $O(n^2)$  time



# Lower-order terms don't matter for our purposes

$n*(n+1)/2$  vs. just  $n^2/2$



We'll discuss why on Monday

In essence, we're mostly concerned with behavior as  $n$  approaches infinity

# Big Oh (also written Big-O)

- Big Oh is used for comparing asymptotic behavior of functions
- We'll get into the definition later, but for now:
  - 'f(n) is O(g(n))' roughly means
    - The function f(n) is at least as small as g(n) as they go toward infinity
    - Think of it as a  $\leq$  for functions
  - BUT: Big Oh ignores constant factors
    - $n+10$  is  $O(n)$ ; we drop out the '+10'
    - $5n$  is  $O(n)$ ; we drop out the 'x5'
    - The following is NOT true though:  $n^2$  is  $O(n)$
  - Also note that 'f(n) is O(g(n))' gives an upper bound for f(n)
    - $n$  is  $O(n^2)$
    - $5$  is  $O(n)$

# Big Oh: Common Categories

*From fastest to slowest*

$O(1)$	constant (same as $O(k)$ for constant $k$ )
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	“ $n \log n$ ”
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^k)$	polynomial (where $k$ is a constant)
$O(k^n)$	exponential (where $k$ is any constant $> 1$ )

Usage note: “exponential” does not mean “grows really fast”, it means “grows at rate proportional to  $k^n$  for some  $k > 1$ ”

- A savings account accrues interest exponentially ( $k=1.01$ ?)



# CSE332: Data Abstractions

## Lecture 3: Asymptotic Analysis

Tyler Robison (covering for James Forgarty)  
Winter 2012

# What do we want to analyze?

- Correctness
- Performance: Algorithm's speed or memory usage: our focus
  - Change in speed as the input grows
    - $n$  increases by 1
    - $n$  doubles
  - Comparison between 2 algorithms
- Security
- Reliability
- Sometimes other properties ('stable' sorts)

# Gauging performance

- Uh, why not just run the program and time it?
  - Too much variability; not reliable:
    - Hardware: processor(s), memory, etc.
    - OS, version of Java, libraries, drivers
    - Choice of input
    - Programs running in the background, OS stuff, etc.: several executions on the same computer with the same settings may well yield different results
    - Implementation dependent
  - Timing doesn't really evaluate the algorithm; it evaluates its implementation in one very specific scenario
  - As computer scientists, we are more interested in the algorithm itself

# Gauging performance (cont.)

- At the core of CS is a backbone of theory & mathematics
  - Examine the algorithm itself, mathematically, not the implementation
  - Reason about performance as a function of  $n$ ; not just ‘it runs fast on this particular test file’
  - Be able to mathematically prove things about performance
- Yet, timing has its place
  - In the real world, we do want to know whether implementation A runs faster than implementation B on data set C
  - Ex: Benchmarking graphics cards
  - May do some timing in projects
- Evaluating an algorithm? Use asymptotic analysis
- Evaluating an implementation of hardware/software?  
Timing can be useful

# Big-Oh

- Say we're given 2 run-time functions  $f(n)$  &  $g(n)$  for input  $n$
- The Definition:  $f(n)$  is in  $O(g(n))$  iff there exist *positive* constants  $c$  and  $n_0$  such that

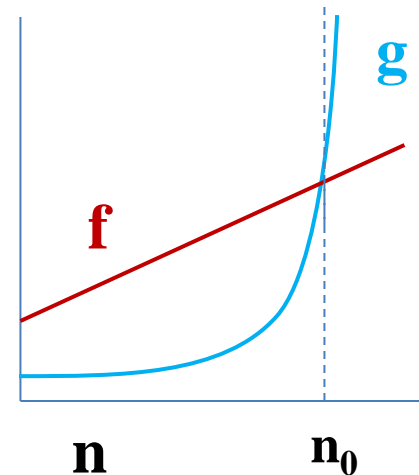
$$f(n) \leq c g(n), \text{ for all } n \geq n_0.$$

- The Idea: Can we find an  $n_0$  such that  $g$  is always greater than  $f$  from there on out?

$c$ : We are allowed to multiply  $g$  by a constant value (say, 10) to make  $g$  larger (more on why this is here in a moment)

$O(g(n))$  is really a set of functions whose asymptotic behavior is less than or equal that of  $g(n)$

Think of ' $f(n)$  is in  $O(g(n))$ ' as  $f(n) \leq g(n)$  (sort of)





# Big Oh (cont.)

- The Intuition:
  - Take functions  $f(n)$  &  $g(n)$ , consider only the most significant term and remove constant multipliers:
    - $5n+3 \rightarrow n$
    - $7n+.5n^2+2000 \rightarrow n^2$
    - $300n+12+n\log n \rightarrow n\log n$
    - $-n \rightarrow ???$  What does it mean to have a negative run-time?
  - Then compare the functions; if  $f(n) \leq g(n)$ , then  $f(n)$  is in  $O(g(n))$
  - Do NOT ignore constants that are not additions or multipliers:
    - $n^3$  is  $O(n^2)$  : **FALSE**
    - $3^n$  is  $O(2^n)$  : **FALSE**
  - When in doubt, refer to the definition (examples in a moment)

# Examples

• True or false?

1.  $4+3n$  is  $O(n)$  True

2.  $n+2\log n$  is  $O(\log n)$  False

3.  $\log n+2$  is  $O(1)$  False

4.  $n^{50}$  is  $O(1.01^n)$  True

5. There exists  $\alpha > 1.0$  s.t. False

$\alpha^n$  is  $O(n^\beta)$

For some finite  $\beta$

# Examples (cont.)

- For  $f(n)=4n$  &  $g(n)=n^2$ , prove  $f(n)$  is in  $O(g(n))$ 
  - A valid proof (for our purposes) is to find valid  $c$  &  $n_0$
  - When  $n=4$ ,  $f=16$  &  $g=16$ ; this is the crossing over point
  - Say  $n_0 = 4$ , and  $c=1$
  - How many possible answers  $(c, n_0)$  are there?
    - \*Infinitely many:  
ex:  $n_0 = 78$ , and  $c=42$

**The Definition:  $f(n)$  is in  $O(g(n))$  iff there exist *positive* constants  $c$  and  $n_0$  such that**

$$f(n) \leq c g(n) \text{ for all } n \geq n_0.$$

# Examples (cont.)

- For  $f(n)=n^3$  &  $g(n)=2^n$ , prove  $f(n)$  is in  $O(g(n))$ 
  - Possible answer:  $n_0=11$ ,  $c=1$

**The Definition:  $f(n)$  is in  $O(g(n))$  iff there exist *positive* constants  $c$  and  $n_0$  such that**

$$f(n) \leq c g(n) \text{ for all } n \geq n_0.$$

# What's with the c?

- To capture this notion of similar asymptotic behavior, we allow a constant multiplier (called c)
- Consider:  
     $f(n)=7n+5$   
     $g(n)=n$
- These have the same asymptotic behavior (linear), so  $f(n)$  is in  $O(g(n))$  even though  $f$  is always larger
- There is no  $n_0$  such that  $f(n) \leq g(n)$  for all  $n \geq n_0$
- The 'c' in the definition allows for that; it allows us to 'throw out constant factors'
- To prove  $f(n)$  is in  $O(g(n))$ , have  $c=12$ ,  $n_0=1$

# Big Oh: Common Categories

*From fastest to slowest*

$O(1)$	constant (same as $O(k)$ for constant $k$ )
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	“ $n \log n$ ”
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^k)$	polynomial (where $k$ is a constant)
$O(k^n)$	exponential (where $k$ is any constant $> 1$ )

Usage note: “exponential” does not mean “grows really fast”, it means “grows at rate proportional to  $k^n$  for some  $k > 1$ ”

- A savings account accrues interest exponentially ( $k=1.01$ ?)

Where does  $\log^2 n$  fit in?

Where does  $\log \log n$  fit in?

# Caveats

- Asymptotic complexity focuses on behavior of the algorithm for large  $n$  and is independent of any computer/coding trick, but results can be misleading
  - Example:  $n^{1/10}$  vs.  $\log n$ 
    - Asymptotically  $n^{1/10}$  grows more quickly
    - But the “cross-over” point is around  $5 * 10^{17}$
    - So if you have input size less than  $2^{58}$ , prefer  $n^{1/10}$

# More Caveats

- Even for more common functions, comparing  $O()$  for small  $n$  values can be misleading
  - Quicksort:  $O(n \log n)$  (expected)
  - Insertion Sort:  $O(n^2)$  (expected)
  - Yet in reality Insertion Sort is faster for small  $n$ 's
  - We'll learn about these sorts later
- Usually talk about an algorithm being  $O(n)$  or whatever
  - But you can also prove bounds for entire problems
  - Ex: Sorting cannot take place faster than  $O(n \log n)$  in the worst case (assuming it's sequential and comparison-based; more on these later)



# Miscellaneous

- Not uncommon to evaluate for:
  - Best-case
  - Worst-case
  - ‘Expected case’
- What are the run-times for BST lookup?
  - Best  $O(1)$  – find at root
  - Worst  $O(n)$  – tree is 1 long branch
  - ‘Expected’  $O(\log n)$  – complicated; see book

# Notational Notes

- We say  $(3n^2+17)$  **is in**  $O(n^2)$ 
  - Confusingly, we also say/write:
    - $(3n^2+17)$  **is**  $O(n^2)$
    - $(3n^2+17) = O(n^2)$  (very common; in the book)
      - But it's not '=' as in 'equality':
      - We would never say  $O(n^2) = (3n^2+17)$
- Perhaps the most accurate notation is  $f(n) \in O(g(n))$ 
  - Because  $O(g(n))$  is a set of functions

# Analyzing code (worst case)

Basic operations take “some amount of” constant time:

- Arithmetic (fixed-width)
- Assignment to a variable
- Access one Java field **or array index**
- Etc.

(This is an *approximation of reality*: a useful “lie”.)

Consecutive statements	Sum of times
Conditionals	Time of test plus slower branch
Loops	Sum of iterations
Calls	Time of call’s body
Recursion	Solve <i>recurrence equation</i>

# Analyzing code

What are the run-times for the following code:

1. `for(int i=0;i<n;i++)`  $O(1)$   $O(n)$
2. `for(int i=0;i<=n+100;i+=14)`  $O(1)$   $O(n)$
3. `for(int i=0;i<n;i++) for(int j=0;j<i;j++)`  $O(1)$   $O(n^2)$
4. `for(int i=0;i<n;i++) for(int j=0;j<n;j++)`  $O(n)$   $O(n^3)$
5. `for(int i=1;i<n;i*=2)`  $O(1)$   $O(\log n)$
6. `for(int i=0;i<n;i++) if(m(i))`  $O(n)$  else  $O(1)$  Depends on  $m()$ ; worst:  $O(n^2)$

# Big Oh's Family

- Big Oh: Upper bound:  $O(f(n))$  is the set of all functions asymptotically less than or equal to  $f(n)$ : ' $\leq$ ' of functions
  - $g(n)$  is in  $O(f(n))$  if there exist constants  $c$  and  $n_0$  such that
$$g(n) \leq c f(n) \text{ for all } n \geq n_0$$
- Big Omega: Lower bound:  $\Omega(f(n))$  is the set of all functions asymptotically greater than or equal to  $f(n)$ : ' $\geq$ ' of functions
  - $g(n)$  is in  $\Omega(f(n))$  if there exist constants  $c$  and  $n_0$  such that
$$g(n) \geq c f(n) \text{ for all } n \geq n_0$$
- Big Theta: Tight bound:  $\theta(f(n))$  is the set of all functions asymptotically equal to  $f(n)$ : '=' of functions
  - Intersection of  $O(f(n))$  and  $\Omega(f(n))$  (use *different constants*)

# Regarding use of terms

Common error is to say  $O(f(n))$  when you mean  $\theta(f(n))$

- People often say  $O()$  to mean a tight bound
- Say we have  $f(n)=n$ ; we could say  $f(n)$  is in  $O(n)$ , which is true, but only conveys the upper-bound
- Somewhat incomplete; instead say it is  $\theta(n)$
- This gives us a tighter bound

Less common notation:

- “little-oh”: like “big-Oh” but strictly less than
  - Example:  $n$  is  $o(n^2)$  but not  $o(n)$
- “little-omega”: like “big-Omega” but strictly greater than
  - Example:  $n$  is  $\omega(\log n)$  but not  $\omega(n)$

# Recurrence Relations

- Computing run-times gets interesting with recursion
- Say we want to perform some computation recursively on a list of size  $n$ 
  - Conceptually, in each recursive call we:
    - Perform some amount of work, call it  $w(n)$
    - Call the function recursively with a smaller portion of the list

So, if we do  $w(n)$  work per step, and reduce the  $n$  in the next recursive call by 1, we do total work:

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

With some base case, like  $T(1) = 5 = O(1)$

# Recursive version of sum array

Recursive:

- Recurrence is  
 $k + k + \dots + k$   
for  $n$  times

```
int sum(int[] arr) {  
    return help(arr, 0);  
}  
int help(int[] arr, int i) {  
    if (i == arr.length)  
        return 0;  
    return arr[i] + help(arr, i + 1);  
}
```

*Recurrence Relation:  $T(n) = O(1) + T(n-1)$*



# Recurrence Relations (cont.)

Say we have the following recurrence relation:

$$T(n) = 2 + T(n-1)$$

$$T(1) = 5$$

Now we just need to solve it; that is, reduce it to a closed form

Start by writing it out:

$$T(n) = 2 + T(n-1) = 2 + 2 + T(n-2) = 2 + 2 + 2 + T(n-3)$$

$$= 2 + 2 + 2 + \dots + 2 + T(1) = 2 + 2 + 2 + \dots + 2 + 5$$

$$= 2k + 5, \text{ where } k \text{ is the \# of times we expanded } T()$$

We expanded it out  $n-1$  times, so

$$T(n) = 2(n-1) + 5 = 2n + 3 = O(n)$$

# Example: Find k

2	3	5	16	37	50	73	75	126
---	---	---	----	----	----	----	----	-----

Find an integer in a *sorted* array

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k){
    ???
}
```

# Linear search

2	3	5	16	37	50	73	75	126
---	---	---	----	----	----	----	----	-----

Find an integer in a *sorted* array

```
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k){
    for(int i=0; i < arr.length; ++i)
        if(arr[i] == k)
            return true;
    return false;
}
```

Best case: 6ish steps =  $O(1)$

Worst case: 6ish\*(arr.length)  
=  $O(\text{arr.length}) = O(n)$

# Binary search

2	3	5	16	37	50	73	75	126
---	---	---	----	----	----	----	----	-----

Find an integer in a *sorted* array

- Can also be done non-recursively (same run-time)

```
// requires array is sorted
// returns whether k is in array
boolean find(int[] arr, int k){
    return help(arr,k,0,arr.length);
}
boolean help(int[] arr, int k, int lo, int hi) {
    int mid = (hi+lo)/2; //i.e., lo+(hi-lo)/2
    if(lo==hi)         return false;
    if(arr[mid]==k)    return true;
    if(arr[mid]< k)    return help(arr,k,mid+1,hi);
    else               return help(arr,k,lo,mid);
}
```

# Binary search

**Best case: 8ish steps =  $O(1)$**

**Worst case:**

**$T(n) = 10ish + T(n/2)$  where  $n$  is  $hi-lo$**

```
// requires array is sorted
// returns whether k is in array
boolean find(int[]arr, int k){
    return help(arr,k,0,arr.length);
}
boolean help(int[]arr, int k, int lo, int hi) {
    int mid = (hi+lo)/2;
    if(lo==hi) return false;
    if(arr[mid]==k) return true;
    if(arr[mid]< k) return help(arr,k,mid+1,hi);
    else return help(arr,k,lo,mid);
}
```

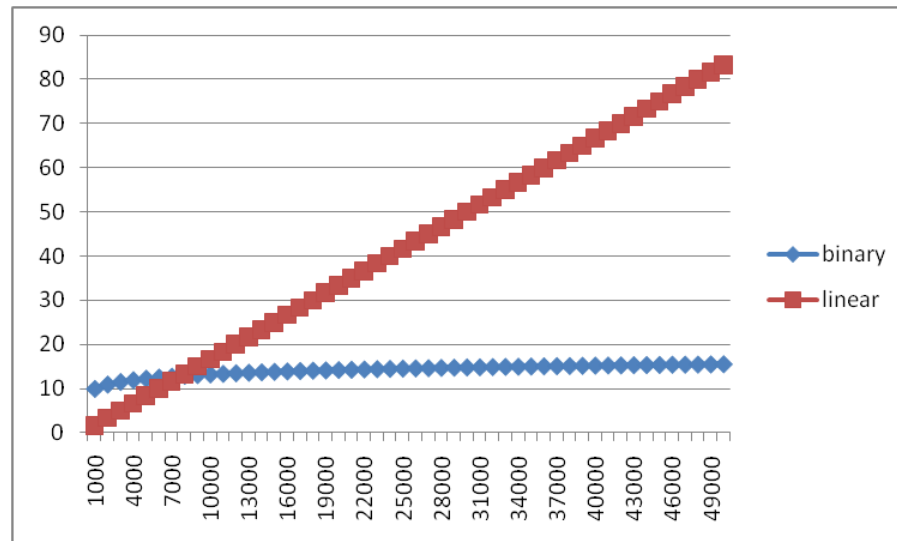
# Solving Recurrence Relations

1. Determine the recurrence relation. What is the base case?
  - $T(n) = 10 + T(n/2)$   $T(1) = 8$
2. “Expand” the original relation to find an equivalent general expression *in terms of the number of expansions*.
  - $T(n) = 10 + 10 + T(n/4)$   
 $= 10 + 10 + 10 + T(n/8)$   
 $= \dots$   
 $= 10k + T(n/(2^k))$  where  $k$  is the # of expansions
3. Find a closed-form expression by setting *the number of expansions* to a value which reduces the problem to a base case
  - $n/(2^k) = 1$  means  $n = 2^k$  means  $k = \mathbf{\log_2 n}$
  - So  $T(n) = 10 \mathbf{\log_2 n} + 8$  (get to base case and do it)
  - So  $T(n)$  is  $O(\mathbf{\log n})$

# Linear vs Binary Search

- So binary search is  $O(\log n)$  and linear is  $O(n)$ 
  - Given the constants, linear search could still be faster for small values of  $n$

Example w/ hypothetical constants:



# What about a binary version of sum?

```
int sum(int[] arr) {
    return help(arr, 0, arr.length);
}
int help(int[] arr, int lo, int hi) {
    if(lo==hi) return 0;
    if(lo==hi-1) return arr[lo];
    int mid = (hi+lo)/2;
    return help(arr, lo, mid) + help(arr, mid, hi);
}
```

Recurrence is  $T(n) = O(1) + 2T(n/2) = O(n)$

(Proof left as an exercise)

“Obvious”: have to read the whole array

You can't do better than  $O(n)$

Or can you...

We'll see a parallel version of this much later

With  $\infty$  processors,  $T(n) = O(1) + 1T(n/2) = O(\log n)$



# Another example

- $T(n) = n + 2T(n/2)$ ,  $T(1) = c$ 
  - Any guesses as to what algorithm(s) this represents?
    - Mergesort & quicksort (assuming good pivot selection)
  - Any guesses as to what the closed form for this is?
    - $O(n \log n)$

# Really common recurrences

Should know how to solve recurrences but also recognize some really common ones:

$T(n) = O(1) + T(n-1)$	linear
$T(n) = O(1) + 2T(n/2)$	linear
$T(n) = O(1) + T(n/2)$	logarithmic
$T(n) = O(1) + 2T(n-1)$	exponential
$T(n) = O(n) + T(n-1)$	quadratic
$T(n) = O(n) + T(n/2)$	linear
$T(n) = O(n) + 2T(n/2)$	$O(n \log n)$

Note big-Oh can also use more than one variable (graphs: vertices & edges)

- Example: you can (and will in proj3!) sum all elements of an  $n$ -by- $m$  matrix in  $O(nm)$



# CSE332: Data Abstractions

## Lecture 4: Priority Queues; Heaps

James Fogarty

Winter 2012

# *New ADT: Priority Queue*

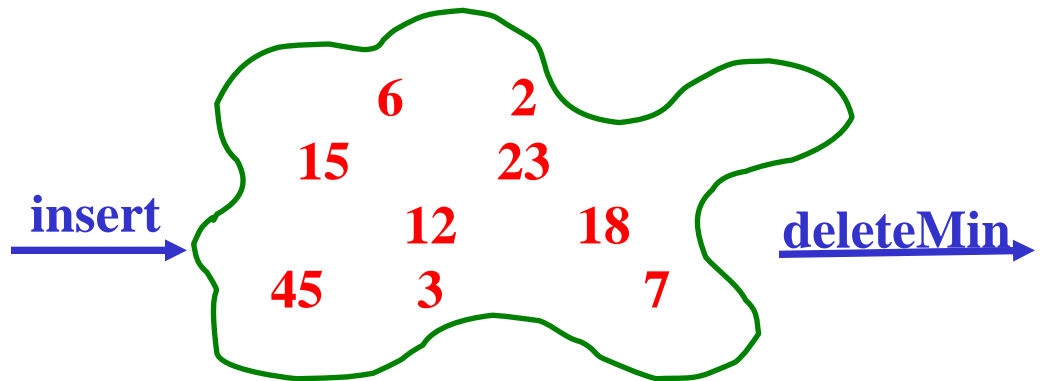
- A priority queue holds compare-able data
- Unlike LIFO stacks and FIFO queues, needs to compare items
  - Given  $x$  and  $y$ : is  $x$  less than, equal to, or greater than  $y$
  - Meaning of the ordering can depend on your data
  - Many data structures will require this: dictionaries, sorting
- Integers are comparable, so will use them in examples
- The priority queue ADT is much more general
  - Typically two fields, the *priority* and the *data*

# New ADT: Priority Queue

- Each item has a “priority”
  - The *next* or *best* item is the one with the *lowest* priority
  - So “priority 1” should come before “priority 4”
  - Simply by convention, could also do maximum priority

- Operations:

- `insert`
- `deleteMin`



- `deleteMin` *returns* and *deletes* item with lowest priority
  - Can resolve ties arbitrarily

# Priority Queue

insert *a* with priority 5

insert *b* with priority 3

insert *c* with priority 4

*w* = deleteMin

*x* = deleteMin

insert *d* with priority 2

insert *e* with priority 6

*y* = deleteMin

*z* = deleteMin

after execution:

*w* = *b*

*x* = *c*

*y* = *d*

*z* = *a*

# *Applications*

- Priority queue is a major and common ADT
  - Sometimes blatant, sometimes less obvious
- Forward network packets in order of urgency
- Execute work tasks in order of priority
  - “critical” before “interactive” before “compute-intensive” tasks
  - allocating idle tasks in cloud hosting environments
- Sort (first *insert* all items, then *deleteMin* all items)
  - Similar to Project 1’s use of a stack to implement reverse

# Advanced Applications

- “Greedy” algorithms
  - Efficiently track what is “best” to try next
- Discrete event simulation (e.g., virtual worlds, system simulation)
  - Every event  $e$  happens at some time  $t$  and generates new events  $e_1, \dots, e_n$  at times  $t+t_1, \dots, t+t_n$
  - Naïve approach:
    - Advance “clock” by 1 unit, exhaustively checking for events
  - Better:
    - Pending events in a priority queue (priority = event time)
    - Repeatedly: **deleteMin** and then **insert** new events
    - Effectively “set clock ahead to next event”



# *Finding a Good Data Structure*

- We will examine an efficient, non-obvious data structure
  - But let's first analyze some “obvious” ideas for  $n$  data items
  - All times worst-case; assume arrays “have room”

<i>data</i>	<i>insert algorithm / time</i>		<i>deleteMin algorithm / time</i>	
unsorted array	add at end	$O(1)$	search	$O(n)$
unsorted linked list	add at front	$O(1)$	search	$O(n)$
sorted circular array	search / shift	$O(n)$	move front	$O(1)$
sorted linked list	put in right place	$O(n)$	remove at front	$O(1)$
binary search tree	put in right place	$O(n)$	leftmost	$O(n)$

# *Our Data Structure: Heap*

- We are about to see a data structure called a “heap”
  - Worst-case  $O(\log n)$  **insert** and  $O(\log n)$  **deleteMin**
  - Average-case  $O(1)$  **insert** (if items arrive in random order)
  - Very good constant factors
- Possible because we only pay for the functionality we need
  - Need something better than scanning unsorted items
  - But do not need to maintain a full sort
- The heap is a tree, so we need to review some terminology

# Tree Terminology

*root(T):*

*leaves(T):*

*children(B):*

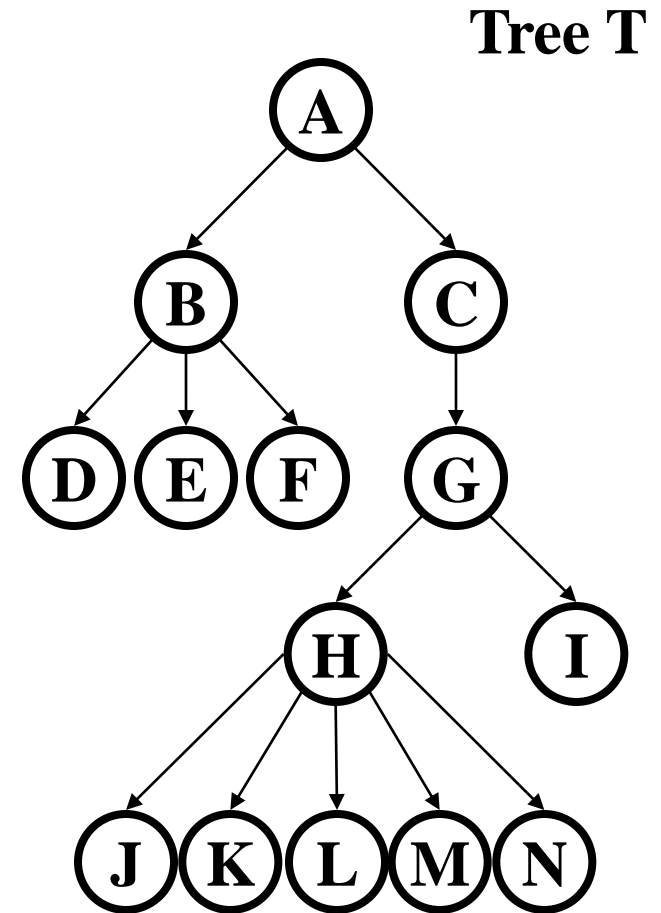
*parent(H):*

*siblings(E):*

*ancestors(F):*

*descendants(G):*

*subtree(C):*



# Tree Terminology

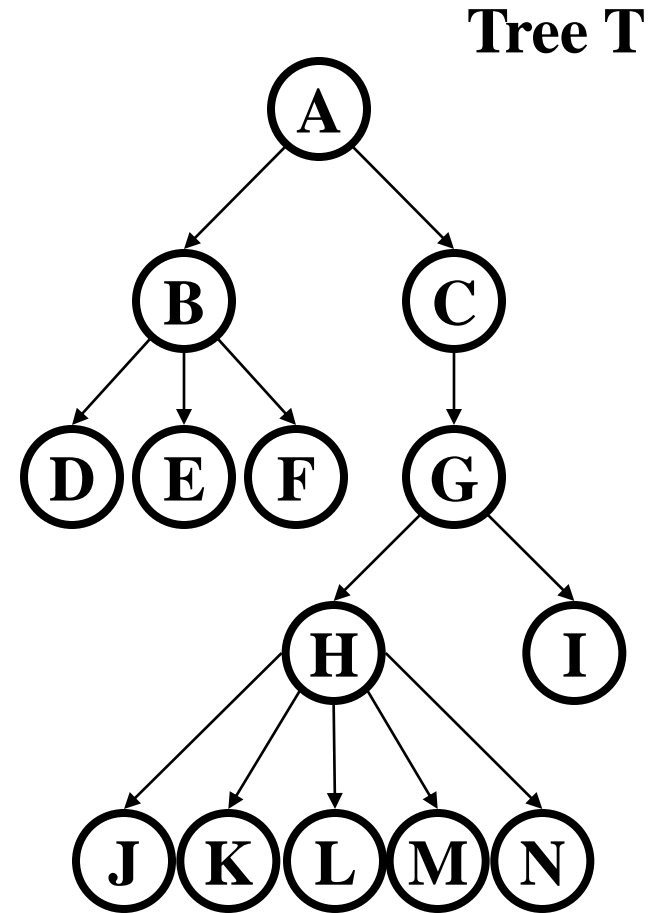
*depth(B):*

*height(G):*

*height(T):*

*degree(B):*

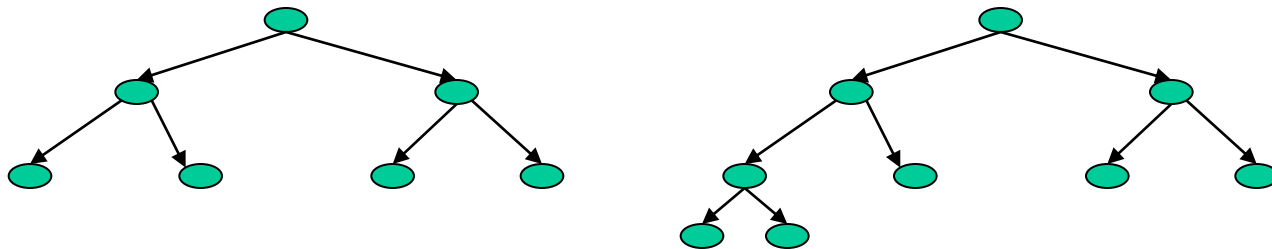
*branching factor(T):*



# Types of Trees

Certain terms define trees with specific structures

- **Binary** tree: Every node has at most 2 children
- **$n$ -ary** tree: Every node has at most  $n$  children
- **Perfect** tree: Every row is completely full
- **Complete** tree: All rows except the bottom are completely full, and it is filled from left to right



What is the height of a **perfect** tree with  $n$  nodes? A **complete** tree?

# *Properties of a Binary Min-Heap*

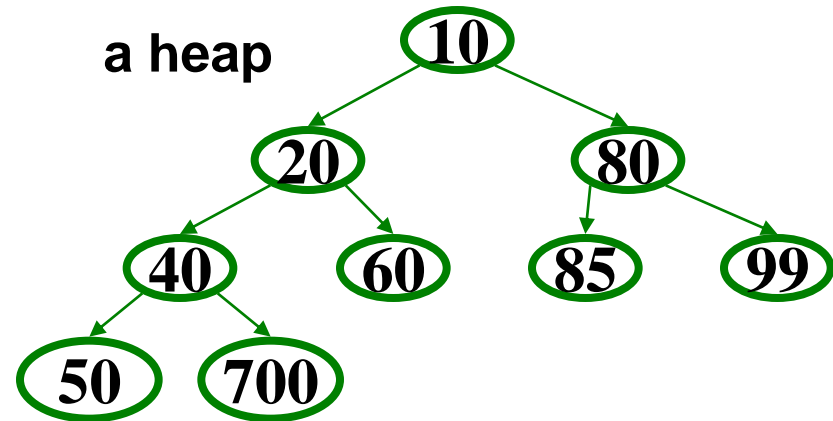
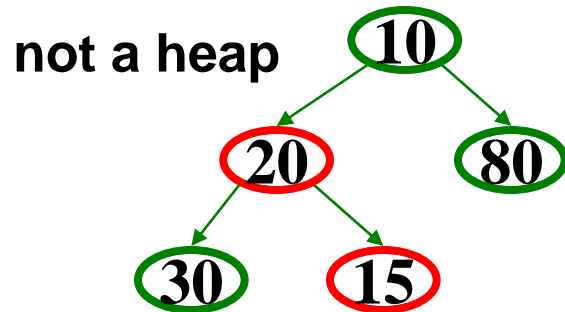
More commonly known as a **binary heap** or simply a **heap**

- **Structure Property:** A complete tree
- **Heap Property:** The priority of every non-root node is greater than the priority of its parent

How is this different from a binary search tree?

# Properties of a Binary Min-Heap

Requires both **structure property** and the **heap property**



Where is the minimum priority item?

What is the height of a heap with  $n$  items?

# Basics of Heap Operations

## findMin:

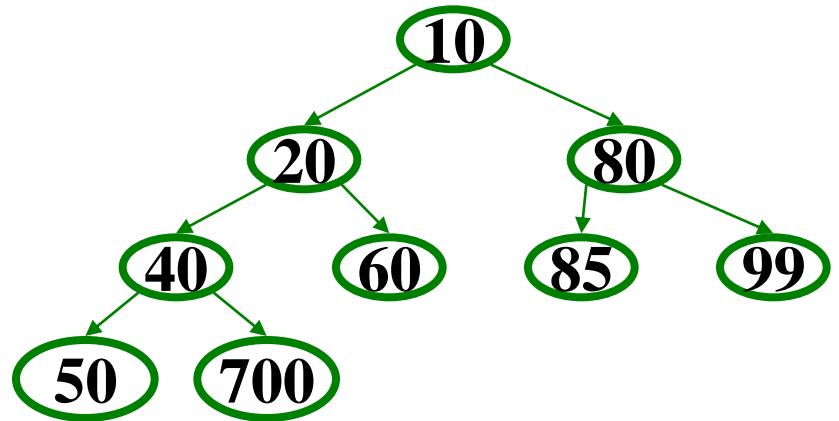
- return `root.data`

## deleteMin:

- Move last node up to root
- Violates heap property, “Percolate Down” to restore

## insert:

- Add node after last position
- Violate heap property, “Percolate Up” to restore



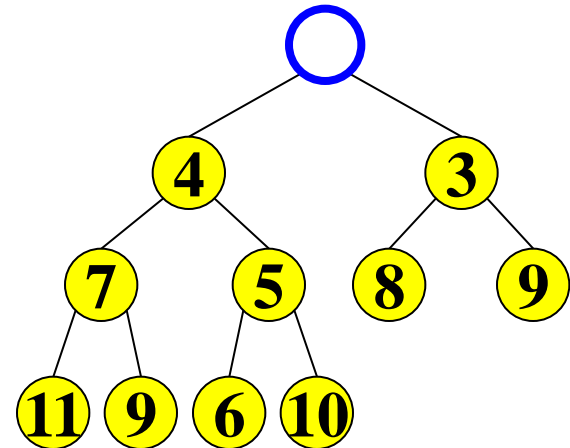
Overall, the strategy is:

- Preserve structure property
- Break and restore heap property



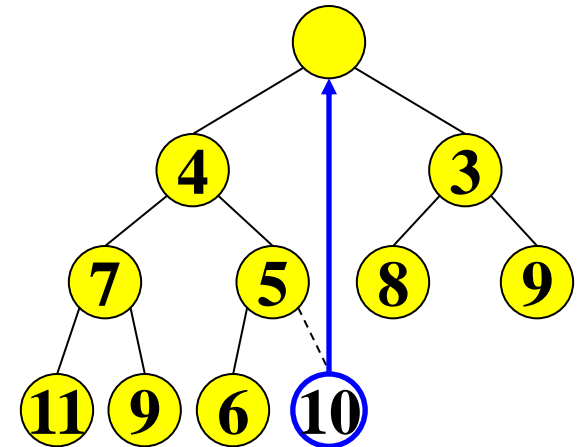
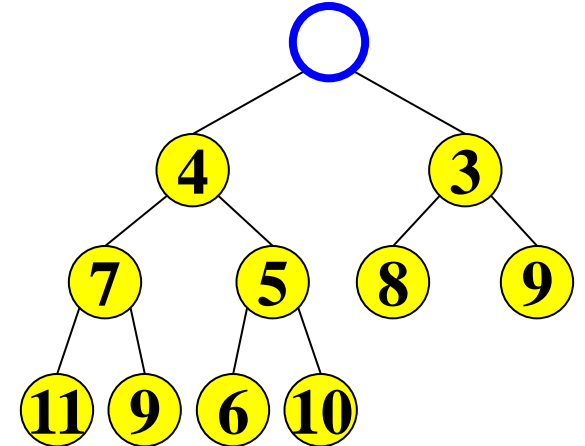
# DeleteMin Implementation

1. Delete value at root node  
(and store it for later return)



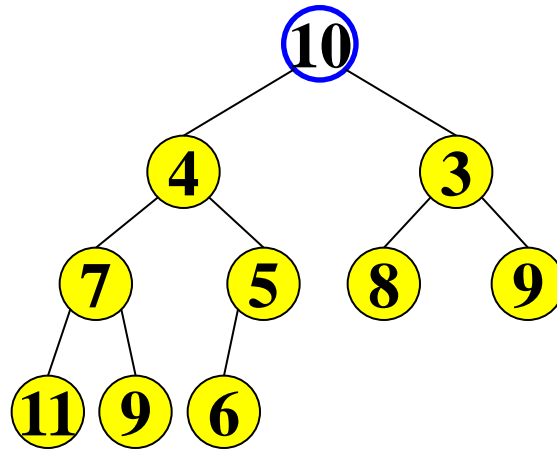
# Restoring the Structure Property

2. We now have a “hole” at the root
3. We must “fill” the hole with another value, must have a tree with one less node, and it must still be a complete tree
4. The “last” node is the obvious choice



# Restoring the Heap Property

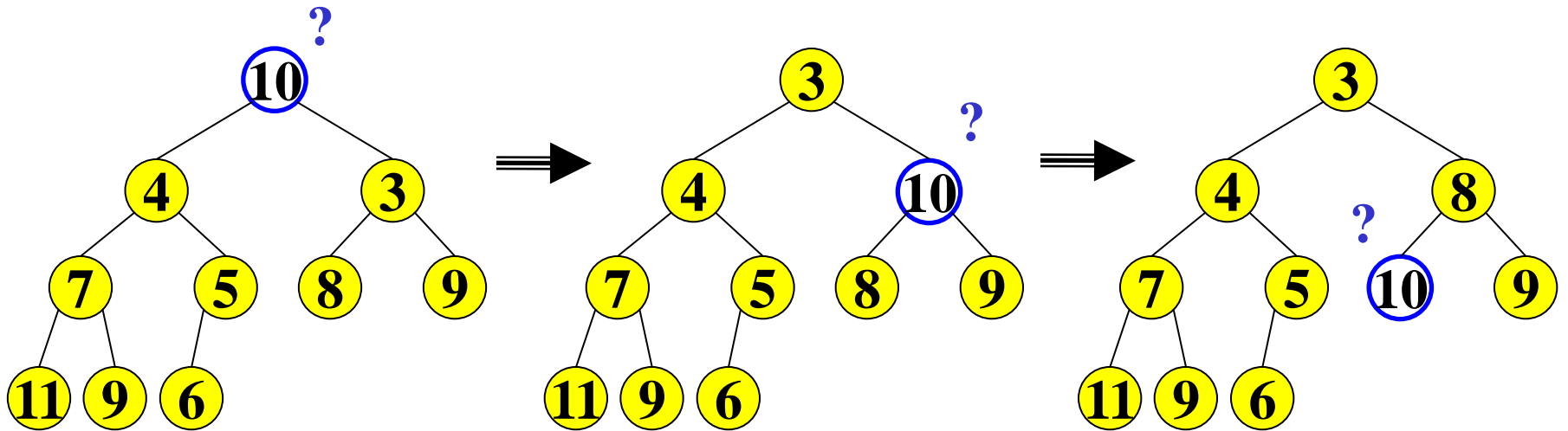
5. Not a heap, it violates the heap property



6. We **percolate down** to fix the heap

**While greater than either child**  
**Swap with smaller child**

# Percolate Down



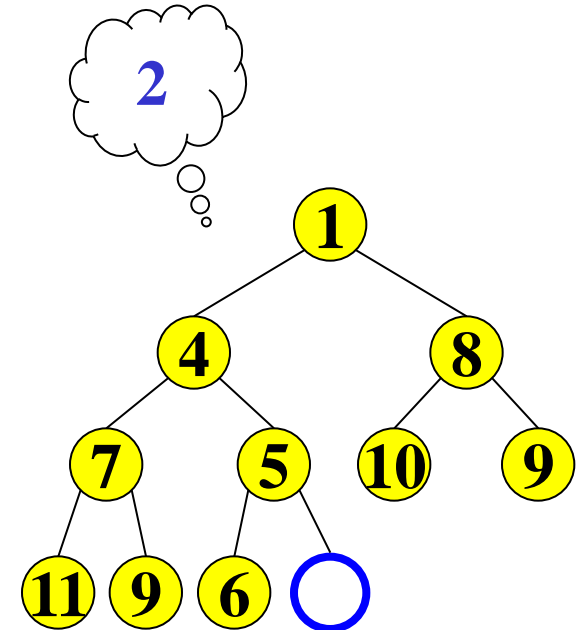
While greater than either child  
Swap with smaller child

What is the runtime?  
 $O(\log n)$

Why does this work?  
Both children are heaps

# Maintaining the Structure Property

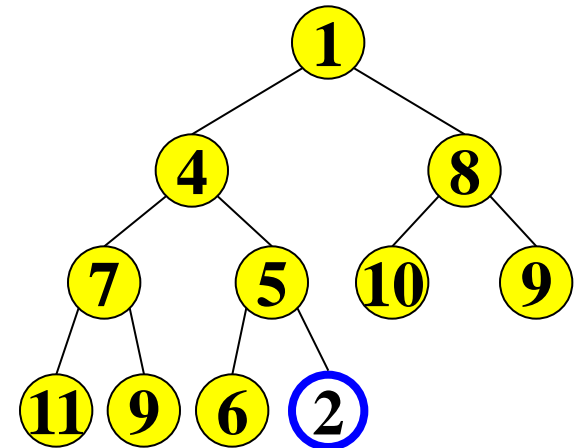
1. There is only one valid shape for our tree after addition of one more node
2. Put our new data there



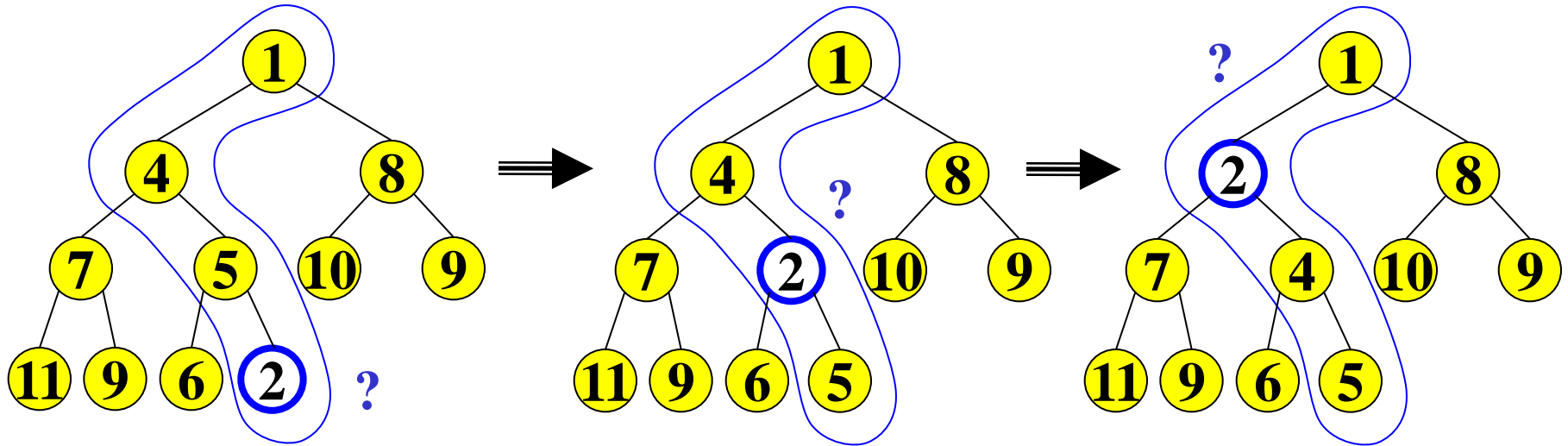
# Restoring the Heap Property

3. Then **percolate up** to fix heap property

**While less than parent**  
**Swap with parent**



# Percolate Up



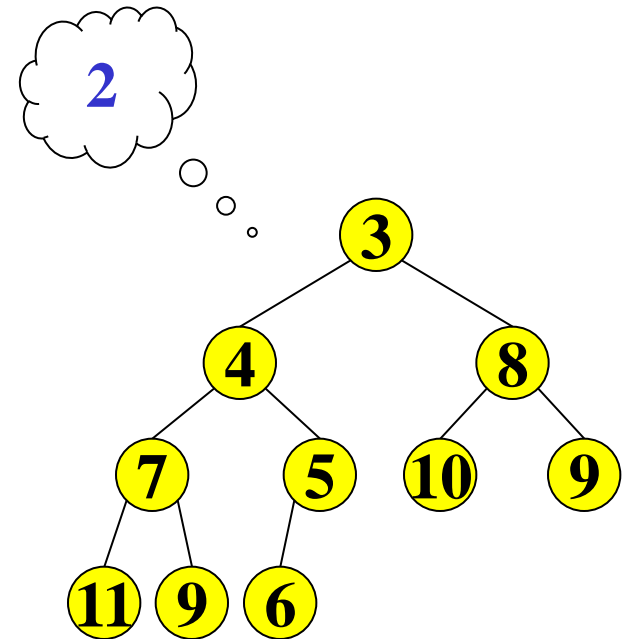
While less than parent  
Swap with parent

What is the runtime?  
 $O(\log n)$

Why does this work?  
Both children are heaps

# *Insert Implementation*

- Add a value to the tree
- Afterwards, structure and heap properties must still be correct

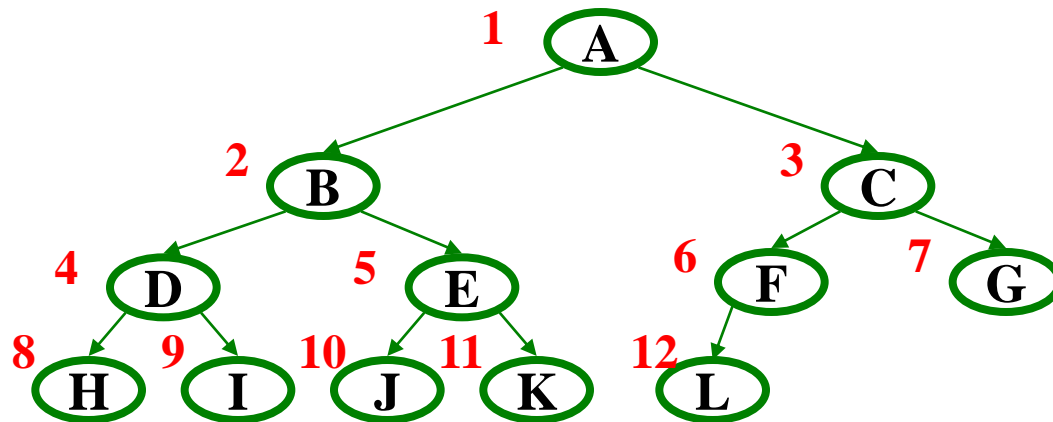




# *A Clever and Important Trick*

- We have seen worst-case  $O(\log n)$  insert and deleteMin
  - But we promised average-case  $O(1)$  insert
- Insert requires access to the “next to use” position in the tree
  - Walking the tree requires  $O(\log n)$  steps
- Remember to only pay for the functionality we need
  - We have said the tree is complete, but have not said why
- All complete trees of size  $n$  contain the same edges
  - So why are we even representing the edges?

# Array Representation of a Binary Heap



From node  $i$ :

left child:  $i*2$

right child:  $i*2+1$

parent:  $i/2$

wasting index 0 is  
convenient for the math

Array implementation:

	A	B	C	D	E	F	G	H	I	J	K	L	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

# *Tradeoffs of the Array Implementation*

## Advantages:

- Non-data space: only index 0 and any unused space on right
  - Contrast to link representation using one edge per node (except root), a total of  $n-1$  wasted space (like linked lists)
  - Array would waste more space if tree were not complete
- Multiplying and dividing by 2 is extremely fast
- The major one: Last used position is at index **size**,  $O(1)$  access

## Disadvantages:

- Same might-be-empty or might-get-full problems we saw with stacks and queues (resize by doubling as necessary)

Advantages outweigh disadvantages: “this is how people do it”



# CSE332: Data Abstractions

## Lecture 5: Heaps

James Fogarty

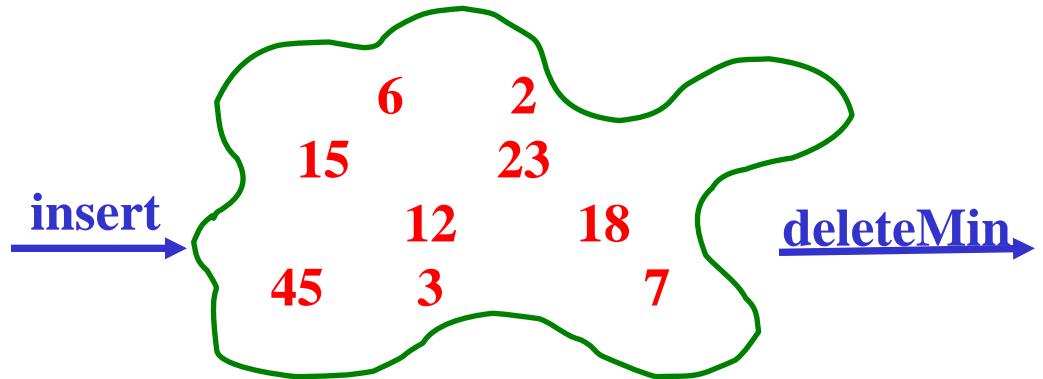
Winter 2012

# ADT: Priority Queue

- Each item has a “priority”
  - The *next* or *best* item is the one with the *lowest* priority
  - So “priority 1” should come before “priority 4”
  - Simply by convention, could also do maximum priority

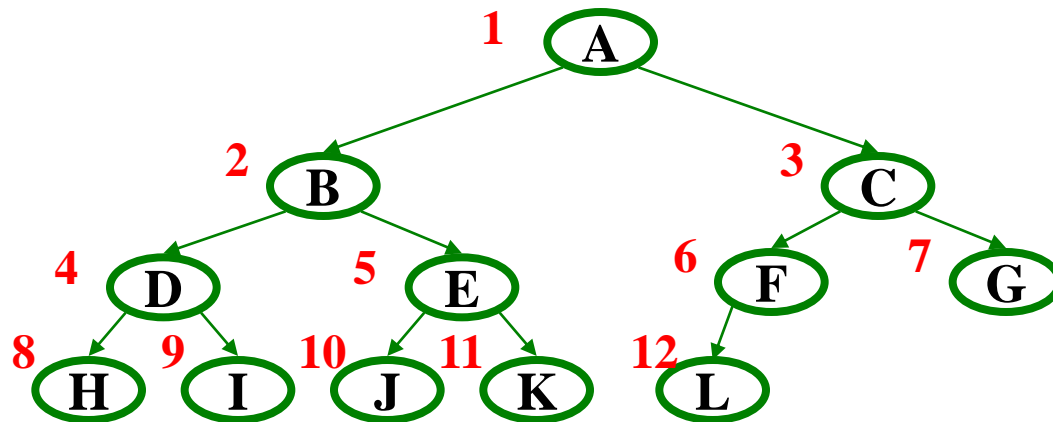
- Operations:

- `insert`
- `deleteMin`



- `deleteMin` *returns* and *deletes* item with lowest priority
  - Can resolve ties arbitrarily

# Array Representation of a Binary Heap



From node  $i$ :

left child:  $i*2$

right child:  $i*2+1$

parent:  $i/2$

wasting index 0 is  
convenient for the math

Array implementation:

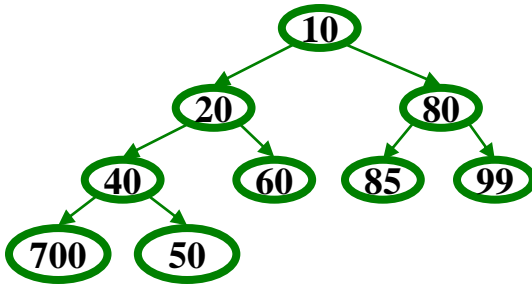
	A	B	C	D	E	F	G	H	I	J	K	L	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

# Pseudocode: insert

```
void insert(int val) {  
    if(size==arr.length-1)  
        resize();  
    size++;  
    i=percolateUp(size,val);  
    arr[i] = val;  
}
```

This pseudocode uses ints. In real use, you will have data nodes with priorities.

```
int percolateUp(int hole,  
                int val) {  
    while(hole > 1 &&  
          val < arr[hole/2])  
        arr[hole] = arr[hole/2];  
        hole = hole / 2;  
    }  
    return hole;  
}
```



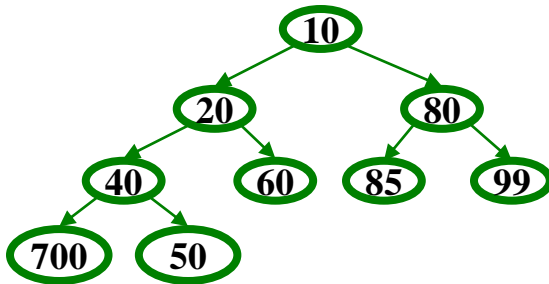
	10	20	80	40	60	85	99	700	50				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

# Pseudocode: deleteMin

This pseudocode uses ints. In real use, you will have data nodes with priorities.

```
int deleteMin() {  
    if(isEmpty()) throw...  
    ans = arr[1];  
    hole = percolateDown  
        (1, arr[size]);  
    arr[hole] = arr[size];  
    size--;  
    return ans;  
}
```

```
int percolateDown(int hole,  
                 int val) {  
    while(2*hole <= size) {  
        left = 2*hole;  
        right = left + 1;  
        if(arr[left] < arr[right]  
           || right > size)  
            target = left;  
        else  
            target = right;  
        if(arr[target] < val) {  
            arr[hole] = arr[target];  
            hole = target;  
        } else  
            break;  
    }  
    return hole;  
}
```

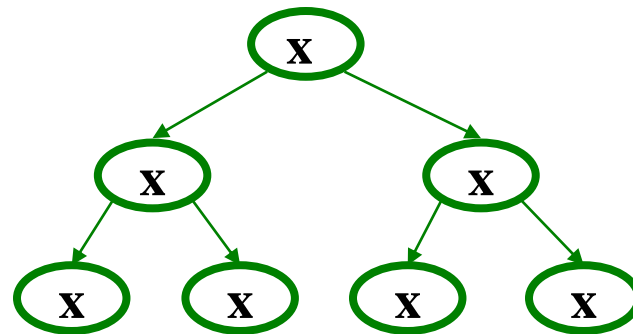
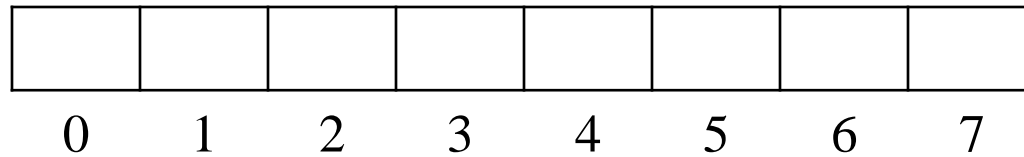


	10	20	80	40	60	85	99	700	50				
0	1	2	3	4	5	6	7	8	9	10	11	12	13



# Example

1. insert: 105, 69, 43, 32, 16, 4, 2
2. deleteMin



# *Other Operations*

What is the runtime?  
 $O(\log n)$

- **decreaseKey:**
  - given pointer to object in priority queue (e.g., its array index), lower its priority to  $p$
  - Change priority and percolate up
- **increaseKey:**
  - given pointer to object in priority queue (e.g., its array index), raise its priority to  $p$
  - Change priority and percolate down
- **remove:**
  - given pointer to object in priority queue (e.g., its array index), remove it from the queue
  - **decreaseKey** to  $p = -\infty$ , then **deleteMin**

# *Build Heap*

- Suppose you have  $n$  items to put in a new priority queue
  - Sequence of  $n$  **inserts**,  $O(n \log n)$
- Can we do better?
  - Above is only choice if ADT does not provide **buildHeap**
- Important issue in ADT design: how many specialized operations
  - Tradeoff: Convenience, Efficiency, Simplicity
- In this case, we are motivated by efficiency
  - We can **buildHeap** using  $O(n)$  algorithm called Floyd's Method

# *Floyd's Method*

Recall our general strategy for working with the heap:

- Preserve structure property
  - Break and restore heap property
- 
1. Use our  $n$  items to make a complete tree
    - Put them in array indices  $1, \dots, n$
  2. Treat it as a heap and fix the heap-order property
    - Exactly how we do this is where we gain efficiency

# *Floyd's Method*

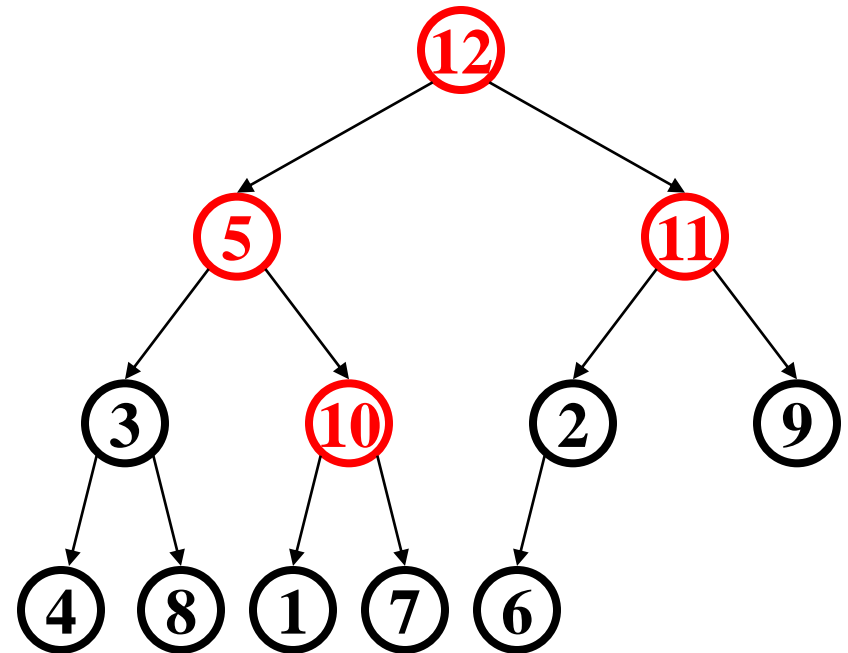
## Bottom-up

- Leaves are already in heap order
- Work up toward the root one level at a time

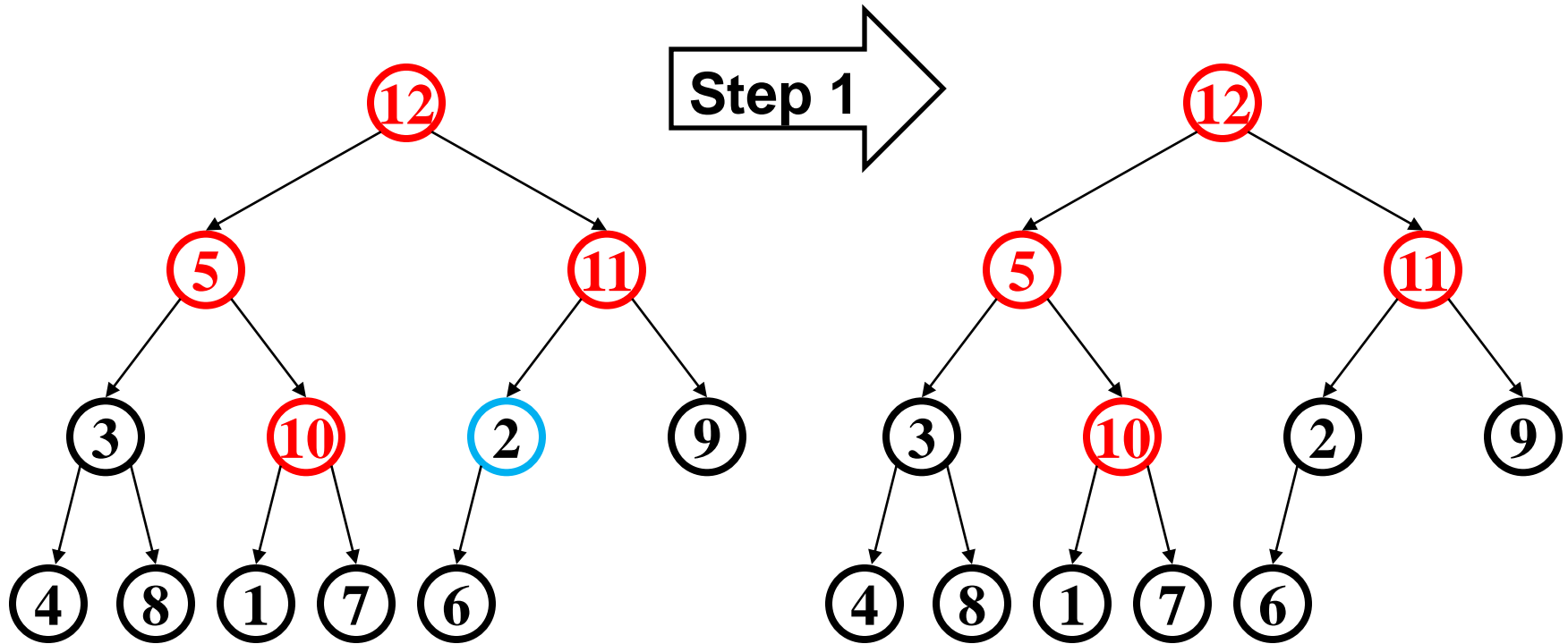
```
void buildHeap() {
    for(i = size/2; i>0; i--) {
        val = arr[i];
        hole = percolateDown(i, val);
        arr[hole] = val;
    }
}
```

# Example

- In tree form for readability
  - Red for nodes which are not less than descendants
  - Notice no leaves are red
  - Check/fix each non-leaf bottom-up (6 steps here)

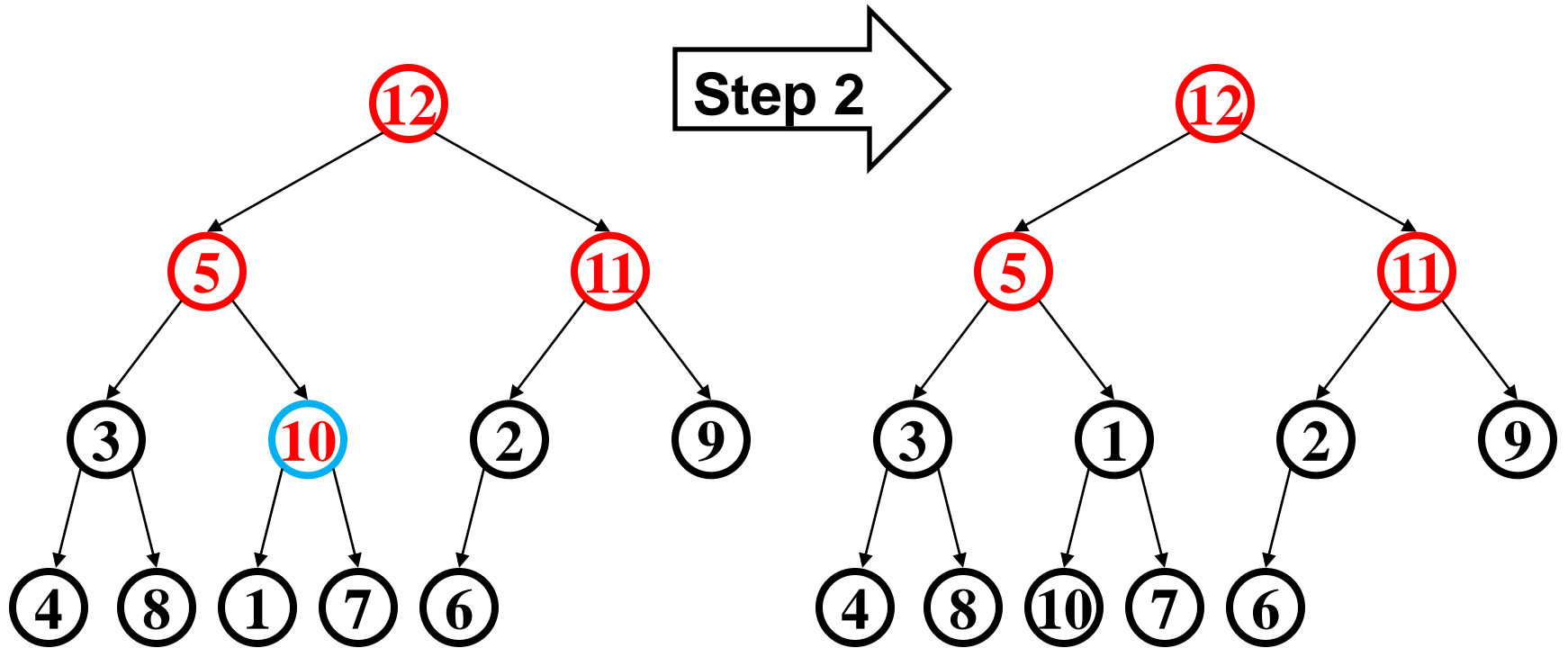


# Example



- Happens to already be less than children

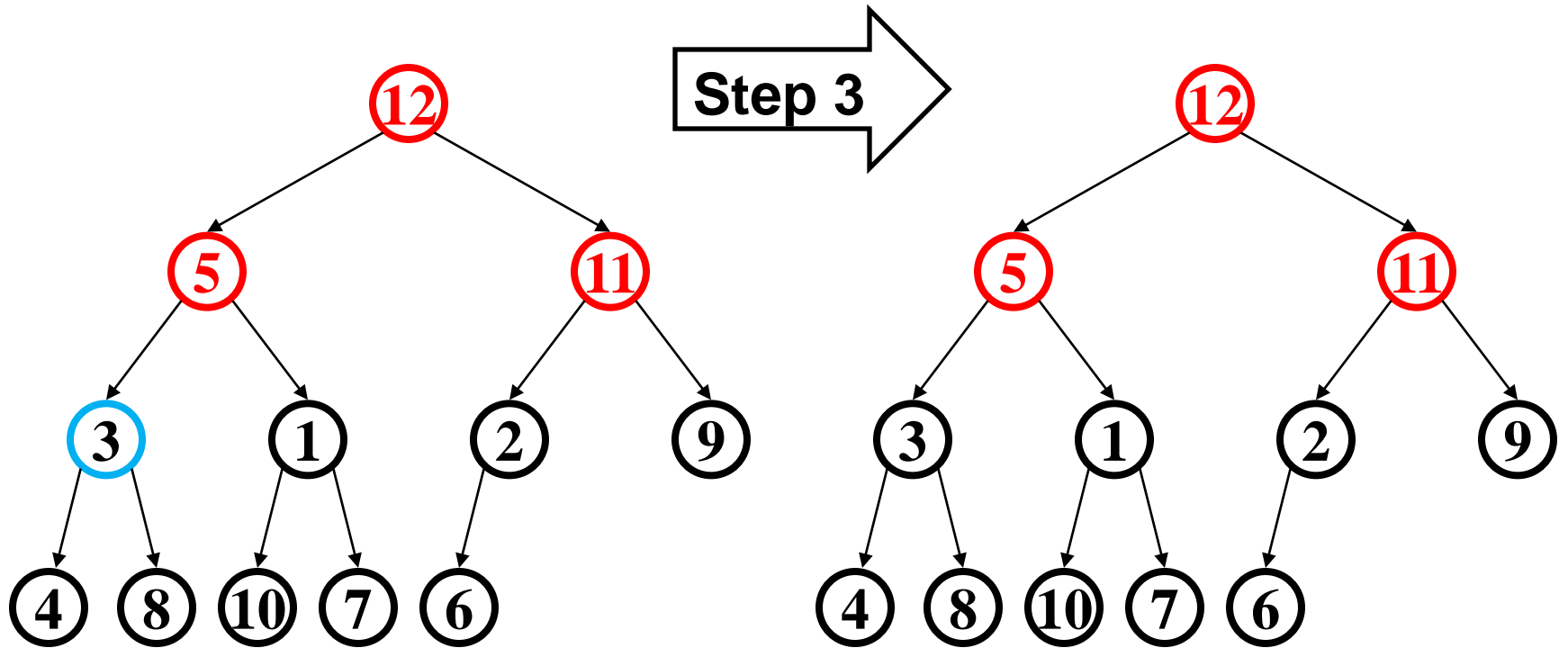
# Example



- 10 percolates down (and notice that 1 moves up)

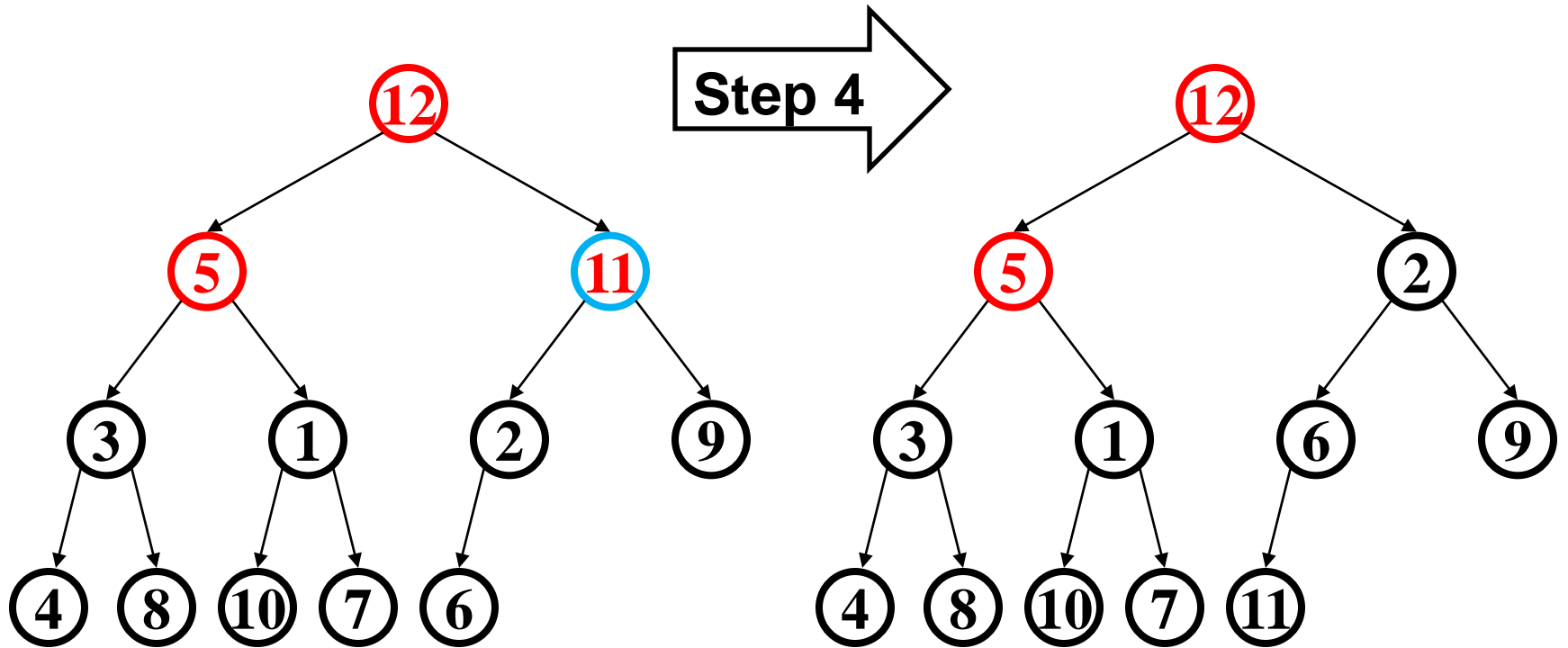


# Example



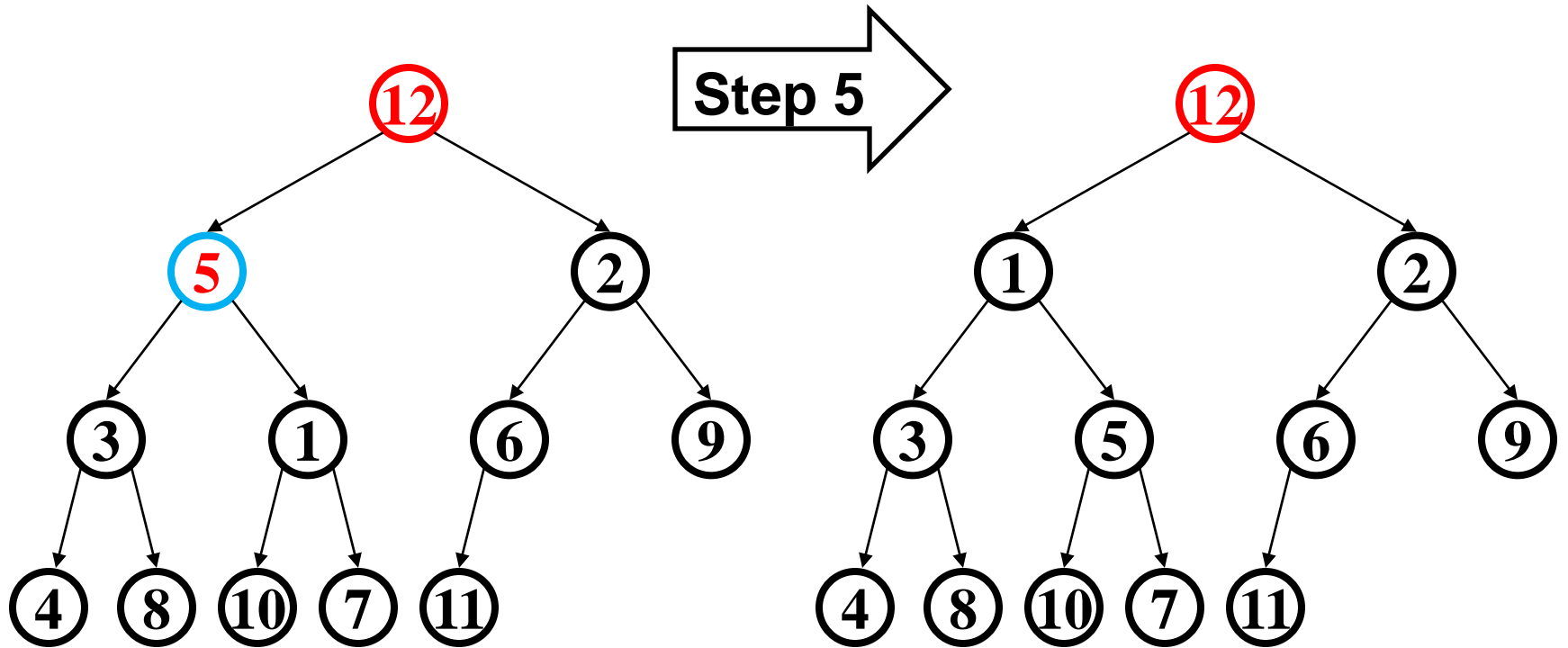
- Another nothing-to-do step

# Example



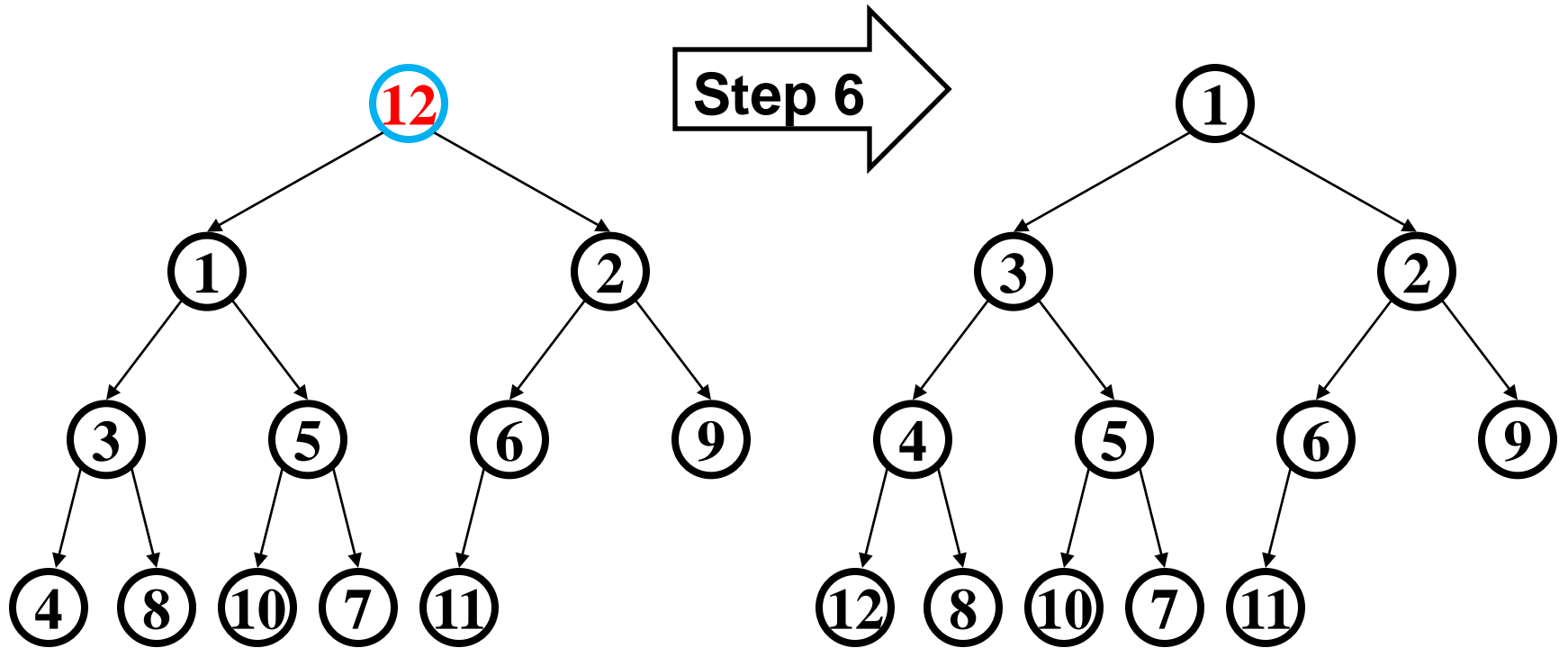
- Percolate down as necessary (first 2, then 6)

# Example



- Percolate down as necessary (the 1 again)

# Example



- Percolate down as necessary (first 1, then 3, then 4)

# *But is it right?*

- “Seems to work”
  - First we will *prove* it restores the heap property (correctness)
  - Then we will *prove* its running time (efficiency)

```
void buildHeap() {
    for(i = size/2; i>0; i--) {
        val = arr[i];
        hole = percolateDown(i, val);
        arr[hole] = val;
    }
}
```

# Correctness

```
void buildHeap() {
    for(i = size/2; i>0; i--) {
        val = arr[i];
        hole = percolateDown(i, val);
        arr[hole] = val;
    }
}
```

*Loop Invariant:* For all  $j > i$ , `arr[j]` is less than its children

- True initially: If  $j > \text{size}/2$ , then  $j$  is a leaf
  - Otherwise its left child would be at position  $> \text{size}$
- True after one more iteration: loop body and `percolateDown` make `arr[i]` less than children without breaking the property for any descendants

So after the loop finishes, all nodes are less than their children

# Efficiency

```
void buildHeap() {  
    for(i = size/2; i>0; i--) {  
        val = arr[i];  
        hole = percolateDown(i, val);  
        arr[hole] = val;  
    }  
}
```

Easy argument: `buildHeap` is  $O(n \log n)$  where  $n$  is `size`

- `size/2` loop iterations
- Each iteration does one `percolateDown`, each is  $O(\log n)$

This is correct, but there is a “tighter” analysis of the algorithm...

# Efficiency

```
void buildHeap() {  
    for(i = size/2; i>0; i--) {  
        val = arr[i];  
        hole = percolateDown(i, val);  
        arr[hole] = val;  
    }  
}
```

Better argument: `buildHeap` is  $O(n)$  where  $n$  is `size`

- `size/2` total loop iterations:  $O(n)$
- 1/2 the loop iterations percolate at most 1 step
- 1/4 the loop iterations percolate at most 2 steps
- 1/8 the loop iterations percolate at most 3 steps
- ...
- $((1/2) + (2/4) + (3/8) + (4/16) + (5/32) + \dots) < 2$  (page 4 of Weiss)
  - So at most  $2(\text{size}/2)$  total percolate steps:  $O(n)$



# *Lessons from* **buildHeap**

- Without **buildHeap**, our ADT already allows clients to implement their own in worst-case  $O(n \log n)$ 
  - Worst case is inserting lower priority values later
- By providing a specialized operation internal to the data structure (with access to the internal data), we can do  $O(n)$  worst case
  - Intuition: Most data is near a leaf, so better to percolate down
- Can analyze this algorithm for:
  - Correctness:
    - Non-trivial inductive proof using loop invariant
  - Efficiency:
    - First analysis easily proved it was  $O(n \log n)$
    - A “tighter” analysis shows same algorithm is  $O(n)$

# *What we are Skipping (see text if curious)*

- $d$ -heaps: have  $d$  children instead of 2
  - Makes heaps shallower, useful for heaps too big for memory
  - The same issue arises for balanced binary search trees and we *will* study “B-Trees”
- **merge**: given two priority queues, make one priority queue
  - How might you merge binary heaps:
    - If one heap is much smaller than the other?
    - If both are about the same size?
  - Different pointer-based data structures for priority queues support logarithmic time **merge** operation (impossible with binary heaps)



# CSE332: Data Abstractions

## Lecture 6: Dictionary, BST, AVL Tree

James Fogarty

Winter 2012

# The Dictionary (a.k.a. Map) ADT

- Data:
  - Set of (key, value) *pairs*
  - keys must be *comparable*

- Operations:

- `insert(key, value)`
- `find(key)`
- `delete(key)`
- ...

`insert(jfogarty, ....)`

`find(trobison)`

Tyler, Robison, ...



*Probably the single most common ADT in everyday programs*

*We will tend to emphasize the keys, don't forget about the stored values*

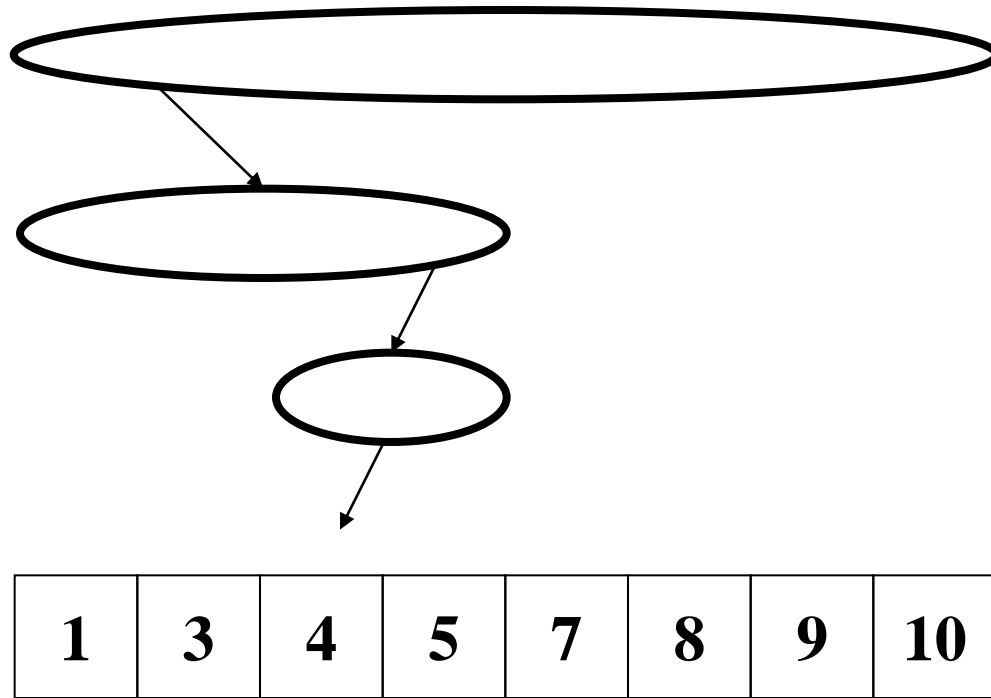
# Simple Implementations

For dictionary with  $n$  key/value pairs

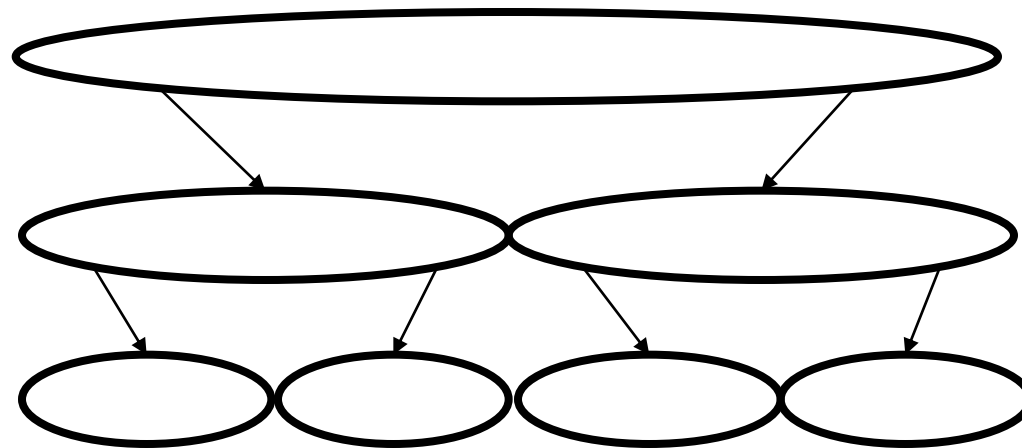
	<b>insert</b>	<b>find</b>	<b>delete</b>
• Unsorted linked-list	$O(1)$	$O(n)$	$O(n)$
• Unsorted array	$O(1)$	$O(n)$	$O(n)$
• Sorted linked list	$O(n)$	$O(n)$	$O(n)$
• Sorted array	$O(n)$	$O(\log n)$	$O(n)$
	$\log n + n$		$\log n + n$

# *Binary Search*

**Target 4**



# *Binary Search Tree*

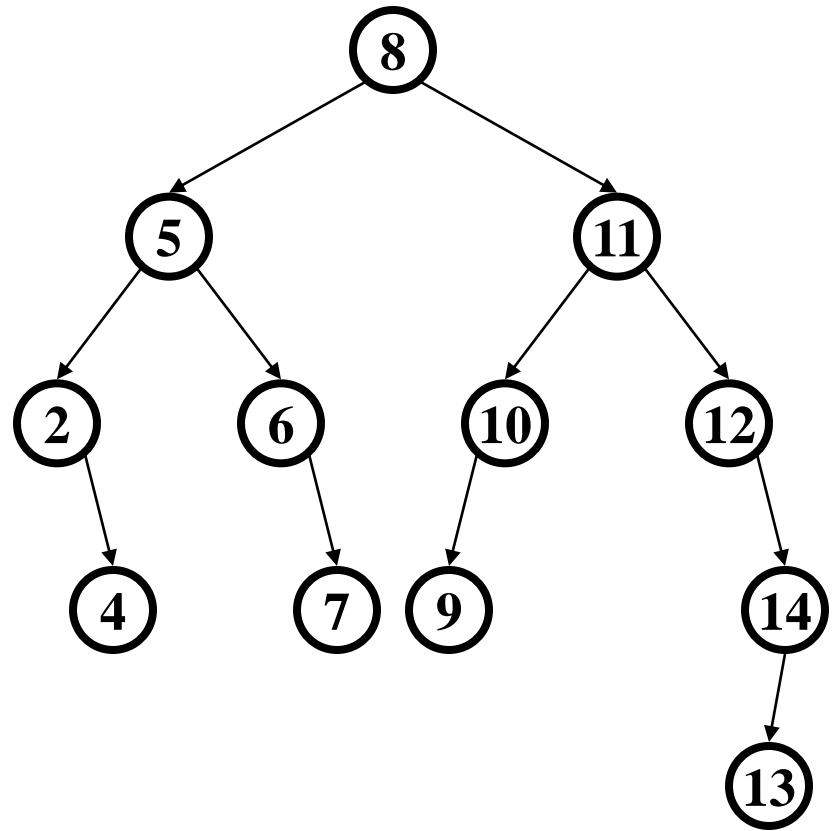


<b>1</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>
----------	----------	----------	----------	----------	----------	----------	-----------

Our goal is the performance of binary search in a tree representation

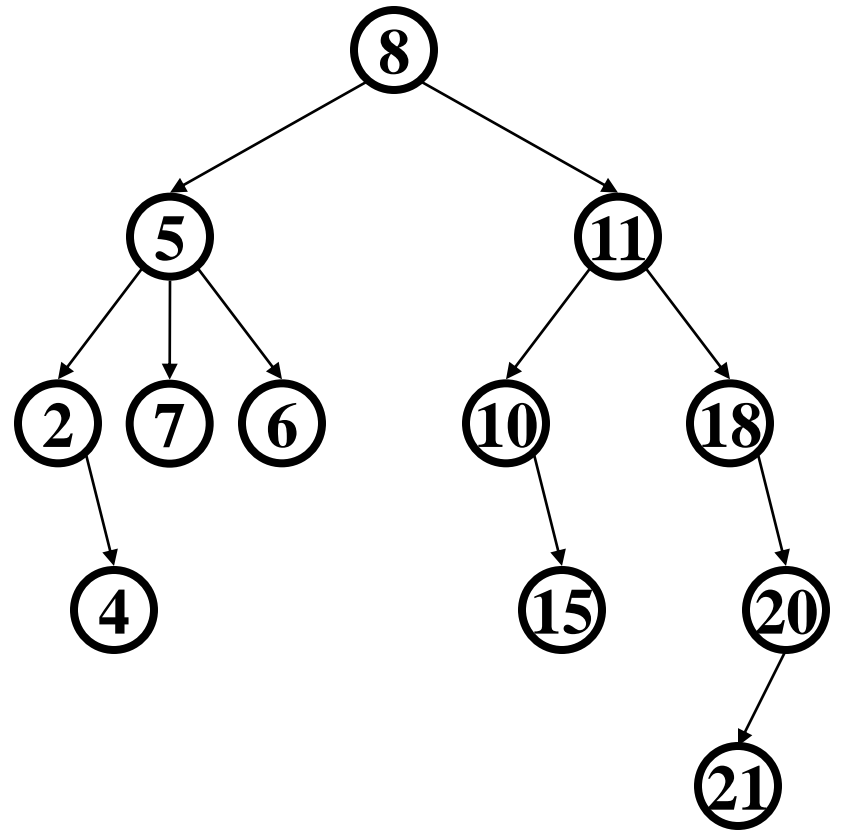
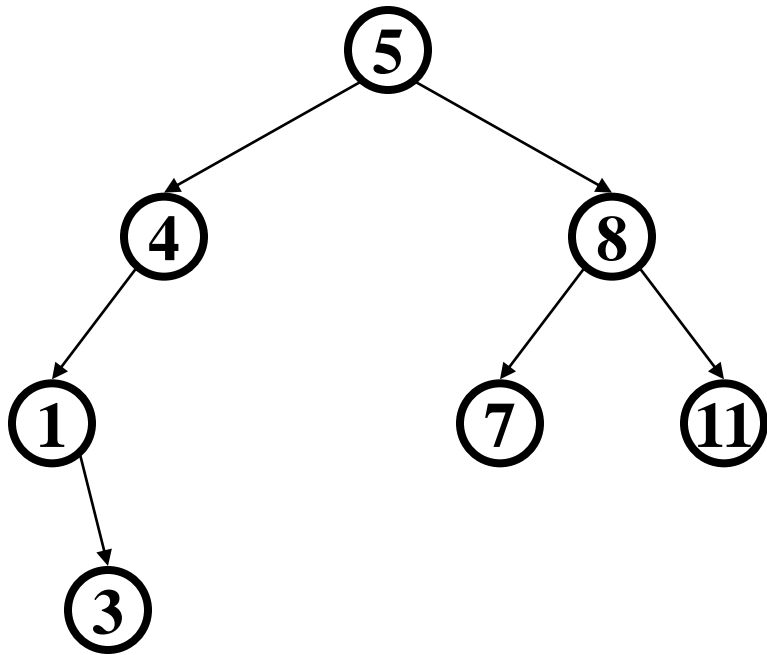
# Binary Search Tree

- Structure Property (“binary”)
  - each node has  $\leq 2$  children
- Order Property
  - all keys in left subtree are smaller than node’s key
  - all keys in right subtree are larger than node’s key

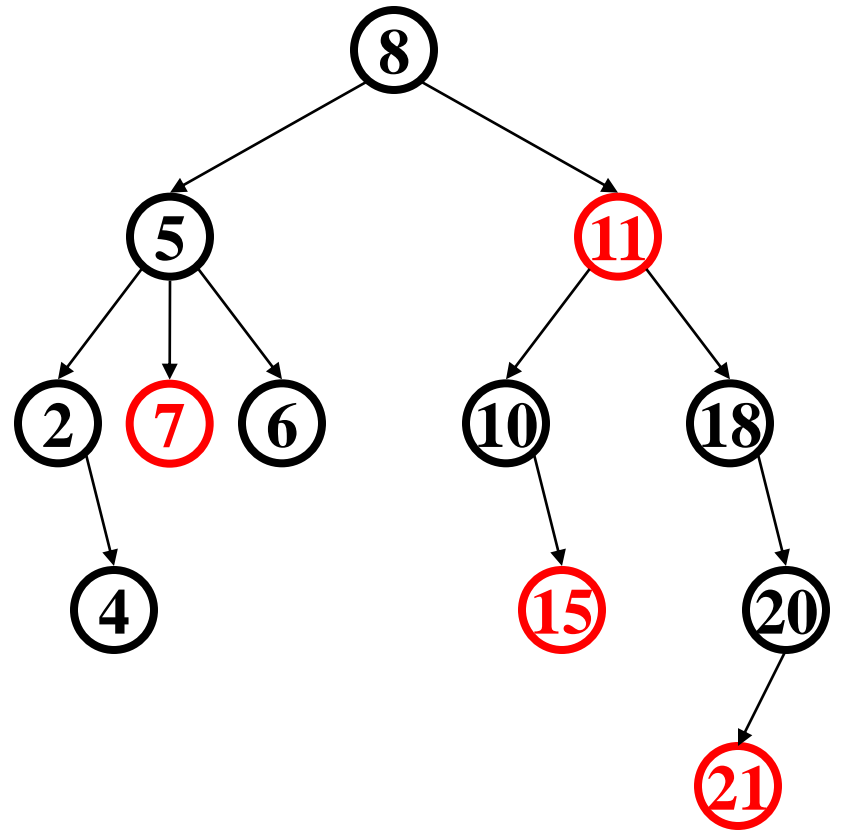
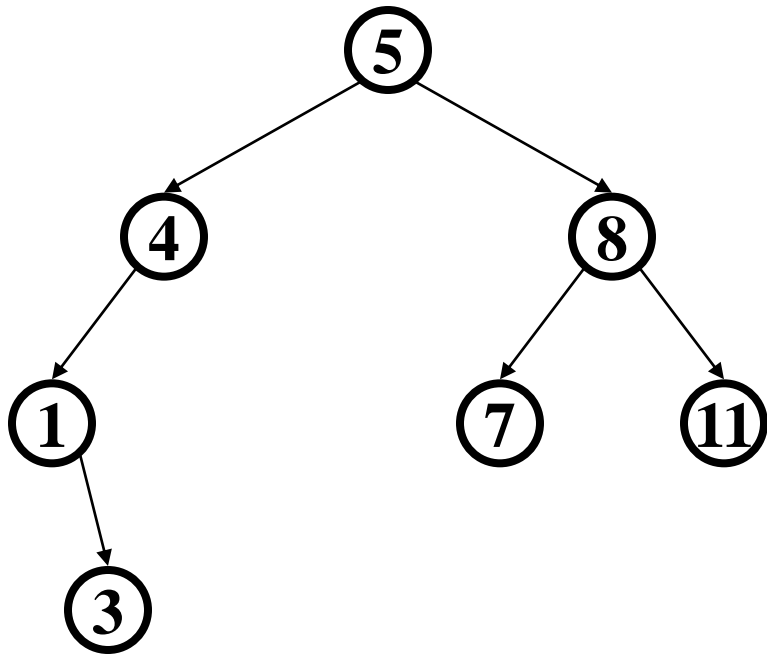




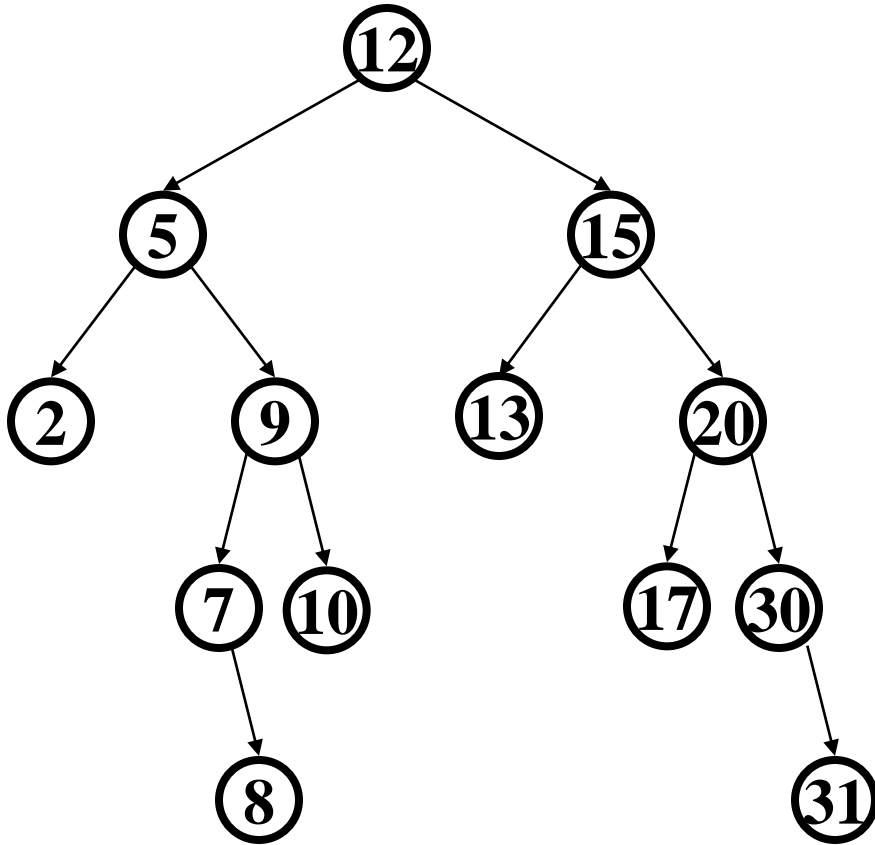
*Are these BSTs?*



*Are these BSTs?*



# Insert and Find in BST



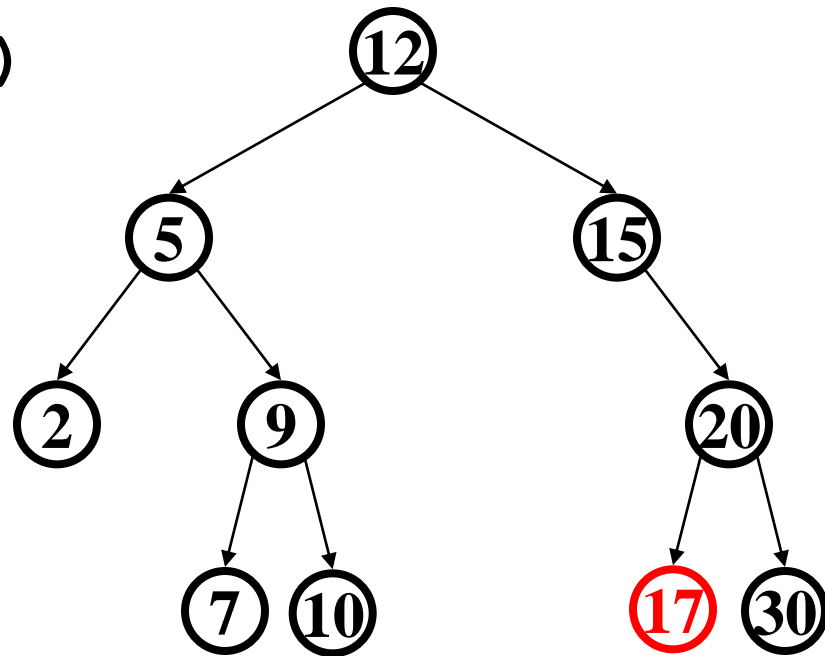
```
insert(13)  
insert(8)  
insert(31)  
find(17)  
find(11)
```

Insertion happens at leaves  
Find walks down tree



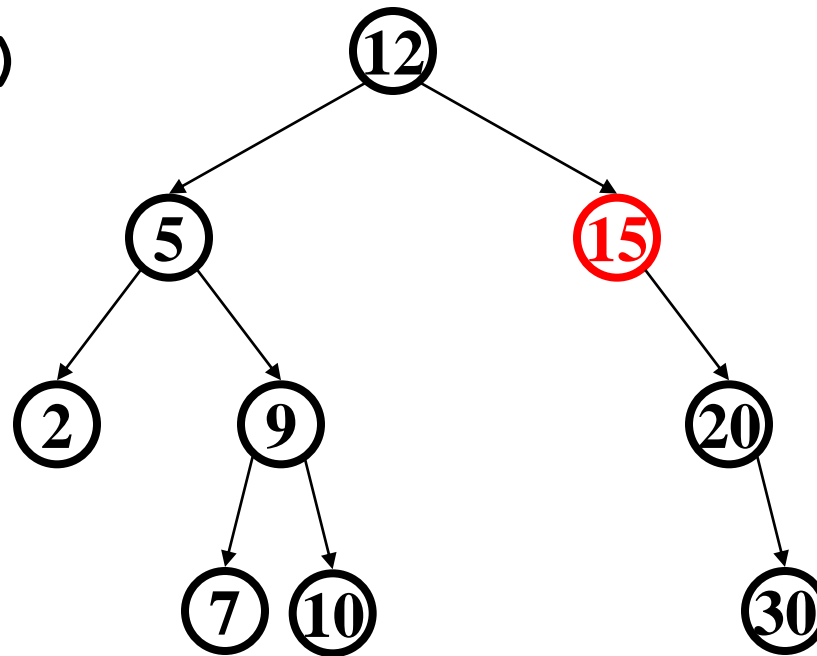
# *Deletion – The Leaf Case*

`delete(17)`



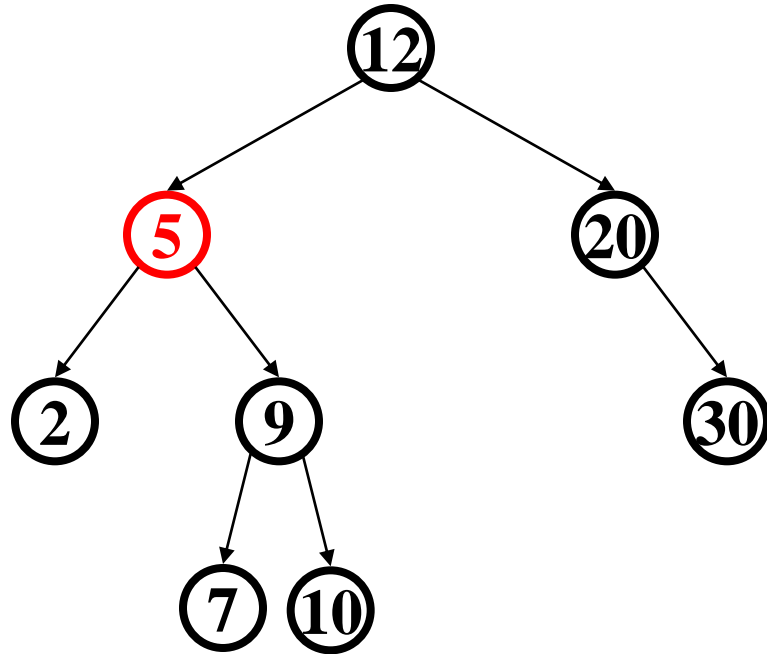
# *Deletion – The One Child Case*

`delete (15)`



# Deletion – The Two Child Case

delete (5)



What can we use to replace the 5?

- *successor* from right subtree: `findMin(node.right)`
- *predecessor* from left subtree: `findMax(node.left)`

# *The Need for a Balanced BST*

## *Observation*

- BST is overall great
  - The shallower, the better!
- But worst case height is  $O(n)$ 
  - Caused by simple cases, such as pre-sorted data

## *Solution*

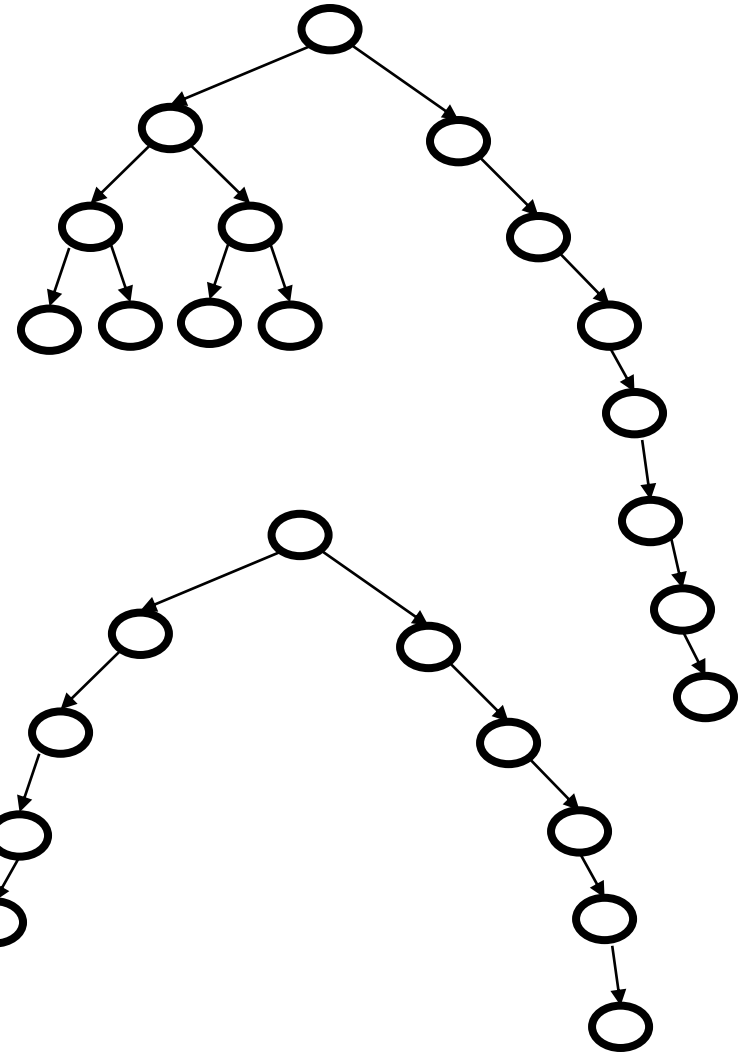
Require a **Balance Condition** that will:

1. ensure depth is always  $O(\log n)$  – strong enough!
2. be easy to maintain – not too strong!

# Potential Balance Conditions

1. Left and right subtrees of the root have equal number of nodes

*Too weak!*  
*Height mismatch example:*



2. Left and right subtrees of the root have equal height

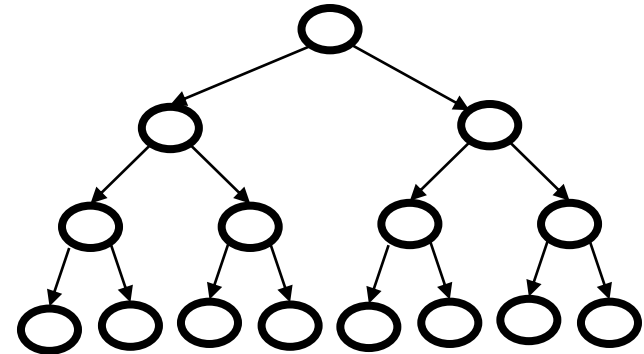
*Too weak!*  
*Double chain example:*



# Potential Balance Conditions

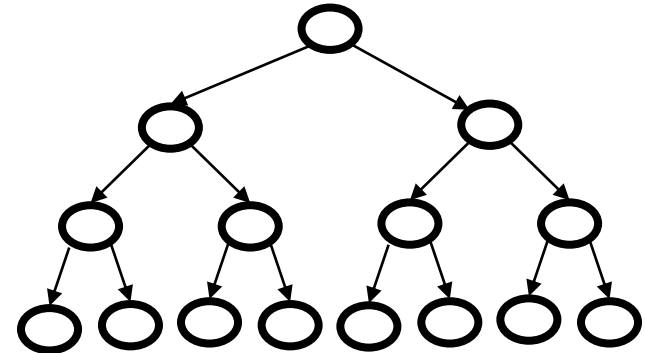
3. Left and right subtrees of every node have equal number of nodes

*Too strong!*  
*Only perfect trees ( $2^n - 1$  nodes)*



4. Left and right subtrees of every node have equal height

*Too strong!*  
*Only perfect trees ( $2^n - 1$  nodes)*



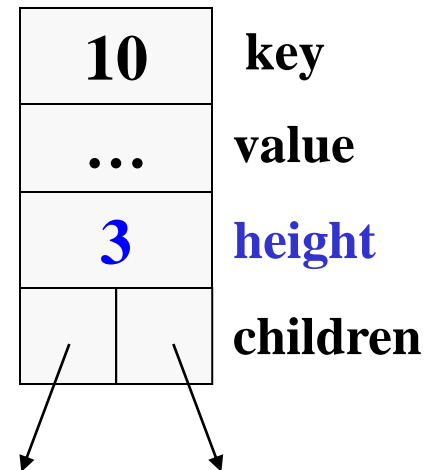
# The AVL Balance Condition

Left and right subtrees of *every node*  
have *heights differing by at most 1*

*Definition:*  $\text{balance}(\text{node}) = \text{height}(\text{node.left}) - \text{height}(\text{node.right})$

*AVL property:* for every node  $x$ ,  $-1 \leq \text{balance}(x) \leq 1$

- Ensures small depth
  - Can prove by showing an AVL tree of height  $h$  must have nodes *exponential* in  $h$
- Efficient to maintain
  - Using single and double rotations



# Calculating Height

What is the height of a tree with root  $r$ ?

```
int treeHeight(Node root) {  
    if (root == null)  
        return -1;  
    return 1 + max(treeHeight(root.left),  
                  treeHeight(root.right));  
}
```

Running time for tree with  $n$  nodes:

$O(n)$  – single pass over tree

Very important detail of definition:

height of a null tree is  $-1$ , height of tree with a single node is  $0$

# An AVL Tree?

This is the minimum  
AVL tree of height 4

Let  $S(h)$  be the  
minimum nodes in height  $h$

$$S(h) = S(h-1) + S(h-2) + 1$$

$$S(-1) = 0$$

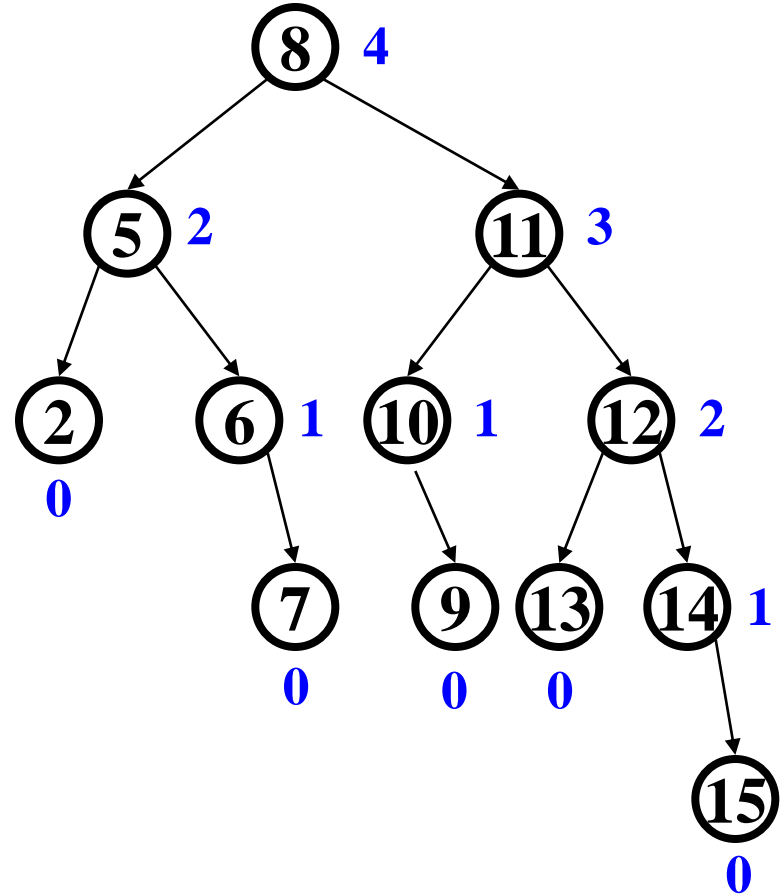
$$S(0) = 1$$

$$S(1) = 2$$

$$S(2) = 4$$

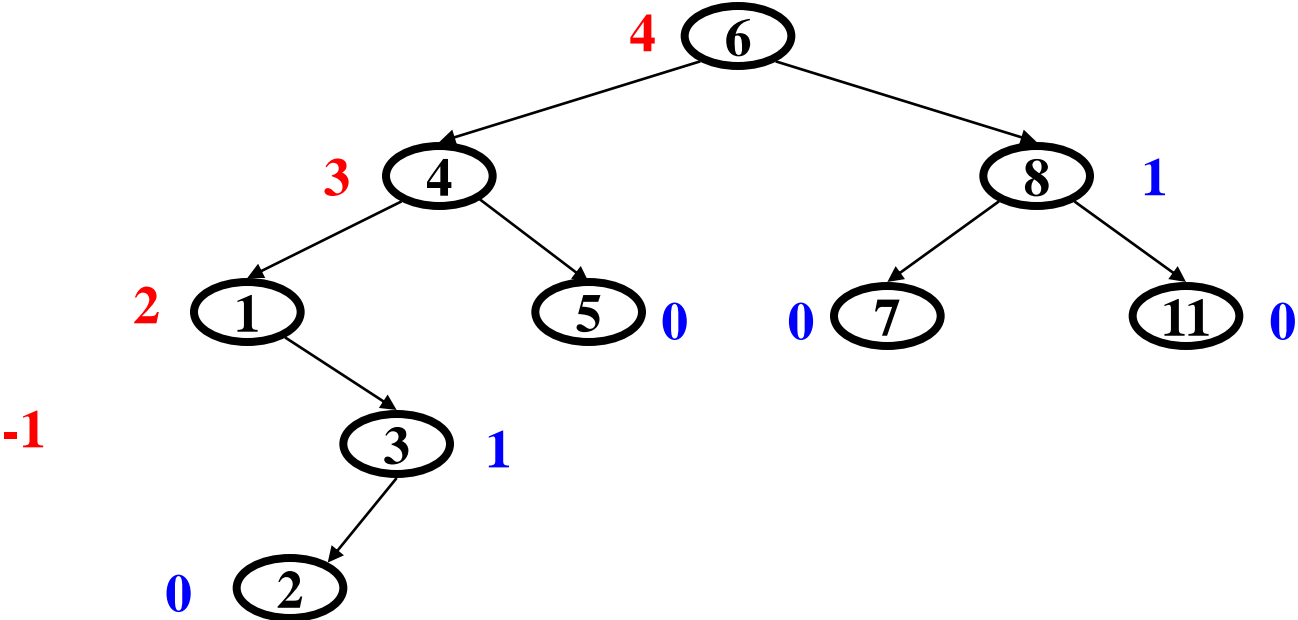
$$S(3) = 7$$

$$S(4) = 12$$



Solution of Recurrence:  $S(h) \approx 1.62^h$

# An AVL Tree?



# *AVL Tree Operations*

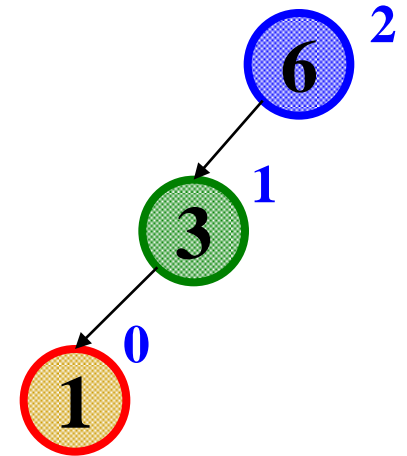
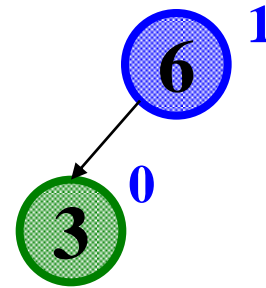
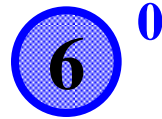
- **AVL find:**
  - Same as **BST find**
- **AVL insert:**
  - Same as **BST insert**
    - then check balance and potentially fix the AVL tree
    - four different imbalance cases
- **AVL delete:**
  - As with insert, do the deletion and then handle imbalance

# Example

Insert(6)

Insert(3)

Insert(1)



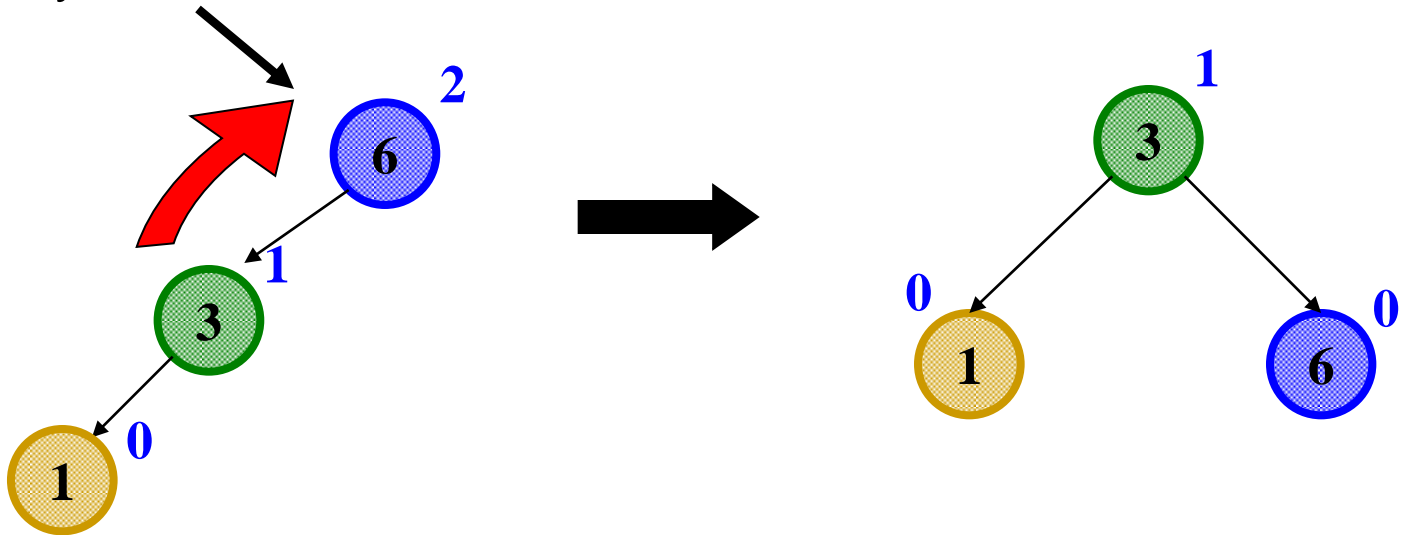
Third insertion violates balance

What is the only way to fix this?

# Single Rotation

- *Single rotation*: The basic operation we use to rebalance
  - Move child of unbalanced node into parent position
  - Parent becomes a “other” child
  - Other subtrees move in **the only way allowed by the BST**

AVL Property violated here





# *Insert and Detect Potential Imbalance*

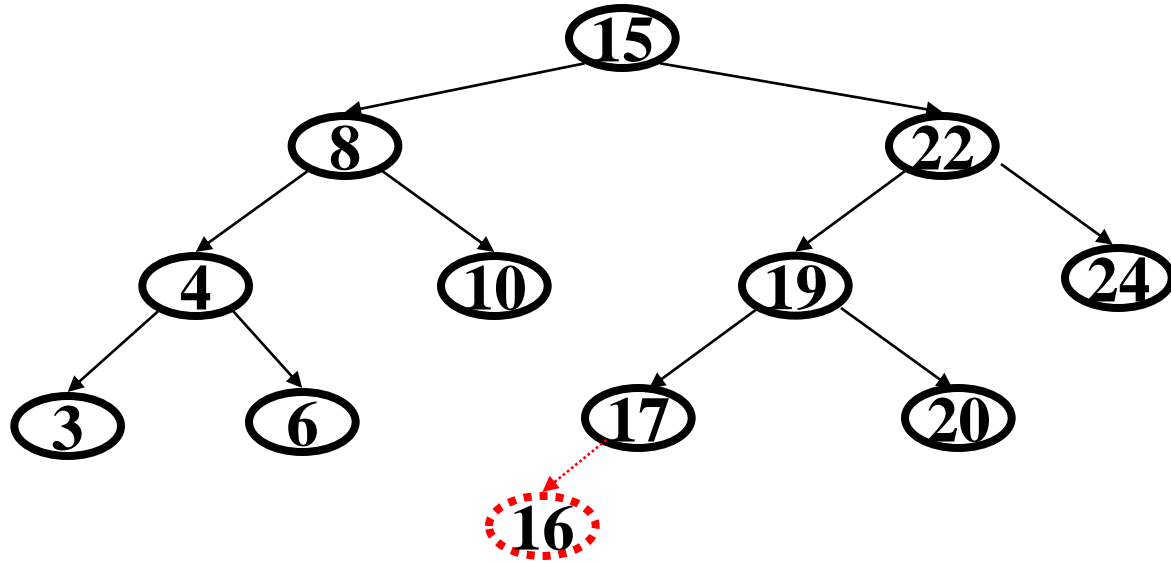
1. Insert the new node (at a leaf, as in a BST)
2. For each node on the path from the new leaf to the root  
the insertion may, or may not, have changed the node's height
3. After recursive insertion in a subtree  
detect height imbalance  
perform a *rotation* to restore balance at that node

*All the action is in defining the correct rotations to restore balance*

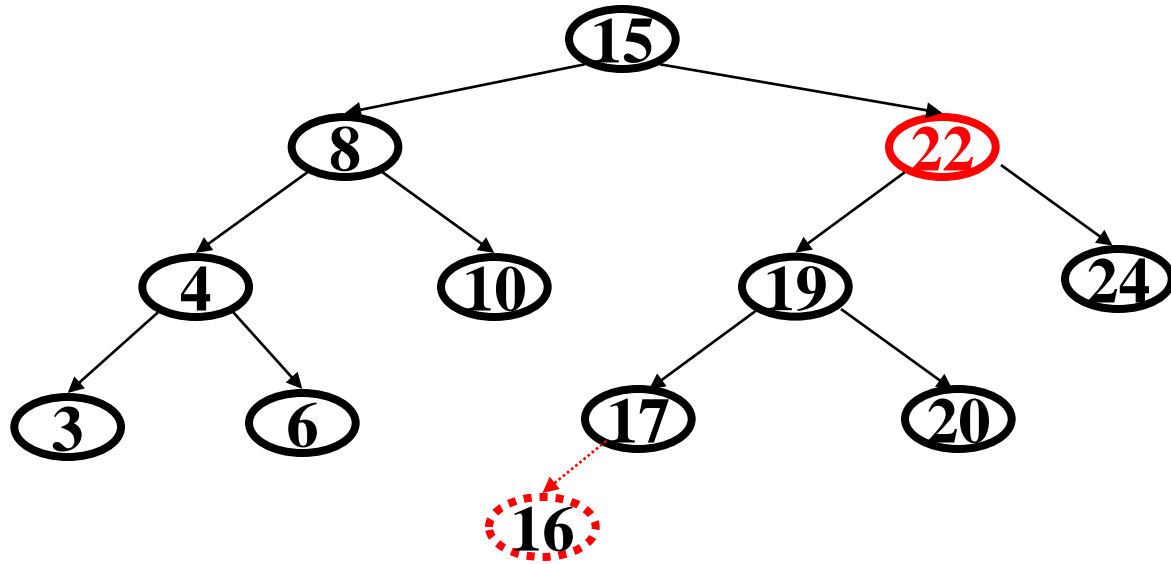
Fact that an implementation can ignore:

- There must be a deepest element that is imbalanced
- After rebalancing this deepest node, every node is balanced
- So at most one node needs to be rebalanced

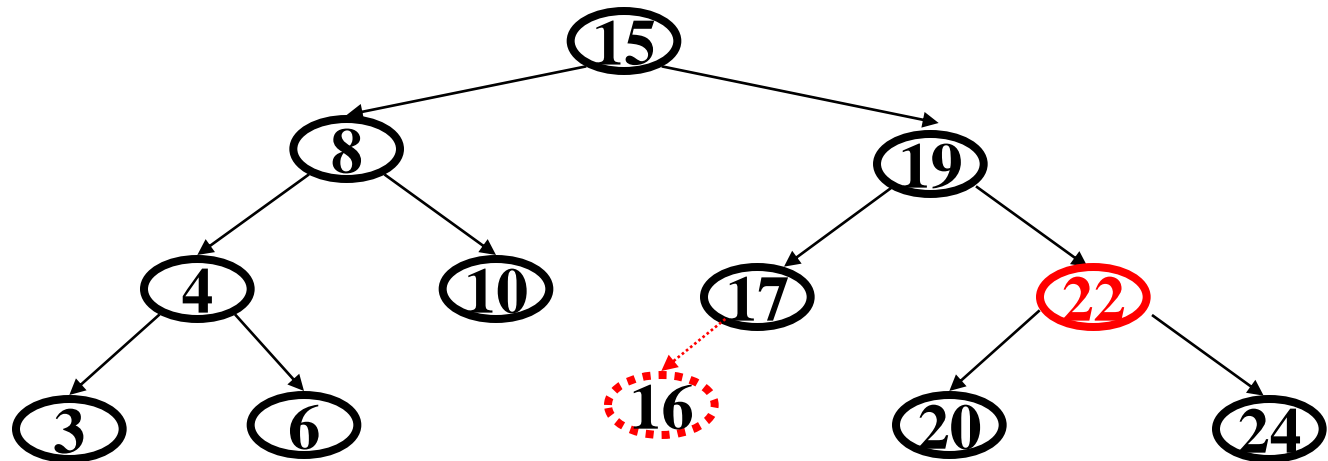
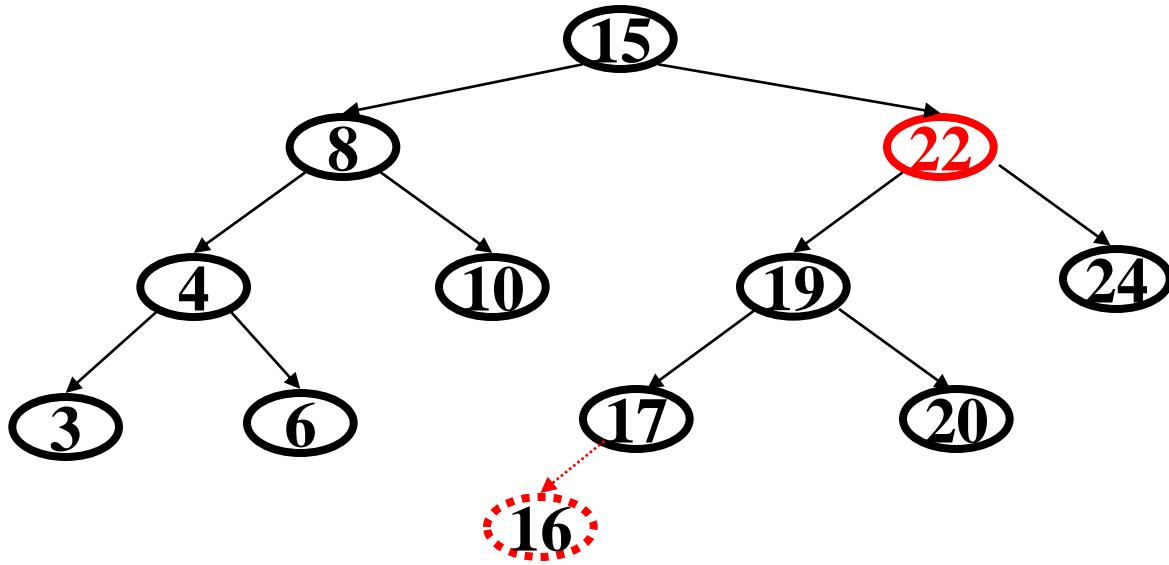
# Single Rotation Example: Insert(16)



# Single Rotation Example: Insert(16)

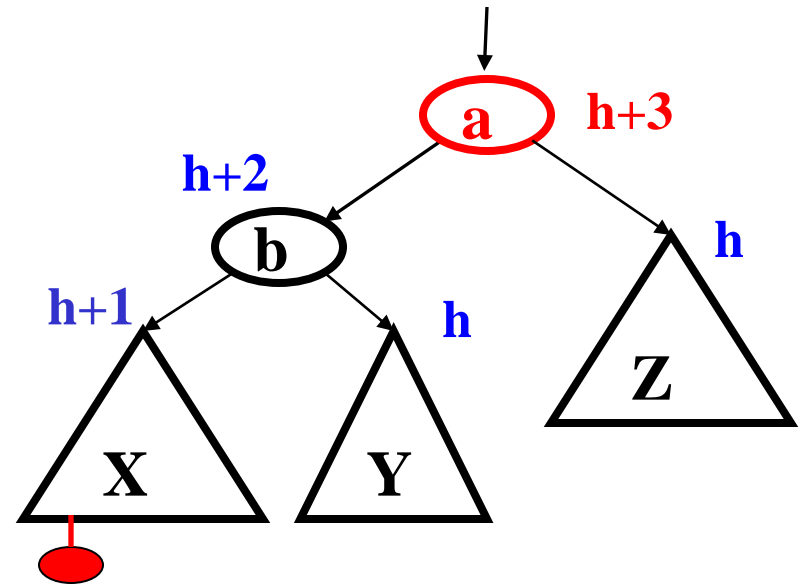
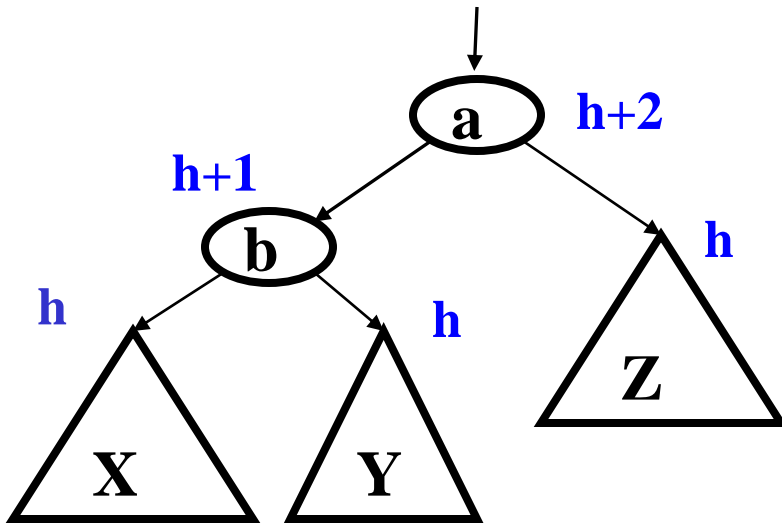


# Single Rotation Example: Insert(16)



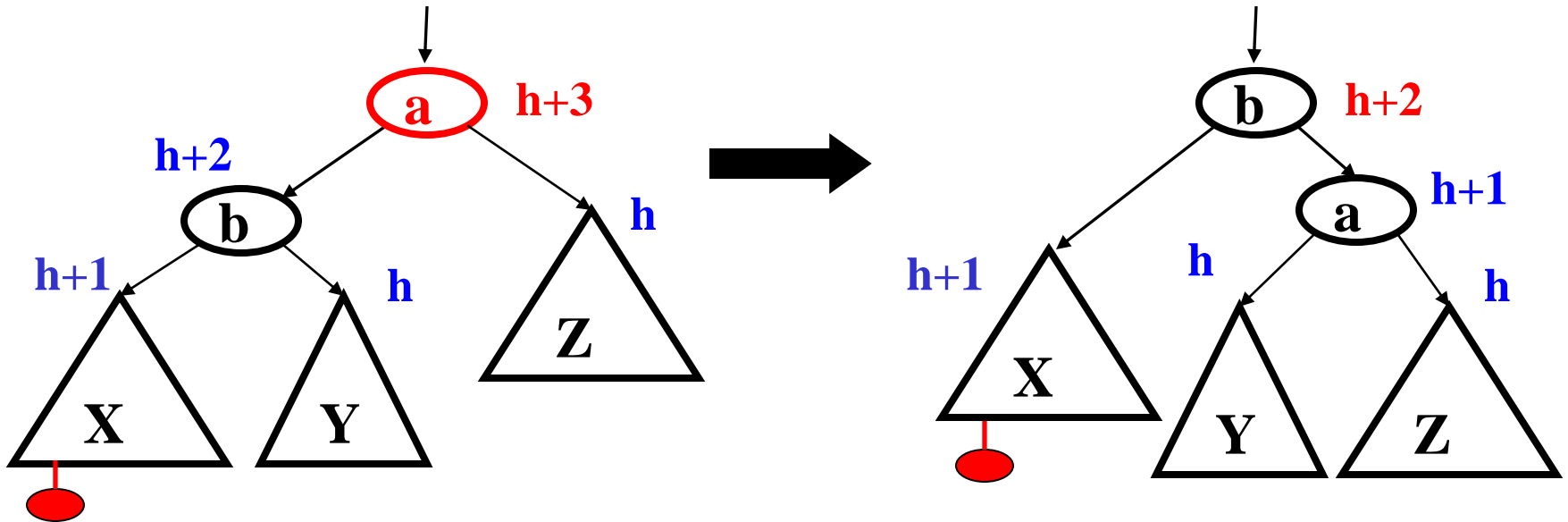
# Left-Left Case

- Node imbalanced due to insertion in **left-left grandchild**
  - This is 1 of 4 possible imbalance cases
- First we did the insertion, which made **a** imbalanced



# Left-Left Case

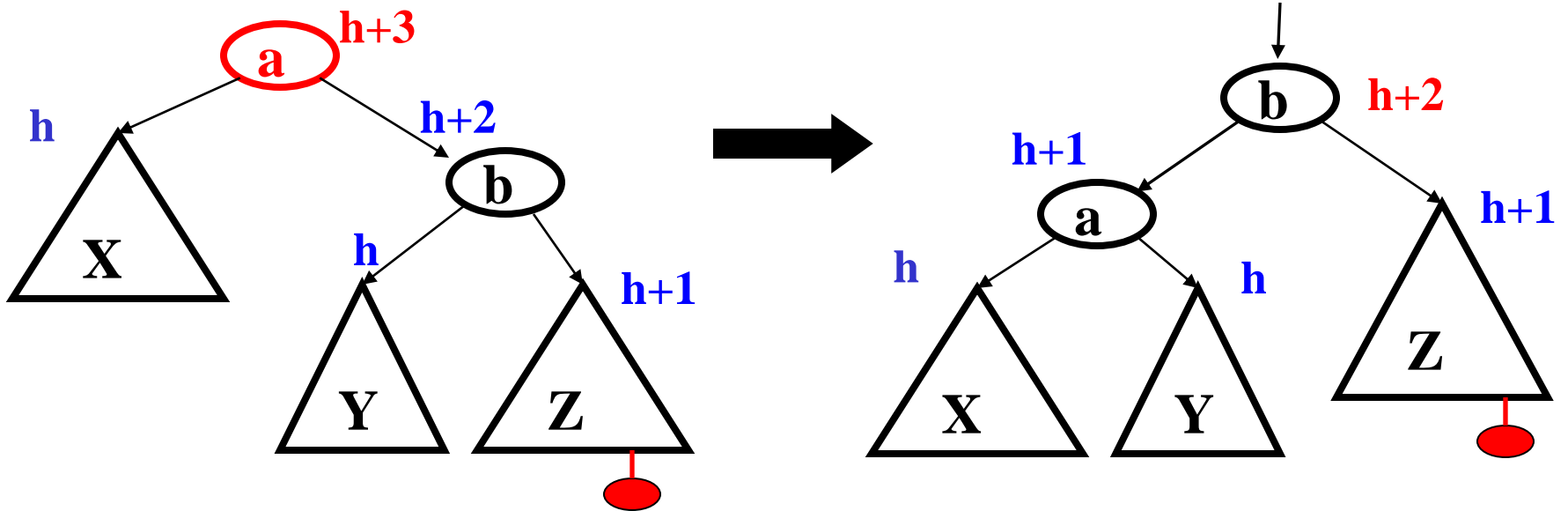
- So we rotate at  $a$ , using BST facts:  $X < b < Y < a < Z$



- A single rotation restores balance at the node
  - Is same height as before insertion, so ancestors now balanced

# Right-Right Case

- Mirror image to left-left case, so you rotate the other way
  - Exact same concept, but need different code

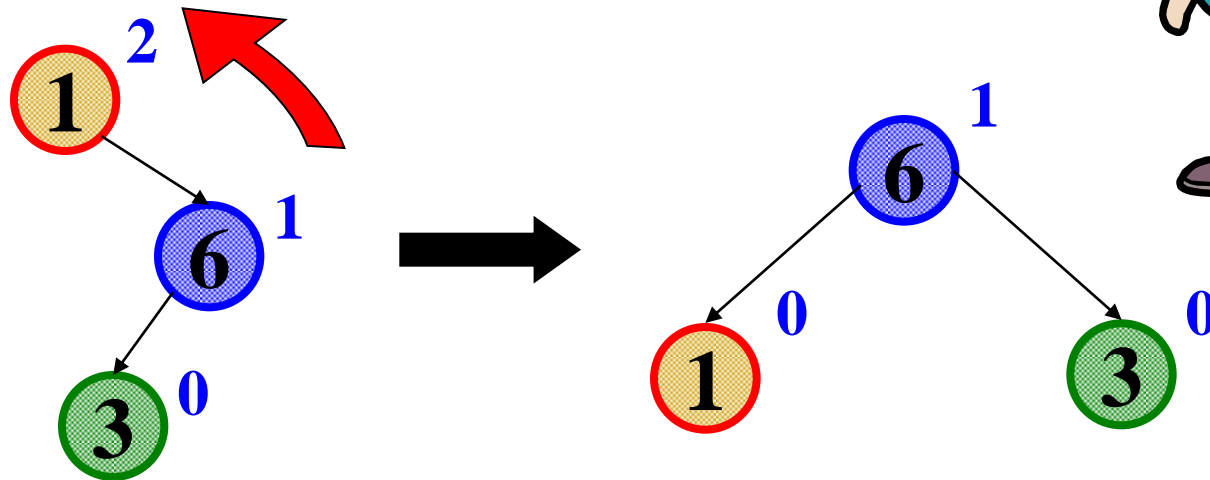


# The Other Two Cases

Single rotations not enough for insertions left-right or right-left subtree

Simple example: `insert(1)`, `insert(6)`, `insert(3)`

First wrong idea: single rotation as before



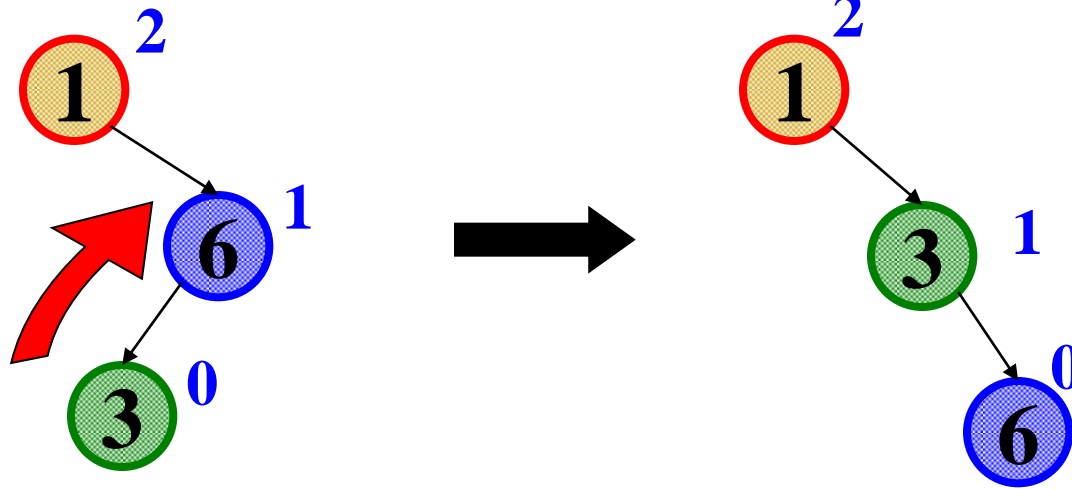


# The Other Two Cases

Single rotations not enough for insertions left-right or right-left subtree

Simple example: `insert(1)`, `insert(6)`, `insert(3)`

Second wrong idea: single rotation on child

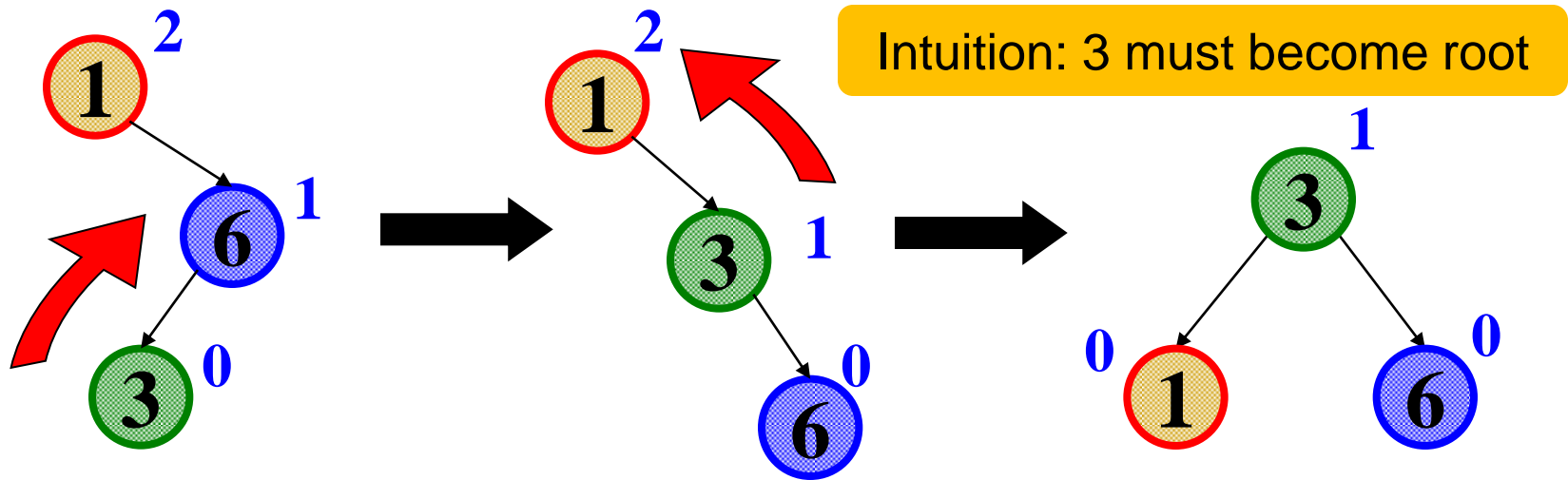


# Double Rotation

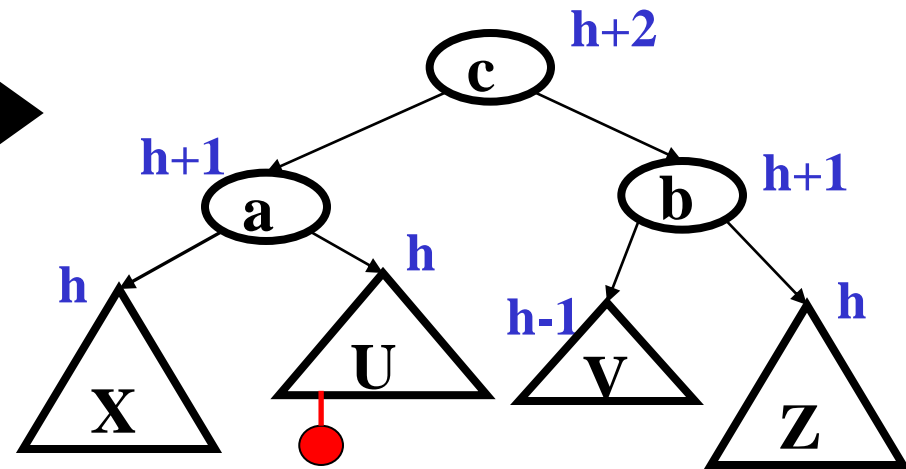
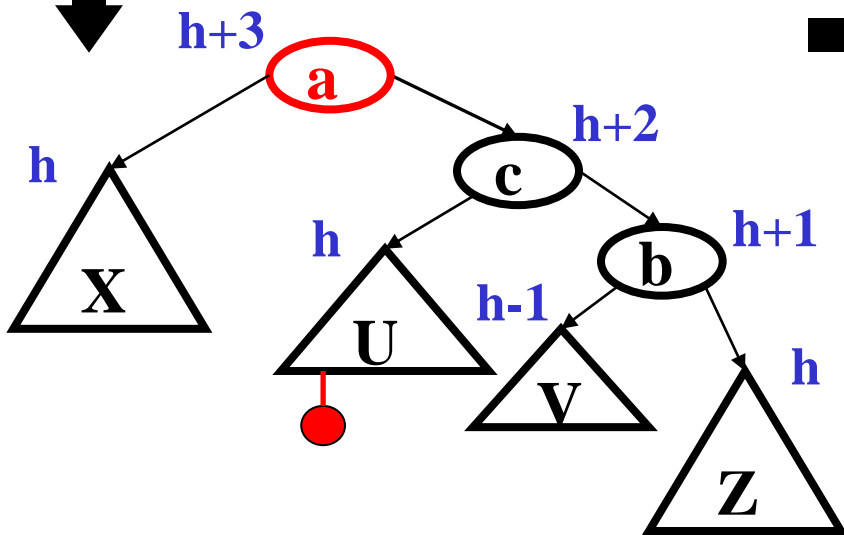
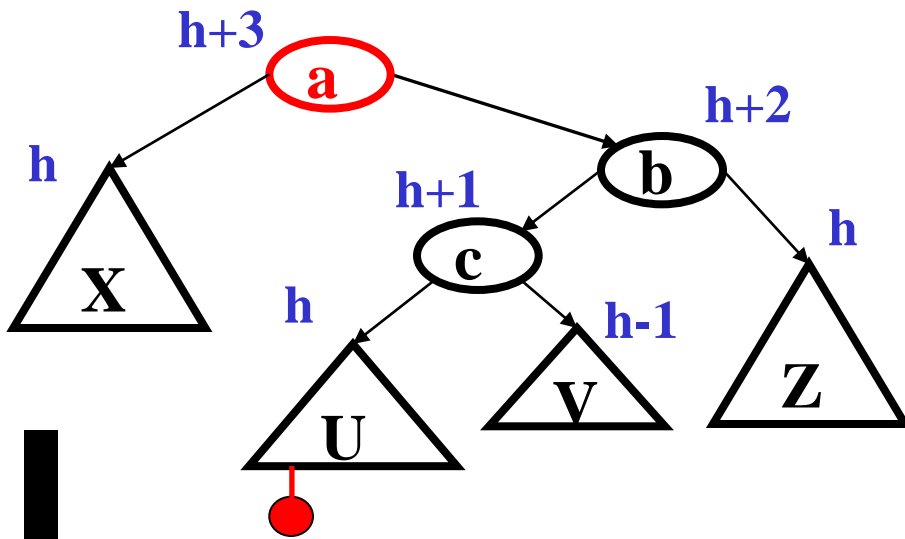
- First attempt at rotation violated the BST property
- Second attempt at rotation did not fix balance
- But if we do both, it works!

Double rotation:

1. Rotate problematic child and grandchild
2. Then rotate between self and new child

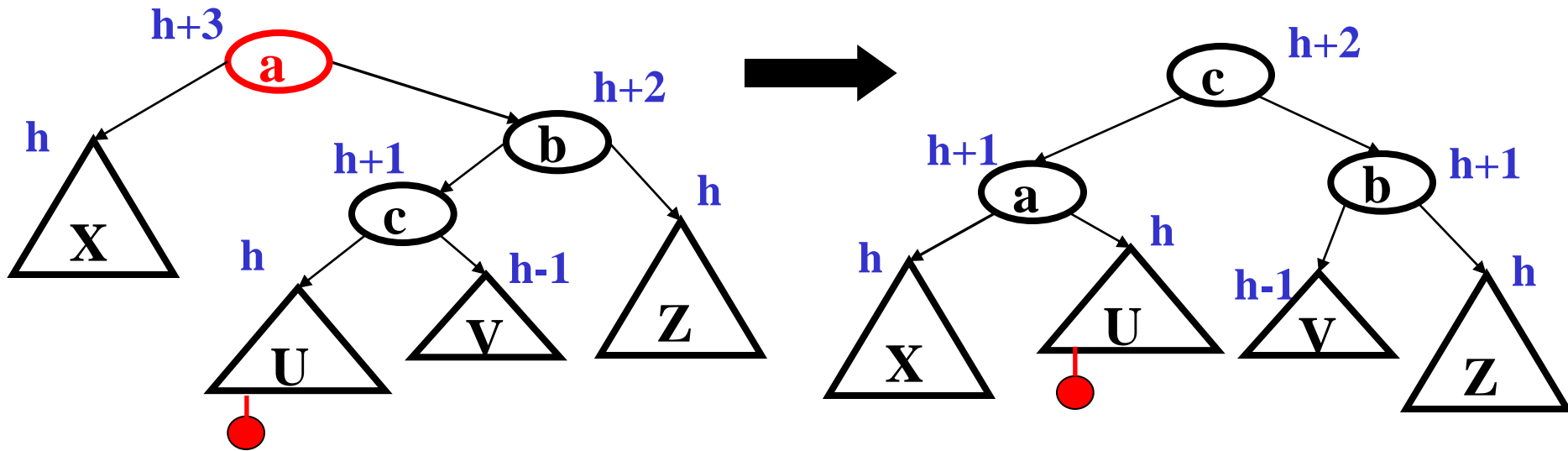


# Right-Left Case



# Right-Left Case

- Height of the subtree after rebalancing is the same as before insert
  - So no ancestor in the tree will need rebalancing
- Does not have to be implemented as two rotations; can just do:



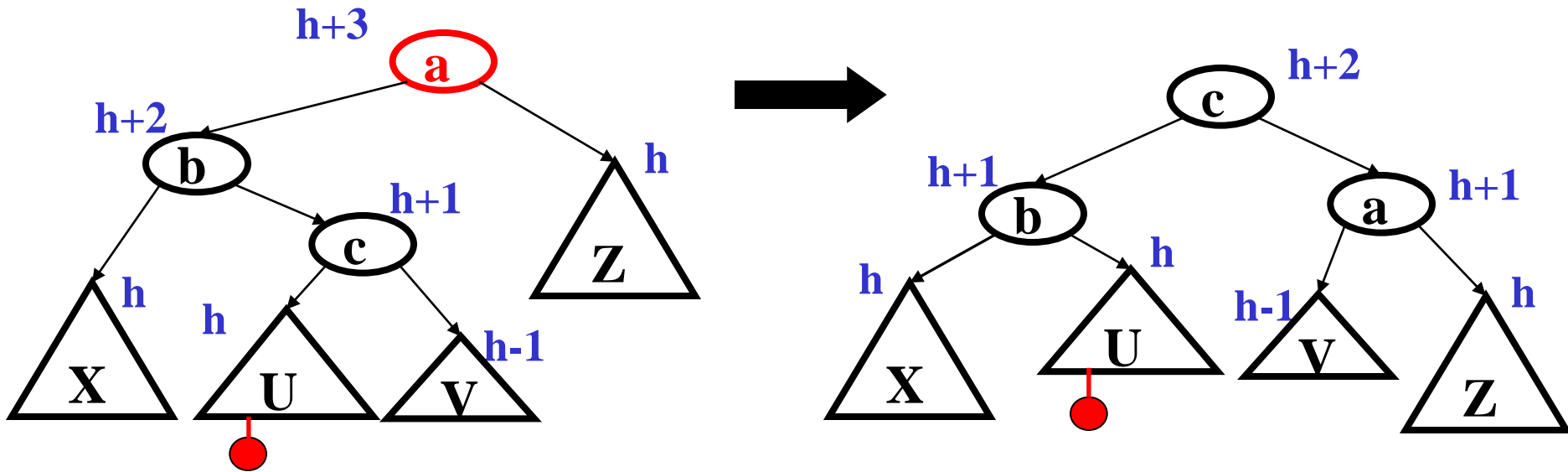
Easier to remember than you may think:

Move **c** to grandparent's position

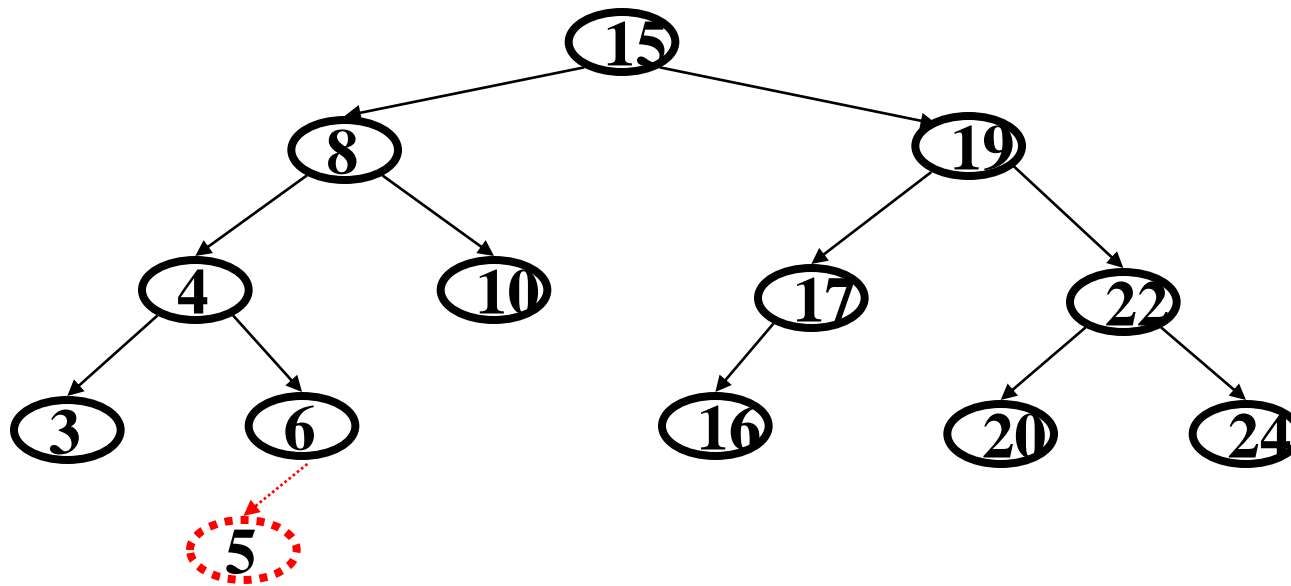
Put **a**, **b**, **X**, **U**, **V**, and **Z** in the **only legal position** for a BST

# Left-Right Case

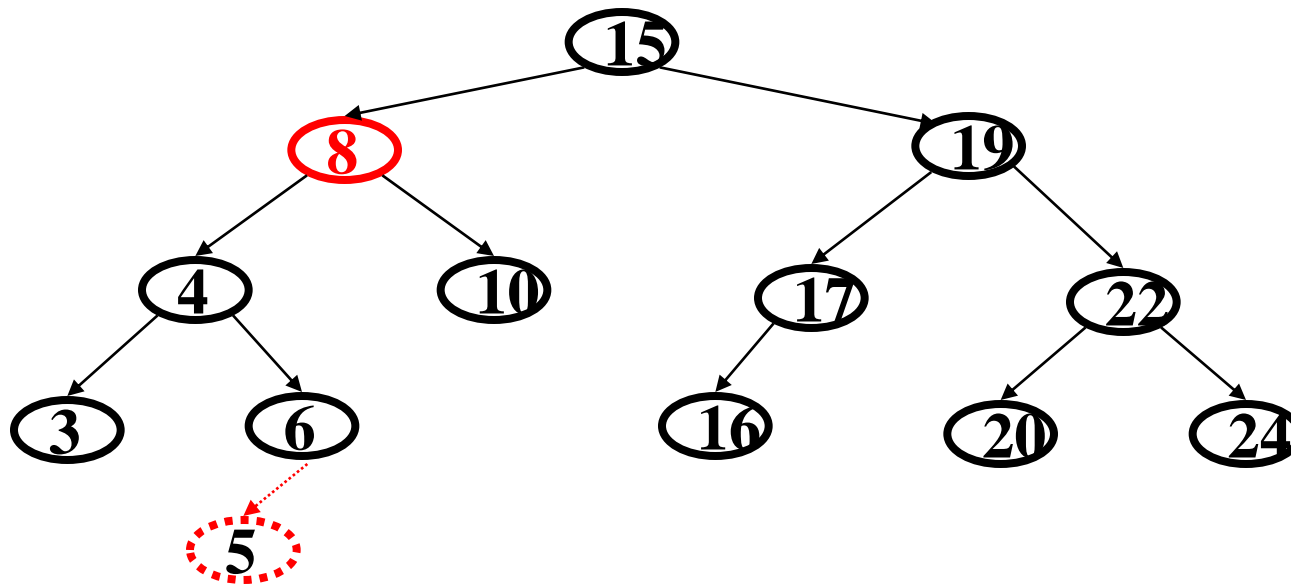
- Mirror image of right-left
  - No new concepts, just additional code to write



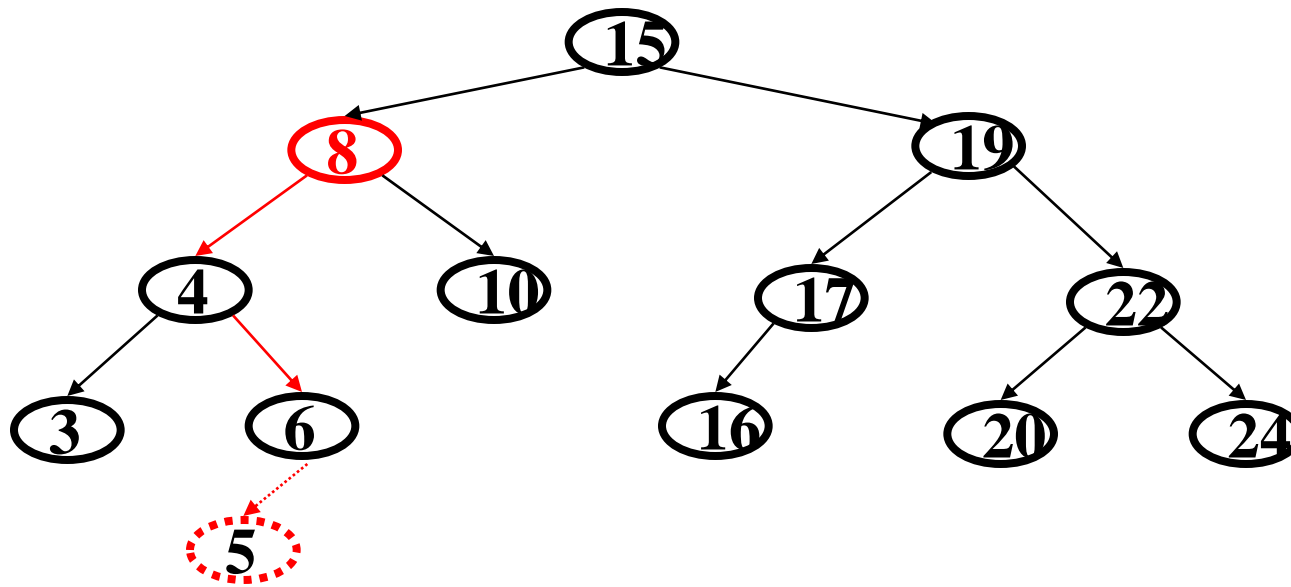
# Double Rotation Example: Insert(5)



# Double Rotation Example: Insert(5)

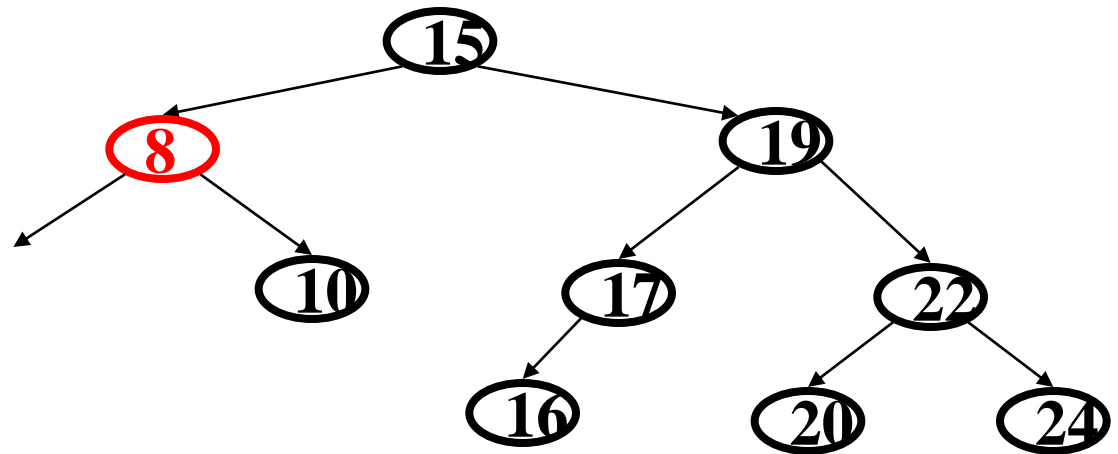
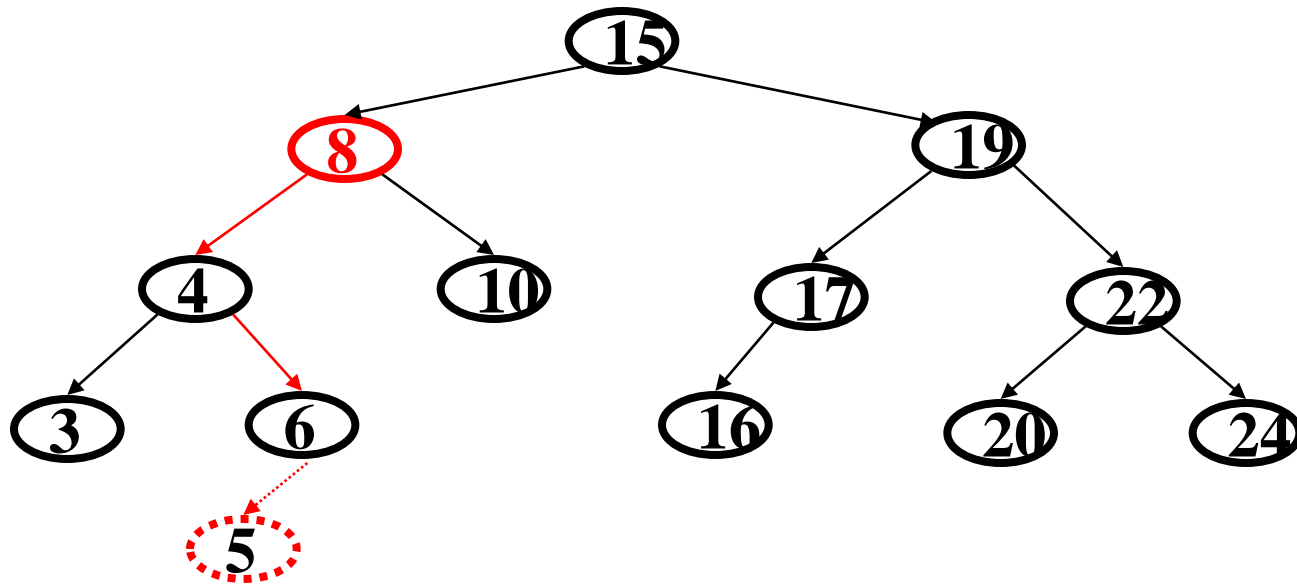


# Double Rotation Example: Insert(5)

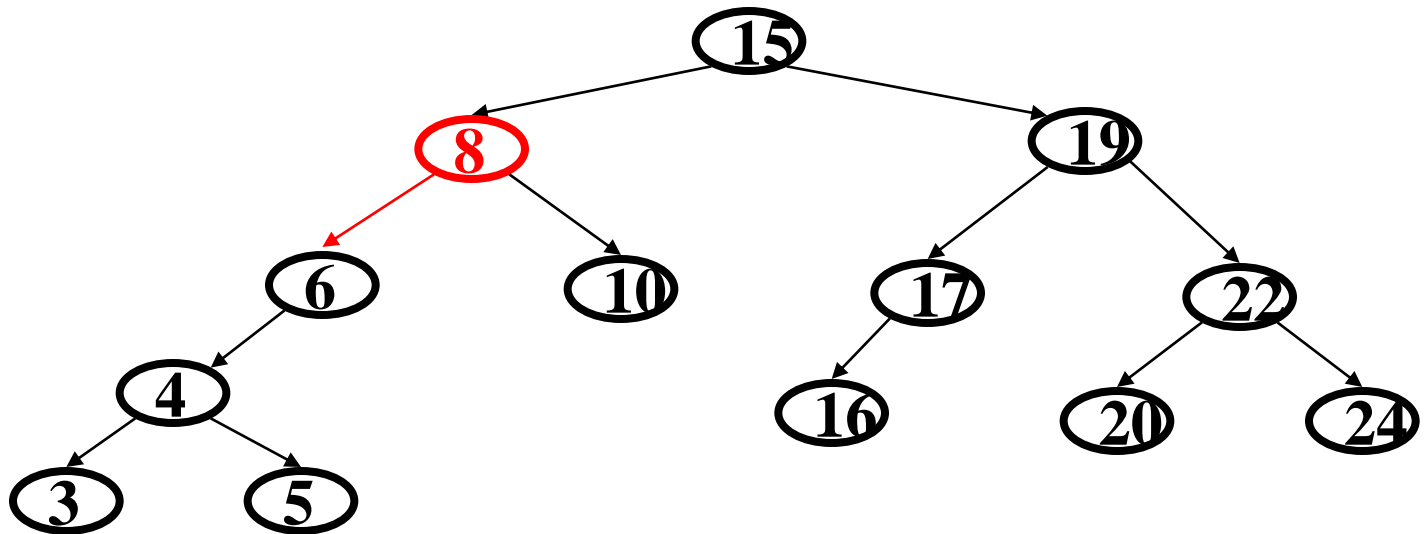
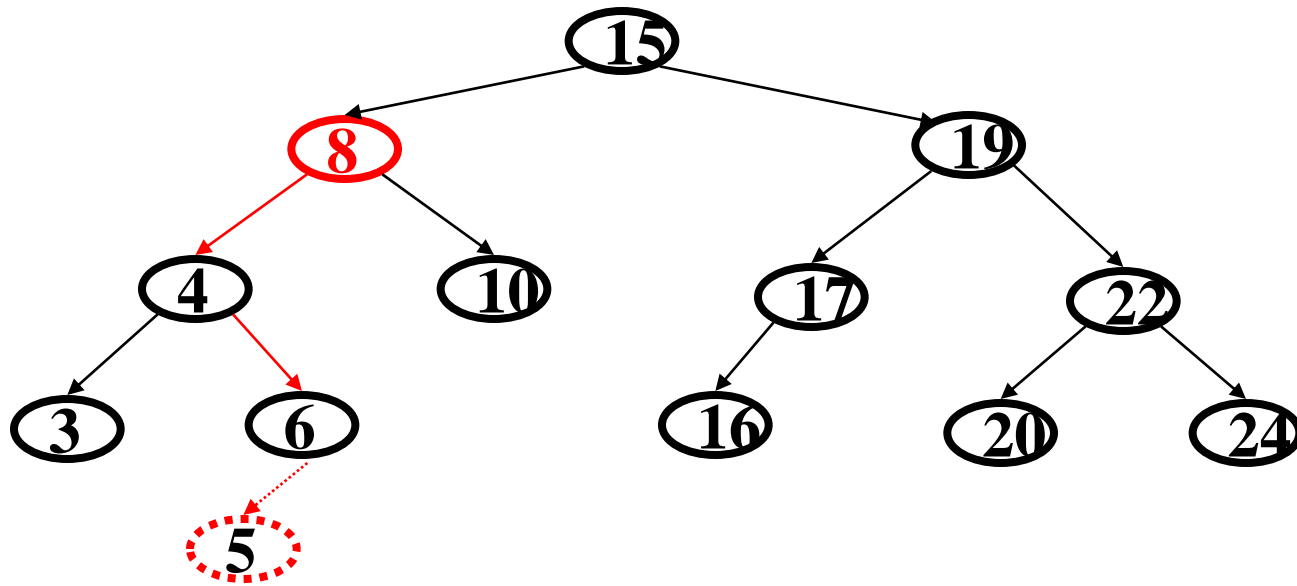




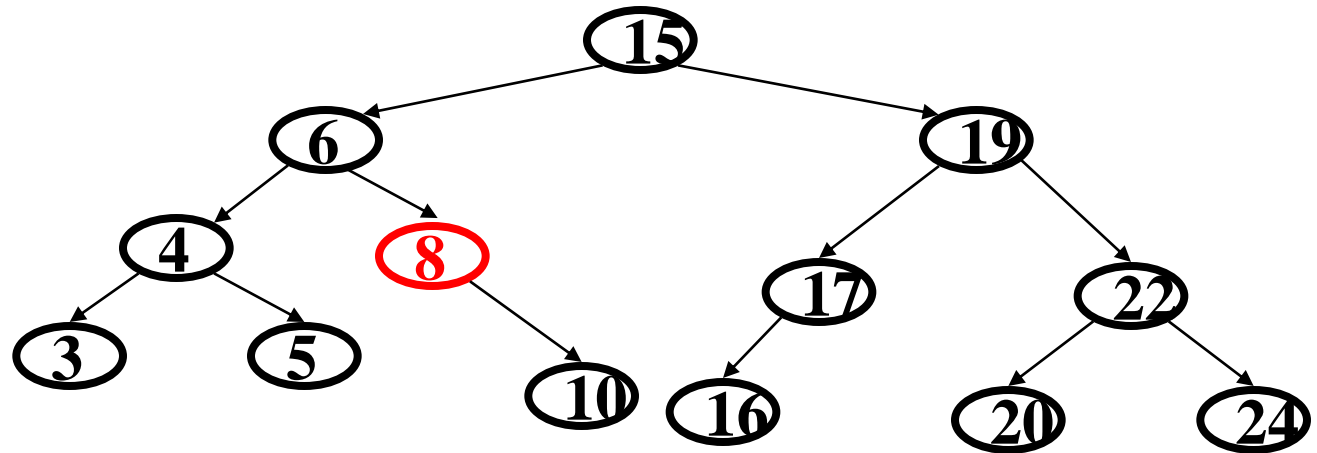
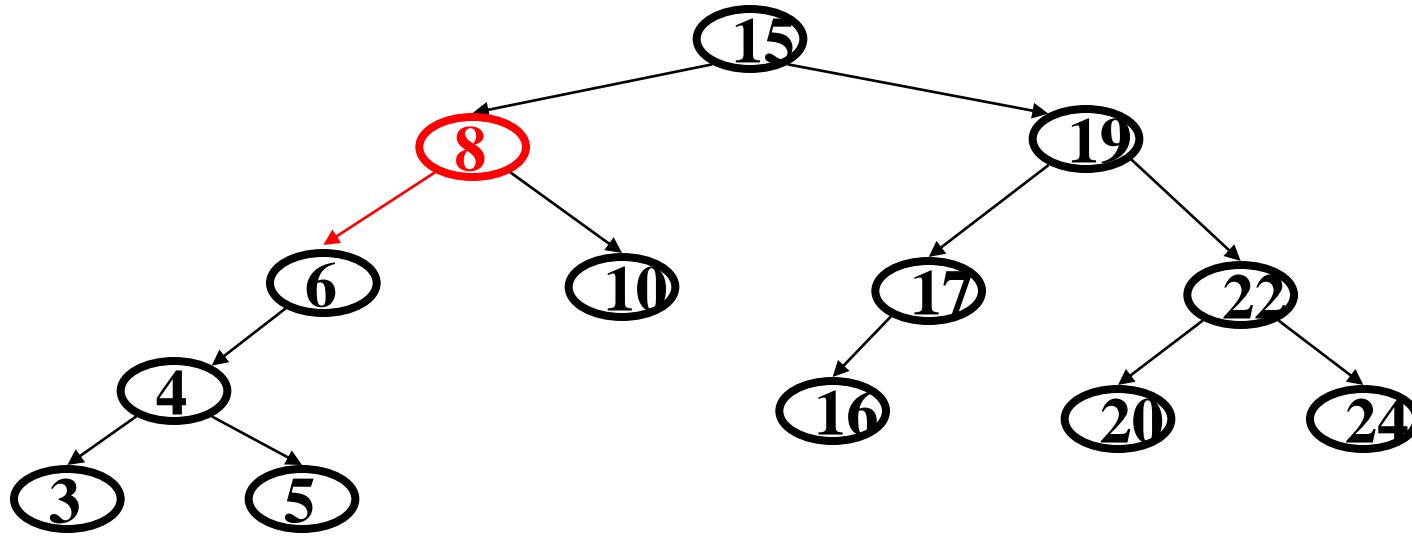
# Double Rotation Example: Insert(5)



# Double Rotation Example: Insert(5)



# Double Rotation Example: Insert(5)



# *Summarizing Insert*

- Insert as in a BST
- Check back up path for imbalance, which will be 1 of 4 cases:
  - node's left-left grandchild is too tall
  - node's left-right grandchild is too tall
  - node's right-left grandchild is too tall
  - node's right-right grandchild is too tall
- Only one case can occur, because tree was balanced before insert
- After the single or double rotation, the smallest-unbalanced subtree now has the same height as before the insertion
  - So all ancestors are now balanced

# *Efficiency*

Worst-case complexity of **find**:  $O(\log n)$

Worst-case complexity of **insert**:  $O(\log n)$

- Rotation is  $O(1)$  and there's an  $O(\log n)$  path to root
- Same complexity even without “one-rotation-is-enough” fact

Worst-case complexity of **buildTree**:  $O(n \log n)$

# Delete

We will not cover delete

- Multiple snow days, something has to give

Do the delete as in a BST, then balance path up from deleted node

- Which may be predecessor or successor

Single and double rotate based on height imbalance

- You are coming up the shorter subtree
- But need to pull up the taller subtree

Rotation reduces height of the tree

- So you need to check all the way to the root

`delete` is also  $O(\log n)$



# CSE332: Data Abstractions

## Lecture 7: B Trees

James Fogarty

Winter 2012

# The Dictionary (a.k.a. Map) ADT

- Data:
  - Set of (key, value) *pairs*
  - keys must be *comparable*

- Operations:

- `insert(key, value)`
- `find(key)`
- `delete(key)`
- ...

`insert(jfogarty, ...)`

`find(trobison)`

Tyler, Robison, ...



*We will tend to emphasize the keys,  
don't forget about the stored values*



# *Comparison: The Set ADT*

The *Set* ADT is like a Dictionary without any values

- A key is *present* or not (i.e., there are no repeats)

For **find**, **insert**, **delete**, there is little difference

- In dictionary, values are “just along for the ride”
- So *same data structure ideas* work for dictionaries and sets

But if your Set ADT has other important operations this may not hold

- **union**, **intersection**, **is\_subset**
- Notice these are binary operators on sets
- There are other approaches to these kinds of operations

# *Dictionary Data Structures*

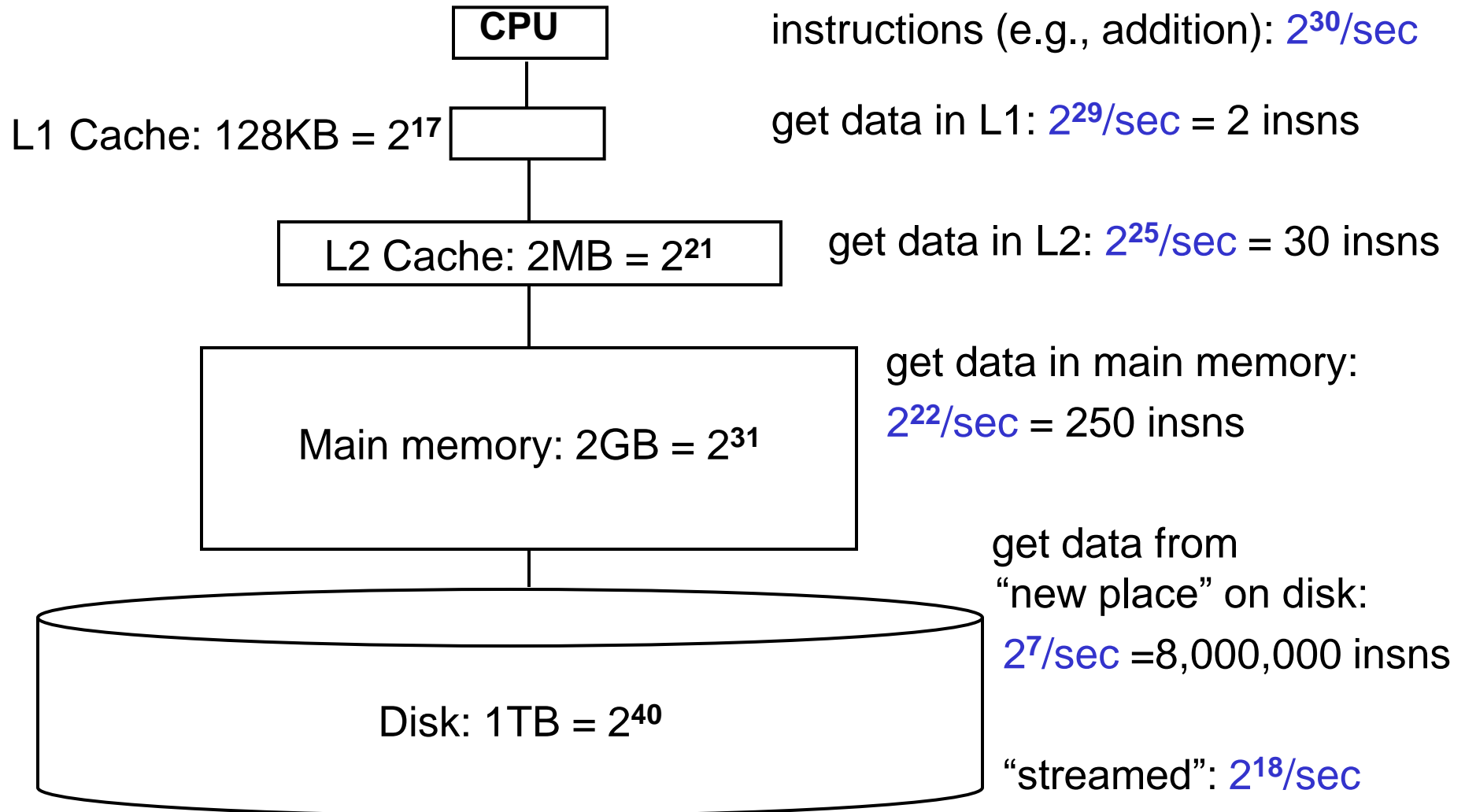
We will see three different data structures implementing dictionaries

1. AVL trees
  - Binary search trees with *guaranteed balancing*
2. B-Trees
  - Also always balanced, but different and shallower
3. Hashtables
  - Not tree-like at all

Skipping: Other balanced trees (e.g., red-black, splay)

# A Typical Hierarchy

*A plausible configuration ...*



# *Morals*

It is much faster to do:	Than:
5 million arithmetic ops	1 disk access
2500 L2 cache accesses	1 disk access
400 main memory accesses	1 disk access

Why are computers built this way?

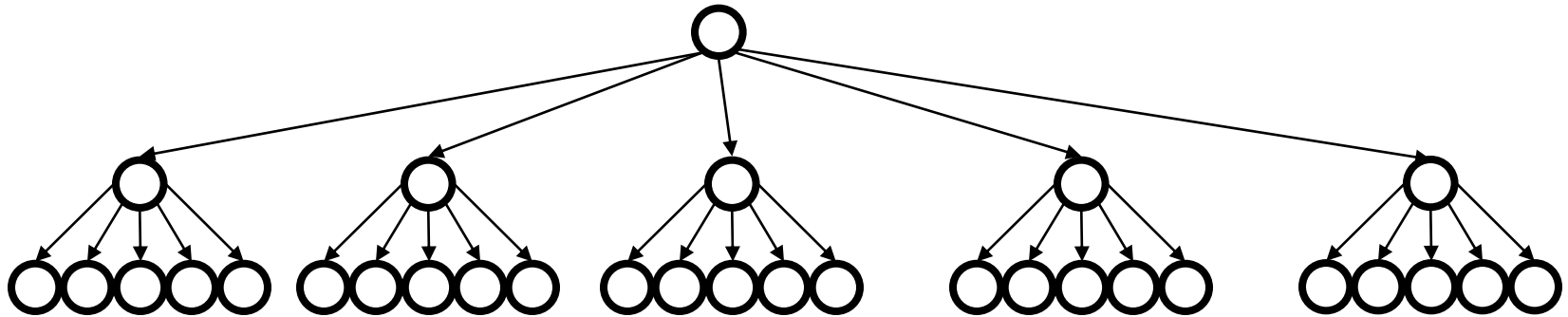
- Physical realities (speed of light, closeness to CPU)
- Cost (price per byte of different technologies)
- Disks get much bigger not much faster
  - Spinning at 7200 RPM accounts for much of the slowness and unlikely to spin faster in the future
- Speedup at higher levels makes lower levels *relatively slower*

# *Block and Line Size*

- Moving data up the memory hierarchy is slow because of *latency*
  - Might as well send more, just in case
  - Send nearby memory because:
    - It is easy, we are here anyways
    - And likely to be asked for soon (locality of reference)
- Amount moved from disk to memory is called “block” or “page” size
  - Not under program control
- Amount moved from memory to cache is called the “line” size
  - Not under program control

# *M-ary Search Tree*

- Build some sort of search tree with branching factor  $M$ :
  - Have an array of sorted children (**Node** [ ])
  - Choose  $M$  to fit snugly into a disk block (1 access for array)



Perfect tree of height  $h$  has  $(M^{h+1}-1)/(M-1)$  nodes (textbook, page 4)

# hops for **find**: If balanced, using  $\log_M n$  instead of  $\log_2 n$

- If  $M=256$ , that's an 8x improvement
- If  $n = 2^{40}$  that's 5 levels instead of 40 (i.e., 5 disk accesses)

Runtime of **find** if balanced:  $O(\log_2 M \log_M n)$

**(binary search children) (walk down the tree)**

# *Problems with M-ary Search Trees*

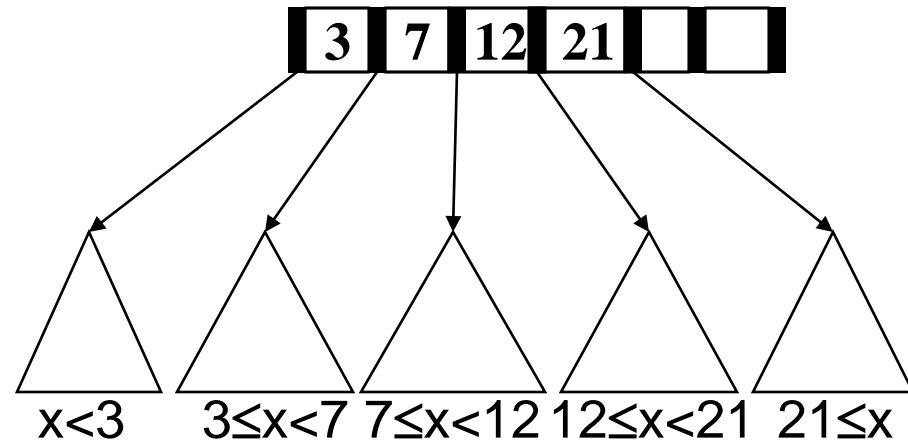
- What should the order property be?
- How would you rebalance (ideally without more disk accesses)?
- Any “useful” data at the internal nodes takes up disk-block space without being used by finds moving past it

Use the branching-factor idea, but for a different kind of balanced tree

- Not a binary search tree
- But still logarithmic height for any  $M > 2$

# *B+ Trees* (we will just say “B Trees”)

- Two types of nodes:
  - internal nodes and leaf nodes
- Each internal node has room for up to  $M-1$  keys and  $M$  children
  - no data; all data at the leaves!
- Order property:
  - Subtree between  $x$  and  $y$ 
    - Data that is  $\geq x$  and  $< y$
  - Notice the  $\geq$
- Leaf has up to  $L$  sorted data items

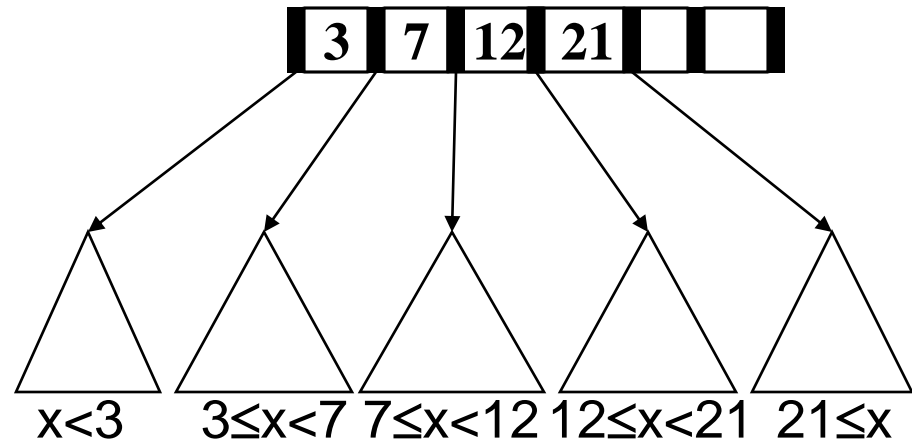


As usual, we will ignore the presence of data in our examples

Remember it is actually not there for internal nodes



# Find



- We are accustomed to data at internal nodes
- But `find` is still an easy root-to-leaf recursive algorithm
  - At each internal node do binary search on the  $\leq M-1$  keys
  - At the leaf do binary search on the  $\leq L$  data items
- To get logarithmic running time, we need a balance condition

# *Structure Properties*

- **Root** (special case)
  - If tree has  $\leq L$  items, root is a leaf (occurs when starting up, otherwise very unusual)
  - Else has between 2 and  $M$  children
- **Internal Nodes**
  - Have between  $\lceil M/2 \rceil$  and  $M$  children (i.e., at least half full)
- **Leaf Nodes**
  - All leaves at the same depth
  - Have between  $\lceil L/2 \rceil$  and  $L$  data items (i.e., at least half full)

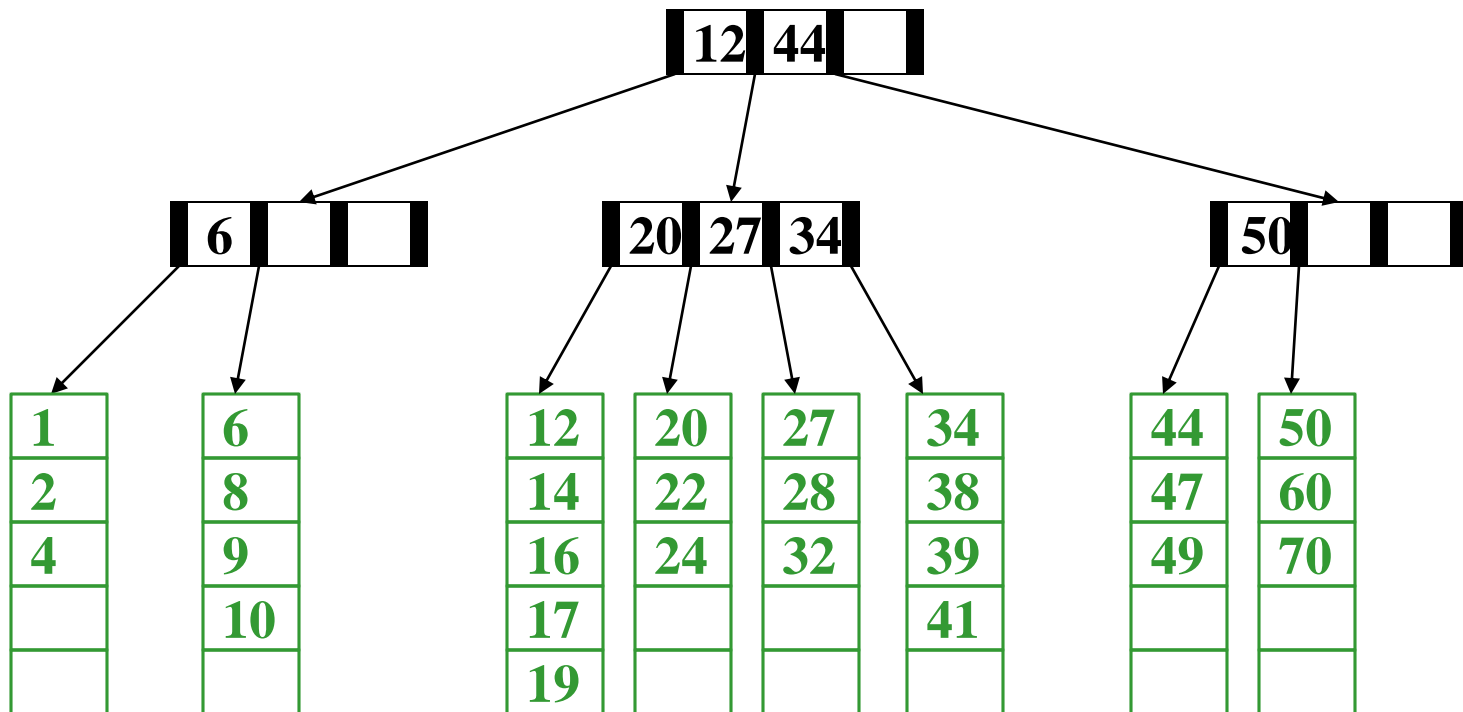
(Any  $M > 2$  and  $L$  will work; ***picked based on disk-block size***)

Note on notation: Inner nodes drawn horizontally, leaves vertically to distinguish. Including empty cells

## Example

Suppose  $M=4$  (max # children / pointers in **internal node**)  
and  $L=5$  (max # data items at **leaf**)

- All **internal nodes** have at least 2 children
- All **leaves** at same depth, have at least 3 data items



# Balanced enough

Not hard to show height  $h$  is logarithmic in number of data items  $n$

- Let  $M > 2$  (if  $M = 2$ , then a list tree is legal, which is no good)
- Because all nodes are at least half full (except root may have only 2 children) and all leaves are at the same level, the minimum number of data items  $n$  for a height  $h > 0$  tree is...

$$n \geq \underbrace{2 \lceil M/2 \rceil^{h-1}}_{\text{minimum number of leaves}} \underbrace{\lceil L/2 \rceil}_{\text{minimum data per leaf}}$$

**Exponential in height**  
because  $\lceil M/2 \rceil > 1$

# *Disk Friendliness*

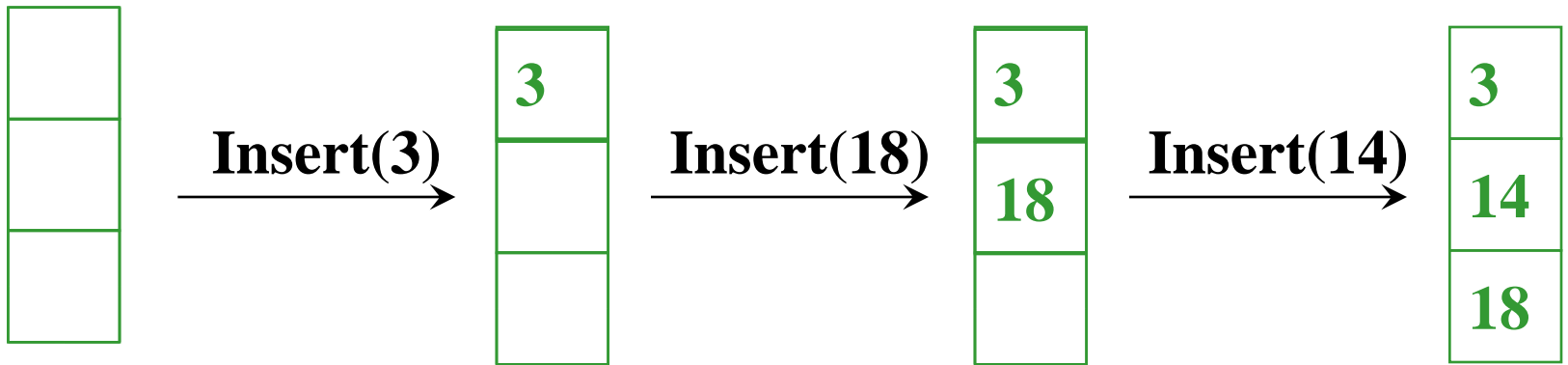
What makes B trees so disk friendly?

- Many keys stored in one **internal node**
  - All brought into memory in one disk access
  - But only if we pick  $M$  wisely
  - Makes the binary search over  $M-1$  keys totally worth it (insignificant compared to disk access times)
- **Internal nodes** contain only keys
  - Any **find** wants only one data item; wasteful to load unnecessary items with internal nodes
  - Only bring one **leaf** of data items into memory
  - Data-item size does not affect what  $M$  is

# *Maintaining Balance*

- So this seems like a great data structure, and it is
- But we haven't implemented the other dictionary operations yet
  - **insert**
  - **delete**
- As with AVL trees, the hard part is maintaining structure properties

# Building a B-Tree

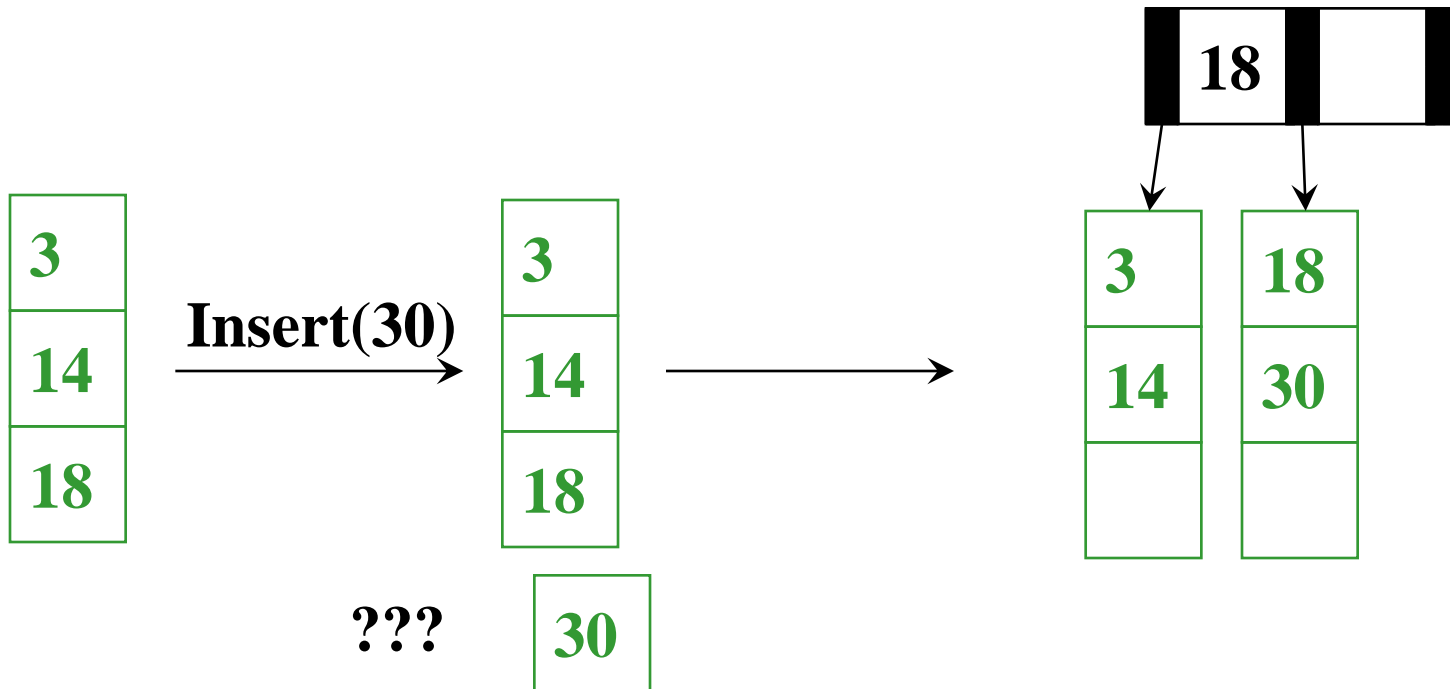


The empty B-Tree  
(the **root** will be a  
leaf at the beginning)

Simply need to  
keep data sorted

$$M = 3 \quad L = 3$$

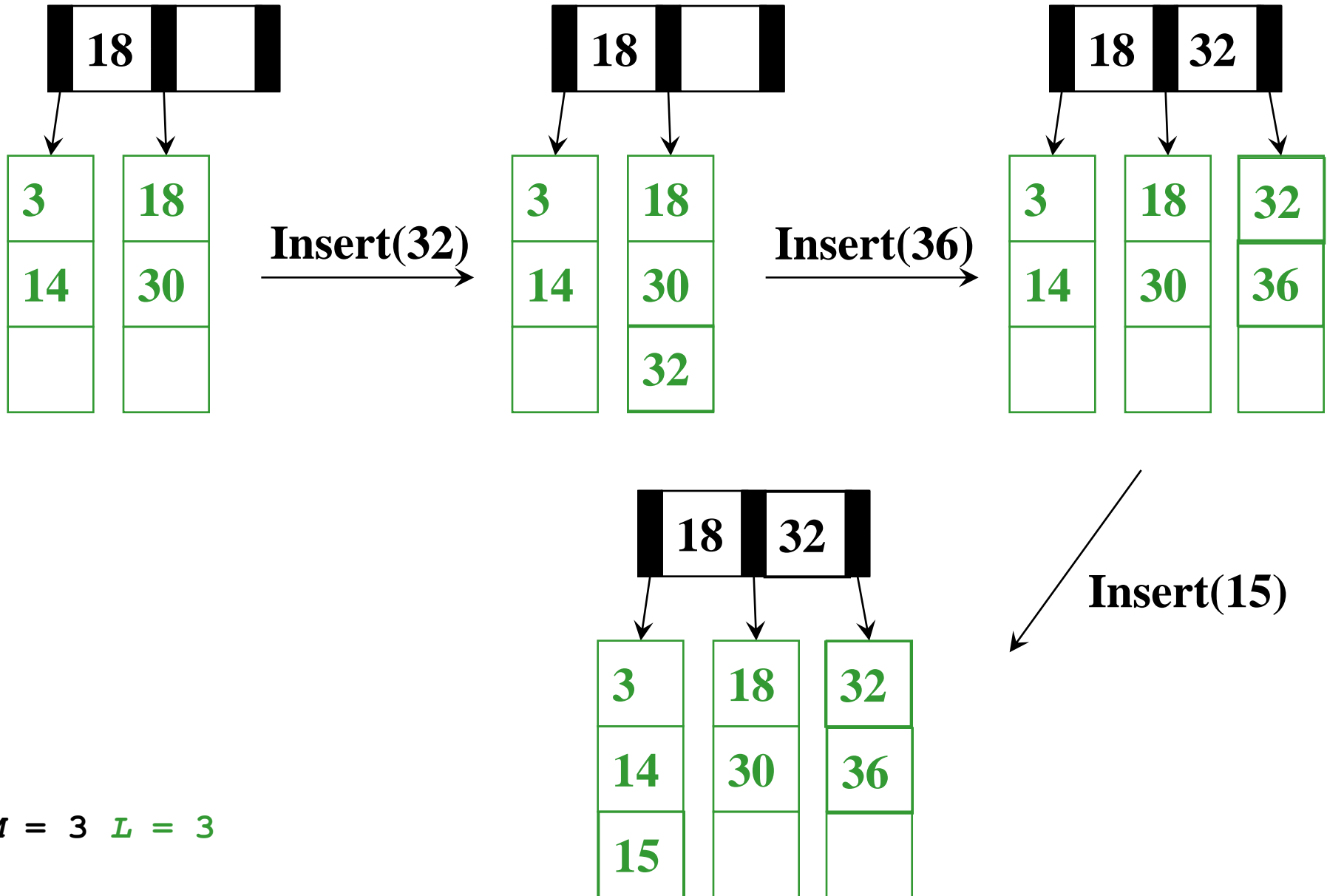
$M = 3$   $L = 3$



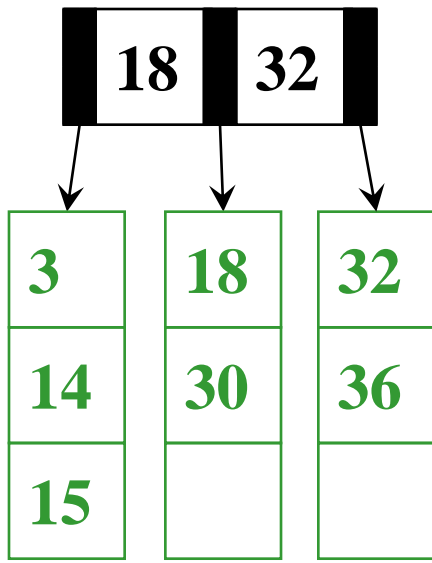
- When we ‘overflow’ a **leaf**, we split it into 2 **leaves**
- Parent gains another child
- If there is no parent, we create one
  
- How do we pick the new key?
  - Smallest element in right tree



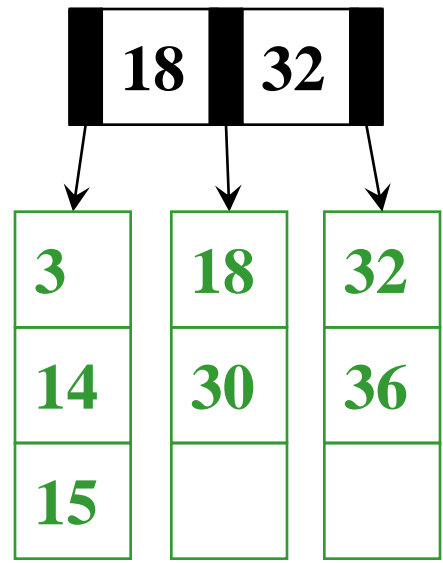
Split leaf again



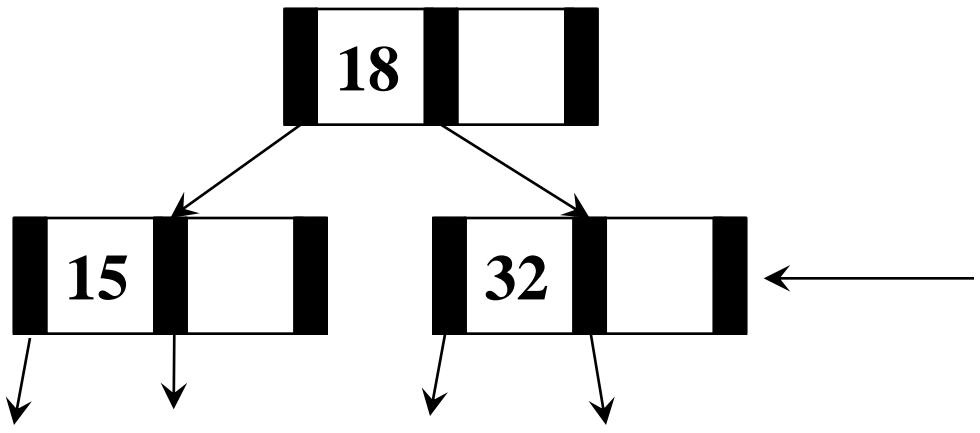
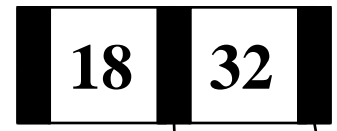
$M = 3$   $L = 3$



Insert(16) →

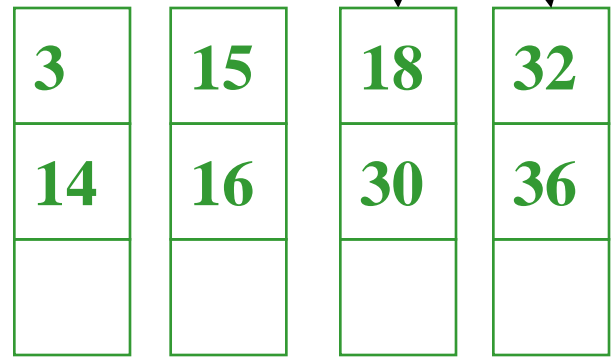


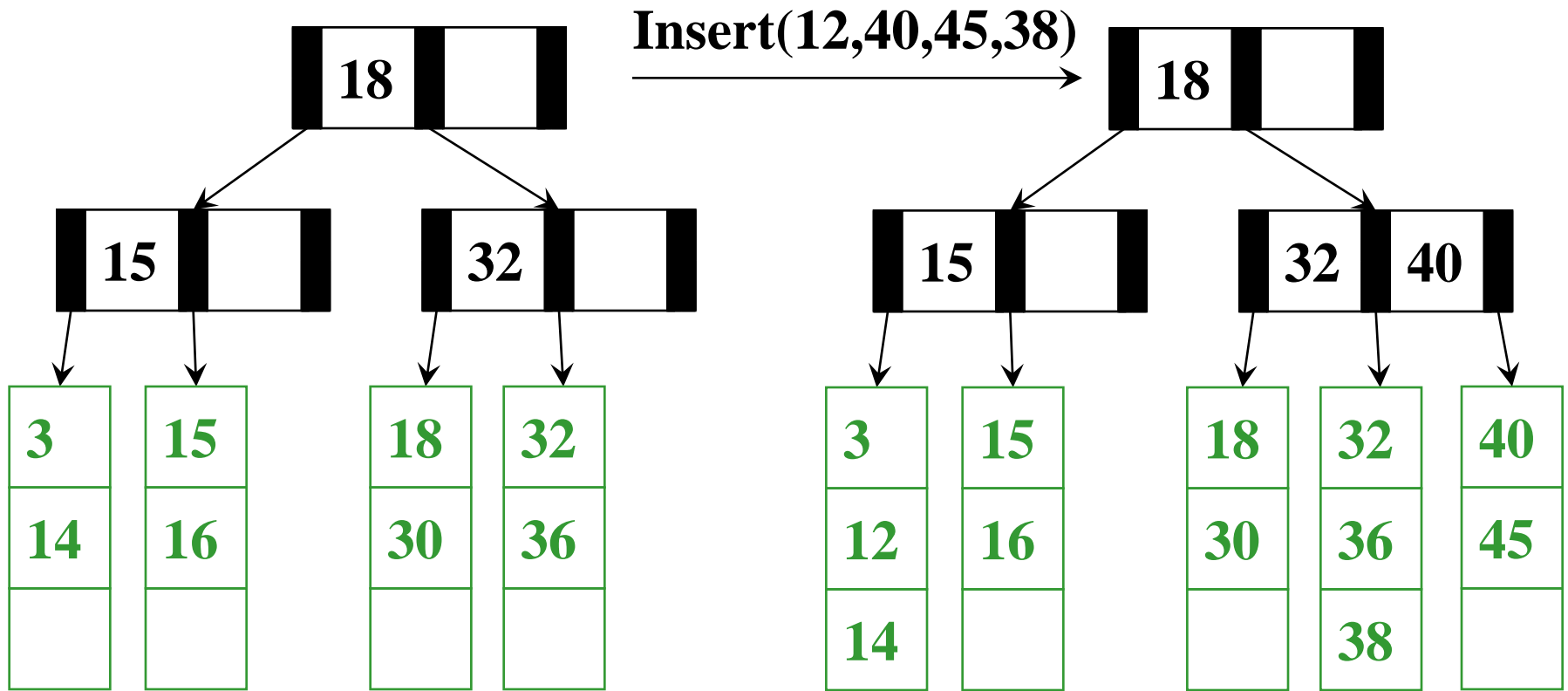
???



$M = 3$   $L = 3$

Split the internal node  
(in this case, the **root**)





$$M = 3 \quad L = 3$$

**Note:** Given the **leaves** and the structure of the tree, we can always fill in internal node keys; ‘the smallest value in my right branch’

# *Insertion Algorithm*

1. Insert the data in its **leaf** in sorted order
2. If the **leaf** now has  $L+1$  items, *overflow!*
  - Split the **leaf** into two nodes:
    - Original **leaf** with  $\lceil (L+1) / 2 \rceil$  smaller items
    - New **leaf** with  $\lfloor (L+1) / 2 \rfloor = \lceil L/2 \rceil$  larger items
  - Attach the new child to the parent
    - Adding new key to parent in sorted order
3. If Step 2 caused the parent to have  $M+1$  children, *overflow!*

# *Insertion Algorithm*

3. If an **internal node** has  $M+1$  children
  - Split the **node** into **two nodes**
    - Original **node** with  $\lceil (M+1) / 2 \rceil$  smaller items
    - New **node** with  $\lfloor (M+1) / 2 \rfloor = \lceil M/2 \rceil$  larger items
  - Attach the new child to the parent
    - Adding new key to parent in sorted order

Step 3 splitting could make the parent overflow too

- *So repeat step 3 up the tree until a node does not overflow*
- If the **root** overflows, make a new **root** with two children
  - This is the only case that increases the tree height

# *Worst-Case Efficiency of Insert*

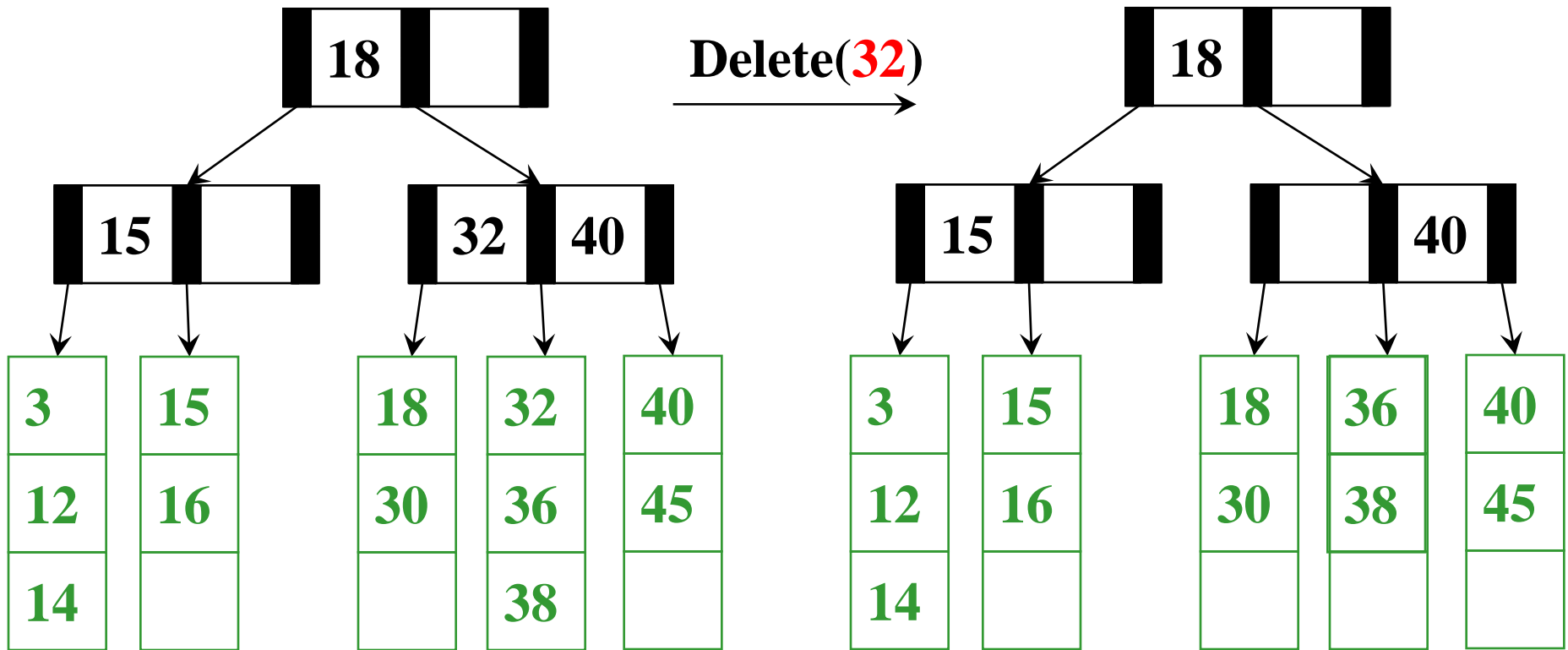
- Find correct leaf:  $O(\log_2 M \log_M n)$
- Insert in leaf:  $O(L)$
- Split leaf:  $O(L)$
- Split parents all the way up to root:  $O(M \log_M n)$

Total:  $O(L + M \log_M n)$

But it's not that bad:

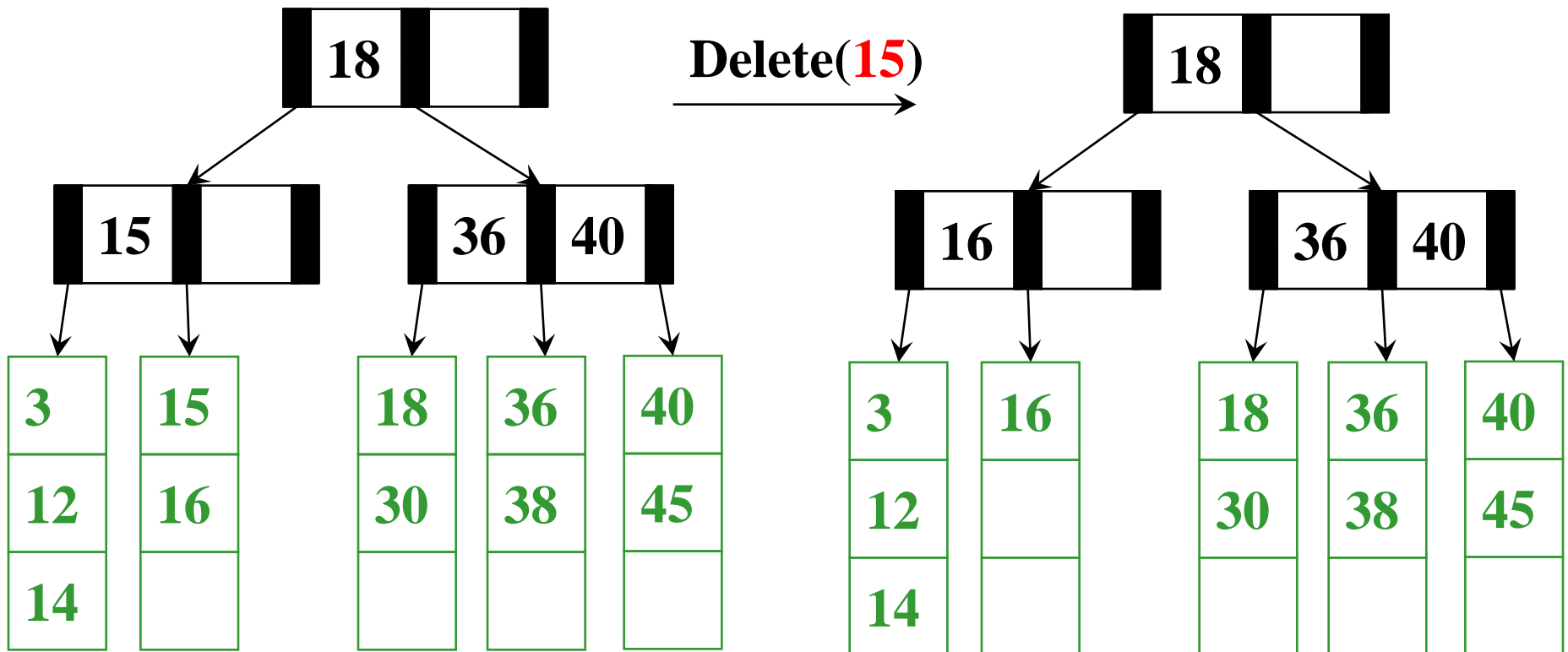
- Splits are not that common (only required when a node is FULL, M and L are likely to be large, and after a split will be half empty)
- Splitting the **root** is extremely rare
- Remember disk accesses is name of the game:  $O(\log_M n)$

# Deletion



$M = 3$   $L = 3$

Let them eat cake!



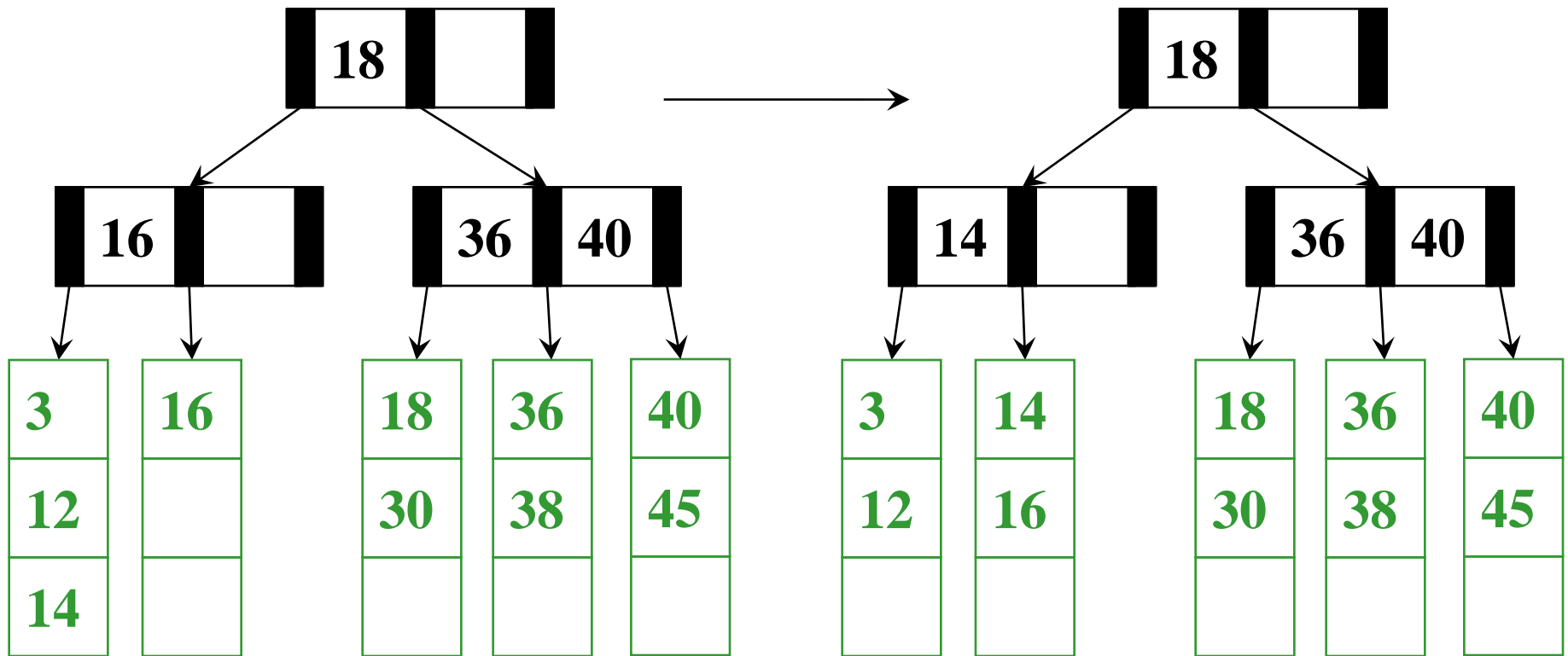
Are we okay?

Are you using that 14?  
Can I borrow it?

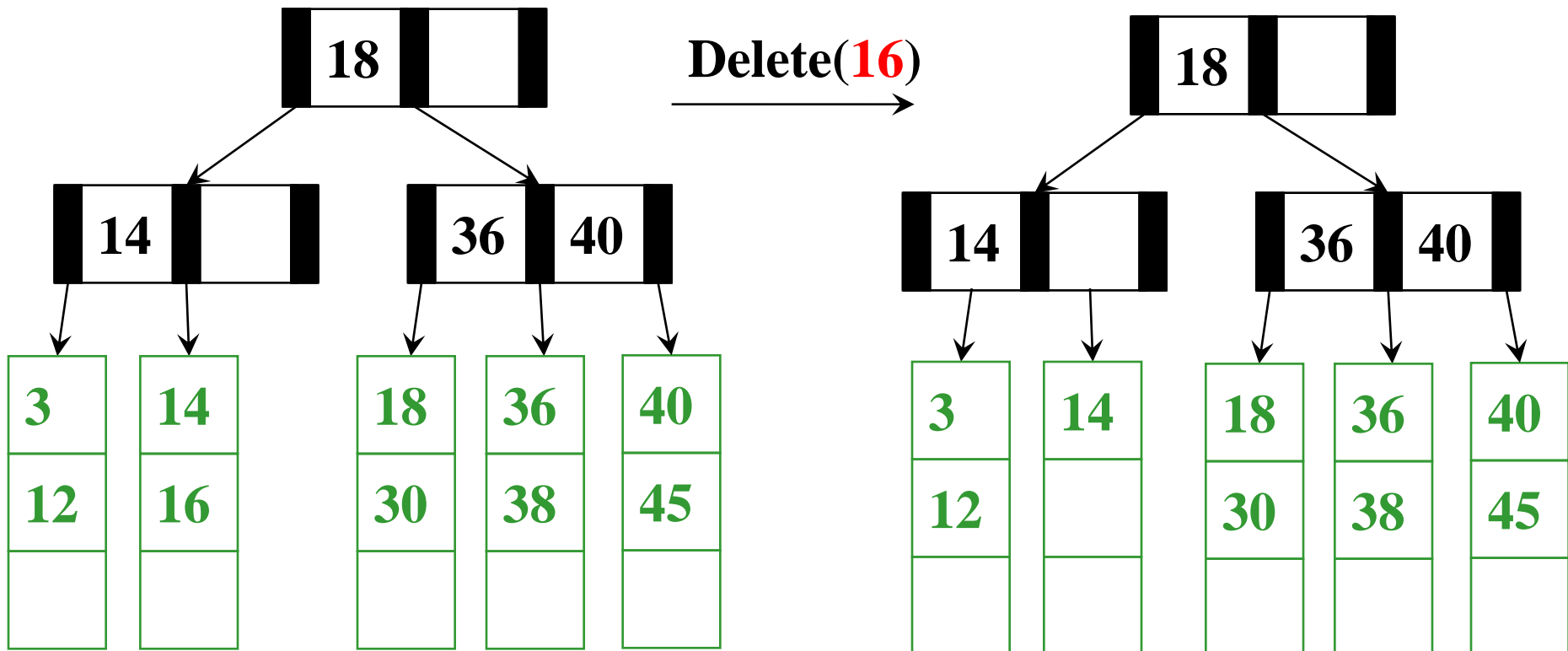
$M = 3$   $L = 3$

Dang, not half full





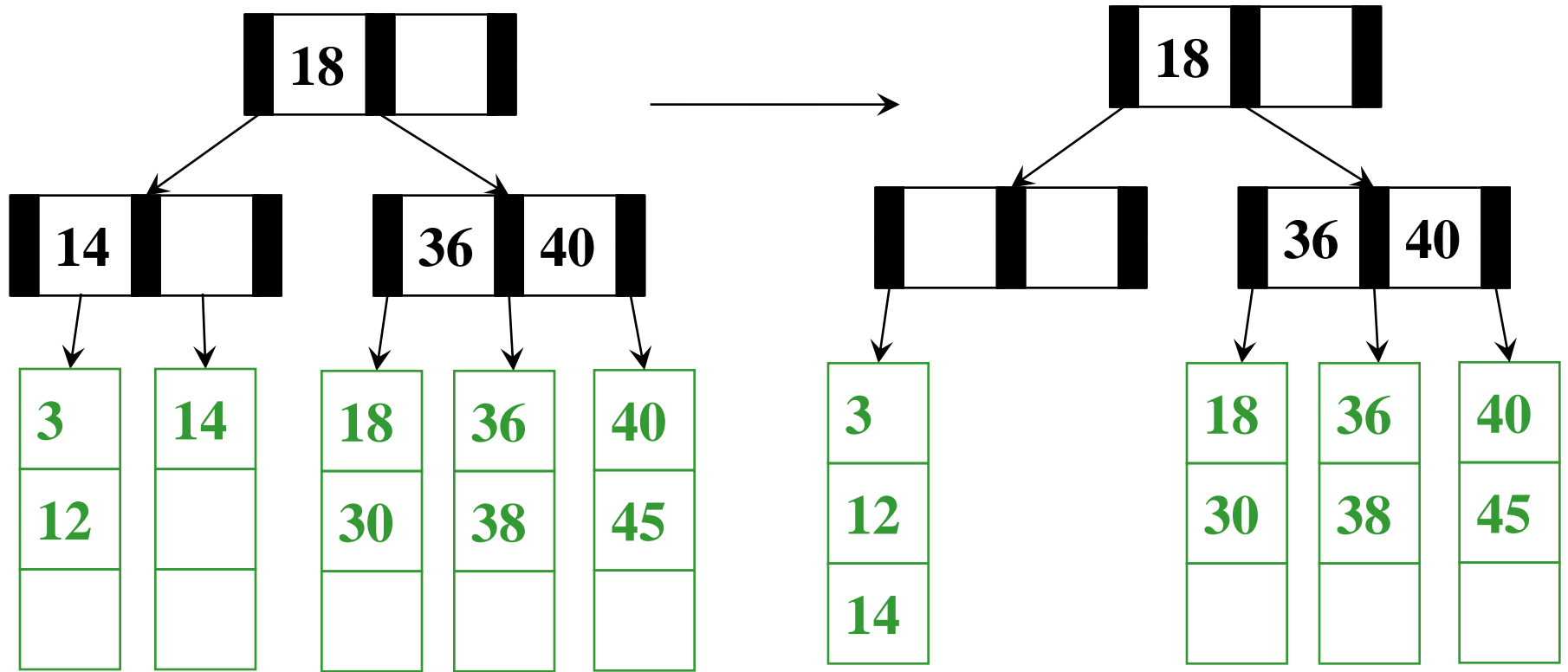
$M = 3$   $L = 3$



Are you using that 12?

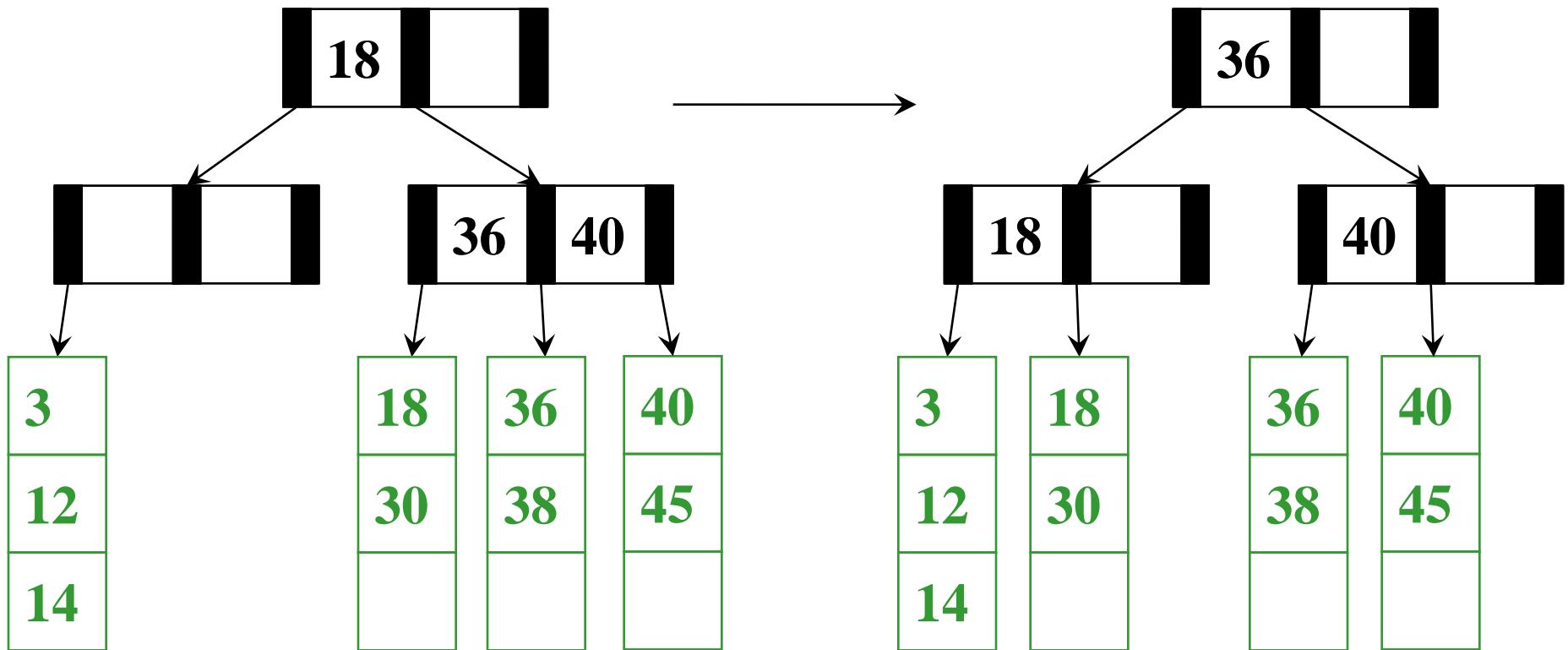
Are you using that 18?

$M = 3$   $L = 3$

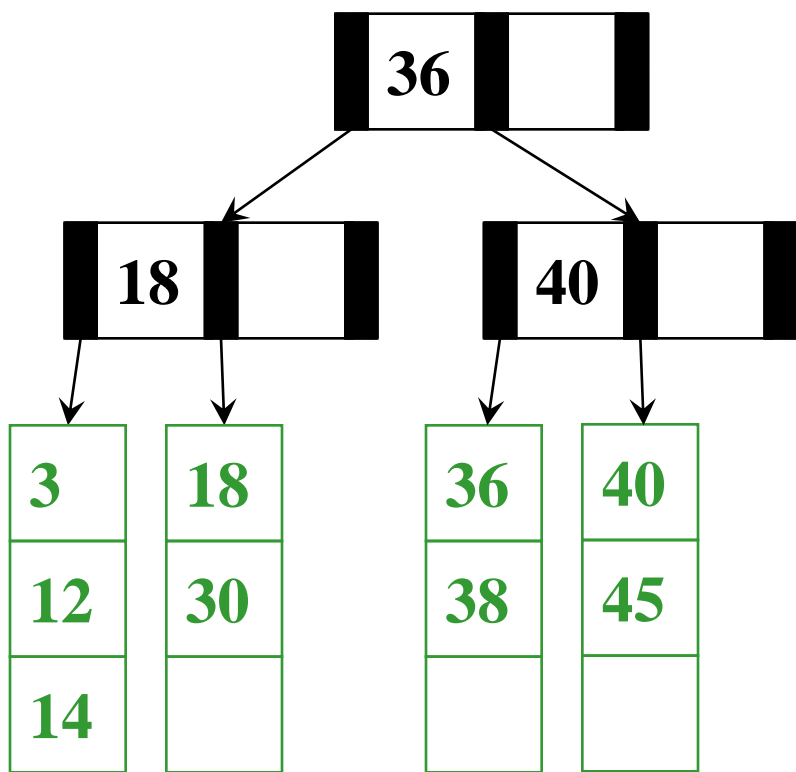


**Are you using that 18/30?**

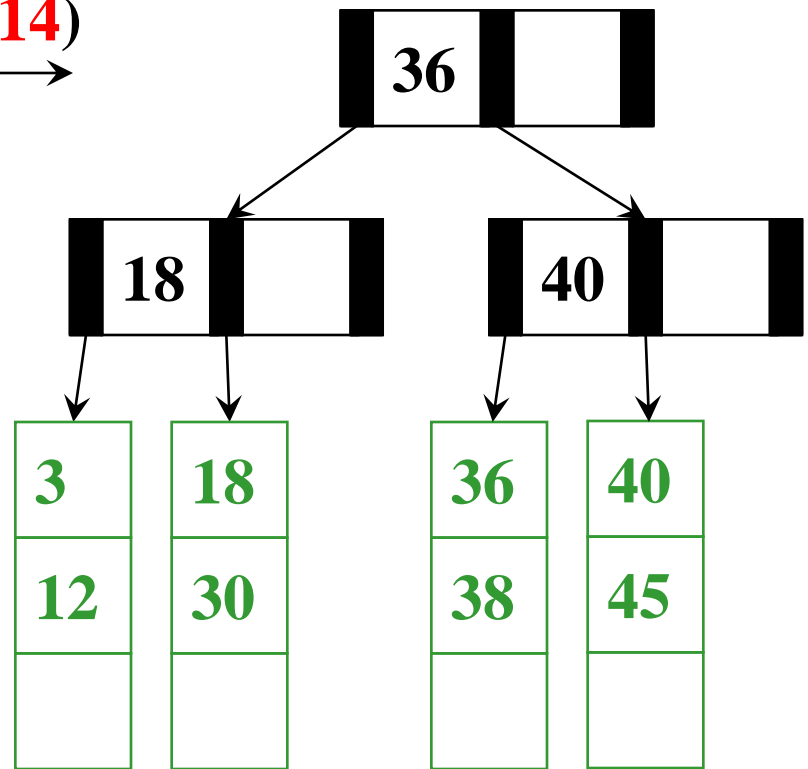
$M = 3 \quad L = 3$



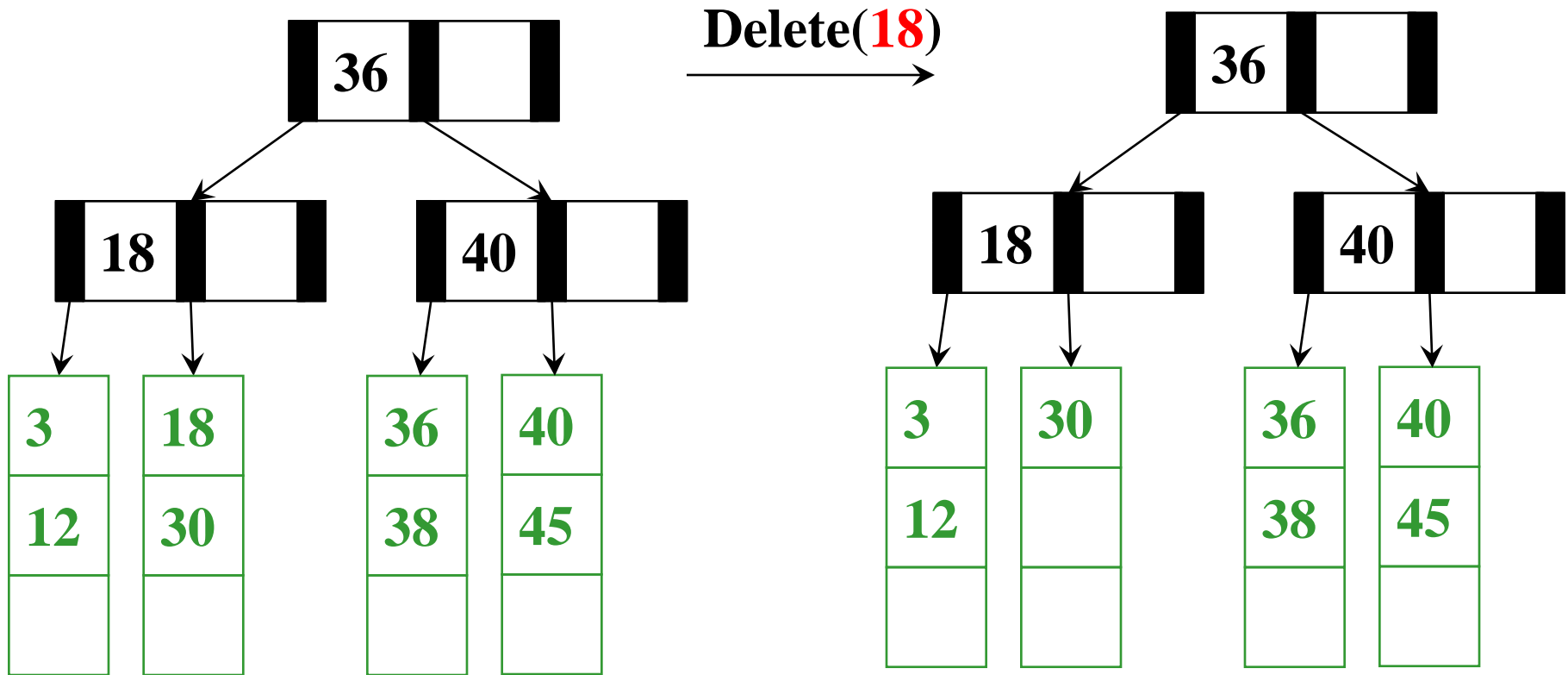
$M = 3$   $L = 3$



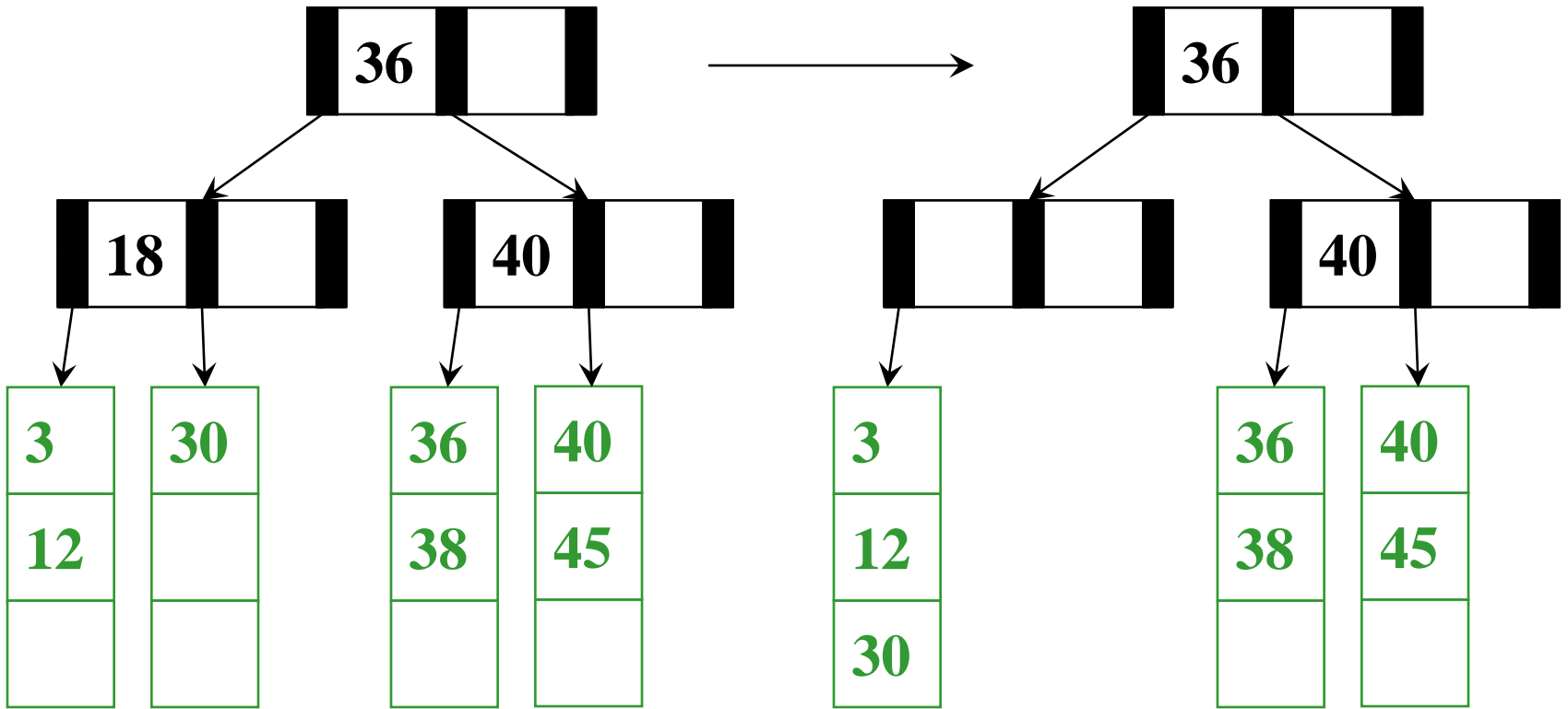
Delete(14) →



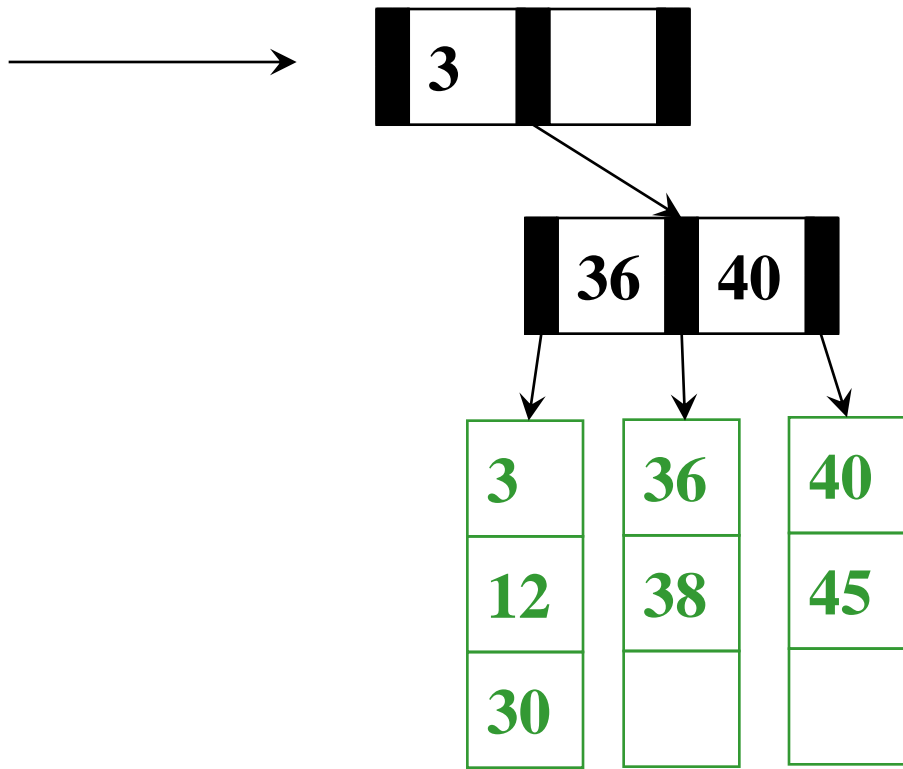
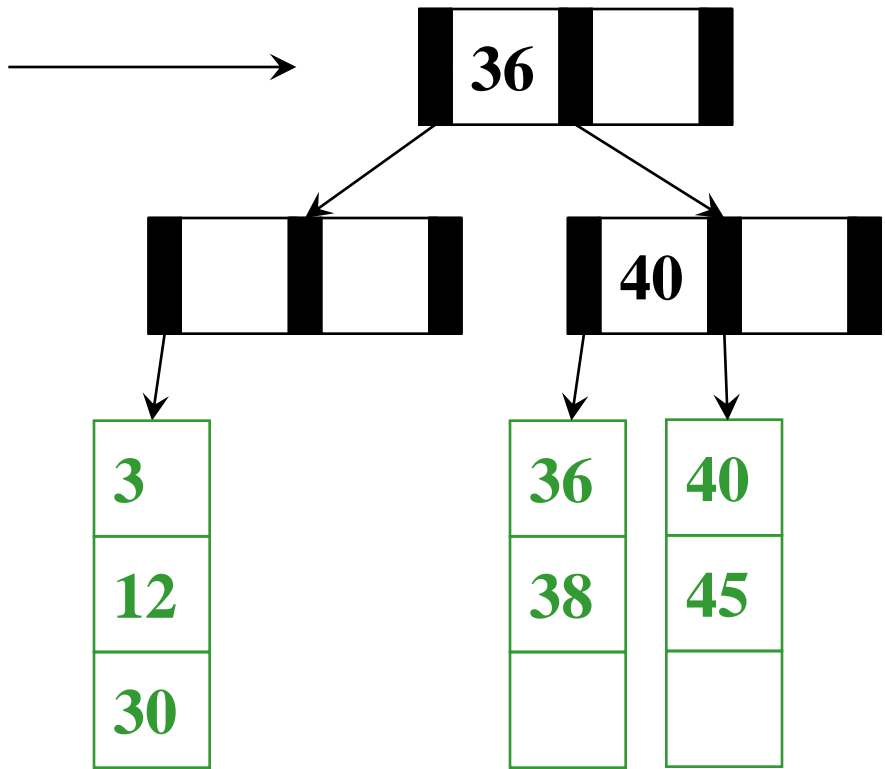
$M = 3$   $L = 3$



$M = 3$   $L = 3$

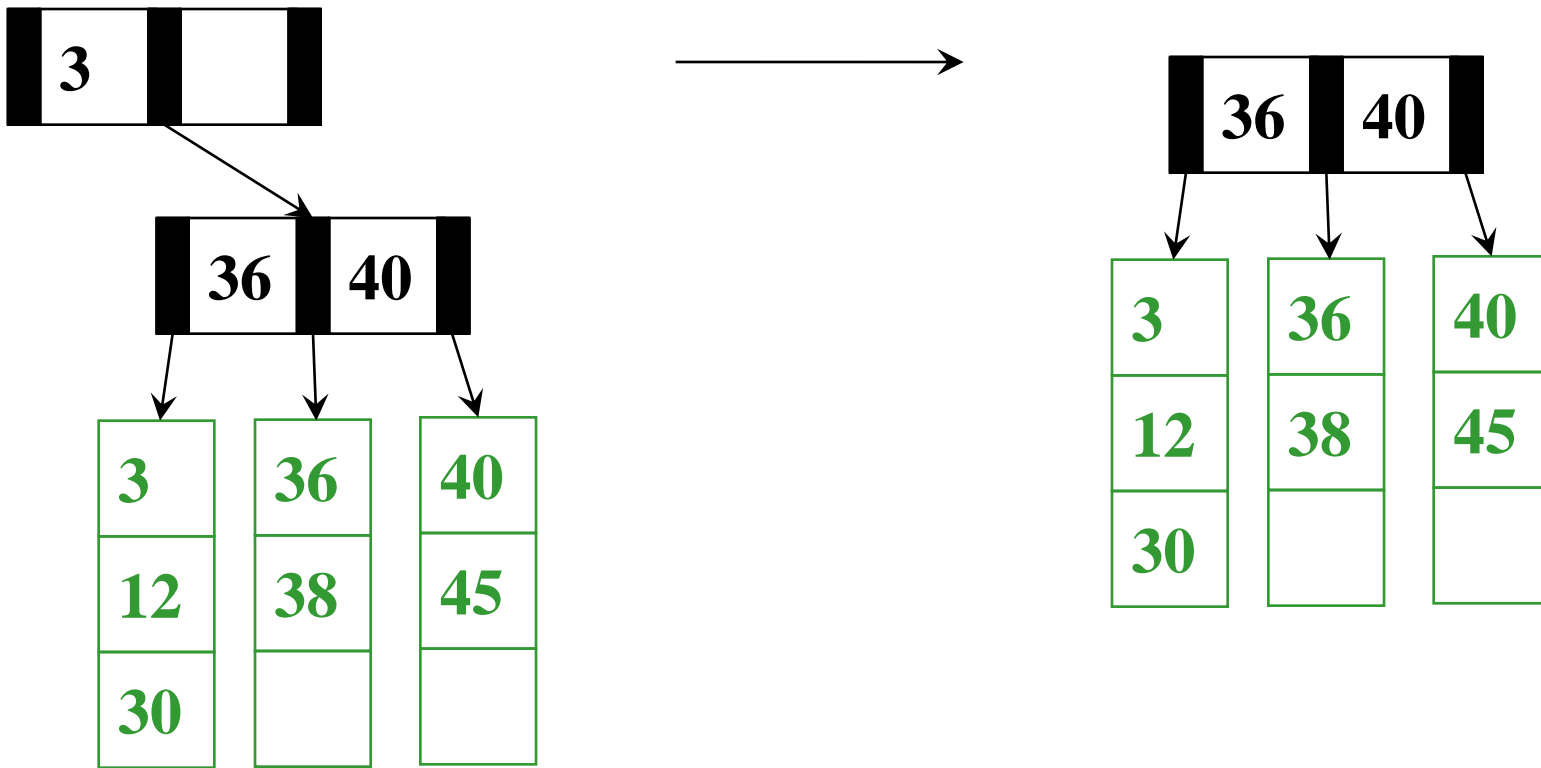


$M = 3$   $L = 3$



$M = 3 \quad L = 3$





$M = 3$   $L = 3$

# *Deletion Algorithm*

1. Remove the data from its leaf
2. If the leaf now has  $\lceil L/2 \rceil - 1$ , *underflow!*
  - If a neighbor has  $> \lceil L/2 \rceil$  items, *adopt* and update parent
  - Else *merge* node with neighbor
    - Guaranteed to have a legal number of items
    - Parent now has one less node
3. If Step 2 caused parent to have  $\lceil M/2 \rceil - 1$  children, *underflow!*

# *Deletion Algorithm*

3. If an internal node has  $\lceil M/2 \rceil - 1$  children
  - If a neighbor has  $> \lceil M/2 \rceil$  items, *adopt* and update parent
  - Else *merge* node with neighbor
    - Guaranteed to have a legal number of items
    - Parent now has one less node, may need to continue underflowing up the tree

Fine if we merge all the way up through the root

- Unless the root went from 2 children to 1
- In that case, delete the root and make child the root
- This is the only case that decreases tree height

# *Worst-Case Efficiency of Delete*

- Find correct leaf:  $O(\log_2 M \log_M n)$
- Remove from leaf:  $O(L)$
- Adopt from or merge with neighbor:  $O(L)$
- Adopt or merge all the way up to root:  $O(M \log_M n)$

Total:  $O(L + M \log_M n)$

But it's not that bad:

- Merges are not that common
- Remember disk access is the name of the game:  $O(\log_M n)$

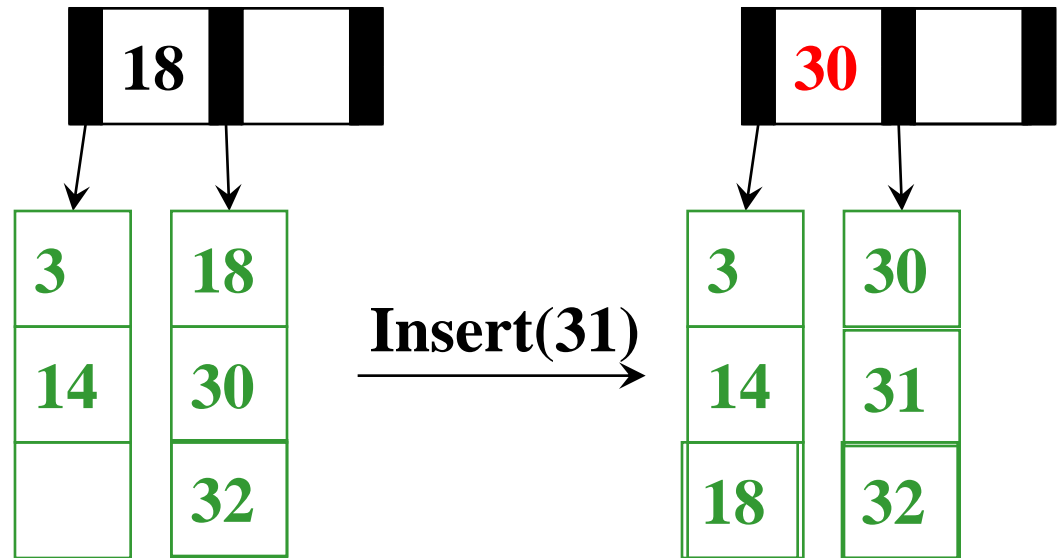
# Adoption for Insert

But can sometimes avoid splitting via *adoption*

- Change what leaf is correct by changing parent keys
- This is simply “borrowing” but “in reverse”
- Not necessary

Example:

**Adoption**



# *B Trees in Java?*

Remember you are learning deep concepts, not just trade skills

For most of our data structures, we have encouraged writing high-level and reusable code, as in Java with generics

It is worthwhile to know enough about “how Java works” and why this is probably a bad idea for B trees

- If you just want balance with worst-case logarithmic operations
  - No problem,  $M=3$  is a 2-3 tree,  $M=4$ , is a 2-3-4 tree
- Assuming our goal is efficient number of disk accesses
  - Java has many advantages, but it wasn't designed for this

The key issue is extra *levels of indirection...*

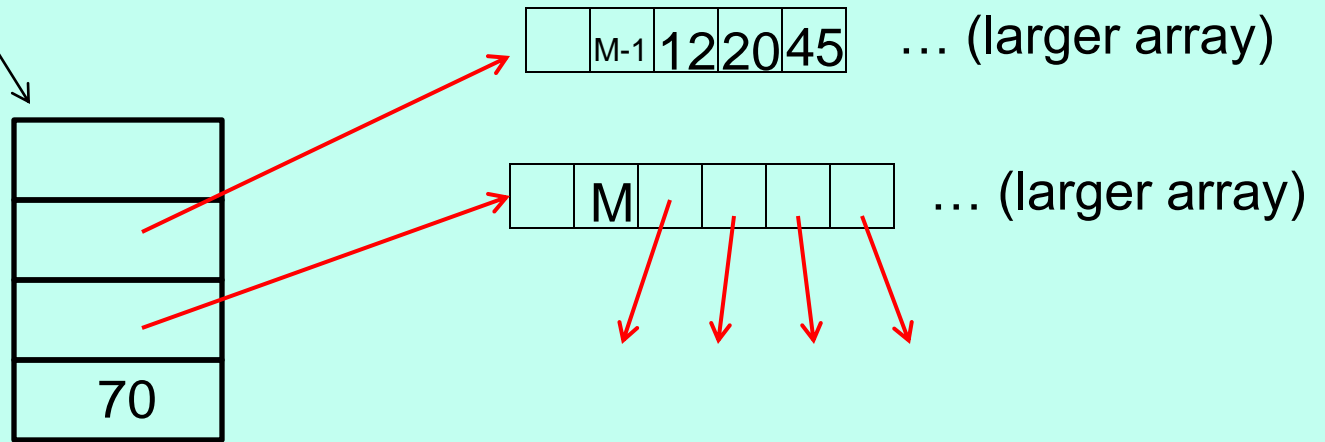
# Naïve Approach

Even if we assume data items have `int` keys, you cannot get the data representation you want for “really big data”

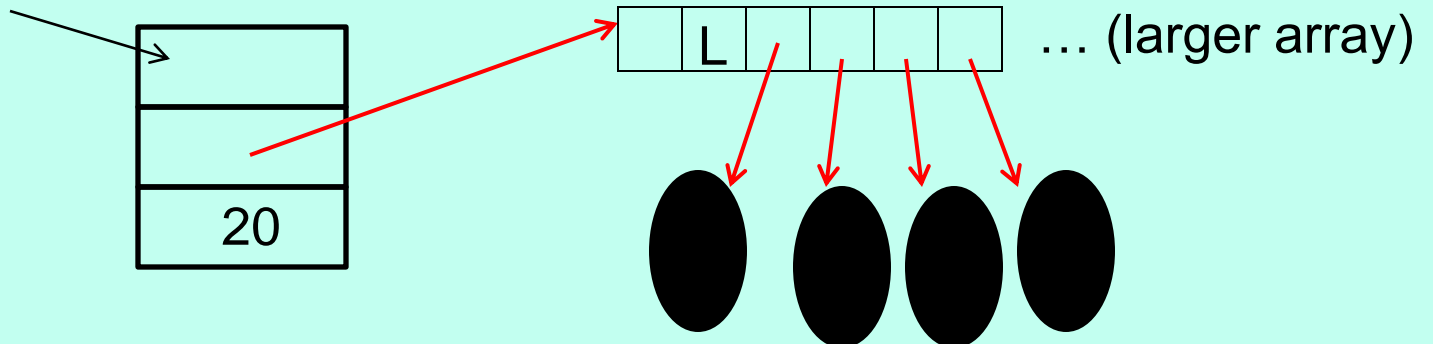
```
interface Keyed<E> {
    int key(E);
}
class BTreeNode<E implements Keyed<E>> {
    static final int M = 128;
    int[] keys = new int[M-1];
    BTreeNode<E>[] children = new BTreeNode[M];
    int numChildren = 0;
    ...
}
class BTreeLeaf<E> {
    static final int L = 32;
    E[] data = (E[])new Object[L];
    int numItems = 0;
    ...
}
```

# What that looks like

## BTreeNode (3 objects with “header words”)



## BTreeLeaf (data objects not in contiguous memory)





## *The moral*

- The point of B trees is to keep related data in contiguous memory
- All the red references on the previous slide are inappropriate
  - As minor point, beware the extra “header words”
- But that is “the best you can do” in Java
  - Again, the advantage is generic, reusable code
  - But for your performance-critical web-index, not the way to implement your B-Tree for terabytes of data
- Other languages better support “flattening objects into arrays”
- Levels of indirection matter!

# *Conclusion: Balanced Trees*

- *Balanced* trees make good dictionaries because they guarantee logarithmic-time **find**, **insert**, and **delete**
  - Essential and beautiful computer science
  - But only if you can maintain balance within the time bound
- **AVL trees** maintain balance by tracking height and allowing all children to differ in height by at most 1
- **B trees** maintain balance by keeping nodes at least half full and all leaves at same height
- Other great balanced trees (see text; worth knowing they exist)
  - **Red-black trees**: all leaves have depth within a factor of 2
  - **Splay trees**: self-adjusting; amortized guarantee;  
no extra space for height information



# CSE332: Data Abstractions

## Lecture 8: Hashing

James Fogarty

Winter 2012

# *Conclusion of Balanced Trees*

- Balanced trees make good dictionaries because they guarantee logarithmic-time **find**, **insert**, and **delete**
  - Essential and beautiful computer science
  - But only if you can maintain balance within the time bound
- **AVL trees** maintain balance by tracking height and allowing all children to differ in height by at most 1
- **B trees** maintain balance by keeping nodes at least half full and all leaves at same height
- Other great balanced trees (see text; worth knowing they exist)
  - **Red-black trees**: all leaves have depth within a factor of 2
  - **Splay trees**: self-adjusting; amortized guarantee; no extra space for height information

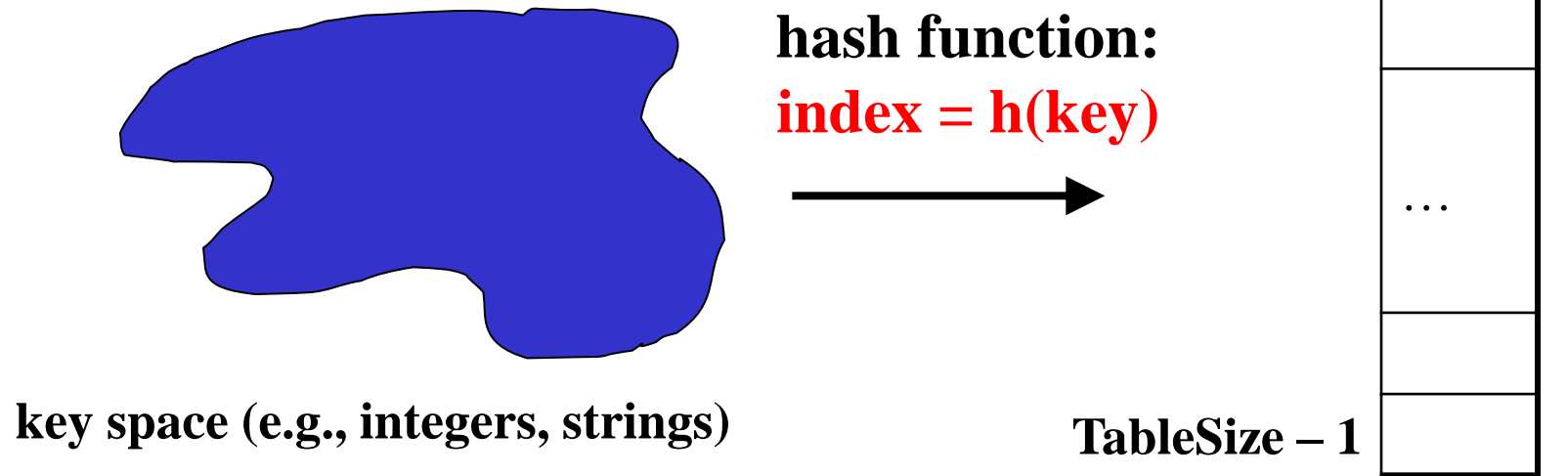
# Simple Implementations

For dictionary with  $n$  key/value pairs

	<b>insert</b>	<b>find</b>	<b>delete</b>	
• Unsorted linked-list	$O(1)$	$O(n)$	$O(n)$	
• Unsorted array	$O(1)$	$O(n)$	$O(n)$	
• Sorted linked list	$O(n)$	$O(n)$	$O(n)$	
• Sorted array	$O(n)$	$O(\log n)$	$O(n)$	
• Balanced tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	
• Magic array	$O(1)$	$O(1)$	$O(1)$	average case

# Hash Tables

- Aim for constant-time **find**, **insert**, and **delete**
  - “On average” under some reasonable **assumptions**
- A hash table is an array of some fixed size
- Basic idea:



# Hash Tables vs. Balanced Trees

- In terms of a Dictionary ADT for just **insert**, **find**, **delete**, hash tables and balanced trees are just different data structures
  - Hash tables  $O(1)$  on average (*assuming* few collisions)
  - Balanced trees  $O(\log n)$  worst-case
- Constant-time is better, right?
  - Yes, but you need “hashing to behave” (must avoid collisions)
  - Yes, but **findMin**, **findMax**, **predecessor**, **successor** go from  $O(\log n)$  to  $O(n)$ , **printSorted** from  $O(n)$  to  $O(n \log n)$
- **Moral:** If you need to frequently use operations based on sort order, then you may prefer a balanced BST instead.

# *Hash Tables*

- There are  $m$  possible keys ( $m$  typically large, even infinite)
- We expect our table to have only  $n$  items
- $n$  is much less than  $m$  (often written  $n \ll m$ )

Many dictionaries have this property

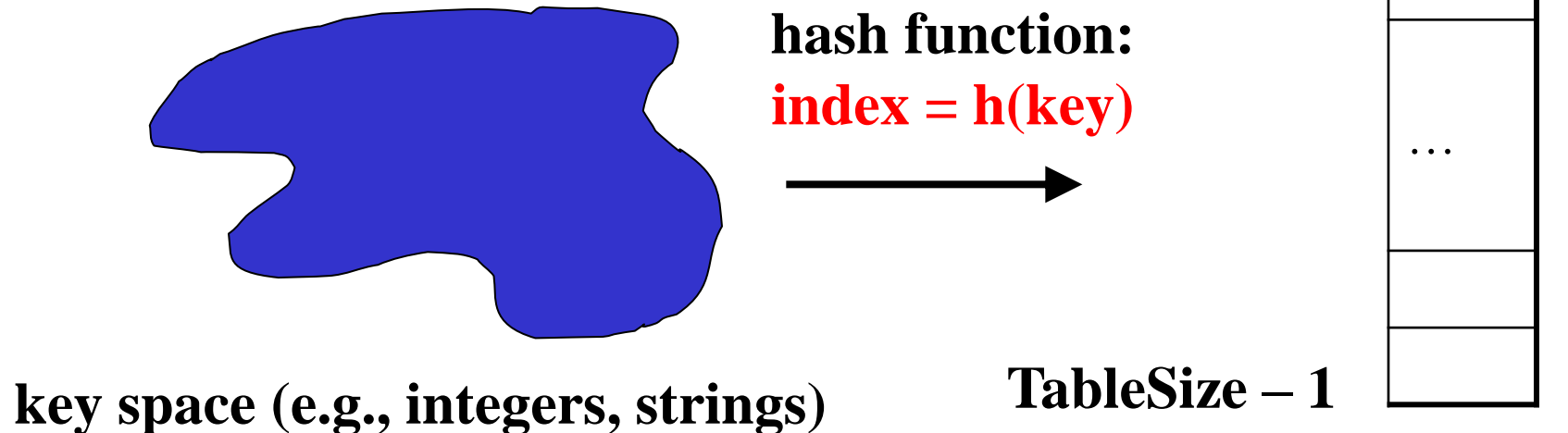
- Compiler: All possible identifiers allowed by the language vs. those used in some file of one program
- Database: All possible student names vs. students enrolled
- AI: All possible chess-board configurations vs. those considered by the current player



# Hash Functions

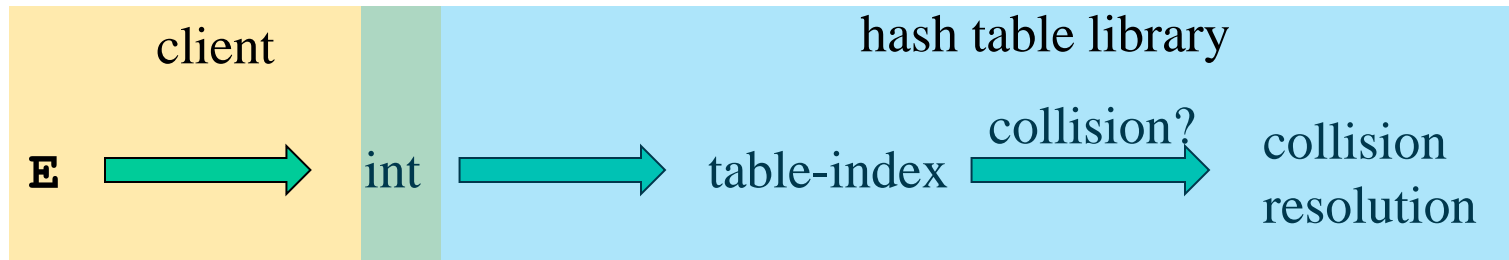
An ideal hash function:

- Is fast to compute
- “Rarely” hashes two “used” keys to the same index
  - Often impossible in theory; easy in practice
  - Will handle *collisions* in later



# Who Hashes What

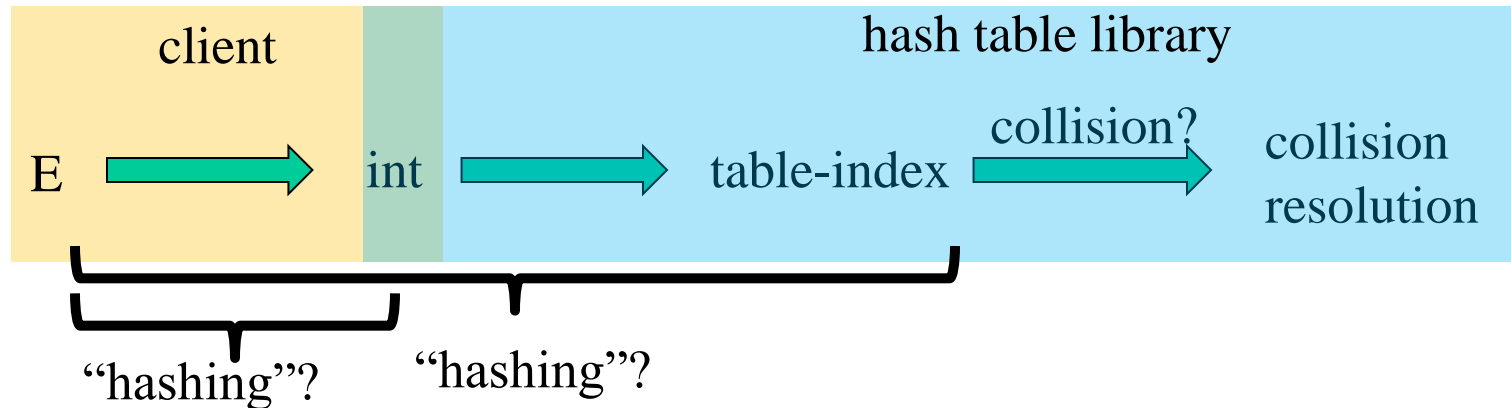
- Hash tables can be generic
  - To store elements of type  $\mathbf{E}$ , we just need  $\mathbf{E}$  to be:
    1. Comparable: order any two  $\mathbf{E}$  (as with all dictionaries)
    2. Hashable: convert any  $\mathbf{E}$  to an `int`
- When hash tables are a reusable library, the division of responsibility generally breaks down into two roles:



- We will learn both roles, but most programmers “in the real world” spend more time as clients while understanding the library

# More on Roles

Some ambiguity in terminology on which parts are “hashing”



Two roles must both contribute to minimizing collisions (heuristically)

- Client should aim for different ints for expected items
  - Avoid “wasting” any part of **E** or the 32 bits of the **int**
- Library should aim for putting “similar” **ints** in different indices
  - conversion to index is almost always “mod table-size”
  - using prime numbers for table-size is common

# *What to Hash?*

We will focus on two most common things to hash: ints and strings

- If you have objects with several fields, it is usually best to hash most of the “identifying fields” to avoid collisions
- Example:

```
class Person {  
    String first; String middle; String last;  
    Date birthdate;  
}
```

- An inherent trade-off: hashing-time vs. collision-avoidance

# Hashing Integers

- key space = integers
- Simple hash function:  
$$h(\text{key}) = \text{key} \% \text{TableSize}$$
  - Client:  $f(x) = x$
  - Library  $g(x) = f(x) \% \text{TableSize}$
  - Fairly fast and natural
- Example:
  - TableSize = 10
  - Insert 7, 18, 41, 34, 10
  - (As usual, ignoring corresponding data)

0	10
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

# *Collision Avoidance*

- With “**x % TableSize**” the number of collisions depends on
  - the ints inserted
  - **TableSize**
- Larger table-size tends to help, but not always
  - Example: 70, 24, 56, 43, 10  
with **TableSize = 10** and **TableSize = 60**
- Technique: Pick table size to be prime. Why?
  - Real-life data tends to have a pattern,
  - “Multiples of 61” are probably less likely than “multiples of 60”
  - We will see some collision strategies do better with prime size

# *More Arguments for a Prime Size*

If **TableSize** is 60 and...

- Lots of data items are multiples of 2, wasting 50% of table
- Lots of data items are multiples of 5, wasting 80% of table
- Lots of data items are multiples of 10, wasting 90% of table

If **TableSize** is 61...

- Collisions can still happen but 2, 4, 6, 8, ... will fill table
- Collisions can still happen, but 5, 10, 15, 20, ... will fill table
- Collisions can still happen but 10, 20, 30, 40, ... will fill table

In general, if **x** and **y** are “co-prime” (means  $\text{gcd}(\mathbf{x}, \mathbf{y}) == 1$ ),  
then  $(\mathbf{a} * \mathbf{x}) \% \mathbf{y} == (\mathbf{b} * \mathbf{x}) \% \mathbf{y}$  if and only if  $\mathbf{a} \% \mathbf{y} == \mathbf{b} \% \mathbf{y}$

- Good to have a **TableSize** that has  
no common factors with any “likely pattern” of **x**

# What if *key* is not an *int*?

- If keys are not *ints*, the client must convert to an *int*
  - Trade-off: speed and distinct keys hashing to distinct *ints*
- Common and important example: Strings
  - Key space  $K = s_0s_1s_2 \dots s_{m-1}$ 
    - where  $s_i$  are chars:  $s_i \in [0,256]$
  - Some choices: Which best avoid collisions?

1.  $h(K) = s_0 \% \text{TableSize}$

2.  $h(K) = \left( \sum_{i=0}^{m-1} s_i \right) \% \text{TableSize}$

3.  $h(K) = \left( \sum_{i=0}^{k-1} s_i \cdot 37^i \right) \% \text{TableSize}$



# *Combining Hash Functions*

A few rules of thumb / tricks:

1. Use all 32 bits (careful, that includes negative numbers)
2. Use different overlapping bits for different parts of the hash
  - This is why a factor of  $37^i$  works better than  $256^i$
  - Example: “abcde” and “ebcda”
3. When smashing two hashes into one hash, use bitwise-xor
  - bitwise-and produces too many 0 bits
  - bitwise-or produces too many 1 bits
4. Rely on expertise of others; consult books and other resources
5. Advanced: If keys are known ahead of time, a *perfect hash*

# *Collision Resolution*

## Collision:

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

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

So hash tables generally need to support [collision resolution](#)

# Separate Chaining

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

Chaining:

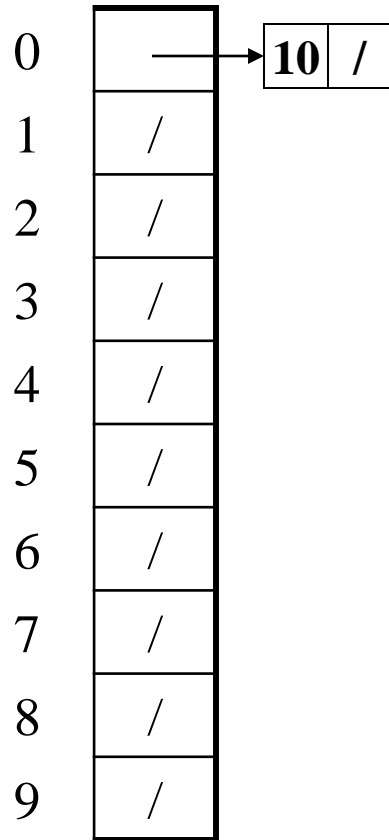
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

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

# Separate Chaining



Chaining:

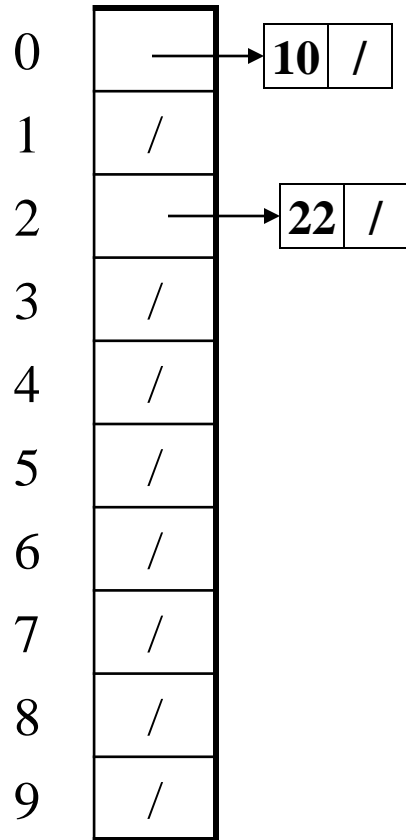
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

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

# Separate Chaining



Chaining:

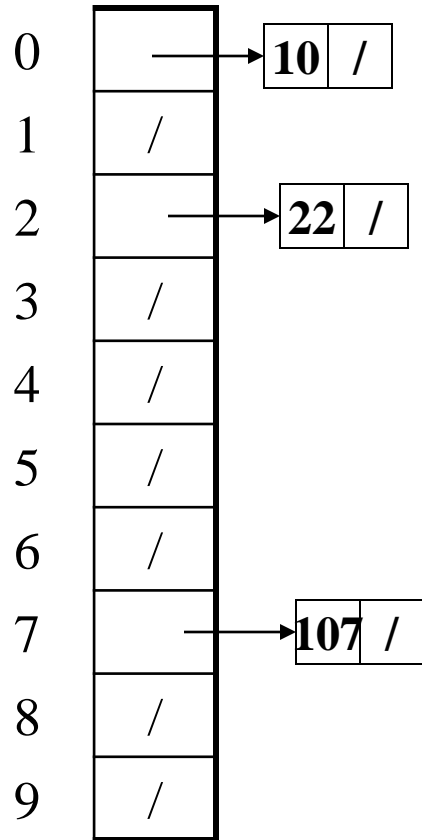
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

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

# Separate Chaining



Chaining:

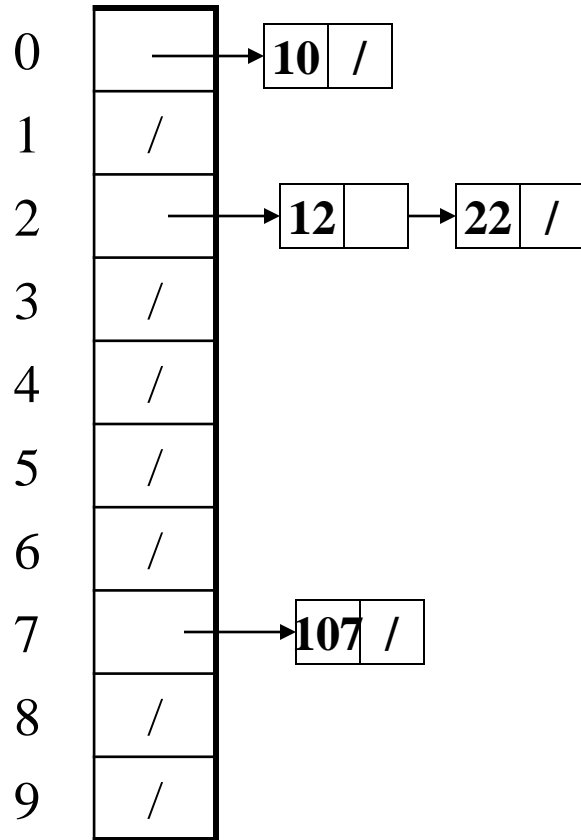
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

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

# Separate Chaining



Chaining:

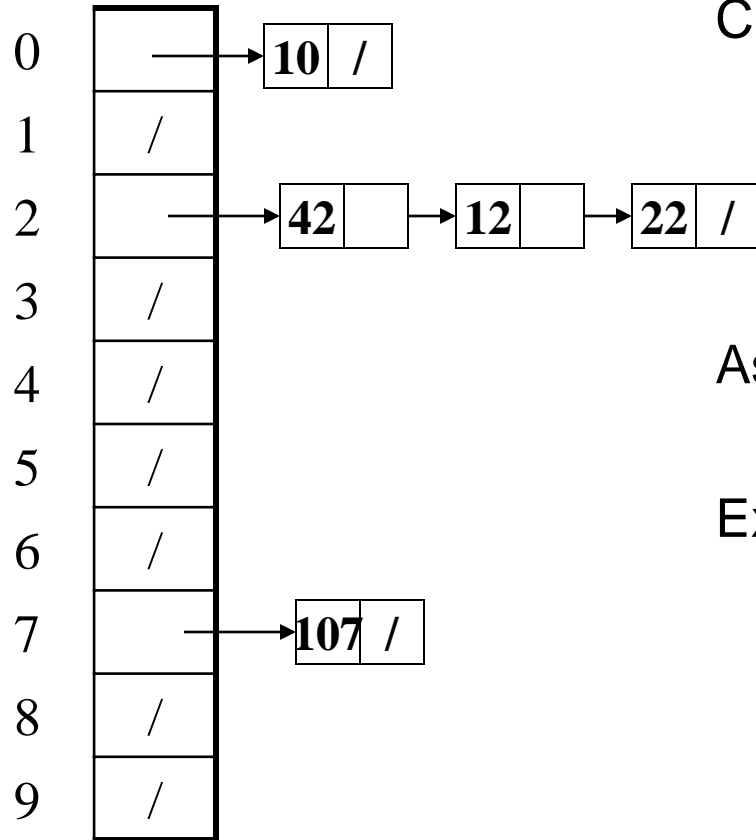
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

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

# Separate Chaining



Chaining:

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

As easy as it sounds

Example:

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



# *Thoughts on Separate Chaining*

- Worst-case time for `find`?
  - Linear
  - But only with really bad luck or bad hash function
  - So not worth avoiding (e.g., with balanced trees at each bucket)
    - Keep small number of items in each bucket
    - Overhead of tree balancing not worthwhile for small  $n$
- Beyond asymptotic complexity, some “data-structure engineering”
  - Linked list, array, or a hybrid
  - Move-to-front list (as in Project 2)
  - Leave one element in the table itself, to optimize constant factors for the common case

# More Rigorous Separate Chaining Analysis

Definition: The **load factor**,  $\lambda$ , of a hash table is

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

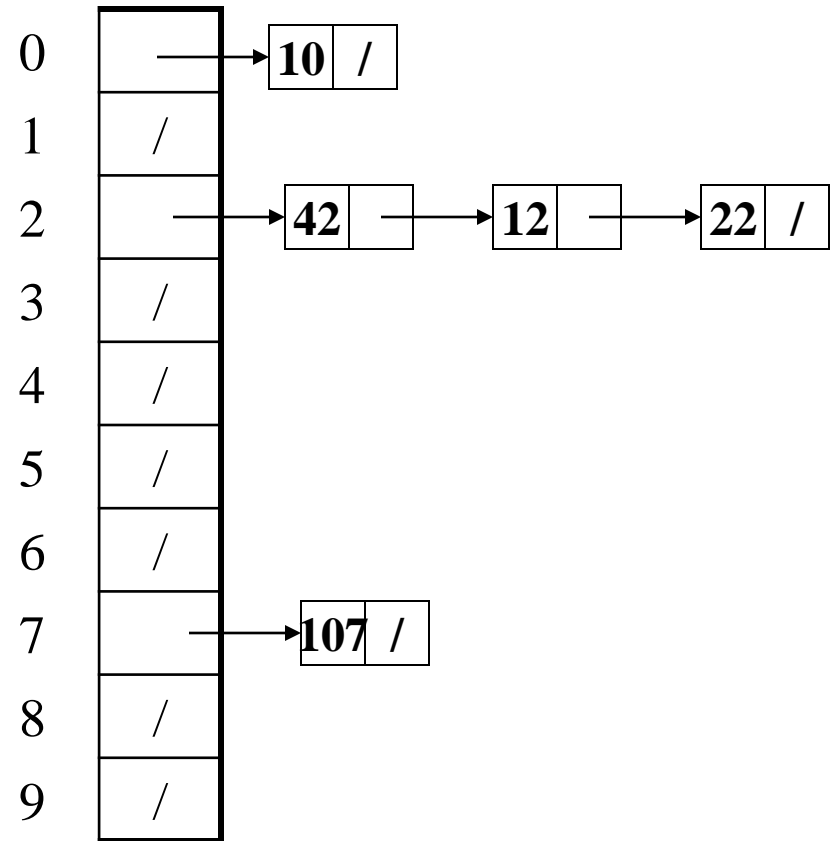
Under chaining, the average number of elements per bucket is  $\lambda$

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

- Each unsuccessful **find** compares against  $\lambda$  items
- Each successful **find** compares against  $\lambda/2$  items
- If  $\lambda$  is low, find & insert likely to be  $O(1)$
- We like to keep  $\lambda$  around 1 for separate chaining

# Separate Chaining Deletion

- Not too bad
  - Find in table
  - Delete from bucket
- Delete 12
- Similar run-time as insert





# CSE332: Data Abstractions

## Lecture 9: Hashing

James Fogarty

Winter 2012

# Open Addressing: Linear Probing

- Why not use up the empty space in the table?
- Store directly in the array cell (no linked list)
- How to deal with collisions?
- If  $h(\text{key})$  is already full,
  - try  $(h(\text{key}) + 1) \% \text{TableSize}$ . If full,
  - try  $(h(\text{key}) + 2) \% \text{TableSize}$ . If full,
  - try  $(h(\text{key}) + 3) \% \text{TableSize}$ . If full...
- Example: insert 38, 19, 8, 109, 10

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

# Open Addressing: Linear Probing

- Why not use up the empty space in the table?
- Store directly in the array cell (no linked list)
- How to deal with collisions?
- If  $h(\text{key})$  is already full,
  - try  $(h(\text{key}) + 1) \% \text{TableSize}$ . If full,
  - try  $(h(\text{key}) + 2) \% \text{TableSize}$ . If full,
  - try  $(h(\text{key}) + 3) \% \text{TableSize}$ . If full...
- Example: insert 38, 19, 8, 109, 10

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

# Open Addressing: Linear Probing

- Why not use up the empty space in the table?
- Store directly in the array cell (no linked list)
- How to deal with collisions?
- If  $h(\text{key})$  is already full,
  - try  $(h(\text{key}) + 1) \% \text{TableSize}$ . If full,
  - try  $(h(\text{key}) + 2) \% \text{TableSize}$ . If full,
  - try  $(h(\text{key}) + 3) \% \text{TableSize}$ . If full...
- Example: insert 38, 19, 8, 109, 10

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

# Open Addressing: Linear Probing

- Why not use up the empty space in the table?
- Store directly in the array cell (no linked list)
- How to deal with collisions?
- If  $h(\text{key})$  is already full,
  - try  $(h(\text{key}) + 1) \% \text{TableSize}$ . If full,
  - try  $(h(\text{key}) + 2) \% \text{TableSize}$ . If full,
  - try  $(h(\text{key}) + 3) \% \text{TableSize}$ . If full...
- Example: insert 38, 19, 8, 109, 10

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



# Open Addressing: Linear Probing

- Why not use up the empty space in the table?
- Store directly in the array cell (no linked list)
- How to deal with collisions?
- If  $h(\text{key})$  is already full,
  - try  $(h(\text{key}) + 1) \% \text{TableSize}$ . If full,
  - try  $(h(\text{key}) + 2) \% \text{TableSize}$ . If full,
  - try  $(h(\text{key}) + 3) \% \text{TableSize}$ . If full...
- Example: insert 38, 19, 8, 109, 10

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

# Open Addressing: Linear Probing

- Why not use up the empty space in the table?
- Store directly in the array cell (no linked list)
- How to deal with collisions?
- If  $h(\text{key})$  is already full,
  - try  $(h(\text{key}) + 1) \% \text{TableSize}$ . If full,
  - try  $(h(\text{key}) + 2) \% \text{TableSize}$ . If full,
  - try  $(h(\text{key}) + 3) \% \text{TableSize}$ . If full...
- Example: insert 38, 19, 8, 109, 10

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

# Open Addressing

This is *one example* of open addressing

In general, **open addressing** means resolving collisions by trying a sequence of other positions in the table

Trying the next spot is called **probing**

- We just did **linear probing**

$$h(\text{key}) + i) \% \text{TableSize}$$

- In general have some **probe function  $f$**  and use

$$h(\text{key}) + f(i) \% \text{TableSize}$$

Open addressing does poorly with high load factor  $\lambda$

- So we want larger tables
- Too many probes means we lose our  $O(1)$

# *Terminology*

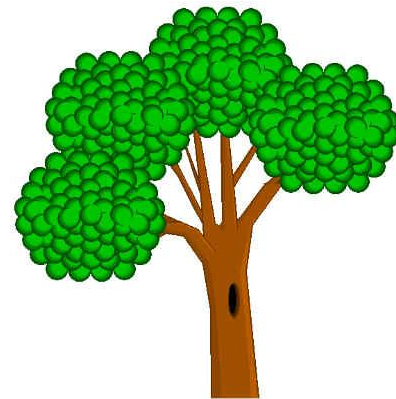
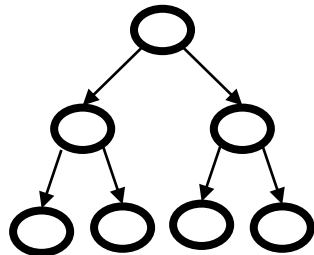
We and the book use the terms

- “chaining” or “separate chaining”
- “open addressing”

Very confusingly,

- “open hashing” is a synonym for “chaining”
- “closed hashing” is a synonym for “open addressing”

We also do trees upside-down



# Other Operations

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

What about **find**?

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

What about **delete**?

- **Must** use “lazy” deletion. Why?
- Marker indicates “no data here, but don’t stop probing”

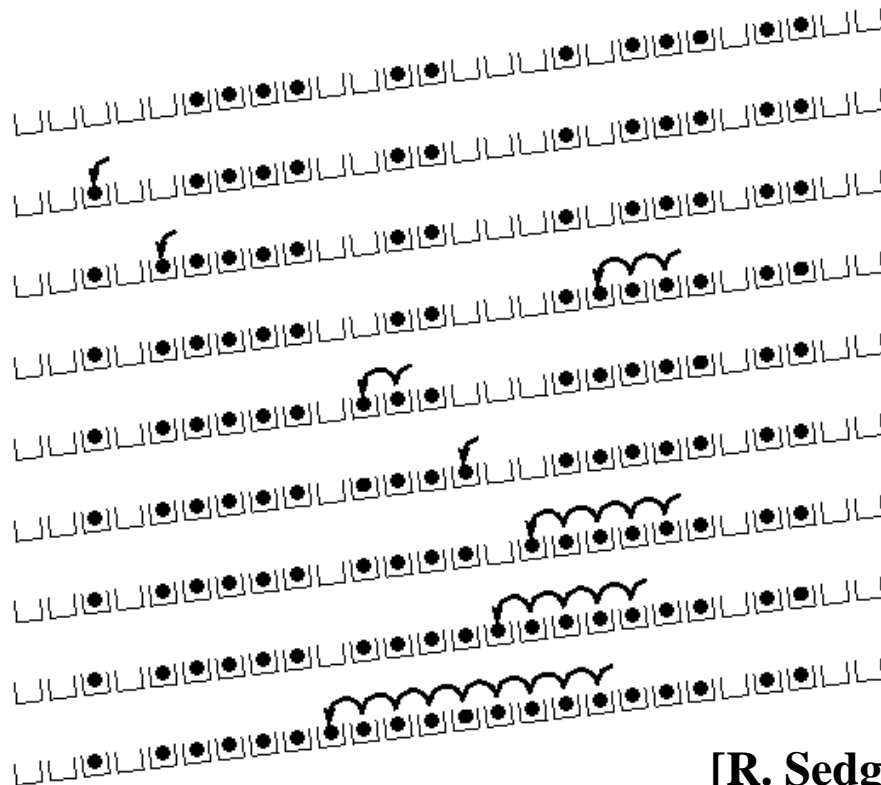
10	x	/	23	/	/	16	x	26
----	---	---	----	---	---	----	---	----

# Primary Clustering

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

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

- Called **primary clustering**
- Saw this starting in our example



[R. Sedgewick]

# *Analysis of Linear Probing*

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

- Non-trivial facts we won't prove:

Average # of probes given  $\lambda$  (in the limit as **TableSize**  $\rightarrow \infty$ )

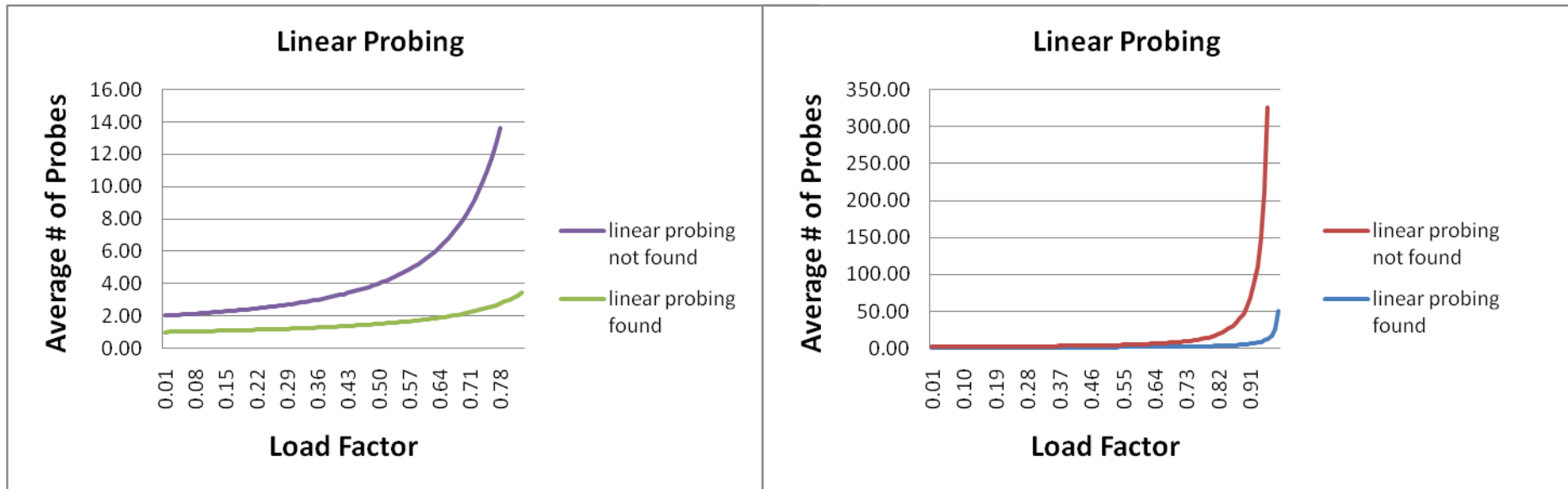
- Unsuccessful search: 
$$\frac{1}{2} \left( 1 + \frac{1}{(1-\lambda)^2} \right)$$

- Successful search: 
$$\frac{1}{2} \left( 1 + \frac{1}{(1-\lambda)} \right)$$

- This is pretty bad: need to leave sufficient empty space in the table to get decent performance (let's look at a chart)

# Analysis in Chart Form

- Linear-probing performance degrades rapidly as table gets full
  - Formula assumes “large table” but point remains



- Chaining performance was linear in  $\lambda$  and has no trouble with  $\lambda > 1$



# Open Addressing: Quadratic Probing

- We can avoid primary clustering by changing the probe function

$$(h(\text{key}) + f(i)) \% \text{TableSize}$$

- For quadratic probing:

$$f(i) = i^2$$

- So probe sequence is:

- 0<sup>th</sup> probe:  $h(\text{key}) \% \text{TableSize}$
- 1<sup>st</sup> probe:  $(h(\text{key}) + 1) \% \text{TableSize}$
- 2<sup>nd</sup> probe:  $(h(\text{key}) + 4) \% \text{TableSize}$
- 3<sup>rd</sup> probe:  $(h(\text{key}) + 9) \% \text{TableSize}$
- ...
- $i^{\text{th}}$  probe:  $(h(\text{key}) + i^2) \% \text{TableSize}$

- Intuition: Probes quickly “leave the neighborhood”

# Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

**TableSize=10**

**Insert:**

**89**

**18**

**49**

**58**

**79**

# Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

**TableSize=10**

**Insert:**

**89**

**18**

**49**

**58**

**79**

# Quadratic Probing Example

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

**TableSize=10**

**Insert:**

**89**

**18**

**49**

**58**

**79**

# Quadratic Probing Example

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

**TableSize=10**

**Insert:**

**89**

**18**

**49**

**58**

**79**

# Quadratic Probing Example

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

**TableSize=10**

**Insert:**

**89**

**18**

**49**

**58**

**79**

# Quadratic Probing Example

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

**TableSize=10**

**Insert:**

**89**

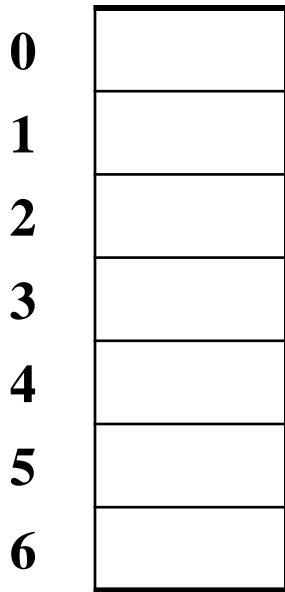
**18**

**49**

**58**

**79**

# *Another Quadratic Probing Example*



**TableSize = 7**

**Insert:**

**76**                    **(76 % 7 = 6)**

**40**                    **(40 % 7 = 5)**

**48**                    **(48 % 7 = 6)**

**5**                      **( 5 % 7 = 5)**

**55**                    **(55 % 7 = 6)**

**47**                    **(47 % 7 = 5)**



# *Another Quadratic Probing Example*

<b>0</b>	
<b>1</b>	
<b>2</b>	
<b>3</b>	
<b>4</b>	
<b>5</b>	
<b>6</b>	76

**TableSize = 7**

**Insert:**

**76**            **(76 % 7 = 6)**

**40**            **(40 % 7 = 5)**

**48**            **(48 % 7 = 6)**

**5**             **( 5 % 7 = 5)**

**55**            **(55 % 7 = 6)**

**47**            **(47 % 7 = 5)**

# *Another Quadratic Probing Example*

<b>0</b>	
<b>1</b>	
<b>2</b>	
<b>3</b>	
<b>4</b>	
<b>5</b>	40
<b>6</b>	76

**TableSize = 7**

**Insert:**

**76**            **(76 % 7 = 6)**

**40**            **(40 % 7 = 5)**

**48**            **(48 % 7 = 6)**

**5**             **( 5 % 7 = 5)**

**55**            **(55 % 7 = 6)**

**47**            **(47 % 7 = 5)**

# Another Quadratic Probing Example

<b>0</b>	48
<b>1</b>	
<b>2</b>	
<b>3</b>	
<b>4</b>	
<b>5</b>	40
<b>6</b>	76

**TableSize = 7**

**Insert:**

**76**            **(76 % 7 = 6)**

**40**            **(40 % 7 = 5)**

**48**            **(48 % 7 = 6)**

**5**             **( 5 % 7 = 5)**

**55**            **(55 % 7 = 6)**

**47**            **(47 % 7 = 5)**

# *Another Quadratic Probing Example*

<b>0</b>	48
<b>1</b>	
<b>2</b>	5
<b>3</b>	
<b>4</b>	
<b>5</b>	40
<b>6</b>	76

**TableSize = 7**

**Insert:**

**76**            **(76 % 7 = 6)**

**40**            **(40 % 7 = 5)**

**48**            **(48 % 7 = 6)**

**5**             **( 5 % 7 = 5)**

**55**            **(55 % 7 = 6)**

**47**            **(47 % 7 = 5)**

# *Another Quadratic Probing Example*

<b>0</b>	48
<b>1</b>	
<b>2</b>	5
<b>3</b>	55
<b>4</b>	
<b>5</b>	40
<b>6</b>	76

**TableSize = 7**

**Insert:**

**76**            **(76 % 7 = 6)**

**40**            **(40 % 7 = 5)**

**48**            **(48 % 7 = 6)**

**5**             **( 5 % 7 = 5)**

**55**            **(55 % 7 = 6)**

**47**            **(47 % 7 = 5)**

# Another Quadratic Probing Example

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

TableSize = 7

Insert:

76	(76 % 7 = 6)
40	(40 % 7 = 5)
48	(48 % 7 = 6)
5	( 5 % 7 = 5)
55	(55 % 7 = 6)
<b>47</b>	(47 % 7 = 5)

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

Proof uses induction and  $(n^2+5) \% 7 = ((n-7)^2+5) \% 7$

In fact, for all  $c$  and  $k$ ,  $(n^2+c) \% k = ((n-k)^2+c) \% k$

# *From Bad News to Good News*

- After **TableSize** quadratic probes, we cycle through the same indices
- The good news:
  - For prime  $T$  and  $0 \leq i, j \leq T/2$  where  $i \neq j$ ,  
$$(h(\text{key}) + i^2) \% T \neq (h(\text{key}) + j^2) \% T$$
  - If  $T = \text{TableSize}$  is *prime* and  $\lambda < 1/2$ ,  
quadratic probing will find an empty slot in at most  $T/2$  probes
  - If you keep  $\lambda < 1/2$ , no need to detect cycles

# *Clustering reconsidered*

- Quadratic probing does not suffer from primary clustering: quadratic nature quickly escapes the neighborhood
- But it's no help if keys *initially hash to the same index*
  - Any 2 keys that hash to the same value will have the same series of moves after that
  - Called **secondary clustering**
- Can avoid secondary clustering with *a probe function that depends on the key*: **double hashing**



# Open Addressing: Double hashing

**Idea:** Given two good hash functions  $h$  and  $g$ ,  
it is very unlikely that for some  $key$ ,  $h(key) == g(key)$

$$(h(key) + f(i)) \% TableSize$$

– For double hashing:

$$f(i) = i * g(key)$$

– So probe sequence is:

- 0<sup>th</sup> probe:  $h(key) \% TableSize$
- 1<sup>st</sup> probe:  $(h(key) + g(key)) \% TableSize$
- 2<sup>nd</sup> probe:  $(h(key) + 2 * g(key)) \% TableSize$
- 3<sup>rd</sup> probe:  $(h(key) + 3 * g(key)) \% TableSize$
- ...
- $i^{\text{th}}$  probe:  $(h(key) + i * g(key)) \% TableSize$

- Detail: Must make sure that  $g(key)$  cannot be 0

# Double Hashing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

$T = 10$  (TableSize)

Hash Functions:

$$h(\text{key}) = \text{key} \bmod T$$

$$g(\text{key}) = 1 + ((\text{key}/T) \bmod (T-1))$$

**Insert these values into the hash table in this order. Resolve any collisions with double hashing:**

**13**

**28**

**33**

**147**

**43**

# Double Hashing

0	
1	
2	
3	13
4	
5	
6	
7	
8	
9	

$T = 10$  (TableSize)

Hash Functions:

$$h(\text{key}) = \text{key} \bmod T$$

$$g(\text{key}) = 1 + ((\text{key}/T) \bmod (T-1))$$

**Insert these values into the hash table in this order. Resolve any collisions with double hashing:**

**13**

**28**

**33**

**147**

**43**

# Double Hashing

0	
1	
2	
3	13
4	
5	
6	
7	
8	28
9	

$T = 10$  (TableSize)

Hash Functions:

$$h(\text{key}) = \text{key} \bmod T$$

$$g(\text{key}) = 1 + ((\text{key}/T) \bmod (T-1))$$

**Insert these values into the hash table in this order. Resolve any collisions with double hashing:**

**13**

**28**

**33**

**147**

**43**

# Double Hashing

0	
1	
2	
3	13
4	
5	
6	
7	33
8	28
9	

$T = 10$  (TableSize)

Hash Functions:

$$h(\text{key}) = \text{key} \bmod T$$

$$g(\text{key}) = 1 + ((\text{key}/T) \bmod (T-1))$$

**Insert these values into the hash table in this order. Resolve any collisions with double hashing:**

**13**

**28**

**33**

**147**

**43**

# Double Hashing

0	
1	
2	
3	13
4	
5	
6	
7	33
8	28
9	

$T = 10$  (TableSize)

Hash Functions:

$$h(\text{key}) = \text{key} \bmod T$$

$$g(\text{key}) = 1 + ((\text{key}/T) \bmod (T-1))$$

**Insert these values into the hash table in this order. Resolve any collisions with double hashing:**

**13**

**28**

**33**

**147**

**43**

# Double Hashing

0	
1	
2	
3	13
4	
5	
6	
7	33
8	28
9	147

$T = 10$  (TableSize)

Hash Functions:

$$h(\text{key}) = \text{key} \bmod T$$

$$g(\text{key}) = 1 + ((\text{key}/T) \bmod (T-1))$$

**Insert these values into the hash table in this order. Resolve any collisions with double hashing:**

**13**

**28**

**33**

**147**

**43**

**Doh:**

$$3 + 0 = 3$$

$$3 + 15 = 18$$

$$3 + 5 = 8$$

$$3 + 20 = 23$$

$$3 + 10 = 13$$

$$3 + 25 = 28$$

# Double Hashing Analysis

- Intuition:

Because each probe is “jumping” by  $g(\mathbf{key})$  each time, we should both “leave the neighborhood” *and* “go different places from the same initial collision”

- But, as in quadratic probing, we could still have a problem where we are not “safe” (infinite loop despite room in table)
- It is known that this cannot happen in at least one case:
  - $h(\mathbf{key}) = \mathbf{key} \% p$
  - $g(\mathbf{key}) = q - (\mathbf{key} \% q)$
  - $2 < q < p$
  - $p$  and  $q$  are prime



# *Where are we?*

- Separate Chaining is easy
  - **find**, **delete** proportional to load factor on average
  - **insert** can be constant if just push on front of list
- Open addressing uses probing, has clustering issues as it gets full
  - Why use it:
    - Less memory allocation?
    - Run-time overhead for list nodes; array could be faster?
    - Easier data representation?
- Now:
  - Growing the table when it gets too full (aka “rehashing”)
  - Relation between hashing/comparing and connection to Java

# Rehashing

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

# Rehashing

- What if we copy all data to the same indices in the new table?
  - Will not work; we calculated the index based on TableSize
- Go through table, do standard insert for each into new table
  - Run-time?
  - $O(n)$ : Iterate through old table
- Resize is an  $O(n)$  operation, involving  $n$  calls to the hash function
  - Is there some way to avoid all those hash function calls?
  - Space/time tradeoff: Could store  $h(\mathbf{key})$  with each data item
  - Growing the table is still  $O(n)$ ; only helps by a constant factor

# Hashing and Comparing

- Our use of `int` key can lead to overlooking a critical detail
  - We initial *hash E*,
  - While chaining or probing, we *compare* to **E**.
    - Just need equality testing (i.e., `compare == 0`)
- So a hash table needs a hash function and a comparator
  - In Project 2, you will use two function objects
  - The Java library uses a more object-oriented approach: each object has an **equals** method and a **hashCode** method:

```
class Object {
    boolean equals(Object o) {...}
    int hashCode() {...}
    ...
}
```

# *Equal Objects Must Hash the Same*

- The Java library (and your project hash table) make a very important assumption that clients must satisfy
- Object-oriented way of saying it:  
    If `a.equals(b)`, then we must require  
    `a.hashCode() == b.hashCode()`
- Function object way of saying it:  
    If `c.compare(a,b) == 0`, then we must require  
    `h.hash(a) == h.hash(b)`
- If you ever override `equals`
  - You need to override `hashCode` also in a consistent way
  - See CoreJava book, Chapter 5 for other “gotchas” with `equals`

# *Comparable/Comparator Have Rules Too*

We have not emphasized important “rules” about comparison for:

- all our dictionaries
- sorting (next major topic)

Comparison must impose a consistent, total ordering:

For all **a**, **b**, and **c**,

- If **compare (a , b) < 0**, then **compare (b , a) > 0**
- If **compare (a , b) == 0**, then **compare (b , a) == 0**
- If **compare (a , b) < 0** and  
**compare (b , c) < 0**, then **compare (a , c) < 0**

# *A Generally Good hashCode()*

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



# *Final Word on Hashing*

- The hash table is one of the most important data structures
  - Efficient **find, insert, and delete**
  - Operations based on sort order are not so efficient
    - e.g., **FindMin, FindMax, predecessor**
- Important to use a good hash function
  - Good distribution, uses enough of key's meaningful values
- Important to keep hash table at a good size
  - Prime #, preferable  $\lambda$  depends on type of table
- Popular topic for job interview questions
  - Also many real-world applications





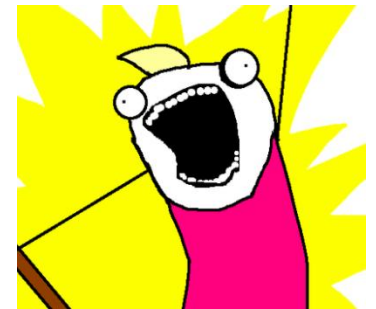
# CSE332: Data Abstractions

## Lecture 10: Comparison Sorting

James Fogarty  
Winter 2012

# *Introduction to Sorting*

- We have covered stacks, queues, priority queues, and dictionaries
  - All focused on providing one element at a time
- But often we know we want “all the things” in some order
  - Anyone can sort, but a computer can sort faster
  - Very common to need data sorted somehow
    - Alphabetical list of people
    - List of countries ordered by population
- Algorithms have different asymptotic and constant-factor trade-offs
  - No single “best” sort for all scenarios
  - Knowing “one way to sort” is not sufficient



# *More Reasons to Sort*

General technique in computing:

*Preprocess data to make subsequent operations faster*

Example: Sort the data so that you can

- Find the  $k^{\text{th}}$  largest in constant time for any  $k$
- Perform binary search to find elements in logarithmic time

Whether the performance of the preprocessing matters depends on

- How often the data will change
- How much data there is

# Careful Statement of the Basic Problem

Assume we have  $n$  comparable elements in an array, and we want to rearrange them to be in increasing order

Input:

- An array  $\mathbf{A}$  of data records
- A key value in each data record (potentially a set of fields)
- A comparison function (must be consistent and total)
  - Given keys  $a$  and  $b$ , what is their relative ordering?  $<$ ,  $=$ ,  $>$ ?

Effect:

- Reorganize the elements of  $\mathbf{A}$  such that for any  $i$  and  $j$ ,  
if  $i < j$  then  $\mathbf{A}[i] \leq \mathbf{A}[j]$
- Unspoken assumption:  $\mathbf{A}$  must have all the data it started with

An algorithm doing this is a **comparison sort**

# *Variations on the basic problem*

1. Maybe elements are in a linked list (could convert to array and back in linear time, but some algorithms need not do so)
2. Maybe ties need to be resolved by “original array position”
  - Sorts that do this naturally are called **stable sorts**
  - Others could tag each item with its original position and adjust their comparisons (non-trivial constant factors)
3. Maybe we must not use more than  $O(1)$  “auxiliary space”
  - Sorts meeting this requirement are called **in-place sorts**
4. Maybe we can do more with elements than just compare
  - Sometimes leads to faster algorithms
5. Maybe we have too much data to fit in memory
  - Use an “**external sorting**” algorithm

# Sorting: The Big Picture

**Simple algorithms:**  
 $O(n^2)$

**Insertion sort**  
**Selection sort**  
**Shell sort**  
...

**Fancier algorithms:**  
 $O(n \log n)$

**Heap sort**  
**Merge sort**  
**Quick sort (avg)**  
...

**Comparison lower bound:**  
 $\Omega(n \log n)$

**Specialized algorithms:**  
 $O(n)$

**Bucket sort**  
**Radix sort**

**Handling huge data sets**

**External sorting**

# *Insertion Sort*

- Idea: At step  $k$ ,  
put the  $k^{\text{th}}$  input element in the correct position  
among the first  $k$  elements
- Alternate way of saying this:
  - Sort first element (this is easy)
  - Now insert 2<sup>nd</sup> element in order
  - Now insert 3<sup>rd</sup> element in order
  - Now insert 4<sup>th</sup> element in order
  - ...
- “Loop invariant”: when loop index is  $i$ , first  $i$  elements are sorted
- Time?  
Best-case \_\_\_\_\_ Worst-case \_\_\_\_\_ “Average” case \_\_\_\_\_

# *Insertion Sort*

- Idea: At step  $k$ ,  
put the  $k^{\text{th}}$  input element in the correct position  
among the first  $k$  elements
- Alternate way of saying this:
  - Sort first element (this is easy)
  - Now insert 2<sup>nd</sup> element in order
  - Now insert 3<sup>rd</sup> element in order
  - Now insert 4<sup>th</sup> element in order
  - ...
- “Loop invariant”: when loop index is  $i$ , first  $i$  elements are sorted
- Time?
  - Best-case  $O(n)$  Worst-case  $O(n^2)$  “Average” case  $O(n^2)$   
start sorted start reverse sorted (see text)



# *Selection Sort*

- Idea: At step  $k$ ,  
find the smallest element among the unsorted elements  
and put it at position  $k$
- Alternate way of saying this:
  - Find smallest element, put it 1<sup>st</sup>
  - Find next smallest element, put it 2<sup>nd</sup>
  - Find next smallest element, put it 3<sup>rd</sup>
  - ...
- “Loop invariant”: when loop index is  $i$ ,  
first  $i$  elements are the  $i$  smallest elements in sorted order
- Time?  
Best-case \_\_\_\_\_ Worst-case \_\_\_\_\_ “Average” case \_\_\_\_\_

# *Selection Sort*

- Idea: At step  $k$ ,  
find the smallest element among the unsorted elements  
and put it at position  $k$
- Alternate way of saying this:
  - Find smallest element, put it 1<sup>st</sup>
  - Find next smallest element, put it 2<sup>nd</sup>
  - Find next smallest element, put it 3<sup>rd</sup>
  - ...
- “Loop invariant”: when loop index is  $i$ ,  
first  $i$  elements are the  $i$  smallest elements in sorted order
- Time?  
Best-case  $O(n^2)$  Worst-case  $O(n^2)$  “Average” case  $O(n^2)$   
*Always*  $T(1) = 1$  and  $T(n) = n + T(n-1)$

# Mystery Sort

This is one implementation of which sorting algorithm (shown for ints)?

```
void mystery(int[] arr) {
    for(int i = 1; i < arr.length; i++) {
        int tmp = arr[i];
        int j;
        for(j=i; j > 0 && tmp < arr[j-1]; j--)
            arr[j] = arr[j-1];
        arr[j] = tmp;
    }
}
```

Note: As with heaps, “moving the hole” is faster than unnecessary swapping (impacts constant factor)

# *Insertion Sort vs. Selection Sort*

- They are different algorithms
- They solve the same problem
- Have the same worst-case and average-case asymptotic complexity
  - Insertion-sort has better best-case complexity; preferable when input is “mostly sorted”
- Other algorithms are more efficient
  - for non-small arrays that are not already almost sorted*
    - Small arrays may do well with Insertion sort

## *Aside: We Will Not Cover Bubble Sort*

- It does not have good asymptotic complexity:  $O(n^2)$
- It is not particularly efficient with respect to constant factors
- Almost everything it is good at,  
some other algorithm is at least as good at
- Perhaps some people teach it just because it was taught to them
- For fun see: “Bubble Sort: An Archaeological Algorithmic Analysis”, Owen Astrachan, SIGCSE 2003

# Sorting: The Big Picture

**Simple algorithms:**  
 $O(n^2)$

**Insertion sort**  
**Selection sort**  
**Shell sort**  
...

**Fancier algorithms:**  
 $O(n \log n)$

**Heap sort**  
**Merge sort**  
**Quick sort (avg)**  
...

**Comparison lower bound:**  
 $\Omega(n \log n)$

**Specialized algorithms:**  
 $O(n)$

**Bucket sort**  
**Radix sort**

**Handling huge data sets**

**External sorting**

# Heap Sort

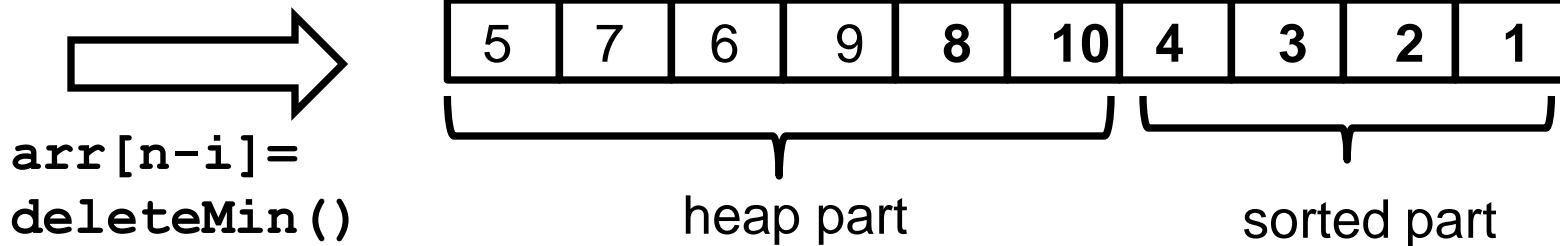
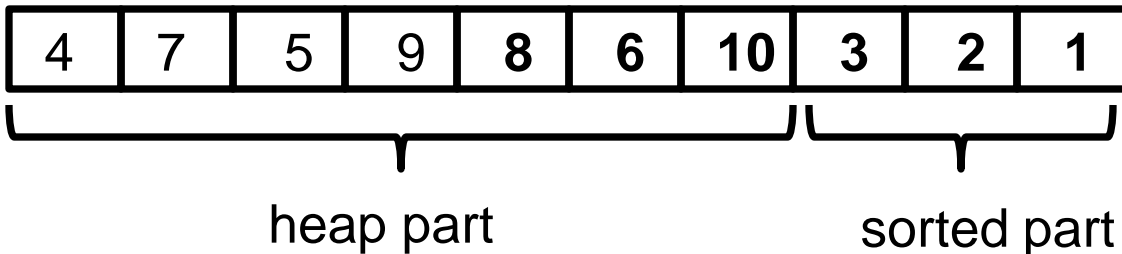
- As you are seeing in Project 2, sorting with a heap is easy:
  - `insert` each `arr[i]`, or better yet do a `buildHeap`
  - `for(i=0; i < arr.length; i++)`  
`arr[i] = deleteMin();`
- Worst-case running time:  $O(n \log n)$  **Why?**
- We have the array-to-sort and the heap
  - So this is not an in-place sort
  - There's a trick to make it in-place

But this reverse sorts –  
how would you fix that?

# *In-Place Heap Sort*

Reverse your comparator,  
so you build a maxHeap

- Treat the initial array as a heap (via `buildHeap`)
- When you delete the  $i^{\text{th}}$  element, put it at `arr[n-i]`
  - That array location is not part of the heap anymore!





# “AVL sort”

- We can also use a balanced tree to:
  - **insert** each element: total time  $O(n \log n)$
  - Repeatedly **deleteMin**: total time  $O(n \log n)$
- But this cannot be made in-place, and it has worse constant factors than heap sort
  - both are  $O(n \log n)$  in worst, best, and average case
  - neither parallelizes well
  - heap sort is better
- Do not even think about trying to sort with a hash table

# *Divide and Conquer*

Very important technique in algorithm design

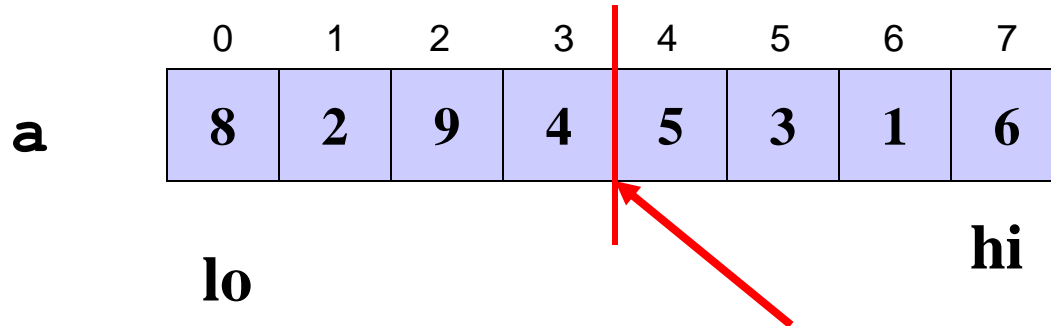
1. Divide problem into smaller parts
2. Independently solve the simpler parts
  - Think recursion
  - Or potential parallelism
3. Combine solution of parts to produce overall solution

# *Divide-and-Conquer Sorting*

Two great sorting methods are fundamentally divide-and-conquer

1. Mergesort: Sort the left half of the elements (recursively)  
Sort the right half of the elements (recursively)  
Merge the two sorted halves into a sorted whole
2. Quicksort: Pick a “pivot” element  
Divide elements into less-than pivot  
and greater-than pivot  
Sort the two divisions (recursively on each)  
Answer is [ *sorted-less-than,*  
*then pivot,*  
*then sorted-greater-than* ]

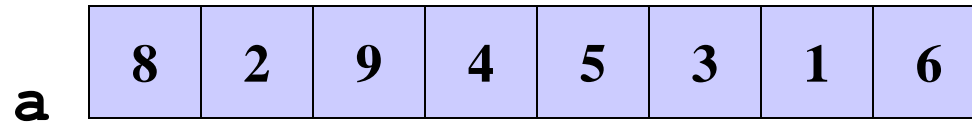
# Mergesort



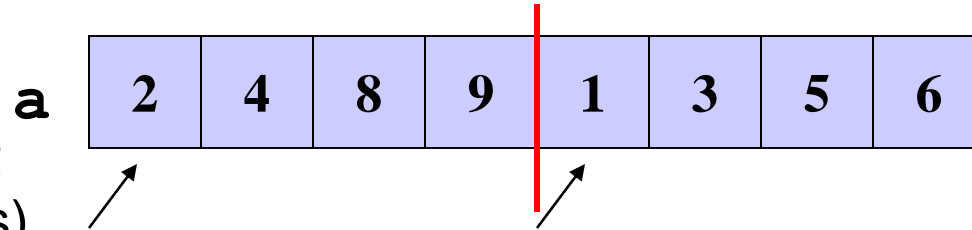
- To sort array from position **lo** to position **hi**:
  - If range is 1 element long, it is already sorted! (our base case)
  - Else, split into two halves:
    - Sort from **lo** to  $(\mathbf{hi} + \mathbf{lo}) / 2$
    - Sort from  $(\mathbf{hi} + \mathbf{lo}) / 2$  to **hi**
    - Merge the two halves together
- Merging takes two sorted parts and sorts everything
  - $O(n)$  but requires auxiliary space...

# Example: Focus on Merging

Start with:

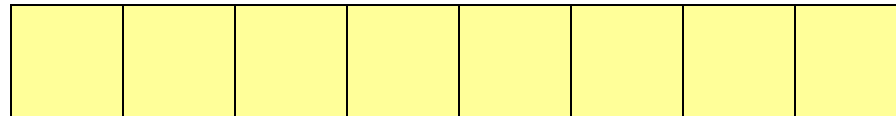


After recursion:  
(for now we just  
assume it works)



Merge:

Use 3 "fingers" **aux**  
and 1 more array

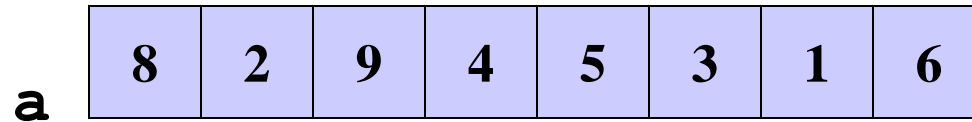


(After merge,  
copy back to  
original array)

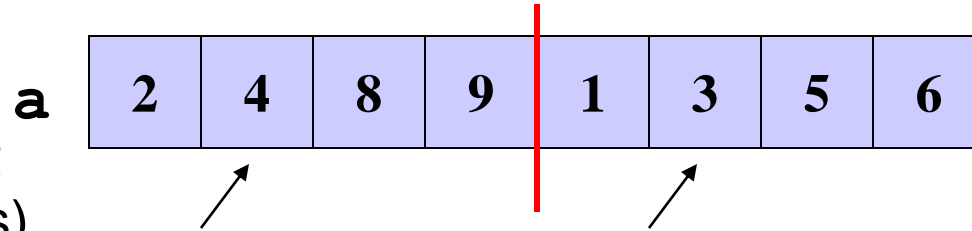


# Example: Focus on Merging

Start with:

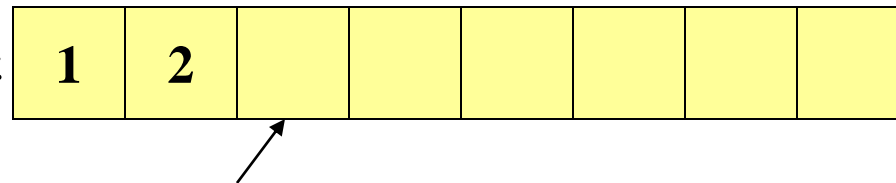


After recursion:  
(for now we just  
assume it works)



Merge:

Use 3 “fingers” **aux**  
and 1 more array



(After merge,  
copy back to  
original array)

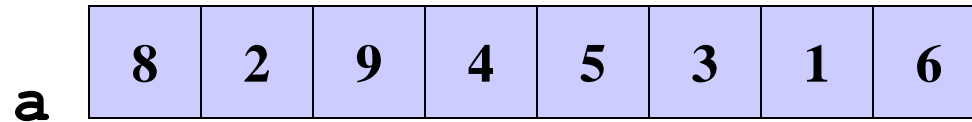




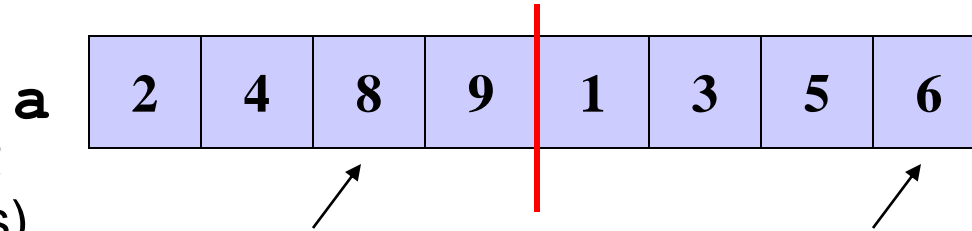


# Example: Focus on Merging

Start with:

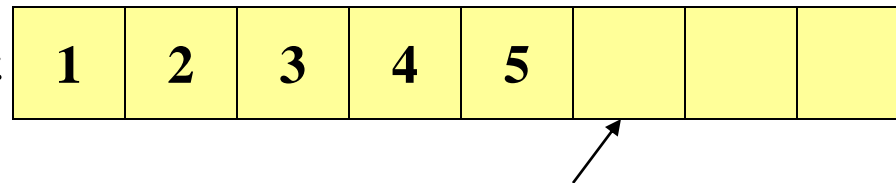


After recursion:  
(for now we just  
assume it works)



Merge:

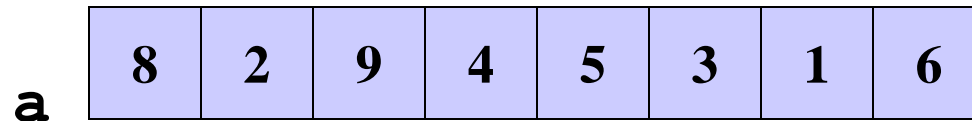
Use 3 “fingers” **aux**  
and 1 more array



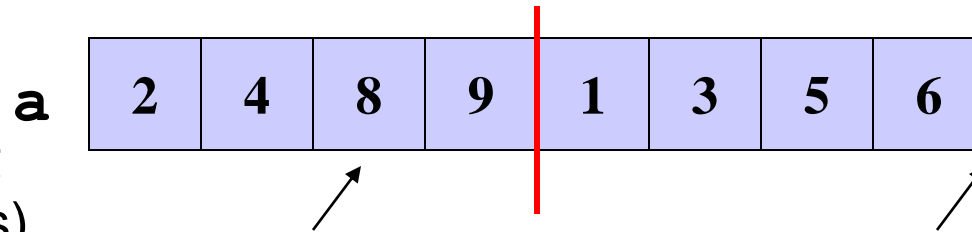
(After merge,  
copy back to  
original array)

# Example: Focus on Merging

Start with:

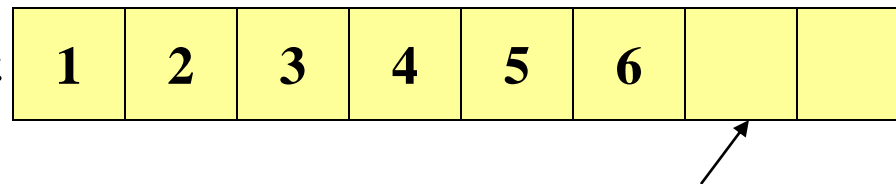


After recursion:  
(for now we just  
assume it works)



Merge:

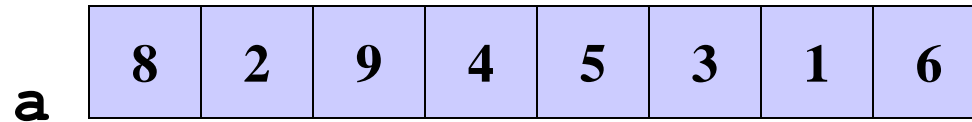
Use 3 “fingers” **aux**  
and 1 more array



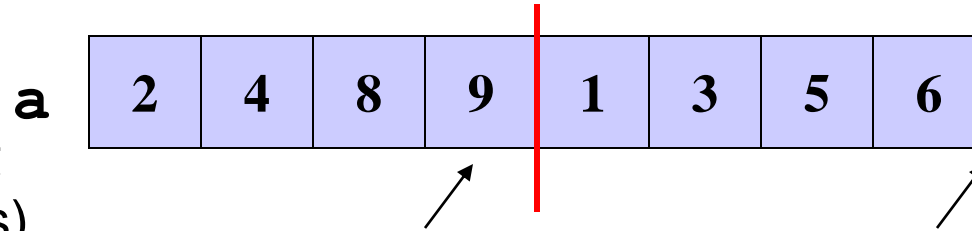
(After merge,  
copy back to  
original array)

# Example: Focus on Merging

Start with:

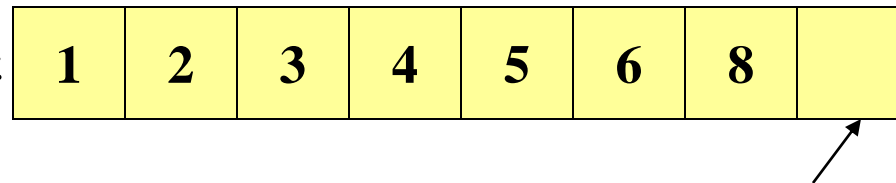


After recursion:  
(for now we just  
assume it works)



Merge:

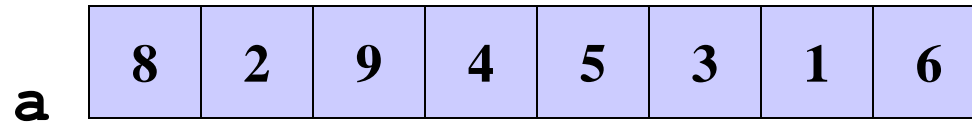
Use 3 “fingers” **aux**  
and 1 more array



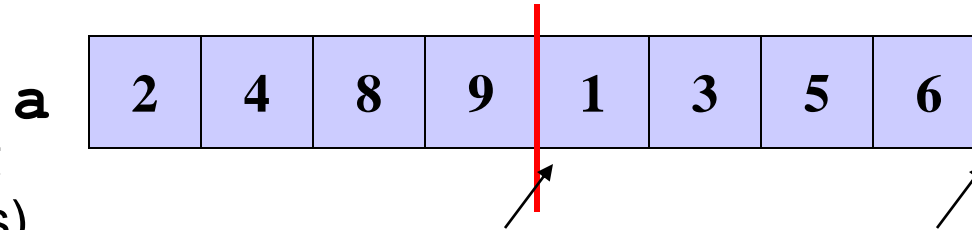
(After merge,  
copy back to  
original array)

# Example: Focus on Merging

Start with:

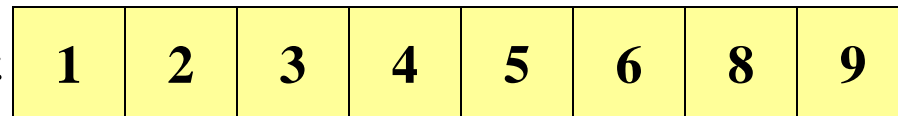


After recursion:  
(for now we just  
assume it works)



Merge:

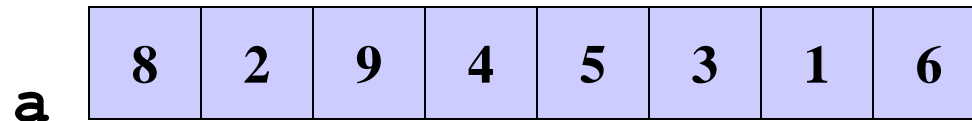
Use 3 “fingers” **aux**  
and 1 more array



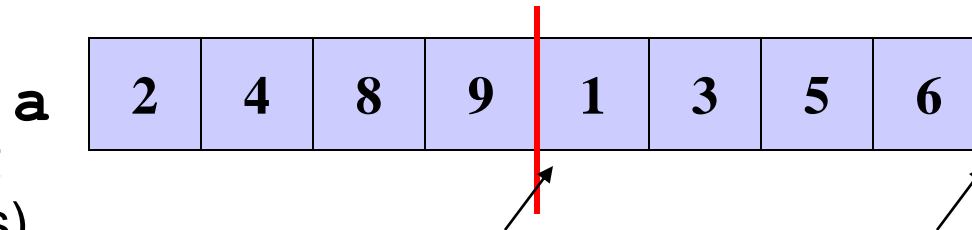
(After merge,  
copy back to  
original array)

# Example: Focus on Merging

Start with:

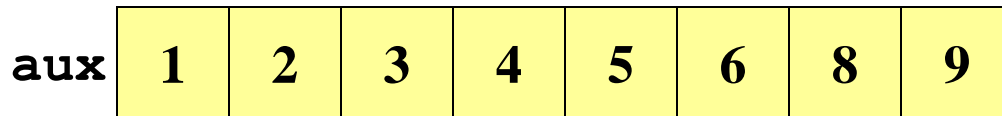


After recursion:  
(for now we just  
assume it works)



Merge:

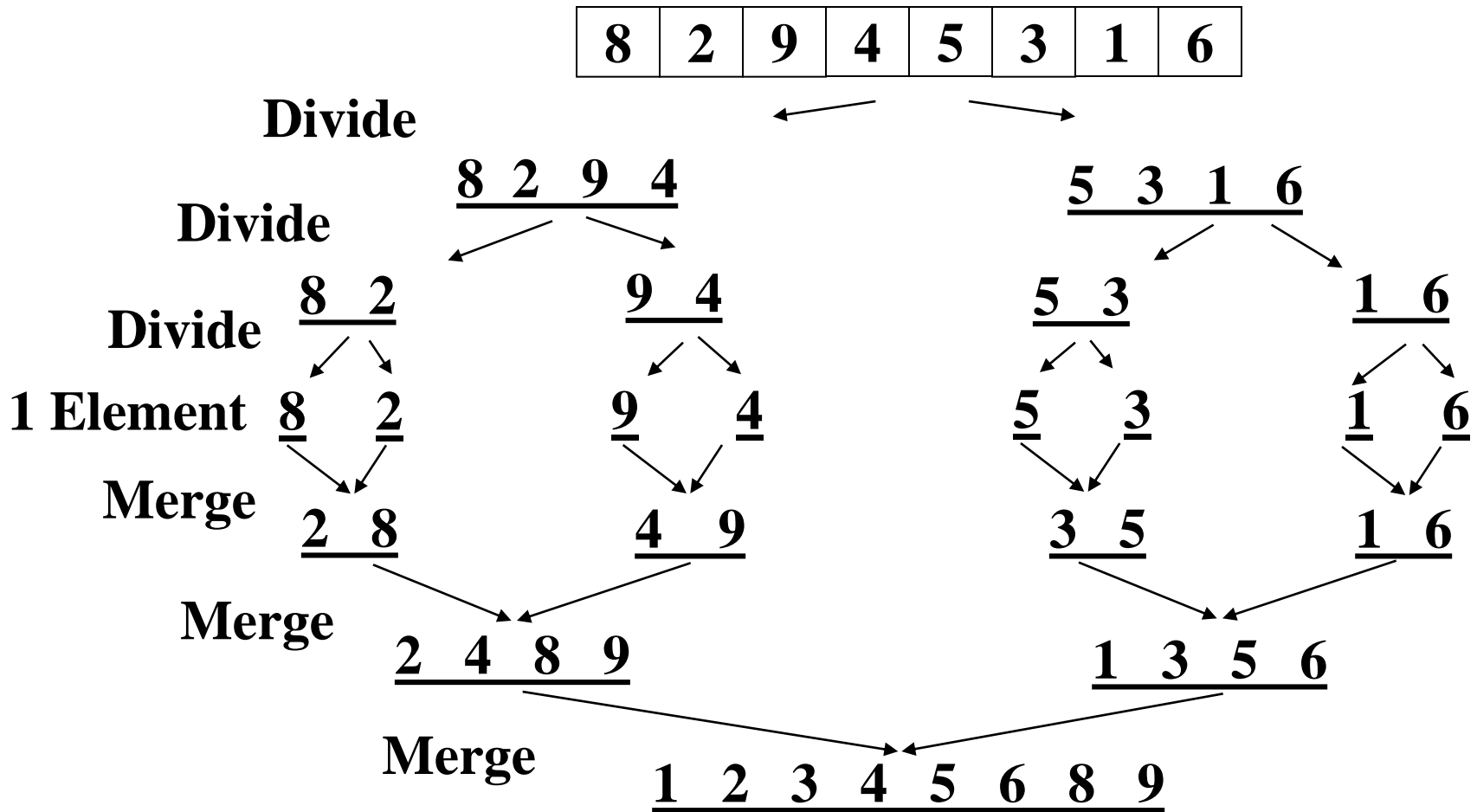
Use 3 “fingers”  
and 1 more array



(After merge,  
copy back to  
original array)

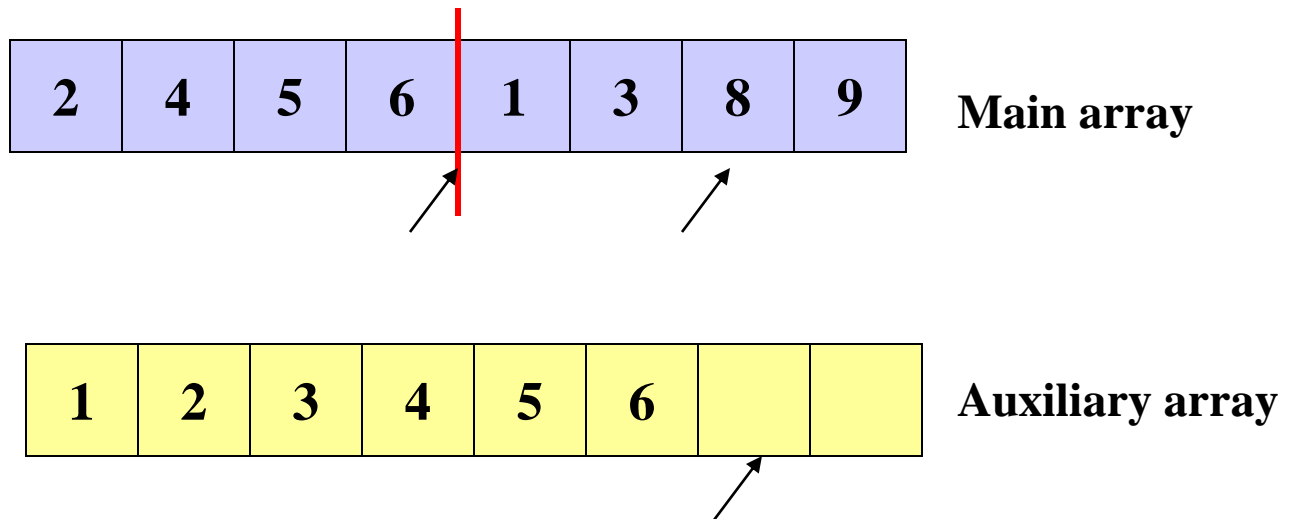


# Example: Mergesort Recursion



# Mergesort: Some Time Saving Details

- What if the final steps of our merge looked like this:

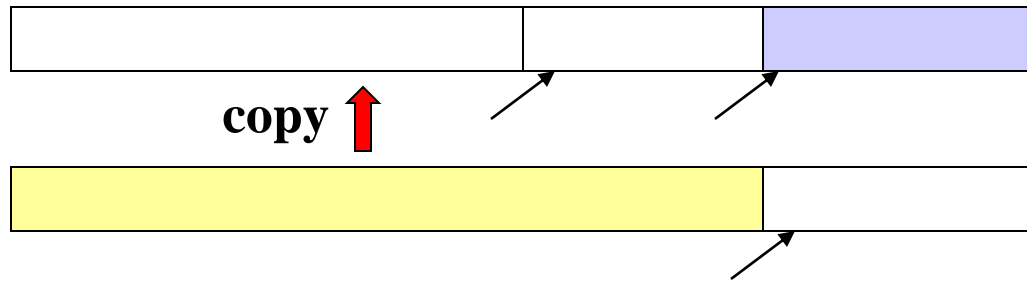


- Wasteful to copy to the auxiliary array just to copy back...

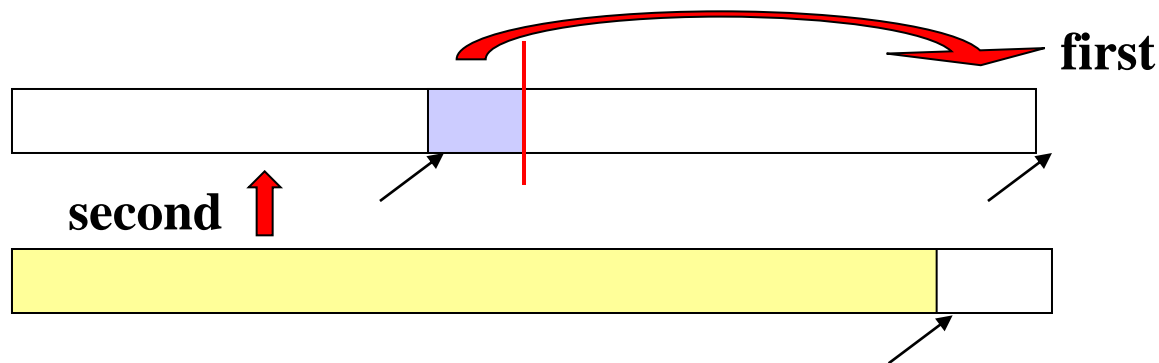


# Mergesort: Some Time Saving Details

- If left-side finishes first, just stop the merge and copy back:



- If right-side finishes first, copy drags into right then copy back:



# *Mergesort: Saving Space and Copying*

Simplest / Worst:

Use a new auxiliary array of size  $(h_i - l_o)$  for every merge

Better:

Use a new auxiliary array of size  $n$  for every merging stage

Better:

Reuse same auxiliary array of size  $n$  for every merging stage

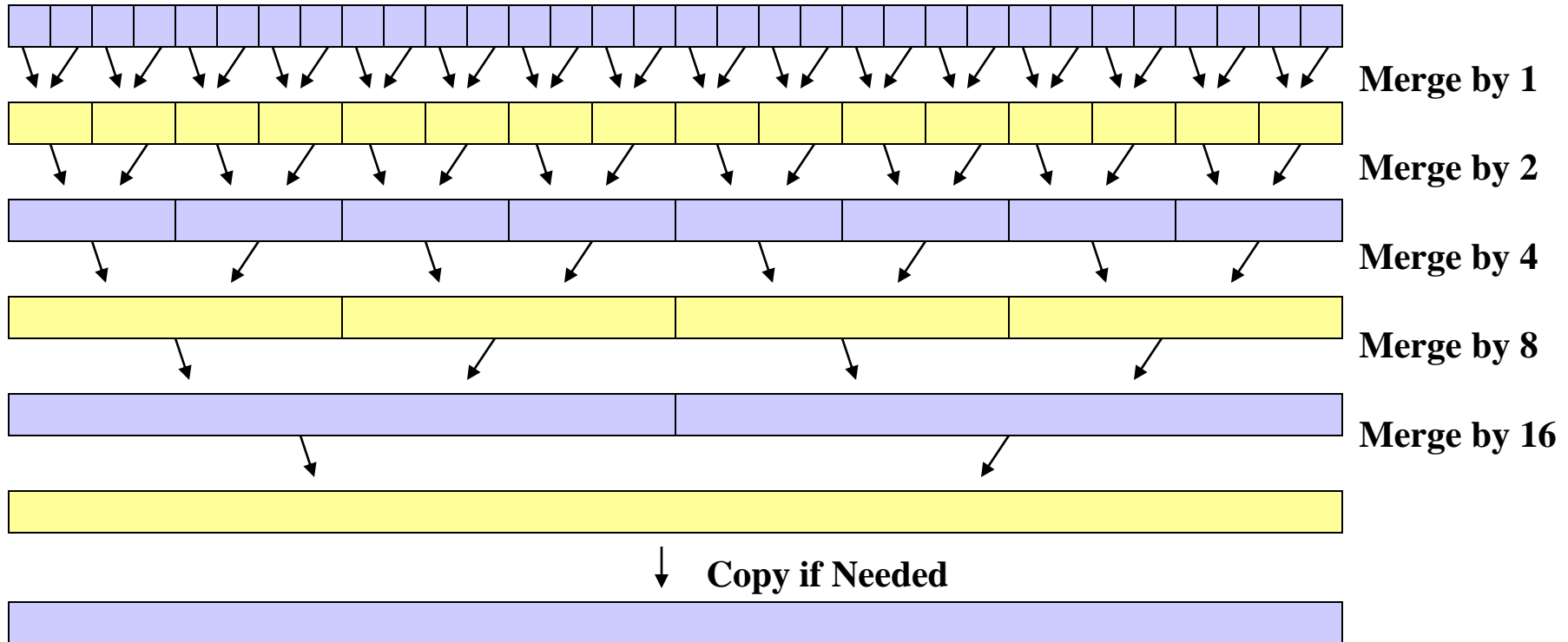
Best:

Do not copy back after merge, instead swap usage of the original and auxiliary array (i.e., even levels move to auxiliary array, odd levels move back to original array)

- Need one copy at end if number of stages is odd

# Swapping Original and Auxiliary Array

- First recurse down to lists of size 1
- As we return from the recursion, swap between arrays



- Arguably easier to code without using recursion at all

# *Mergesort Analysis*

Having defined an algorithm and argued it is correct, we can analyze its running time and space:

To sort  $n$  elements, we:

- Return immediately if  $n=1$
- Else do 2 subproblems of size  $n/2$  and then an  $O(n)$  merge

Recurrence relation:

$$T(1) = c_1$$

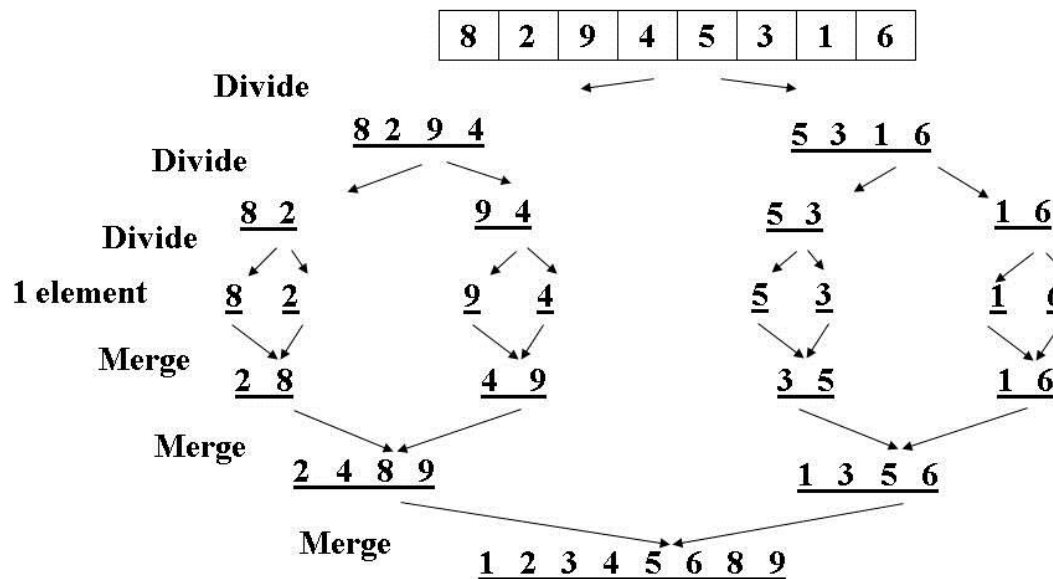
$$T(n) = 2T(n/2) + c_2n$$

# Mergesort Analysis

This recurrence is common enough you just “know” it’s  $O(n \log n)$

Merge sort is relatively easy to intuit (best, worst, and average):

- The recursion “tree” will have  $\log n$  height
- At each level we do a *total* amount of merging equal to  $n$



# Quicksort

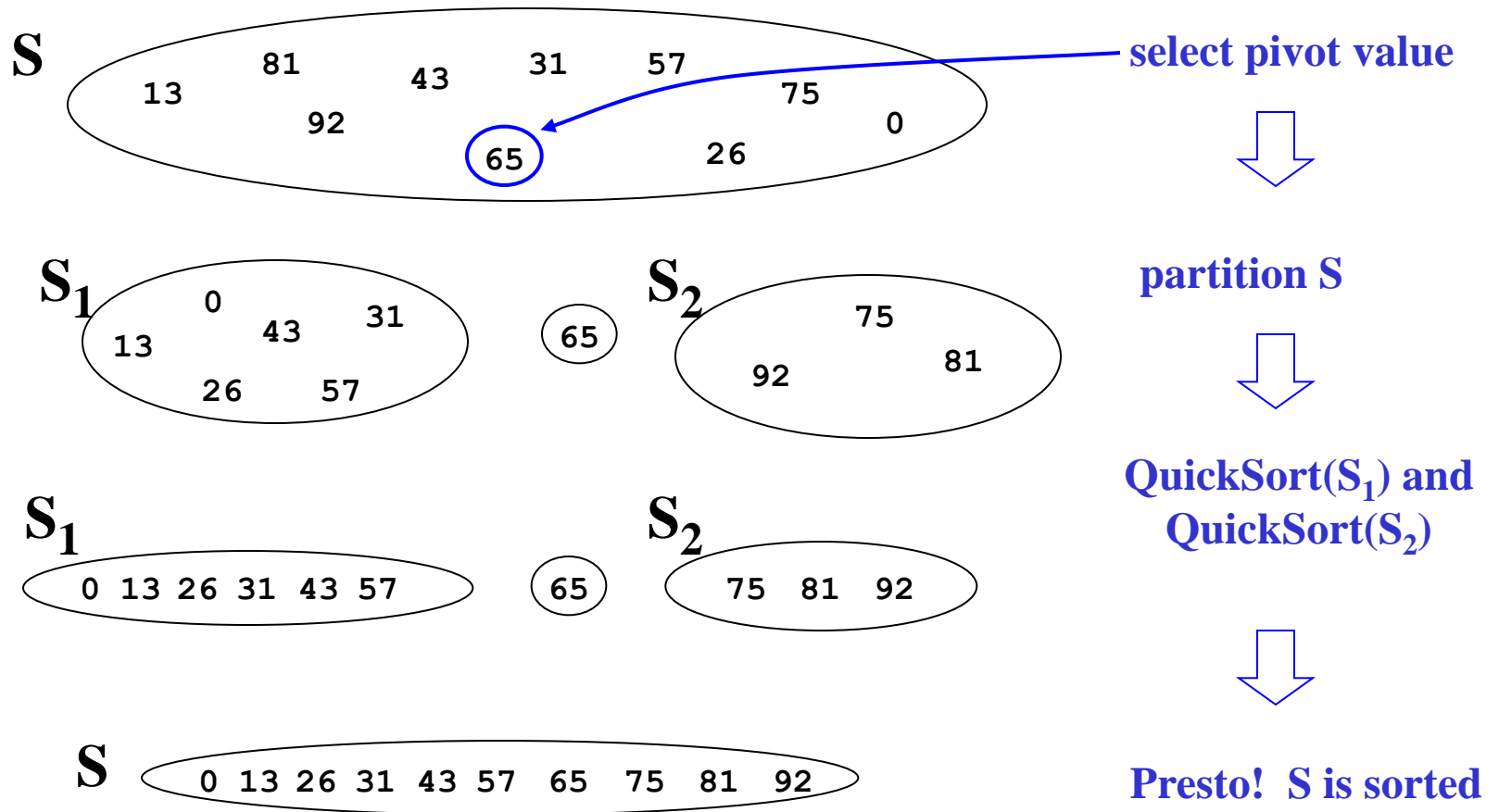
- Also uses divide-and-conquer
  - Recursively chop into halves
  - Instead of doing all the work as we merge together, we will do all the work as we recursively split into halves
  - Unlike MergeSort, does not need auxiliary space
- $O(n \log n)$  on average, but  $O(n^2)$  worst-case
  - MergeSort is always  $O(n \log n)$
  - So why use QuickSort at all?
- Can be faster than Mergesort
  - Believed by many to be faster
  - Quicksort does fewer copies and more comparisons, so it depends on the relative cost of these two operations!

# *Quicksort Overview*

1. Pick a pivot element
2. Partition all the data into:
  - A. The elements less than the pivot
  - B. The pivot
  - C. The elements greater than the pivot
3. Recursively sort A and C
4. The answer is as simple as “A, B, C”

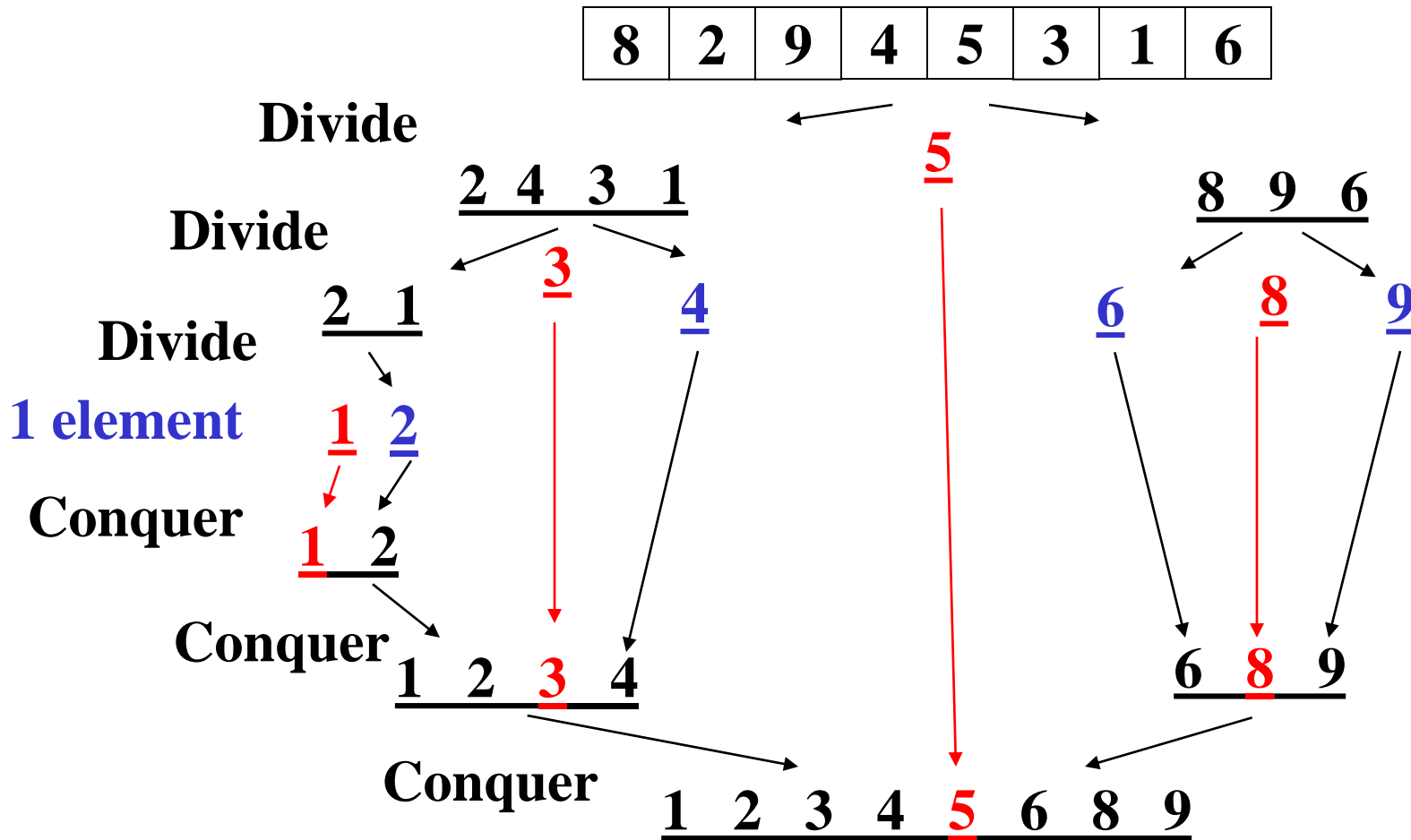
Alas, there are some details lurking in this algorithm

# Quicksort: Think in Terms of Sets





# Example: Quicksort Recursion



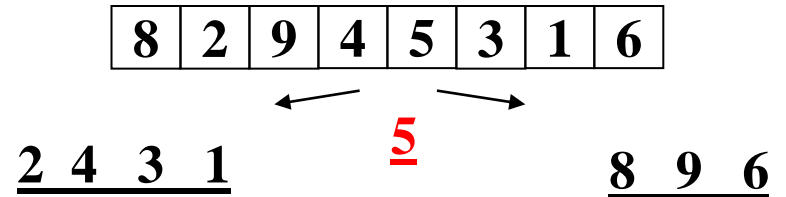
# *Quicksort Details*

We have not explained:

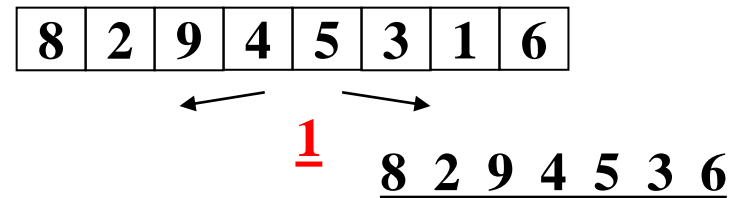
- How to pick the pivot element
  - Any choice is correct: data will end up sorted
  - But we want the two partitions to be about equal in size
- How to implement partitioning
  - In linear time
  - In place

# Pivots

- Best pivot?
  - Median
  - Halve each time



- Worst pivot?
  - Greatest/least element
  - Problem of size  $n - 1$
  - $O(n^2)$



# *Quicksort: Potential Pivot Rules*

While sorting `arr` from `lo` (inclusive) to `hi` (exclusive):

- Pick `arr[lo]` or `arr[hi-1]`
  - Fast, but worst-case occurs with approximately sorted input
- Pick random element in the range
  - Does as well as any technique
    - But random number generation can be slow
    - Still probably the most elegant approach
- Median of 3, (e.g., `arr[lo]`, `arr[hi-1]`, `arr[(hi+lo)/2]`)
  - Common heuristic that tends to work well

# *Partitioning*

- Conceptually simple, but hardest part to code up correctly
  - After picking pivot, need to partition in linear time in place
- One approach (there are slightly fancier ones):
  1. Swap pivot with `arr[lo]`
  2. Use two fingers `i` and `j`, starting at `lo+1` and `hi-1`
  3. `while (i < j)`
    - `if (arr[j] >= pivot) j--`
    - `else if (arr[i] =< pivot) i++`
    - `else swap arr[i] with arr[j]`
  4. Swap pivot with `arr[i]`

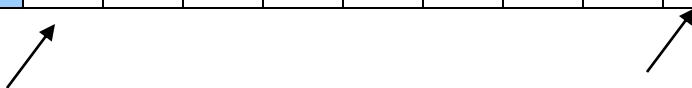
# Quicksort Example

- Step One: Pick Pivot as Median of 3
  - $lo = 0, hi = 10$

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

- Step Two: Move Pivot to the  $lo$  Position

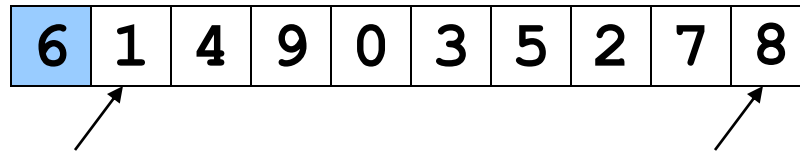
0	1	2	3	4	5	6	7	8	9
6	1	4	9	0	3	5	2	7	8



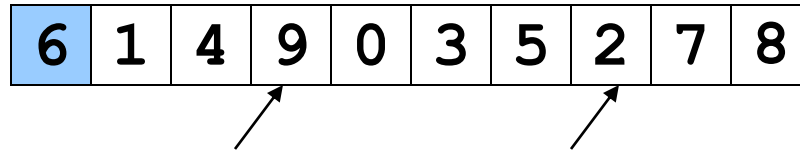
# Quicksort Example

Often have more than one swap during partition – this is a short example

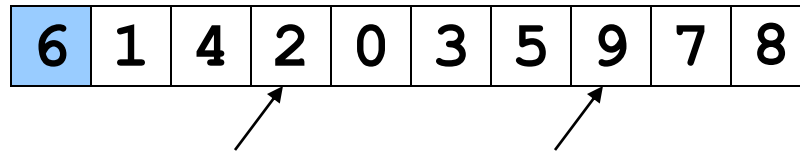
Now partition in place



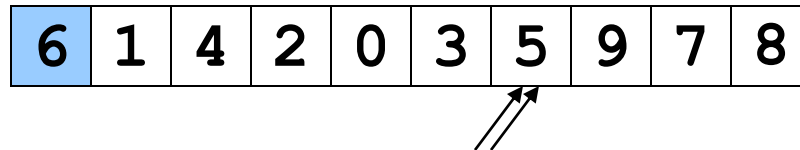
Move fingers



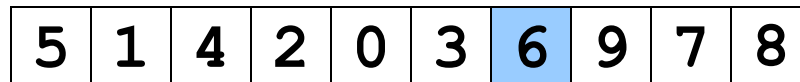
Swap



Move fingers



Move pivot



# Quicksort Analysis

- Best-case: Pivot is always the median

$$T(0)=T(1)=1$$

$$T(n)=2T(n/2) + n \quad \text{-- linear-time partition}$$

Same recurrence as mergesort:  $O(n \log n)$

- Worst-case: Pivot is always smallest or largest element

$$T(0)=T(1)=1$$

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

Basically same recurrence as selection sort:  $O(n^2)$

- Average-case (e.g., with random pivot)
  - $O(n \log n)$  (see text)



# Quicksort Cutoffs

- For small  $n$ , recursion tends to cost more than a quadratic sort
  - Remember asymptotic complexity is for large  $n$
  - Also, recursive calls add a lot of overhead for small  $n$
- Common technique: switch algorithm below a **cutoff**
  - Reasonable rule of thumb: use insertion sort for  $n < 10$
- Notes:
  - Could also use a cutoff for merge sort
  - Cutoffs are also the norm with parallel algorithms
    - Switch to sequential algorithm
  - None of this affects asymptotic complexity

# Quicksort Cutoff Skeleton

```
void quicksort(int[] arr, int lo, int hi) {  
    if (hi - lo < CUTOFF)  
        insertionSort(arr, lo, hi);  
    else  
        ...  
}
```

This cuts out the vast majority of the recursive calls

- Think of the recursive calls to quicksort as a tree
- Trims out the bottom layers of the tree



# CSE332: Data Abstractions

## Lecture 11: Beyond Comparison Sorting

James Fogarty

Winter 2012

# Sorting: The Big Picture

**Simple  
algorithms:  
 $O(n^2)$**

**Insertion sort**  
**Selection sort**  
**Shell sort**  
...

**Fancier  
algorithms:  
 $O(n \log n)$**

**Heap sort**  
**Merge sort**  
**Quick sort (avg)**  
...

**Comparison  
lower bound:  
 $\Omega(n \log n)$**

**Specialized  
algorithms:  
 $O(n)$**

**Bucket sort**  
**Radix sort**

**Handling  
huge data  
sets**

**External  
sorting**

# *Divide-and-Conquer Sorting*

Two great sorting methods are fundamentally divide-and-conquer

1. Mergesort: Sort the left half of the elements (recursively)  
Sort the right half of the elements (recursively)  
Merge the two sorted halves into a sorted whole
2. Quicksort: Pick a “pivot” element  
Divide elements into less-than pivot  
and greater-than pivot  
Sort the two divisions (recursively on each)  
Answer is [ *sorted-less-than,*  
*then pivot,*  
*then sorted-greater-than* ]

# Quicksort Analysis

- Best-case: Pivot is always the median

$$T(0)=T(1)=1$$

$$T(n)=2T(n/2) + n \quad \text{-- linear-time partition}$$

Same recurrence as mergesort:  $O(n \log n)$

- Worst-case: Pivot is always smallest or largest element

$$T(0)=T(1)=1$$

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

Basically same recurrence as selection sort:  $O(n^2)$

- Average-case (e.g., with random pivot)
  - $O(n \log n)$  (see text)

# Quicksort Cutoffs

- For small  $n$ , recursion tends to cost more than a quadratic sort
  - Remember asymptotic complexity is for large  $n$
  - Also, recursive calls add a lot of overhead for small  $n$
- Common technique: switch algorithm below a **cutoff**
  - Reasonable rule of thumb: use insertion sort for  $n < 10$
- Notes:
  - Could also use a cutoff for merge sort
  - Cutoffs are also the norm with parallel algorithms
    - Switch to sequential algorithm
  - None of this affects asymptotic complexity

# Quicksort Cutoff Skeleton

```
void quicksort(int[] arr, int lo, int hi) {  
    if (hi - lo < CUTOFF)  
        insertionSort(arr, lo, hi);  
    else  
        ...  
}
```

This cuts out the vast majority of the recursive calls

- Think of the recursive calls to quicksort as a tree
- Trims out the bottom layers of the tree



# *Linked Lists and Big Data*

We defined sorting over an array, but sometimes you want to sort lists

One approach:

- Convert to array:  $O(n)$ , Sort:  $O(n \log n)$ , Convert to list:  $O(n)$

Mergesort can very nicely work directly on linked lists

- heapsort and quicksort do not
- insertion sort and selection sort can, but they are slower

Mergesort is also the sort of choice for external sorting

- Quicksort and Heapsort jump all over the array
- Mergesort scans linearly through arrays
- In-memory sorting of blocks can be combined with larger sorts
- Mergesort can leverage multiple disks

# *The Big Picture*

**Simple  
algorithms:  
 $O(n^2)$**

**Insertion sort**  
**Selection sort**  
**Shell sort**  
...

**Fancier  
algorithms:  
 $O(n \log n)$**

**Heap sort**  
**Merge sort**  
**Quick sort (avg)**  
...

**Comparison  
lower bound:  
 $\Omega(n \log n)$**

**Specialized  
algorithms:  
 $O(n)$**

**Bucket sort**  
**Radix sort**

**Handling  
huge data  
sets**

**External  
sorting**

# *How Fast can we Sort?*

- Heapsort & Mergesort have  $O(n \log n)$  worst-case running time
- Quicksort has  $O(n \log n)$  average-case running times
- These bounds are all tight, actually  $\Theta(n \log n)$
- So maybe we need to dream up another algorithm with a lower asymptotic complexity, such as  $O(n)$  or  $O(n \log \log n)$ 
  - Instead we *prove* that this is *impossible* when the primary operation is comparison of pairs of elements

# Permutations

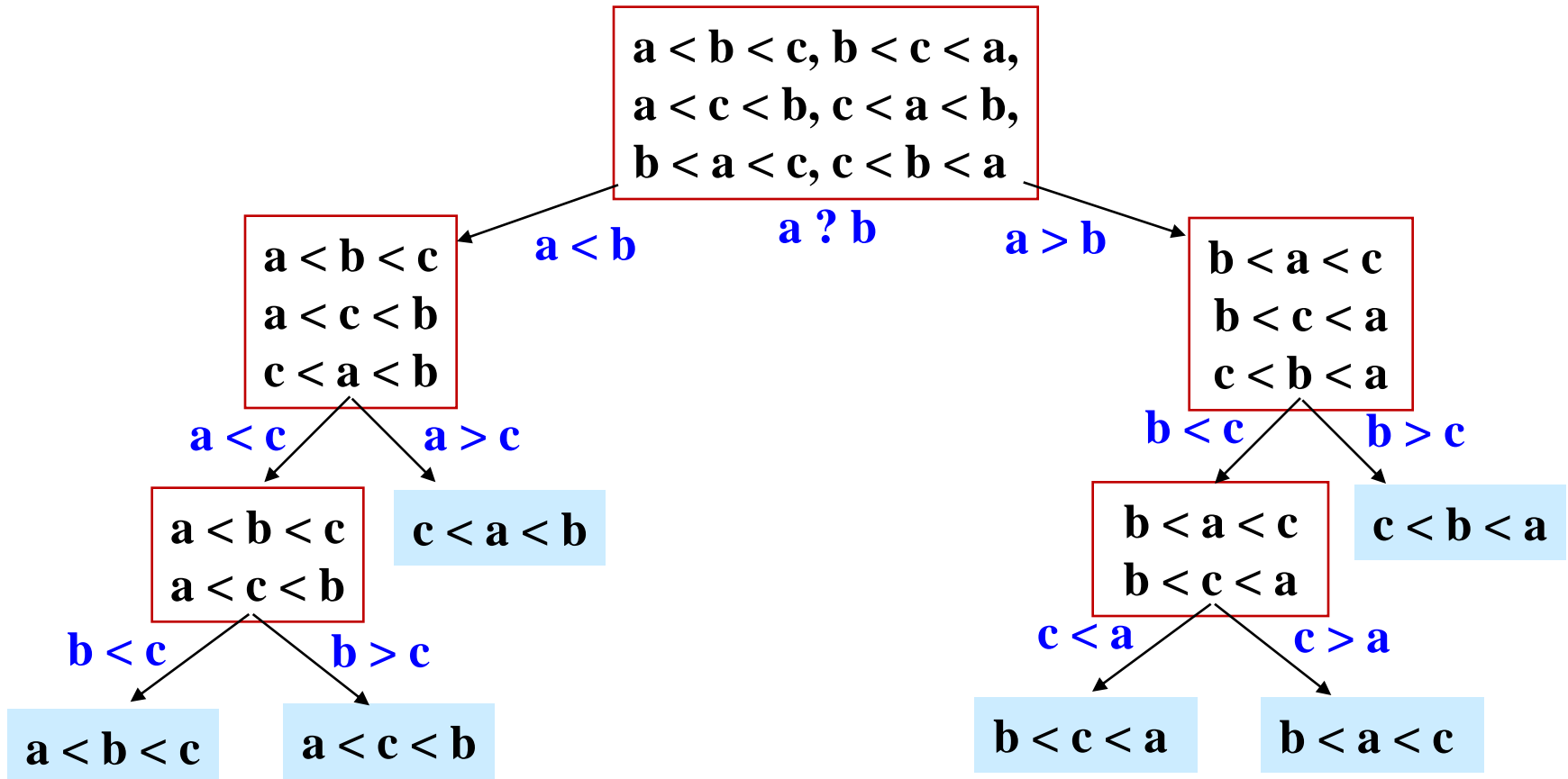
- Assume we have  $n$  elements to sort
  - And for simplicity, assume none are equal (i.e., no duplicates)
- How many permutations of the elements (possible orderings)?
- Example,  $n=3$ 
  - $a[0]<a[1]<a[2]$     $a[0]<a[2]<a[1]$     $a[1]<a[0]<a[2]$
  - $a[1]<a[2]<a[0]$     $a[2]<a[0]<a[1]$     $a[2]<a[1]<a[0]$

6 possible orderings
- In general,  $n$  choices for first,  $n-1$  for next,  $n-2$  for next, etc.
  - $n(n-1)(n-2)\dots(2)(1) = n!$  possible orderings

# *Representing Every Comparison Sort*

- Algorithm must “find” the right answer among  $n!$  possible answers
- Starts “knowing nothing” and gains information with each comparison
  - Intuition is that each comparison can, at best, eliminate half of the remaining possibilities
- Can represent this process as a decision tree
  - Nodes contain “remaining possibilities”
  - Edges are “answers from a comparison”
  - This is not a data structure, it’s what our proof uses to represent “the most any algorithm could know”

# Decision Tree for $n = 3$

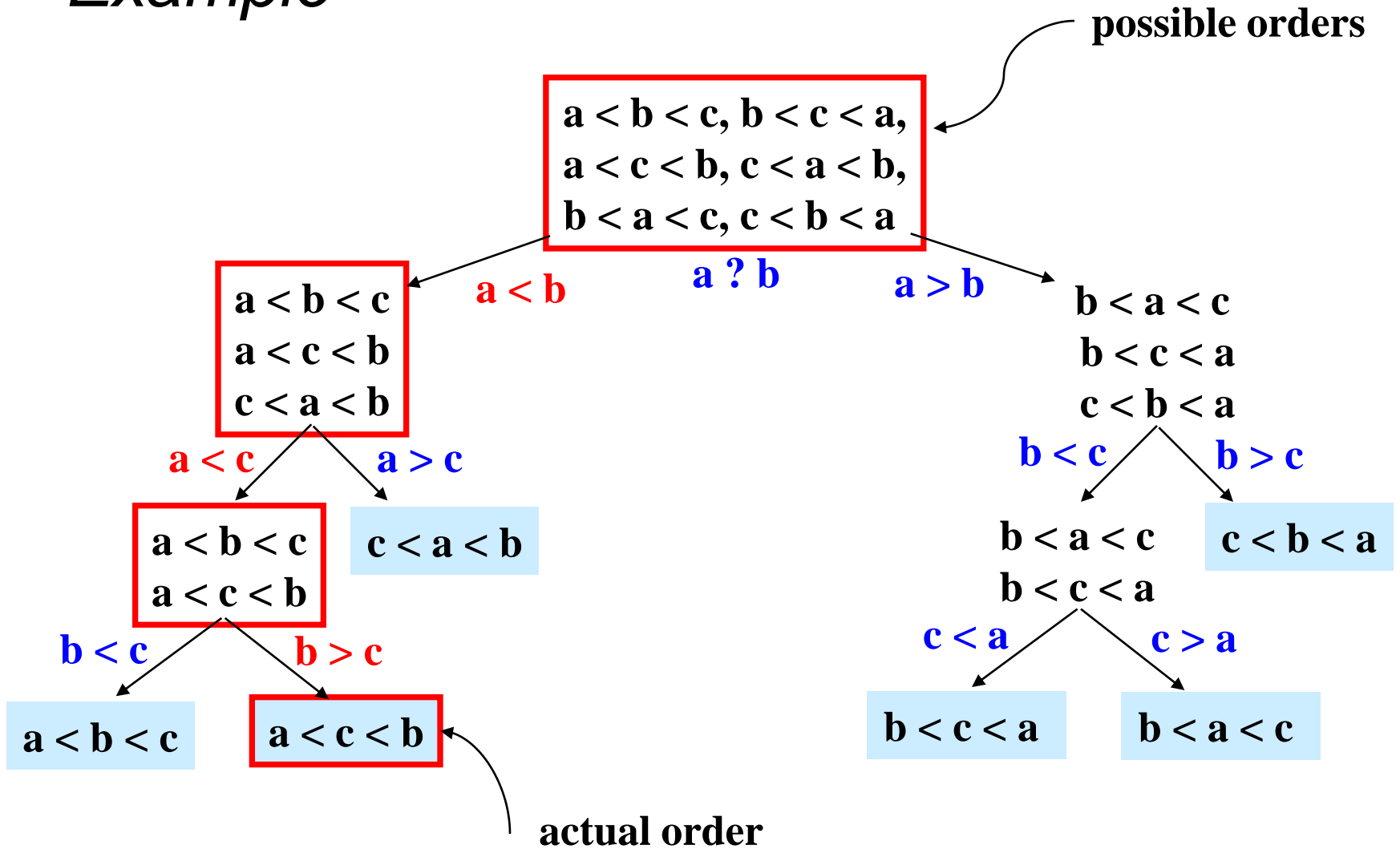


The leaves contain all the possible orderings of  $a, b, c$

# *What the Decision Tree Tells Us*

- A binary tree because each comparison has 2 outcomes
  - No duplicate elements
  - Assume algorithm not so dumb as to ask redundant questions
- Because any data is possible, any algorithm needs to ask enough questions to decide among all  $n!$  answers
  - Every answer is a leaf (no more questions to ask)
  - So the tree must be big enough to have  $n!$  leaves
  - Running any algorithm on any input will at best correspond to one root-to-leaf path in the decision tree
  - So no algorithm can have worst-case running time better than the height of the decision tree

# Example





# Where are We

**Proven:** No comparison sort can have worst-case better than:  
the height of a binary tree with  $n!$  leaves

- Turns out average-case is same asymptotically
- So how tall is a binary tree with  $n!$  leaves?

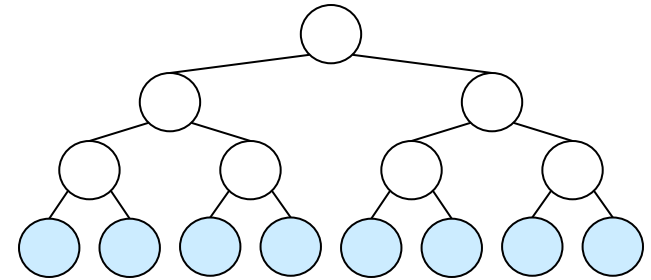
**Now:** Show that a binary tree with  $n!$  leaves has height  $\Omega(n \log n)$

- $n \log n$  is the lower bound, the height must be at least this
- It could be more (in other words, your comparison sorting algorithm could take longer than this, but can not be faster)
- Factorial function grows very quickly

Conclude that: (Comparison) Sorting is  $\Omega(n \log n)$

- This is an amazing computer-science result: proves all the clever programming in the world can't sort in linear time!

# Lower Bound on Height



- The height of a binary tree with  $L$  leaves is at least  $\lceil \log_2 L \rceil$
- So the height of our decision tree,  $h$ :

$$\begin{aligned}
 h &\geq \lceil \log_2 (n!) \rceil && \text{property of binary trees} \\
 &= \lceil \log_2 (n \cdot (n-1) \cdot (n-2) \cdots (2)(1)) \rceil && \text{definition of factorial} \\
 &= \lceil \log_2 n + \log_2 (n-1) + \dots + \log_2 1 \rceil && \text{property of logarithms} \\
 &\geq \lceil \log_2 n + \log_2 (n-1) + \dots + \log_2 (n/2) \rceil && \text{keep first } n/2 \text{ terms} \\
 &\geq (n/2) \lceil \log_2 (n/2) \rceil && \text{each of the } n/2 \text{ terms left is } \geq \lceil \log_2 (n/2) \rceil \\
 &\geq (n/2)(\lceil \log_2 n \rceil - \lceil \log_2 2 \rceil) && \text{property of logarithms} \\
 &\geq (1/2)n \lceil \log_2 n \rceil - (1/2)n && \text{arithmetic} \\
 &= \Omega(n \log n)
 \end{aligned}$$

# *The Big Picture*

**Simple  
algorithms:  
 $O(n^2)$**

**Insertion sort**  
**Selection sort**  
**Shell sort**  
...

**Fancier  
algorithms:  
 $O(n \log n)$**

**Heap sort**  
**Merge sort**  
**Quick sort (avg)**  
...

**Comparison  
lower bound:  
 $\Omega(n \log n)$**

**Specialized  
algorithms:  
 $O(n)$**

**Bucket sort**  
**Radix sort**

**Handling  
huge data  
sets**

**External  
sorting**

# BucketSort (a.k.a. BinSort)

- If all values to be sorted are known to be integers between 1 and  $K$  (or any small range),
  - Create an array of size  $K$
  - Put each element in its proper **bucket (a.k.a. bin)**
  - *If* data is only integers, no need to store anything more than a *count* of how times that bucket has been used
- Output result via linear pass through array of buckets

count array	
1	
2	
3	
4	
5	

Example:

$K=5$

Input: (5,1,3,4,3,2,1,1,5,4,5)

Output:

# BucketSort (a.k.a. BinSort)

- If all values to be sorted are known to be integers between 1 and  $K$  (or any small range),
  - Create an array of size  $K$
  - Put each element in its proper **bucket (a.k.a. bin)**
  - *If* data is only integers, no need to store anything more than a *count* of how times that bucket has been used
- Output result via linear pass through array of buckets

count array	
1	3
2	1
3	2
4	2
5	3

Example:

$K=5$

Input (5,1,3,4,3,2,1,1,5,4,5)

Output:

# BucketSort (a.k.a. BinSort)

- If all values to be sorted are known to be integers between 1 and  $K$  (or any small range),
  - Create an array of size  $K$
  - Put each element in its proper **bucket (a.k.a. bin)**
  - *If* data is only integers, no need to store anything more than a *count* of how times that bucket has been used
- Output result via linear pass through array of buckets

count array	
1	3
2	1
3	2
4	2
5	3

Example:

$K=5$

Input (5,1,3,4,3,2,1,1,5,4,5)

Output: 1,1,1,2,3,3,4,4,5,5,5

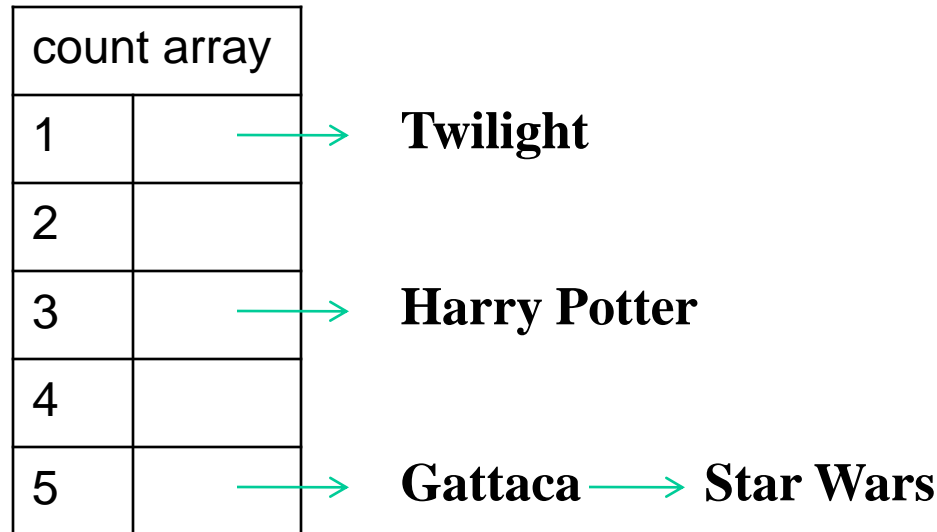
What is the running time?

# Analyzing Bucket Sort

- Overall:  $O(n+K)$ 
  - Linear in  $n$ , but also linear in  $K$
  - $\Omega(n \log n)$  lower bound does not apply because this is not a comparison sort
- Good when  $K$  is smaller (or not much larger) than  $n$ 
  - Do not spend time doing comparisons of duplicates
- Bad when  $K$  is much larger than  $n$ 
  - Wasted space; wasted time during final linear  $O(K)$  pass
- For data in addition to integer keys, use list at each bucket

# Bucket Sort with Data

- For data in addition to integer keys, use list at each bucket



- Bucket sort illustrates a more general trick
  - Imagine a heap for a small range of integer priorities



# Radix Sort

- Radix = “the base of a number system”
  - Examples will use 10 because we are familiar with that
  - In implementations use larger numbers
    - For example, for ASCII strings, might use 128
- Idea:
  - Bucket sort on one digit at a time
    - Number of buckets = radix
    - Starting with *least* significant digit, sort with Bucket Sort
    - Keeping sort *stable*
  - Do one pass per digit
  - After  $k$  passes, the last  $k$  digits are sorted
- Aside: Origins go back to the 1890 U.S. census

# Example: Radix Sort: Pass #1

Bucket sort  
by 1's digit

Input data

478  
537  
9  
721  
3  
38  
123  
67

0	1	2	3	4	5	6	7	8	9
	72 <u>1</u>		<u>3</u> 12 <u>3</u>				53 <u>7</u> 6 <u>7</u>	47 <u>8</u> 3 <u>8</u>	<u>9</u>

After 1<sup>st</sup> pass

721  
3  
123  
537  
67  
478  
38  
9

This example uses  $B=10$  and base 10 digits for simplicity of demonstration. Larger bucket counts should be used in an actual implementation.

# Example: Radix Sort: Pass #2

After 1<sup>st</sup> pass

721  
3  
123  
537  
67  
478  
38  
9

Bucket sort  
by 10's digit

0	1	2	3	4	5	6	7	8	9
<u>0</u> 3		<u>7</u> 21	<u>5</u> 37			<u>6</u> 7	<u>4</u> 78		
<u>0</u> 9		<u>1</u> 23	<u>3</u> 8						

After 2<sup>nd</sup> pass

3  
9  
721  
123  
537  
38  
67  
478

# Example: Radix Sort: Pass #3

After 2<sup>nd</sup> pass

3  
9  
721  
123  
537  
38  
67  
478

Bucket sort  
by 100's digit

0	1	2	3	4	5	6	7	8	9
<u>0</u> 03	<u>1</u> 23			<u>4</u> 78	<u>5</u> 37		<u>7</u> 21		

After 3<sup>rd</sup> pass

3  
9  
38  
67  
123  
478  
537  
721

**Invariant: after k passes the low order k digits are sorted.**

# *Analysis*

Input size:  $n$

Number of buckets = Radix:  $B$

Number of passes = “Digits”:  $P$

Work per pass is 1 bucket sort:  $O(B+n)$

Total work is  $O(P(B+n))$

Compared to comparison sorts, sometimes a win, but often not

- Example: Strings of English letters up to length 15
  - $15*(52 + n)$
  - This is less than  $n \log n$  only if  $n > 33,000$ 
    - Of course, cross-over point depends on constant factors of the implementations

# *Last Slide on Sorting*

- Simple  $O(n^2)$  sorts can be fastest for small  $n$ 
  - selection sort, insertion sort (which is linear for mostly-sorted)
  - good for “below a cut-off” to help divide-and-conquer sorts
- $O(n \log n)$  sorts
  - heap sort, in-place but not stable nor parallelizable
  - merge sort, not in place but stable and works as external sort
  - quick sort, in place but not stable and  $O(n^2)$  in worst-case
    - often fastest, but depends on costs of comparisons/copies
- $\Omega(n \log n)$  worst and average bound for comparison sorting
- Non-comparison sorts
  - Bucket sort good for small number of key values
  - Radix sort uses fewer buckets and more phases
- Best way to sort? It depends!



# CSE332: Data Abstractions

## Lecture 12: Graphs

James Fogarty

Winter 2012

# Graphs

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

- A **graph** is a pair

$$G = (V, E)$$

- A set of **vertices**, also known as **nodes**

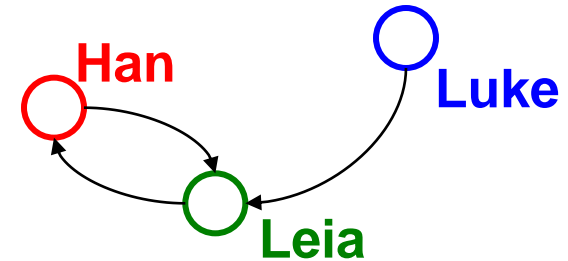
$$V = \{v_1, v_2, \dots, v_n\}$$

- A set of **edges**

$$E = \{e_1, e_2, \dots, e_m\}$$

- Each edge  $e_i$  is a pair of vertices  $(v_j, v_k)$
- An edge “connects” the vertices

- Graphs can be **directed** or **undirected**



$$V = \{\text{Han}, \text{Leia}, \text{Luke}\}$$

$$E = \{(\text{Luke}, \text{Leia}), (\text{Han}, \text{Leia}), (\text{Leia}, \text{Han})\}$$



# *An ADT?*

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

# *Some Graphs*

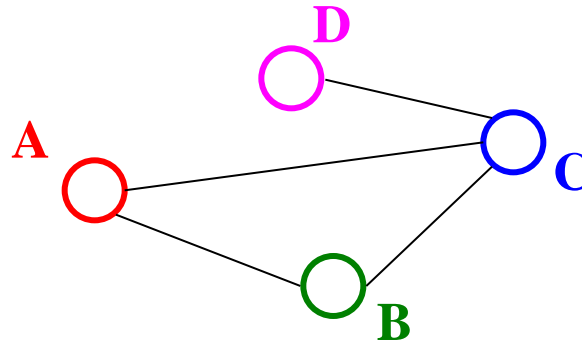
For each, what are the **vertices** and what are the **edges**?

- Web pages with links
- Facebook friends
- “Input data” for the Kevin Bacon game
- Methods in a program that call each other
- Road maps
- Airline routes
- Family trees
- Course pre-requisites
- ...

Core algorithms that work across such domains is why we are CSE

# Undirected Graphs

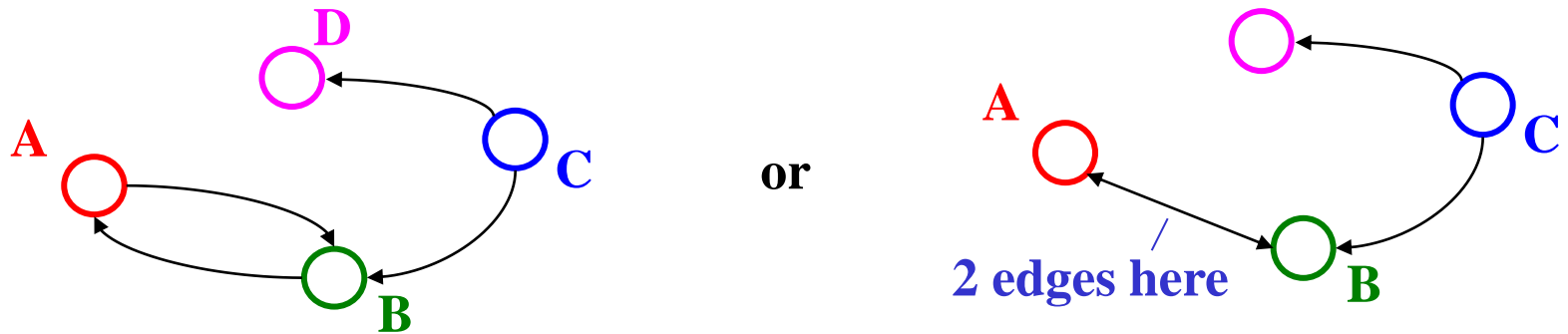
- In **undirected graphs**, edges have no specific direction
  - Edges are always “two-way”



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

# Directed Graphs

- In directed graphs (a.k.a. digraphs), edges have a direction



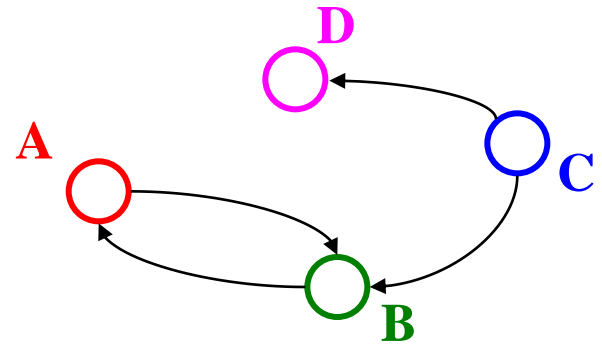
- Thus,  $(u, v) \in \mathbf{E}$  does *not* imply  $(v, u) \in \mathbf{E}$ .
  - Let  $(u, v) \in \mathbf{E}$  mean  $u \rightarrow v$
  - Call  $u$  the **source** and  $v$  the **destination**
- In-Degree** of a vertex: number of in-bound edges, i.e., edges where the vertex is the destination
- Out-Degree**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

# *Self-Edges, Connectedness*

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

# More Notation

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

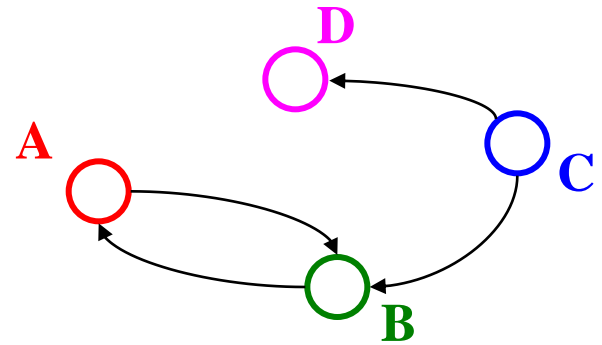


- $|V|$  is the number of vertices
- $|E|$  is the number of edges
  - Minimum?
  - Maximum for undirected?
  - Maximum for directed?
- If  $(u, v) \in E$ 
  - Then  $v$  is a **neighbor** of  $u$  (i.e.,  $v$  is **adjacent** to  $u$ )
  - Order matters for directed edges
    - $u$  is not **adjacent** to  $v$  unless  $(v, u) \in E$

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

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

# More Notation



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

- $|V|$  is the number of vertices
- $|E|$  is the number of edges
  - Minimum?  $0$
  - Maximum for undirected?  $|V|(|V+1|)/2 \in O(|V|^2)$
  - Maximum for directed?  $|V|^2 \in O(|V|^2)$   
(assuming self-edges allowed, else subtract  $|V|$ )
- If  $(u, v) \in E$ 
  - Then  $v$  is a **neighbor** of  $u$  (i.e.,  $v$  is **adjacent** to  $u$ )
  - Order matters for directed edges
    - $u$  is not **adjacent** to  $v$  unless  $(v, u) \in E$

# *Examples again*

Which would use **directed edges**?

Which would have **self-edges**?

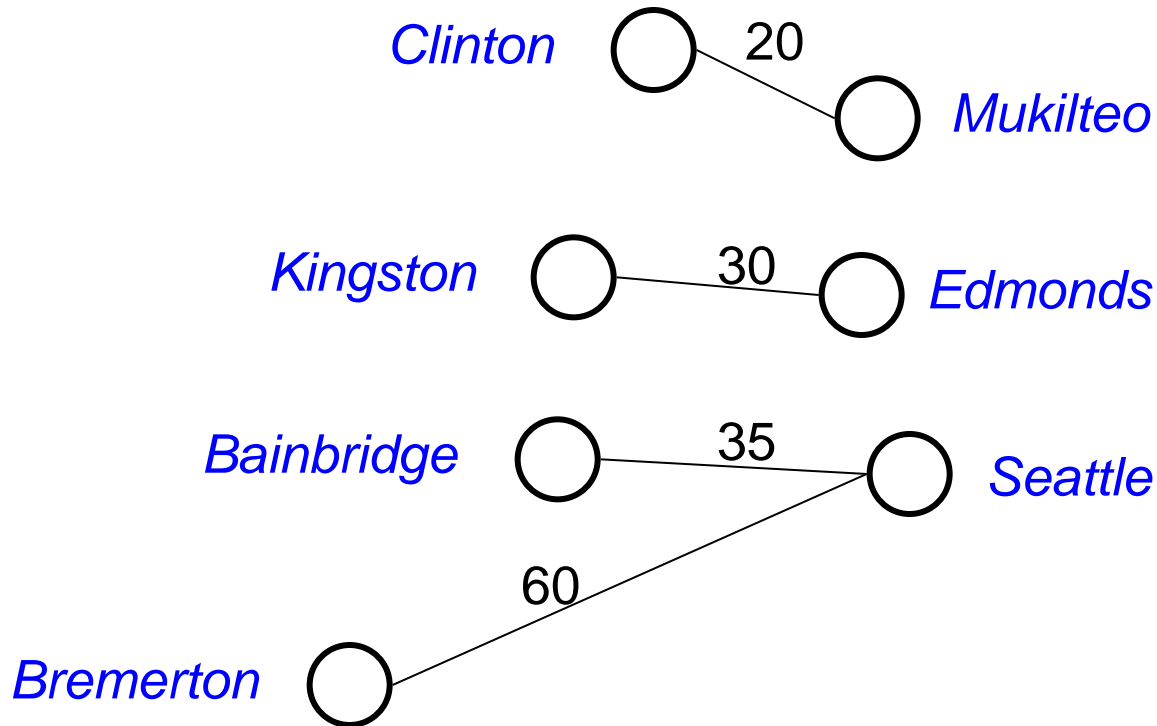
Which could have **0-degree nodes**?

- Web pages with links
- Facebook friends
- “Input data” for the Kevin Bacon game
- Methods in a program that call each other
- Road maps
- Airline routes
- Family trees
- Course pre-requisites
- ...



# Weighted Graphs

- In a weighted graph, each edge has a **weight** a.k.a. **cost**
  - Typically numeric (our examples use ints, but not required)
  - *Orthogonal* to whether graph is directed
  - Some graphs allow *negative weights*; many do not



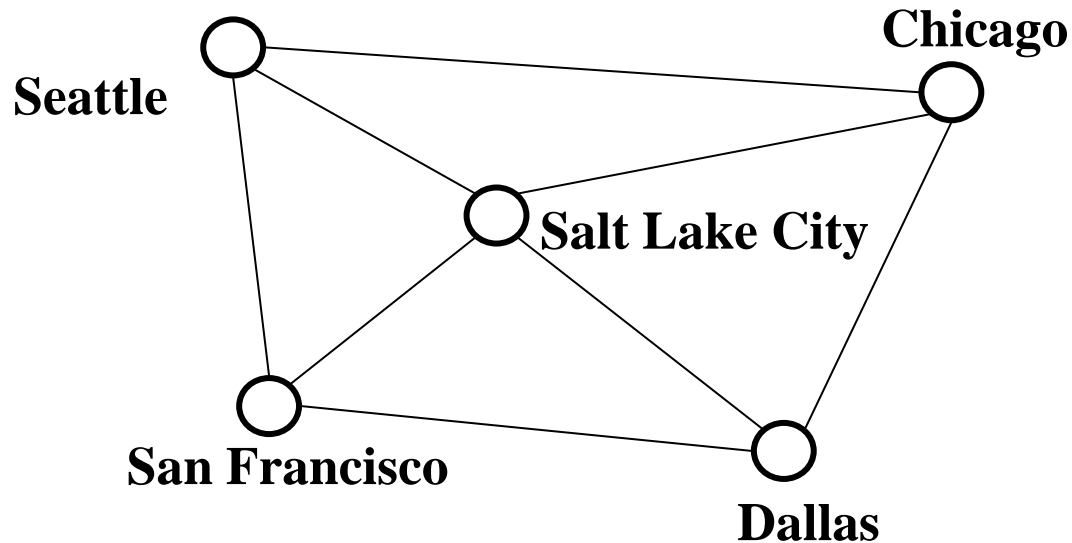
# *Examples*

What, if anything, might **weights** represent for each of these?  
Do **negative weights** make sense?

- Web pages with links
- Facebook friends
- “Input data” for the Kevin Bacon game
- Methods in a program that call each other
- Road maps
- Airline routes
- Family trees
- Course pre-requisites
- ...

# Paths and Cycles

- A **path** is a list of vertices  $[v_0, v_1, \dots, v_n]$  such that  $(v_i, v_{i+1}) \in E$  for all  $0 \leq i < n$ . We say “a path from  $v_0$  to  $v_n$ ”
- A **cycle** is a path that begins and ends at the same node ( $v_0 = v_n$ )



Example path (that also happens to be a cycle):

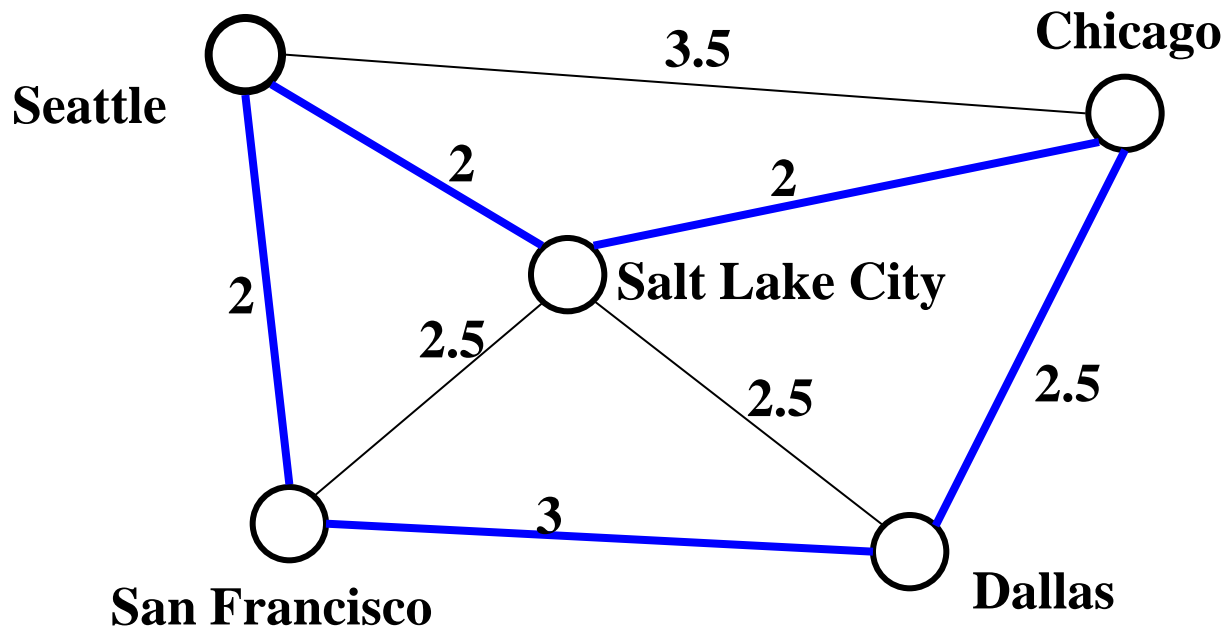
[Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle]

# Path Length and Cost

- **Path length:** Number of *edges* in a path
- **Path cost:** Sum of the weights of each edge

Example where

$P = [\text{Seattle}, \text{Salt Lake City}, \text{Chicago}, \text{Dallas}, \text{San Francisco}, \text{Seattle}]$



**length(P) = 5**  
**cost(P) = 11.5**

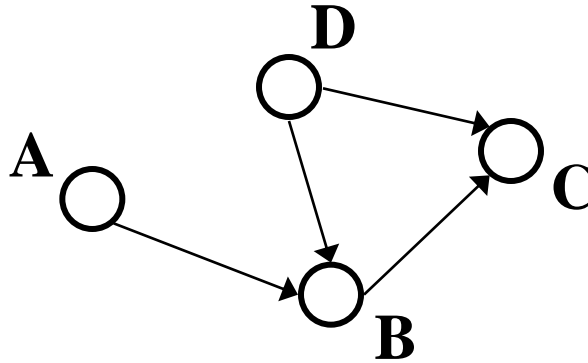
Length can sometimes be called “unweighted cost”

# *Simple Paths and Cycles*

- A **simple path** repeats no vertices, (except the first might be the last):  
[Seattle, Salt Lake City, San Francisco, Dallas]  
[Seattle, Salt Lake City, San Francisco, Dallas, Seattle]
- Recall, a **cycle** is a path that ends where it begins:  
[Seattle, Salt Lake City, San Francisco, Dallas, Seattle]  
[Seattle, Salt Lake City, Seattle, Dallas, Seattle]
- A **simple cycle** is a cycle and a simple path:  
[Seattle, Salt Lake City, San Francisco, Dallas, Seattle]

# *Paths and Cycles in Directed Graphs*

Example:



Is there a **path** from A to D?

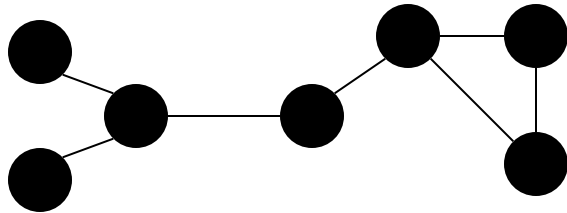
No

Does the graph contain any **cycles**?

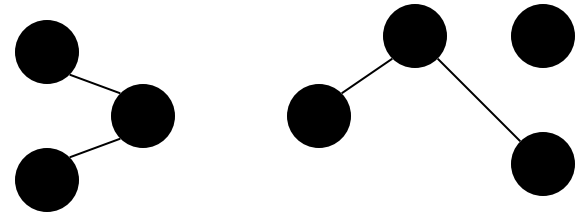
No

# Undirected Graph Connectivity

- An undirected graph is **connected** if for all pairs of vertices  $u, v$ , there exists a *path* from  $u$  to  $v$

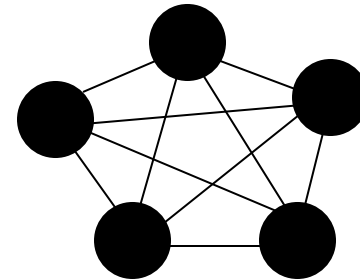


**Connected graph**



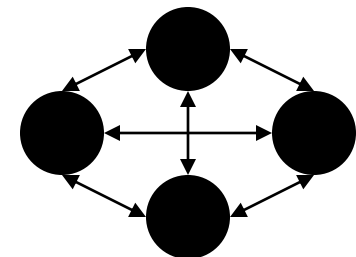
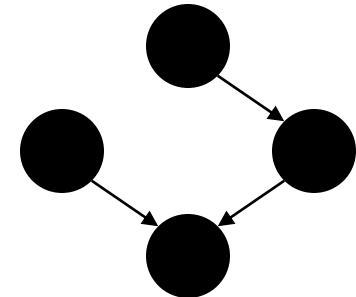
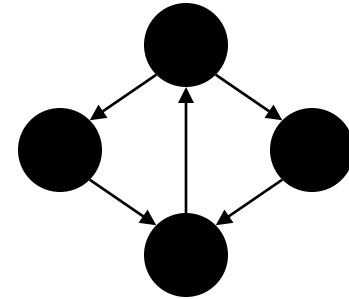
**Disconnected graph**

- An undirected graph is **complete**, a.k.a. **fully connected**, if for all pairs of vertices  $u, v$ , there exists an *edge* from  $u$  to  $v$



# Directed Graph Connectivity

- A directed graph is **strongly connected** if there is a path from every vertex to every other vertex
- A directed graph is **weakly connected** if there is a path from every vertex to every other vertex *ignoring direction of edges*
- A direct graph is **complete**, a.k.a. **fully connected**, if for all pairs of vertices  $u, v$ , there exists an *edge* from  $u$  to  $v$





# *Examples*

For undirected graphs: **connected?**

For directed graphs: **strongly connected?** **weakly connected?**

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

# *Trees as Graphs*

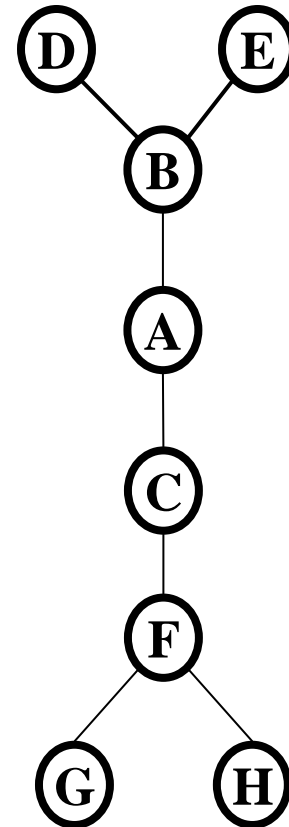
When talking about graphs,  
we say a **tree** is a graph that is:

- undirected
- acyclic
- connected

So all trees are graphs,  
but not all graphs are trees

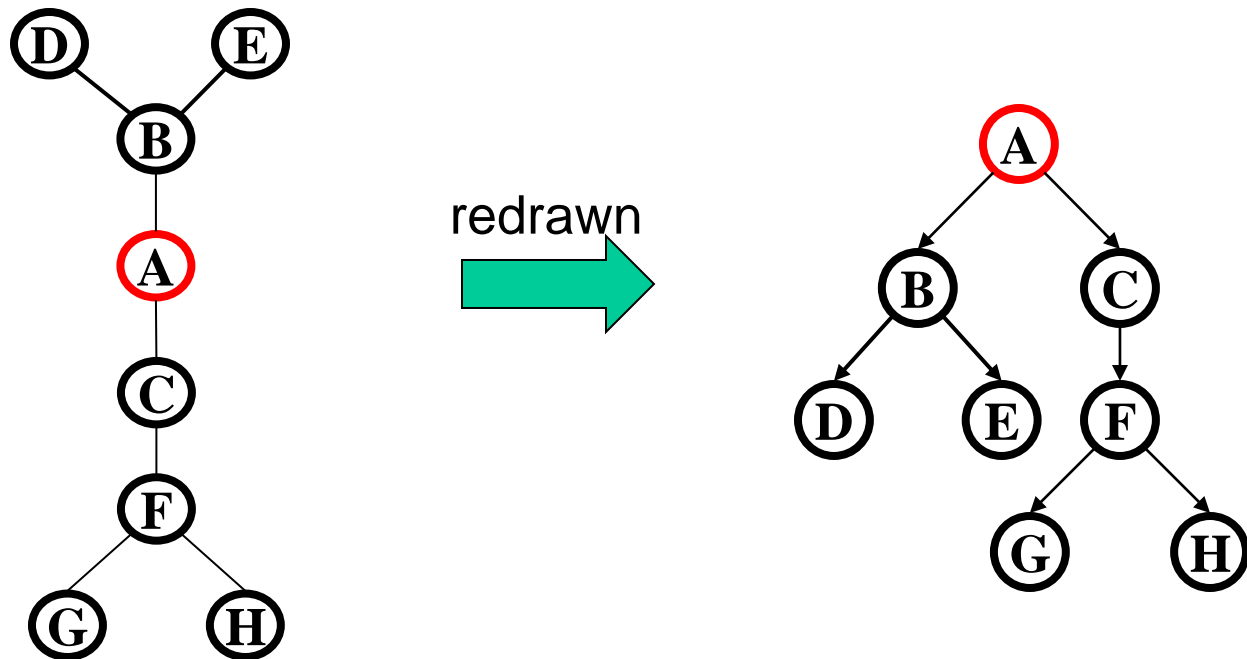
How does this relate to  
the trees we know and love?

Example:



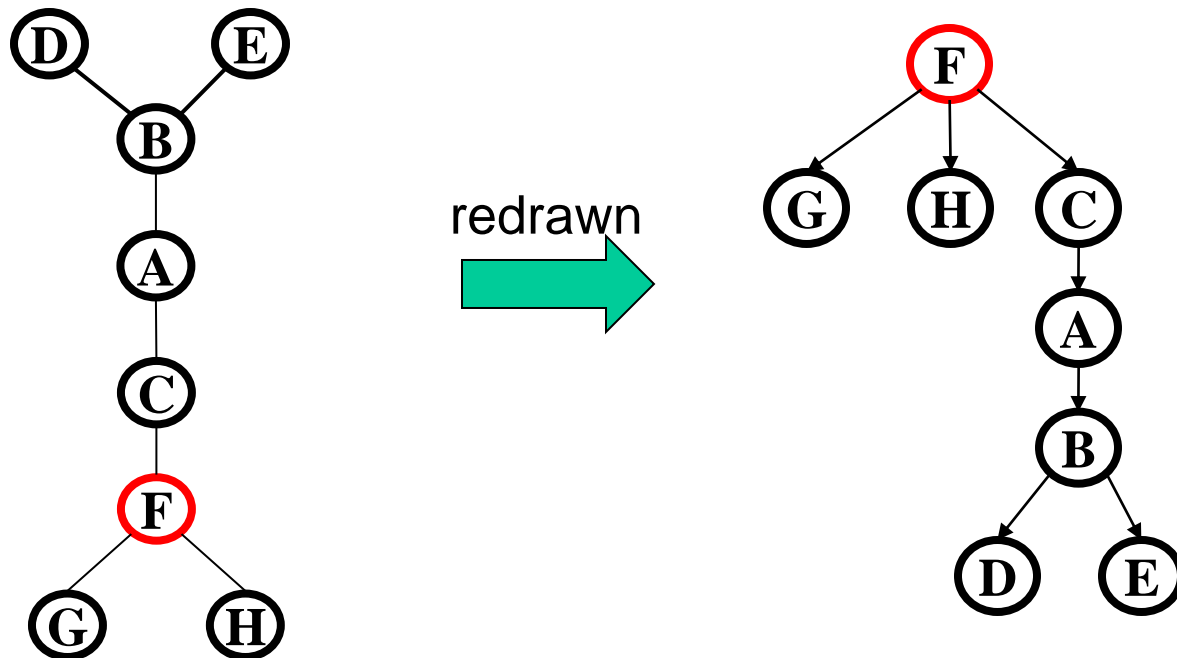
# Rooted Trees

- We are more accustomed to **rooted trees** where:
  - We identify a unique **root**
  - We think of edges as directed: parent to children
- Given a tree, picking a root gives a unique rooted tree
  - The tree is simply drawn differently and with undirected edges



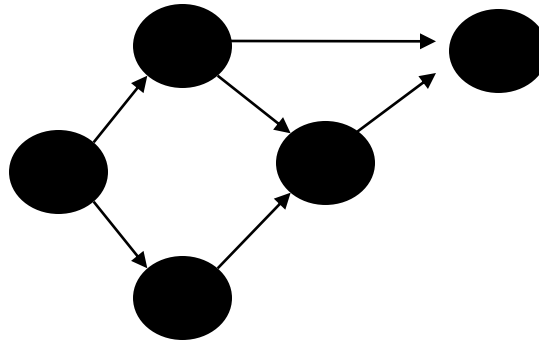
# Rooted Trees

- We are more accustomed to **rooted trees** where:
  - We identify a unique **root**
  - We think of edges as directed: parent to children
- Given a tree, picking a root gives a unique rooted tree
  - The tree is simply drawn differently and with undirected edges

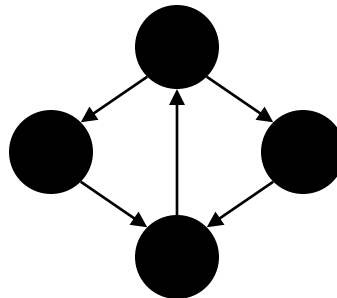


# Directed Acyclic Graphs (DAGs)

- A **DAG** is a directed graph with no directed cycles
  - Every rooted directed tree is a DAG
  - But not every DAG is a rooted directed tree



- Every DAG is a directed graph
- But not every directed graph is a DAG



# *Examples*

Which of our **directed**-graph examples do you expect to be a **DAG**?

- Web pages with links
- “Input data” for the Kevin Bacon game
- Methods in a program that call each other
- Airline routes
- Family trees
- Course pre-requisites
- ...

# Density / Sparsity

- Recall: In an undirected graph,  $0 \leq |E| < |V|^2$
- Recall: In a directed graph,  $0 \leq |E| \leq |V|^2$
- So for any graph,  $|E|$  is  $O(|V|^2)$
- Another fact: If an undirected graph is *connected*, then  $|E| \geq |V|-1$
- Because  $|E|$  is often much smaller than its maximum size, we do not always approximate as  $|E|$  as  $O(|V|^2)$ 
  - This is a correct bound, it just is often not tight
  - If it is tight (i.e.,  $|E|$  is  $\Theta(|V|^2)$ ), we say the graph is **dense**
    - More sloppily, dense means “lots of edges”
  - If  $|E|$  is  $O(|V|)$  we say the graph is **sparse**
    - More sloppily, sparse means “most possible edges missing”

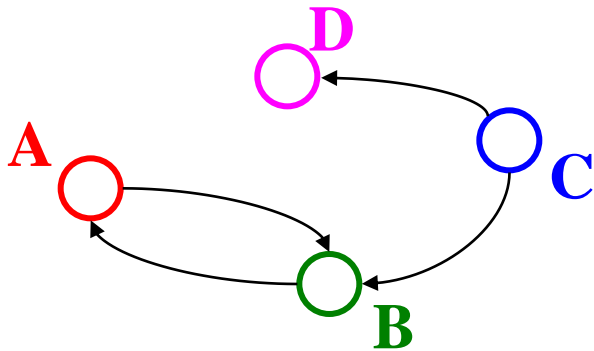
# *What's the Data Structure?*

- So graphs are really useful for lots of data and questions
  - For example, “what’s the lowest-cost path from  $x$  to  $y$ ”
- But we need a data structure that represents graphs
- Which data structure is “best” can depend on:
  - properties of the graph  
(e.g., dense versus sparse)
  - the common queries about the graph  
(e.g., “is  $(u, v)$  an edge?” vs “what are the neighbors of node  $u$ ?”)
- So we will discuss the two standard graph representations
  - **Adjacency Matrix** and **Adjacency List**
  - Different trade-offs, particularly time versus space



# Adjacency Matrix

- Assign each node a number from 0 to  $|V| - 1$
- A  $|V| \times |V|$  matrix of Booleans (or 0 vs. 1)
  - Then  $M[u][v] == \text{true}$   
means there is an edge from  $u$  to  $v$



	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

# Adjacency Matrix Properties

- Running time to:
  - Get a vertex's out-edges:
  - Get a vertex's in-edges:
  - Decide if some edge exists:
  - Insert an edge:
  - Delete an edge:
- Space requirements:
- Best for sparse or dense graphs?

	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

# Adjacency Matrix Properties

- Running time to:
  - Get a vertex's out-edges:  $O(|V|)$
  - Get a vertex's in-edges:  $O(|V|)$
  - Decide if some edge exists:  $O(1)$
  - Insert an edge:  $O(1)$
  - Delete an edge:  $O(1)$
- Space requirements:
  - $|V|^2$  bits
- Best for sparse or dense graphs?
  - Best for dense graphs

	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

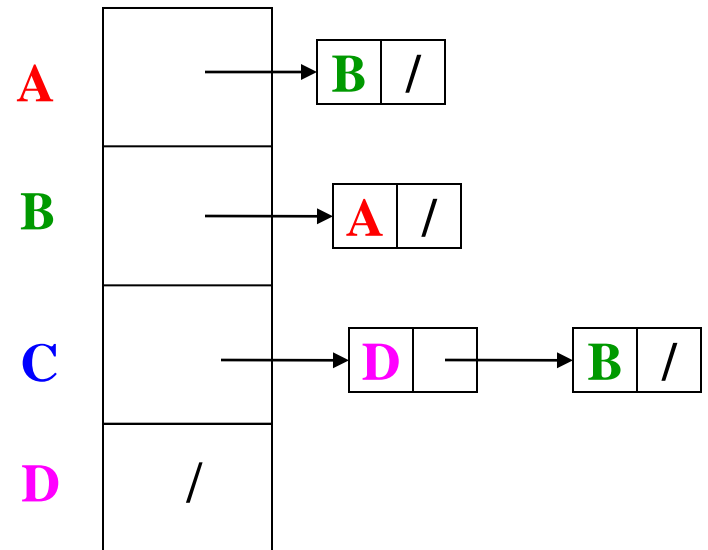
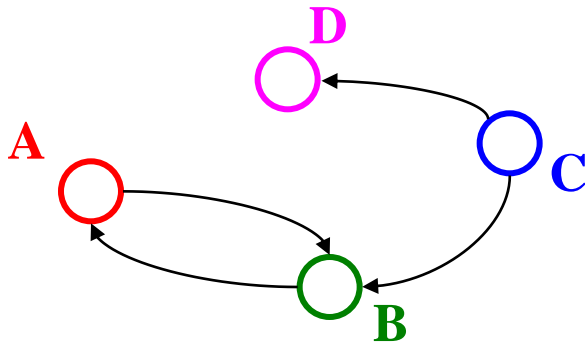
# Adjacency Matrix Properties

- How will the adjacency matrix vary for an **undirected graph**?
  - Undirected will be symmetric about diagonal axis
- How can we adapt the representation for **weighted graphs**?
  - Instead of a Boolean, store an number in each cell
  - Need some value to represent 'not an edge'
    - 0, -1, or some other value based on how you are using the graph

	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

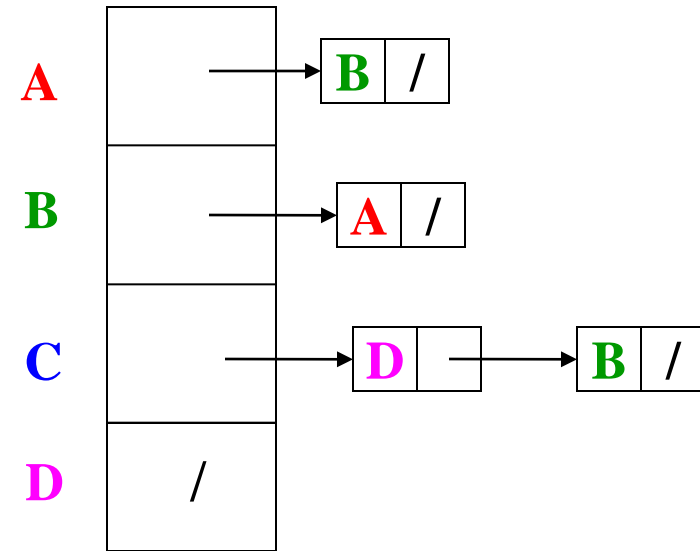
# Adjacency List

- Assign each node a number from 0 to  $|\mathbf{V}| - 1$
- An array of length  $|\mathbf{V}|$  in which each entry stores a list of all adjacent vertices (e.g., linked list)



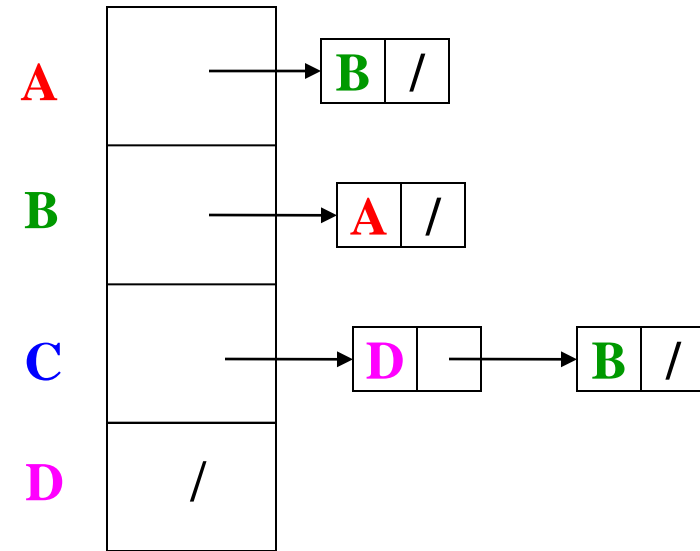
# Adjacency List Properties

- Running time to:
  - Get all of a vertex's out-edges:
  - Get all of a vertex's in-edges:
  - Decide if some edge exists:
  - Insert an edge:
  - Delete an edge:
- Space requirements:
  -
- Best for dense or sparse graphs?
  -



# Adjacency List Properties

- Running time to:
  - Get all of a vertex's out-edges:  
 $O(d)$  where  $d$  is out-degree of vertex
  - Get all of a vertex's in-edges:  
 $O(|E|)$  (but could keep a second adjacency list for this!)
  - Decide if some edge exists:  
 $O(d)$  where  $d$  is out-degree of source
  - Insert an edge:  $O(1)$  (unless you need to check if it's there)
  - Delete an edge:  $O(d)$  where  $d$  is out-degree of source
- Space requirements:
  - $O(|V|+|E|)$
- Best for dense or sparse graphs?
  - Best for sparse graphs, so usually just stick with linked lists

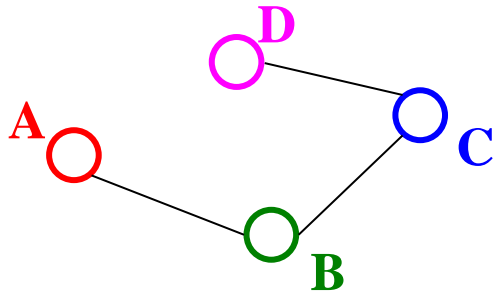


# Undirected Graphs

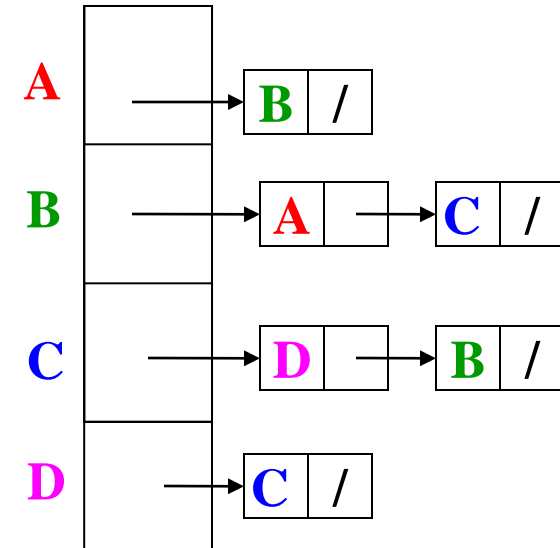
Adjacency matrices & adjacency lists both do fine for undirected graphs

- Matrix: Could save space by using only about half the array
  - How would you “get all neighbors”?
- Lists: Each edge in two lists to support efficient “get all neighbors”

Example:



	A	B	C	D
A	F	T	F	F
B	T	F	T	F
C	F	T	F	T
D	F	F	T	F







# CSE332: Data Abstractions

## Lecture 13: Graph Traversal / Topological Sort

James Fogarty

Winter 2012

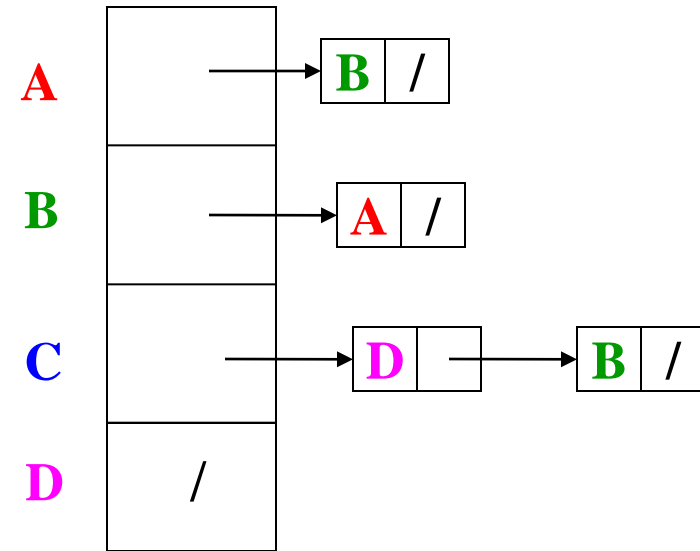
# Adjacency Matrix Properties

- Running time to:
  - Get a vertex's out-edges:  $O(|V|)$
  - Get a vertex's in-edges:  $O(|V|)$
  - Decide if some edge exists:  $O(1)$
  - Insert an edge:  $O(1)$
  - Delete an edge:  $O(1)$
- Space requirements:
  - $|V|^2$  bits
- Best for sparse or dense graphs?
  - Best for dense graphs

	A	B	C	D
A	F	T	F	F
B	T	F	F	F
C	F	T	F	T
D	F	F	F	F

# Adjacency List Properties

- Running time to:
  - Get all of a vertex's out-edges:  
 $O(d)$  where  $d$  is out-degree of vertex
  - Get all of a vertex's in-edges:  
 $O(|E|)$  (but could keep a second adjacency list for this!)
  - Decide if some edge exists:  
 $O(d)$  where  $d$  is out-degree of source
  - Insert an edge:  $O(1)$  (unless you need to check if it's there)
  - Delete an edge:  $O(d)$  where  $d$  is out-degree of source
- Space requirements:
  - $O(|V|+|E|)$
- Best for dense or sparse graphs?
  - Best for sparse graphs, so usually just stick with linked lists

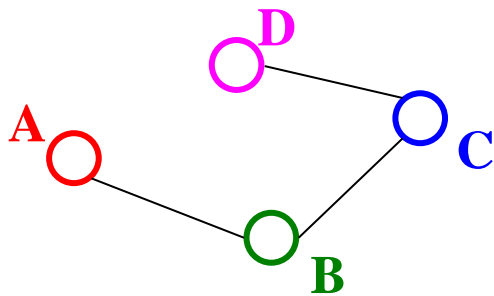


# Undirected Graphs

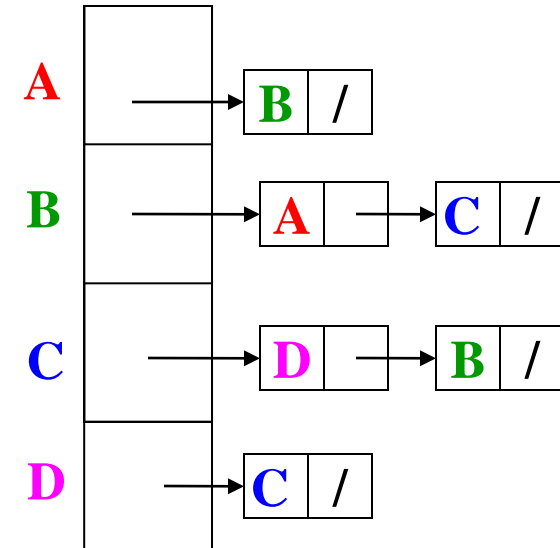
Adjacency matrices & adjacency lists both do fine for undirected graphs

- Matrix: Could save space by using only about half the array
  - How would you “get all neighbors”?
- Lists: Each edge in two lists to support efficient “get all neighbors”

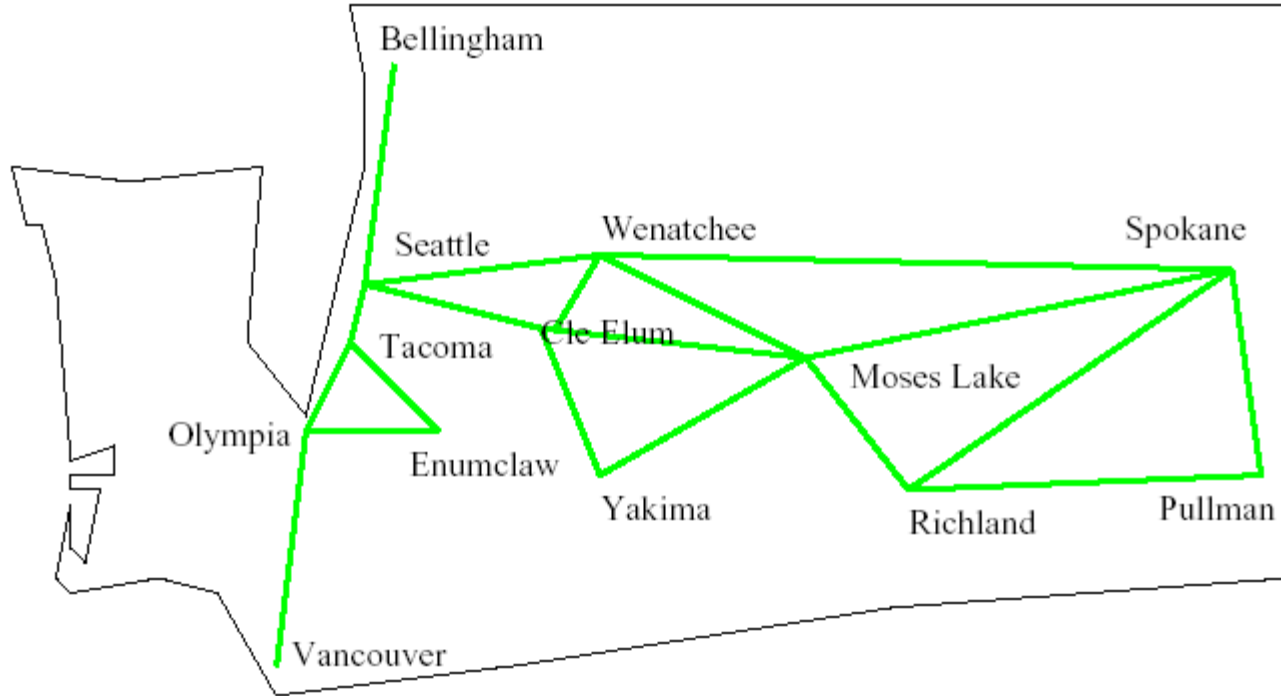
Example:



	A	B	C	D
A	F	T	F	F
B	T	F	T	F
C	F	T	F	T
D	F	F	T	F

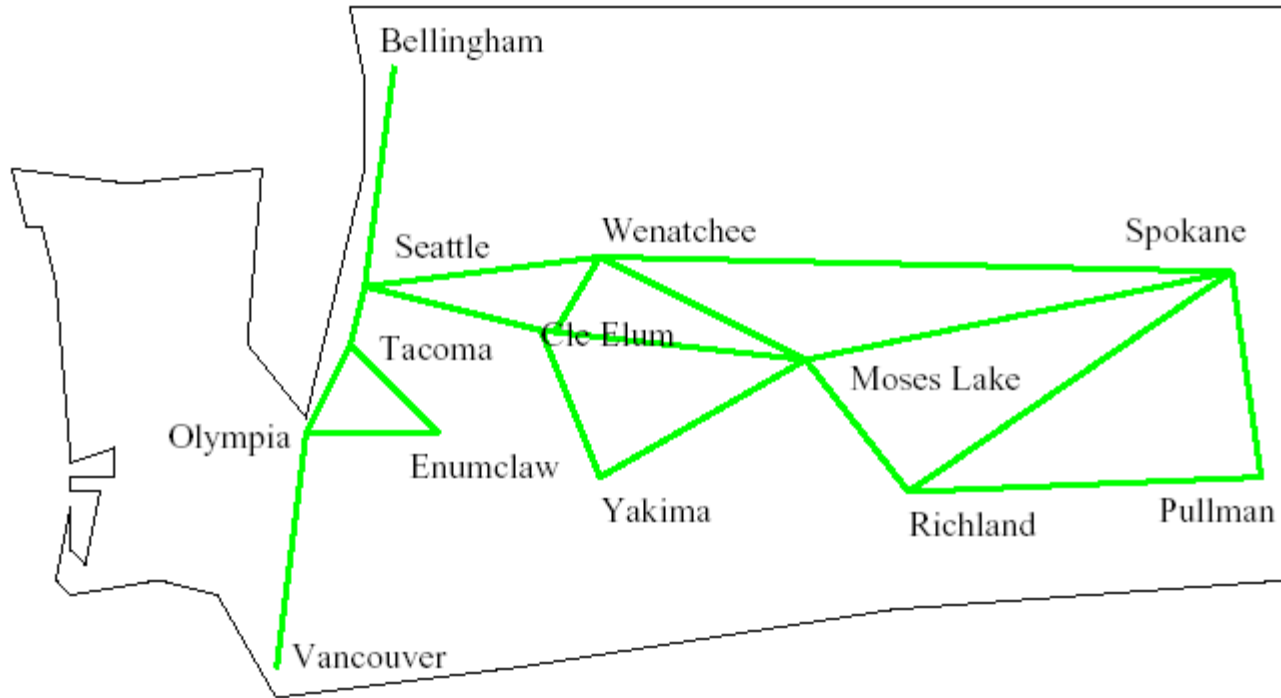


# *Some Applications: Moving Around Washington*



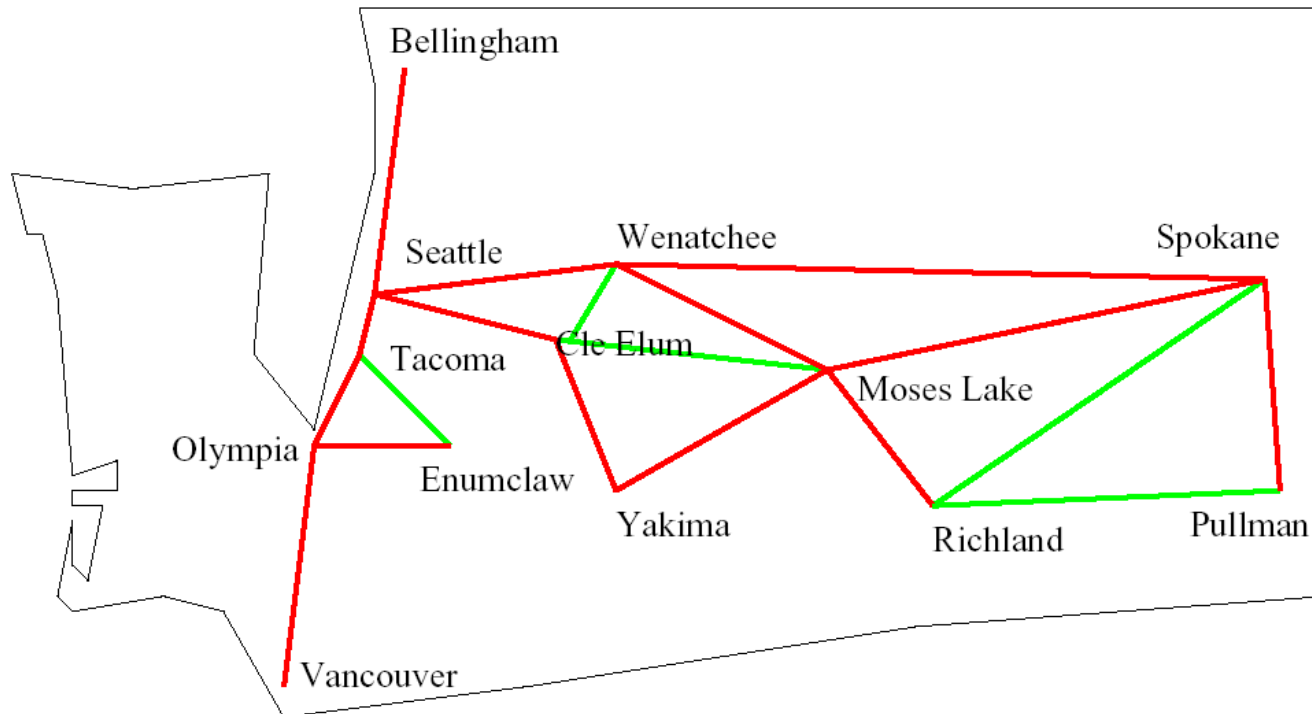
**What's the *shortest way* to get from Seattle to Pullman?**

# *Some Applications: Moving Around Washington*



**What's the *fastest way* to get from Seattle to Pullman?**

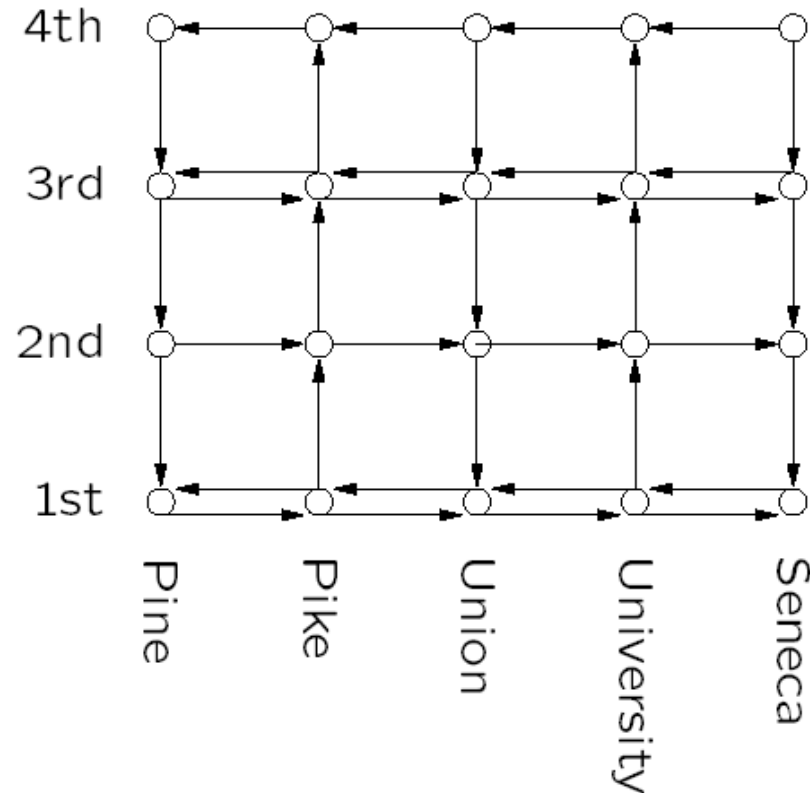
# *Some Applications: Reliability of Communication*



**If Wenatchee's phone exchange *goes down*,  
can Seattle still talk to Pullman?**

# *Some Applications:*

## *Bus Routes in Downtown Seattle*



**If we're at 3<sup>rd</sup> and Pine, how can we get to 1<sup>st</sup> and University using Metro?  
How about 4<sup>th</sup> and Seneca?**



# *Graph Traversals*

For an arbitrary graph and a starting node  $v$ ,  
find all nodes *reachable* from  $v$  (i.e., there exists a path)

- Possibly “do something” for each node
- e.g., print to output, set some field, return from iterator, etc.

Related Problems:

- Is an undirected graph connected?
- Is a directed graph weakly / strongly connected?
  - For strongly, need a cycle back to starting node

Basic Idea:

- Keep following nodes
- But “mark” nodes after visiting them, so the traversal terminates and processes each reachable node exactly once

# *Abstract Idea*

```
traverseGraph(Node start) {
    Set pending = emptySet();
    pending.add(start)
    mark start as visited
    while(pending is not empty) {
        next = pending.remove()
        for each node u adjacent to next
            if(u is not marked) {
                mark u
                pending.add(u)
            }
    }
}
```

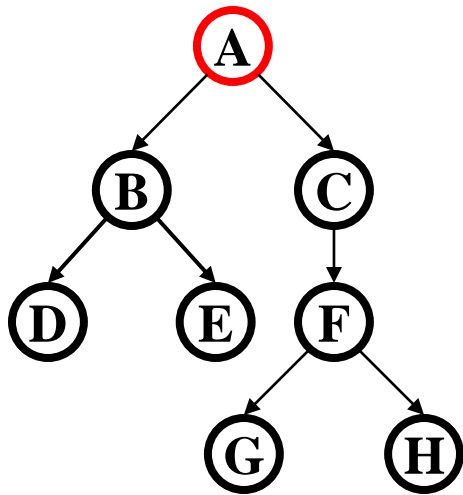
Why do we need to **mark** nodes?

# *Running Time and Options*

- Assuming add and remove are  $O(1)$ , entire traversal is  $O(|E|)$ 
  - Use an adjacency list representation
- The order we traverse depends entirely on add and remove
  - Popular choice: a stack “depth-first graph search” “DFS”
  - Popular choice: a queue “breadth-first graph search” “BFS”
- DFS and BFS are “big ideas” in computer science
  - Depth: recursively explore one part before going back to the other parts not yet explored
  - Breadth: Explore areas closer to the start node first

# Recursive DFS, Example with Tree

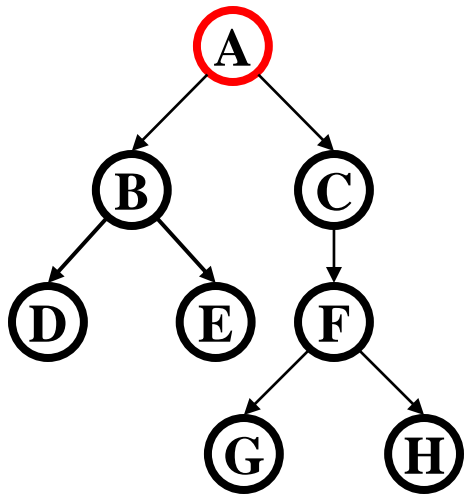
- A tree is a graph and DFS and BFS are particularly easy to “see”



```
DFS(Node start) {  
    mark and process start  
    for each node u adjacent to start  
        if u is not marked  
            DFS(u)  
}
```

- Order processed: A, B, D, E, C, F, G, H
- Exactly what we called a “pre-order traversal” for trees
  - The marking is because we support arbitrary graphs and we want to process each node exactly once

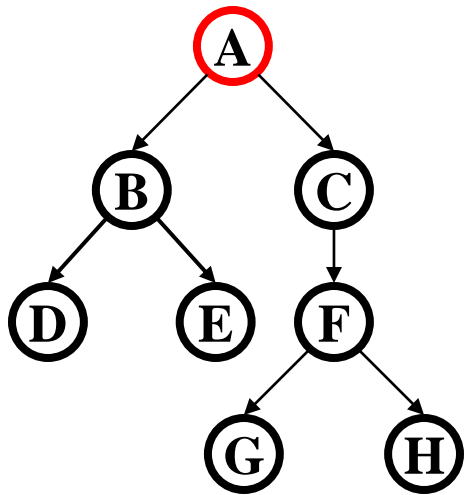
# DFS with Stack, Example with Tree



```
DFS2(Node start) {  
    initialize stack s to hold start  
    mark start as visited  
    while(s is not empty) {  
        next = s.pop() // and "process"  
        for each node u adjacent to next  
            if(u is not marked)  
                mark u and push onto s  
    }  
}
```

- Order processed: A, C, F, H, G, B, E, D
- A different but perfectly fine traversal

# BFS with Queue, Example with Tree



```
BFS(Node start) {  
    initialize queue q to hold start  
    mark start as visited  
    while(q is not empty) {  
        next = q.dequeue() // and "process"  
        for each node u adjacent to next  
            if(u is not marked)  
                mark u and enqueue onto q  
    }  
}
```

- Order processed: A, B, C, D, E, F, G, H
- A "level-order" traversal

# Comparison

- Breadth-first always finds shortest paths, i.e. “optimal solutions”
  - Better for “what is the shortest path from  $\mathbf{x}$  to  $\mathbf{y}$ ”
- But depth-first can use less space in finding a path
  - If *longest path* in the graph is  $\mathbf{p}$  and highest out-degree is  $\mathbf{d}$  then DFS stack never has more than  $\mathbf{d} \cdot \mathbf{p}$  elements
  - But a queue for BFS may hold  $O(|V|)$  nodes
- A third approach:
  - *Iterative deepening (IDFS)*:
    - Try DFS up to recursion of  $\mathbf{k}$  levels deep.
    - If that fails, increment  $\mathbf{k}$  and start the entire search over
  - Like BFS, finds shortest paths. Like DFS, less space.

# *Saving the Path*

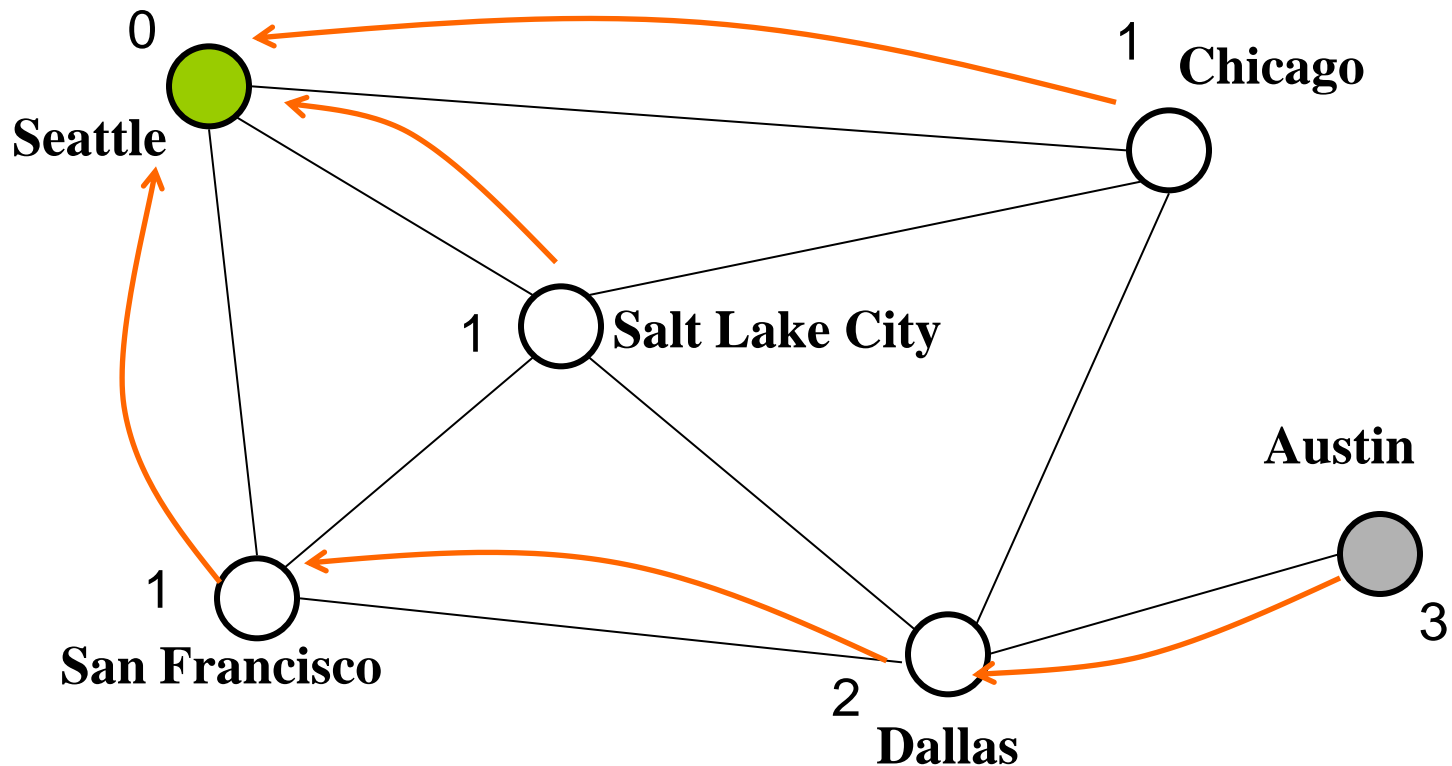
- Our graph traversals can answer the reachability question:
  - “Is there a path from node  $x$  to node  $y$ ?”
- But what if we want to actually output the path?
- Easy:
  - Instead of just “marking” a node, store the previous node along the path (when processing  $u$  causes us to add  $v$  to the search, set  $v.path$  field to be  $u$ )
  - When you reach the goal, follow `path` fields back to where you started (and then reverse the answer)



# Example using BFS

What is a path from Seattle to Austin

- Remember marked nodes are not re-enqueued
- Note shortest paths may not be unique

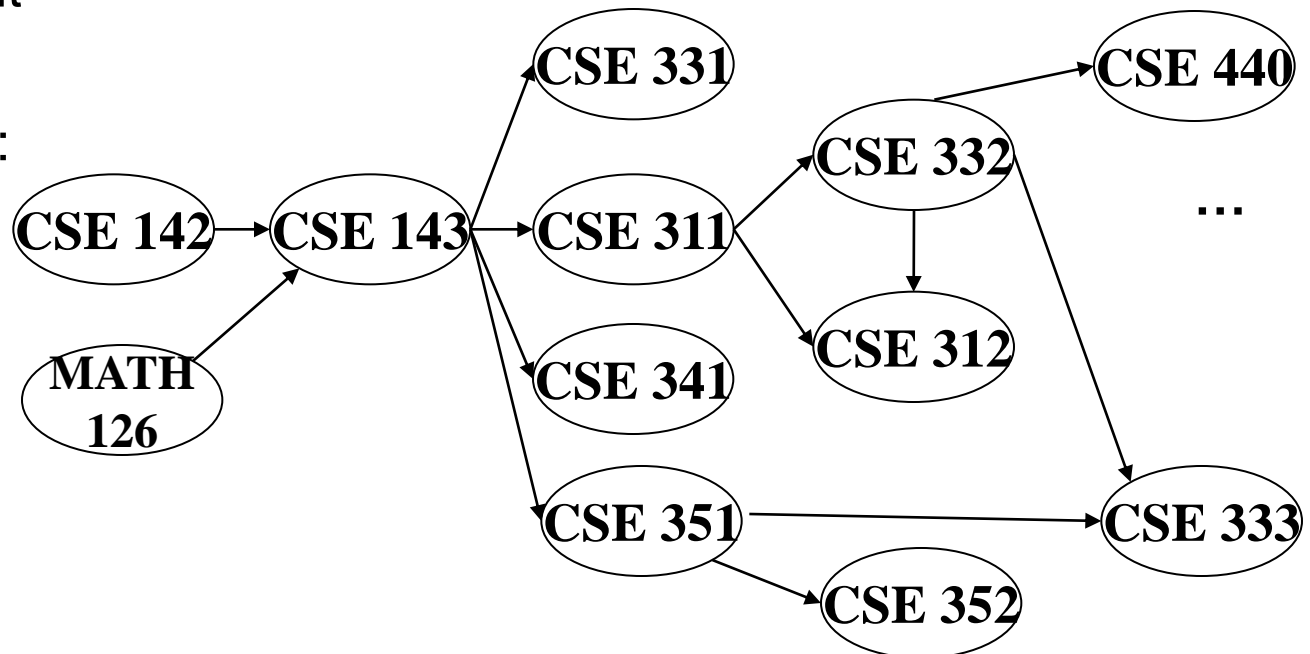


Disclaimer: Do not use for official advising purposes!  
(Implies that CSE 332 is a pre-req for CSE 312 – not true)

# Topological Sort

Problem: Given a DAG  $G = (V, E)$ , output all the vertices in order such that if no vertex appears before any other vertex that has an edge to it

Example input:



Example output:

142, 126, 143, 311, 331, 332, 312, 341, 351, 333, 440, 352

# *Questions and Comments*

- Why do we perform topological sorts only on DAGs?
  - Because a cycle means there is no correct answer
- Is there always a unique answer?
  - No, there can be 1 or more answers; depends on the graph
- What DAGs have exactly 1 answer?
  - Lists
- Terminology: A DAG represents a **partial order** and a topological sort produces a **total order** that is consistent with it

# *Uses*

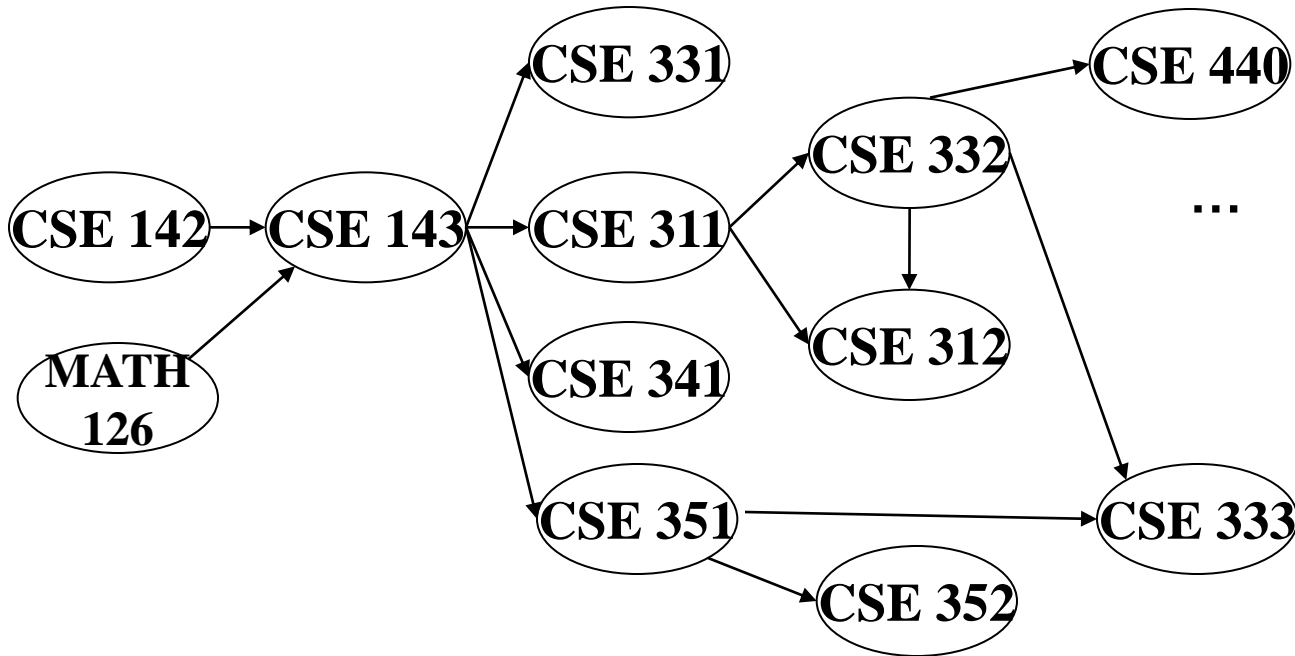
- Figuring out how to finish your degree
- Computing order in which to recompute cells in a spreadsheet
- Determining the order to compile files with dependencies
- In general, using a dependency graph to find an order of execution

# *A First Algorithm for Topological Sort*

1. Label each vertex with its in-degree
  - Think “write in a field in the vertex”
  - You could also do this with a data structure on the side
  
2. While there are vertices not yet output:
  - a) Choose a vertex  $\mathbf{v}$  labeled with in-degree of 0
  - b) Output  $\mathbf{v}$  and conceptually “remove it” from the graph
  - c) For each vertex  $\mathbf{u}$  adjacent to  $\mathbf{v}$ , **decrement in-degree** of  $\mathbf{u}$ 
    - (i.e.,  $\mathbf{u}$  such that  $(\mathbf{v}, \mathbf{u})$  in  $\mathbf{E}$ )

# Example

Output:



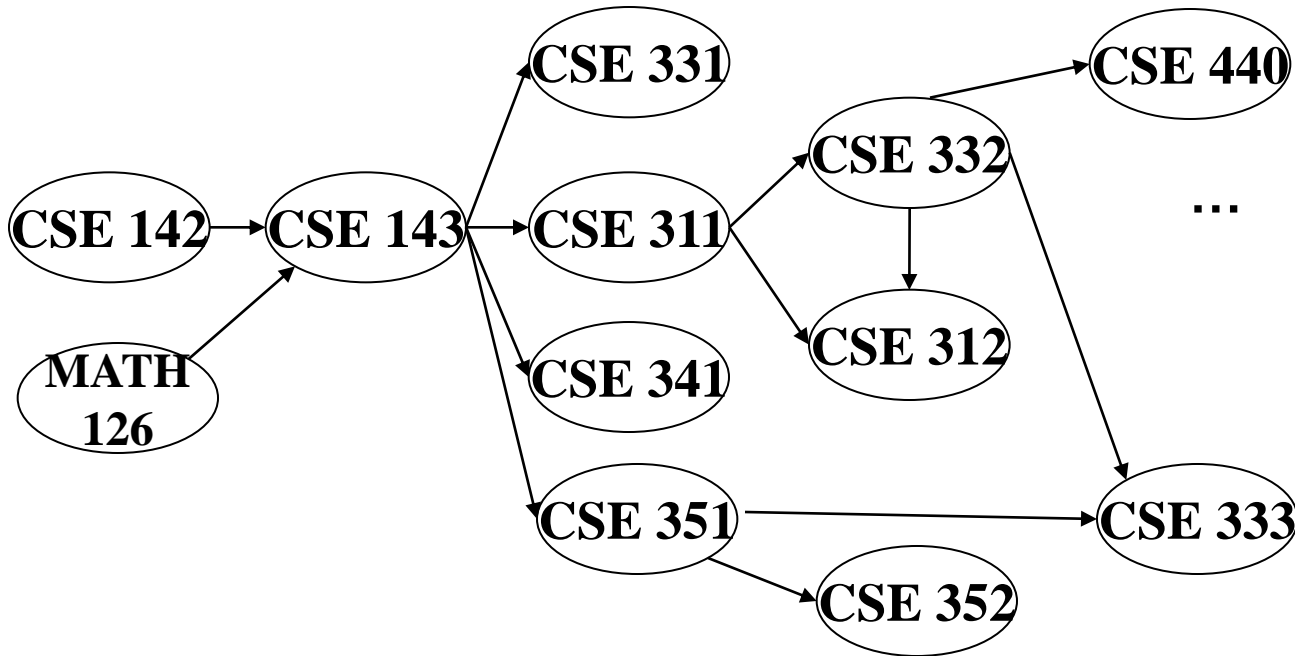
Node:            126 142 143 311 312 331 332 333 341 351 352 440

Removed?:

In-degree:

# Example

Output:



Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?												
In-degree:	0	0	2	1	2	1	1	2	1	1	1	1







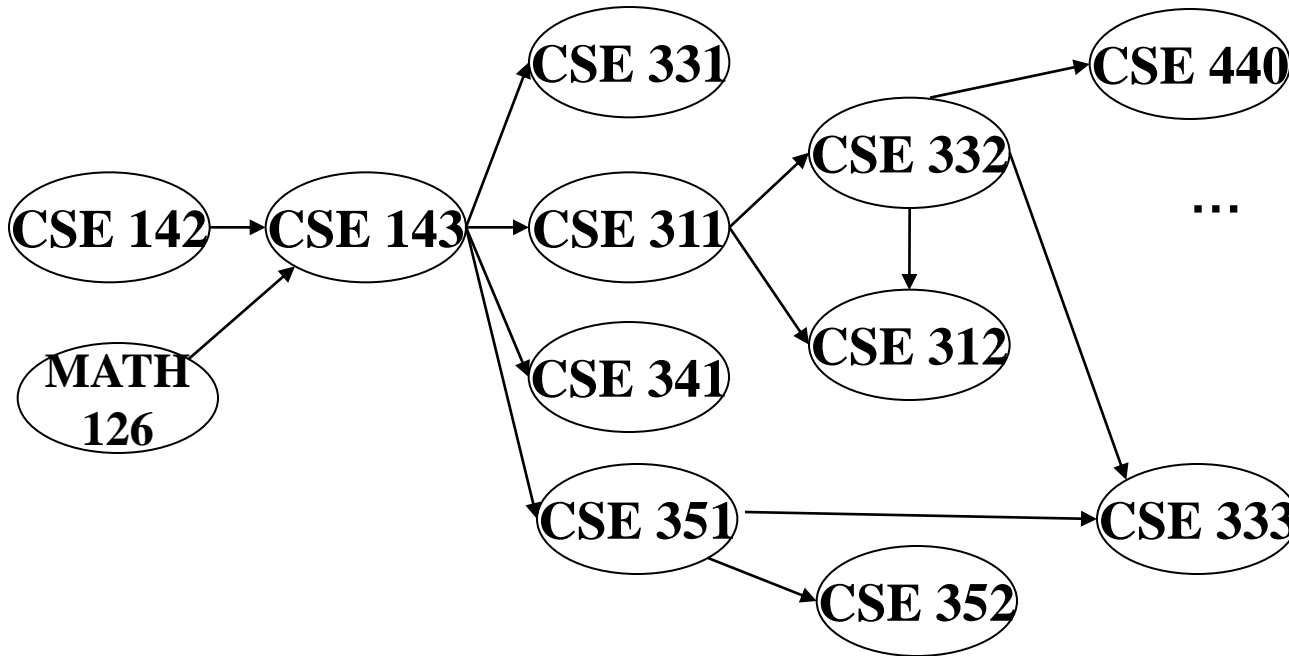






# Example

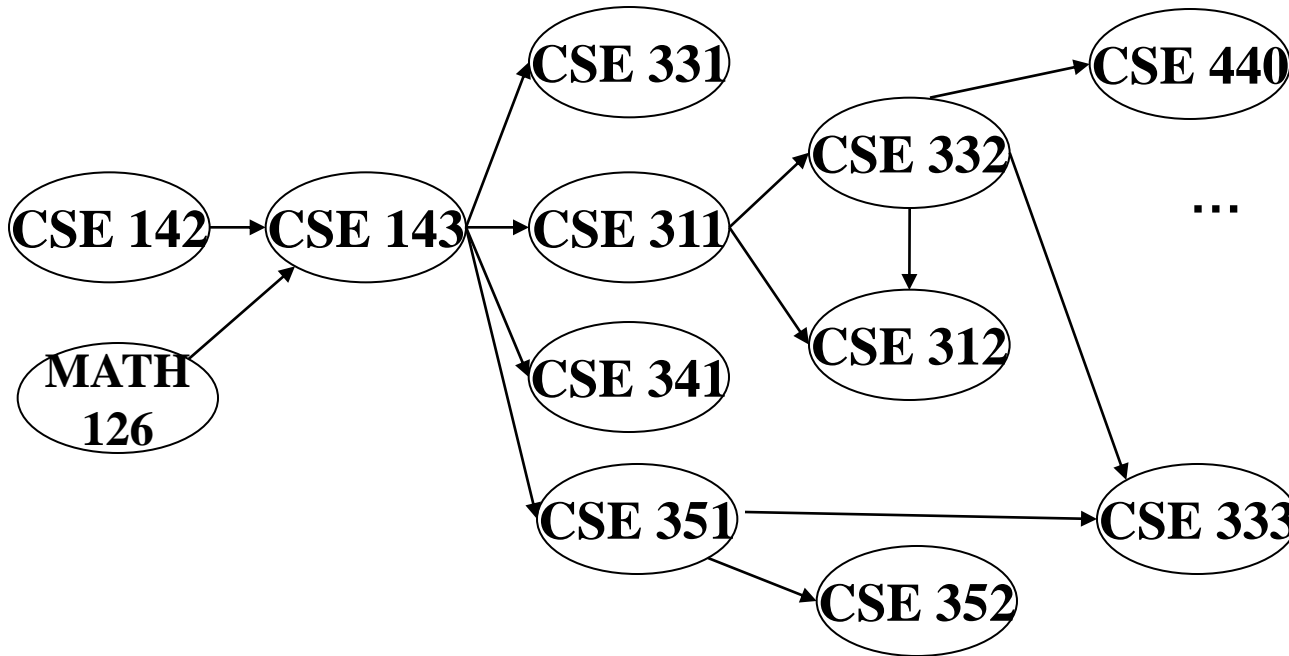
Output: 126  
 142  
 143  
 311  
 331  
 332



Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x		x	x					
In-degree:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0		0
			0		0							

# Example

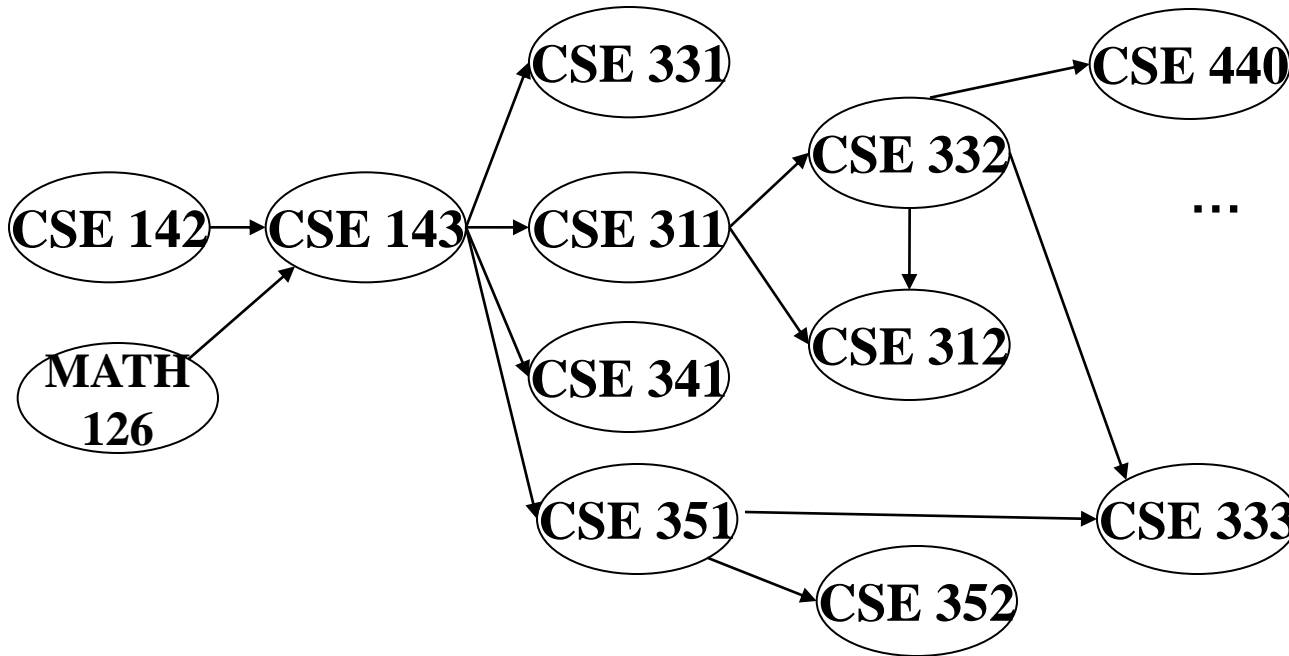
Output: 126  
 142  
 143  
 311  
 331  
 332  
 312



Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x					
In-degree:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0		0
			0		0							

# Example

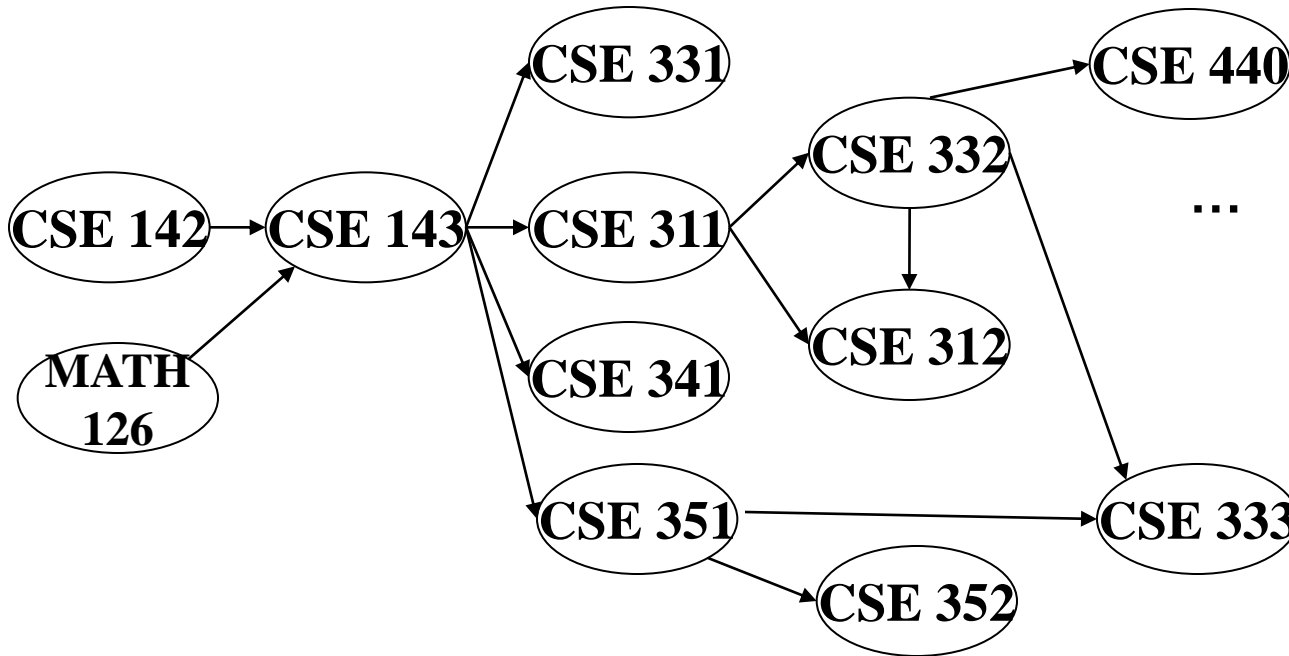
Output: 126  
 142  
 143  
 311  
 331  
 332  
 312  
 341



Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x		x			
In-degree:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0		0
			0		0							

# Example

Output: 126  
 142  
 143  
 311  
 331  
 332  
 312  
 341  
 351

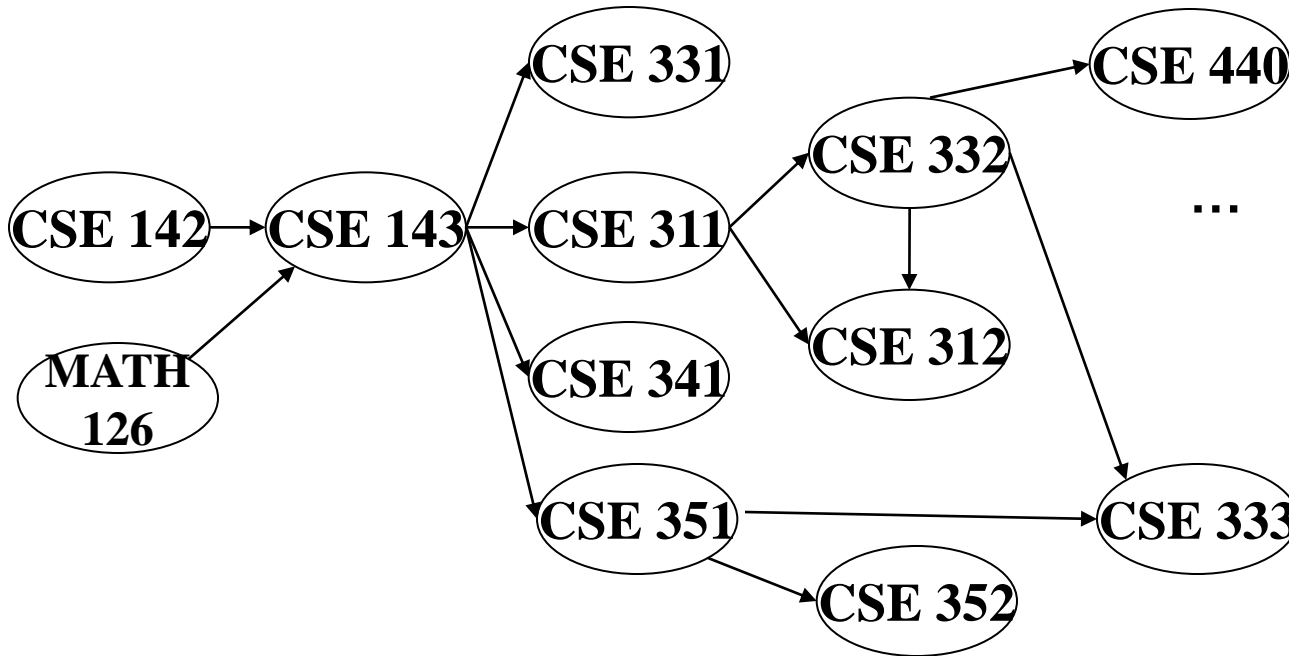


Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x		x	x		
In-degree:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0	0	0
			0		0			0				



# Example

Output: 126  
 142  
 143  
 311  
 331  
 332  
 312  
 341  
 351  
 333  
 352  
 440



Node:	126	142	143	311	312	331	332	333	341	351	352	440
Removed?	x	x	x	x	x	x	x	x	x	x	x	x
In-degree:	0	0	2	1	2	1	1	2	1	1	1	1
			1	0	1	0	0	1	0	0	0	0
			0		0			0				

## *Running Time?*

```
labelEachVertexWithItsInDegree();  
for(ctr=0; ctr < numVertices; ctr++){  
    v = findNewVertexOfDegreeZero();  
    put v next in output  
    for each w adjacent to v  
        w.indegree--;  
}
```

# Running Time?

```
labelEachVertexWithItsInDegree();  
for(ctr=0; ctr < numVertices; ctr++){  
    v = findNewVertexOfDegreeZero();  
    put v next in output  
    for each w adjacent to v  
        w.indegree--;  
}
```

- What is the worst-case running time?
  - Initialization  $O(|V| + |E|)$  (assuming adjacency list)
  - Sum of all find-new-vertex  $O(|V|^2)$  (because each  $O(|V|)$ )
  - Sum of all decrements  $O(|E|)$  (assuming adjacency list)
  - So total is  $O(|V|^2 + |E|)$  – not good for a sparse graph!

# Doing Better

The trick is to avoid searching for a zero-degree node every time!

- Keep the “pending” zero-degree nodes in a list, stack, queue, bag, or something
- Order we process them affects the output but not correctness or efficiency, assuming add/remove are both  $O(1)$

Using a queue:

1. Label each vertex with its in-degree, enqueue 0-degree nodes
2. While queue is not empty
  - a)  $v = \text{dequeue}()$
  - b) Output  $v$  and remove it from the graph
  - c) For each vertex  $u$  adjacent to  $v$ , decrement the in-degree of  $u$ , if new degree is 0, enqueue it

# *Running Time?*

```
labelAllAndEnqueueZeros();  
for(ctr=0; ctr < numVertices; ctr++){  
    v = dequeue();  
    put v next in output  
    for each w adjacent to v {  
        w.indegree--;  
        if(w.indegree==0)  
            enqueue(w);  
    }  
}
```

# Running Time?

```
labelAllAndEnqueueZeros();
for(ctr=0; ctr < numVertices; ctr++){
    v = dequeue();
    put v next in output
    for each w adjacent to v {
        w.indegree--;
        if(w.indegree==0)
            enqueue(w);
    }
}
```

- Initialization:  $O(|V| + |E|)$  (assuming adjacency list)
- Sum of all enqueues and dequeues:  $O(|V|)$
- Sum of all decrements:  $O(|E|)$  (assuming adjacency list)
- So total is  $O(|E| + |V|)$  – much better for sparse graph!



# CSE332: Data Abstractions

## Lecture 14: Shortest Paths

James Fogarty  
Winter 2012

# Single Source Shortest Paths

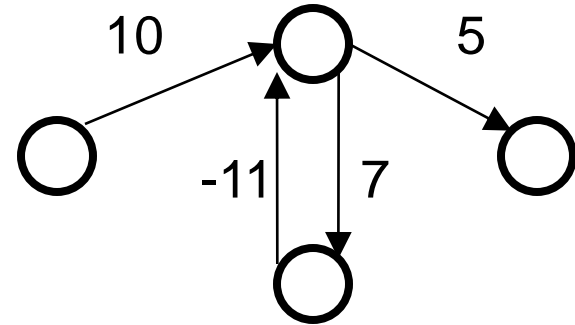
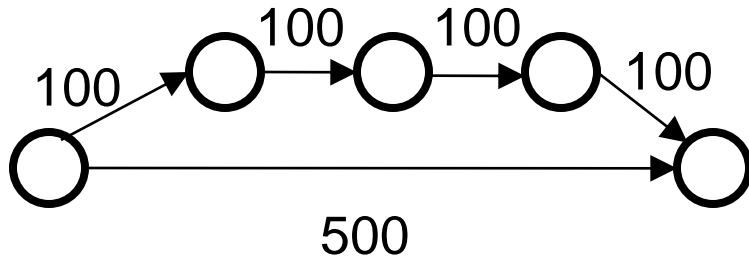
- Done: BFS for minimum path length from  $v$  to  $u$  in time  $O(|E|+(|V|))$
- Actually, can find the minimum path length from  $v$  to *every node*
  - Still  $O(|E|+(|V|))$
  - No faster way for a “distinguished” destination in the worst-case
- Now: Weighted graphs

Given a weighted graph and node  $v$ ,  
find the minimum-cost path from  $v$  to every node

- As before, asymptotically no harder than for one destination
- Unlike before, BFS will not work



# Not as Easy



Why BFS won't work: Shortest path may not have the fewest edges  
– Annoying when this happens with costs of flights

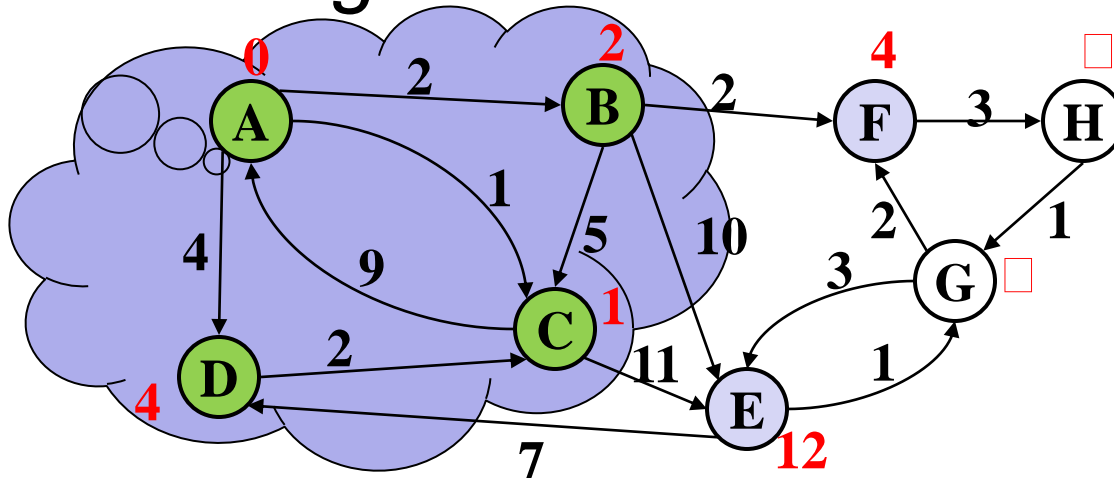
We will assume there are no negative weights

- Problem is ill-defined if there are negative-cost *cycles*
- Today's algorithm is wrong if *edges* can be negative

# *Dijkstra's Algorithm*

- Named after its inventor Edsger Dijkstra (1930-2002)
  - Truly one of the “founders” of computer science; this is just one of his many contributions
  - Sample quotation: “computer science is no more about computers than astronomy is about telescopes”
- The idea: reminiscent of BFS, but adapted to handle weights
  - A priority queue will prove useful for efficiency
  - Grow set of nodes whose shortest distance has been computed
  - Nodes not in the set will have a “best distance so far”

# Dijkstra's Algorithm: Idea



- Initially, start node has cost 0 and all other nodes have cost  $\infty$
- At each step:
  - Pick closest unknown vertex  $v$
  - Add it to the “cloud” of known vertices
  - Update distances for nodes with edges from  $v$
- That's it! But we need to prove it produces correct answers

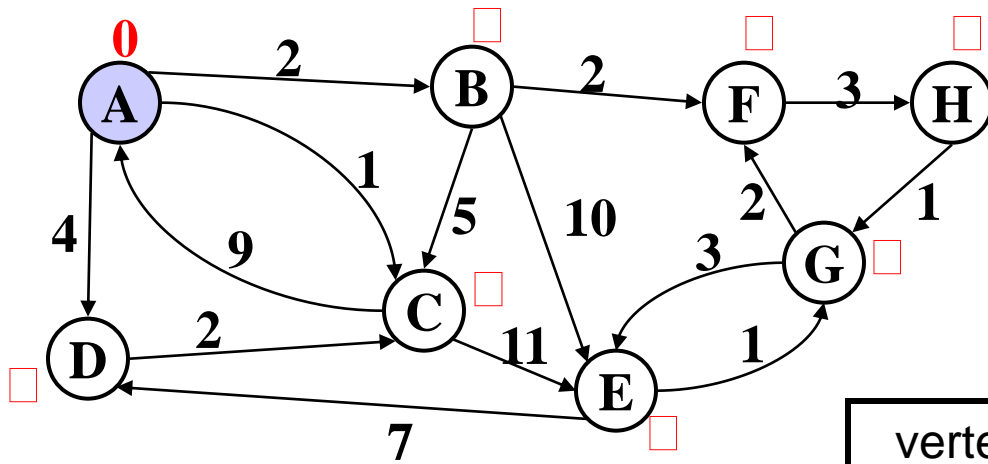
# *The Algorithm*

1. For each node  $v$ , set  $v.cost = \infty$  and  $v.known = false$
2. Set  $source.cost = 0$
3. While there are unknown nodes in the graph
  - a) Select the unknown node  $v$  with lowest cost
  - b) Mark  $v$  as known
  - c) For each edge  $(v, u)$  with weight  $w$ ,
    - $c1 = v.cost + w$  // cost of best path through  $v$  to  $u$
    - $c2 = u.cost$  // cost of best path to  $u$  previously known
    - $if(c1 < c2) \{$  // if the path through  $v$  is better
      - $u.cost = c1$
      - $u.path = v$  // for computing actual paths
    - $\}$

# *Important Features*

- When a vertex is marked known,  
the cost of the shortest path to that node is known
  - The path is also known by following back-pointers
- While a vertex is still not known,  
another shorter path to it **might** still be found

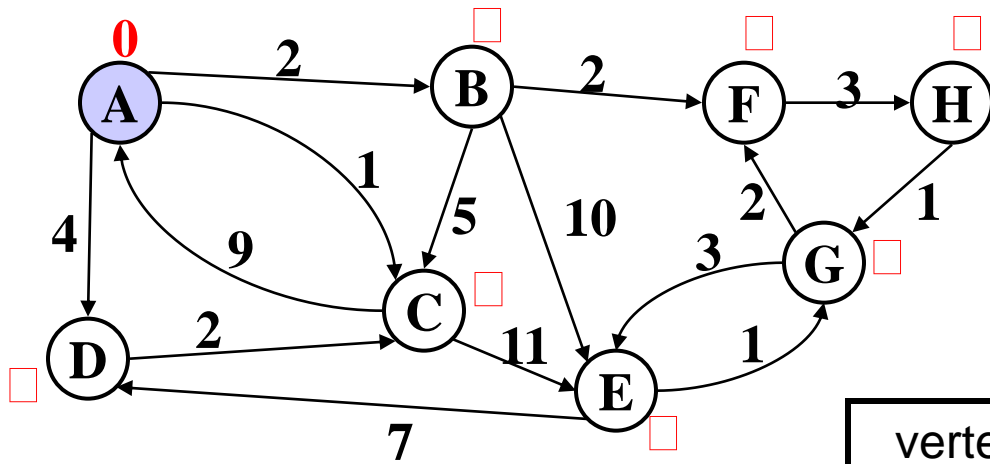
# Example #1



vertex	known?	cost	path
A			
B			
C			
D			
E			
F			
G			
H			

Order Added to Known Set:

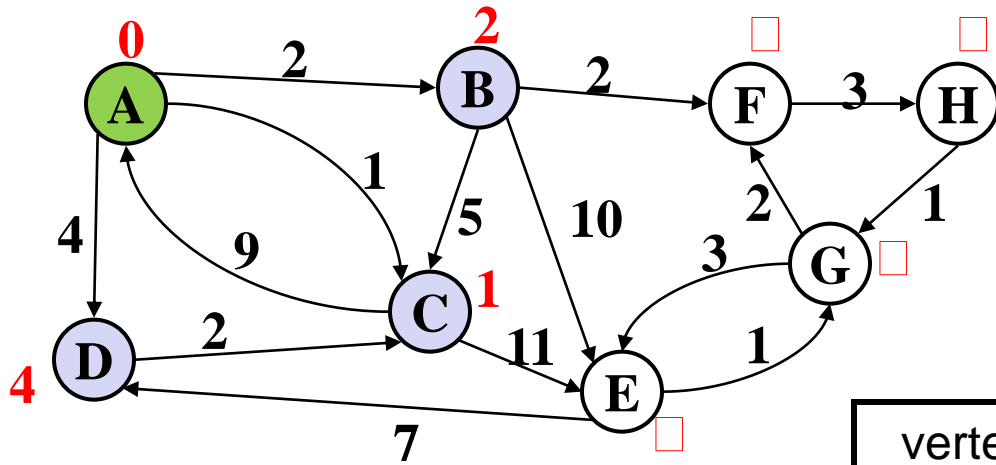
# Example #1



vertex	known?	cost	path
A		0	
B		??	
C		??	
D		??	
E		??	
F		??	
G		??	
H		??	

Order Added to Known Set:

# Example #1



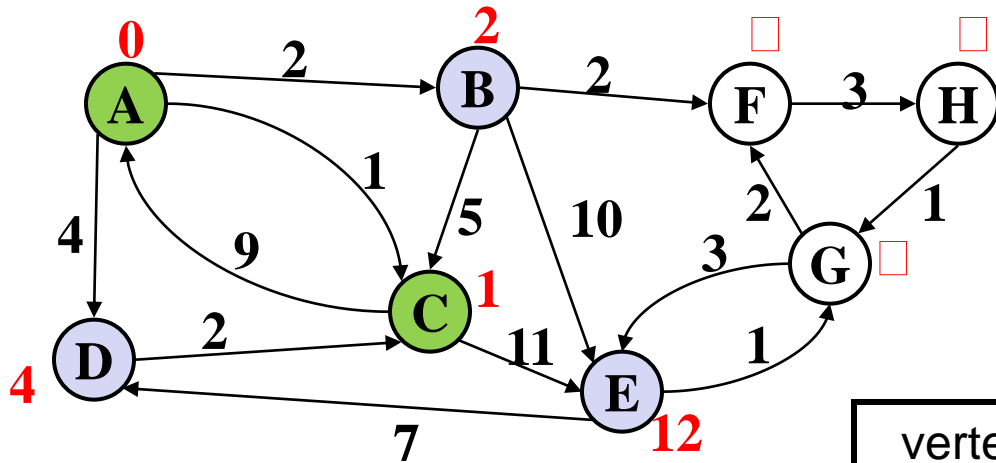
vertex	known?	cost	path
A	Y	0	
B		$\leq 2$	A
C		$\leq 1$	A
D		$\leq 4$	A
E		??	
F		??	
G		??	
H		??	

Order Added to Known Set:

A



# Example #1

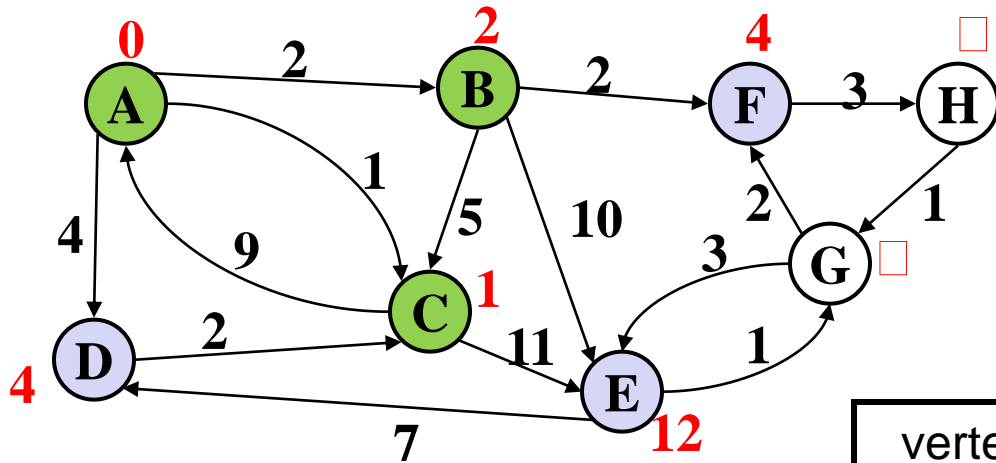


vertex	known?	cost	path
A	Y	0	
B		$\leq 2$	A
C	Y	1	A
D		$\leq 4$	A
E		$\leq 12$	C
F		??	
G		??	
H		??	

Order Added to Known Set:

A, C

# Example #1

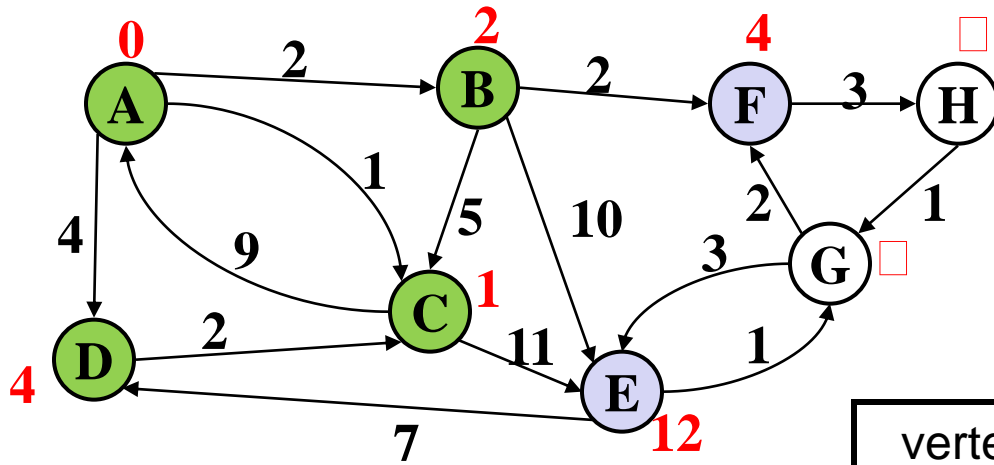


vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D		$\leq 4$	A
E		$\leq 12$	C
F		$\leq 4$	B
G		??	
H		??	

Order Added to Known Set:

A, C, B

# Example #1

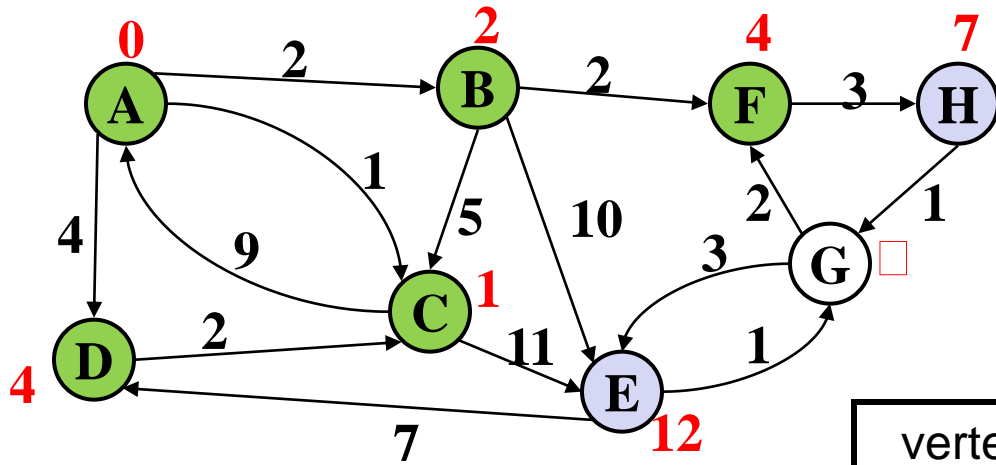


vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		$\leq 12$	C
F		$\leq 4$	B
G		??	
H		??	

Order Added to Known Set:

A, C, B, D

# Example #1

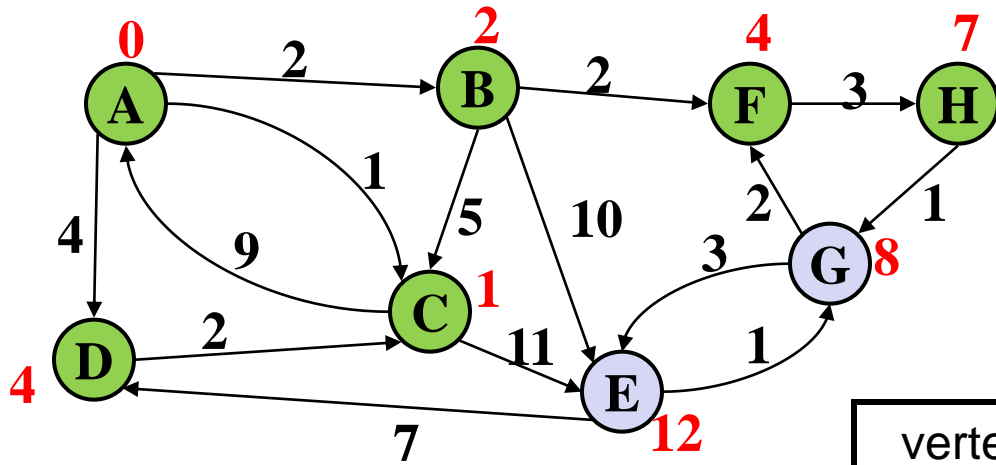


vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		$\leq 12$	C
F	Y	4	B
G		??	
H		$\leq 7$	F

Order Added to Known Set:

A, C, B, D, F

# Example #1

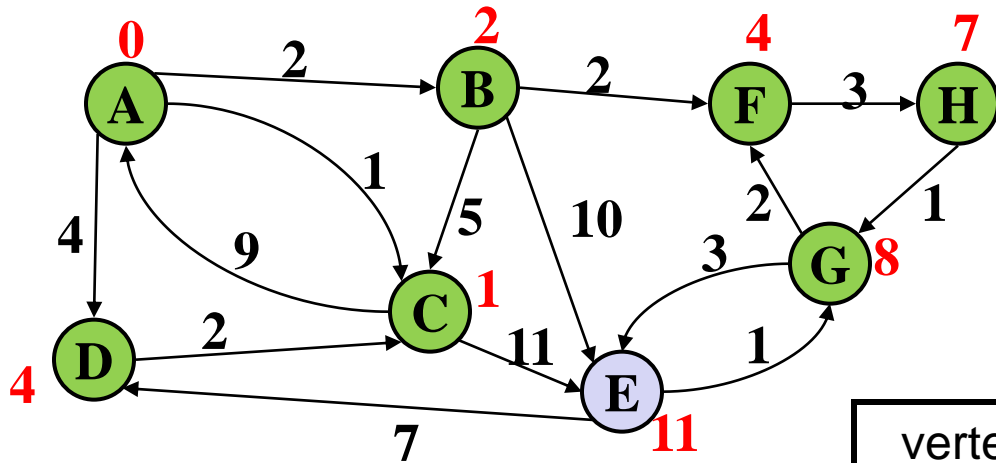


vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		$\leq 12$	C
F	Y	4	B
G		$\leq 8$	H
H	Y	7	F

Order Added to Known Set:

A, C, B, D, F, H

# Example #1

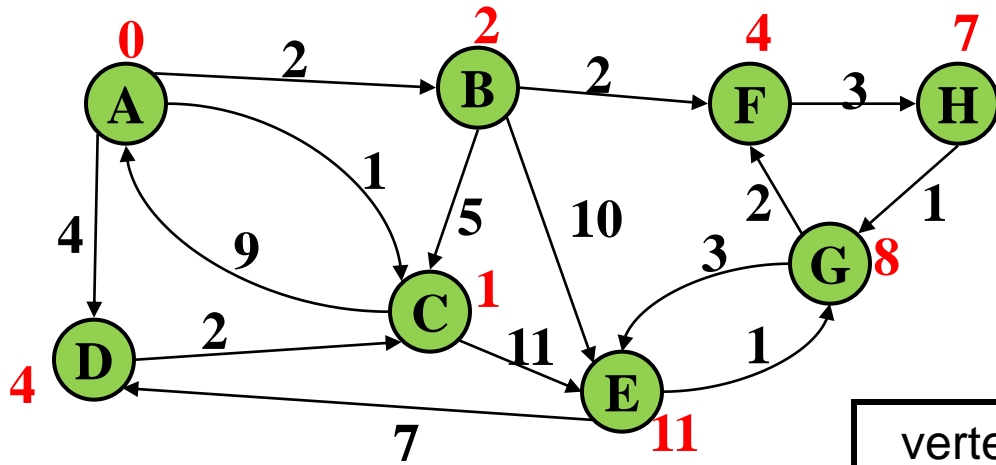


vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		$\leq 11$	<b>G</b>
F	Y	4	B
G	Y	8	H
H	Y	7	F

Order Added to Known Set:

A, C, B, D, F, H, G

# Example #1



vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Order Added to Known Set:

A, C, B, D, F, H, G, E

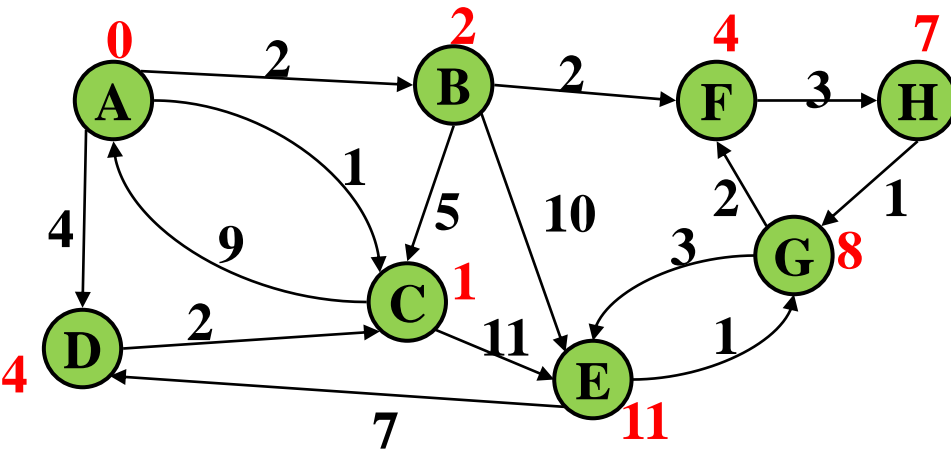
# *Important Features*

- When a vertex is marked known,  
the cost of the shortest path to that node is known
  - The path is also known by following back-pointers
- While a vertex is still not known,  
another shorter path to it **might** still be found



# Interpreting the Results

- Now that we're done, how do we get the path from, say, A to E?



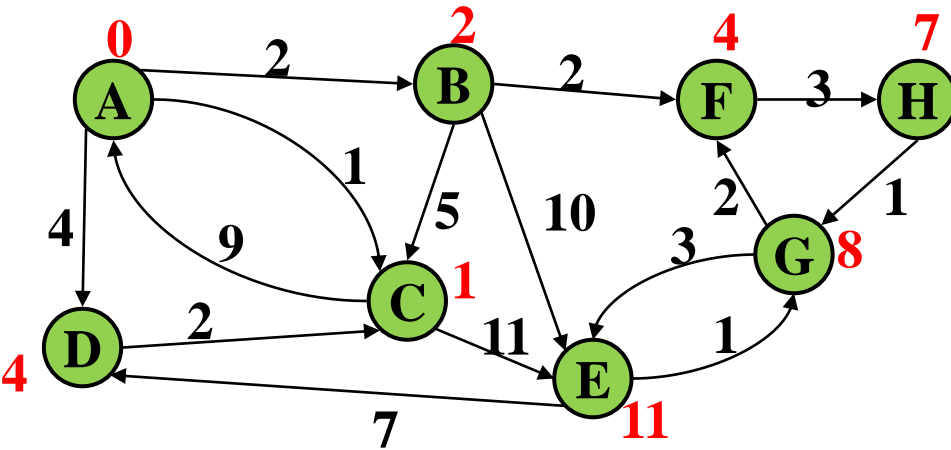
Order Added to Known Set:

A, C, B, D, F, H, G, E

vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

# Stopping Short

- How would this have worked differently if we were only interested in:
  - the path from A to G?
  - the path from A to E?

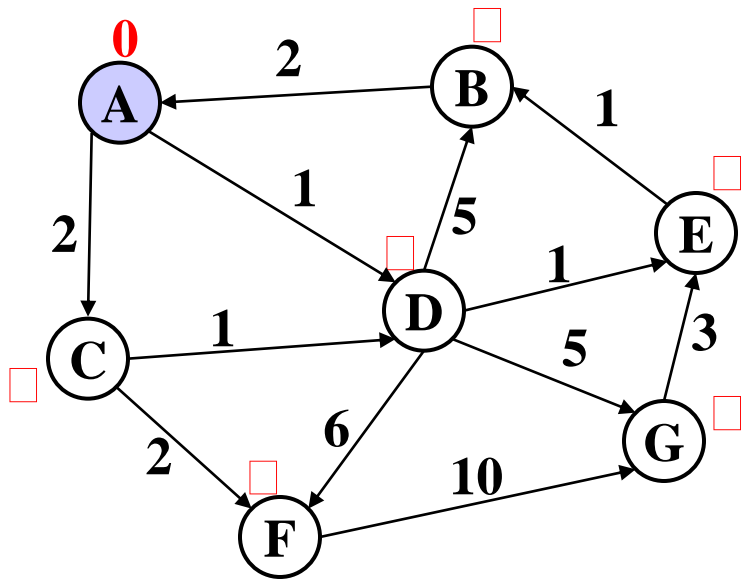


Order Added to Known Set:

A, C, B, D, F, H, G, E

vertex	known?	cost	path
A	Y	0	
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

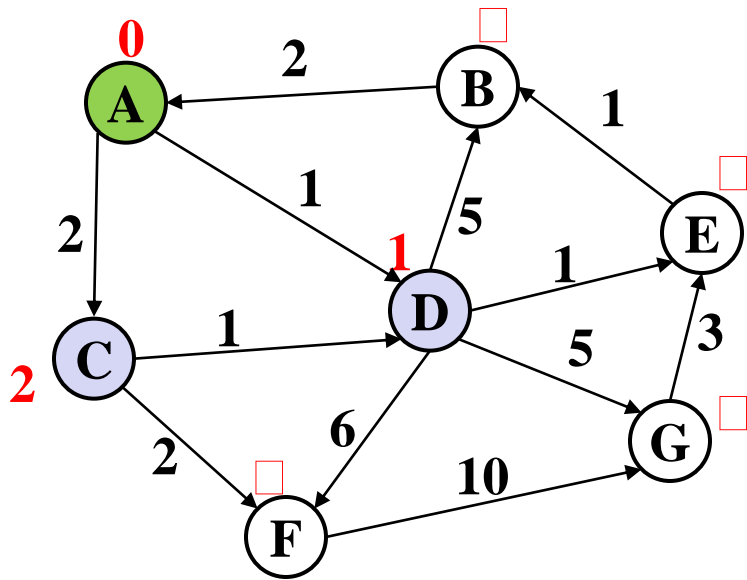
## Example #2



vertex	known?	cost	path
A		0	
B		??	
C		??	
D		??	
E		??	
F		??	
G		??	

Order Added to Known Set:

# Example #2

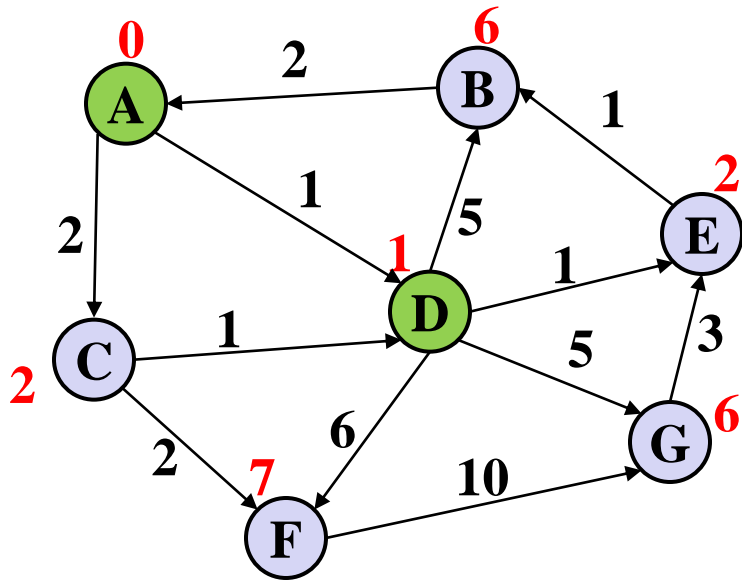


vertex	known?	cost	path
A	Y	0	
B		??	
C		$\leq 2$	A
D		$\leq 1$	A
E		??	
F		??	
G		??	

Order Added to Known Set:

A

## Example #2

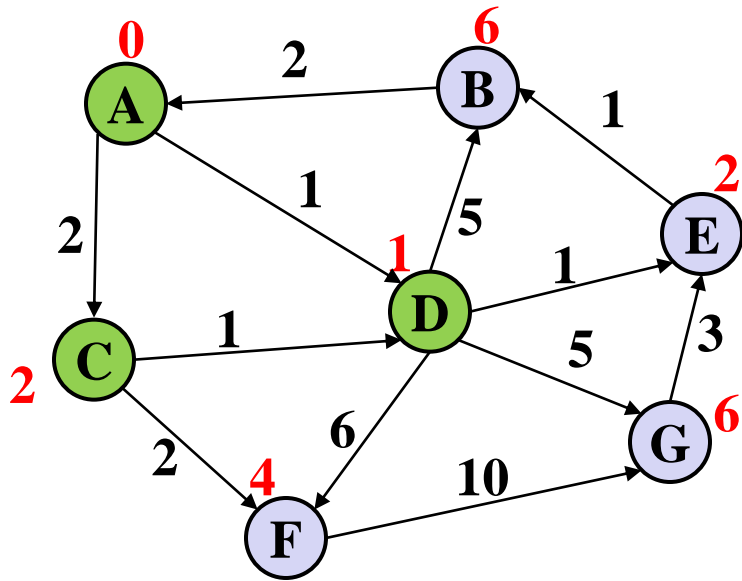


Order Added to Known Set:

A, D

vertex	known?	cost	path
A	Y	0	
B		$\leq 6$	D
C		$\leq 2$	A
D	Y	1	A
E		$\leq 2$	D
F		$\leq 7$	D
G		$\leq 6$	D

## Example #2

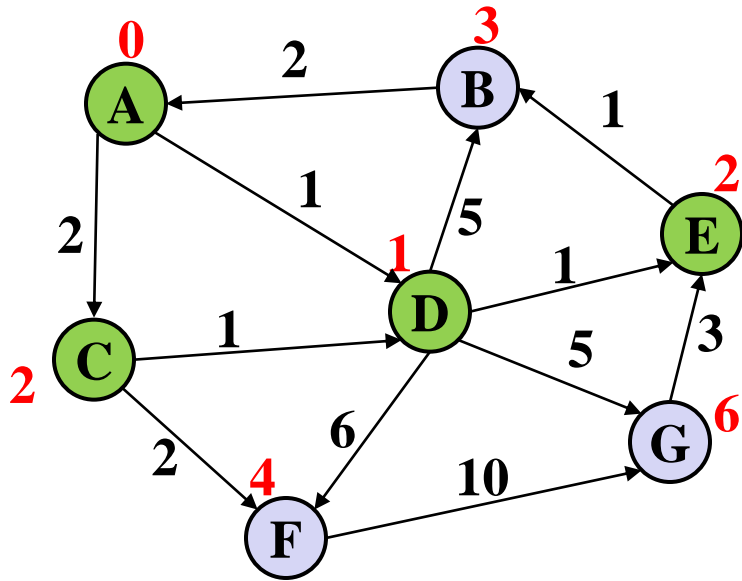


vertex	known?	cost	path
A	Y	0	
B		$\leq 6$	D
C	Y	2	A
D	Y	1	A
E		$\leq 2$	D
F		$\leq 4$	C
G		$\leq 6$	D

Order Added to Known Set:

A, D, C

## Example #2

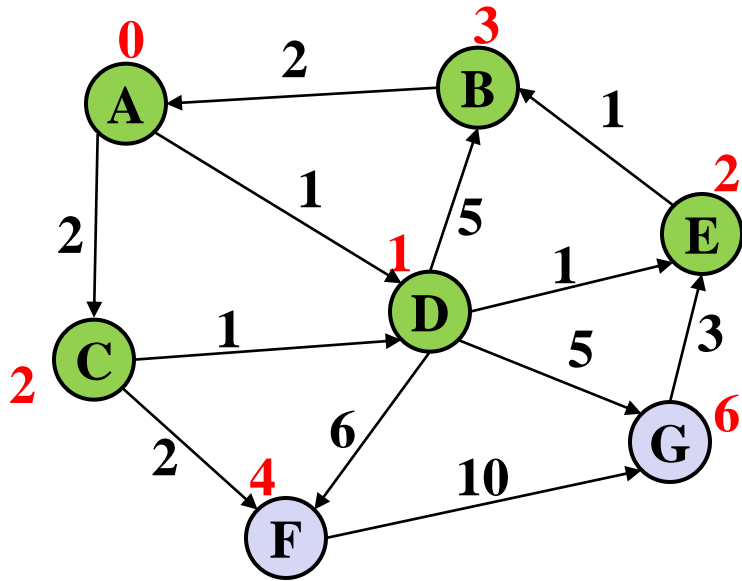


Order Added to Known Set:

A, D, C, E

vertex	known?	cost	path
A	Y	0	
B		$\leq 3$	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F		$\leq 4$	C
G		$\leq 6$	D

## Example #2



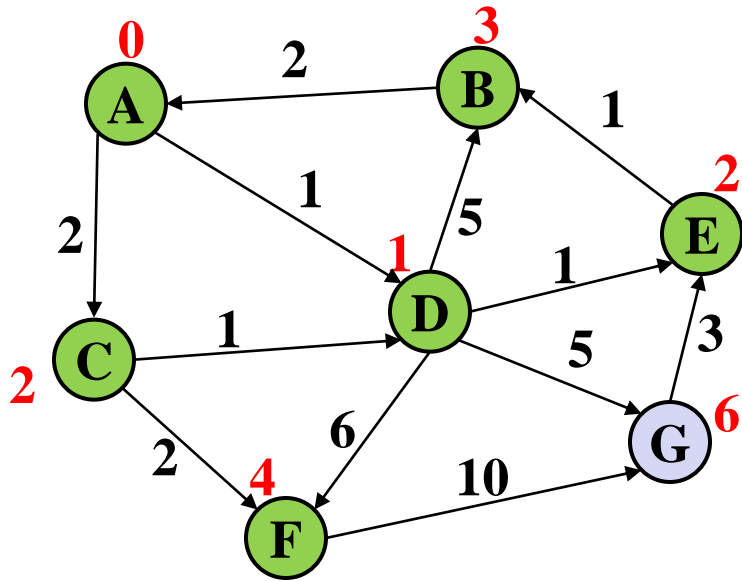
Order Added to Known Set:

A, D, C, E, B

vertex	known?	cost	path
A	Y	0	
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F		$\leq 4$	C
G		$\leq 6$	D



## Example #2

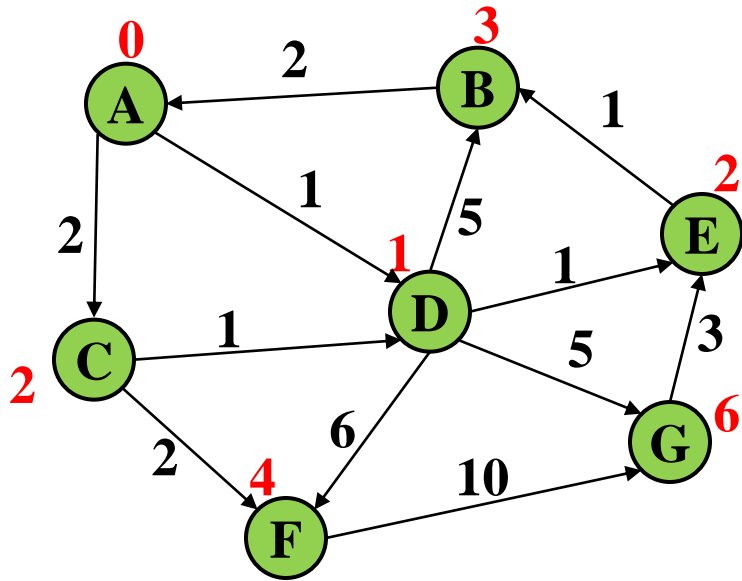


vertex	known?	cost	path
A	Y	0	
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F	Y	4	C
G		$\leq 6$	D

Order Added to Known Set:

A, D, C, E, B, F

## Example #2

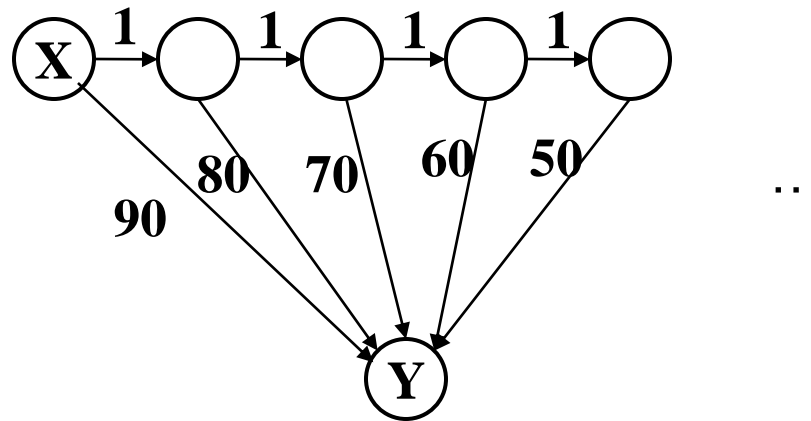


Order Added to Known Set:

A, D, C, E, B, F, G

vertex	known?	cost	path
A	Y	0	
B	Y	3	E
C	Y	2	A
D	Y	1	A
E	Y	2	D
F	Y	4	C
G	Y	6	D

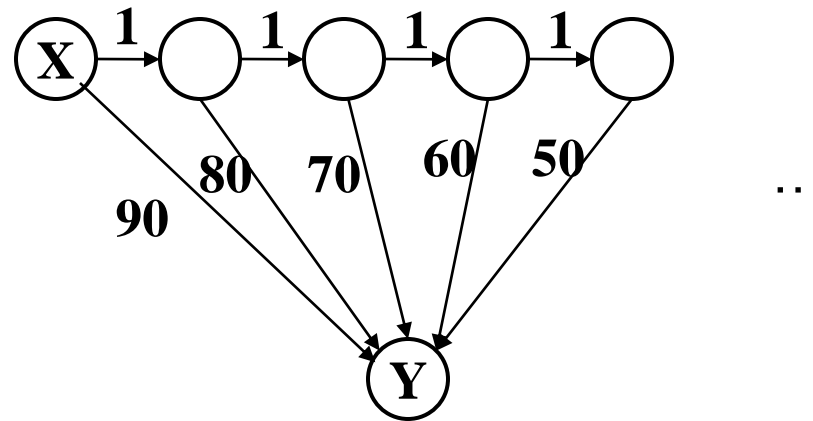
## Example #3



How will the best-cost-so-far for Y proceed?

Is this expensive?

## Example #3



How will the best-cost-so-far for Y proceed? 90, 81, 72, 63, 54, ...

Is this expensive? No, each edge is processed only once

# *A Greedy Algorithm*

- Dijkstra's algorithm
  - For single-source shortest paths in a weighted graph (directed or undirected) with no negative-weight edges
  - An example of a *greedy algorithm*:
    - At each step, irrevocably does what seems best at that step
      - once a vertex is in the known set, does not go back and readjust its decision
    - Locally optimal
      - does not always mean globally optimal

# *Where are We?*

- Have described Dijkstra's algorithm
  - For single-source shortest paths in a weighted graph (directed or undirected) with no negative-weight edges
- What should we do after learning an algorithm?
  - Prove it is correct
    - Not obvious!
    - We will sketch the key ideas
  - Analyze its efficiency
    - Will do better by using a data structure we learned earlier!

# *Correctness: Intuition*

Rough intuition:

All the “known” vertices have the correct shortest path

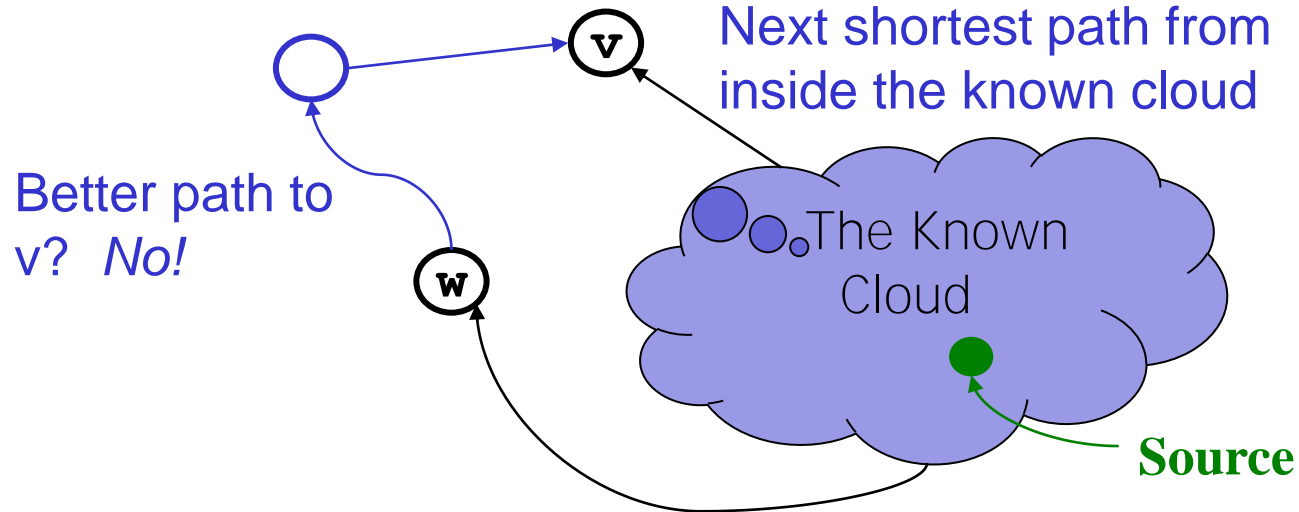
- True initially: shortest path to start node has cost 0
- If it stays true every time we mark a node “known”, then by induction this holds and eventually everything is “known”

Key fact we need:

When we mark a vertex “known” we won’t discover a shorter path later!

- This holds only because Dijkstra’s algorithm picks the node with the next shortest path-so-far
- The proof is by contradiction...

# Correctness: The Cloud (Rough Sketch)



Suppose  $v$  is the next node to be marked known ("added to the cloud")

- The **best-known path** to  $v$  must have only nodes "in the cloud"
  - We have selected it, and we only know about paths through the cloud to a node at the edge of the cloud
- Assume the **actual shortest path** to  $v$  is different
  - It is not entirely within the cloud, or else we would know about it
    - So it must use non-cloud nodes
  - Let  $w$  be the *first* non-cloud node on this path.
  - The part of the path up to  $w$  is **already known** and must be shorter than the best-known path to  $v$ . So  $v$  would not have been picked. Contradiction.

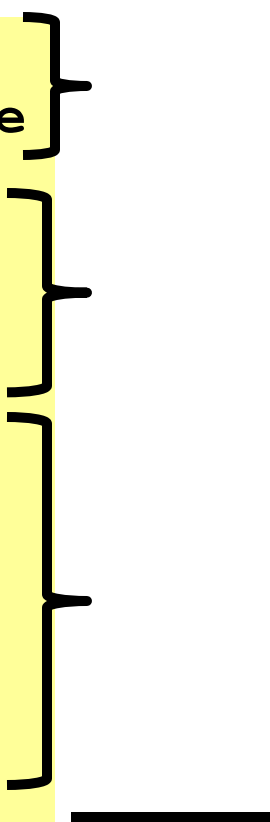


# *Efficiency, First Approach*

Use pseudocode to determine asymptotic run-time

- Notice each edge is processed only once

```
dijkstra(Graph G, Node start) {  
  for each node: x.cost=infinity, x.known=false  
  start.cost = 0  
  while(not all nodes are known) {  
    b = find unknown node with smallest cost  
    b.known = true  
    for each edge (b,a) in G  
      if(!a.known)  
        if(b.cost + weight((b,a)) < a.cost) {  
          a.cost = b.cost + weight((b,a))  
          a.path = b  
        }  
  }  
}
```



# Efficiency, First Approach

Use pseudocode to determine asymptotic run-time

- Notice each edge is processed only once

```
dijkstra(Graph G, Node start) {  
  for each node: x.cost=infinity, x.known=false  
  start.cost = 0  
  while(not all nodes are known) {  
    b = find unknown node with smallest cost  
    b.known = true  
    for each edge (b,a) in G  
      if(!a.known)  
        if(b.cost + weight((b,a)) < a.cost) {  
          a.cost = b.cost + weight((b,a))  
          a.path = b  
        }  
  }  
}
```

$O(|V|)$

$O(|V|^2)$

$O(|E|)$

---

$O(|V|^2)$

# *Improving Asymptotic Running Time*

- So far:  $O(|V|^2)$
- We had a similar “problem” with topological sort being  $O(|V|^2)$  due to each iteration looking for the node to process next
  - We solved it with a queue of zero-degree nodes
  - But here we need the lowest-cost node and costs can change as we process edges
- Solution?

# *Improving Asymptotic Running Time*

- So far:  $O(|V|^2)$
- We had a similar “problem” with topological sort being  $O(|V|^2)$  due to each iteration looking for the node to process next
  - We solved it with a queue of zero-degree nodes
  - But here we need the lowest-cost node and costs can change as we process edges
- Solution?
  - A priority queue holding all unknown nodes, sorted by cost
  - But must support **decreaseKey** operation
    - Must maintain a reference from each node to its position in the priority queue
    - Conceptually simple, but can be a pain to code up

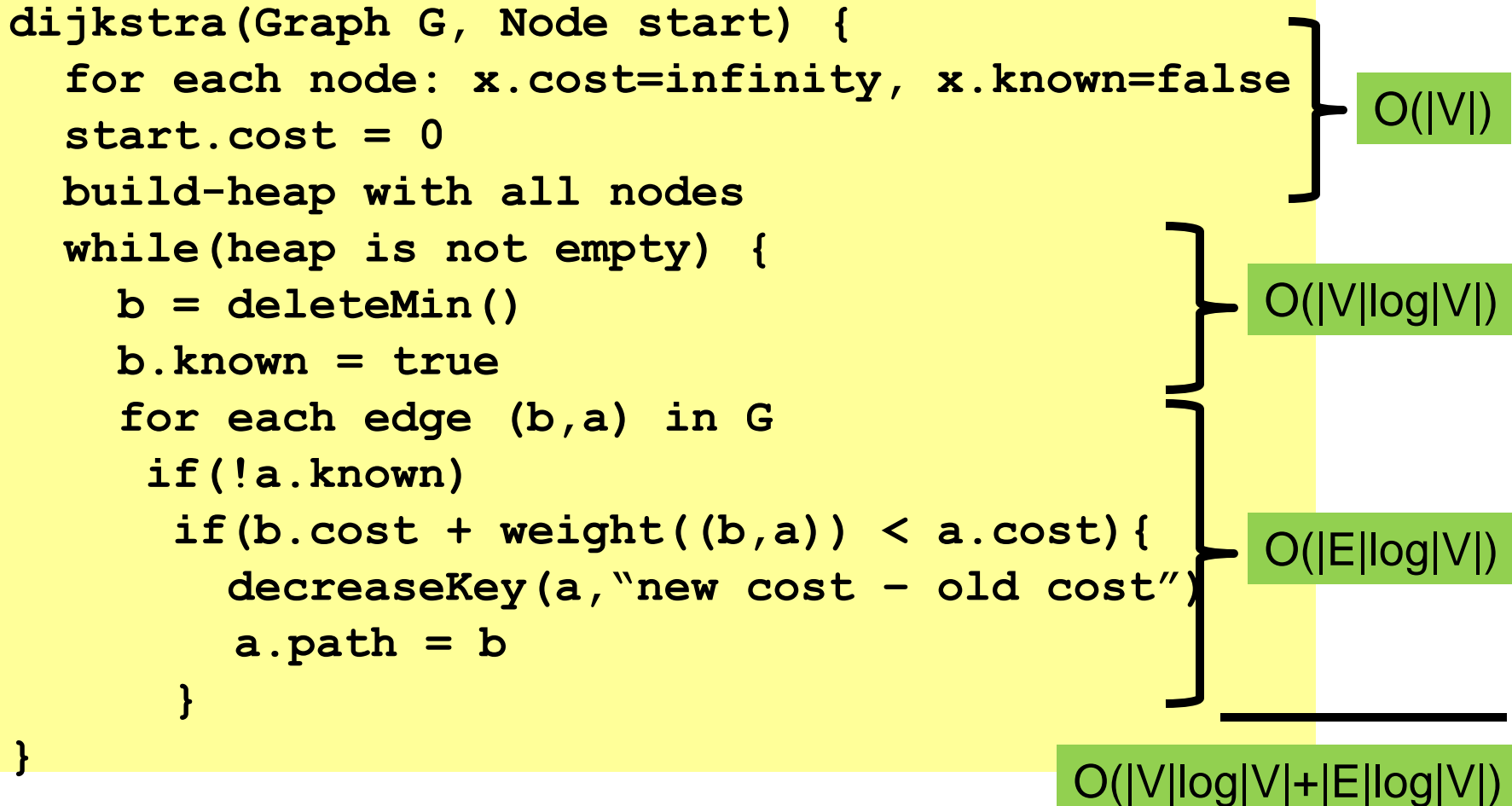
# Efficiency, Second Approach

Use pseudocode to determine asymptotic run-time

```
dijkstra(Graph G, Node start) {  
  for each node: x.cost=infinity, x.known=false  
  start.cost = 0  
  build-heap with all nodes  
  while(heap is not empty) {  
    b = deleteMin()  
    b.known = true  
    for each edge (b,a) in G  
      if(!a.known)  
        if(b.cost + weight((b,a)) < a.cost) {  
          decreaseKey(a, "new cost - old cost")  
          a.path = b  
        }  
  }  
}
```

# Efficiency, Second Approach

Use pseudocode to determine asymptotic run-time



# *Dense vs. Sparse Again*

- First approach:  $O(|V|^2)$
- Second approach:  $O(|V|\log|V|+|E|\log|V|)$
- So which is better?
  - Sparse:  $O(|V|\log|V|+|E|\log|V|)$  (if  $|E| > |V|$ , then  $O(|E|\log|V|)$ )
  - Dense:  $O(|V|^2)$
- But, remember these are worst-case and asymptotic
  - Priority queue might have slightly worse constant factors
  - On the other hand, for “normal graphs”, we might rarely call **decreaseKey** (or not percolate far), making  $|E|\log|V|$  more like  $|E|$

# *All-Pairs Shortest Path*

- Find the shortest path between all pairs of vertices in the graph
- How?



# *Dynamic Programming*

Algorithmic technique that systematically records the answers to sub-problems in a table and re-uses those recorded results (rather than re-computing them).

**Simple Example:** Calculating the Nth Fibonacci number.

$$\text{Fib}(N) = \text{Fib}(N-1) + \text{Fib}(N-2)$$

Recursion would be insanely expensive,  
but it is cheap if you already know results of prior computations

# *Floyd-Warshall*

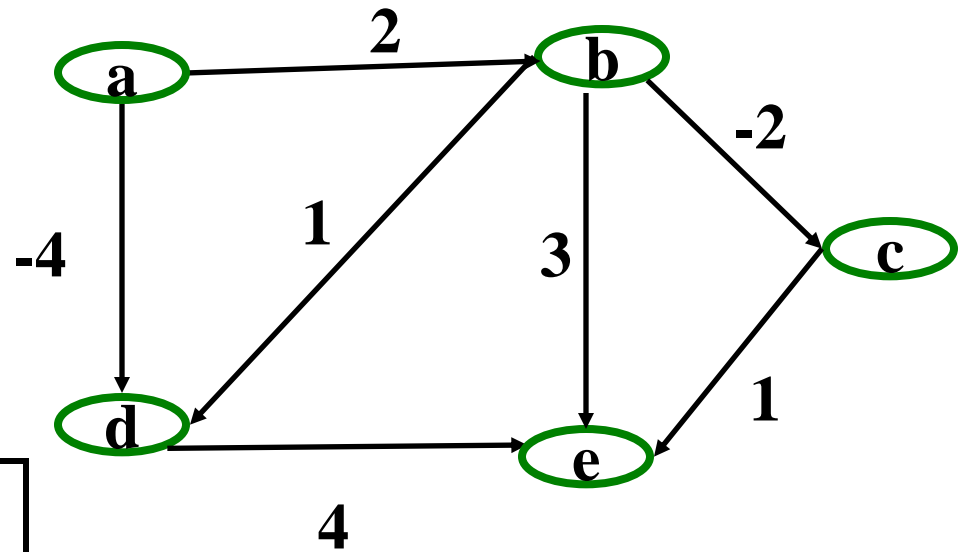
```
for (int k = 1; k <= V; k++)  
  for (int i = 1; i <= V; i++)  
    for (int j = 1; j <= V; j++)  
      if ( ( M[i][k] + M[k][j] ) < M[i][j] )  
          M[i][j] = M[i][k] + M[k][j]
```

## **Invariant:**

**After the kth iteration, for all pairs of vertices the matrix includes the shortest path containing only vertices 1..k as intermediate vertices**

**Initial state  
of the matrix:**

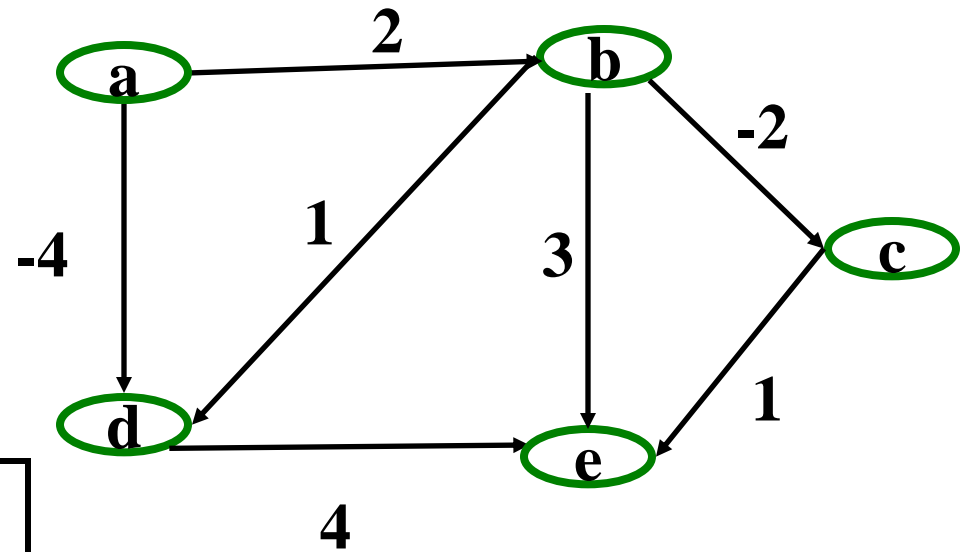
	a	b	c	d	e
a	0	2	-	-4	-
b	-	0	-2	1	3
c	-	-	0	-	1
d	-	-	-	0	4
e	-	-	-	-	0



$$M[i][j] = \min(M[i][j], M[i][k] + M[k][j])$$

**Initial state  
of the matrix:**

	a	b	c	d	e
a	0	2	-	-4	-
b	-	0	-2	1	3
c	-	-	0	-	1
d	-	-	-	0	4
e	-	-	-	-	0

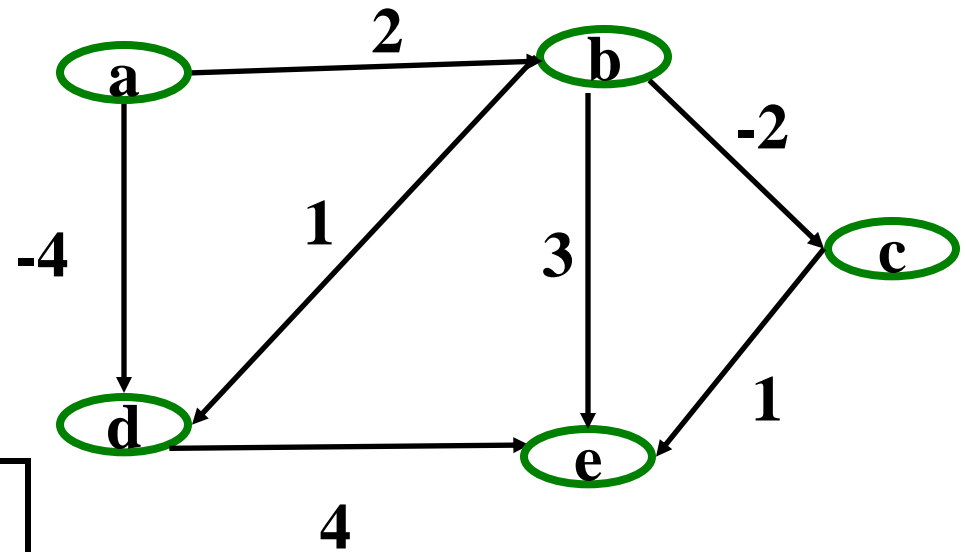


**k = 1**

$$M[i][j] = \min(M[i][j], M[i][k] + M[k][j])$$

**Initial state  
of the matrix:**

	a	b	c	d	e
a	0	2	-	-4	-
b	-	0	-2	1	3
c	-	-	0	-	1
d	-	-	-	0	4
e	-	-	-	-	0

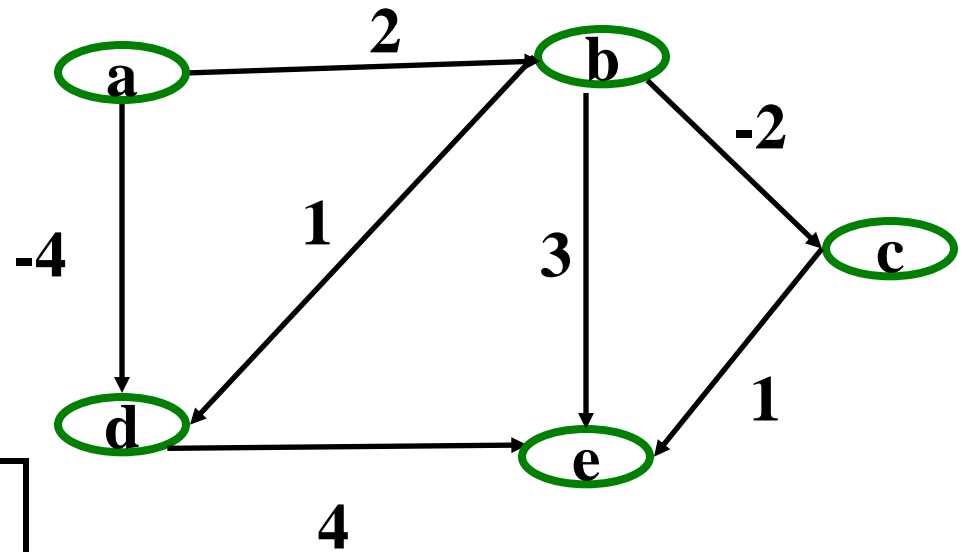


**k = 2**

$$M[i][j] = \min(M[i][j], M[i][k] + M[k][j])$$

**Initial state  
of the matrix:**

	a	b	c	d	e
a	0	2	0	-4	5
b	-	0	-2	1	3
c	-	-	0	-	1
d	-	-	-	0	4
e	-	-	-	-	0

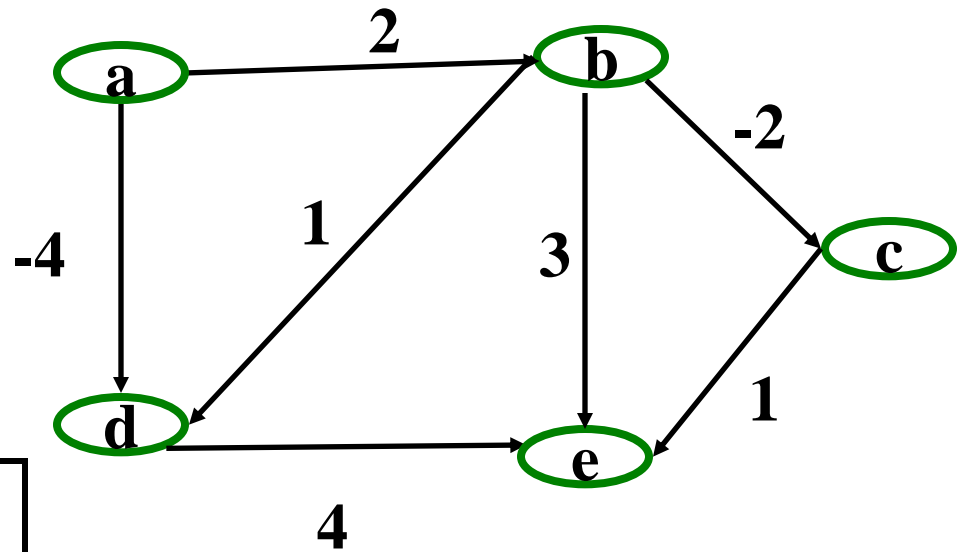


**k = 2**

$$M[i][j] = \min(M[i][j], M[i][k] + M[k][j])$$

**Initial state  
of the matrix:**

	a	b	c	d	e
a	0	2	0	-4	5
b	-	0	-2	1	3
c	-	-	0	-	1
d	-	-	-	0	4
e	-	-	-	-	0

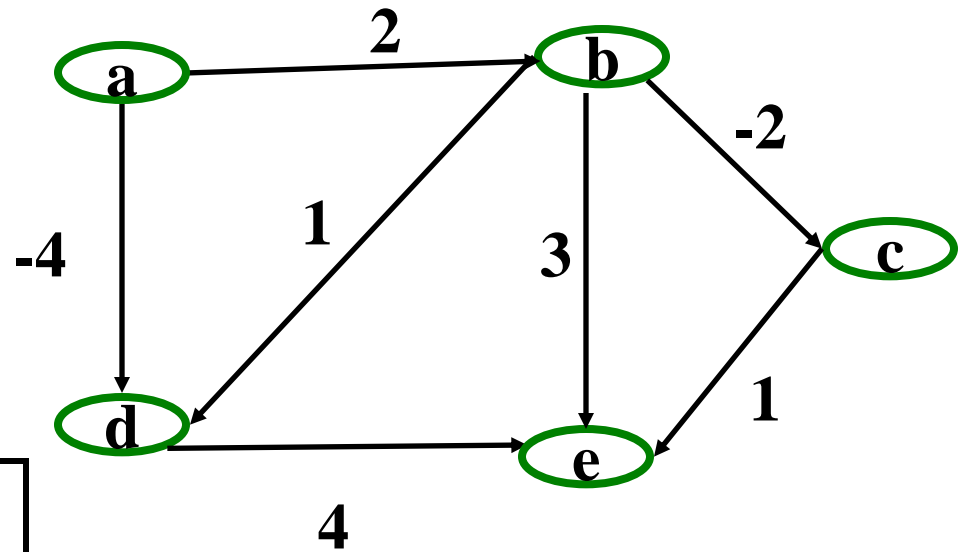


**k = 3**

$$M[i][j] = \min(M[i][j], M[i][k] + M[k][j])$$

**Initial state  
of the matrix:**

	a	b	c	d	e
a	0	2	0	-4	1
b	-	0	-2	1	-1
c	-	-	0	-	1
d	-	-	-	0	4
e	-	-	-	-	0



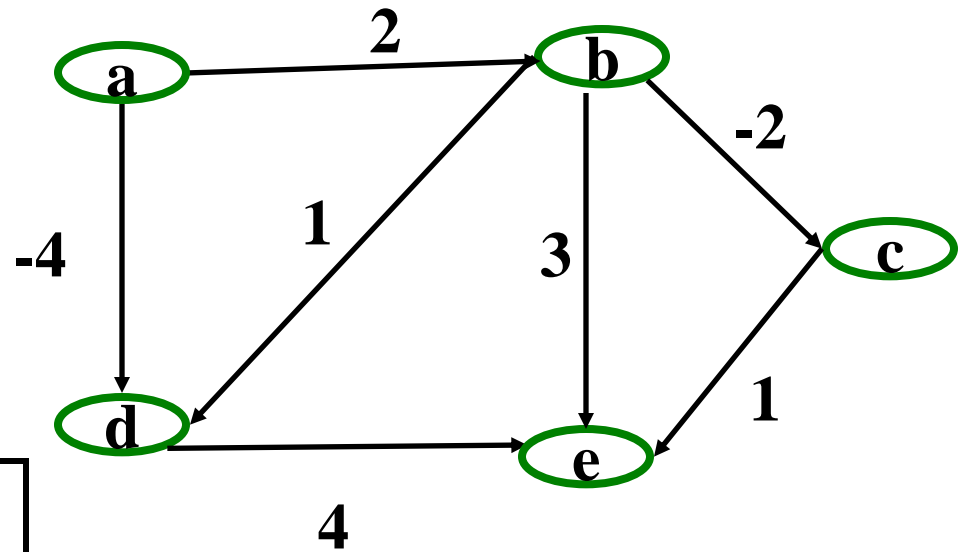
**k = 3**

$$M[i][j] = \min(M[i][j], M[i][k] + M[k][j])$$



**Initial state  
of the matrix:**

	a	b	c	d	e
a	0	2	0	-4	1
b	-	0	-2	1	-1
c	-	-	0	-	1
d	-	-	-	0	4
e	-	-	-	-	0

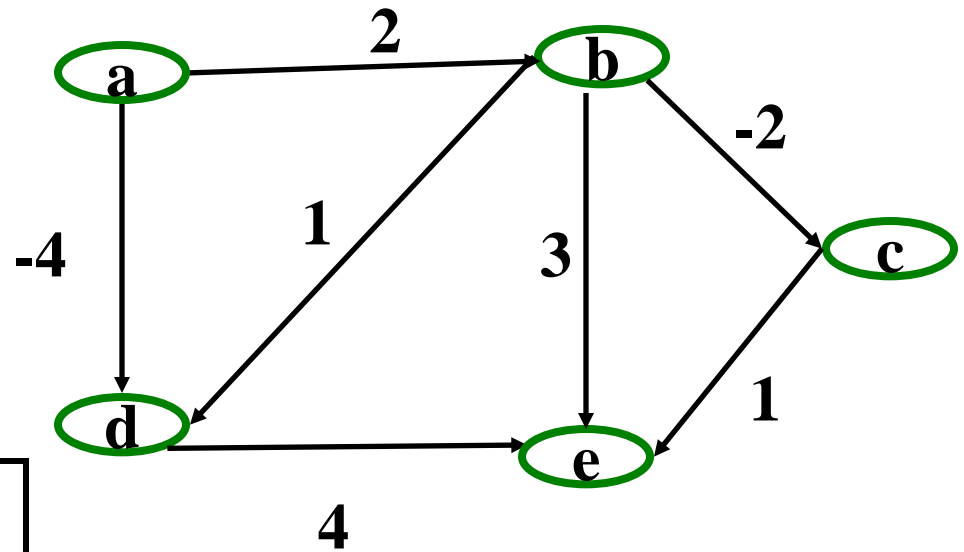


**k = 4**

$$M[i][j] = \min(M[i][j], M[i][k] + M[k][j])$$

**Initial state  
of the matrix:**

	a	b	c	d	e
a	0	2	0	-4	0
b	-	0	-2	1	-1
c	-	-	0	-	1
d	-	-	-	0	4
e	-	-	-	-	0

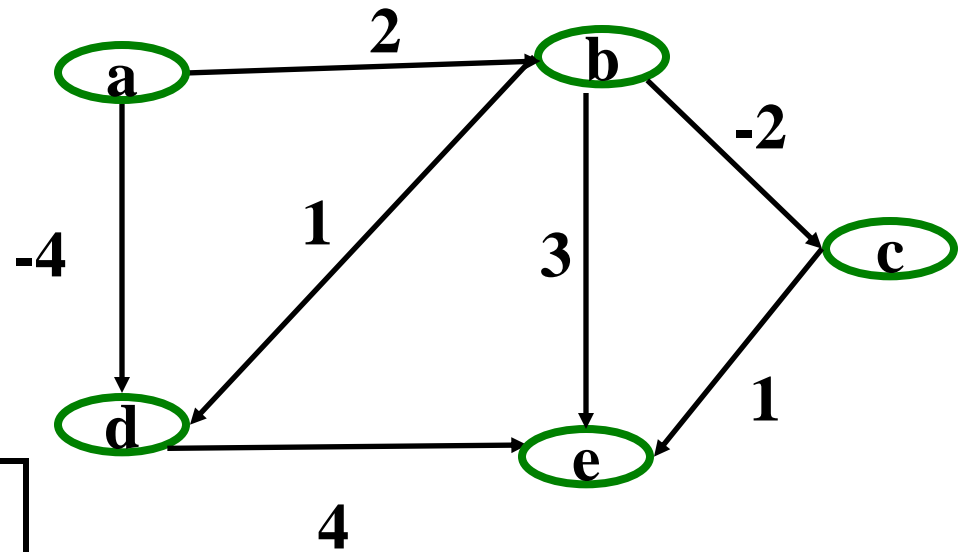


**k = 4**

$$M[i][j] = \min(M[i][j], M[i][k] + M[k][j])$$

**Initial state  
of the matrix:**

	a	b	c	d	e
a	0	2	0	-4	0
b	-	0	-2	1	-1
c	-	-	0	-	1
d	-	-	-	0	4
e	-	-	-	-	0



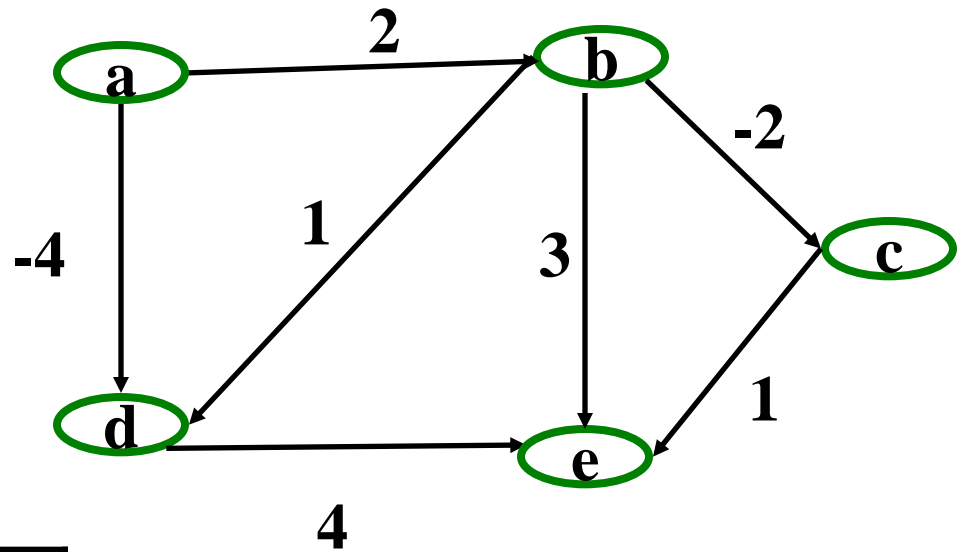
**k = 5**

$$M[i][j] = \min(M[i][j], M[i][k] + M[k][j])$$

# Floyd-Warshall

## All-Pairs

## Shortest Path



	a	b	c	d	e
a	0	2	0	-4	0
b	-	0	-2	1	-1
c	-	-	0	-	1
d	-	-	-	0	4
e	-	-	-	-	0

**Final Matrix  
Contents**

# *What Comes Next?*

In the logical course progression, we would next study

1. Minimum spanning trees

But to align lectures with projects and homeworks, instead we will

- Start parallelism and concurrency
- Come back to graphs at the end of the course

Note toward the future:

- We cannot do all of graphs last because of the CSE312 co-requisite (needed for study of NP)



# CSE332: Data Abstractions

## Lecture 15: Into to Parallelism and Concurrency

James Fogarty

Winter 2012

Including slides developed in part by  
Ruth Anderson, James Fogarty, Dan Grossman

# *Changing a Major Assumption*

So far most or all of your study of computer science has assumed

*One thing happened at a time*

Called **sequential programming** – everything part of one sequence

Removing this assumption creates major challenges & opportunities

- Programming: Divide work among **threads of execution** and coordinate among them (i.e., **synchronize** their work)
- Algorithms: How can parallel activity provide speed-up (more **throughput**, more work done per unit time)
- Data structures: May need to support **concurrent access** (multiple threads operating on data at the same time)

# *A Simplified View of History*

Writing correct and efficient multithreaded code is often much more difficult than single-threaded code

- Especially in typical languages like Java and C
- So we typically stay sequential whenever possible

From roughly 1980-2005, desktop computers got exponentially faster at running sequential programs

- About twice as fast every couple years

But nobody knows how to continue this

- Increasing clock rate generates too much heat
- Relative cost of memory access is too high
- Still making “wires exponentially smaller” (per Moore’s “Law”), so we put multiple processors on the same chip (i.e., “multicore”)



# *What to do with Multiple Processors?*

- Next computer you buy will likely have 4 processors
  - Wait a few years and it will be 8, 16, 32, ...
  - The chip companies have decided to do this (it is not a “law”)
- What can you do with them?
  - Run multiple totally different programs at the same time
    - Already do that? Yes, but with [time-slicing](#)
  - Do multiple things at once in one program
    - This will be our focus, and it is more difficult
    - Requires rethinking everything from asymptotic complexity to how to implement data-structure operations

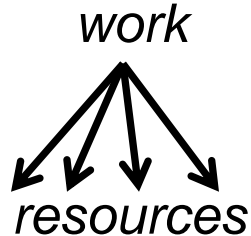
# *Parallelism vs. Concurrency*

Note: Terms not yet standard but the perspective is essential

- Many programmers confuse these concepts

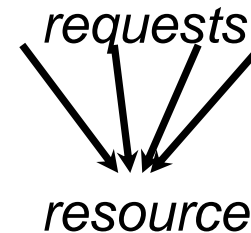
## Parallelism:

Use extra resources to solve a problem faster



## Concurrency:

Correctly and efficiently manage access to shared resources



There is some connection:

- Common to use threads for both
- If parallel computations need access to shared resources, then the concurrency needs to be managed

# *An Analogy*

CS1 idea: A program is like a recipe for a cook

- One cook who does one thing at a time!

Parallelism:

- Have lots of potatoes to slice?
- Hire helpers, hand out potatoes and knives
- But too many chefs and you spend all your time coordinating

Concurrency:

- Lots of cooks making different things,  
but only 4 stove burners in the kitchen
- Want to allow access to all 4 burners,  
but not cause spills or incorrect burner settings

# Parallelism Example

**Parallelism:** Use extra resources to solve a problem faster  
(increasing throughput via simultaneous execution)

*Pseudocode* for array sum

- No 'FORALL' construct in Java, but we will see something similar
- Bad style for reasons we'll see, but may get roughly 4x speedup

```
int sum(int[] arr) {
    result = new int[4];
    len = arr.length;
    FORALL(i=0; i < 4; i++) { //parallel iterations
        result[i] = sumRange(arr, i*len/4, (i+1)*len/4);
    }
    return result[0]+result[1]+result[2]+result[3];
}

int sumRange(int[] arr, int lo, int hi) {
    result = 0;
    for(j=lo; j < hi; j++)
        result += arr[j];
    return result;
}
```

# Concurrency Example

**Concurrency:** Correctly and efficiently manage access to shared resources  
(from multiple possibly-simultaneous clients)

*Pseudocode* for a shared chaining hashtable

- Prevent *bad interleavings* (critical ensure correctness)
- But allow some concurrent access (critical to preserve performance)

```
class Hashtable<K,V> {
    ...
    void insert(K key, V value) {
        int bucket = ...;
        prevent-other-inserts/lookups in table[bucket]
        do the insertion
        re-enable access to arr[bucket]
    }
    V lookup(K key) {
        (similar to insert,
        but can allow concurrent lookups to same bucket)
    }
}
```

# *Shared Memory with Threads*

The model we will assume is **shared memory** with **explicit threads**

**Old story:** A running program has

- One *program counter* (the current statement that is executing)
- One *call stack* (with each *stack frame* holding local variables)
- Objects in the *heap* created by memory allocation (i.e., **new**) (same name, but no relation to the heap data structure)
- *Static* fields in the class shared among objects

**New story:**

- A set of *threads*, each with a program and call stack
  - No access to another thread's local variables
- Threads can implicitly share objects and static fields
  - To *communicate among threads*, write values to a shared location that another thread reads

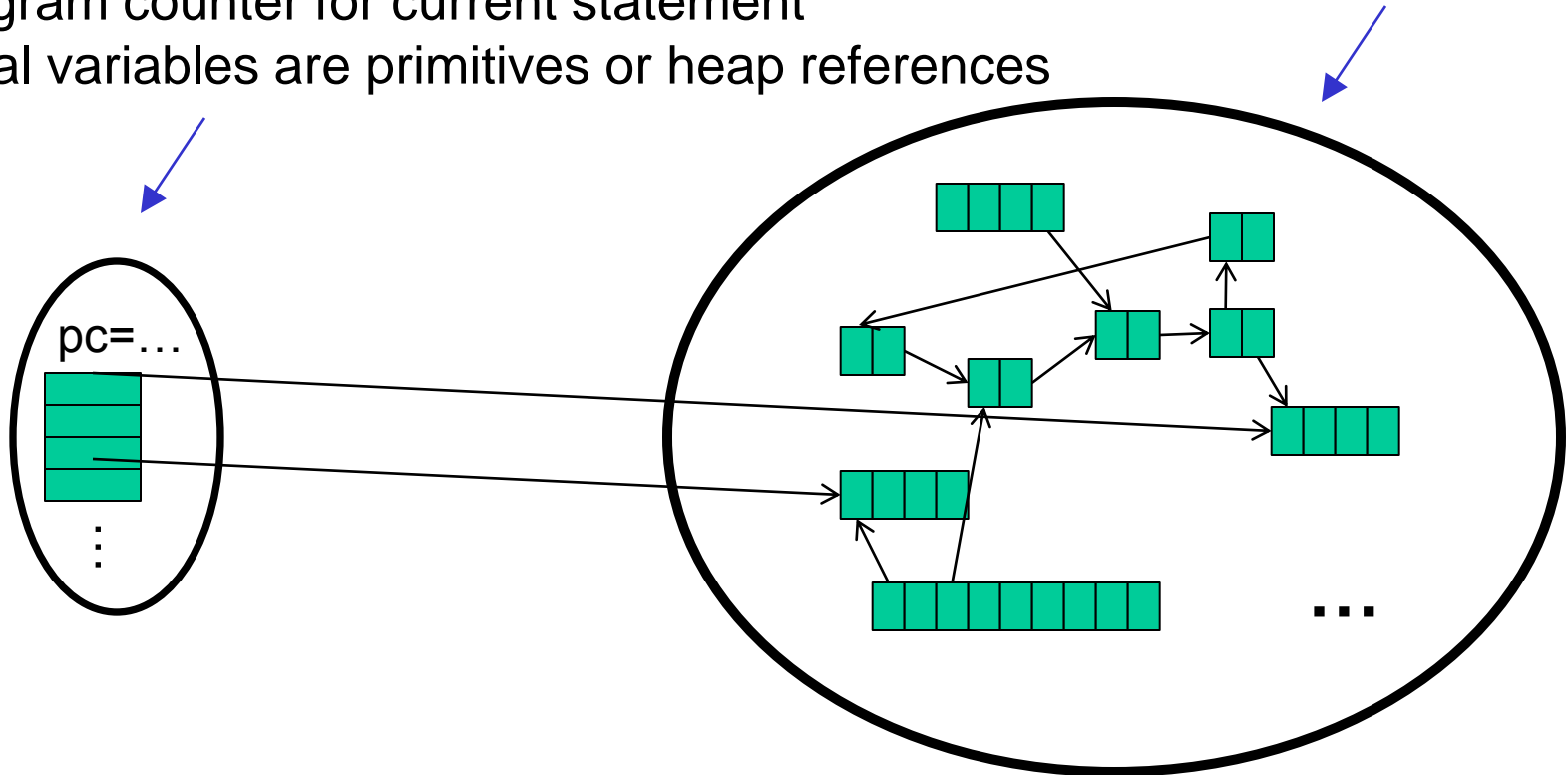
# Old Story: Single-Threaded

Call stack with local variables

Program counter for current statement

Local variables are primitives or heap references

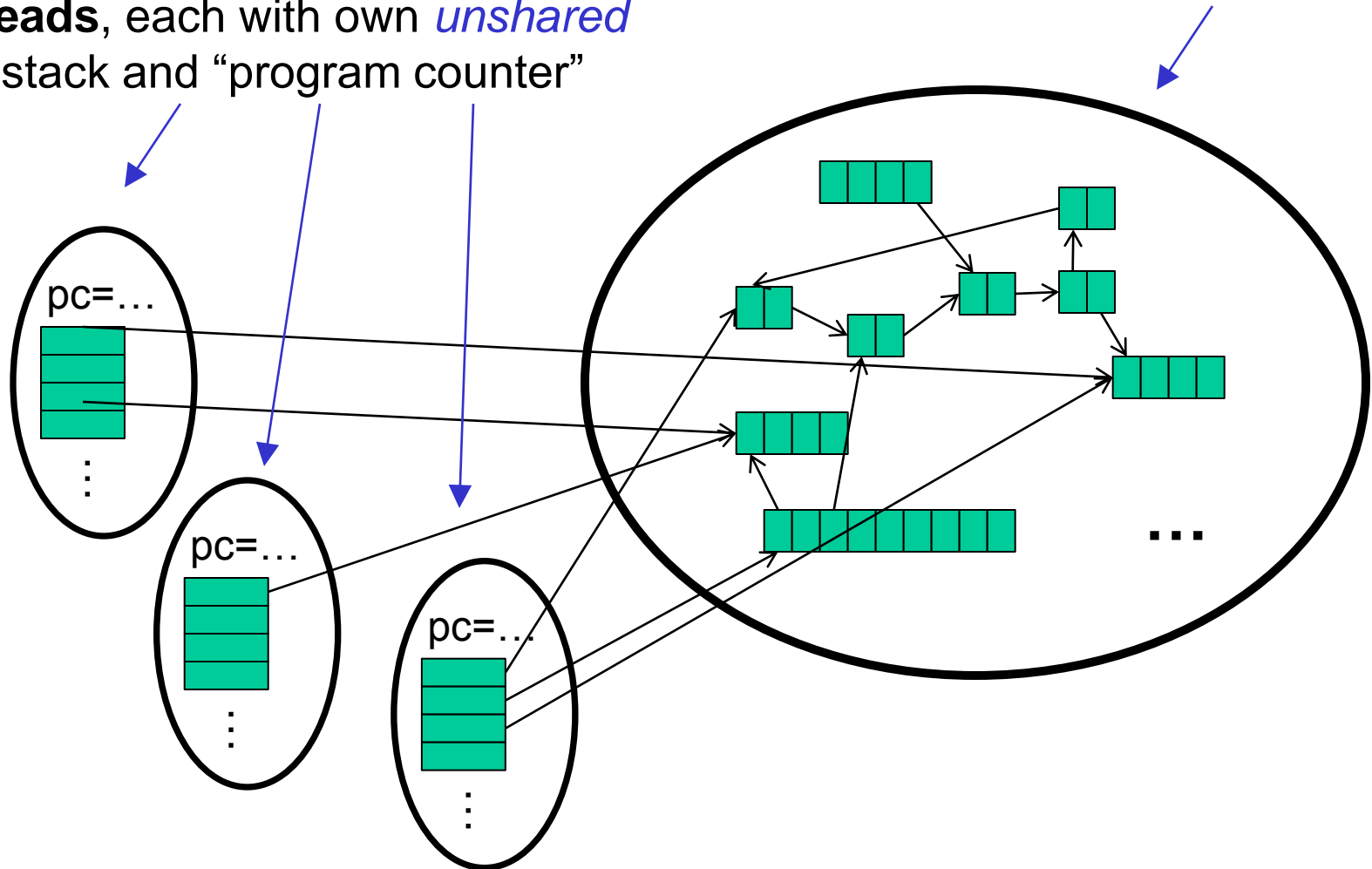
**Heap** for all objects  
and static fields



# New Story: Shared Memory with Threads

**Threads**, each with own *unshared* call stack and “program counter”

**Heap** for all objects and static fields, *shared* by all threads





# *Other Models*

We will focus on shared memory, but you should know several other models exist and have their own advantages

- **Message-passing:** Each thread has its own collection of objects. Communication is via explicitly sending/receiving messages
  - Cooks working in separate kitchens, mail around ingredients
- **Dataflow:** Programmers write programs in terms of a DAG. A node executes after all of its predecessors in the graph
  - Cooks wait to be handed results of previous steps
- **Data parallelism:** Have primitives for things like “apply function to every element of an array in parallel”

# *Our Needs*

To write a shared-memory parallel program, need new primitives from a programming language or library

- Ways to create and *run multiple things at once*
  - Let's call these things threads
- Ways for threads to *share memory*
  - Often just have threads with references to the same objects
- Ways for threads to *coordinate* (a.k.a. *synchronize*)
  - For now, a way for one thread to wait for another to finish
  - Other primitives when we study concurrency

# *Java Basics*

First learn some basics built into Java via `java.lang.Thread`

- Then we will learn a better library for parallel programming

To get a new thread running:

1. Define a subclass `C` of `java.lang.Thread`, overriding `run`
2. Create an object of class `C`
3. Call that object's `start` method
  - `start` sets off a new thread, using `run` as its “main”

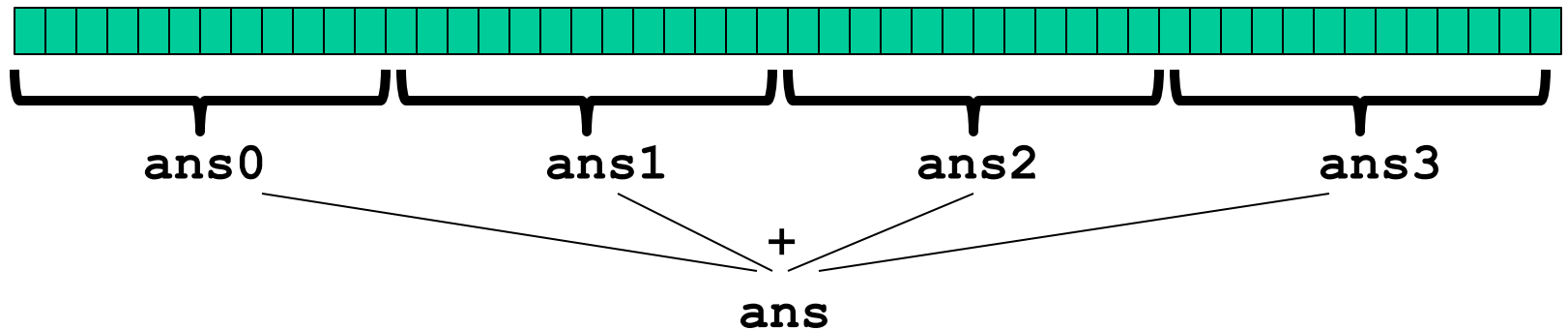
What if we instead called the `run` method of `C`?

- This would just be a normal method call, in the current thread

Then see how to share memory and coordinate via an example...

# Parallelism Idea

- Example: Sum elements of a large array
- Idea Have 4 threads simultaneously sum 1/4 of the array
  - Warning: This is the inferior first approach, do not do this



- Create 4 *thread objects*, each given a portion of the work
- Call `start()` on each thread object to actually *run* it in parallel
- Somehow 'wait' for threads to finish
- Add together their 4 answers for the *final result*

# *First Attempt: The Thread*

```
class SumThread extends java.lang.Thread {  
  
    int lo; // arguments  
    int hi;  
    int[] arr;  
  
    int ans = 0; // result  
  
    SumThread(int[] a, int l, int h) {  
        lo=l; hi=h; arr=a;  
    }  
  
    public void run() { //override must have this type  
        for(int i=lo; i < hi; i++)  
            ans += arr[i];  
    }  
}
```

Because we override a no-arguments/no-result `run`, we use fields to communicate data across threads

## First Attempt: Creating Threads (*wrong*)

```
class SumThread extends java.lang.Thread {
    int lo, int hi, int[] arr; // arguments
    int ans = 0; // result
    SumThread(int[] a, int l, int h) { ... }
    public void run() { ... } // override
}
```

```
int sum(int[] arr) { // can be a static method
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++) // do parallel computations
        ts[i] = new SumThread(arr, i*len/4, (i+1)*len/4);
    for(int i=0; i < 4; i++) // combine results
        ans += ts[i].ans;
    return ans;
}
```

## Second Attempt: Starting Threads *(still wrong)*

```
class SumThread extends java.lang.Thread {
    int lo, int hi, int[] arr; // arguments
    int ans = 0; // result
    SumThread(int[] a, int l, int h) { ... }
    public void run() { ... } // override
}
```

```
int sum(int[] arr) { // can be a static method
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++) { // do parallel computations
        ts[i] = new SumThread(arr, i*len/4, (i+1)*len/4);
        ts[i].start(); // start not run
    }
    for(int i=0; i < 4; i++) // combine results
        ans += ts[i].ans;
    return ans;
}
```

# *Join: Our ‘Wait for Thread’ Method*

- The **Thread** class defines various methods that provide primitive operations you could not implement on your own
  - For example: **start**, which calls **run** in a new thread
- The **join** method is another such method, essential for coordination in this kind of computation
  - Caller blocks until/unless the receiver is done executing (meaning its **run** method returns after its execution)
  - Without join, we would have a ‘**race condition**’ on **ts[i].ans**
    - In short, problem if variable can be read/written simultaneously
- This style of parallel programming is called “fork/join”
  - If we write in this style, we avoid many concurrency issues
  - But certainly not all of them



## *Third Attempt: Correct in Spirit*

```
class SumThread extends java.lang.Thread {
    int lo, int hi, int[] arr; // arguments
    int ans = 0; // result
    SumThread(int[] a, int l, int h) { ... }
    public void run() { ... } // override
}
```

```
int sum(int[] arr) { // can be a static method
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++) { // do parallel computations
        ts[i] = new SumThread(arr, i*len/4, (i+1)*len/4);
        ts[i].start();
    }
    for(int i=0; i < 4; i++) { // combine results
        ts[i].join(); // wait for helper to finish!
        ans += ts[i].ans;
    }
    return ans;
}
```

# *Shared Memory?*

- Fork-join programs thankfully do not require a lot of focus on sharing memory among threads
- But in languages like Java, there is memory being shared
- In our example:
  - `lo`, `hi`, `arr` fields written by “main” thread, read by helper thread
  - `ans` field written by helper thread, read by “main” thread
- When using shared memory, you must avoid race conditions
  - While studying parallelism, we’ll stick with `join`
  - With concurrency, we’ll learn other ways to synchronize



# CSE332: Data Abstractions

## Lecture 16: Into to Parallelism and Concurrency

James Fogarty

Winter 2012

## *From Our Previous Lecture*

```
class SumThread extends java.lang.Thread {
    int lo, int hi, int[] arr; // arguments
    int ans = 0; // result
    SumThread(int[] a, int l, int h) { ... }
    public void run() { ... } // override
}
```

```
int sum(int[] arr) { // can be a static method
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++) { // do parallel computations
        ts[i] = new SumThread(arr, i*len/4, (i+1)*len/4);
        ts[i].start();
    }
    for(int i=0; i < 4; i++) { // combine results
        ts[i].join(); // wait for helper to finish!
        ans += ts[i].ans;
    }
    return ans;
}
```

# *A Better Approach*

Several reasons why this is a poor parallel algorithm

1. Want code to be reusable and efficient across platforms
  - “Forward-portable” as core count grows
  - So at the very least, parameterize by the number of threads

```
int sum(int[] arr, int numThreads) {
    ... // note: shows idea, but has integer-division bug
    int subLen = arr.length / numThreads;
    SumThread[] ts = new SumThread[numThreads];
    for(int i=0; i < numThreads; i++) {
        ts[i] = new SumThread(arr, i*subLen, (i+1)*subLen);
        ts[i].start();
    }
    for(int i=0; i < numThreads; i++) {
        ...
    }
    ...
}
```

# *A Better Approach*

2. Want to use only the processors “available to you now”
  - Not used by other programs or threads in your program
    - Maybe caller is also using parallelism
    - Available cores can change even while your threads run
  - If 3 processors available and 3 threads would take time **x**, creating 4 threads can have worst-case time of **1.5x**

```
// numThreads == numProcessors is bad
// if some are needed for other things
int sum(int[] arr, int numThreads) {
    ...
}
```

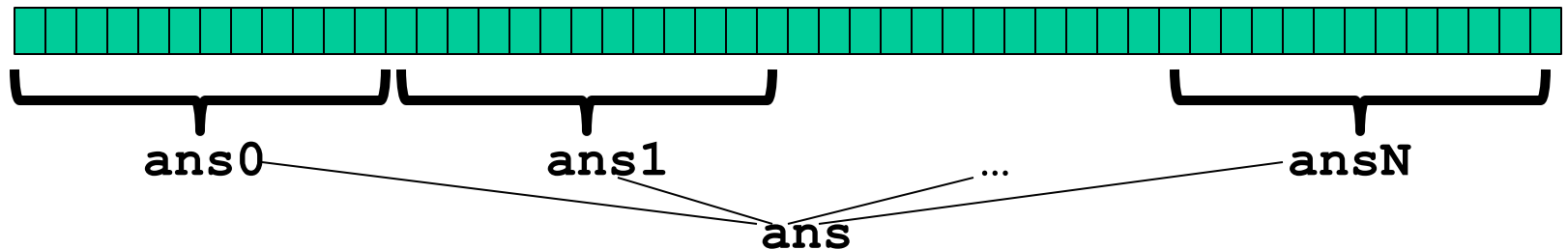
# *A Better Approach*

3. Though unlikely for `sum`, in general subproblems may take significantly different amounts of time
  - Example: Apply method `f` to every array element, but maybe `f` is much slower for some data items
    - Example: Is a large integer prime?
  - If we create 4 threads and all the slow data is processed by 1 of them, we won't get nearly a 4x speedup
    - Example of a [load imbalance](#)

# A Better Approach

The perhaps counterintuitive solution to all these problems is:  
to use lots of threads, far more than the number of processors

- When a processor finishes a piece, it can start another
- Require a different algorithm, and will abandon Java threads



1. **Forward-Portable:** Lots of helpers each doing a small piece
2. **Processors Available:** Hand out “work chunks” as you go
  - If 3 processors available and have 100 threads, worst-case extra time is  $< 3\%$  (if we ignore constant factors and load imbalance)
3. **Load Imbalance:** No problem if slow thread scheduled early enough
  - Variation probably small if pieces of work are small



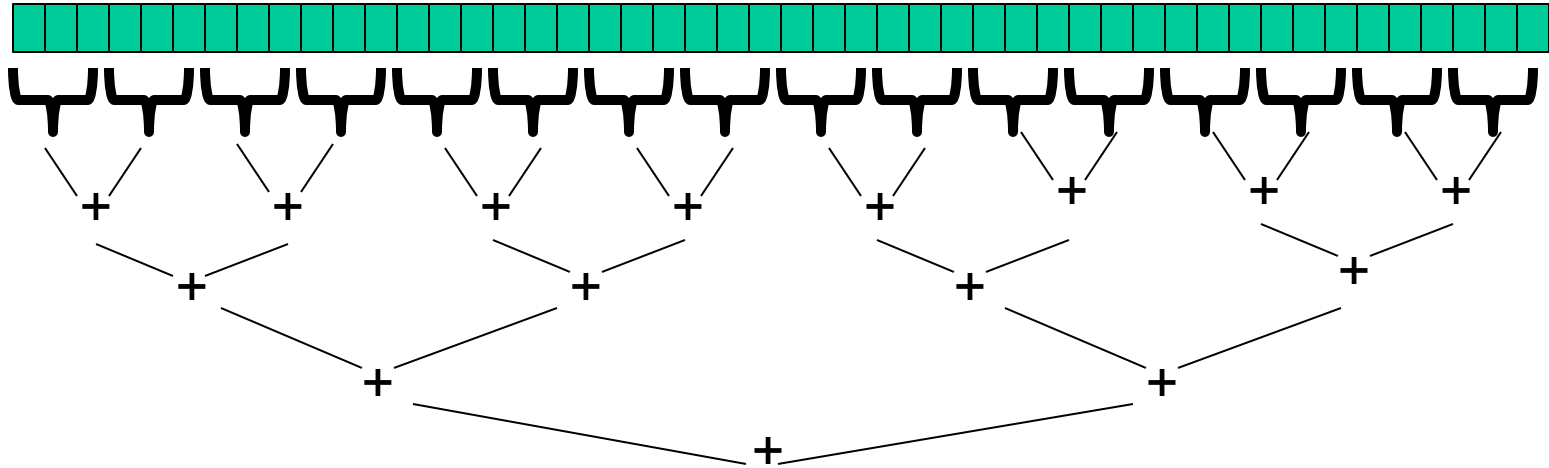
# Naïve Algorithm is Poor

- Suppose we create 1 thread to process every 100 elements

```
int sum(int[] arr) {  
    ...  
    // How many pieces of size 100 do we have?  
    int numThreads = arr.length / 100;  
    SumThread[] ts = new SumThread[numThreads];  
    ...  
}
```

- Combining results will require `arr.length / 100` additions
  - Linear in size of array
  - Previously we only had 4 pieces,  $\Theta(1)$  to combine
- In the extreme, suppose we create one thread per element
  - Using a loop to combine the results requires  $N$  iterations

# *A Better Idea*



This is straightforward to implement using divide-and-conquer

- Parallelism for the recursive calls

Halve and make new thread until size is at some cutoff

Combine answers in pairs as we return

This will start small, and 'grow' threads to fit the problem

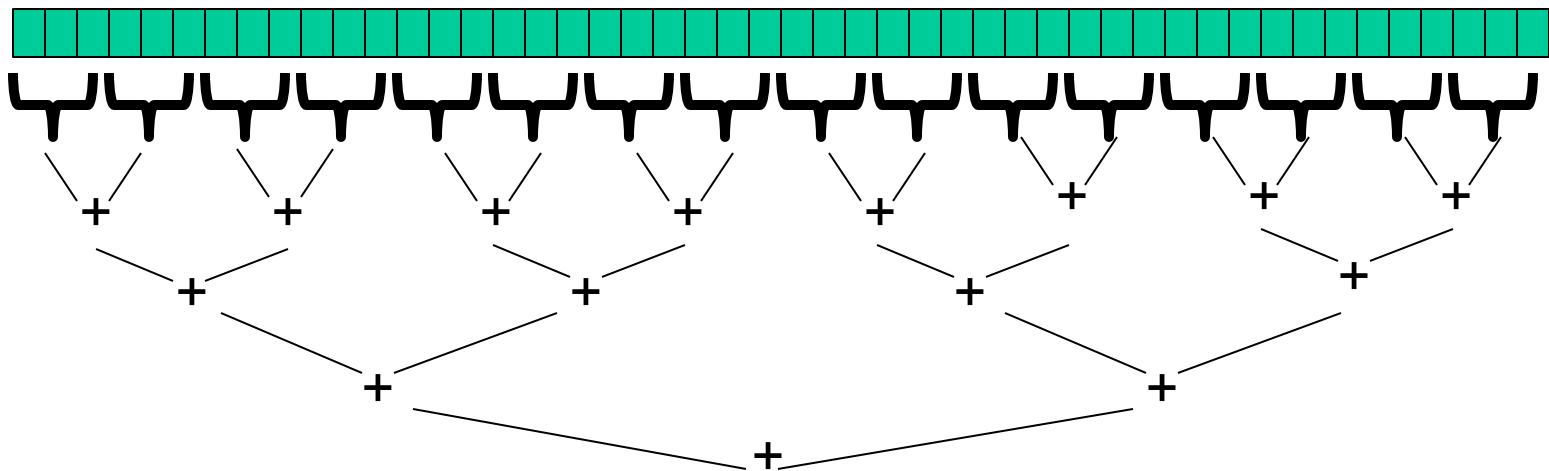
# Divide-and-Conquer

```
class SumThread extends java.lang.Thread {
    int lo; int hi; int[] arr; // arguments
    int ans = 0; // result
    SumThread(int[] a, int l, int h) { ... }
    public void run() { // override
        if(hi - lo < SEQUENTIAL CUTOFF)
            for(int i=lo; i < hi; i++)
                ans += arr[i];
        else {
            SumThread left = new SumThread(arr, lo, (hi+lo)/2);
            SumThread right = new SumThread(arr, (hi+lo)/2, hi);
            left.start();
            right.start();
            left.join(); // don't move this up a line - why?
            right.join();
            ans = left.ans + right.ans;
        }
    }
}

int sum(int[] arr) {
    SumThread t = new SumThread(arr, 0, arr.length);
    t.run();
    return t.ans;
}
```

# Divide-and-Conquer Really Works

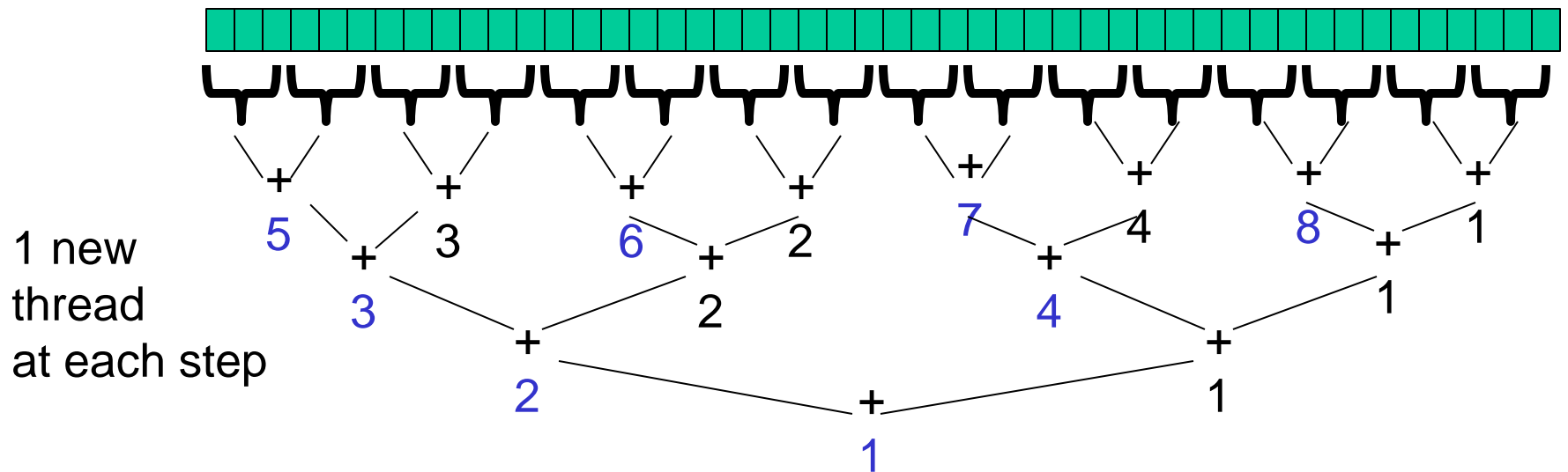
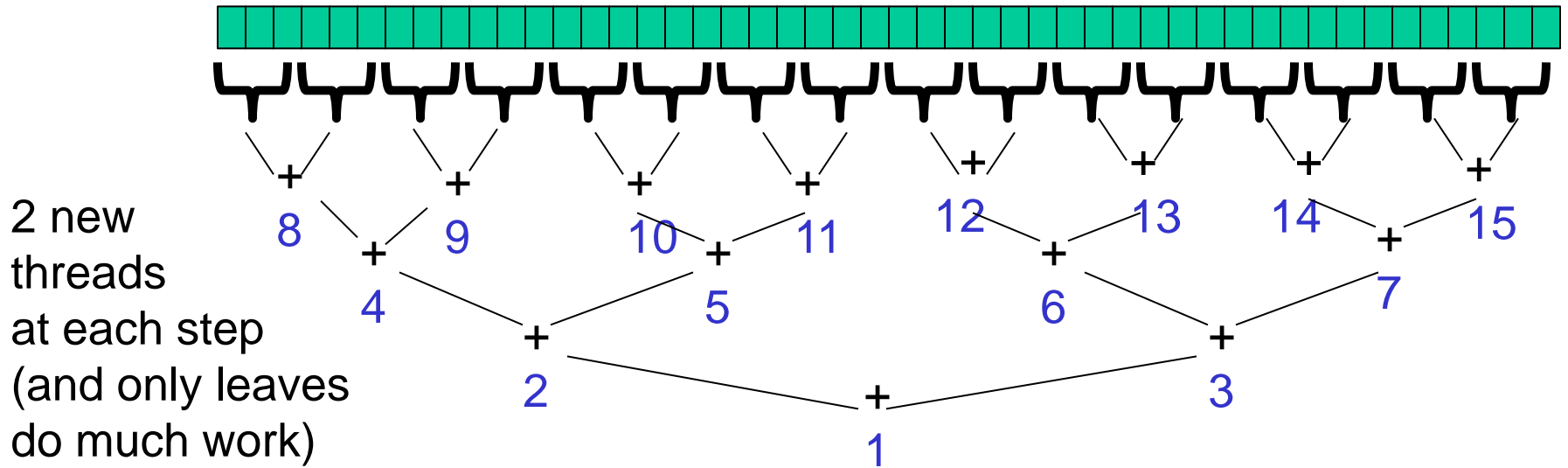
- The key is divide-and-conquer parallelizes the result-combining
  - If you have enough processors, total time is height of the tree:  $O(\log n)$  (optimal, exponentially faster than sequential  $O(n)$ )
- Will write our parallel algorithms in this style
  - But using a special library designed and engineered for this style
    - Takes care of scheduling the computation well
  - Often relies on operations being associative (as with +)



# *Being Realistic*

- In theory, you can divide down to single elements, do all your result-combining in parallel and get optimal speedup
- In practice, creating all those threads and communicating amongst them swamps the savings, so:
  - Use a *sequential cutoff*, typically around 500-1000
    - Eliminates *almost all* the recursive thread creation (because it eliminates the bottom levels of tree)
    - *Exactly* like quicksort switching to insertion sort for small subproblems, but more important here

# Illustration of Fewer Threads



# Half the Threads

```
// wasteful: don't
SumThread left = ...
SumThread right = ...
left.start();
right.start();
left.join();
right.join();
ans=left.ans+right.ans;
```

```
// better: do
SumThread left = ...
SumThread right = ...
// order of next 4 lines
// essential - why?
left.start();
right.run();
left.join();
ans=left.ans+right.ans;
```

# *Half the Threads*

Do not create two threads; create one and do the other “yourself”

- Cuts the number of threads created by 2x
- And the difference is surprisingly substantial

If a *language* had built-in support for fork-join parallelism, we would expect this hand-optimization to be unnecessary

The *library* we are using allows you to do it yourself

- `ForkJoinTask.invokeAll(...)` probably does something similar
- You will do this yourselves for the same reason you implement your own data structures

But no difference in theory or asymptotic analysis



# *The Library*

- Even with all this care, Java's threads are too "heavyweight"
  - Constant factors, especially space overhead
  - Creating 20,000 Java threads just a bad idea
- The [ForkJoin Framework](#) is designed and engineered to meet the needs of divide-and-conquer fork-join parallelism
  - Included in the Java 7 standard libraries
    - Also available as a downloaded `.jar` file for Java 6
  - Section will discuss some pragmatics/logistics
  - Similar libraries available for other languages
    - C/C++: Cilk, Intel's Thread Building Blocks
    - C#: Task Parallel Library
  - Library implementation is an advanced topic

# *Different Terms but Same Basic Idea*

To use the [ForkJoin Framework](#):

- A little standard set-up code (e.g., create a `ForkJoinPool`)

Don't subclass <code>Thread</code>
Don't override <code>run</code>
Don't use an <code>ans</code> field
Don't call <code>start</code>
Don't just call <code>join</code>
Don't call <code>run</code> to hand-optimize
Don't have topmost call to <code>run</code>

Java Threads

Do subclass <code>RecursiveTask&lt;V&gt;</code>
Do override <code>compute</code>
Do return a <code>V</code> from <code>compute</code>
Do call <code>fork</code>
Do call <code>join</code> which returns answer
Do call <code>compute</code> to hand-optimize
Do create a pool and call <code>invoke</code>
See <code>ForkJoinTask.invokeAll(...)</code>

ForkJoin Framework

See the Dan's web page for

["A Beginner's Introduction to the ForkJoin Framework"](#)

# Example: Final Version in ForkJoin Framework

```
class SumArray extends RecursiveTask<Integer> {
    int lo; int hi; int[] arr; // arguments
    SumArray(int[] a, int l, int h) { ... }
    protected Integer compute() { // return answer
        if(hi - lo < SEQUENTIAL_CUTOFF) {
            int ans = 0;
            for(int i=lo; i < hi; i++)
                ans += arr[i];
            return ans;
        } else {
            SumArray left = new SumArray(arr, lo, (hi+lo)/2);
            SumArray right = new SumArray(arr, (hi+lo)/2, hi);
            left.fork();
            int rightAns = right.compute();
            int leftAns = left.join();
            return leftAns + rightAns;
        }
    }
}

static final ForkJoinPool fjPool = new ForkJoinPool();
int sum(int[] arr) {
    return fjPool.invoke(new SumArray(arr, 0, arr.length));
}
```

## For Comparison: Java Threads Version

```
class SumThread extends java.lang.Thread {
    int lo; int hi; int[] arr; //fields to know what to do
    int ans = 0; // for communicating result
    SumThread(int[] a, int l, int h) { ... }
    public void run() {
        if(hi - lo < SEQUENTIAL CUTOFF)
            for(int i=lo; i < hi; i++)
                ans += arr[i];
        else { // create 2 threads, each will do 1/2 the work
            SumThread left = new SumThread(arr, lo, (hi+lo)/2);
            SumThread right = new SumThread(arr, (hi+lo)/2, hi);
            left.start();
            right.start();
            left.join(); // don't move this up a line - why?
            right.join();
            ans = left.ans + right.ans;
        }
    }
}

class C {
    static int sum(int[] arr) {
        SumThread t = new SumThread(arr, 0, arr.length);
        t.run(); // only creates one thread
        return t.ans;
    }
}
```

# *Getting Good Results in Practice*

- Sequential threshold
  - Library documentation recommends doing approximately 100-5000 basic operations in each “piece” of your algorithm
- Library needs to “warm up”
  - May see slow results before the Java virtual machine re-optimizes the library internals
  - When evaluating speed, put your computations in a loop to see the “long-term benefit” after these optimizations have occurred
- Wait until your computer has more processors
  - Seriously, overhead may dominate at 4 processors, but parallel programming is likely to become much more important
- Beware memory-hierarchy issues
  - Will not focus on this, but can be crucial for parallel performance

# Work and Span

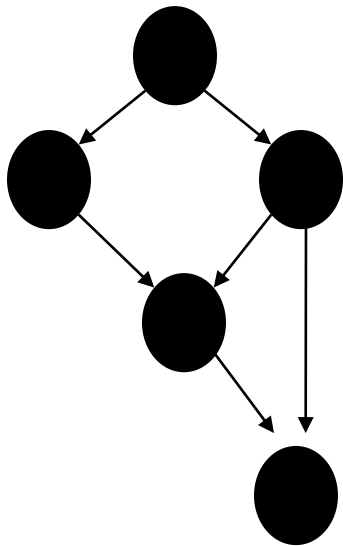
Let  $T_P$  be the running time if there are  $P$  processors available

Two key measures of run-time:

- **Work**: How long it would take 1 processor =  $T_1$ 
  - Just “sequentialize” the recursive forking
- **Span**: How long it would take infinity processors =  $T_\infty$ 
  - The longest dependence-chain
  - Example:  $O(\log n)$  for summing an array
    - Notice having  $> n/2$  processors is no additional help
  - Also called “critical path length” or “computational depth”

# The DAG

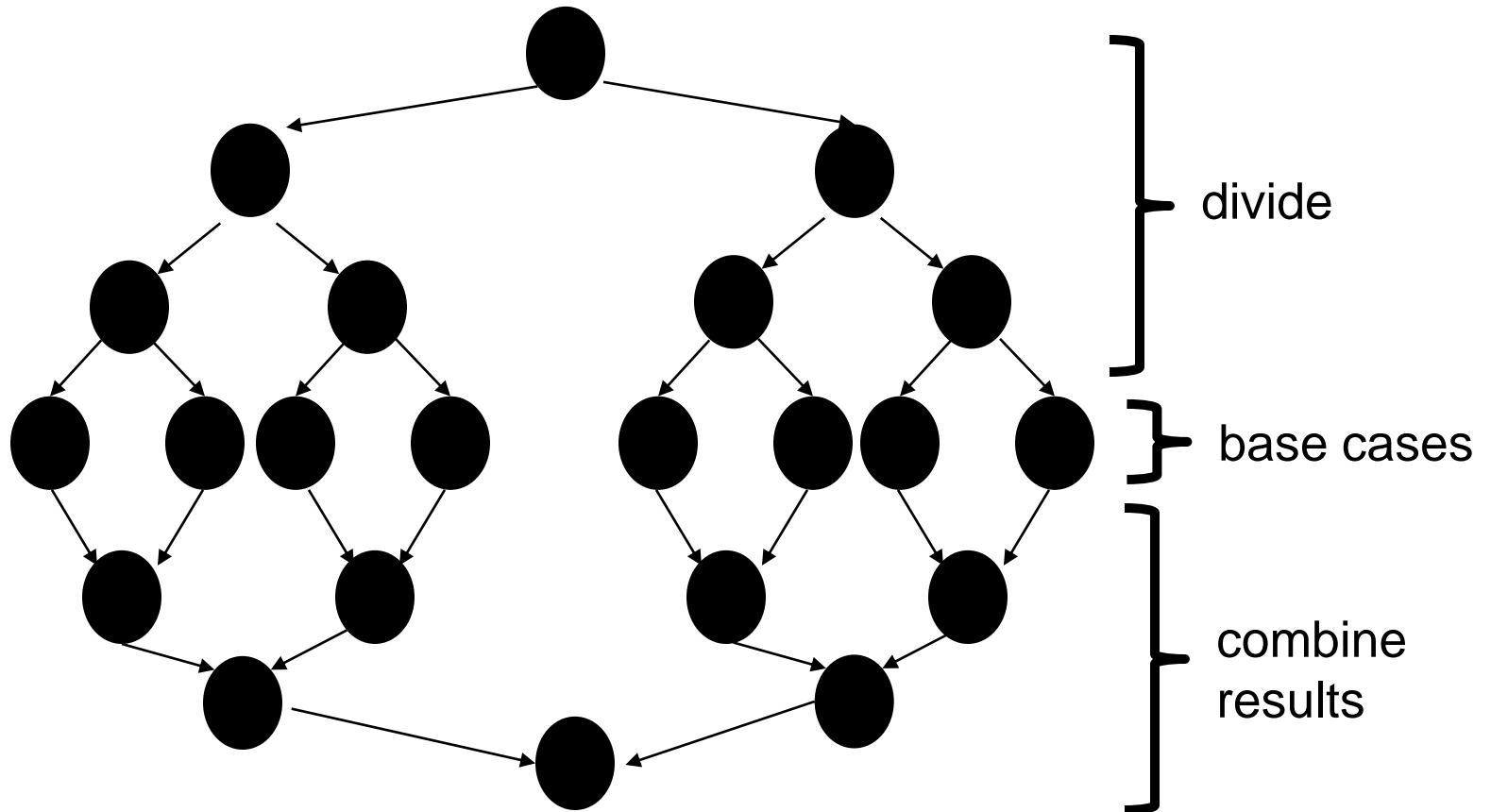
- A program execution using **fork** and **join** can be seen as a DAG
  - Nodes: Pieces of work
  - Edges: Source must finish before destination starts



- A **fork** “ends a node” and makes two outgoing edges
  - New thread
  - Continuation of current thread
- A **join** “ends a node” and makes a node with two incoming edges
  - Node just ended
  - Last node of thread joined on

# *Our Simple Examples*

- **fork** and **join** are very flexible, but divide-and-conquer maps and reductions use them in a very basic way:
  - A tree on top of an upside-down tree



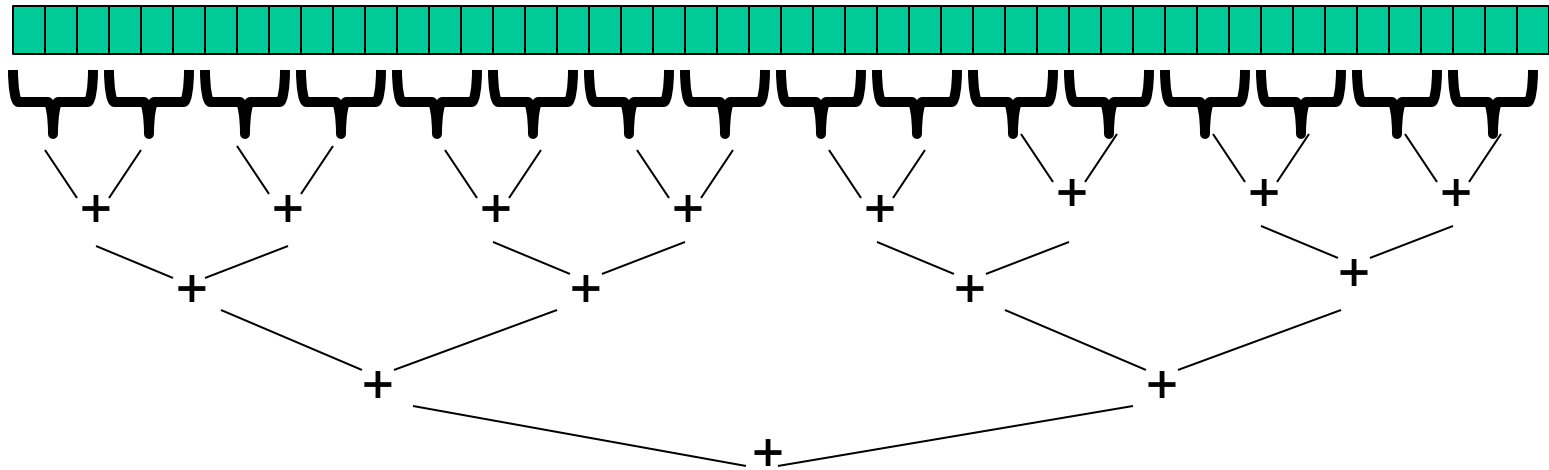


# *More Interesting DAGs?*

- The DAGs are not always this simple
- Example:
  - Suppose combining two results might be expensive enough that we want to parallelize each one
  - Then each node in the inverted tree on the previous slide would itself expand into another set of nodes for that parallel computation

# What Else Looks Like This?

- Summing an array went from  $O(n)$  sequential to  $O(\log n)$  parallel (assuming **a lot** of processors and very large  $n$ )
  - An exponential speed-up in theory



- Anything that can use results from two halves and merge them in  $O(1)$  time has the same property...

# *Examples*

- Maximum or minimum element
- Is there an element satisfying some property (e.g., is there a 17)?
- Left-most element satisfying some property (e.g., first 17)
  - What should the recursive tasks return?
  - How should we merge the results?
- Corners of a rectangle containing all points (a “bounding box”)
- Counts, for example, number of strings that start with a vowel
  - This is just summing with a different base case

# *Reductions*

- Computations of this form are called **reductions** (or **reduces**?)
- Produce single answer from collection via an **associative operator**
  - Examples: max, count, leftmost, rightmost, sum, ...
  - Non-example: median
- Recursive results don't have to be single numbers or strings. They can be arrays or objects with multiple fields.
  - Example: Histogram of test results is a variant of sum
- But some things are inherently sequential
  - How we process `arr[i]` may depend entirely on the result of processing `arr[i-1]`

# Maps and Data Parallelism

- A `map` operates on each element of a collection independently to create a new collection of the same size
  - No combining results
  - For arrays, this is so trivial some hardware has direct support
- Canonical example: Vector addition

```
int[] vector add(int[] arr1, int[] arr2) {  
    assert (arr1.length == arr2.length);  
    result = new int[arr1.length];  
    FORALL(i=0; i < arr1.length; i++) {  
        result[i] = arr1[i] + arr2[i];  
    }  
    return result;  
}
```

# Maps in ForkJoin Framework

```
class VecAdd extends RecursiveAction {
    int lo; int hi; int[] res; int[] arr1; int[] arr2;
    VecAdd(int l, int h, int[] r, int[] a1, int[] a2) { ... }
    protected void compute() {
        if (hi - lo < SEQUENTIAL_CUTOFF) {
            for (int i = lo; i < hi; i++)
                res[i] = arr1[i] + arr2[i];
        } else {
            int mid = (hi + lo) / 2;
            VecAdd left = new VecAdd(lo, mid, res, arr1, arr2);
            VecAdd right = new VecAdd(mid, hi, res, arr1, arr2);
            left.fork();
            right.compute();
            left.join();
        }
    }
}

static final ForkJoinPool fjPool = new ForkJoinPool();
int[] add(int[] arr1, int[] arr2) {
    assert (arr1.length == arr2.length);
    int[] ans = new int[arr1.length];
    fjPool.invoke(new VecAdd(0, arr1.length, ans, arr1, arr2));
    return ans;
}
```

# *Maps and Reductions*

Maps and reductions: the “workhorses” of parallel programming

- By far the two most important and common patterns
  - We will discuss two more advanced patterns later
- Learn to recognize when an algorithm can be written in terms of maps and reductions
- Often Use maps and reductions to describe parallel algorithms
- Programming them becomes “trivial” with a little practice
  - Exactly like sequential for-loops seem second-nature

# *Digression: MapReduce on Clusters*

- You may have heard of Google's "map/reduce"
  - Or the open-source version Hadoop
- Idea: Perform maps/reduces on data using many machines
  - The system takes care of distributing the data and managing fault tolerance
  - You just write code to map one element and reduce elements to a combined result
- Separates how to do recursive divide-and-conquer from what computation to perform
  - Old idea in higher-order functional programming transferred to large-scale distributed computing
  - Complementary approach to declarative queries for databases

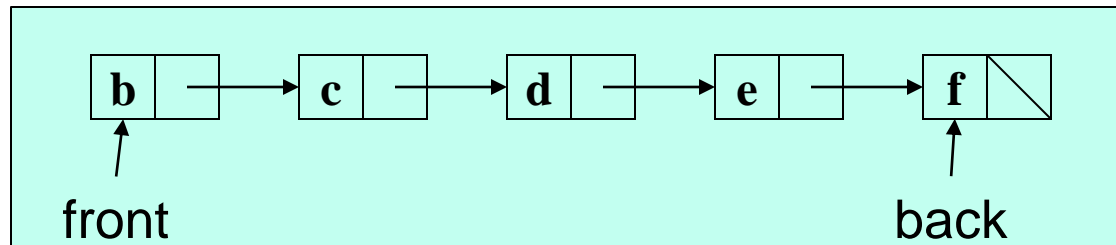


# Trees

- Maps and reductions work just fine on balanced trees
  - Divide-and-conquer each child rather than array subranges
  - Correct for unbalanced trees, but won't get much speed-up
- Example: minimum element in an unsorted but balanced binary tree in  $O(\log n)$  time given enough processors
- How to do the sequential cut-off?
  - Store number-of-descendants at each node (easy to maintain)
  - Or could approximate it with, e.g., AVL-tree height

# Linked Lists

- Can you parallelize maps or reduces over linked lists?
  - Example: Increment all elements of a linked list
  - Example: Sum all elements of a linked list



- Once again, data structures matter!
- For parallelism, balanced trees generally better than lists so that we can get to all the data exponentially faster  $O(\log n)$  vs.  $O(n)$ 
  - Trees have the same flexibility as lists compared to arrays



# CSE332: Data Abstractions

## Lecture 17: Parallel Analysis and Parallel Prefix

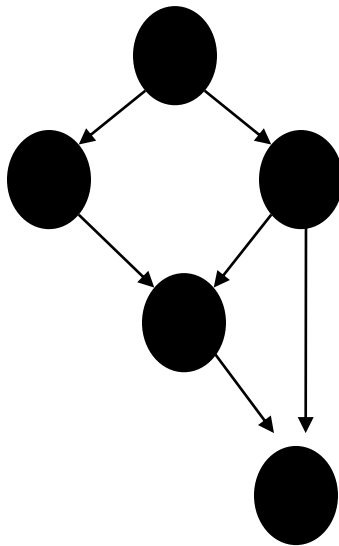
James Fogarty

Winter 2012

Including slides developed in part by  
Ruth Anderson, James Fogarty, Dan Grossman

# *Work and Span in the DAG*

- **fork** and **join** execution can be seen as a DAG
  - Nodes: Pieces of work
  - Edges: Source must finish before destination starts



- A **fork** “ends a node” and makes two outgoing edges
  - New thread
  - Continuation of current thread
- A **join** “ends a node” and makes a node with two incoming edges
  - Node just ended
  - Last node of thread joined on

# Work and Span

Run-time costs are on the **nodes**, not on the edges

- **Work:**  $T_1$  = sum of all nodes in the DAG
  - One processor has to do all the work
  - Any topological sort is a legal execution
- **Span:**  $T_\infty$  = sum of all nodes on **most-expensive** path in the DAG
  - Can do everything that is ready, but still must wait for results
  - If all nodes are roughly equal cost, this is the **longest** path
  - Example:  $O(\log n)$  for summing an array
    - Notice having  $> n/2$  processors is no additional help

Let  $T_P$  be the running time if there are  $P$  processors available

# More Definitions

- **Speed-up** on  $P$  processors:  $T_1 / T_P$
- If speed-up is  $P$  as we vary  $P$ , we call it **perfect linear speed-up**
  - Perfect linear speed-up means doubling  $P$  halves running time
  - Usually our goal; hard to get in practice
- **Parallelism** is the maximum possible speed-up:  $T_1 / T_\infty$ 
  - At some point, adding processors will not help
  - What that point is depends on the span

Parallel algorithms are about decreasing span without increasing work  
*... or at least not increasing work too much ...*

# Optimal $T_P$

- So we know  $T_1$  and  $T_\infty$  but actually care about  $T_P$  (e.g.,  $P=4$ )
- Ignoring memory-hierarchy issues,  $T_P$  cannot beat
  - $T_1 / P$       why not?
  - $T_\infty$       why not?
- So an *asymptotically* optimal execution would be:

$$T_P = O((T_1 / P) + T_\infty)$$

First term dominates for small  $P$ , second for large  $P$

# *Division of Responsibility*

- Our job as users of a ForkJoin Framework:
  - Pick a good algorithm, write a program
  - When run, it creates a DAG of things to do
  - Make all nodes small-ish and approximately equal work
- The job of the framework developer:
  - Assign work to available processors to avoid **idling**
  - Keep constant factors low
  - Give the **expected-time optimal guarantee**

$$T_p = O((T_1 / P) + T_\infty)$$

assuming framework-user did their job

- We will not study how the framework does this



# *What That Means: Mostly Good News*

The fork-join framework guarantee:

$$T_P = O((T_1 / P) + T_\infty)$$

- No implementation can beat  $O(T_\infty)$  by more than a constant factor
- No implementation on  $P$  processors can beat  $O(T_1 / P)$
- So the framework on average gets within a constant factor of the best you can do, assuming framework user did their part correctly

You can focus on your algorithm, data structures, and cut-offs

Do not worry about number of processors and scheduling

- Analyze running time given  $T_1$ ,  $T_\infty$ , and  $P$

# Examples

$$T_P = O((T_1 / P) + T_\infty)$$

- In the algorithms seen so far (e.g., sum an array):
  - $T_1 = O(n)$
  - $T_\infty = O(\log n)$
  - So expect (ignoring overheads):  $T_P = O(n/P + \log n)$
- Suppose instead:
  - $T_1 = O(n^2)$
  - $T_\infty = O(n)$
  - So expect (ignoring overheads):  $T_P = O(n^2/P + n)$

# *Amdahl's Law: Mostly Bad News*

- We have analyzed a parallel program in terms of **work** and **span**
- In practice, it is common that your program has:
  - a) parts that **parallelize well**:
    - Such as maps/reduces over arrays and trees
  - b) ...and parts that **don't parallelize at all**:
    - Such as reading a linked list, getting input, or just doing computations where each step needs the results of previous step
- These **unparallelized** parts can turn out to be a big bottleneck

# *Amdahl's Law: Mostly Bad News*

Let the **work** be 1 unit time

Let **S** be the portion of the execution that cannot be parallelized

Then:  $T_1 = S + (1-S) = 1$

Suppose we get perfect linear speedup *on the parallel portion*

Then:  $T_p = S + (1-S)/P$

So the overall speedup with **P** processors (this is Amdahl's Law):

$$T_1 / T_p = 1 / (S + (1-S)/P)$$

And the parallelism is (with infinite processors):

$$T_1 / T_\infty = 1 / S$$

# *Amdahl's Law Example*

Suppose:

$$T_1 = S + (1-S) = 1 \text{ (aka total program execution time)}$$

$$T_1 = 1/3 + 2/3 = 1$$

$$T_1 = 33 \text{ sec} + 67 \text{ sec} = 100 \text{ sec}$$

Time on P processors:  $T_p = S + (1-S)/P$

So:

$$T_p = 33 \text{ sec} + (67 \text{ sec})/P$$

$$T_3 = 33 \text{ sec} + (67 \text{ sec})/3$$

$$T_3 = 33 \text{ sec} + 22.33 \text{ sec} = 55.33 \text{ sec}$$

# Why Such Bad News?

$$T_1 / T_P = 1 / (S + (1-S)/P)$$

$$T_1 / T_\infty = 1 / S$$

- Suppose 33% of a program is sequential
  - Then a billion processors will not give a speedup over 3
- Suppose you miss the good old days where you could get 100x speedup by just waiting about 12 years
- Now suppose in 12 years, clock speed is the same but you get 256 processors instead of 1
- For 256 processors to get at least 100x speedup, we need

$$100 \leq 1 / (S + (1-S)/256)$$

Which means  $S \leq .0061$  (i.e., 99.4% perfectly parallelizable)

# *Plots You Need to See*

1. Assume 256 processors
  - x-axis: sequential portion **S**, ranging from .01 to .25
  - y-axis: speedup  $T_1 / T_P$  (will go down as **S** increases)
2. Assume **S** = .01 or .1 or .25 (three separate lines)
  - x-axis: number of processors **P**, ranging from 2 to 32
  - y-axis: speedup  $T_1 / T_P$  (will go up as **P** increases)

*Do this as a homework problem!*

- More practice with a spreadsheet or graphing program
- Compare against your intuition
- A picture is worth 1000 words, especially if you made it

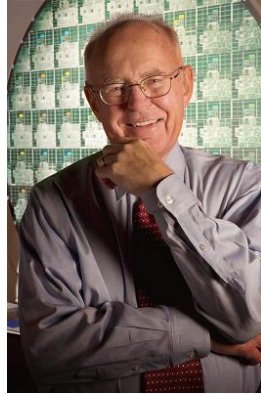
# *All is Not Lost*

Amdahl's Law is a harsh reality

- But it does not mean additional processors are worthless
- We can find new parallel algorithms
  - Some things that seem sequential are actually parallelizable
- We can change the problem or do new things
  - Video games use tons of parallel processors
    - They are not rendering 10-year-old graphics faster
    - They are rendering better monsters, better scenery



# *Moore and Amdahl*



- Moore's "Law" is an **observation** about the progress of the semiconductor industry
  - Transistor density doubles roughly every 18 months
- Amdahl's Law is a **mathematical theorem**
  - Implies diminishing returns of adding more processors
- Both are incredibly important in designing computer systems

# *Moving Forward*

Done:

- Simple ways to use parallelism for counting, summing, finding
- Analysis of running time and implications of Amdahl's Law

Now:

- Clever ways to parallelize more than is intuitively possible
- **Parallel prefix:**
  - This “key trick” typically underlies surprising parallelization
  - Enables other things like **packs**
- **Parallel sorting:** mergesort and quicksort (though not in place)
  - Easy to get a little parallelism
  - With cleverness can get a lot of parallelism

# The Prefix-Sum Problem

Given `int[] input`, produce `int[] output` where:

`output[i]` is the sum of `input[0]+input[1]+...+input[i]`

Sequential can be a CS1 exam problem:

```
int[] prefix_sum(int[] input) {
    int[] output = new int[input.length];
    output[0] = input[0];
    for(int i=1; i < input.length; i++)
        output[i] = output[i-1]+input[i];
    return output;
}
```

Does not seem parallelizable

- Work:  $O(n)$ , Span:  $O(n)$

This *algorithm* is sequential, but a *different algorithm* has

- Work:  $O(n)$ , Span:  $O(\log n)$

# Parallel Prefix-Sum

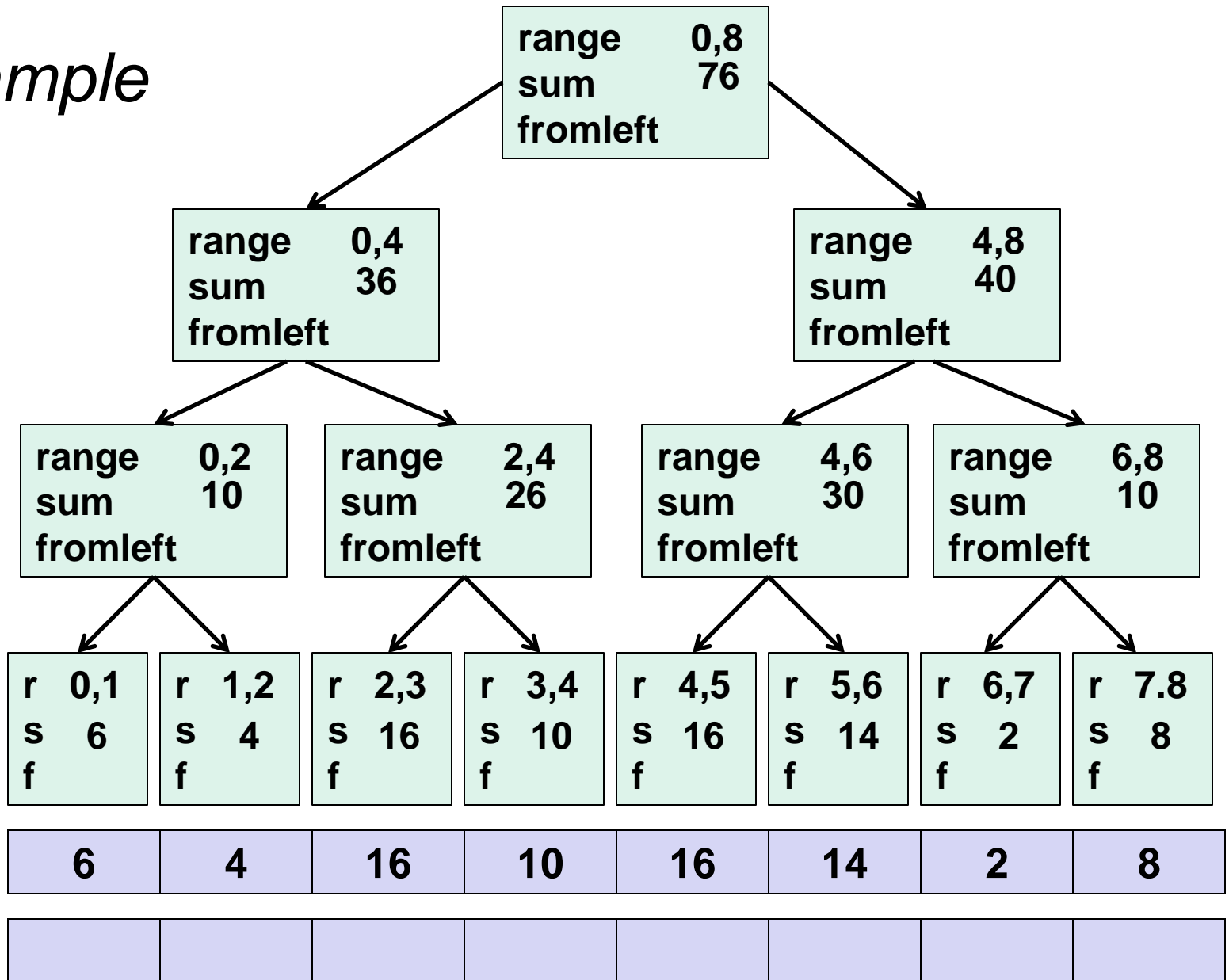
- The parallel-prefix algorithm does two passes
  - Each pass has  $O(n)$  work and  $O(\log n)$  span
  - So in total there is  $O(n)$  work and  $O(\log n)$  span
  - So just like with array summing, parallelism is  $n / \log n$
  - An exponential speedup
- The first pass builds a tree bottom-up: the “up” pass
- The second pass traverses the tree top-down: the “down” pass

Historical note:

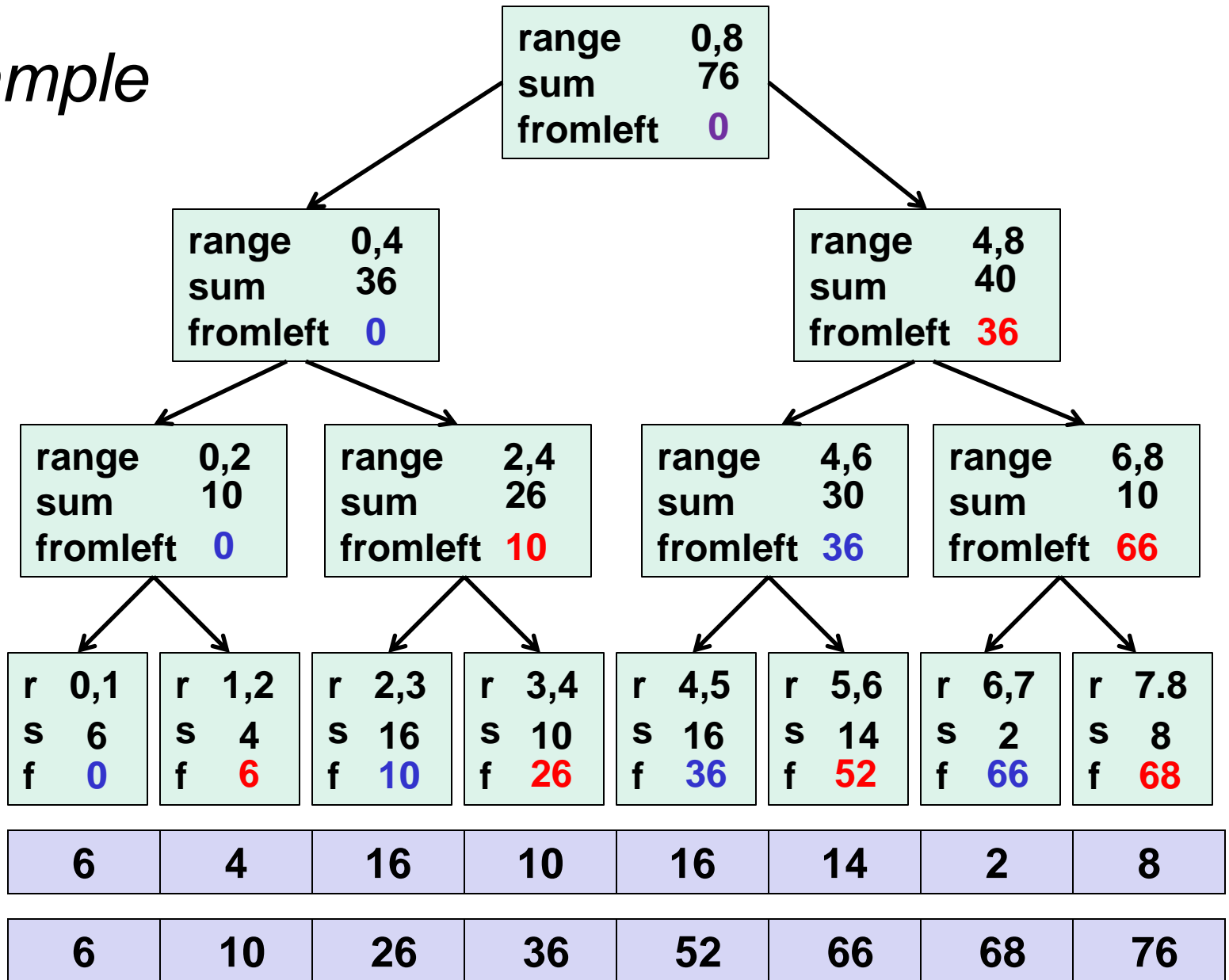


Original algorithm due to  
R. Ladner and M. Fischer at the  
University of Washington in 1977

# Example



# Example



# *The Algorithm: Part 1*

1. Up: Build a binary tree where
  - Root has sum of the range  $[x, y)$
  - If a node has sum of  $[lo, hi)$  and  $hi > lo$ ,
    - Left child has sum of  $[lo, middle)$
    - Right child has sum of  $[middle, hi)$
    - A leaf has sum of  $[i, i+1)$  *i.e., `input[i]`*

This is an easy fork-join computation:

combine results by actually building a binary tree with the range-sums

- Tree built bottom-up in parallel
- Could be more clever in an array, as we were with heaps

Analysis:  $O(n)$  work,  $O(\log n)$  span

## *The Algorithm: Part 2*

2. Down: Pass down a value **fromLeft**
  - Root given a **fromLeft** of 0
  - Node takes its **fromLeft** value and
    - Passes its left child
      - the same **fromLeft**
    - Passes its right child
      - its **fromLeft** plus its left child's **sum** (stored in part 1)
  - At the leaf for array position **i**,  
**output[i]=fromLeft+input[i]**

This is an easy fork-join computation:

traverse the tree built in step 1 and produce no result

- Leaves assign to **output**
- Invariant: **fromLeft** is sum of elements left of the node's range

Analysis:  $O(n)$  work,  $O(\log n)$  span



# *Sequential Cut-Off*

Adding a sequential cut-off is easy as always:

- Up:  
just a sum, have leaf node hold the sum of a range

- Down:

```
output[lo] = fromLeft + input[lo];  
for(i=lo+1; i < hi; i++)  
    output[i] = output[i-1] + input[i]
```

# *Generalizing Parallel Prefix*

Just as sum-array was the simplest example of a common pattern, prefix-sum illustrates a pattern that can be used in many problems

- Minimum, maximum of all elements to the left of  $i$
- Is there an element to the left of  $i$  satisfying some property?
- Count of elements to the left of  $i$  satisfying some property
  - This last one is perfect for an efficient parallel pack
  - Perfect for building on top of the “parallel prefix trick”

# *Pack*

[Non-standard terminology, `filter` does not emphasize stability]

Given an array `input`,  
produce an array `output`  
containing only elements such that `f(elt)` is `true`

Example: `input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]`  
`f: is elt > 10`  
`output [17, 11, 13, 19, 24]`

Parallelizable

- Finding elements for the output is easy
- But getting them in the right place seems hard

# *Parallel Map, Parallel Prefix, Parallel Map*

1. Parallel map to compute a **bit-vector** for true elements

`input` [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]

`bits` [1, 0, 0, 0, 1, 0, 1, 1, 0, 1]

2. Parallel-prefix sum on the bit-vector

`bitsum` [1, 1, 1, 1, 2, 2, 3, 4, 4, 5]

3. Parallel map to produce the output

`output` [17, 11, 13, 19, 24]

```
output = new array of size bitsum[n-1]
FORALL(i=0; i < input.length; i++){
    if(bits[i]==1)
        output[bitsum[i]-1] = input[i];
}
```

# *Pack Comments*

- First two steps can be combined into one pass
  - Use a different base case for the prefix sum
  - No effect on asymptotic complexity
- Can also combine third step into the down pass of the prefix sum
  - Again no effect on asymptotic complexity
- Analysis:  $O(n)$  work,  $O(\log n)$  span
  - Multiple passes, but this is a constant
- Parallelized packs will help us parallelize quicksort



# CSE332: Data Abstractions

## Lecture 18: Parallel Sort

James Fogarty  
Winter 2012

Including slides developed in part by  
Ruth Anderson, James Fogarty, Dan Grossman

# *Reductions*

- Computations of this form are called **reductions**
- Produce single answer from collection via an **associative operator**
  - Examples: max, count, leftmost, rightmost, sum, ...
  - Non-example: median
- Recursive results don't have to be single numbers or strings. They can be arrays or objects with multiple fields.
  - Example: Histogram of test results is a variant of sum
- But some things are inherently sequential
  - How we process `arr[i]` may depend entirely on the result of processing `arr[i-1]`

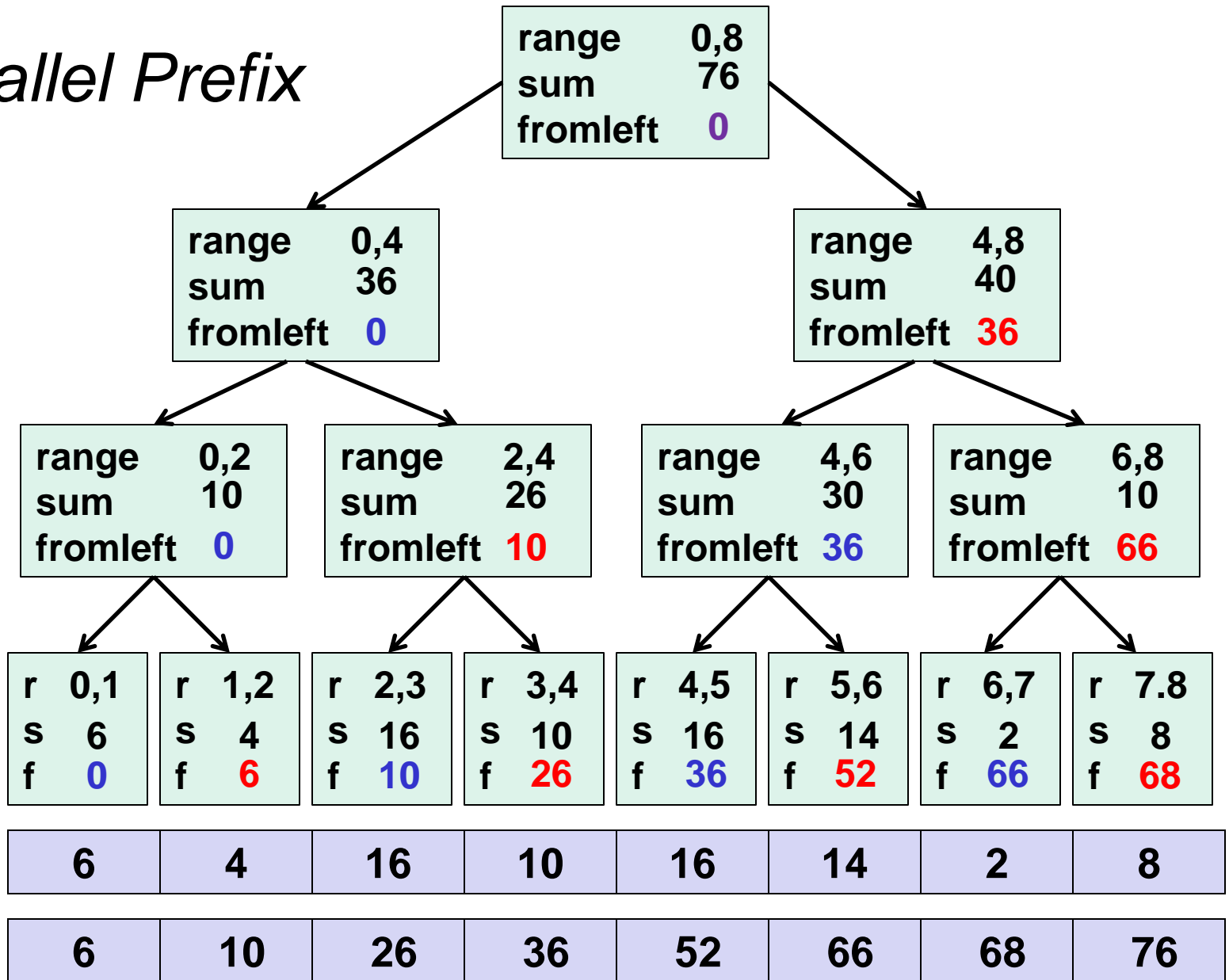
# Maps and Data Parallelism

- A **map** operates on each element of a collection independently to create a new collection of the same size
  - No combining results
  - For arrays, this is so trivial some hardware has direct support
- Canonical example: Vector addition

```
int[] vector add(int[] arr1, int[] arr2) {
    assert (arr1.length == arr2.length);
    result = new int[arr1.length];
    FORALL(i=0; i < arr1.length; i++) {
        result[i] = arr1[i] + arr2[i];
    }
    return result;
}
```



# Parallel Prefix



# *Pack*

[Non-standard terminology, `filter` does not emphasize stability]

Given an array `input`,  
produce an array `output`  
containing only elements such that `f(elt)` is `true`

Example: `input [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]`  
`f: is elt > 10`  
`output [17, 11, 13, 19, 24]`

Parallelizable

- Finding elements for the output is easy
- But getting them in the right place seems hard

# *Pack as Map, Parallel Prefix, Map*

1. Parallel map to compute a **bit-vector** for true elements

`input` [17, 4, 6, 8, 11, 5, 13, 19, 0, 24]

`bits` [1, 0, 0, 0, 1, 0, 1, 1, 0, 1]

2. Parallel-prefix sum on the bit-vector

`bitsum` [1, 1, 1, 1, 2, 2, 3, 4, 4, 5]

3. Parallel map to produce the output

`output` [17, 11, 13, 19, 24]

```
output = new array of size bitsum[n-1]
FORALL(i=0; i < input.length; i++){
    if(bits[i]==1)
        output[bitsum[i]-1] = input[i];
}
```

# Quicksort Review

Recall quicksort was sequential, in-place, expected time  $O(n \log n)$

## Best / expected case work

- |  |           |
|--|-----------|
| 1. Pick a pivot element                | $O(1)$    |
| 2. Partition all the data into:        | $O(n)$    |
| A. The elements less than the pivot    |           |
| B. The pivot                           |           |
| C. The elements greater than the pivot |           |
| 3. Recursively sort A and C            | $2T(n/2)$ |

How should we parallelize this?

# Quicksort

	Best / expected case work
1. Pick a pivot element	$O(1)$
2. Partition all the data into:	$O(n)$
A. The elements less than the pivot	
B. The pivot	
C. The elements greater than the pivot	
3. Recursively sort A and C	$2T(n/2)$

Easy: Do the two recursive calls in parallel

- Work: unchanged  $O(n \log n)$
- Span:  $T(n) = O(n) + T(n/2) = O(n) + O(n/2) + T(n/4) = O(n)$
- So parallelism is  $O(\log n)$  (i.e., work / span)

# *Doing Better*

- $O(\log n)$  speed-up with infinite number of processors is okay, but a bit underwhelming
  - Sort  $10^9$  elements 30 times faster
- Google searches strongly suggest quicksort cannot do better because the partition cannot be parallelized
  - The Internet has been known to be wrong
  - But we need auxiliary storage (will no longer in place)
  - In practice, constant factors may make it not worth it, but remember Amdahl's Law and the long-term situation
- Already have everything we need to parallelize the partition

# *Parallel Partition with Auxiliary Storage*

**Partition all the data into:**

**A. The elements less than the pivot**

**B. The pivot**

**C. The elements greater than the pivot**

- This is just two packs
  - We know a pack is  $O(n)$  work,  $O(\log n)$  span
  - Pack elements less than pivot into left side of **aux** array
  - Pack elements greater than pivot into right side of **aux** array
  - Put pivot between them and recursively sort
  - With a little more cleverness, can do both packs at once
    - But no effect on asymptotic complexity

# *Analysis*

With  $O(\log n)$  span for partition, the total span for quicksort is

$$\begin{aligned}T(n) &= O(\log n) + T(n/2) \\ &= O(\log n) + O(\log n/2) + T(n/4) \\ &= O(\log n) + O(\log n/2) + O(\log n/4) + T(n/8) \\ &\dots \\ &= O(\log^2 n)\end{aligned}$$

So parallelism (work / span) is  $O(n / \log n)$



# Example

- Step 1: pick pivot as median of three

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

- Steps 2a and 2c (combinable):  
pack less than and pack greater than into a second array
  - Fancy parallel prefix to pull this off not shown

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

- Step 3: Two recursive sorts in parallel
  - Can sort back into original array (swapping like in mergesort)

# Mergesort

Recall mergesort: sequential, not-in-place, worst-case  $O(n \log n)$

- |   |                             |
|---|-----------------------------|
| <b>1. Sort left half and right half</b> | <b><math>2T(n/2)</math></b> |
| <b>2. Merge results</b>                 | <b><math>O(n)</math></b>    |

Just like quicksort, doing the two recursive sorts in parallel changes the recurrence for the span to  $T(n) = O(n) + 1T(n/2) = O(n)$

- Again, parallelism is  $O(\log n)$
- To do better, need to parallelize the merge
  - The trick this time will not use parallel prefix

# Parallelizing the Merge

Need to merge two *sorted* subarrays (may not have the same size)

0	1	4	8	9
---	---	---	---	---

2	3	5	6	7
---	---	---	---	---

Idea: Suppose the larger subarray has  $n$  elements. In parallel:

- merge the first  $n/2$  elements of the larger half with the “appropriate” elements of the smaller half
- merge the second  $n/2$  elements of the larger half with the remainder of the smaller half

# *Parallelizing the Merge*

0	4	6	8	9
---	---	---	---	---

1	2	3	5	7
---	---	---	---	---

# *Parallelizing the Merge*



1. Get median of bigger half:

# *Parallelizing the Merge*

0	4	6	8	9
---	---	---	---	---

1	2	3	5	7
---	---	---	---	---

1. Get median of bigger half:  $O(1)$  to compute middle index

# *Parallelizing the Merge*



1. Get median of bigger half:  $O(1)$  to compute middle index
2. Find how to split the smaller half at the same value:

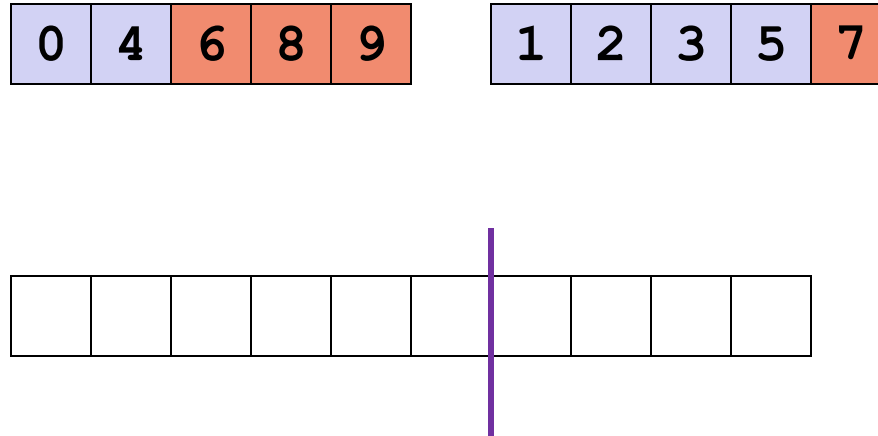
# *Parallelizing the Merge*



1. Get median of bigger half:  $O(1)$  to compute middle index
2. Find how to split the smaller half at the same value:  
 $O(\log n)$  to do binary search on the sorted small half

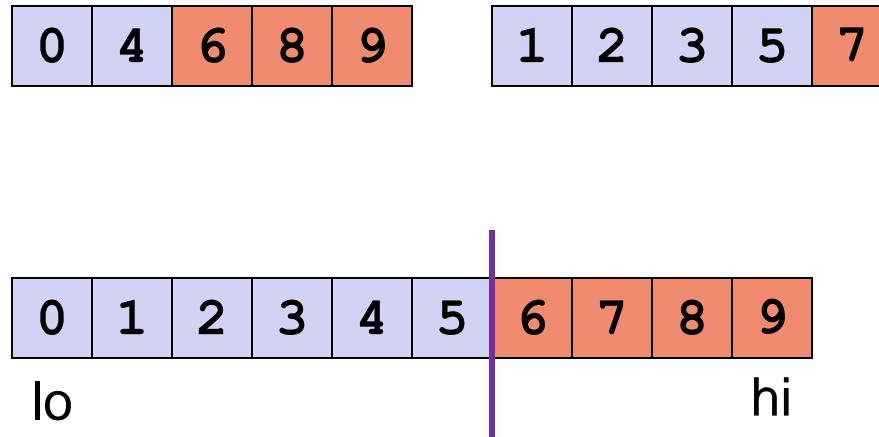


# Parallelizing the Merge



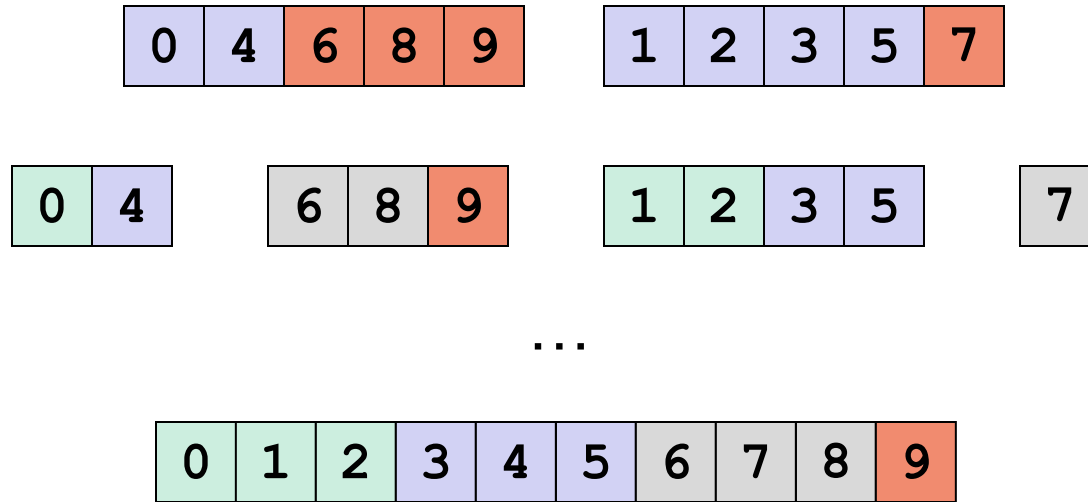
1. Get median of bigger half:  $O(1)$  to compute middle index
2. Find how to split the smaller half at the same value:  
 $O(\log n)$  to do binary search on the sorted small half
3. Size of two sub-merges conceptually splits output array:  $O(1)$

# Parallelizing the Merge



1. Get median of bigger half:  $O(1)$  to compute middle index
2. Find how to split the smaller half at the same value:  
 $O(\log n)$  to do binary search on the sorted small half
3. Size of two sub-merges conceptually splits output array:  $O(1)$
4. Do two submerges in parallel

# *The Recursion*



When we do each merge in parallel,  
we split the bigger array in half,  
and use binary search to split the smaller array,  
in base case we do the copy

# Analysis

- Sequential recurrence for mergesort:

$$T(n) = 2T(n/2) + O(n) \text{ which is } O(n \log n)$$

- Doing the two recursive calls in parallel but a sequential merge:  
work: same as sequential    span:  $T(n) = 1T(n/2) + O(n)$  which is  $O(n)$
- Parallel merge makes work and span harder to compute
  - Each merge step does an extra  $O(\log n)$   
binary search to find how to split the smaller subarray
  - To merge  $n$  elements total,  
do two smaller merges of possibly different sizes
  - But worst-case split is  $(1/4)n$  and  $(3/4)n$ 
    - When subarrays same size and “smaller” splits “all” / “none”

# Analysis

For just a parallel merge of  $n$  elements:

- Span is  $T(n) = T(3n/4) + O(\log n)$ , which is  $O(\log^2 n)$
- Work is  $T(n) = T(3n/4) + T(n/4) + O(\log n)$  which is  $O(n)$
- Neither bound is immediately obvious, but “trust us”

So for mergesort with parallel merge overall:

- Span is  $T(n) = 1T(n/2) + O(\log^2 n)$ , which is  $O(\log^3 n)$
- Work is  $T(n) = 2T(n/2) + O(n)$ , which is  $O(n \log n)$

So parallelism (work / span) is  $O(n / \log^2 n)$

- Not quite as good as quicksort’s  $O(n / \log n)$ 
  - But worst-case guarantee
- And as always this is just the asymptotic result

# *Toward Sharing Resources*

Have been studying [parallel algorithms](#) using fork-join

- Lower span via parallel tasks

Algorithms all had a very simple *structure* to avoid race conditions

- Each thread had memory “only it accessed”
  - Example: array sub-range
- Or used **fork** and **join** as contract for who “had” memory
  - On **fork**, “loan” some memory to “forkee” and do not access that memory again until after **join** on the “forkee”

Strategy will not work well when:

- Memory accessed by threads is overlapping or unpredictable
- Threads are doing independent tasks needing access to same resources (as opposed to implementing the same algorithm)

# Concurrent Programming

## Concurrency:

Correctly and efficiently managing access to shared resources from multiple possibly-simultaneous clients

Requires *coordination*, particularly **synchronization** to avoid **incorrect simultaneous access**: make somebody *block*

- `join` is not what we want
- Want to block until another thread is “done with what we need”, not the more extreme “until completely done executing”

Even correct concurrent applications are usually highly **non-deterministic**: how threads are scheduled affects what each thread sees in its different operations

- non-repeatability complicates testing and debugging

# *Examples*

Multiple threads:

1. Processing different bank-account operations
  - What if 2 threads change the same account at the same time?
2. Using a shared cache of recent files (e.g., hashtable)
  - What if 2 threads insert the same file at the same time?
3. Creating a pipeline with a queue for handing work to next thread in sequence (i.e., a virtual assembly line)?
  - What if enqueueer and dequeueer adjust a circular array queue at the same time?



# *Why Threads?*

Unlike parallelism, not about implementing algorithms faster

But threads still useful for:

- *Code structure for responsiveness*
  - Respond to GUI events in one thread while another thread is performing an expensive computation
- *Processor utilization (mask I/O latency)*
  - If 1 thread “goes to disk,” have something else to do
- *Failure isolation*
  - Convenient structure if want to *interleave* multiple tasks and do not want an exception in one to stop the other



# CSE332: Data Abstractions

## Lecture 19: Mutual Exclusion and Locking

James Fogarty

Winter 2012

# Banking Example

This code is correct in a single-threaded world

```
class BankAccount {
    private int balance = 0;
    int getBalance()      { return balance; }
    void setBalance(int x) { balance = x; }
    void withdraw(int amount) {
        int b = getBalance();
        if(amount > b)
            throw new InsufficientFundsException();
        setBalance(b - amount);
    }
    ... // other operations like deposit, etc.
}
```

# *Interleaving*

Suppose:

- Thread **T1** calls `x.withdraw(100)`
- Thread **T2** calls `y.withdraw(100)`

If second call starts before first finishes, we say the calls **interleave**

- Could happen even with one processor,  
as a thread can be **pre-empted** for time-slicing  
(e.g., T1 runs for 50 ms, T2 runs for 50ms, T1 resumes)

If **x** and **y** refer to different accounts, no problem

- “You cook in your kitchen while I cook in mine”

But if **x** and **y** alias, possible trouble...

# Bad Interleaving

Interleaved `withdraw(100)` calls on the same account

- Assume initial `balance == 150`

Thread 1

```
int b = getBalance();  
  
if (amount > b)  
    throw new ...;  
setBalance(b - amount);
```

Thread 2

```
int b = getBalance();  
if (amount > b)  
    throw new ...;  
setBalance(b - amount);
```

Time



# *Incorrect Attempt to “Fix”*

Interleaved `withdraw(100)` calls on the same account

- Assume initial `balance == 150`

Thread 1

```
int b = getBalance();  
  
if (amount > getBalance())  
    throw new ...;  
setBalance(b - amount);
```

Thread 2

```
int b = getBalance();  
if (amount > getBalance())  
    throw new ...;  
setBalance(b - amount);
```

Time



This interleaving would throw an exception

# *Incorrect Attempt to “Fix”*

Interleaved `withdraw(100)` calls on the same account

- Assume initial `balance == 150`

Thread 1

```
int b = getBalance();  
if (amount > getBalance())  
    throw new ...;  
  
setBalance(b - amount);
```

Thread 2

```
int b = getBalance();  
if (amount > getBalance())  
    throw new ...;  
setBalance(b - amount);
```

Time



But this interleaving loses the withdrawal

# *Incorrect Attempt to “Fix”*

Interleaved `withdraw(100)` calls on the same account

- Assume initial `balance == 150`

Thread 1

```
int b = getBalance();  
if(amount > getBalance())  
    throw new ...;
```

Thread 2

```
int b = getBalance();  
if(amount > getBalance())  
    throw new ...;  
setBalance(  
    getBalance() - amount  
);
```

Time



```
setBalance(  
    getBalance() - amount  
);
```

Does not “lose” money in this particular interleaving, but is still wrong



## *Incorrect Attempt to “Fix”*

It can be tempting, but is generally **wrong**, to attempt to “fix” a bad interleaving by rearranging or repeating operations

```
void withdraw(int amount) {  
    if (amount > getBalance())  
        throw new InsufficientFundsException();  
    // maybe balance changed  
    setBalance(getBalance() - amount);  
}
```

Only narrows the problem by one statement

- Imagine a withdrawal is interleaved after computing the value of the parameter `getBalance() - amount` but before invocation of the function `setBalance`

Your compiler might even remove the second call to `getBalance()`, because you have not told it you need to synchronize

# *Mutual Exclusion*

The sane fix is to allow only one thread withdrawing from **A** at a time

- Also exclude other simultaneous operations on **A** that could potentially result in bad interleavings (e.g., deposit)

**Mutual exclusion**: One thread doing something with a resource means that any other thread must wait until the resource is available

- Define **critical sections**; areas of code that are mutually exclusive

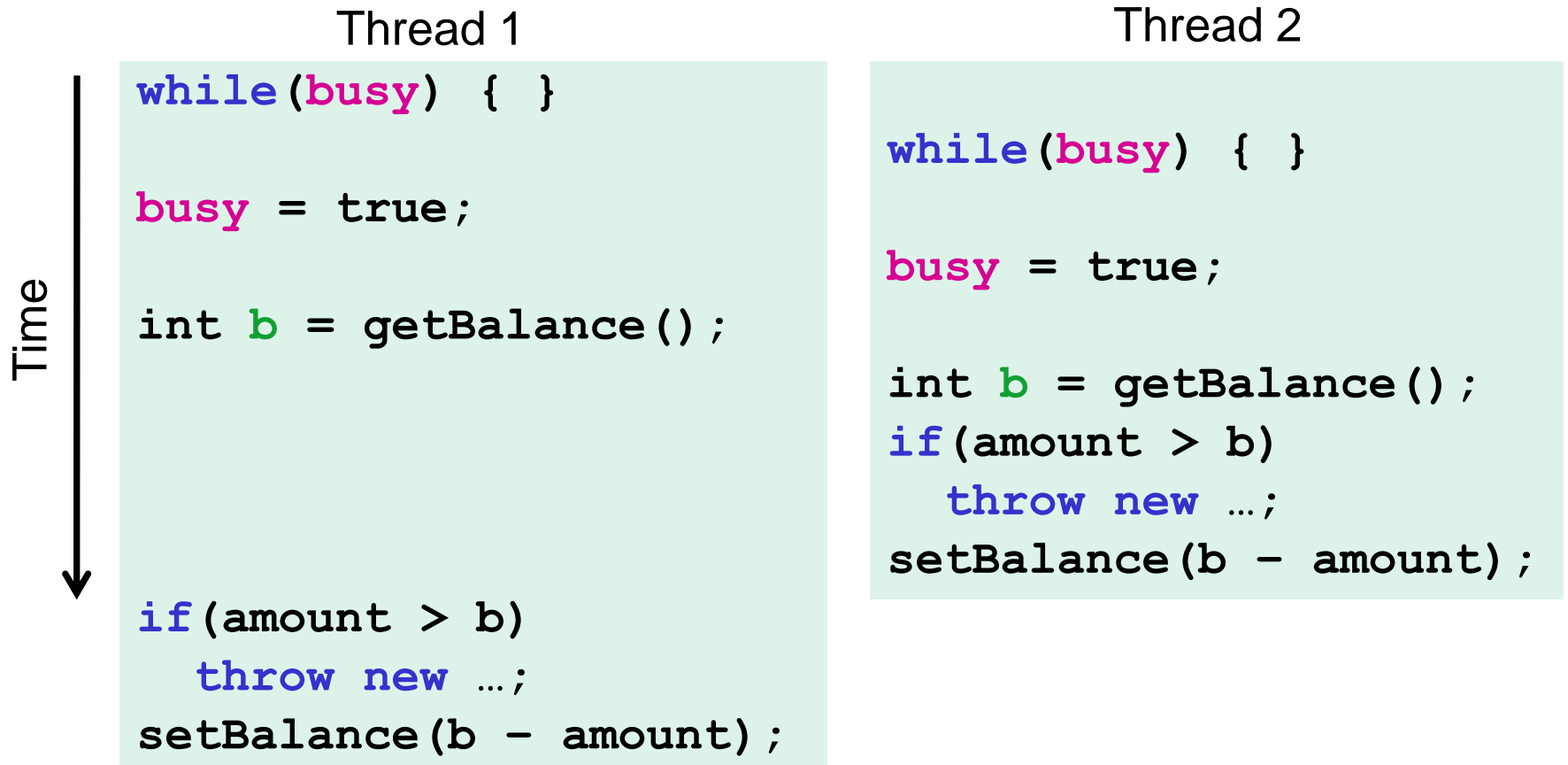
Programmer must implement critical sections

- “The compiler” has no idea what interleavings should or should not be allowed in your program
- But you will need language primitives to do this

## *Incorrect Attempt to “Do it Ourselves”*

```
class BankAccount {
    private int balance = 0;
    private boolean busy = false;
    void withdraw(int amount) {
        while(busy) { /* “spin-wait” */ }
        busy = true;
        int b = getBalance();
        if(amount > b)
            throw new InsufficientFundsException();
        setBalance(b - amount);
        busy = false;
    }
    // deposit would spin on same boolean
}
```

# *This Just Moves the Problem*



# *Need Help from the Language*

- There are many ways out of this conundrum
- One basic solution: **Locks**
  - Still on a conceptual, ‘Lock’ is not a Java class
- We will define **Lock** as an ADT with operations:
  - **new**: make a new lock
  - **acquire**: If lock is “*not held*”, makes it “*held*”
    - Blocks if this lock is already currently “*held*”
    - Checking & Setting happen **atomically**, cannot be interrupted
      - Details of that require hardware and system support
  - **release**: makes this lock “*not held*”
    - if  $\geq 1$  threads are blocked on it, exactly 1 will acquire it

## *Still Incorrect Pseudocode*

```
class BankAccount {
    private int balance = 0;
    private Lock lk = new Lock();
    ...
    void withdraw(int amount) {
        lk.acquire(); /* may block */
        int b = getBalance();
        if(amount > b)
            throw new InsufficientFundsException();
        setBalance(b - amount);
        lk.release();
    }
    // deposit would also acquire/release lk
}
```

# Some Mistakes

- A lock is a very primitive mechanism
  - Still must be used correctly to implement critical sections
- **Incorrect:** Forget to release a lock, thus blocks other threads forever
  - Previous slide is wrong because of the exception possibility

```
if (amount > b) {  
    lk.release(); // hard to remember!  
    throw new WithdrawTooLargeException();  
}
```

- **Incorrect** : Use different locks for **withdraw** and **deposit**
  - Mutual exclusion works only when using same lock
  - Balance is the shared resource that is being protected
- **Poor performance:** Use same lock for every bank account
  - No simultaneous withdrawals from *different* accounts

# Other Operations

- If **withdraw** and **deposit** use the same lock, then simultaneous calls to these methods are properly synchronized
- But what about **getBalance** and **setBalance**
  - Assume they are **public**, which may be reasonable
- If they **do not acquire the same lock**, then a race between **setBalance** and **withdraw** could produce a wrong result
- If they **do acquire the same lock**, then **withdraw** would block forever because it tries to acquire a lock it already has

```
...  
    lk.acquire();  
    int b = getBalance();  
...
```



# One Bad Option

```
int setBalanceUnsafe(int x) {
    balance = x;
}
int setBalanceSafe(int x) {
    lk.acquire();
    balance = x;
    lk.release();
}
void withdraw(int amount) {
    lk.acquire();
    ...
    setBalanceUnsafe(b - amount);
    lk.release();
}
```

Two versions of setBalance

- Safe and unsafe versions
- Use one or the other, depending on whether you already have the lock

Technically could work

- Hard to always remember
- And definitely poor style

Better to modify meaning of the **Lock ADT** to support **re-entrant locks**

# *Re-Entrant Locking*

A **re-entrant lock** is also known as a **recursive lock**

- “Remembers” the thread that currently holds it
- Stores a *count* of “how many” times it is held
- When lock goes from ***not-held*** to ***held***, the count is set to 0
- If the current holder calls **acquire**:
  - it does not block
  - it increments the count
- On **release**:
  - if the count is  $> 0$ , the count is decremented
  - if the count is 0, the lock becomes ***not-held***

**withdraw** can acquire the lock, and then call **setBalance**

# Java's Re-Entrant Lock

`java.util.concurrent.locks.ReentrantLock`

- Has methods `lock()` and `unlock()`

Important to guarantee that lock is always released

```
myLock.lock();  
try {  
    // method body  
} finally {  
    myLock.unlock();  
}
```

Regardless what happens in 'try', finally code will execute

# A Java Convenience: **Synchronized**

Java has built-in support for re-entrant locks

- You can use the **synchronized** statement as an alternative to declaring a **ReentrantLock**

```
synchronized (expression) {  
    statements  
}
```

1. Evaluates *expression* to an **object**
  - Every **object** “is a lock” in Java (but not primitive types)
2. Acquires the lock, blocking if necessary
  - “If you get past the {, you have the lock”
3. Releases the lock “at the matching }”
  - Even if control leaves due to **throw**, **return**, or whatever
  - So it is impossible to forget to release the lock

# *Java Version #1: Correct but not “Java Style”*

```
class BankAccount {
    private int balance = 0;
    private Object lk = new Object();
    int getBalance()
        { synchronized (lk) { return balance; } }
    void setBalance(int x)
        { synchronized (lk) { balance = x; } }
    void withdraw(int amount) {
        synchronized (lk) {
            int b = getBalance();
            if(amount > b)
                throw ...
            setBalance(b - amount);
        }
    }
    // deposit would also use synchronized(lk)
}
```

# *Improving the Java*

- As written, the lock is **private**
  - Might seem like a good idea
  - But also prevents code in other classes from writing operations that synchronize with the account operations
- Example motivations with our bank record?
- It is more common to synchronize on **this**
  - It is also convenient; no need to declare an extra object

## Java Version #2: Still Missing Sugar

```
class BankAccount {
    private int balance = 0;
    int getBalance()
        { synchronized (this){ return balance; } }
    void setBalance(int x)
        { synchronized (this){ balance = x; } }
    void withdraw(int amount) {
        synchronized (this) {
            int b = getBalance();
            if(amount > b)
                throw ...
            setBalance(b - amount);
        }
    }
    // deposit would also use synchronized(this)
}
```

# *Syntactic Sugar*

Java provides a concise and standard way to say the same thing:

Applying the **synchronized** keyword to a method declaration means the entire method body is surrounded by

```
synchronized (this) {  
    ...  
}
```

Next version means exactly the same thing,  
but is more concise and more the “style of Java”



## *Java Version #3: Final Version*

```
class BankAccount {
    private int balance = 0;
    synchronized int getBalance()
        { return balance; }
    synchronized void setBalance(int x)
        { balance = x; }
    synchronized void withdraw(int amount) {
        int b = getBalance();
        if(amount > b)
            throw ...
        setBalance(b - amount);
    }
    // deposit would also use synchronized
}
```

# Races

A **race condition** occurs when the computation result depends on scheduling (how threads are interleaved on one or more processors)

- If T1 and T2 are scheduled in a particular way, then things go wrong
- As programmers, we cannot control scheduling of threads; we need to write programs that are correct independent of scheduling

Race conditions are bugs that exist only due to concurrency

- No interleaved scheduling with 1 thread

Typically, the problem is some *intermediate state* that “messes up” a concurrent thread that “sees” that state

We will distinguish between **data races** and **bad interleavings**, both of which are types of race condition bugs

# *Data Races*

- A **data race** is a type of **race condition** that can happen in 2 ways:
  - Two threads can **potentially** write a variable at the same time
  - One thread can **potentially** write a variable while another reads it
- Simultaneous reads are fine; not a data race, and no bad results
- ‘Potentially’ is important; we say the code itself has a data race
  - This is independent of any particular actual execution
- Data races are bad, but are not the only form of race condition
  - We can have a race, and bad behavior, without any data race

# Stack Example

```
class Stack<E> {
    private E[] array = (E[])new Object[SIZE];
    int index = -1;
    synchronized boolean isEmpty() {
        return index==-1;
    }
    synchronized void push(E val) {
        array[++index] = val;
    }
    synchronized E pop() {
        if(isEmpty())
            throw new StackEmptyException();
        return array[index--];
    }
}
```

# A Race Condition: But Not a Data Race

```
class Stack<E> {  
    ...  
    synchronized boolean isEmpty() { ... }  
    synchronized void push(E val) { ... }  
    synchronized E pop(E val) {  
        if(isEmpty())  
            throw new StackEmptyException();  
        ...  
    }  
}  
  
E peek() {  
    E ans = pop();  
    push(ans);  
    return ans;  
}
```

- In a sequential world, this code is of questionable style, but *correct*
- The “algorithm” is the only way to write a **peek** helper method if all you have is this interface

# Examining `peek` in a Concurrent Context

- `peek` has no *overall* effect on the shared data
  - It is a “reader” not a “writer”
  - State should be the same after it executes as before
- This implementation creates an inconsistent *intermediate state*
  - Calls to `push` and `pop` are synchronized, so there are no ***data races*** on the underlying array
  - But there is still a ***race condition***
- This intermediate state should not be exposed
  - Leads to several *bad interleavings*

```
E peek () {  
    E ans = pop ();  
    push (ans) ;  
    return ans ;  
}
```

# Example 1: peek and isEmpty

- **Property we want:**  
If there has been a **push** (and no **pop**),  
then **isEmpty** should return **false**
- With **peek** as written, property can be violated – how?

Thread 1 (**peek**)

```
E ans = pop();  
  
push(ans);  
  
return ans;
```

Thread 2

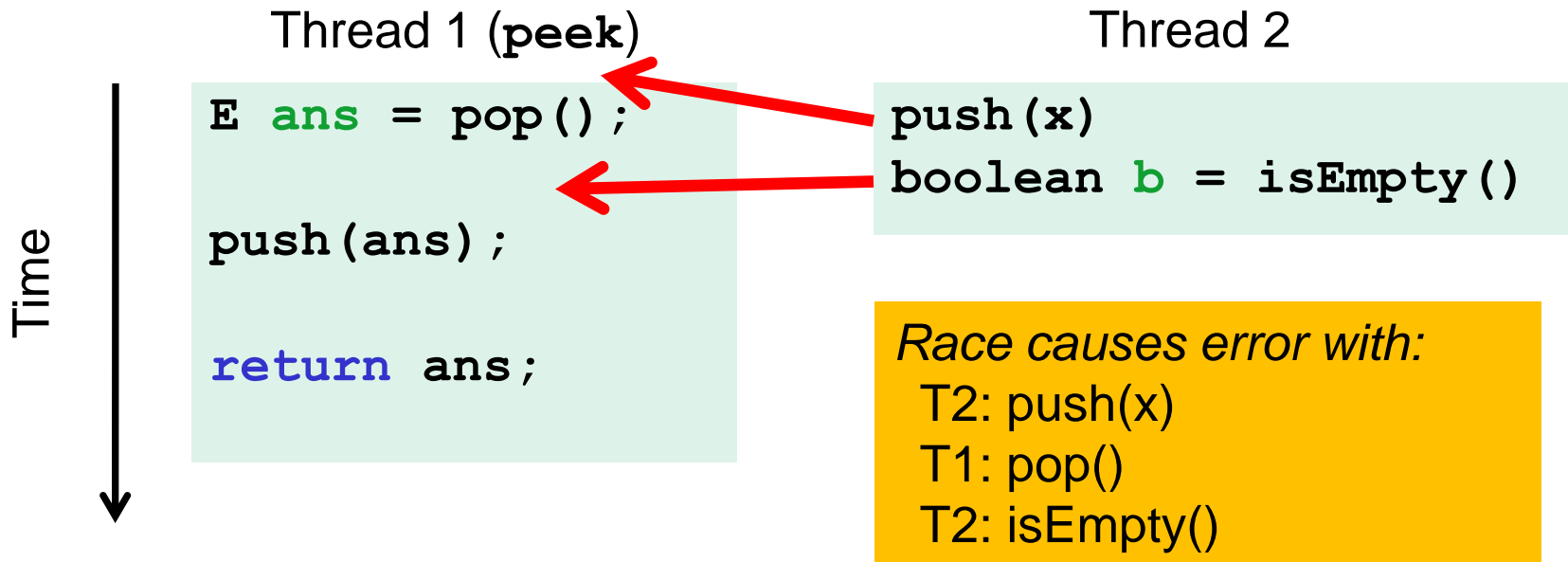
```
push(x)  
boolean b = isEmpty()
```

Time



# Example 1: peek and isEmpty

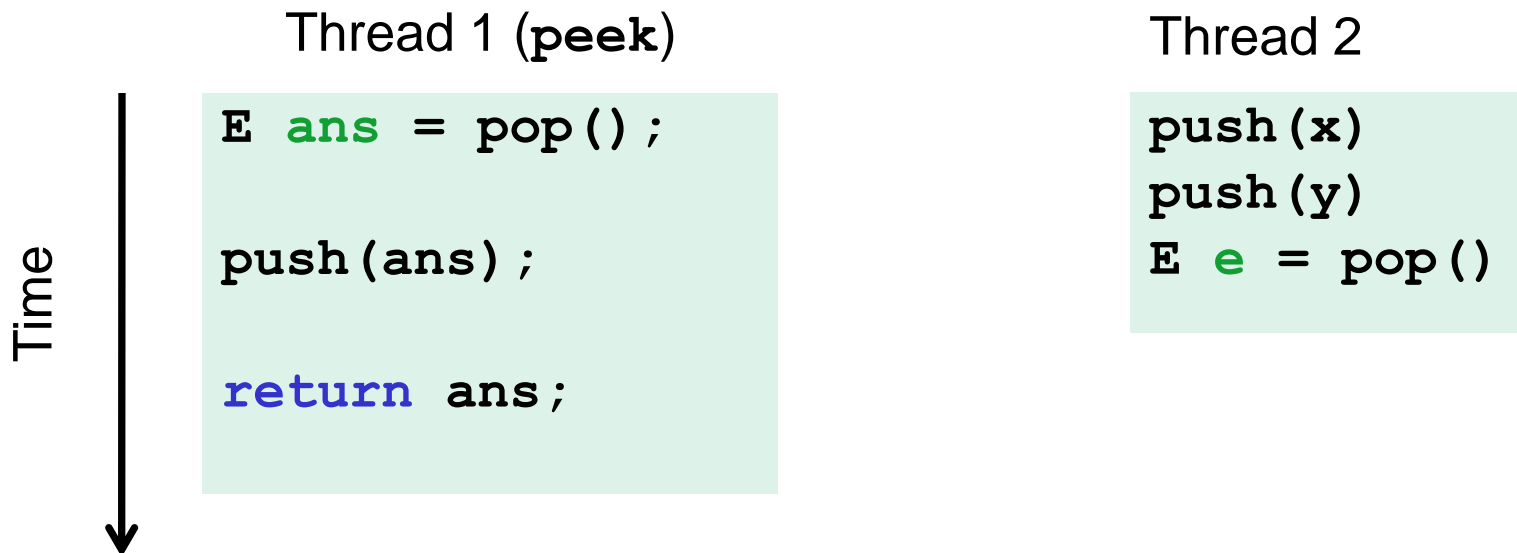
- **Property we want:**  
If there has been a **push** (and no **pop**), then **isEmpty** should return **false**
- With **peek** as written, property can be violated – how?





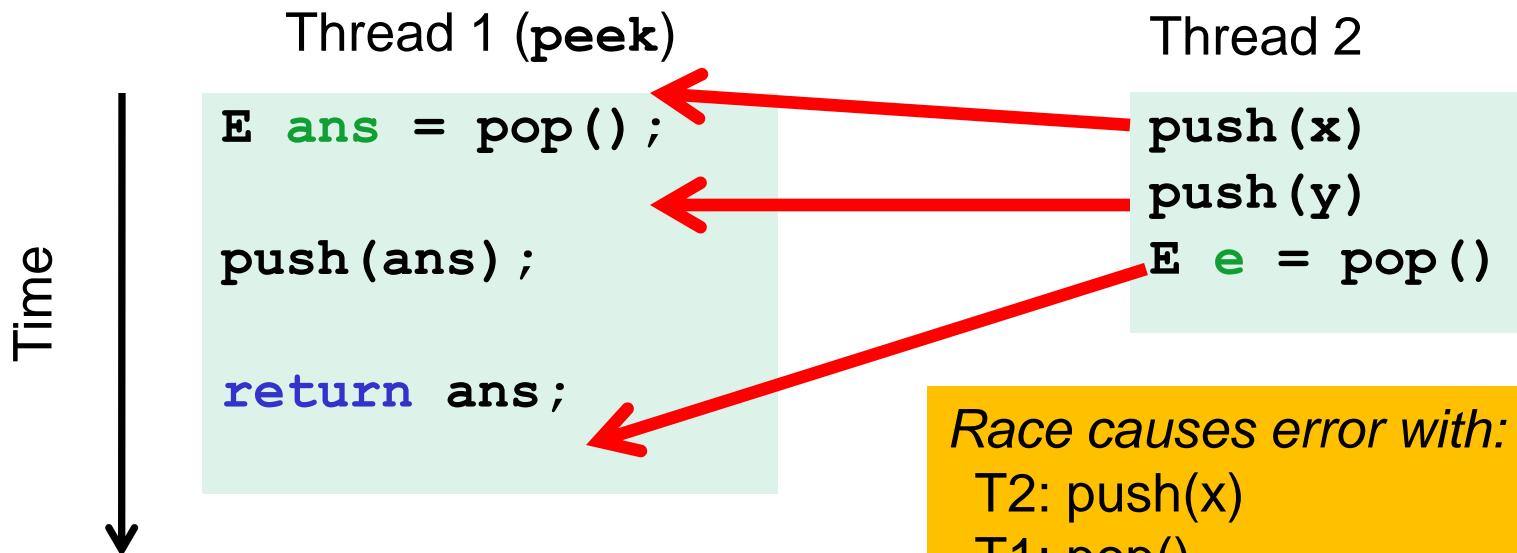
## Example 2: peek and push

- **Property we want:**  
Values are returned from `pop` in LIFO order
- With `peek` as written, property can be violated – how?



## Example 2: peek and push

- **Property we want:**  
Values are returned from `pop` in LIFO order
- With `peek` as written, property can be violated – how?

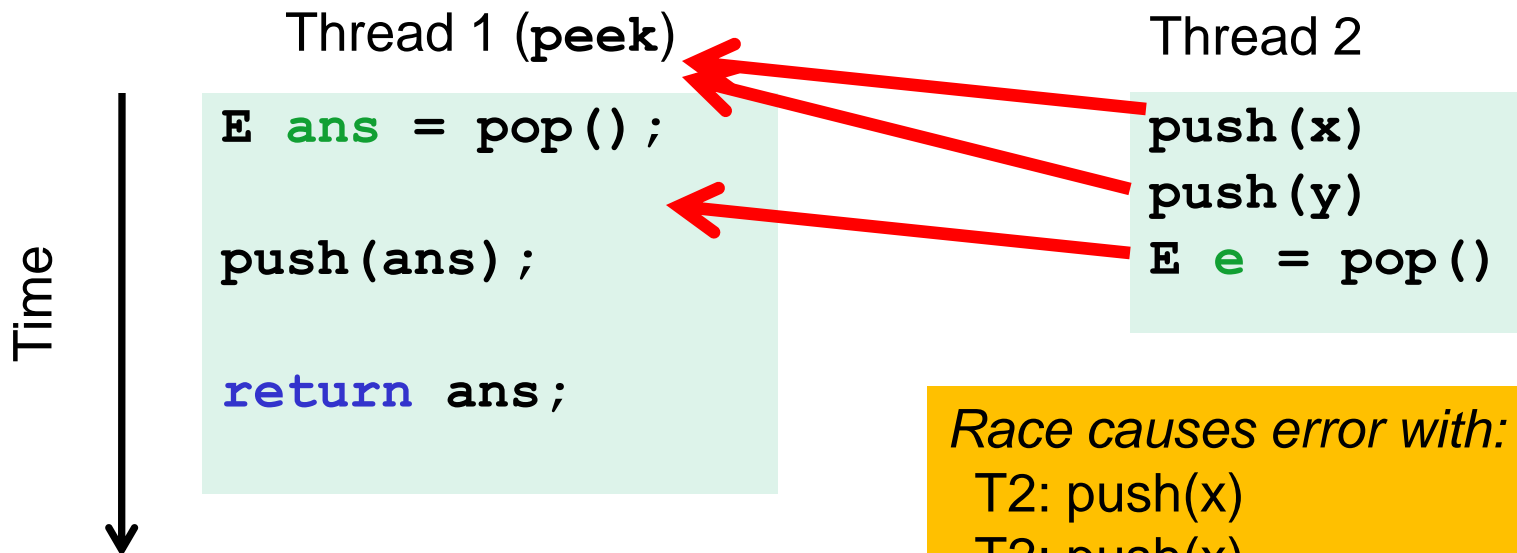


*Race causes error with:*

T2: `push(x)`  
T1: `pop()`  
T2: `push(x)`  
T1: `push(x)`

## Example 2: peek and push

- **Property we want:**  
Values are returned from `pop` in LIFO order
- With `peek` as written, property can be violated – how?



*Race causes error with:*

```
T2: push(x)  
T2: push(x)  
T1: pop()  
T2: pop()
```

## Example 3: peek and peek

- **Property we want:**  
    **peek** does not throw an exception unless stack is empty
- With **peek** as written, property can be violated – how?

Thread 1 (**peek**)

```
E ans = pop ();  
push (ans) ;  
return ans ;
```

Thread 2 (**peek**)

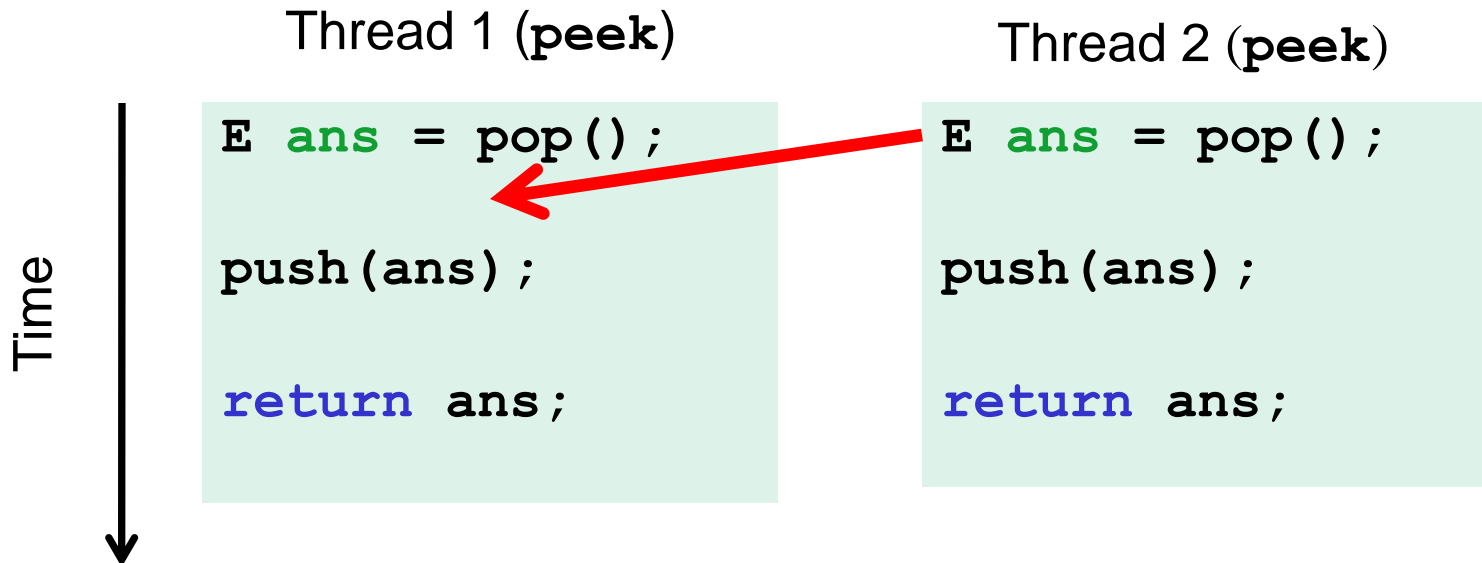
```
E ans = pop ();  
push (ans) ;  
return ans ;
```

Time



## Example 3: peek and peek

- **Property we want:**  
peek does not throw an exception unless stack is empty
- With **peek** as written, property can be violated – how?



# The Fix

- In short, **peek** needs synchronization to disallow interleavings
  - The key is to make a *larger critical section*
    - This protects the intermediate state of `peek`
  - Use re-entrant locks; will allow calls to **push** and **pop**
  - Can be done in stack (on left) or an external class (on right)

```
class Stack<E> {  
    ...  
    synchronized E peek() {  
        E ans = pop();  
        push(ans);  
        return ans;  
    }  
}
```

```
class C {  
    <E> E myPeek(Stack<E> s) {  
        synchronized (s) {  
            E ans = s.pop();  
            s.push(ans);  
            return ans;  
        }  
    }  
}
```

## *An Incorrect “Fix”*

- So far we have focused on problems created when **peek** performs **writes** that lead to an incorrect intermediate state
- A tempting but incorrect perspective
  - If an implementation of **peek** does not write anything, then maybe we can skip the synchronization?
- Does **not** work due to *data races* with **push** and **pop**
  - Same issue applies with other readers, such as **isEmpty**

## *Another Incorrect Example*

```
class Stack<E> {
    private E[] array = (E[])new Object[SIZE];
    int index = -1;
    boolean isEmpty() { // unsynchronized: wrong?!
        return index==-1;
    }
    synchronized void push(E val) {
        array[++index] = val;
    }
    synchronized E pop() {
        return array[index--];
    }
    E peek() { // unsynchronized: wrong!
        return array[index];
    }
}
```



# Why Wrong?

- It *looks like* `isEmpty` and `peek` can “get away with this” because `push` and `pop` adjust the state “in one tiny step”
- But this code is still *wrong* and depends on language-implementation details you cannot assume
  - Even “tiny steps” may require multiple steps in implementation:  
`array[++index] = val` probably takes at least two steps
  - Code has a [data race](#), allowing very strange behavior
- Do not introduce a data race, even if every interleaving you can think of is correct

# *Getting it Right*

Avoiding race conditions on shared resources is difficult

- Decades of bugs have led to some *conventional wisdom*, general techniques that are known to work

Rest of lecture distills key ideas and trade-offs

- More available in the suggested additional readings
- But none of this is specific to Java or a particular book
  - May be hard to appreciate in beginning
  - Come back to these guidelines over the years
  - Do not try to be fancy



# CSE332: Data Abstractions

## Lecture 20: Mutual Exclusion and Locking

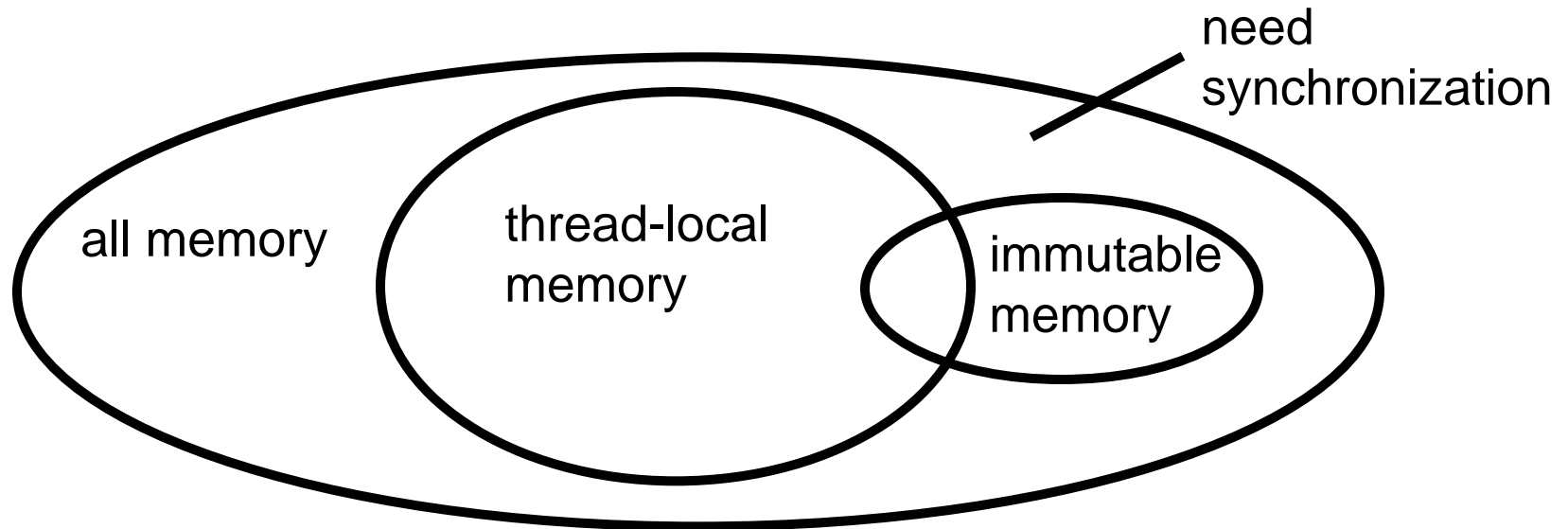
James Fogarty

Winter 2012

# *Pick From These 3 Choices for Memory:*

For every **memory location** in your program (e.g., object field), you must obey at least one of the following:

1. **Thread-local:** Do not use the location in  $> 1$  thread
2. **Immutable:** Do not write to the memory location
3. **Synchronized:** Use synchronization to control access



# *Thread-Local*

Whenever possible, do not share resources

- Easier for each thread have its own thread-local *copy* of a resource instead of one with shared updates
- Correct only if threads do not communicate through resource
  - In other words, multiple copies are a correct approach
  - Example: **Random** objects
- Note:
  - Because each call-stack is thread-local,  
never need to synchronize on local variables

*In typical concurrent programs, the vast majority of objects should be thread-local: shared-memory usage should be minimized*

# *Immutable*

Whenever possible, do not update objects

- Make new objects instead

One of the key tenets of *functional programming* (see CSE 341)

- Generally helpful to avoid *side-effects*
- Much more helpful in a concurrent setting

If a location is only read, never written, no synchronization needed

- Simultaneous reads are *not* races and *not* a problem

*In practice, programmers usually over-use mutation – minimize it*

# *Everything Else: Keep it Synchronized*

After minimizing the amount of memory that is both (1) thread-shared and (2) mutable, we need guidelines for how to use locks to keep that data consistent

## Guideline #0: No data races

- Never allow two threads to read/write or write/write the same location at the same time

*Necessary:*

In Java or C, a program with a data race is almost always wrong

*But Not Sufficient:*

Our `peek` example had no data races

# *Consistent Locking*

## Guideline #1: Consistent Locking

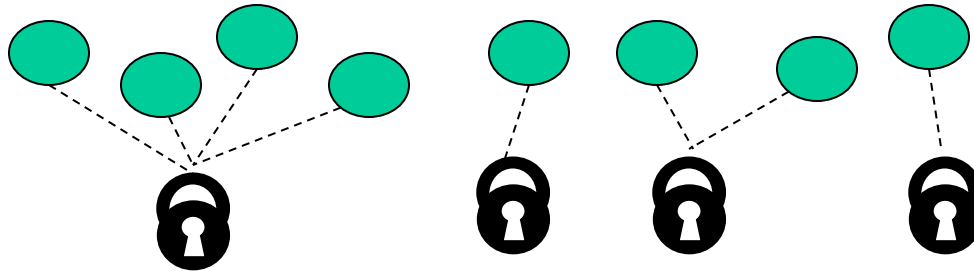
For each location that requires synchronization, have a lock that is always held when reading or writing the location

- We say the lock **guards** the location
- The same lock can guard multiple locations (and often should)
- Clearly document the guard for each location
- In Java, the guard is often the object containing the location
  - **this** inside object methods
  - But also common to guard a larger structure with one lock to ensure mutual exclusion on the structure



# Consistent Locking

- The mapping from locations to guarding locks is *conceptual*, and must be enforced by you as the programmer
- It partitions the shared-&-mutable locations into “which lock”



Consistent locking is:

*Not Sufficient:*

It prevents all data races, but still allows bad interleavings

- Our **peek** example used consistent locking, but had exposed intermediate states and bad interleavings

*Not Necessary:*

Can dynamically change the locking protocol

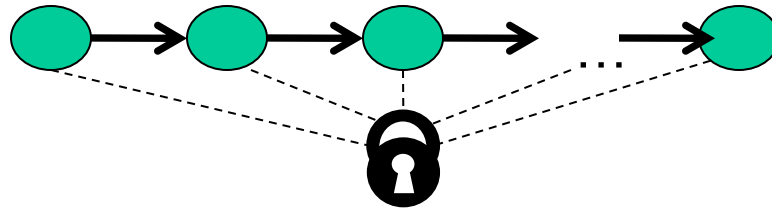
# *Beyond Consistent Locking*

- Consistent locking is an excellent guideline
  - A “default assumption” about program design
  - You will save yourself many a headache using this guideline
- But it is not required for correctness:  
Different *program phases* can use different locking techniques
  - Provided all threads coordinate moving to the next phase
- Example from Project 3 Version 5:
  - A shared grid being updated, so use a lock for each entry
  - But after the grid is filled out, all threads except 1 terminate
    - So synchronization no longer necessary (i.e., thread local)
  - And later the grid is only read in response to queries
    - Makes synchronization doubly unnecessary (i.e., immutable)

# Lock Granularity

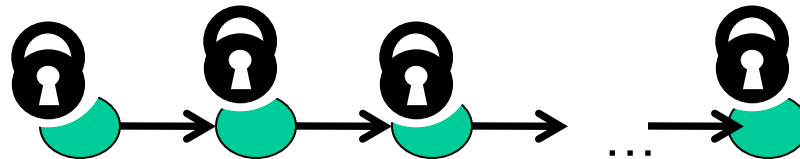
**Coarse-Grained:** Fewer locks (i.e., more objects per lock)

- Example: One lock for entire data structure (e.g., array)
- Example: One lock for all bank accounts



**Fine-Grained:** More locks (i.e., fewer objects per lock)

- Example: One lock per data element (e.g., array index)
- Example: One lock per bank account



“Coarse-grained vs. fine-grained” is really a continuum

# Trade-Offs

## Coarse-grained advantages

- Simpler to implement
- Faster/easier to implement operations that access multiple locations (because all guarded by the same lock)
- Much easier to implement modifications of data-structure shape

## Fine-grained advantages

- More simultaneous access (improves performance when coarse-grained would lead to unnecessary blocking)

## Guideline #2: Lock Granularity

Start with coarse-grained (simpler), move to fine-grained (performance) only if *contention* on coarse locks is an issue. Alas, often leads to bugs.

# *Example: Separate Chaining Hashtable*

- Coarse-grained: One lock for entire hashtable
- Fine-grained: One lock for each bucket

Which supports more concurrency for **insert** and **lookup**?

Fine-grained; allows simultaneous access to diff. buckets

Which makes implementing **resize** easier?

Coarse-grained; just grab one lock and proceed

– How would you do it?

Maintaining a **numElements** field will destroy the potential benefits of using separate locks for each bucket, why?

Updating it each insert w/o a coarse lock would be a data race

# *Critical-Section Granularity*

A second, orthogonal granularity issue is critical-section size

- How much work to do while holding lock(s)

If critical sections run for too long:

- Performance loss because other threads are blocked

If critical sections are too short:

- Bugs because you broke up something where other threads should not be able to see intermediate state

## Guideline #3: Granularity

Do not do expensive computations or I/O in critical sections, but also do not introduce race conditions

# *Example: Critical-Section Granularity*

Suppose we want to change the value for a key in a hashtable without removing it from the table

- Assume `lock` guards the whole table

*Papa Bear's  
critical section  
was too long*

*(table locked  
during  
expensive call)*

```
synchronized(lock) {  
    v1 = table.lookup(k);  
    v2 = expensive(v1);  
    table.remove(k);  
    table.insert(k, v2);  
}
```

# Example: Critical-Section Granularity

Suppose we want to change the value for a key in a hashtable without removing it from the table

- Assume `lock` guards the whole table

*Mama Bear's  
critical section  
was too short*

*(if another thread  
updated the entry,  
we will lose an  
update)*

```
synchronized(lock) {  
    v1 = table.lookup(k);  
}  
v2 = expensive(v1);  
synchronized(lock) {  
    table.remove(k);  
    table.insert(k, v2);  
}
```



# Example: Critical-Section Granularity

Suppose we want to change the value for a key in a hashtable without removing it from the table

- Assume `lock` guards the whole table

*Baby Bear's  
critical section  
was just right*

*(if another update  
occurred, try our  
update again)*

```
done = false;
while (!done) {
    synchronized(lock) {
        v1 = table.lookup(k);
    }
    v2 = expensive(v1);
    synchronized(lock) {
        if (table.lookup(k) == v1) {
            done = true;
            table.remove(k);
            table.insert(k, v2);
        }
    }
}
```

# *Atomicity*

An operation is *atomic* if no other thread can see it partly executed

- Atomic as in “appears indivisible”
- Typically want ADT operations atomic, even to other threads running operations on the same ADT

## *Guideline #4: Atomicity*

- Think in terms of what operations need to be *atomic*
- Make critical sections just long enough to preserve atomicity
- *Then* design locking protocol to implement the critical sections

*In other words:*

*Think about atomicity first and locks second*

# *Do Not Roll Your Own*

- It is rare that you should write your own data structure
  - Excellent implementations provided in standard libraries
  - Point of CSE 332 is to understand the key trade-offs, abstractions, and analysis of such implementations
- Especially true for concurrent data structures
  - Far too difficult to provide fine-grained synchronization without race conditions
  - Standard **thread-safe** libraries like **ConcurrentHashMap** written by world experts

## **Guideline #5: Libraries**

*Use built-in libraries whenever they meet your needs*

# Motivating Memory-Model Issues

Tricky and *surprisingly wrong* unsynchronized concurrent code

```
class C {  
    private int x = 0;  
    private int y = 0;  
  
    void f() {  
        x = 1;  
        y = 1;  
    }  
    void g() {  
        int a = y;  
        int b = x;  
        assert(b >= a);  
    }  
}
```

First understand why it looks like the assertion cannot fail:

- Easy case: call to `g` ends before any call to `f` starts
- Easy case: at least one call to `f` completes before call to `g` starts
- If calls to `f` and `g` *interleave*...

# Interleavings are Not Enough

There is no interleaving of  $f$  and  $g$  where the assertion fails

- Proof #1: Exhaustively consider all possible orderings of access to shared memory (there are 6)

- Proof #2:

If  $!(b \geq a)$ , then  $a == 1$  and  $b == 0$ .

But if  $a == 1$ , then  $y = 1$  happened before  $a = y$ .

Because programs execute in order:

$a = y$  happened before  $b = x$  and  $x = 1$  happened before  $y = 1$ .

So by transitivity,  $b == 1$ . Contradiction.

Thread 1:  $f$

```
x = 1;
```

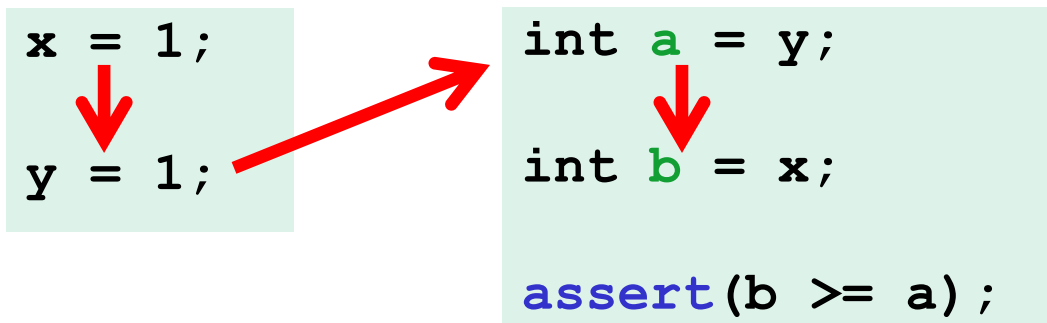
```
y = 1;
```

Thread 2:  $g$

```
int a = y;
```

```
int b = x;
```

```
assert(b >= a);
```



# Wrong

However, the code has a *data race*

- Unsynchronized read/write or write/write of same location

If code has data races, you cannot reason about it with interleavings

- This is simply the rules of Java (and C, C++, C#, other languages)
- Otherwise we would slow down all programs just to “help” those with data races, and that would not be a good engineering trade-off
- So the assertion can fail

# Why

For performance reasons, the compiler and the hardware will often reorder memory operations

- Take a compiler or computer architecture course to learn more

Thread 1: **f**

```
x = 1;
```

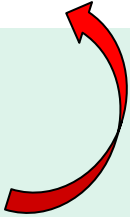
```
y = 1;
```

Thread 2: **g**

```
int a = y;
```

```
int b = x;
```

```
assert(b >= a);
```



Of course, we cannot just let them reorder anything they want

- Each thread computes things by executing code in order
- Consider: **x=17; y=x;**

# *The Grand Compromise*

The compiler/hardware will never perform a memory reordering that affects the result of a single-threaded program

The compiler/hardware will never perform a memory reordering that affects the result of a **data-race-free** multi-threaded program

So:      If no interleaving of your program has a data race,  
          then you can *forget about all this reordering nonsense*:  
          the result will be equivalent to some interleaving

Your job: Avoid data races

Compiler/hardware job: Give illusion of interleaving *if you do your job*



# Fixing Our Example

- Naturally, we can use synchronization to avoid data races
  - Then, indeed, the assertion cannot fail

```
class C {
    private int x = 0;
    private int y = 0;
    void f() {
        synchronized(this) { x = 1; }
        synchronized(this) { y = 1; }
    }
    void g() {
        int a, b;
        synchronized(this) { a = y; }
        synchronized(this) { b = x; }
        assert (b >= a) ;
    }
}
```

# *A Second Fix: Stay Away from This*

- Java has `volatile` fields: accesses do not count as data races
  - But you cannot read-update-write

```
class C {
    private volatile int x = 0;
    private volatile int y = 0;
    void f() {
        x = 1;
        y = 1;
    }
    void g() {
        int a = y;
        int b = x;
        assert(b >= a);
    }
}
```

- Implementation: slower than regular fields, faster than locks
- Really for experts: avoid them; use standard libraries instead
- And why do you need code like this anyway?

# Code That is Wrong

- Here is a more realistic example of code that is wrong
  - No *guarantee* Thread 2 will ever stop (as there is a data race)
  - But honestly it will “likely work in practice”

```
class C {  
    boolean stop = false;  
    void f() {  
        while(!stop) {  
            // draw a monster  
        }  
    }  
    void g() {  
        stop = didUserQuit();  
    }  
}
```

Thread 1: f()

Thread 2: g()

# Motivating Deadlock Issues

Consider a method to transfer money between bank accounts

```
class BankAccount {
    ...
    synchronized void withdraw(int amt) {...}
    synchronized void deposit(int amt) {...}
    synchronized void transferTo(int amt,
                                   BankAccount a) {
        this.withdraw(amt);
        a.deposit(amt);
    }
}
```

Notice during call to `a.deposit`, thread holds **two** locks

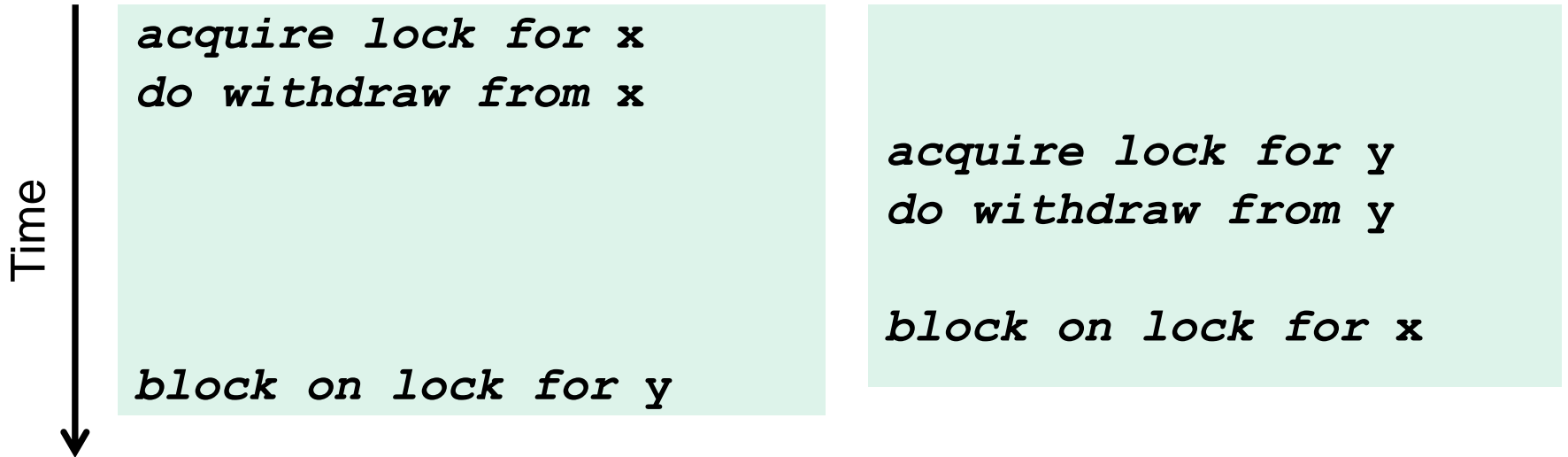
- Need to investigate when this may be a problem

# The Deadlock

Suppose **x** and **y** are fields holding accounts

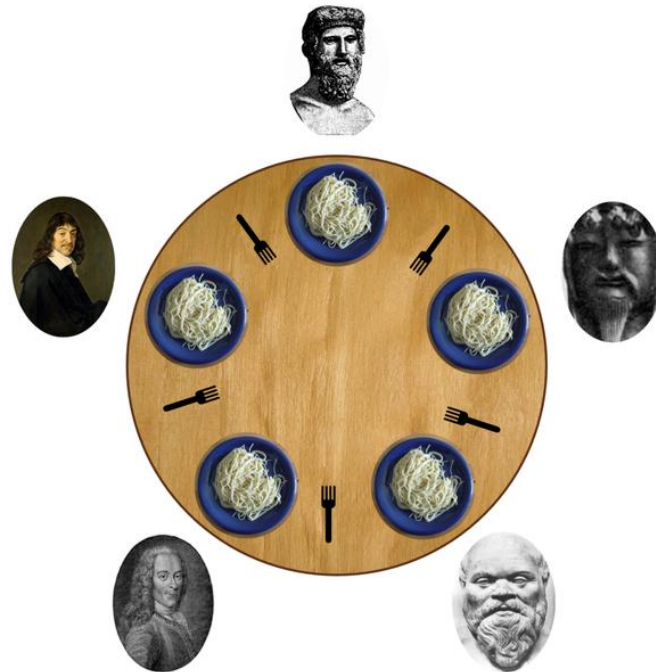
Thread 1: **x.transferTo(1, y)**

Thread 2: **y.transferTo(1, x)**



# *The Dining Philosophers*

- 5 philosophers go out to dinner together at an Italian restaurant
- Sit at a round table; one fork per setting
- When the spaghetti comes, each philosopher proceeds to grab their right fork, then their left fork, then eats
- 'Locking' for each fork results in a **deadlock**



# *Deadlock*

A deadlock occurs when there are threads **T1**, ..., **Tn** such that:

- For  $i=1, \dots, n-1$ , **T<sub>i</sub>** is waiting for a resource held by **T(i+1)**
- **T<sub>n</sub>** is waiting for a resource held by **T1**

In other words, there is a *cycle* of waiting

- Can formalize as a graph of dependencies with cycles bad

Deadlock avoidance in programming amounts to techniques to ensure a cycle can never arise

# *Back to Our Example*

Options for deadlock-proof transfer:

1. Make a smaller critical section: **transferTo** not synchronized
  - Exposes intermediate state after **withdraw** before **deposit**
  - May be okay, but exposes wrong total amount in bank
2. Coarsen lock granularity:  
one lock for all accounts allowing transfers between them
  - Works, but sacrifices concurrent deposits/withdrawals
3. Give every bank-account a unique number  
and always acquire locks in the same order
  - *Entire program* should obey this order to avoid cycles
  - Code acquiring only one lock can ignore the order



# Ordering Locks

```
class BankAccount {
    ...
    private int acctNumber; // must be unique
    void transferTo(int amt, BankAccount a) {
        if(this.acctNumber < a.acctNumber)
            synchronized(this) {
                synchronized(a) {
                    this.withdraw(amt);
                    a.deposit(amt);
                }
            }
        else
            synchronized(a) {
                synchronized(this) {
                    this.withdraw(amt);
                    a.deposit(amt);
                }
            }
    }
}
```

# StringBuffer Example

From the Java standard library

```
class StringBuffer {
    private int count;
    private char[] value;
    ...
    synchronized append(StringBuffer sb) {
        int len = sb.length();
        if(this.count + len > this.value.length)
            this.expand(...);
        sb.getChars(0, len, this.value, this.count);
    }
    ...
    synchronized getChars(int x, int, y,
                           char[] a, int z) {
        "copy this.value[x..y] into a starting at z"
    }
}
```

# *Two Problems*

## Problem #1:

- Lock for `sb` not held between calls to `sb.length` and `sb.getChars`
  - So `sb` could get longer
  - Would cause `append` to throw an `ArrayBoundsException`

## Problem #2:

Deadlock potential if two threads try to `append` in opposite directions, identical to the bank-account first example

Not easy to fix both problems without extra copying:

- Do not want unique ids on every `StringBuffer`
- Do not want one lock for all `StringBuffer` objects

Actual Java library: fixed neither (left code as is; changed documentation)

- Up to clients to avoid such situations with own protocols

# *Perspective*

- Code like account-transfer and string-buffer append are difficult to deal with for deadlock
- Easier case: different types of objects
  - Can document a fixed order among types
  - Example: “When moving an item from the hashtable to the work queue, never try to acquire the queue lock while holding the hashtable lock”
- Easier case: objects are in an acyclic structure
  - Can use the data structure to determine a fixed order
  - Example: “If holding a tree node’s lock, do not acquire other tree nodes’ locks unless they are children in the tree”



# CSE332: Data Abstractions

## Lecture 21: Readers/Writer Locking

James Fogarty  
Winter 2012

Including slides developed in part by  
Ruth Anderson, James Fogarty, Dan Grossman

# *Reading vs. Writing*

Recall:

- Multiple concurrent reads of same memory: *Not* a problem
- Multiple concurrent writes of same memory: Problem
- Multiple concurrent read & write of same memory: Problem

So far:

- If concurrent write/write or read/write might occur, use synchronization to ensure one-thread-at-a-time

But this is unnecessarily conservative:

- Could still allow multiple simultaneous readers!

# *Example*

Consider a hashtable with one coarse-grained lock

- So only one thread can perform operations at a time

But suppose:

- There are many simultaneous **lookup** operations
- **insert** operations are very rare

Note: Important that **lookup** does not actually mutate shared memory, like a move-to-front list operation would

# Readers/Writer locks

A new synchronization ADT: The **readers/writer lock**

- A lock's states fall into three categories:
  - “not held”
  - “held for writing” by one thread
  - “held for reading” by *one or more* threads

**$0 \leq \text{writers} \leq 1$**   
 **$0 \leq \text{readers}$**   
 **$\text{writers} * \text{readers} == 0$**

- **new**: make a new lock, initially “not held”
- **acquire\_write**: block if currently “held for reading” if or “held for writing”, else make “held for writing”
- **release\_write**: make “not held”
- **acquire\_read**: block if currently “held for writing”, else make/keep “held for reading” and increment *readers count*
- **release\_read**: decrement readers count, if 0, make “not held”



# Pseudocode Example (not Java)

```
class Hashtable<K,V> {  
    ...  
    // coarse-grained, one lock for table  
    RWLock lk = new RWLock();  
    V lookup(K key) {  
        int bucket = hasher(key);  
        lk.acquire_read();  
        ... read array[bucket] ...  
        lk.release_read();  
    }  
    void insert(K key, V val) {  
        int bucket = hasher(key);  
        lk.acquire_write();  
        ... write array[bucket] ...  
        lk.release_write();  
    }  
}
```

# *Readers/Writer Lock Details*

- A readers/writer lock implementation (which is “not our problem”) usually gives *priority* to writers:
  - After a writer blocks, no readers *arriving later* will get the lock before the writer
  - Otherwise an **insert** could *starve*
- Re-entrant?
  - Mostly an orthogonal issue
  - But some libraries support *upgrading* from reader to writer
- Why not use readers/writer locks with more fine-grained locking?
  - Like on each bucket?
  - Not wrong, but likely not worth it due to low contention

# *In Java*

[Note: Not needed in your project/homework]

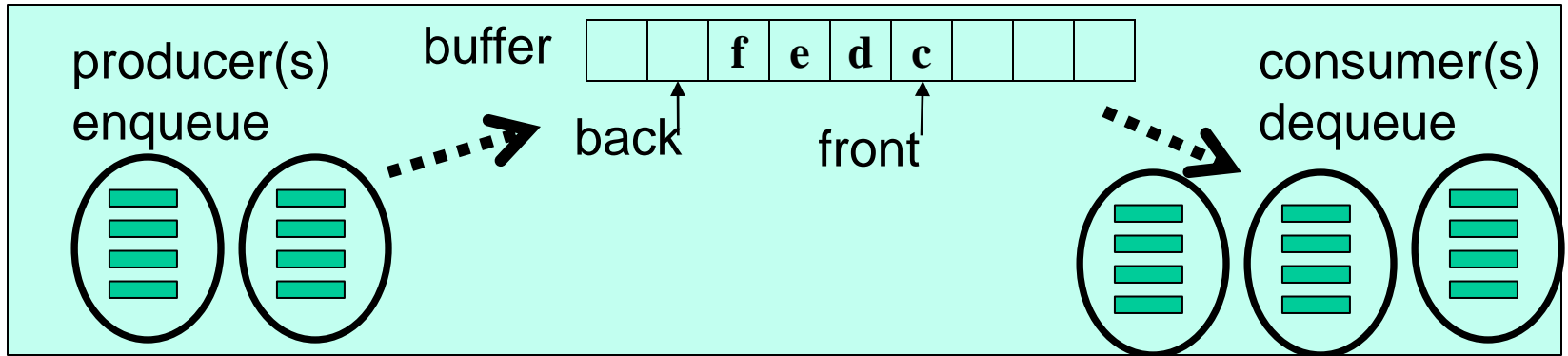
Java's **synchronized** statement does not support readers/writer

Instead, library

`java.util.concurrent.locks.ReentrantReadWriteLock`

- Different interface: methods `readLock` and `writeLock` return objects that themselves have `lock` and `unlock` methods
- Does *not* have writer priority or reader-to-writer upgrading
  - Always read the documentation

# Motivating Condition Variables



To motivate condition variables, consider the canonical example of a **bounded buffer** for sharing work among threads

Bounded buffer: A queue with a fixed size

- Only slightly simpler if unbounded, core need still arises

For sharing work – think an assembly line:

- Producer thread(s) do some work and enqueue result objects
- Consumer thread(s) dequeue objects and do next stage
- Must synchronize access to the queue

# First Attempt

```
class Buffer<E> {
    E[] array = (E[])new Object[SIZE];
    ... // front, back fields, isEmpty, isFull methods
    synchronized void enqueue(E elt) {
        if(isFull())
            ???
        else
            ... add to array and adjust back ...
    }
    synchronized E dequeue()
        if(isEmpty())
            ???
        else
            ... take from array and adjust front ...
    }
}
```

# Waiting

- **enqueue** to a full buffer should *not* raise an exception
  - Wait until there is room
- **dequeue** from an empty buffer should *not* raise an exception
  - Wait until there is data

**Bad approach** is to *spin* (wasted work and keep grabbing lock)

```
void enqueue(E elt) {
    while(true) {
        synchronized(this) {
            if(isFull()) continue;
            ... add to array and adjust back ...
            return;
        }
    }
}
// dequeue similar
```

# *What we Want*

- Better would be for a thread to *wait* until it can proceed
  - Be *notified* when it should try again
  - In the meantime, let other threads run
- Like locks, not something you can implement on your own
  - Language or library gives it to you, typically implemented with operating-system support
- An ADT that supports this: **condition variable**
  - Informs waiter(s) when the *condition* that causes it/them to wait has *varied*
- Terminology not completely standard; will mostly stick with Java

# Java Approach: Not Quite Right

```
class Buffer<E> {  
    ...  
    synchronized void enqueue(E elt) {  
        if(isFull())  
            this.wait(); // releases lock and waits  
            add to array and adjust back  
        if(buffer was empty)  
            this.notify(); // wake somebody up  
    }  
    synchronized E dequeue() {  
        if(isEmpty())  
            this.wait(); // releases lock and waits  
            take from array and adjust front  
        if(buffer was full)  
            this.notify(); // wake somebody up  
    }  
}
```



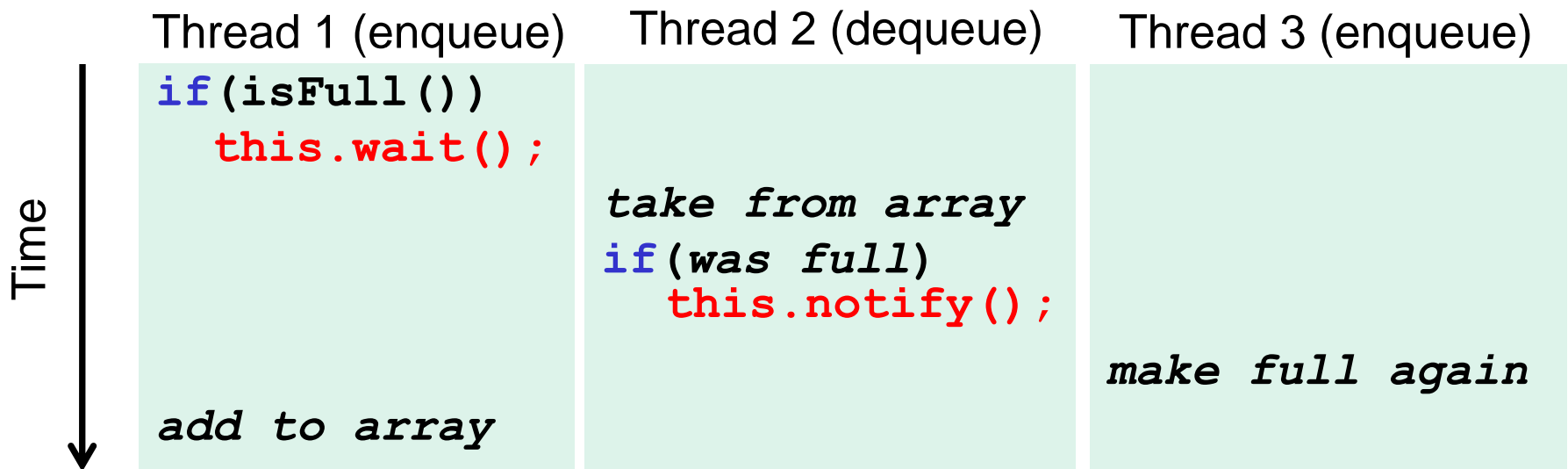
# Key Ideas

- Java weirdness: every object “is” a condition variable (also a lock)
  - other languages/libraries often make them separate
- **wait:**
  - “register” running thread as interested in being woken up
  - then atomically: release the lock and block
  - when execution resumes, *thread again holds the lock*
- **notify:**
  - pick one waiting thread and wake it up
  - no guarantee woken up thread runs next, just that it is no longer blocked on the *condition*, now waiting for the *lock*
  - if no thread is waiting, then do nothing

# Bug

```
synchronized void enqueue(E elt) {  
    if(isFull())  
        this.wait();  
    add to array and adjust back  
    ...  
}
```

Between the time a thread is notified and it re-acquires the lock, the condition can become false again!



# Bug Fix

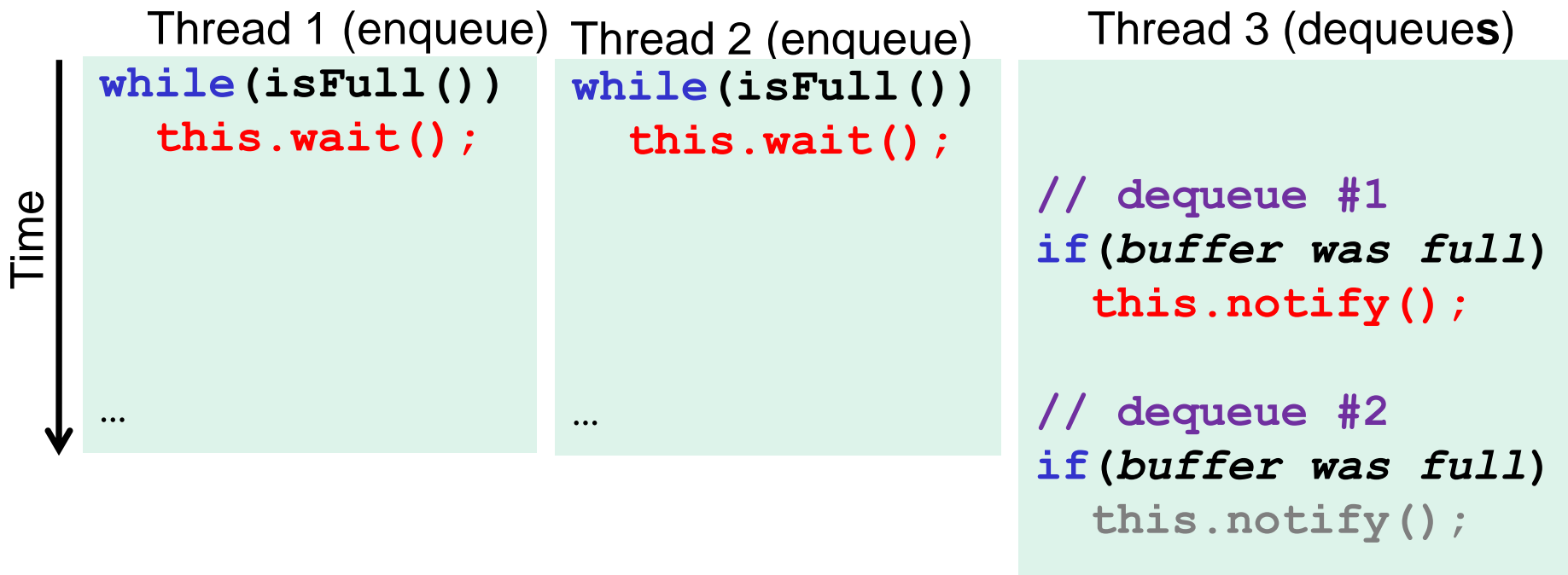
```
synchronized void enqueue(E elt) {
    while (isFull())
        this.wait();
    ...
}
synchronized E dequeue() {
    while (isEmpty())
        this.wait();
    ...
}
```

Guideline: *Always* re-check the condition after re-gaining the lock

- For obscure reasons, Java is technically allowed to notify a thread *spuriously* (i.e., for no reason without any call to **notify**)

# Another Bug

- If multiple threads are waiting, we wake up only one
  - Sure only one can do work *now*, but cannot forget the others!



# Bug Fix

```
synchronized void enqueue(E elt) {  
    ...  
    if(buffer was empty)  
        this.notifyAll(); // wake everybody up  
}  
synchronized E dequeue() {  
    ...  
    if(buffer was full)  
        this.notifyAll(); // wake everybody up  
}
```

`notifyAll` wakes up all current waiters on the condition variable

Guideline: If in any doubt, use `notifyAll`

- Wasteful waking is much better than never waking up (because you already need to re-check condition)
- So why does `notify` exist?
  - Well, it is faster when correct...

# *Alternate Approach*

- An alternative is to call `notify` (not `notifyAll`) on every `enqueue` / `dequeue`, not just when the buffer was empty / full
  - Easy: just remove the `if` statement
- Alas, makes our code subtly `wrong` since it is technically possible that an `enqueue` and a `dequeue` are both waiting.
  - See notes for the step-by-step details of how this can happen
- Works fine if buffer is unbounded because only dequeuers wait

# *Alternate Approach Fixed*

- The alternate approach works if the enqueueers and dequeuers wait on *different* condition variables
  - But for mutual exclusion both condition variables must be associated with the same lock
- Java’s “everything is a lock / condition variable” does not support this: each condition variable is associated with itself
- Instead, Java has classes in `java.util.concurrent.locks` for when you want multiple conditions with one lock
  - `class ReentrantLock` has a method `newCondition` that returns a new `Condition` object associate with the lock
  - See the documentation if curious

# *Final Comments on Condition-Variable*

- `notify/notifyAll` often called `signal/broadcast` or `pulse/pulseAll`
- Condition variables are subtle and harder to use than locks
- But when you need them, you need them
  - Spinning and other work-arounds do not work well
- Fortunately, like most things you see in a data-structures course, the common use-cases are provided in libraries written by experts
  - Example:  
`java.util.concurrent.ArrayBlockingQueue<E>`
    - All condition variables hidden; just call `put` and `take`



# *Concurrency summary*

- Access to shared resources introduces new kinds of bugs
  - Data races
  - Critical sections too small
  - Critical sections use wrong locks
  - Deadlocks
- Requires synchronization
  - Locks for mutual exclusion (common, various flavors)
  - Condition variables for signaling others (less common)
- Guidelines for correct use help avoid common pitfalls
- Not always clear shared-memory is worth the pain
  - But other models not a panacea (e.g., message passing)



# CSE332: Data Abstractions

## Lecture 22: Minimum Spanning Trees

James Fogarty

Winter 2012

# *Making Connections*

You have a set of nodes (numbered 1-9) on a network. You are given a sequence of pairwise connections between them:

3-5

4-2

1-6

5-7

4-8

3-7

**Q:** Are nodes 2 and 4 connected? Indirectly?

**Q:** How about nodes 3 and 8?

**Q:** Are any of the paired connections redundant due to indirect connections?

**Q:** How many sub-networks do you have?

# Making Connections

Answering these questions is much easier if we create disjoint sets of nodes that are connected:

Start: {1} {2} {3} {4} {5} {6} {7} {8} {9}  
3-5 {1} {2} {3, 5} {4} {6} {7} {8} {9}  
4-2 {1} {2, 4} {3, 5} {6} {7} {8} {9}  
1-6 {1, 6} {2, 4} {3, 5} {7} {8} {9}  
5-7 {1, 6} {2, 4} {3, 5, 7} {8} {9}  
4-8 {1, 6} {2, 4, 8} {3, 5, 7} {9}  
3-7

**Q:** Are nodes 2 and 4 connected? Indirectly?

**Q:** How about nodes 3 and 8?

**Q:** Are any of the paired connections redundant due to indirect connections?

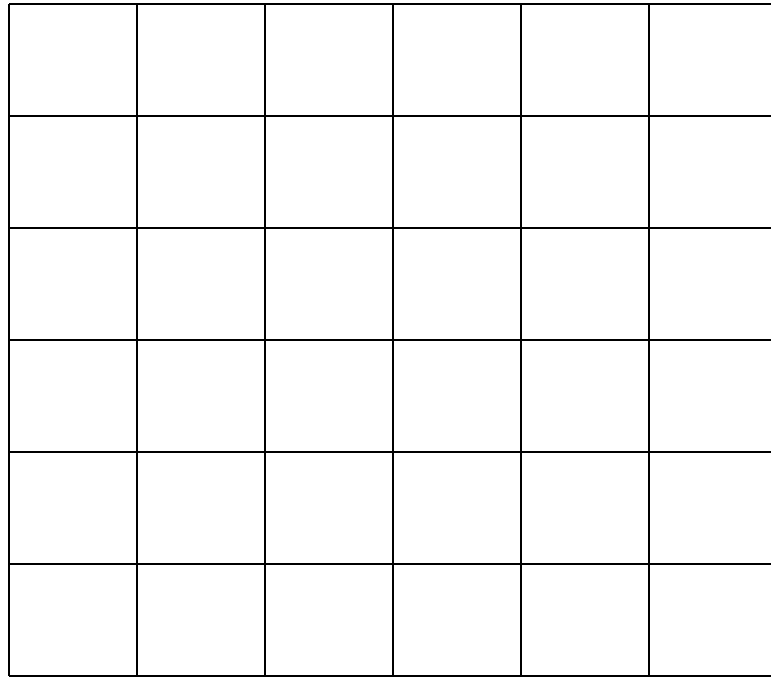
**Q:** How many sub-networks do you have?

# Union-Find aka Disjoint Set ADT

- **Union(x,y)** – take the union of two sets named x and y
  - Given sets: {3,5,7} , {4,2,8}, {9}, {1,6}
  - **Union(5,1)**  
Result: {3,5,7,1,6}, {4,2,8}, {9},
  - To perform the union operation, we replace sets x and y by  $(x \cup y)$
- **Find(x)** – return the name of the set containing x.
  - Given sets: {3,5,7,1,6}, {4,2,8}, {9},
  - **Find(1)** returns 5
  - **Find(4)** returns 8
- We can do Union in constant time.
- We can get Find to be **amortized** constant time (worst case  $O(\log n)$  for an individual Find operation).

# *Cute Application*

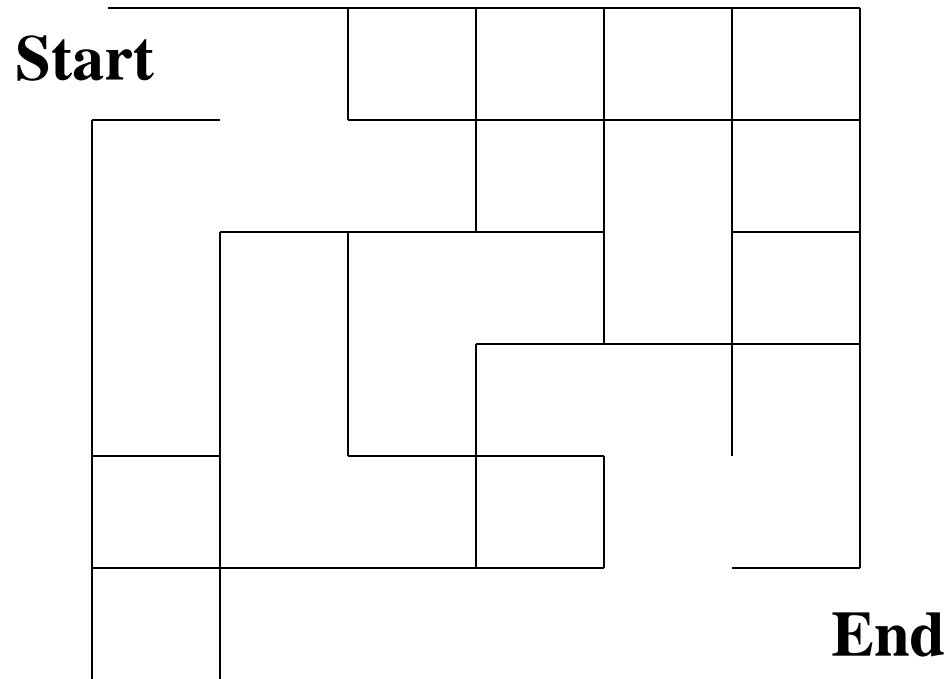
- Build a random maze by erasing edges.





# *Cute Application*

- Repeatedly pick random edges to delete.





# *Number the Cells*

Disjoint sets  $\mathbf{S} = \{ \{1\}, \{2\}, \{3\}, \{4\}, \dots, \{36\} \}$ , each cell is unto itself.  
We have all edges  $\mathbf{W} = \{ (1,2), (1,7), (2,8), (2,3), \dots \}$  60 walls total.

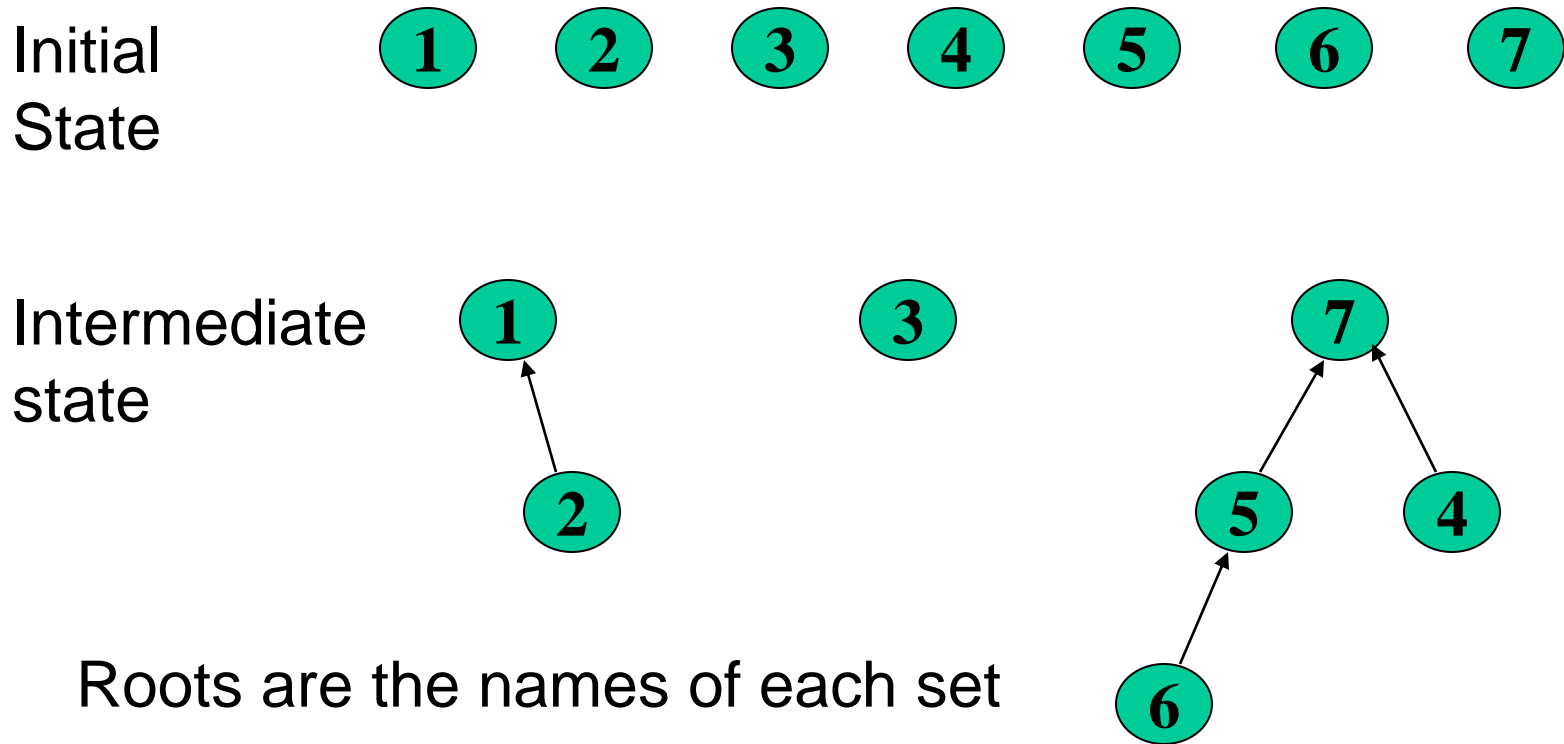
<b>Start</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	
	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>	<b>11</b>	<b>12</b>	
	<b>13</b>	<b>14</b>	<b>15</b>	<b>16</b>	<b>17</b>	<b>18</b>	
	<b>19</b>	<b>20</b>	<b>21</b>	<b>22</b>	<b>23</b>	<b>24</b>	
	<b>25</b>	<b>26</b>	<b>27</b>	<b>28</b>	<b>29</b>	<b>30</b>	
	<b>31</b>	<b>32</b>	<b>33</b>	<b>34</b>	<b>35</b>	<b>36</b>	<b>End</b>

# *Maze Building with Disjoint Union/Find*

- Algorithm sketch:
  - Choose wall at random.
    - Boundary walls are not in wall list, because we cannot delete them
  - Erase wall if the neighbors are in disjoint sets
    - Avoids cycles
  - Take union of those sets
  - Repeat until there is only one set
    - Every cell reachable from every other cell

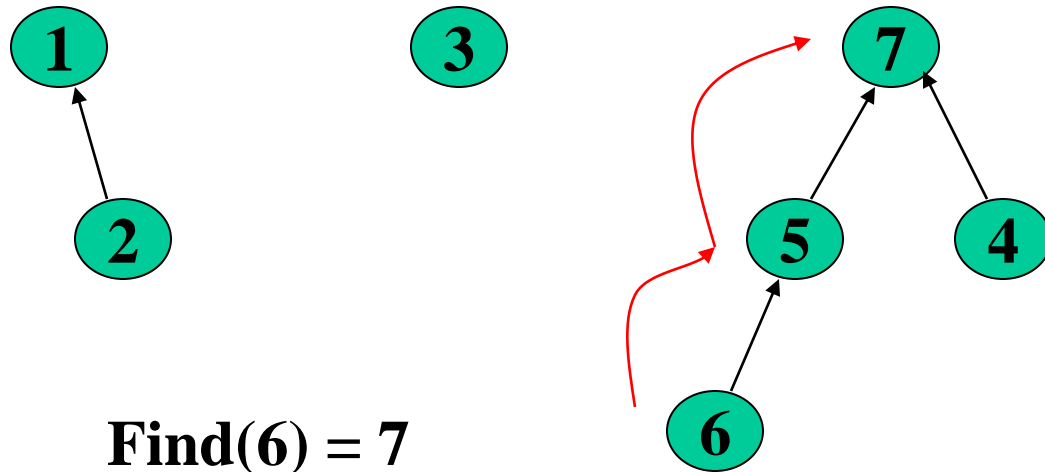


# *Up-Tree for Disjoin Union/Find*



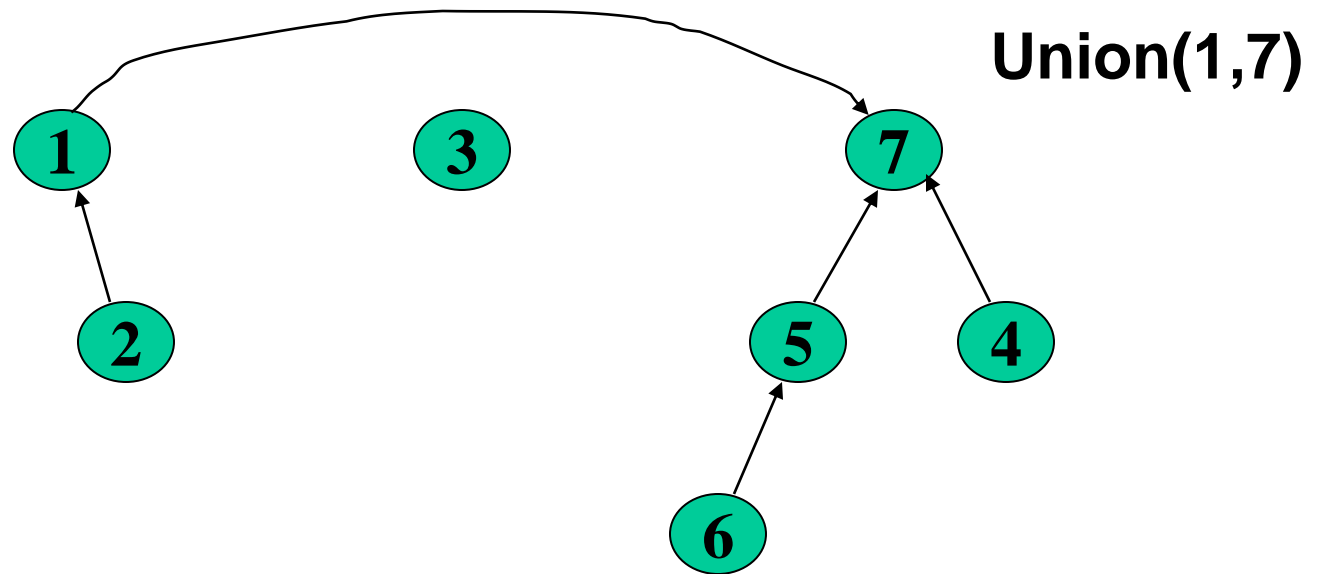
# Find Operation

- Find(x):  
follow x to the root and return the root



# *Union Operation*

- Union(i,j):  
assuming i and j roots, point i to j.

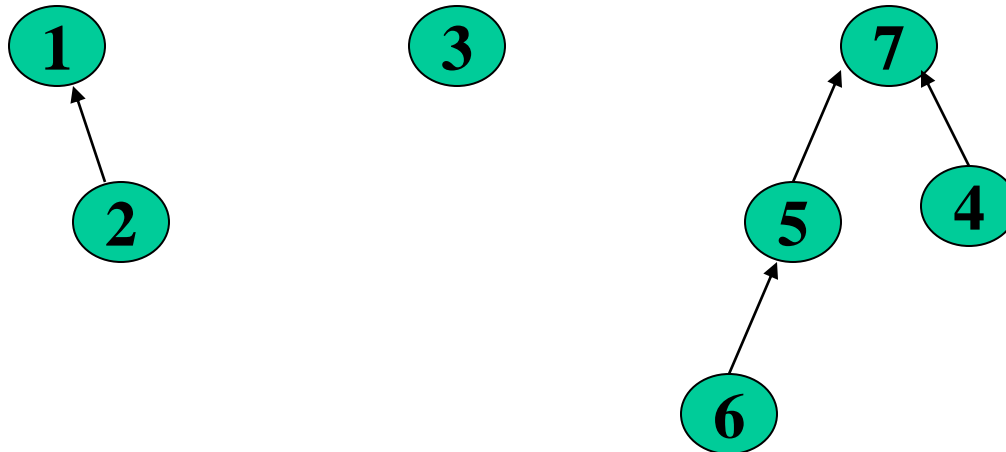


# Simple Implementation

- Array of indices

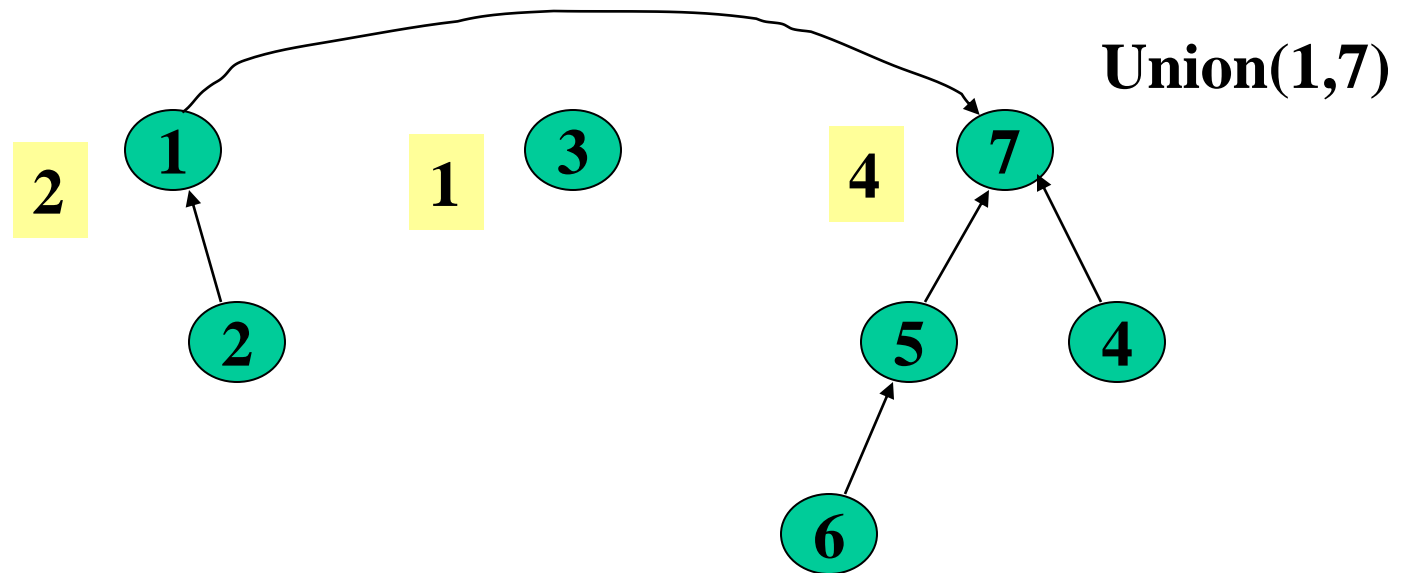
	1	2	3	4	5	6	7
up	0	1	0	7	7	5	0

**Up[x] =**  
**0** means  
**x** is a root



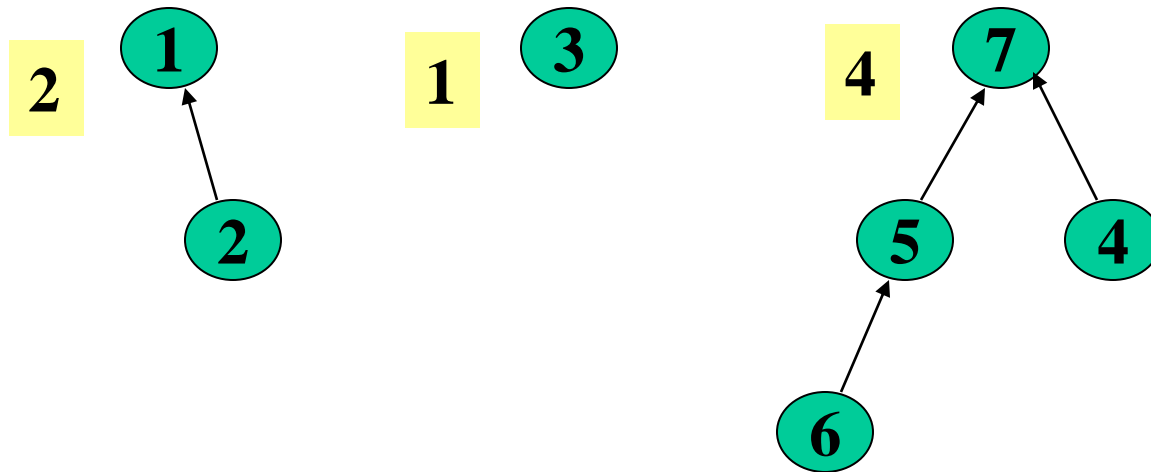
# Weighted Union

- Weighted Union
  - Instead of arbitrarily joining two roots, always point the smaller root to the larger root





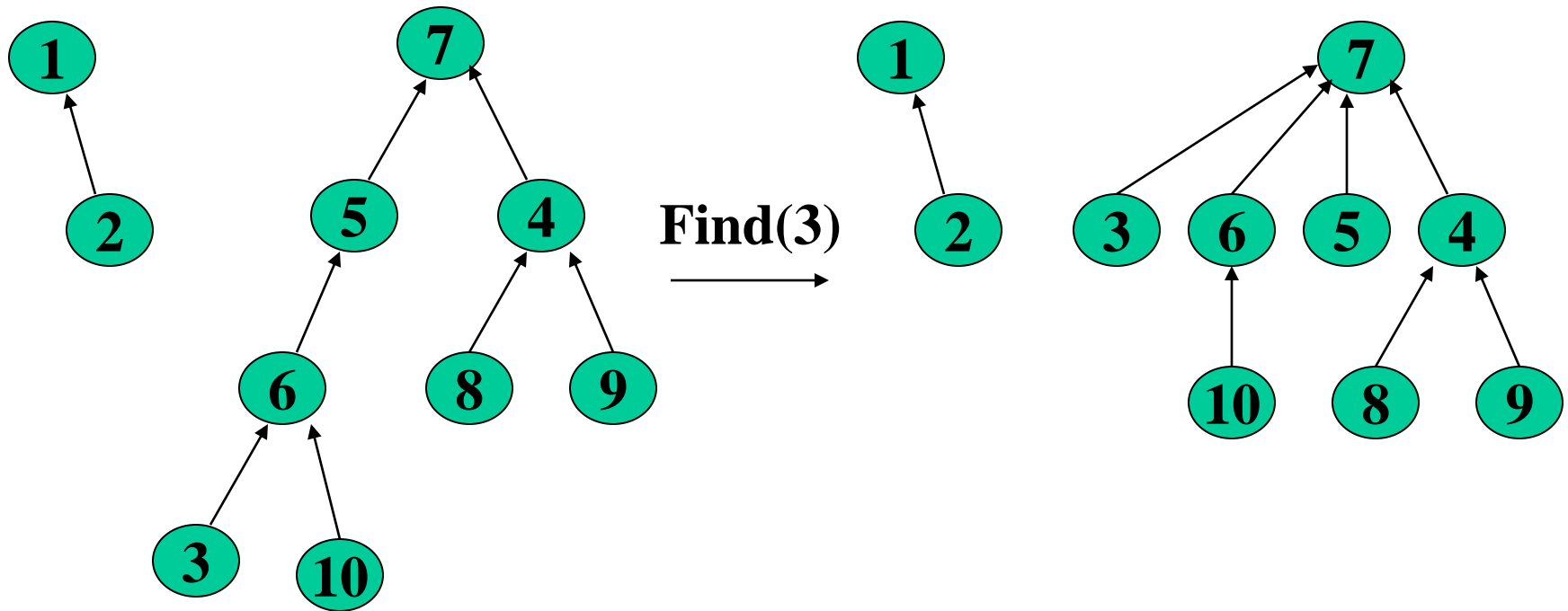
# *Elegant Array Implementation*



	1	2	3	4	5	6	7
up	0	1	0	7	7	5	0
weight	2		1				4

# Path Compression

- On a Find operation point all the nodes on the search path directly to the root.

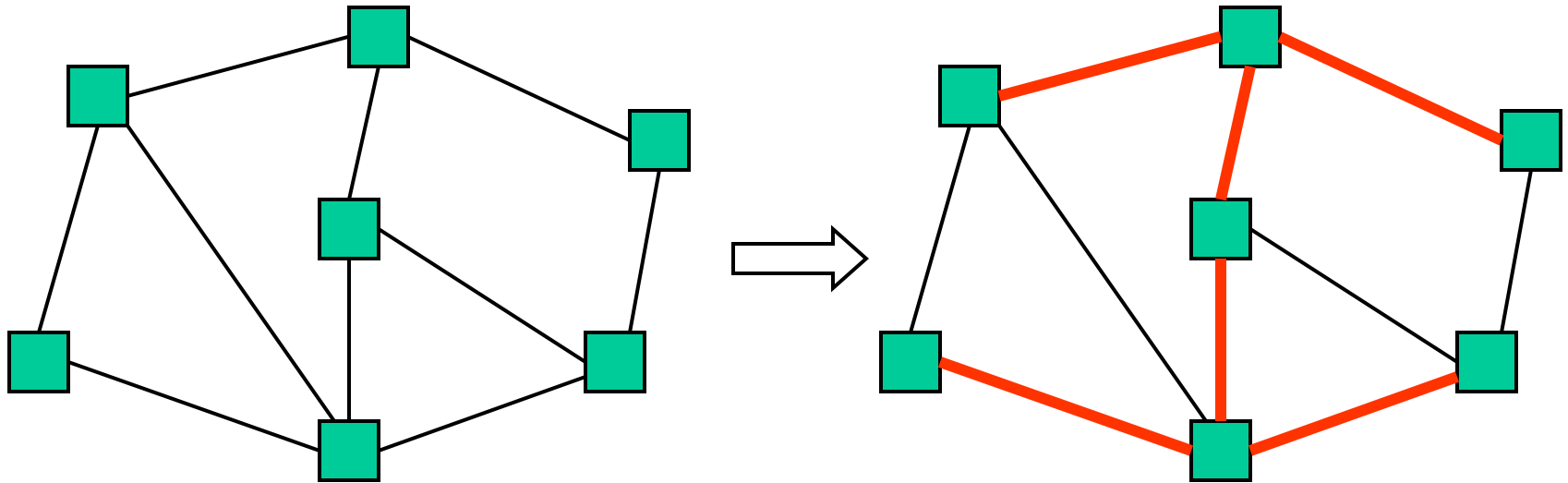


# Analyzing Disjoint Sets

- For  $n$  elements, total cost of  $m$  finds, at most  $n-1$  unions
- Total work is:  $O(m+n)$ , i.e.  $O(1)$  amortized
  - With  $O(1)$  worst-case for union
  - And  $O(\log n)$  worst-case for find
- Find and union *cannot* both be worst-case  $O(1)$

# Spanning Trees

- A simple problem: Given a *connected* graph  $\mathbf{G}=(\mathbf{V},\mathbf{E})$ , find a minimal subset of the edges such that the graph is still connected
  - A graph  $\mathbf{G2}=(\mathbf{V},\mathbf{E2})$  such that  $\mathbf{G2}$  is connected and removing any edge from  $\mathbf{E2}$  makes  $\mathbf{G2}$  disconnected



# *Observations*

1. Any solution to this problem is a tree
  - Recall a tree does not need a root; just means acyclic
  - For any cycle, could remove an edge and still be connected
2. Solution not unique unless original graph was already a tree
3. Problem ill-defined if original graph not connected
4. A tree with  $|V|$  nodes has  $|V|-1$  edges
  - Every spanning tree solution has  $|V|-1$  edges

# *Motivation*

A **spanning tree** connects all the nodes with as few edges as possible

- Example: A “phone tree” so everybody gets the message and no unnecessary calls get made
  - Bad example since would prefer a balanced tree

In most compelling uses, we have a *weighted* undirected graph and we want a tree of least total cost

- Example: Electrical wiring for a house or clock wires on a chip
- Example: Road network if you cared about asphalt cost

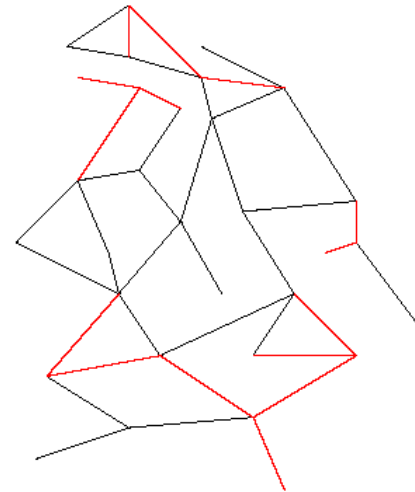
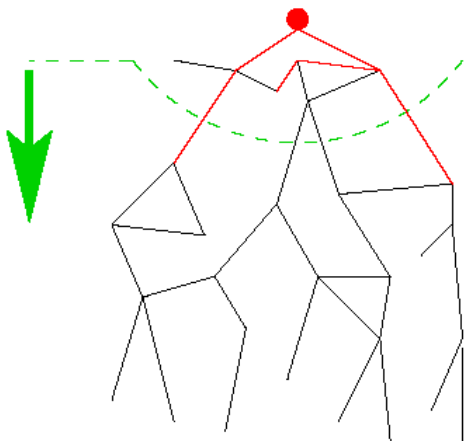
This is the **minimum spanning tree** problem

- Will do that next, after intuition from the simpler case

# *Two Approaches*

Different algorithmic approaches to the spanning-tree problem:

1. Do a graph traversal  
(e.g., depth-first search, but any traversal will do),  
keeping track of edges that form a tree
2. Iterate through edges;  
add to output any edge that doesn't create a cycle



# Spanning Tree via DFS

```
spanning_tree(Graph G) {
    for each node i: i.marked = false
    for some node i: f(i)
}
f(Node i) {
    i.marked = true
    for each j adjacent to i:
        if(!j.marked) {
            add(i,j) to output
            f(j) // DFS
        }
}
```

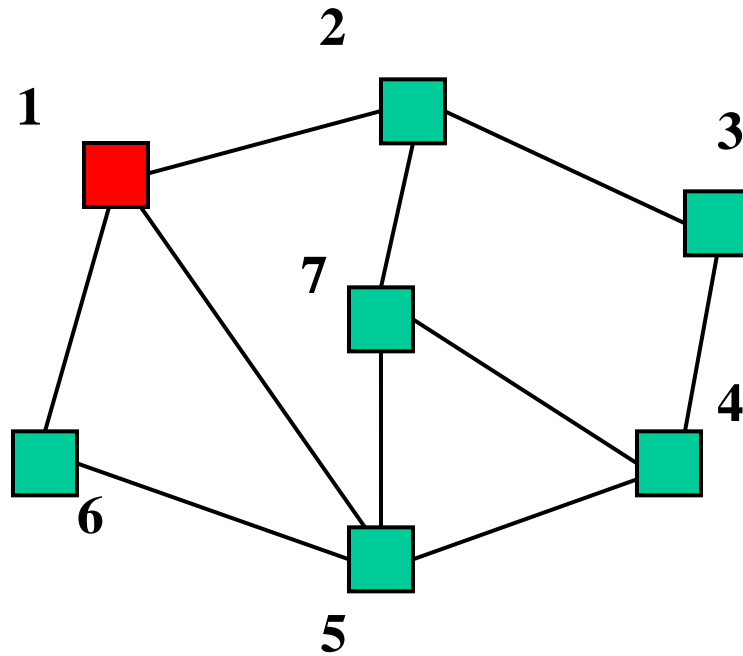
Correctness: DFS reaches each node. We add one edge to connect it to the already visited nodes. Order affects result, not correctness.

Time:  $O(|E|)$



# Example

Stack  
f(1)



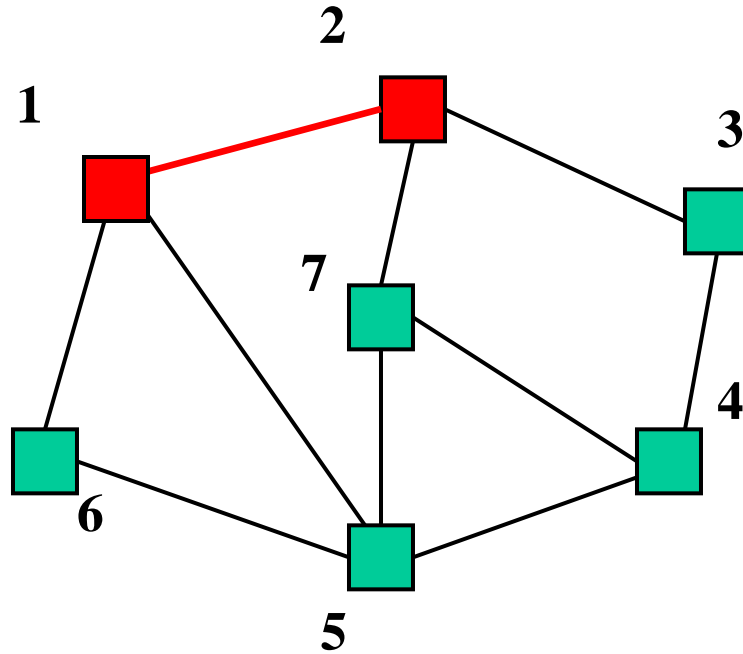
Output:

# Example

Stack

f(1)

f(2)



Output: (1,2)

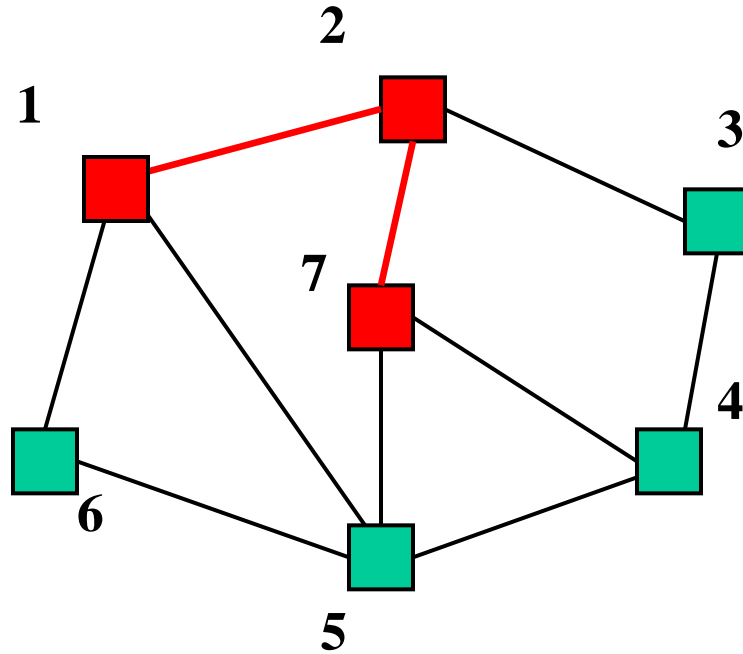
# Example

Stack

f(1)

f(2)

f(7)



Output: (1,2), (2,7)

# Example

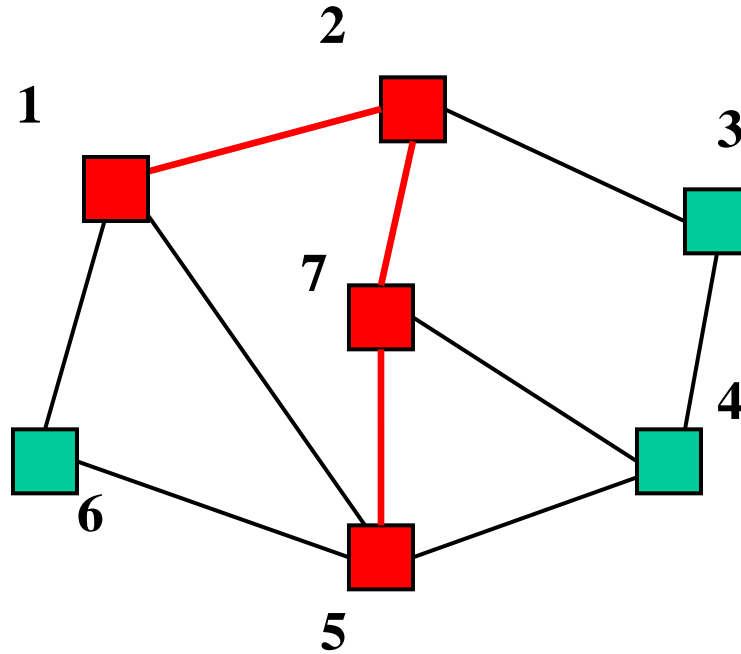
Stack

f(1)

f(2)

f(7)

f(5)



Output: (1,2), (2,7), (7,5)

# Example

Stack

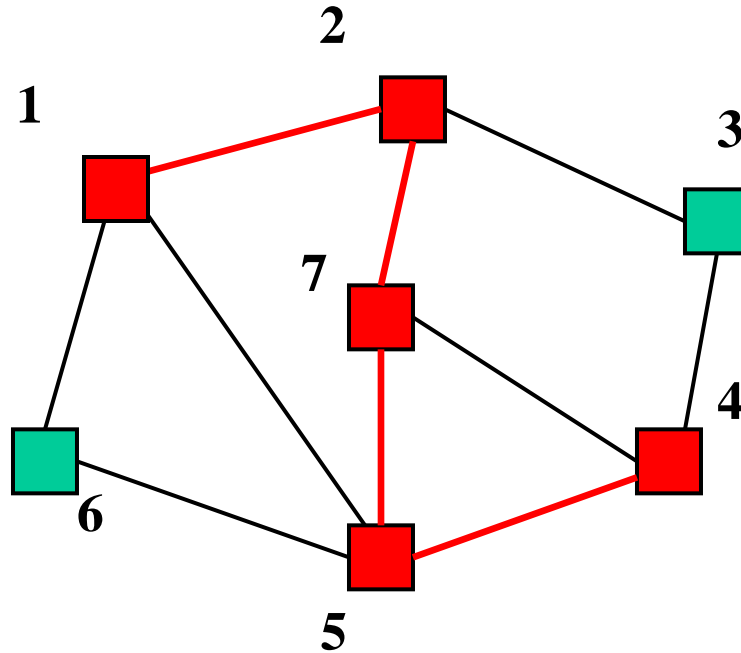
f(1)

f(2)

f(7)

f(5)

f(4)



Output: (1,2), (2,7), (7,5), (5,4)

# Example

Stack

f(1)

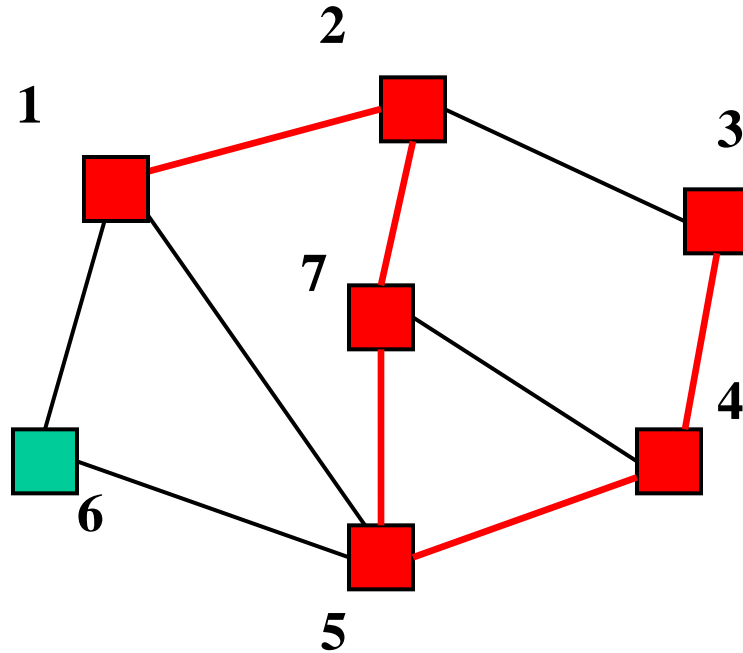
f(2)

f(7)

f(5)

f(4)

f(3)



Output: (1,2), (2,7), (7,5), (5,4),(4,3)

# Example

Stack

f(1)

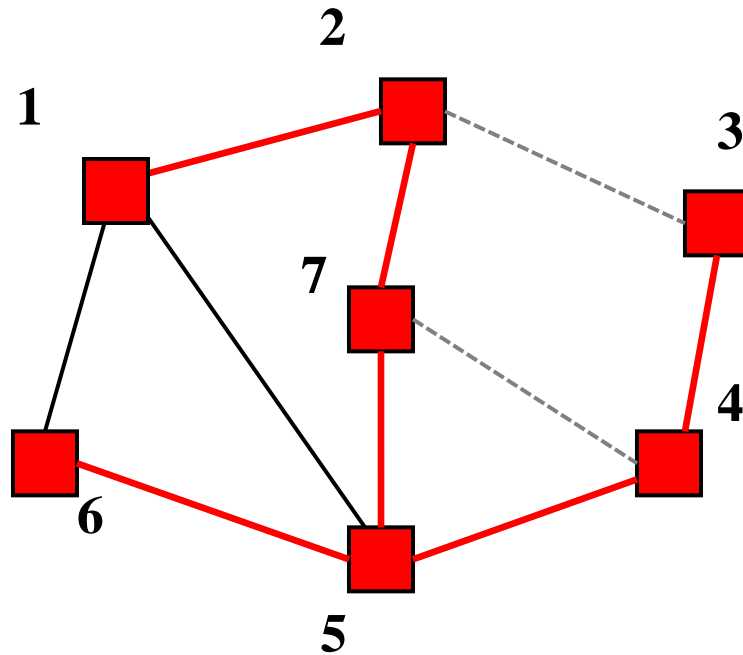
f(2)

f(7)

f(5)

f(4) f(6)

f(3)

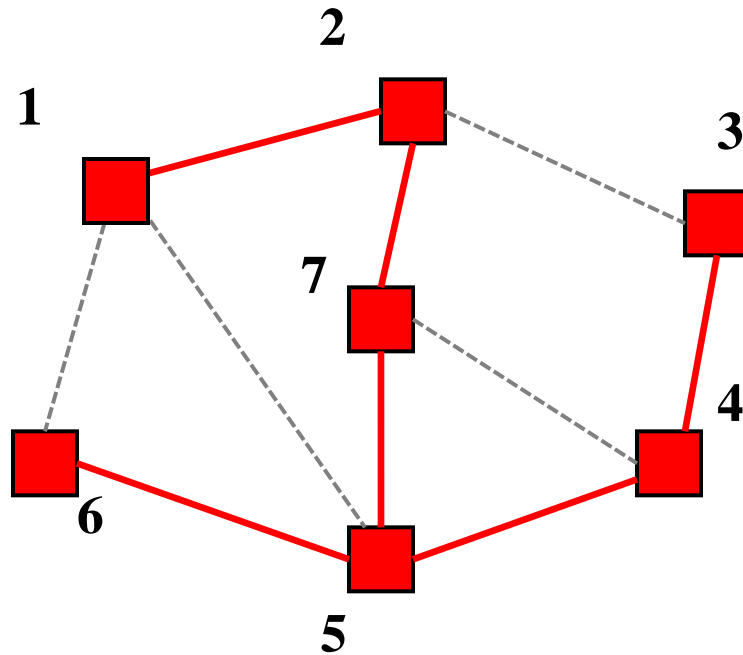


Output: (1,2), (2,7), (7,5), (5,4), (4,3), (5,6)

# Example

Stack

f(1)  
f(2)  
f(7)  
f(5)  
f(4) f(6)  
f(3)



Output: (1,2), (2,7), (7,5), (5,4), (4,3), (5,6)



# *Second Approach*

Iterate through edges; output any edge that does not create a cycle

Correctness (hand-wavy):

- Goal is to build an acyclic connected graph
- When we add an edge, it adds a vertex to the tree (or else it would have created a cycle)
- The graph is connected, we consider all edges

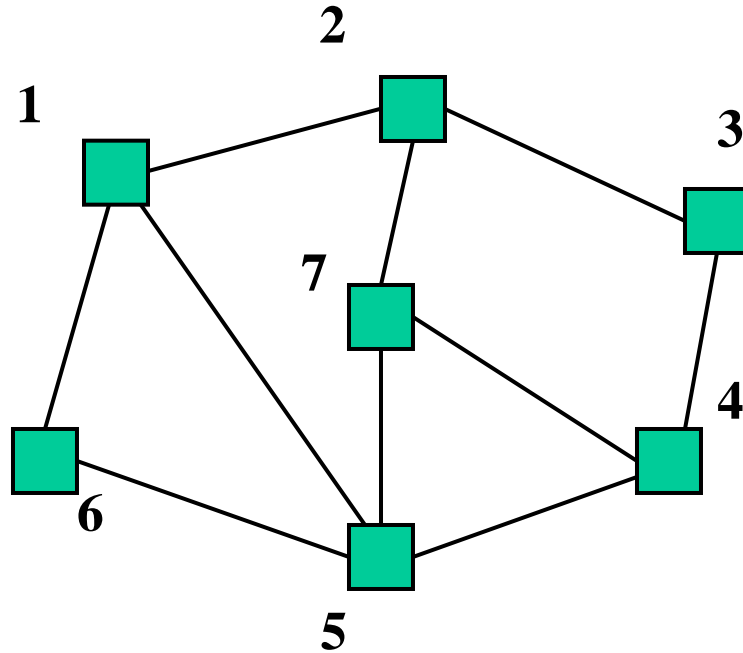
Efficiency:

- Depends on how quickly you can detect cycles
- Reconsider after the example

# Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

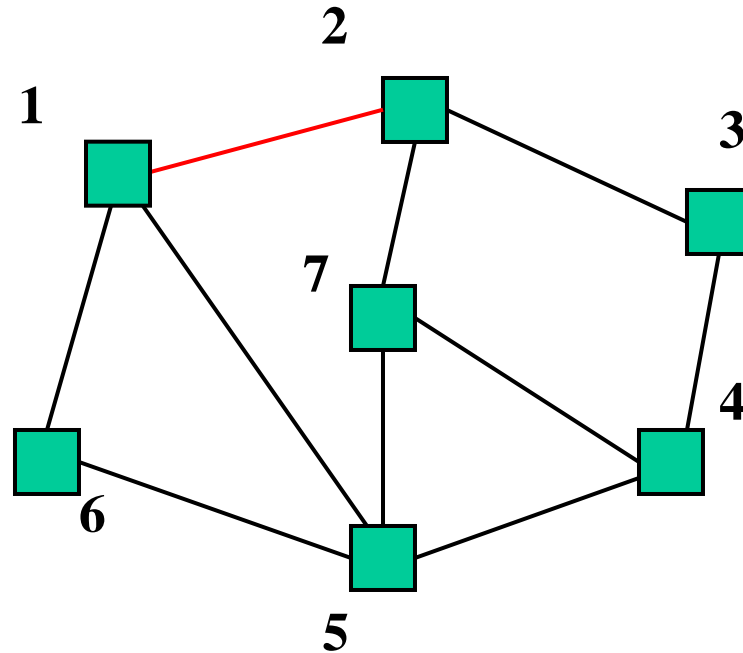


Output:

# Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

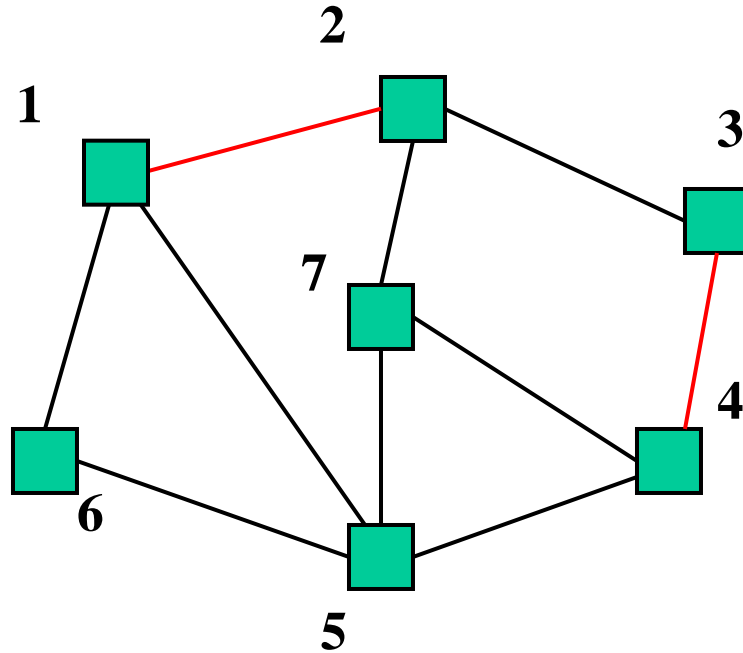


Output: (1,2)

# Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

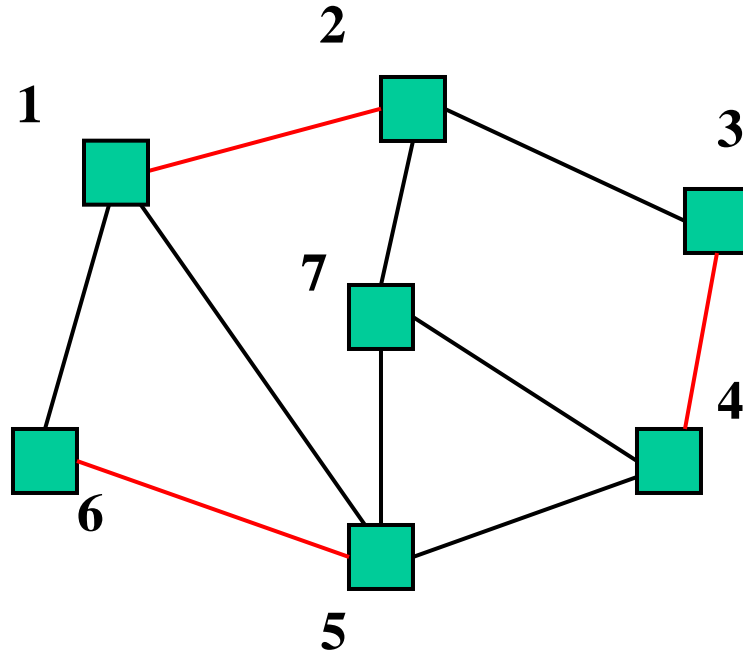


Output: (1,2), (3,4)

# Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

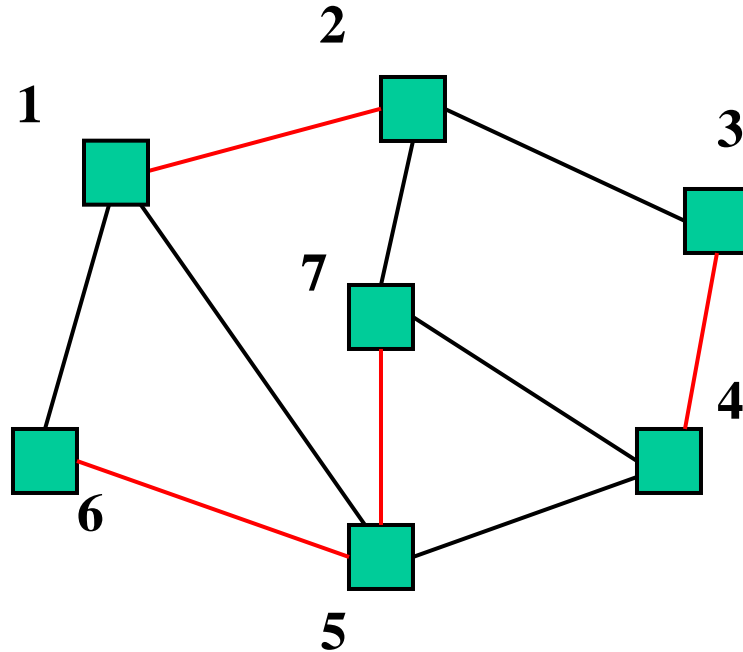


Output: (1,2), (3,4), (5,6),

# Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

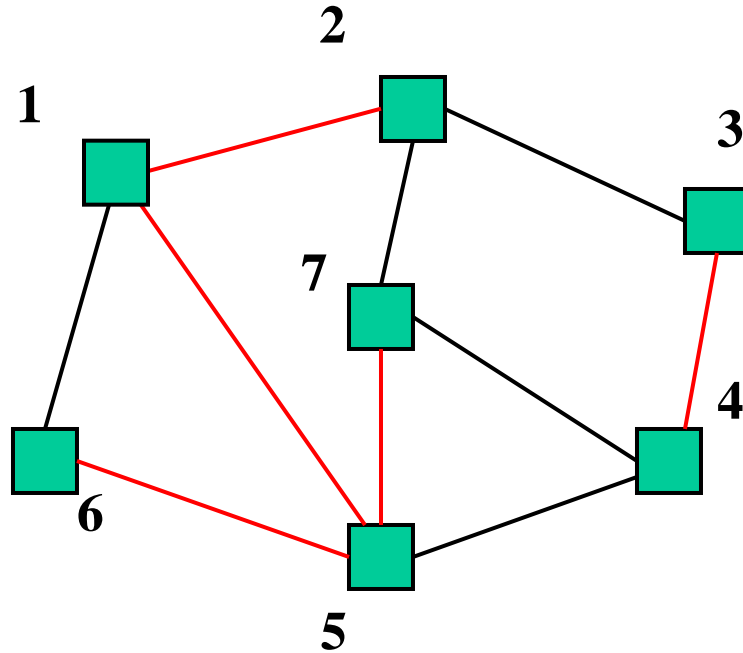


Output: (1,2), (3,4), (5,6), (5,7)

# Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

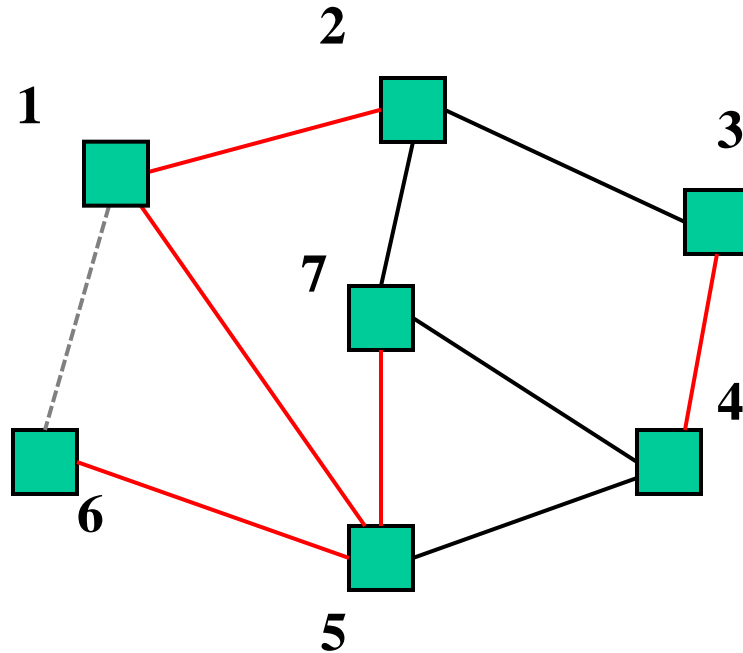


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

# Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)



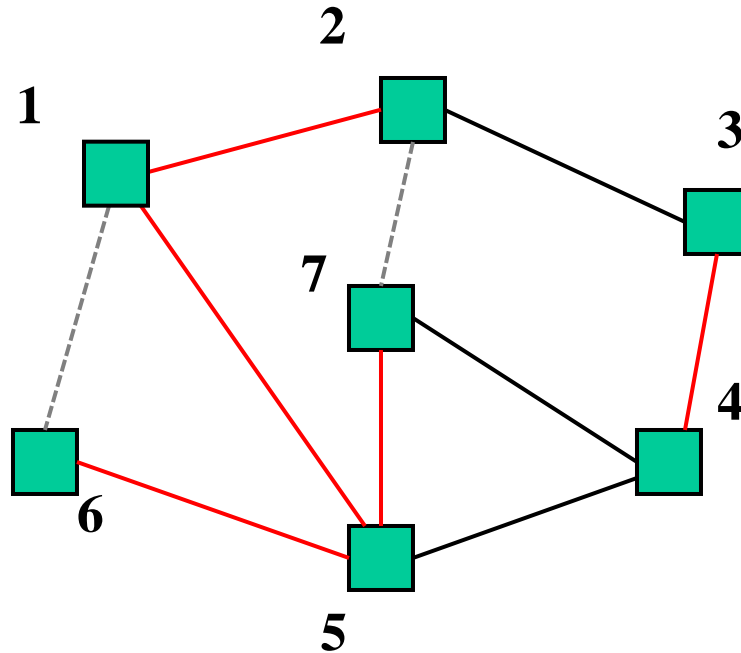
Output: (1,2), (3,4), (5,6), (5,7), (1,5)



# Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

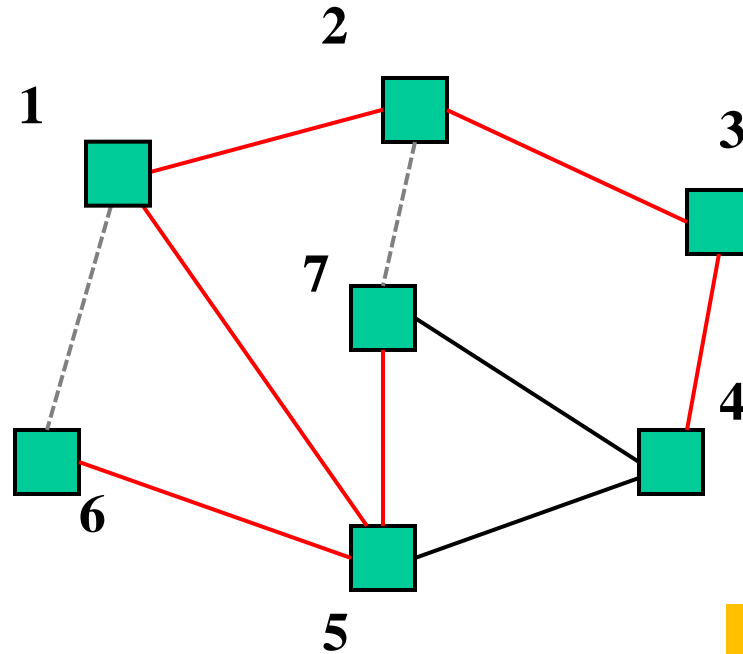


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

# Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)



Output: (1,2), (3,4), (5,6), (5,7), (1,5), (2,3)

Can stop once we have  $|V|-1$  edges

# *Cycle Detection*

- To decide if an edge could form a cycle is  $O(|V|)$  because we may need to traverse all edges already in the output
- So overall algorithm would be  $O(|V||E|)$
- But there is a faster way using the [disjoint-set ADT](#)
  - Initially, each item is in its own 1-element set
  - **find**( $u$ ): what set contains  $u$ ?
  - **union**( $u, v$ ): union (combine) the sets containing  $u$  and  $v$

## *Aside: Union-Find aka Disjoint Set ADT*

- **Union(x,y)** – take the union of two sets named x and y
  - Given sets: {3,5,7} , {4,2,8}, {9}, {1,6}
  - **Union(5,1)**  
Result: {3,5,7,1,6}, {4,2,8}, {9},

To perform the union operation, we replace sets x and y by  $(x \cup y)$
- **Find(x)** – return the name of the set containing x.
  - Given sets: {3,5,7,1,6}, {4,2,8}, {9},
  - **Find(1)** returns 5
  - **Find(4)** returns 8
- We can do Union in constant time.
- We can get Find to be ***amortized*** constant time (worst case  $O(\log n)$  for an individual Find operation).

# *Using Disjoint-Set*

Can use a disjoint-set implementation  
in our spanning-tree algorithm to detect cycles:

Invariant:  $u$  and  $v$  are connected in output-so-far  
iff  
 $u$  and  $v$  in the same set

- Initially, each node is in its own set
- When processing edge  $(u, v)$ :
  - If  $\mathbf{find}(u) == \mathbf{find}(v)$ , then do not add the edge
  - Else add the edge and  $\mathbf{union}(u, v)$

# *Summary so Far*

## The **spanning-tree problem**

- Add nodes to partial tree approach is  $O(|E|)$
- Add acyclic edges approach is  $O(|E| \log |V|)$ 
  - Using the disjoint-set ADT “as a black box”

## But really want to solve the **minimum-spanning-tree problem**

- Given a weighted undirected graph, give a spanning tree of minimum weight
- Same two approaches will work with minor modifications
- Both will be  $O(|E| \log |V|)$

# *Getting to the Point*

## Algorithm #1

Shortest-path is to Dijkstra's Algorithm

as

Minimum Spanning Tree is to [Prim's Algorithm](#)

(Both based on expanding cloud of known vertices,  
basically using a priority queue instead of a DFS stack)

## Algorithm #2

[Kruskal's Algorithm](#) for Minimum Spanning Tree

is

Exactly our forest-merging approach to spanning tree

but process edges in cost order

# *Prim's Algorithm Idea*

Idea: Grow a tree by adding an edge from the “known” vertices to the “unknown” vertices. *Pick the edge with the smallest weight that connects “known” to “unknown.”*

Recall Dijkstra picked “edge with closest known distance to source.”

- But that is not what we want here
- Otherwise identical
- Feel free to look back and compare

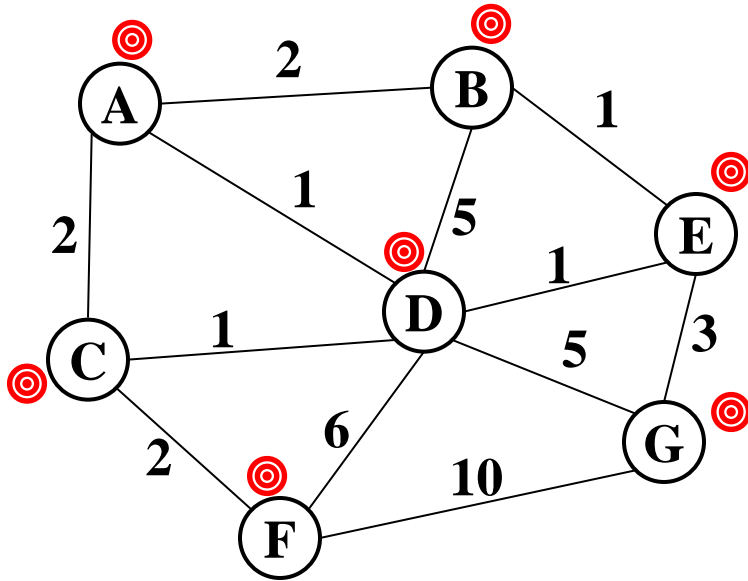


# *The Algorithm*

1. For each node  $v$ , set  $v.cost = \infty$  and  $v.known = false$
2. Choose any node  $v$ .
  - a) Mark  $v$  as known
  - b) For each edge  $(v, u)$  with weight  $w$ ,  
set  $u.cost = w$  and  $u.prev = v$
3. While there are unknown nodes in the graph
  - a) Select the unknown node  $v$  with lowest cost
  - b) Mark  $v$  as known and add  $(v, v.prev)$  to output
  - c) For each edge  $(v, u)$  with weight  $w$ ,  

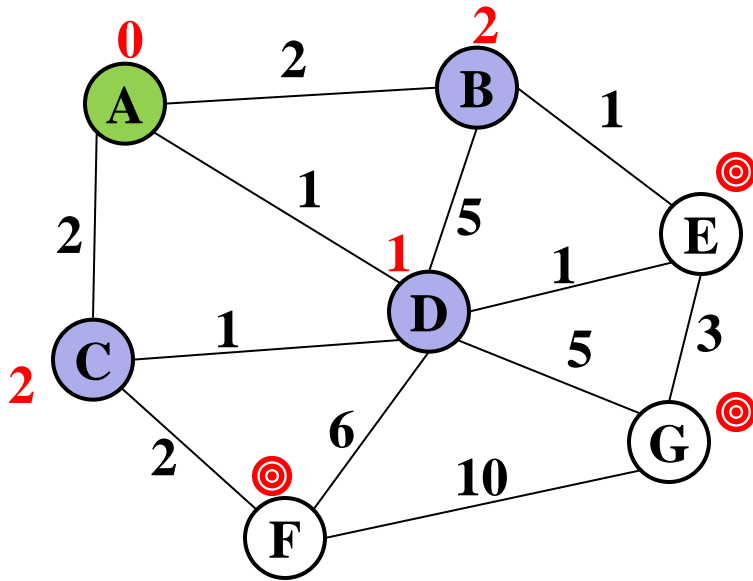
```
        if(w < u.cost) {  
            u.cost = w;  
            u.prev = v;  
        }
```

# Example



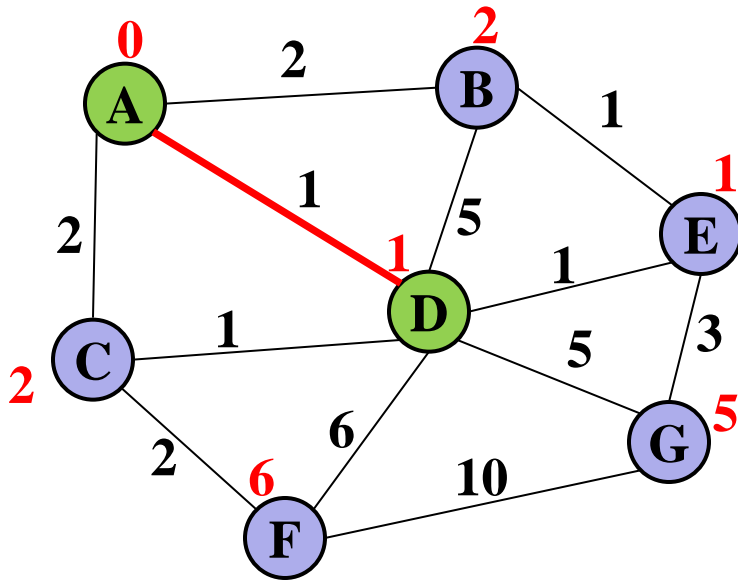
vertex	known?	cost	prev
A		??	
B		??	
C		??	
D		??	
E		??	
F		??	
G		??	

# Example



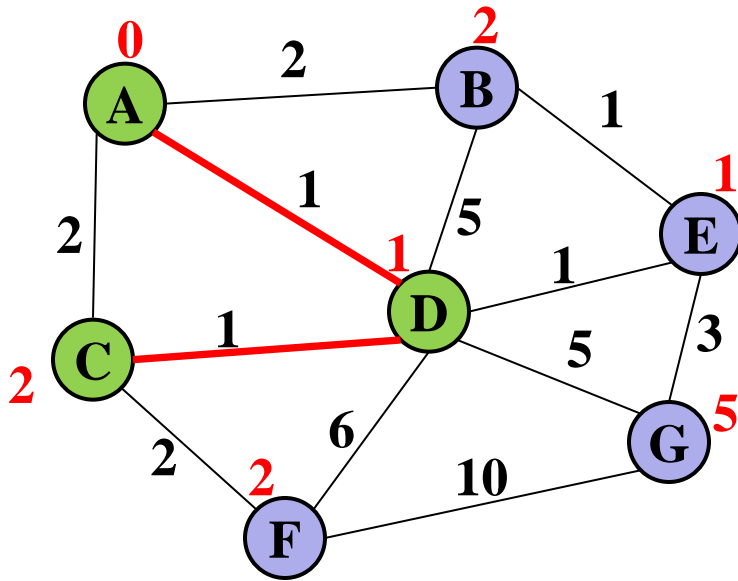
vertex	known?	cost	prev
A	Y	0	
B		2	A
C		2	A
D		1	A
E		??	
F		??	
G		??	

# Example



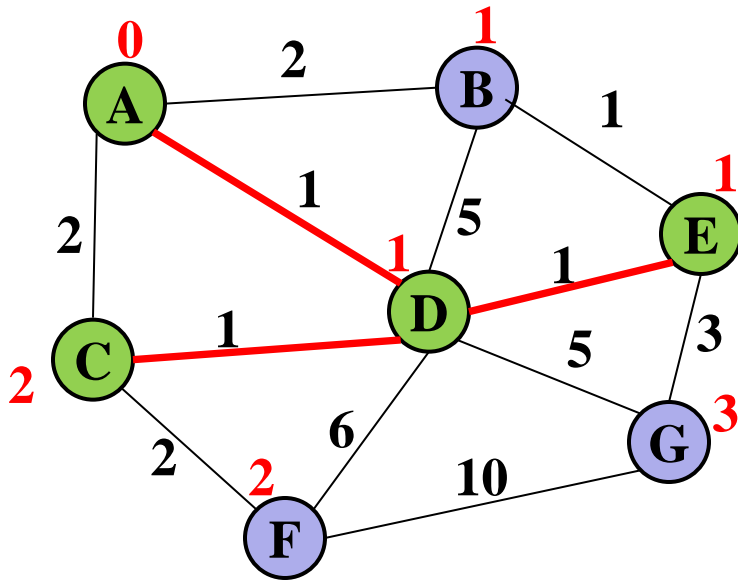
vertex	known?	cost	prev
A	Y	0	
B		2	A
C		1	D
D	Y	1	A
E		1	D
F		6	D
G		5	D

# Example



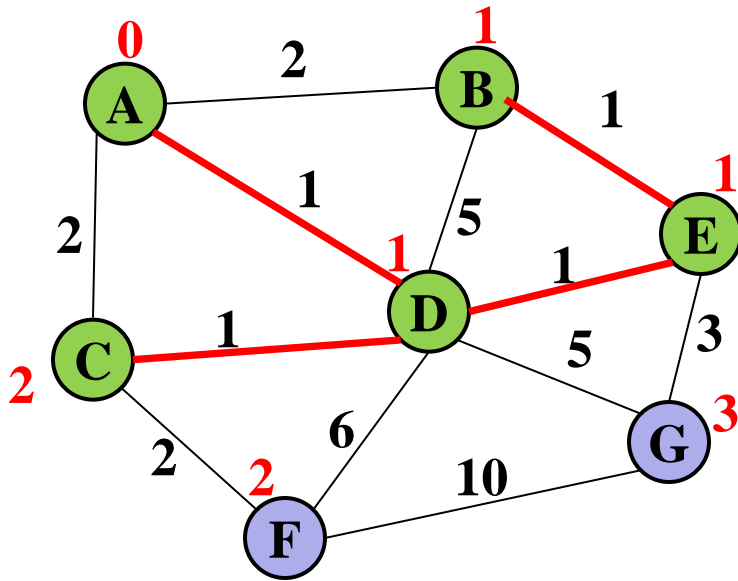
vertex	known?	cost	prev
A	Y	0	
B		2	A
C	Y	1	D
D	Y	1	A
E		1	D
F		2	C
G		5	D

# Example



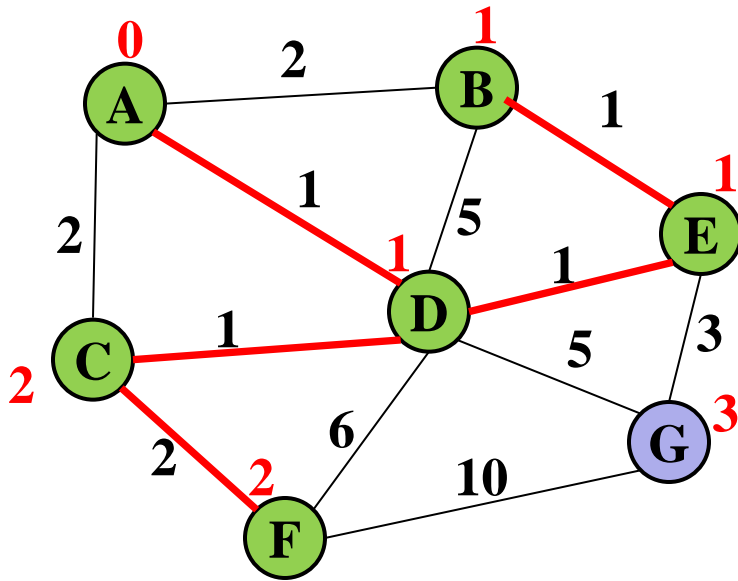
vertex	known?	cost	prev
A	Y	0	
B		1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E

# Example



vertex	known?	cost	prev
A	Y	0	
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E

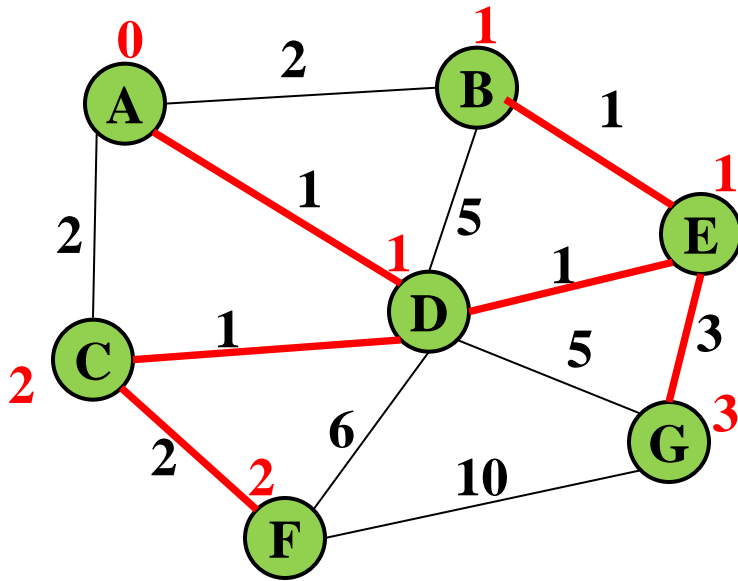
# Example



vertex	known?	cost	prev
A	Y	0	
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F	Y	2	C
G		3	E



# Example



vertex	known?	cost	prev
A	Y	0	
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F	Y	2	C
G	Y	3	E

# *Analysis*

- Correctness
  - Intuitively similar to Dijkstra
- Run-time
  - Same as Dijkstra
  - $O(|E| \log |V|)$  using a priority queue

# Kruskal's Algorithm

Idea: Grow a forest out of edges that do not grow a cycle, just like for the spanning tree problem.

- But now consider the edges in order by weight

So:

- Sort edges:  $O(|E| \log |E|) = O(|E| \log |V|)$
- Iterate through edges using union-find for cycle detection  $O(|E| \log |V|)$

Somewhat better:

- Floyd's algorithm to build min-heap with edges  $O(|E|)$
- Iterate through edges using union-find for cycle detection and **deleteMin** to get next edge  $O(|E| \log |V|)$
- Not better worst-case asymptotically, but often stop long before considering all edges

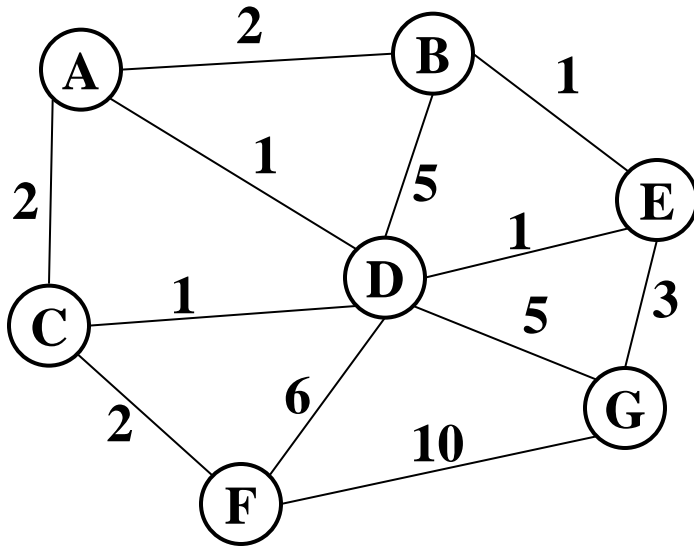
# *Pseudocode*

1. Sort edges by weight (better: put in min-heap)
2. Each node in its own set
3. While output size  $< |V|-1$ 
  - Consider next smallest edge  $(u, v)$
  - if `find(u, v)` indicates  $u$  and  $v$  are in different sets
    - output  $(u, v)$
    - `union(u, v)`

Recall invariant:

$u$  and  $v$  in same set if and only if connected in output-so-far

# Example



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

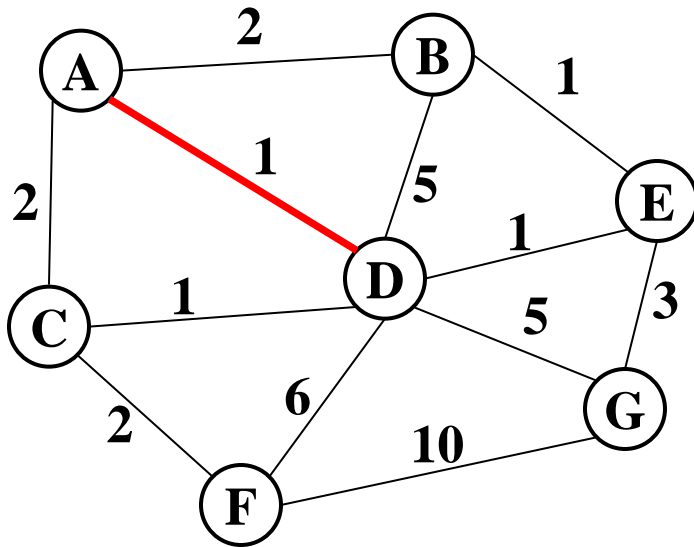
10: (F,G)

Sets: (A) (B) (C) (D) (E) (F) (G)

Output:

Note: At each step, the union/find sets are the trees in the forest

# Example



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

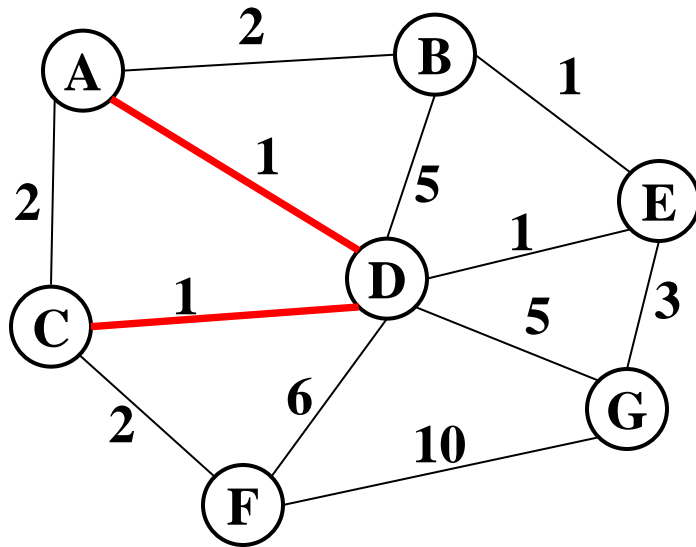
10: (F,G)

Sets: (A, D) (B) (C) (E) (F) (G)

Output: (A,D)

Note: At each step, the union/find sets are the trees in the forest

# Example



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

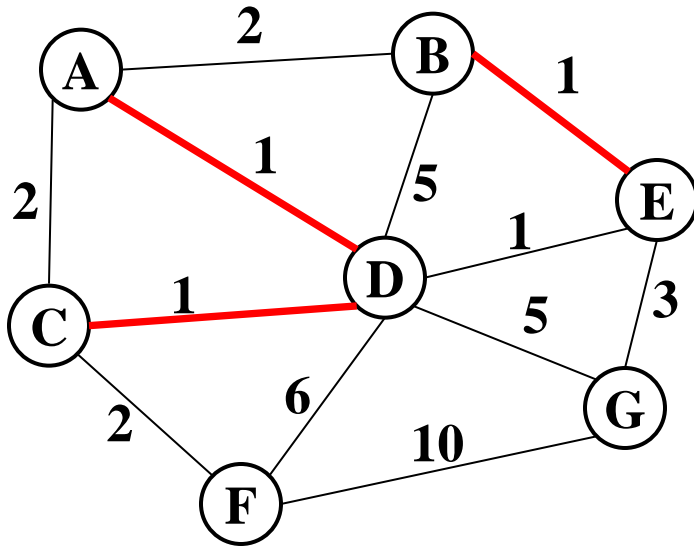
10: (F,G)

Sets: (A, C, D) (B) (E) (F) (G)

Output: (A,D), (C,D)

Note: At each step, the union/find sets are the trees in the forest

# Example



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

10: (F,G)

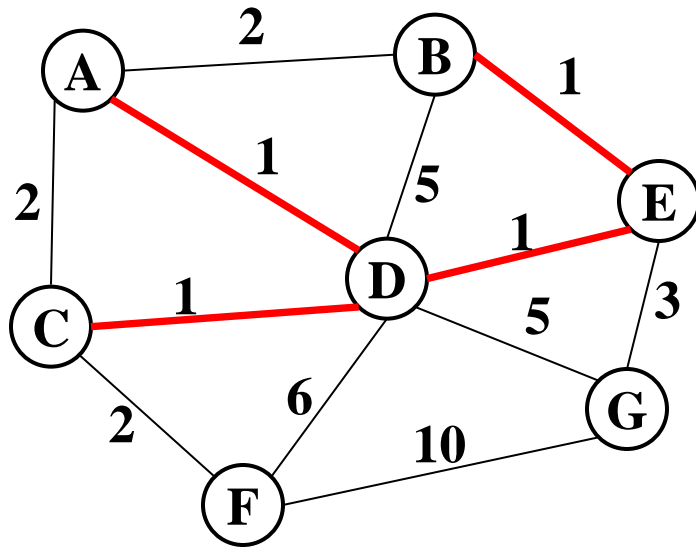
Sets: (A, C, D) (B, E) (F) (G)

Output: (A,D), (C,D), (B,E)

Note: At each step, the union/find sets are the trees in the forest



# Example



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

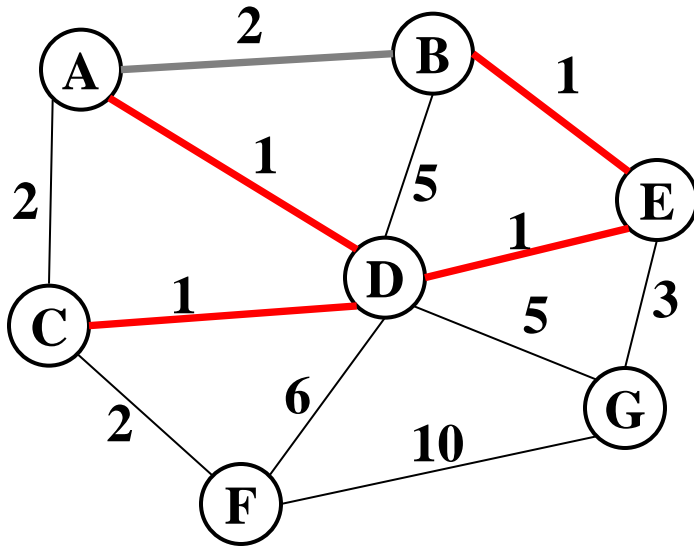
10: (F,G)

Sets: (A, B, C, D, E) (F) (G)

Output: (A,D), (C,D), (B,E), (D,E)

Note: At each step, the union/find sets are the trees in the forest

# Example



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

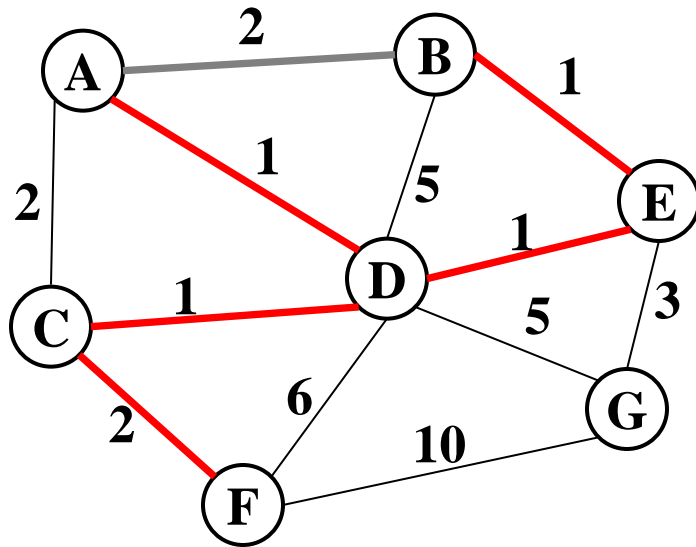
10: (F,G)

Sets: (A, B, C, D, E) (F) (G)

Output: (A,D), (C,D), (B,E), (D,E)

Note: At each step, the union/find sets are the trees in the forest

# Example



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

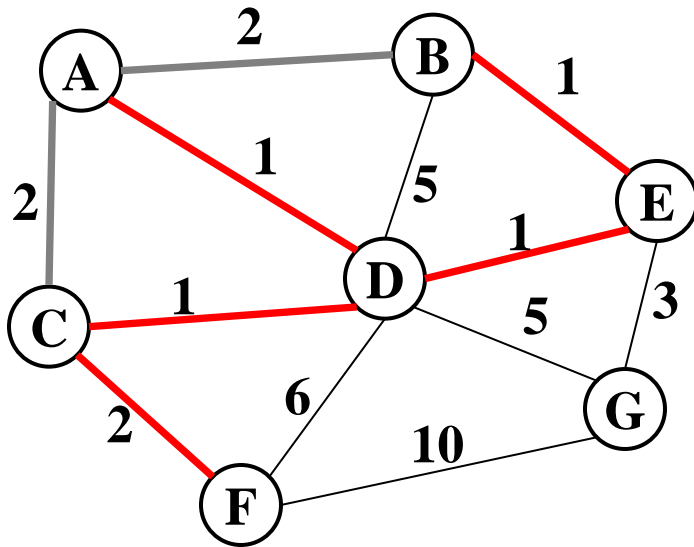
10: (F,G)

Sets: (A, B, C, D, E, F) (G)

Output: (A,D), (C,D), (B,E), (D,E), (C,F)

Note: At each step, the union/find sets are the trees in the forest

# Example



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

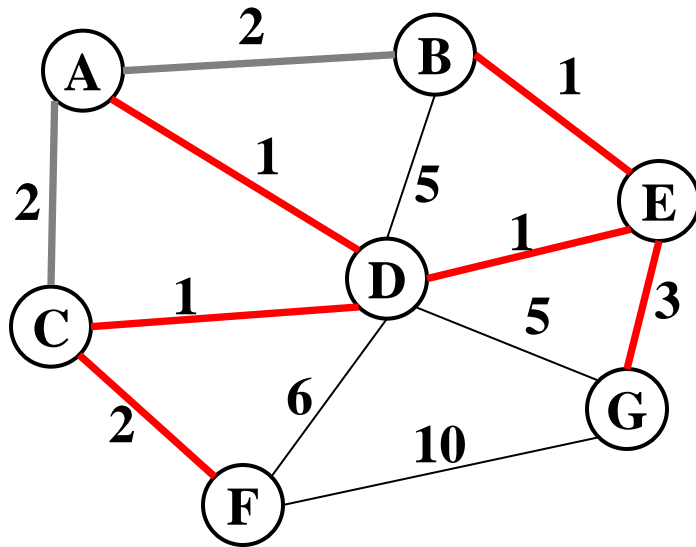
10: (F,G)

Sets: (A, B, C, D, E, F) (G)

Output: (A,D), (C,D), (B,E), (D,E), (C,F)

Note: At each step, the union/find sets are the trees in the forest

# Example



Edges in sorted order:

1: (A,D), (C,D), (B,E), (D,E)

2: (A,B), (C,F), (A,C)

3: (E,G)

5: (D,G), (B,D)

6: (D,F)

10: (F,G)

Sets: (A, B, C, D, E, F, G)

Output: (A,D), (C,D), (B,E), (D,E), (C,F), (E,G)

Note: At each step, the union/find sets are the trees in the forest

# *Analysis*

- Correctness
  - That it is a spanning tree
    - When we add an edge, it adds a vertex to the tree (or else it would have created a cycle)
    - The graph is connected, we consider all edges
  - That it is minimum
    - By induction
    - At every step, the output is a subset of a minimum tree
- Run-time
  - $O(|E| \log |V|)$