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

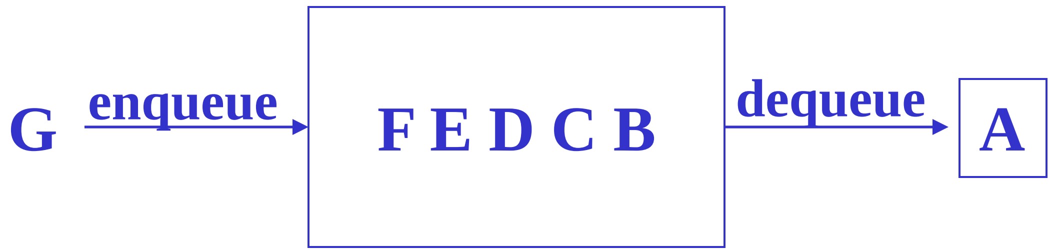
  `create`

  `destroy`

  `enqueue`

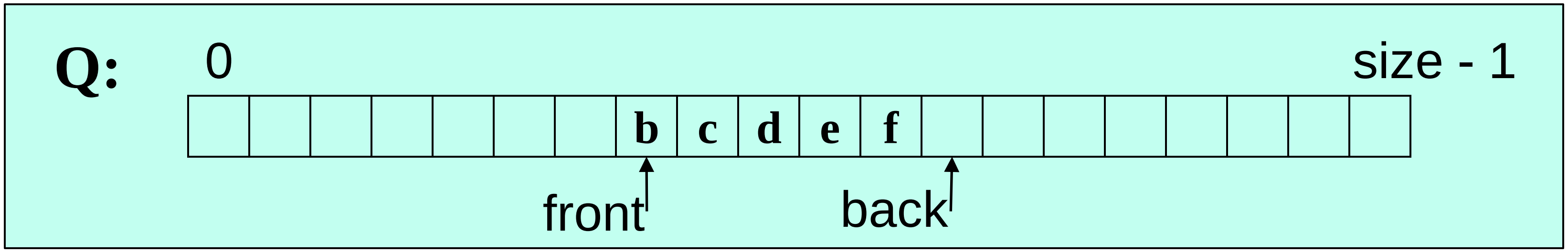
  `dequeue`

  `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

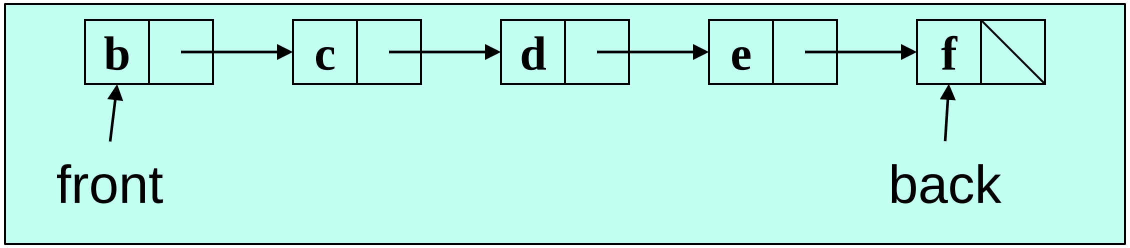



```
// Basic idea only!
enqueue(x) {
  Q[back] = x;
  back = (back + 1) % size
}
```

```
// 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^{th} element in the queue?

Linked List Queue Data Structure

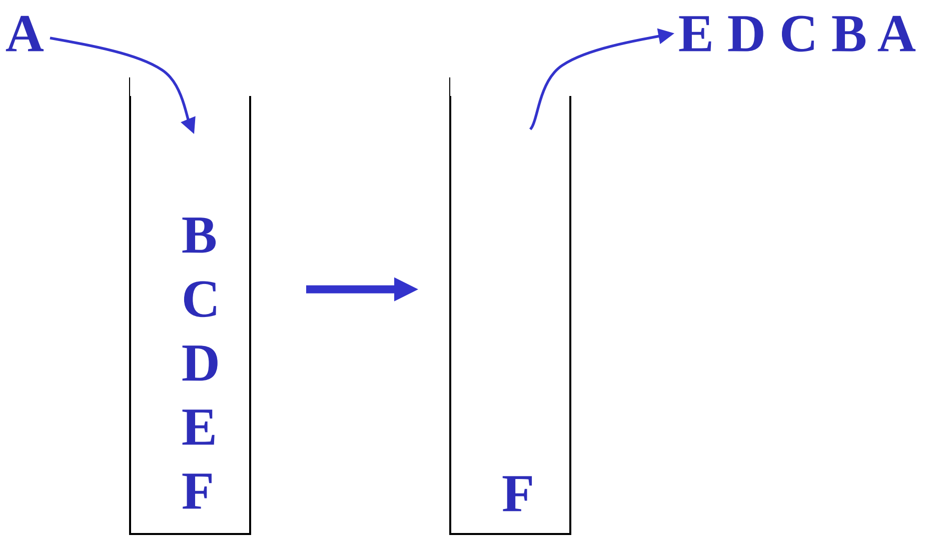



```
// Basic idea only!
enqueue(x) {
  back.next = new Node(x);
  back = back.next;
}
```

```
// 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^{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^{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^{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* ≥ 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:

$$1 + 2 + 4 + ... + 2^{k-1} = 2^k - 1$$

Add the next power of 2 to both sides…

$$1 + 2 + 4 + ... + 2^{k-1} + 2^k = 2^k - 1 + 2^k$$

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

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

Another Example

For all $n \geq 1$

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

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:

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

Add k+1 to both sides…

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

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

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

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…

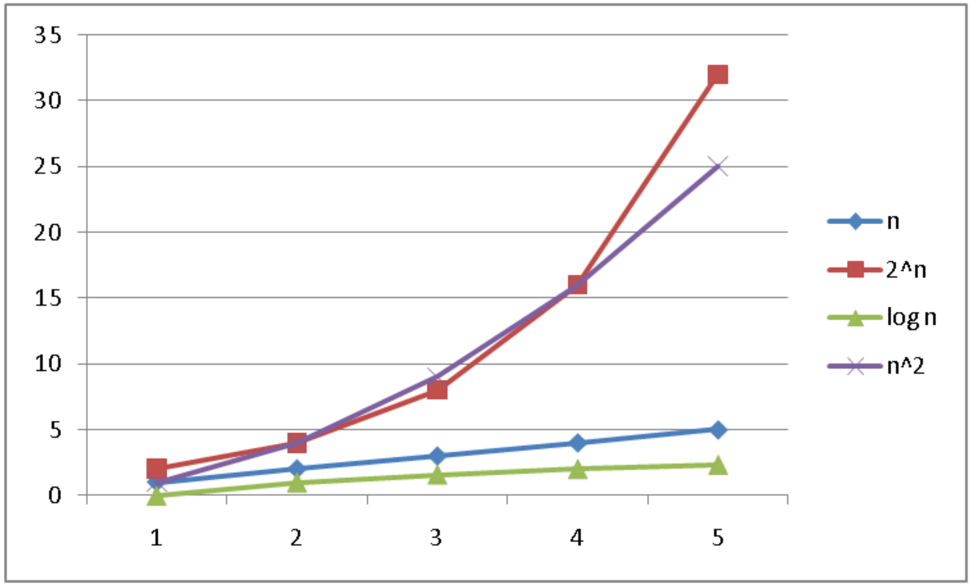
- Every person in this room with 7 bits
- Every person in the U.S. with 29 bits
- Every person in the world with 33 bits
- Every person to have ever lived with 38 bits (estimate)
- Every atom in the universe with 250-300 bits

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

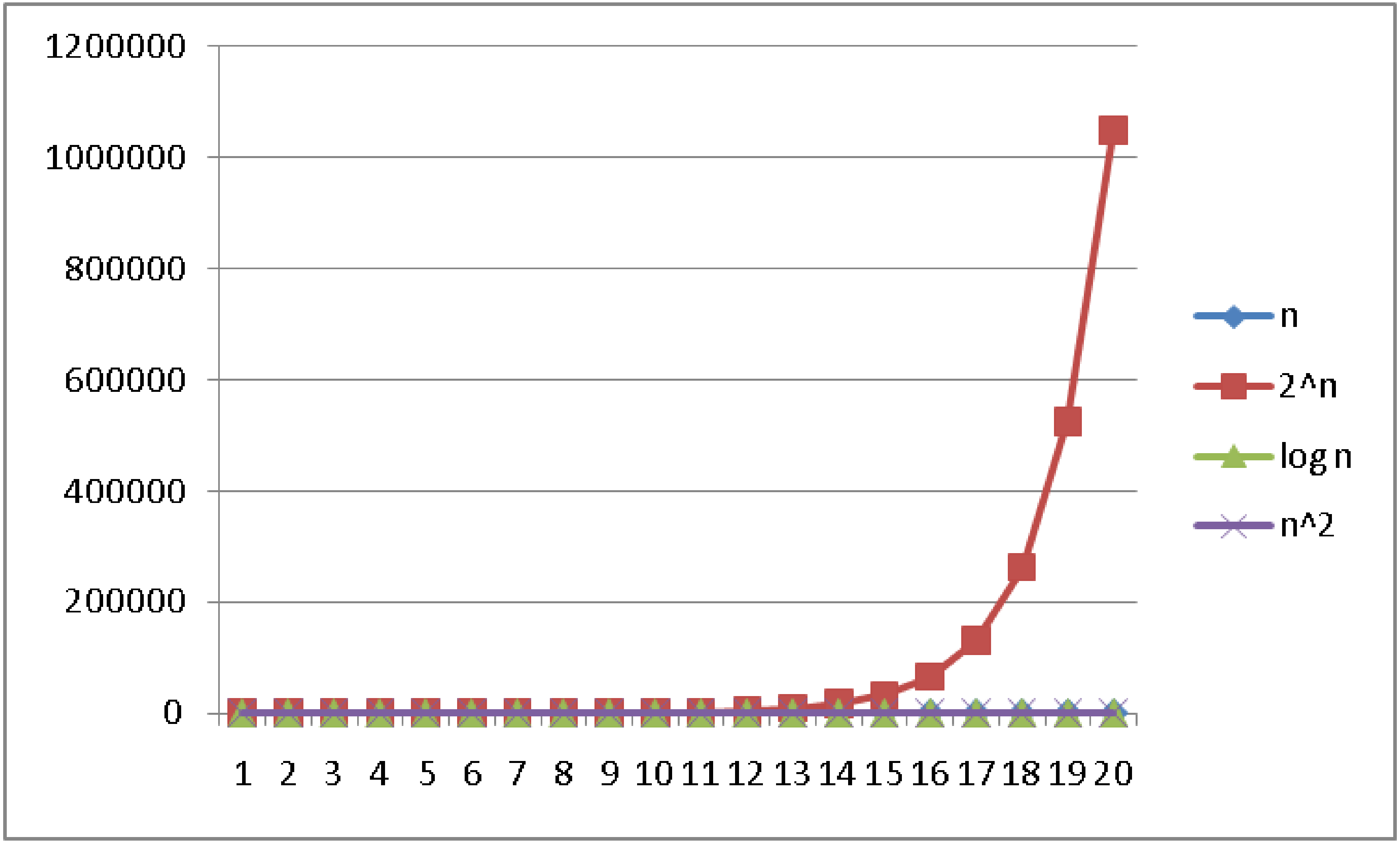
Logarithms and Exponents

- Since so much is binary in CS, **`log`** almost always means $\log_2$
- Definition: $\log_2 x = y$ iff $x = 2^y$
- So, $\log_2$ 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: $\log_2 \log_2 4billion \sim \log_2 \log_2 2^{32} = \log_2 32 = 5$
- **(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.
  - $1+2+3+\ldots+n = n(n+1)/2$
  - x gets raised by 3 each time

Example

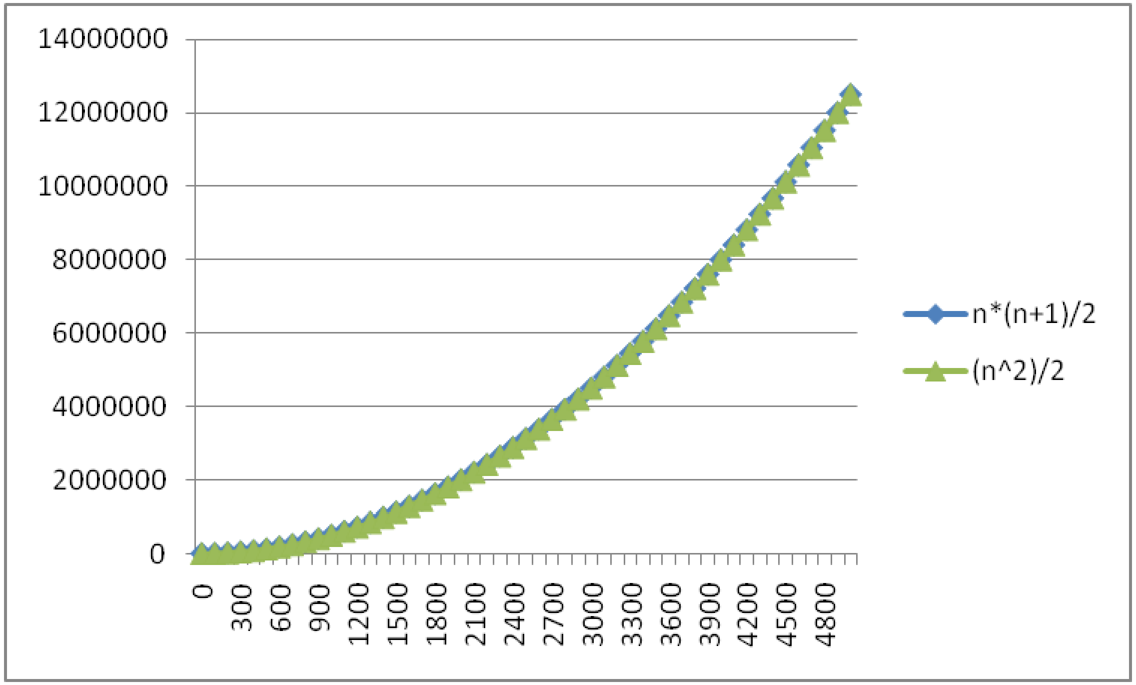
- How long does this pseudocode run?

```
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 ≥ 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*(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 ≤ 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)
$O(\texttt{log}\ n)$	logarithmic
$O(n)$	linear
$O(n\ \texttt{log}\ n)$	"n `log` *n*"
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^k)$	polynomial (where is k is an 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

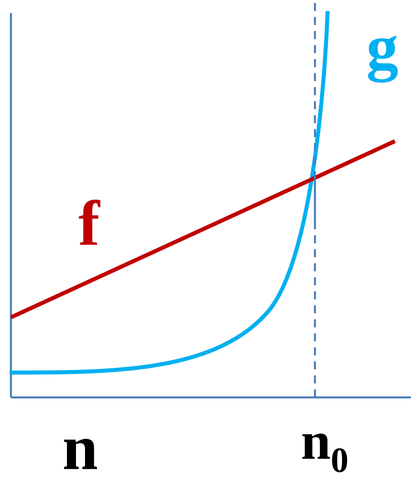
Big-Oh

- 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

  $f(n) \le c\, g(n)$, for all $n \ge n_0$.

- 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) ≤ 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.

   α^n is $O(n^\beta)$ — False

For some finite β

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

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

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 <u>always</u> 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)
$O(\texttt{log}\ n)$	logarithmic
$O(n)$	linear
$O(n\ \texttt{log}\ n)$	"n $\texttt{log}\ n$"
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^k)$	polynomial (where is k is an 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 loglogn 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(nlogn) (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(nlogn) 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(logn) – 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 *recurrence equation*

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(logn)
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

    $g(n) \leq c\, f(n)$ for all $n \geq n_0$

- Big Omega: Lower bound: Ω(f(n)) is the set of all functions asymptotically greater than or equal to f(n): '$\geq$' of functions
  - g(n) is in Ω(f(n)) if there exist constants c and n_0 such that

    $g(n) \geq c\, f(n)$ for all $n \geq n_0$

- Big Theta: Tight bound: θ(f(n)) is the set of all functions asymptotically equal to f(n): '=' of functions
  - Intersection of O(f(n)) and Ω(f(n)) (use *different constants*)

Regarding use of terms

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

- 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 $\omega(\texttt{log}\ n)$ 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 $k + k + \ldots + k$ 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: $T(n) = O(1) + T(n\text{-}1)$

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:

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

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

$=2k+5$, where k 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*.
   – $T(n) = 10 + 10 + T(n/4)$

     $= 10 + 10 + 10 + T(n/8)$

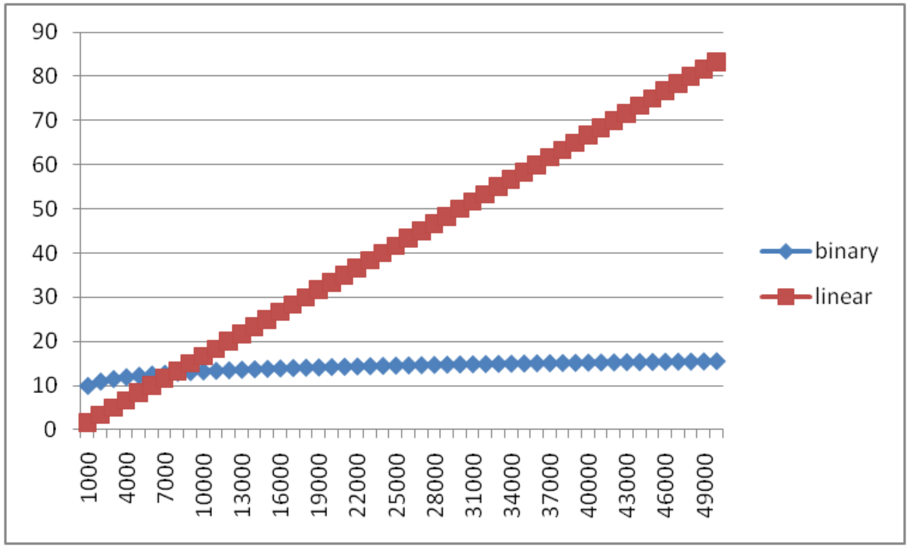
     $= \ldots$

     $= 10k + T(n/(2^k))$ 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 $k = \log_2 n$
   – So $T(n) = 10 \log_2 n + 8$ (get to base case and do it)
   – So $T(n)$ is $O(\log n)$

Linear vs Binary Search

- So binary search is $O(\texttt{log}\ n)$ 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 ∞ 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:

$T(n) = O(1) + T(n\text{-}1)$	linear
$T(n) = O(1) + 2T(n/2)$	linear
$T(n) = O(1) + T(n/2)$	logarithmic
$T(n) = O(1) + 2T(n\text{-}1)$	exponential
$T(n) = O(n) + T(n\text{-}1)$	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

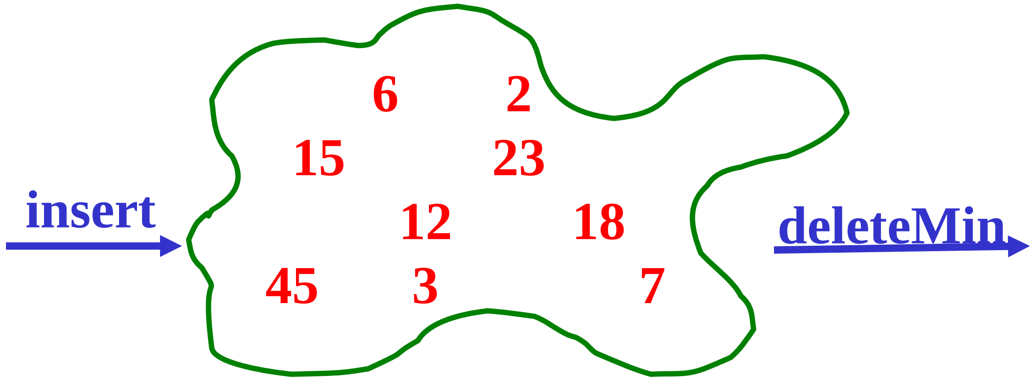
- Integers are comparable, so will use them in examples

- 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 *e1*, …, *en* at times *t+t1*, …, *t+tn*
  - 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"

data	*insert algorithm / time*		*deleteMin algorithm / time*	
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 $O(\texttt{log}\ n)$ **`insert`** and $O(\texttt{log}\ n)$ **`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):**

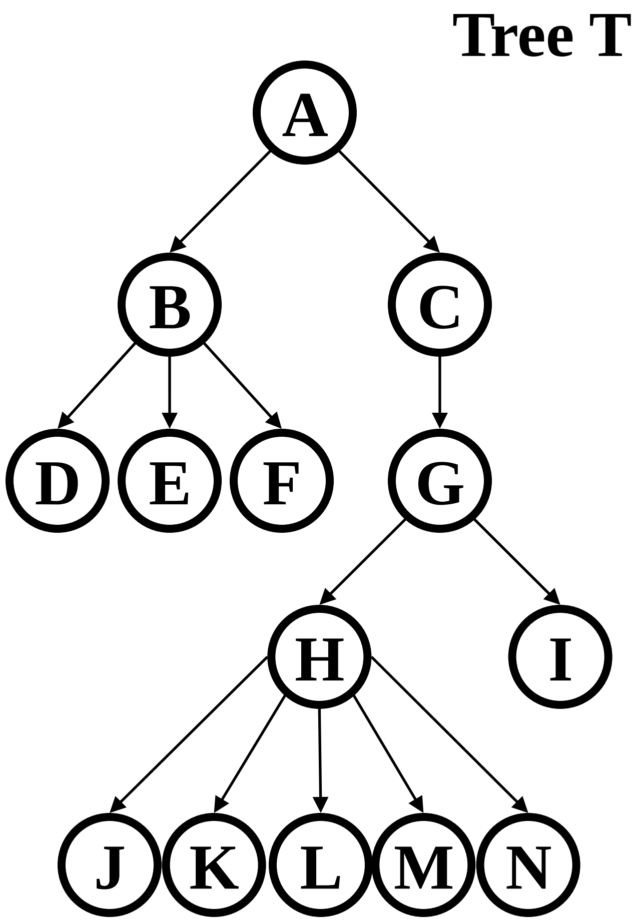
***parent*(H):**

***siblings*(E):**

***ancestors*(F):**

***descendents*(G):**

***subtree*(C):**



Tree Terminology

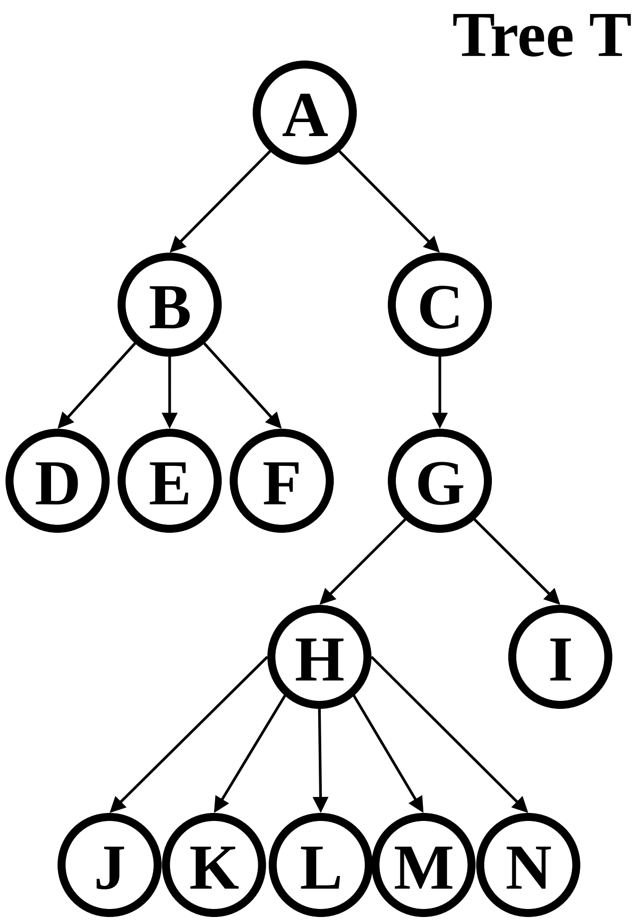
***depth*(B):**

***height*(G):**

***height*(T):**

***degree*(B):**

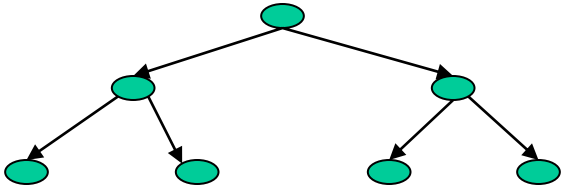
***branching factor*(T):**

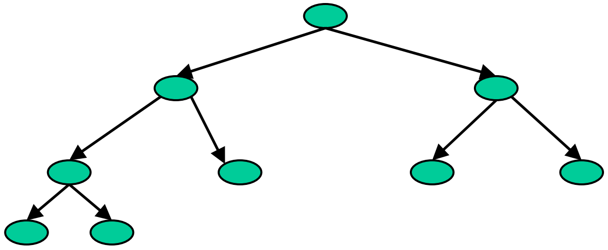


Types of Trees

Certain terms define trees with specific structures

- Binary tree: Every node has at most 2 children
- *n*-ary tree: Every node as 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

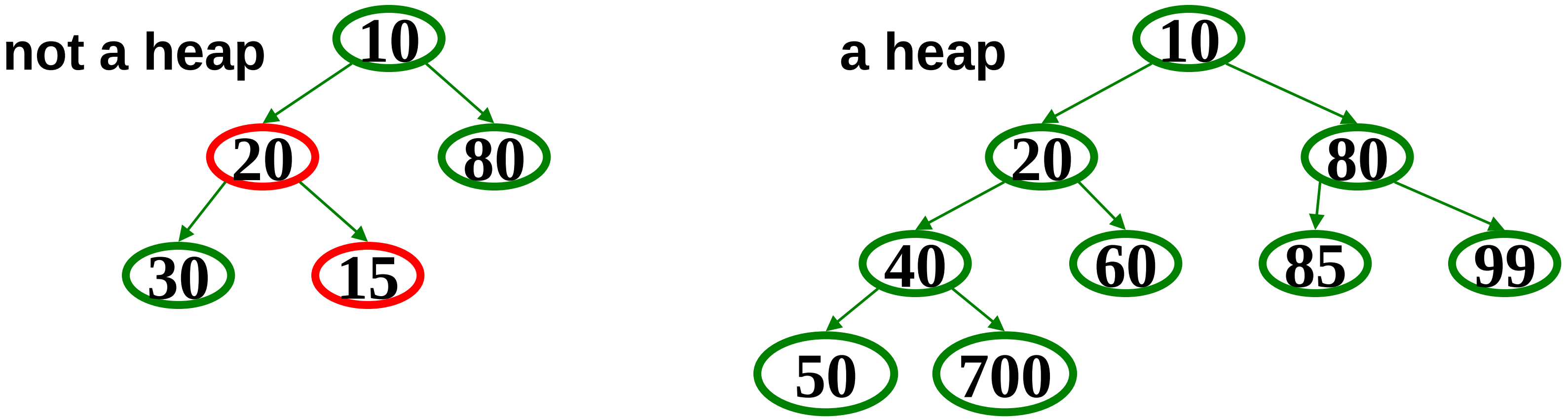
More commonly known as a binary heap or simply a heap

- 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



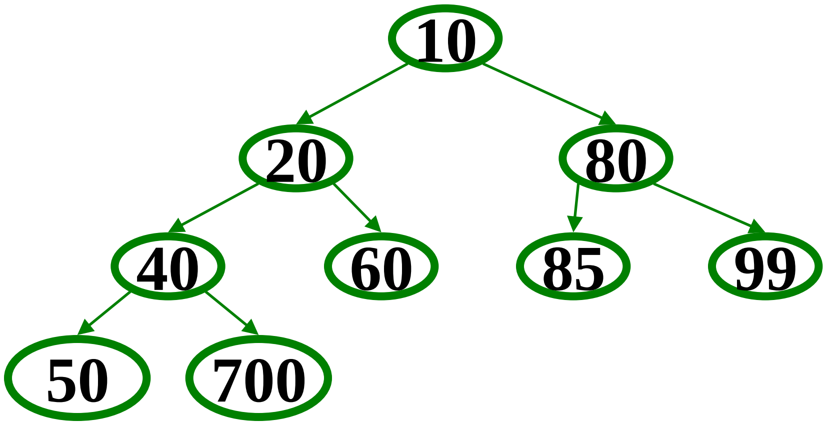
Where is the minimum priority item?

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`:

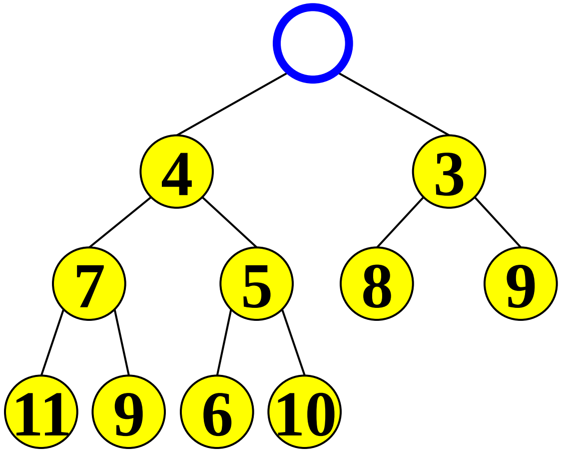
- Add node after last position
- Violate heap property, “Percolate Up” to restore

Overall, the strategy is:

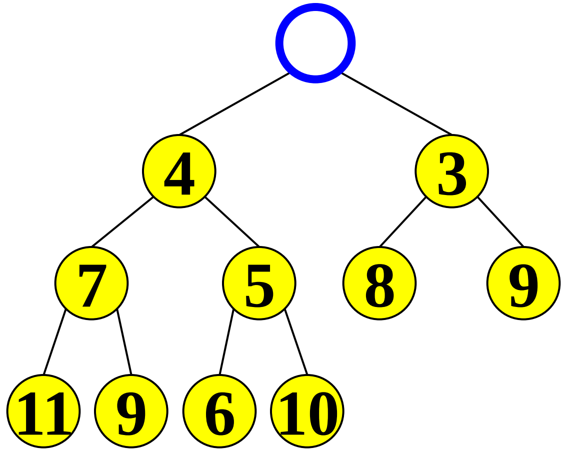
- Preserve structure property
- Break and restore heap property

DeleteMin Implementation

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



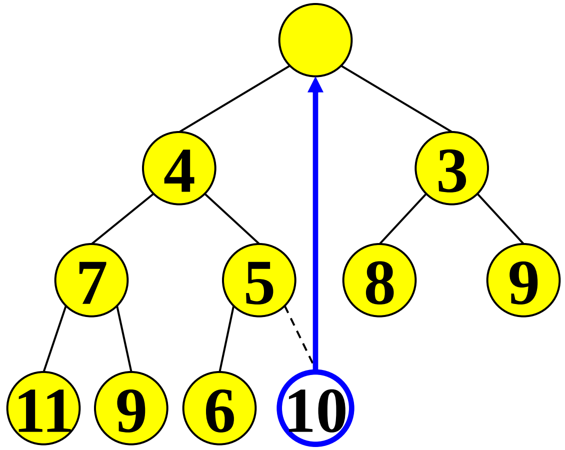
Restoring the Structure Property



2. We now have a "hole" at the root

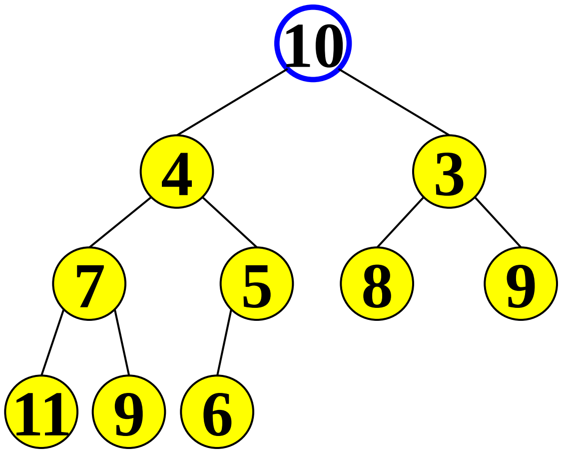
3. We must "fill" the hole with another value, must have a tree with one less node, and it must still be a complete tree

4. The "last" node is the is obvious choice



Restoring the Heap Property

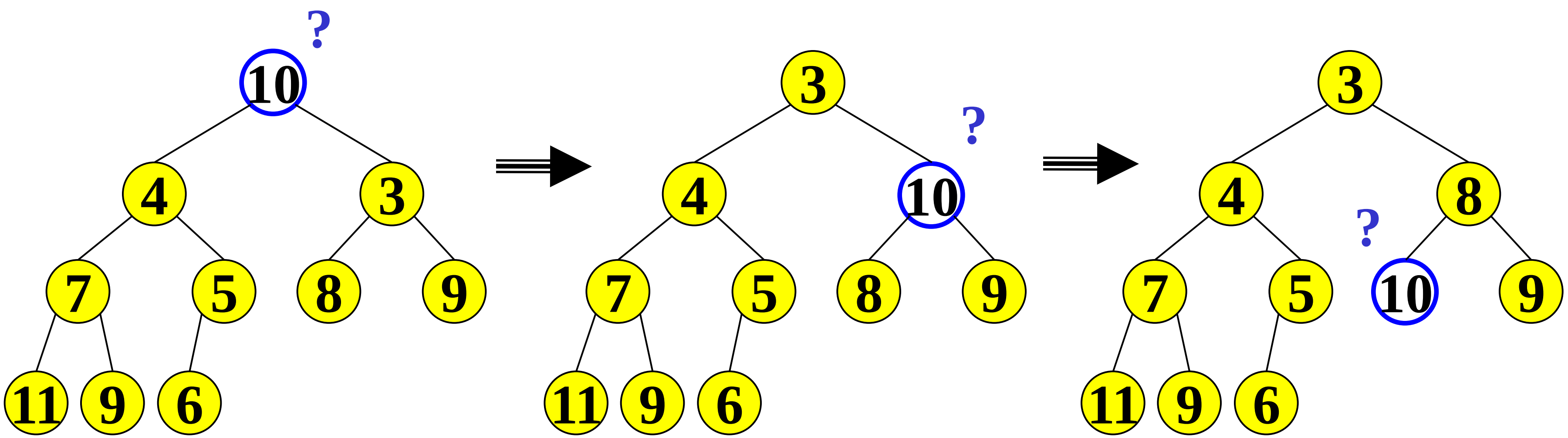
5. Not a heap, it violates the heap property



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
```

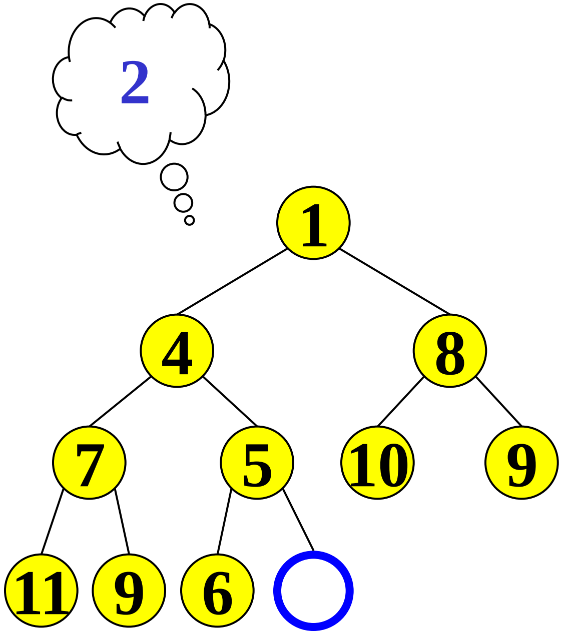
What is the runtime?
O(log n)

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

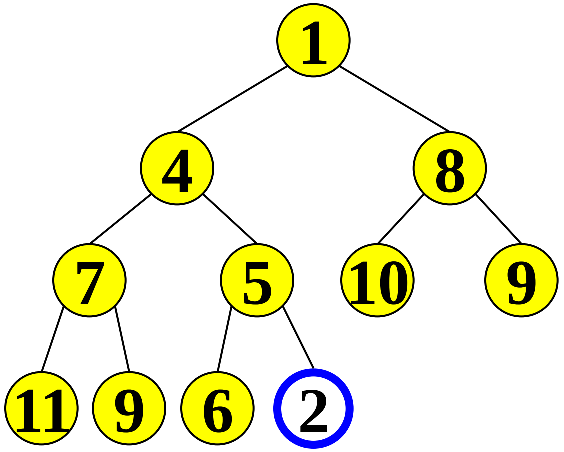
2. Put our new data there



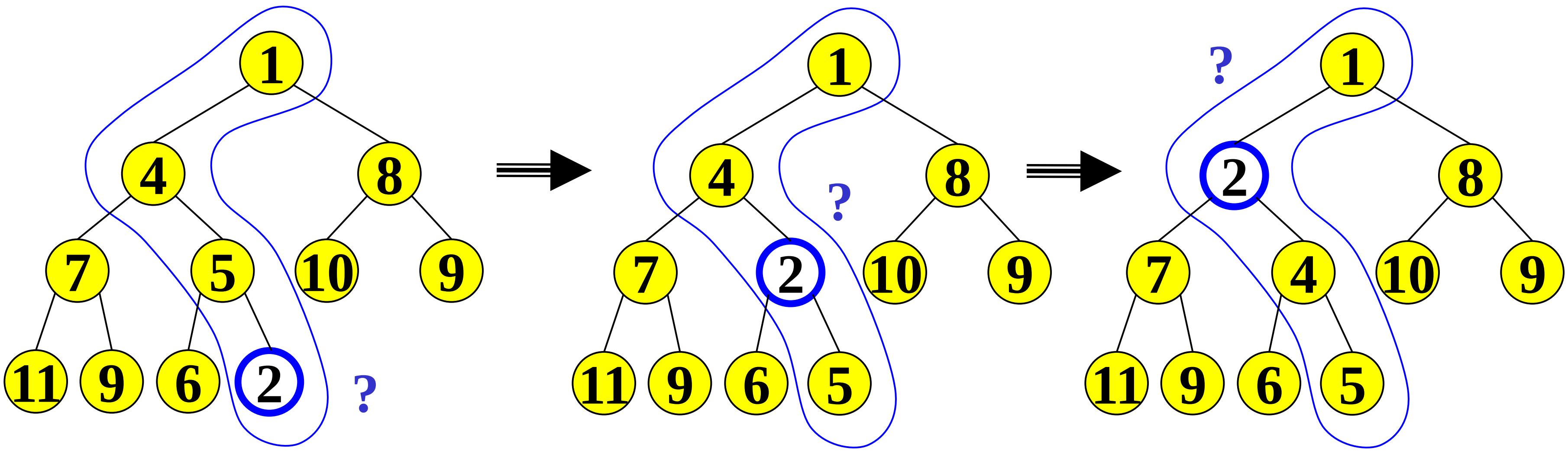
Restoring the Heap Property

3. Then percolate up to fix heap property

```
While less than parent
   Swap with parent
```



Percolate Up




```
While less than parent
   Swap with parent
```

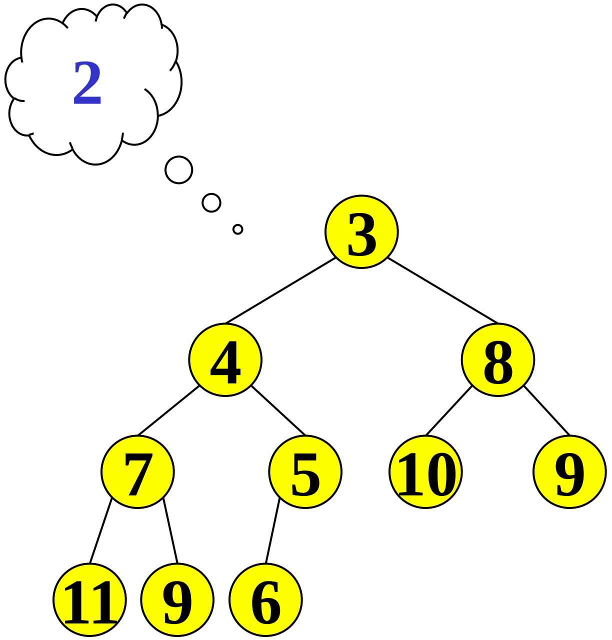
What is the runtime?
O(log n)

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

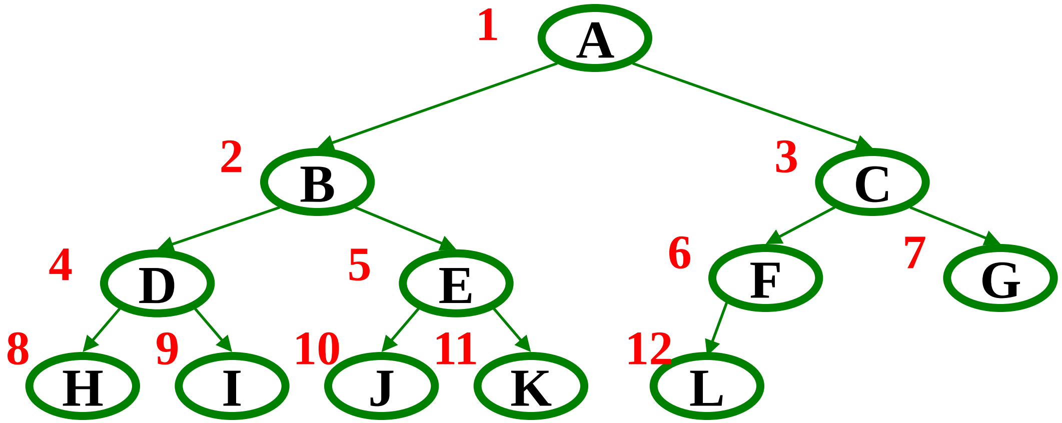
- We have seen worst-case O(log n) insert and deleteMin
  - But we promised average-case O(1) insert

- 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

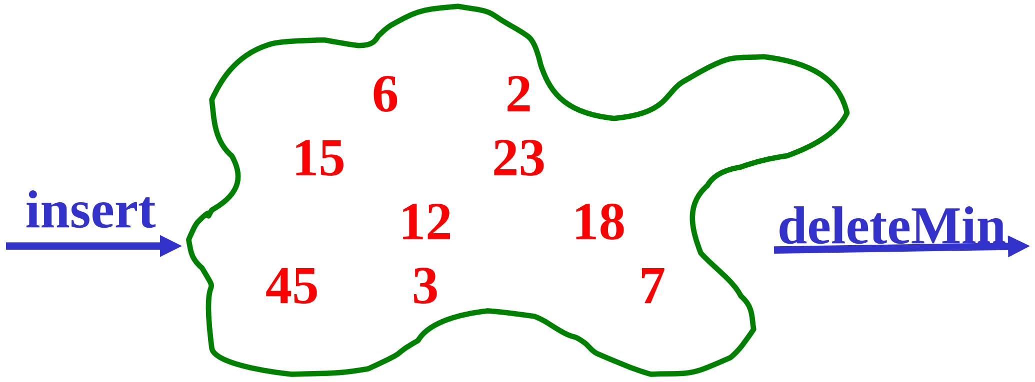
James Fogarty

Winter 2012

ADT: Priority Queue

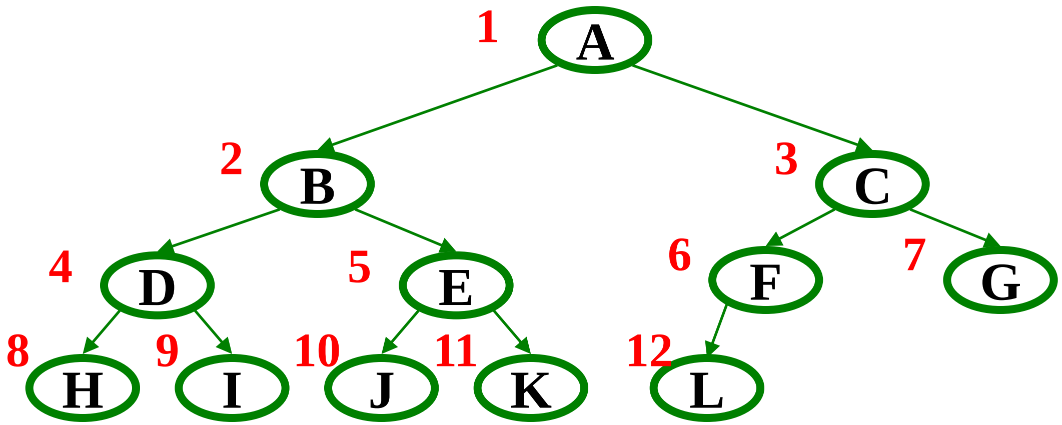
- Each item has a "priority"
  - The *next* or *best* item is the one with the *lowest* priority
  - So "priority 1" should come before "priority 4"
  - Simply by convention, could also do maximum priority

- 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:

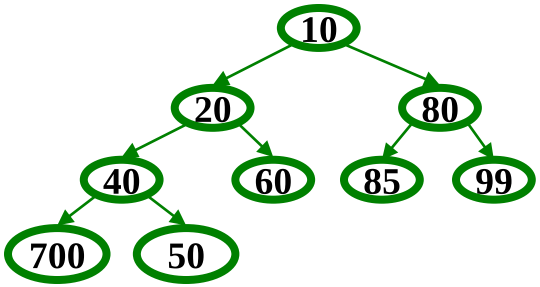
	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Pseudocode: insert

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

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

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



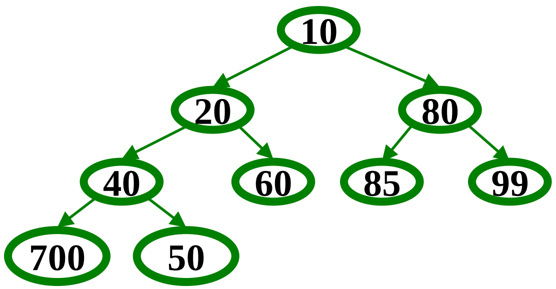
	10	20	80	40	60	85	99	700	50				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

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;
}
```

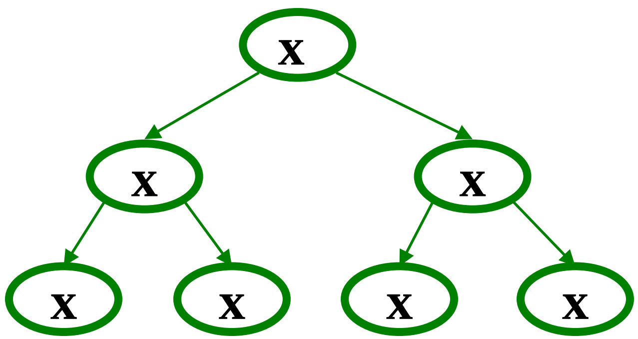


	10	20	80	40	60	85	99	700	50				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Example

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

0	1	2	3	4	5	6	7



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* **`insert`**s, *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,…,*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

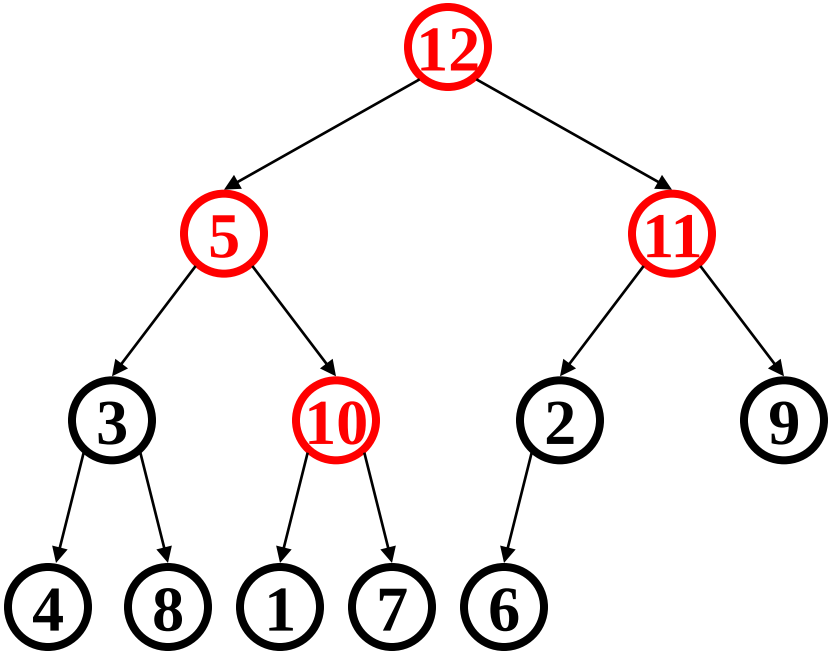
Bottom-up

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

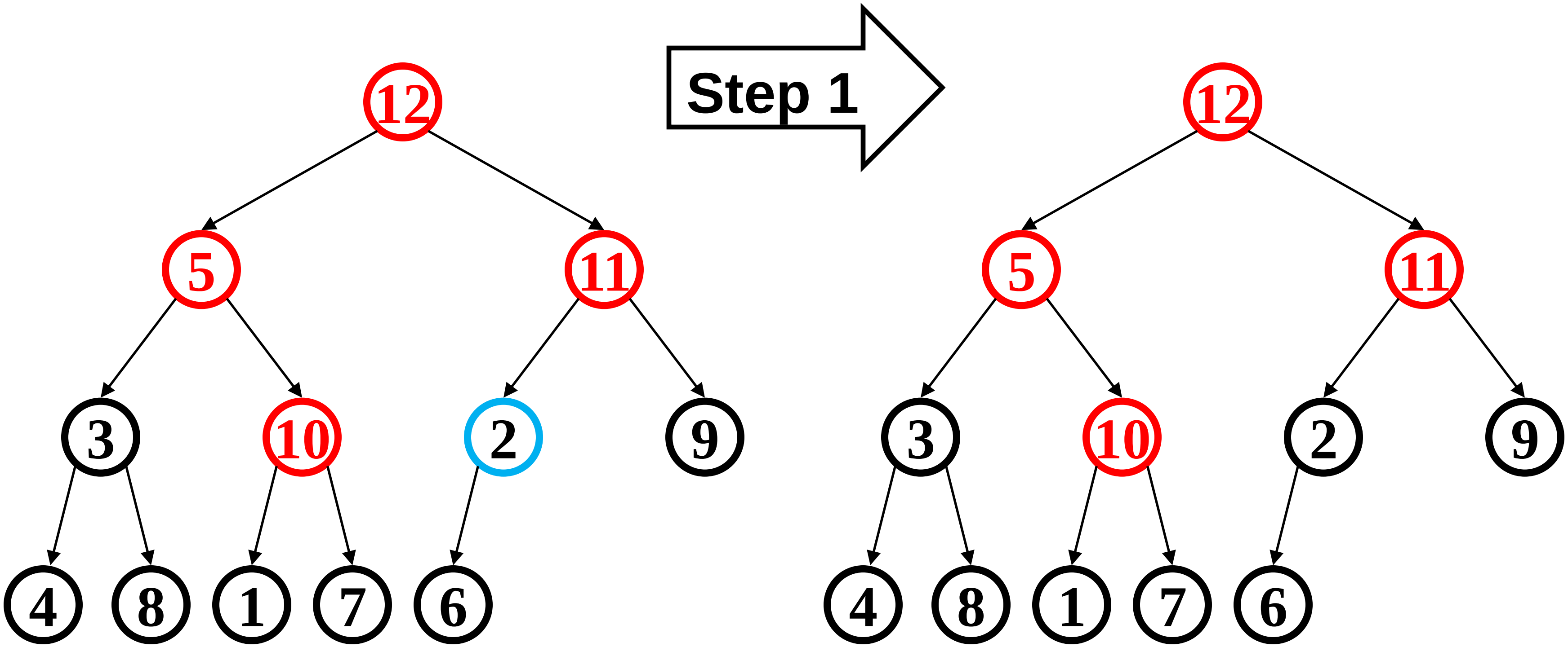
```
void buildHeap() {
   for(i = size/2; i>0; i--) {
     val  = arr[i];
     hole = percolateDown(i,val);
     arr[hole] = val;
   }
}
```

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)

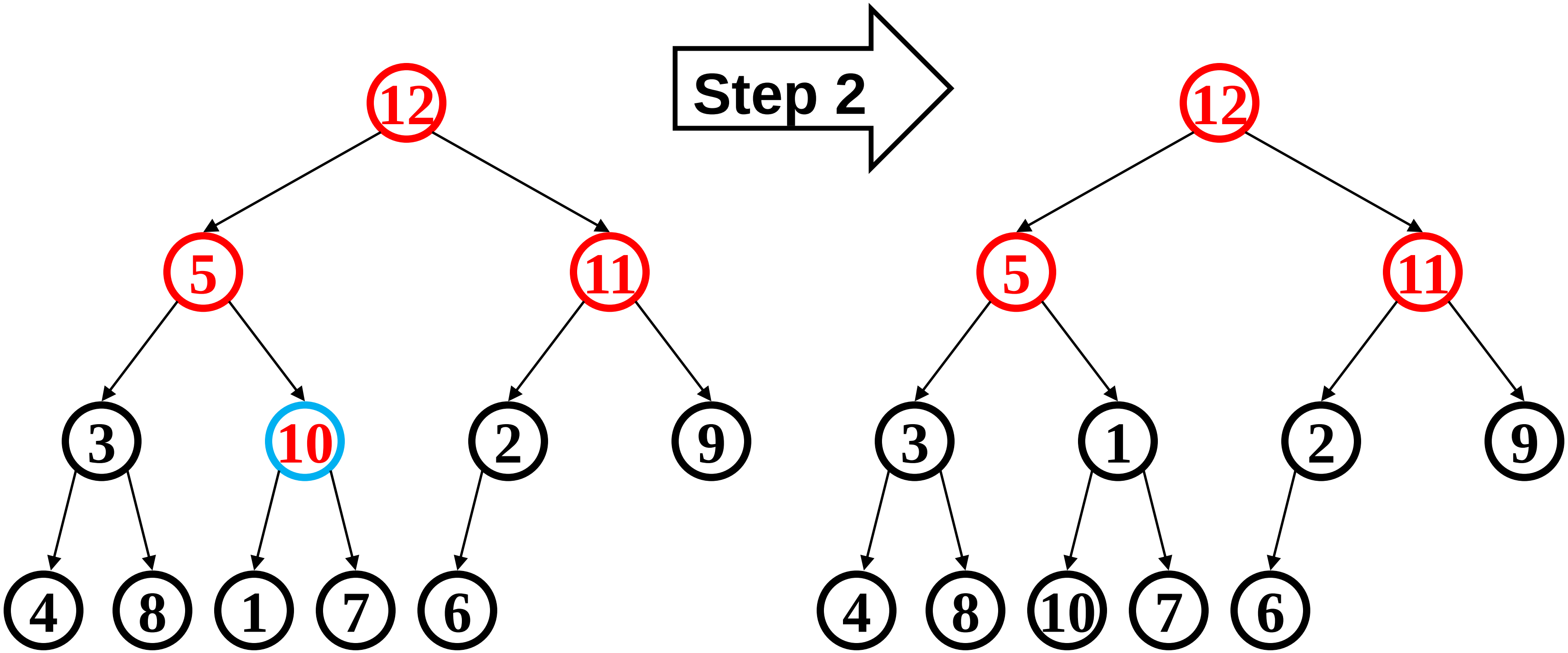


Example



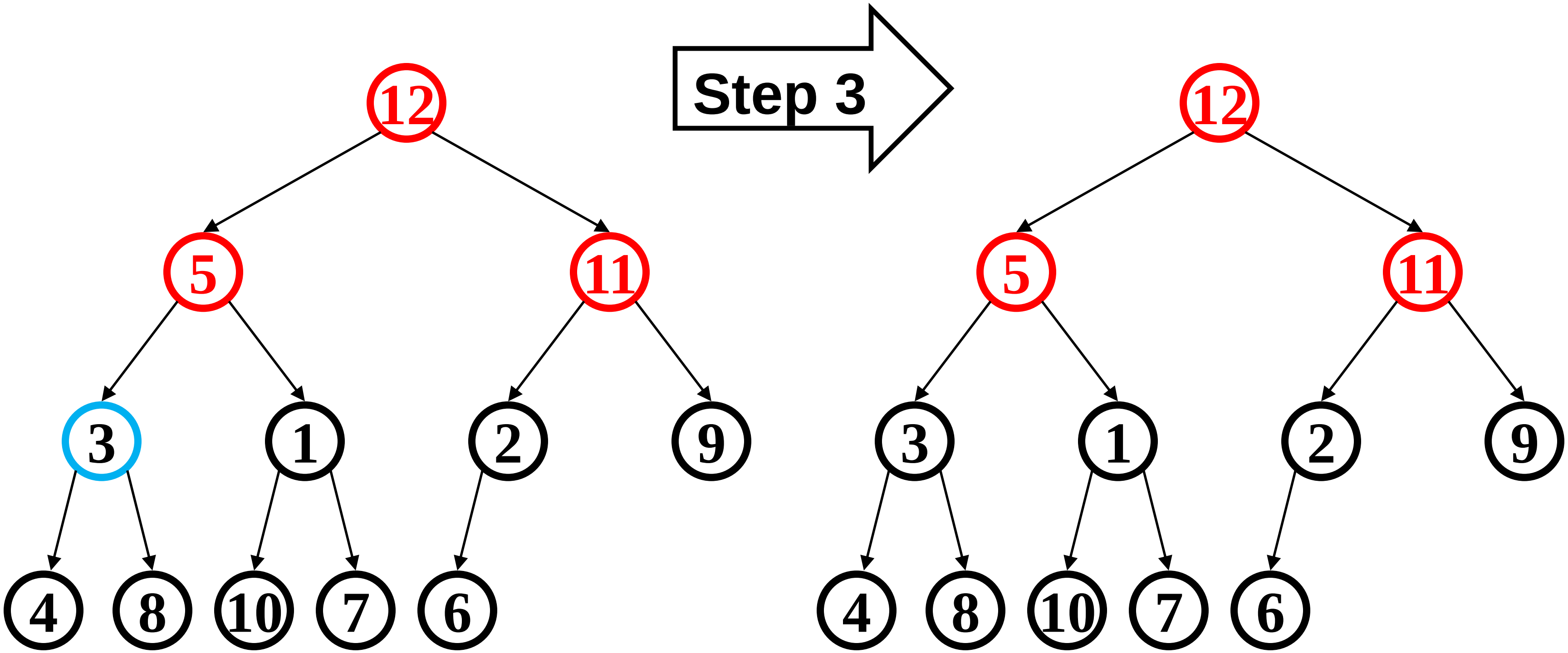
- Happens to already be less than children

Example



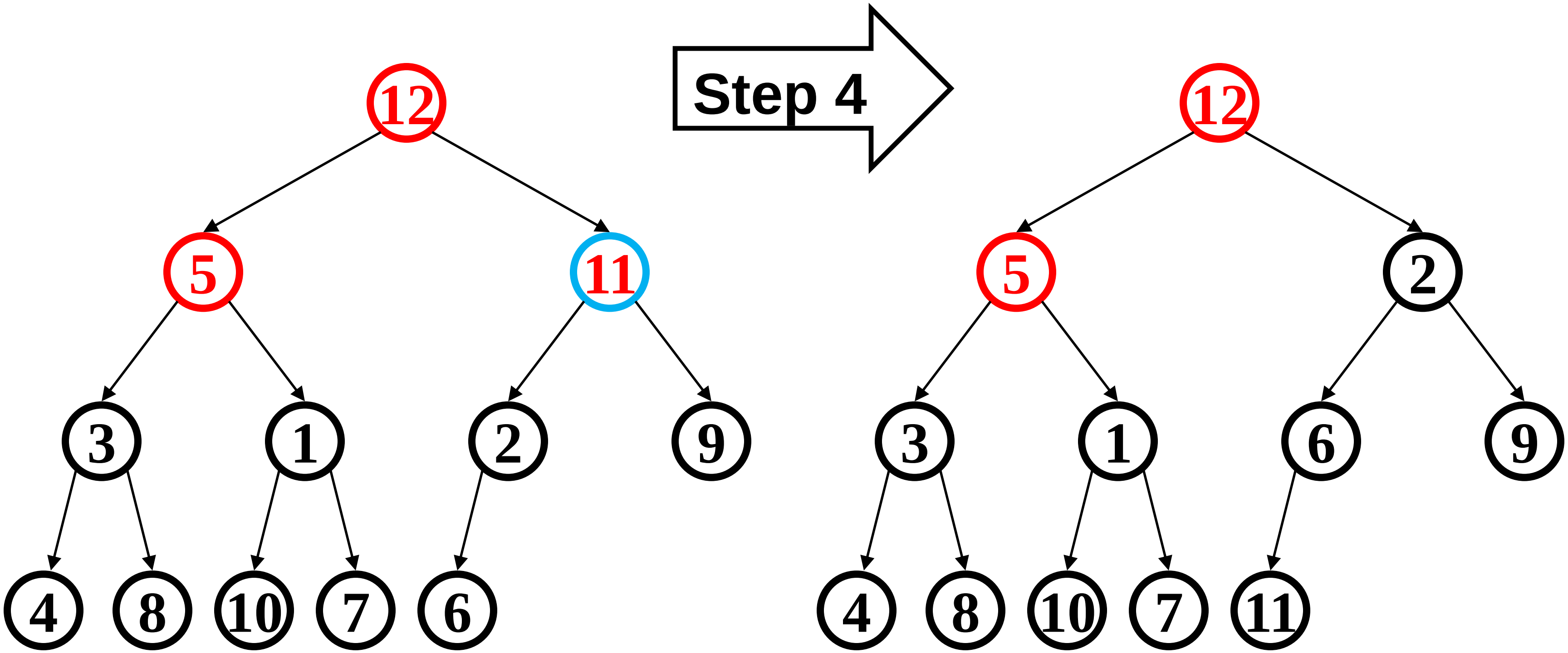
- 10 percolates down (and notice that 1 moves up)

Example



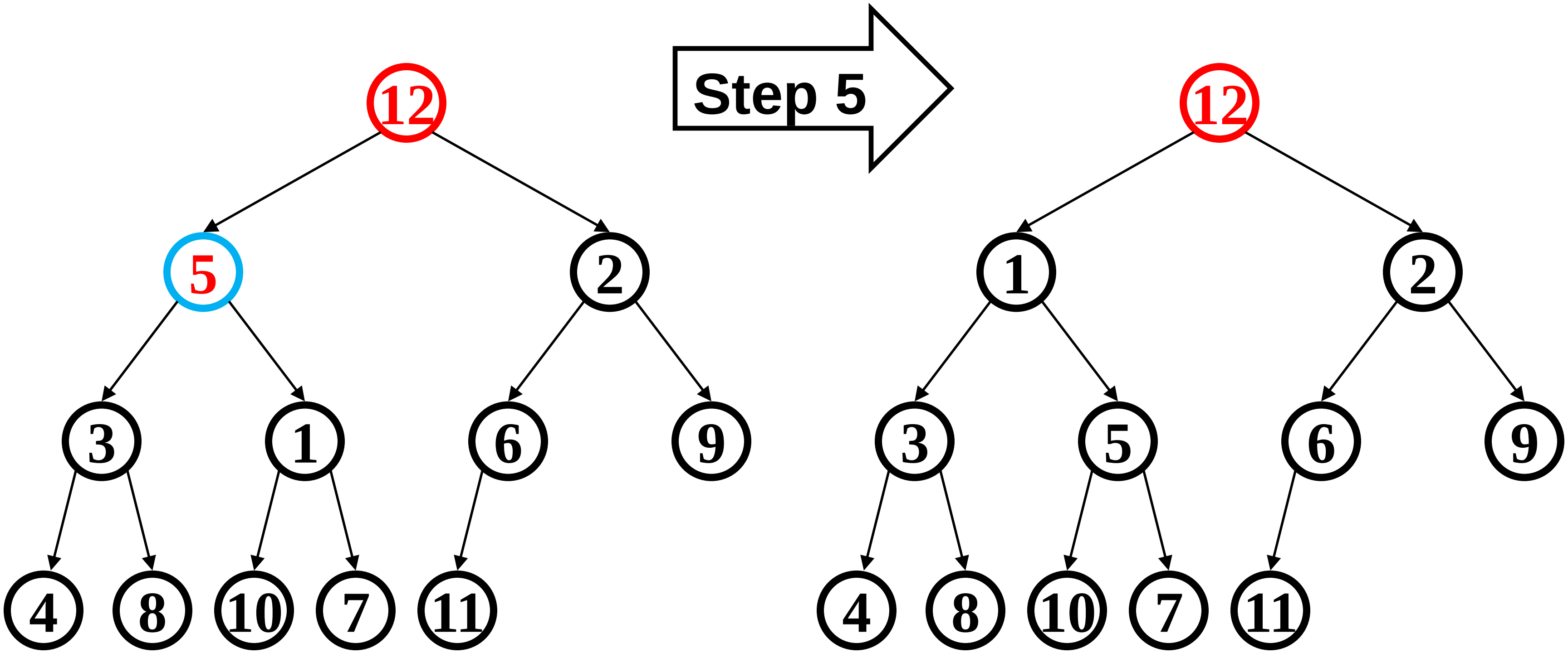
- Another nothing-to-do step

Example



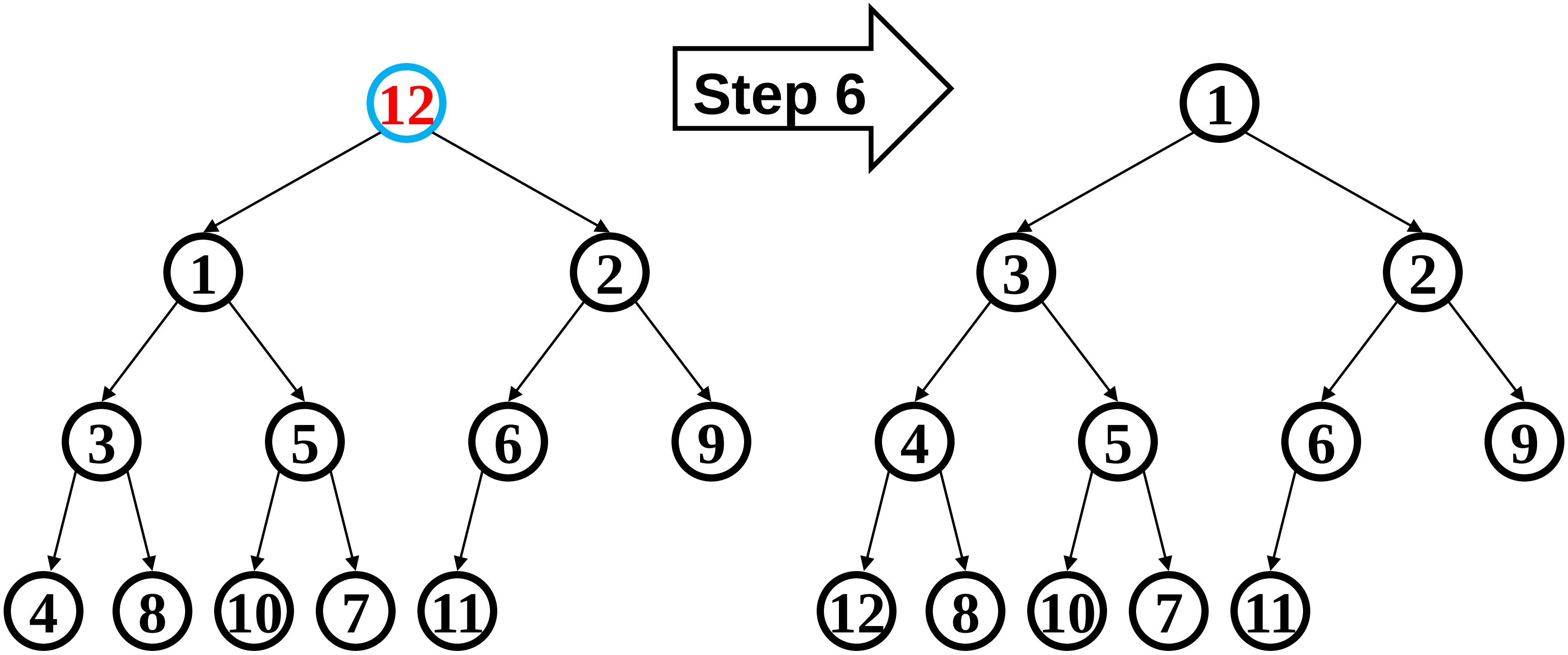
- Percolate down as necessary (first 2, then 6)

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 > size/2`**, then **`j`** is a leaf
  - Otherwise its left child would be at position > **`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) + …) < 2 (page 4 of Weiss)
  - So at most **`2(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

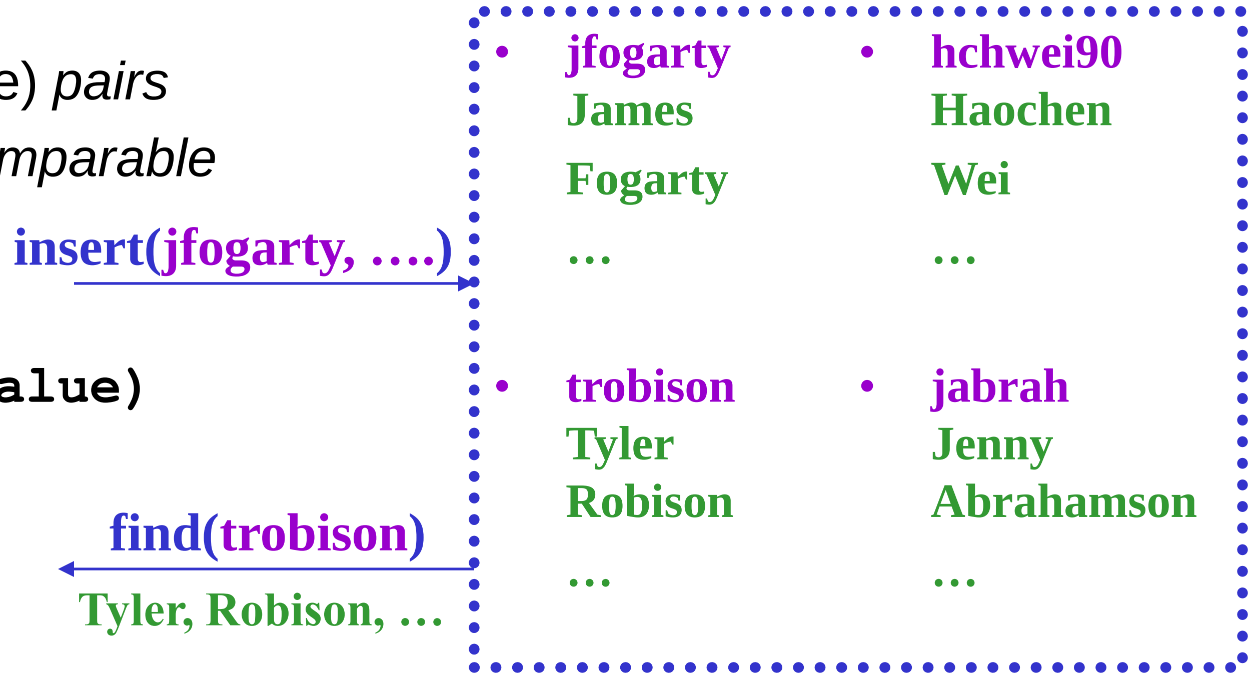
Lecture 6: Dictionary, BST, AVL Tree

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)`**
  - …



Probably the single most common ADT in everyday programs

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

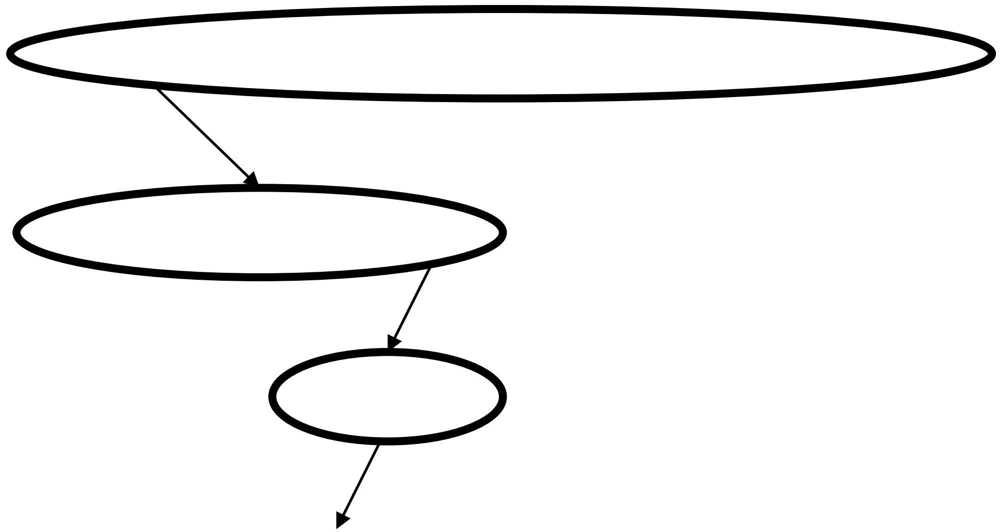
Simple Implementations

For dictionary with *n* key/value pairs

	`insert`	`find`	`delete`
• 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	$O(n)$	$O(\texttt{log}\ n)$	$O(n)$
	|		|
	$\texttt{log}\ n + n$		$\texttt{log}\ n + n$

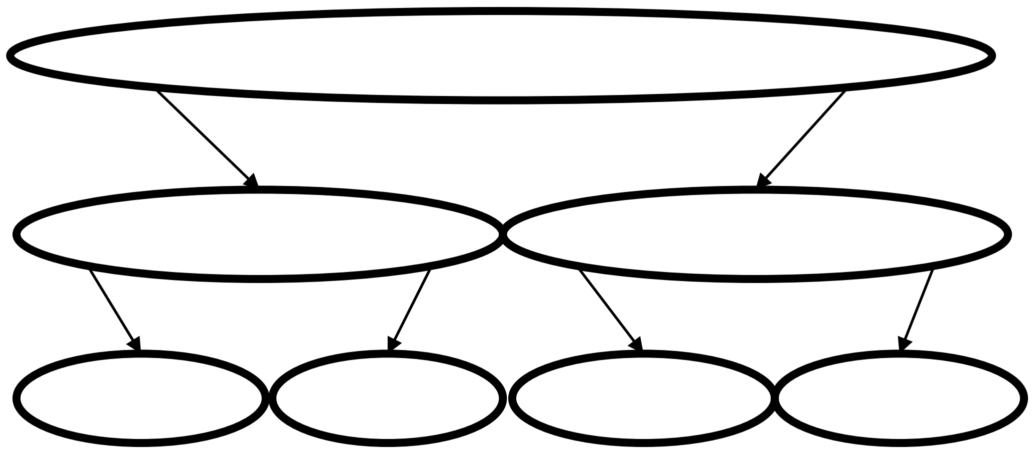
Binary Search

Target 4



1	**3**	**4**	**5**	**7**	**8**	**9**	**10**

Binary Search Tree

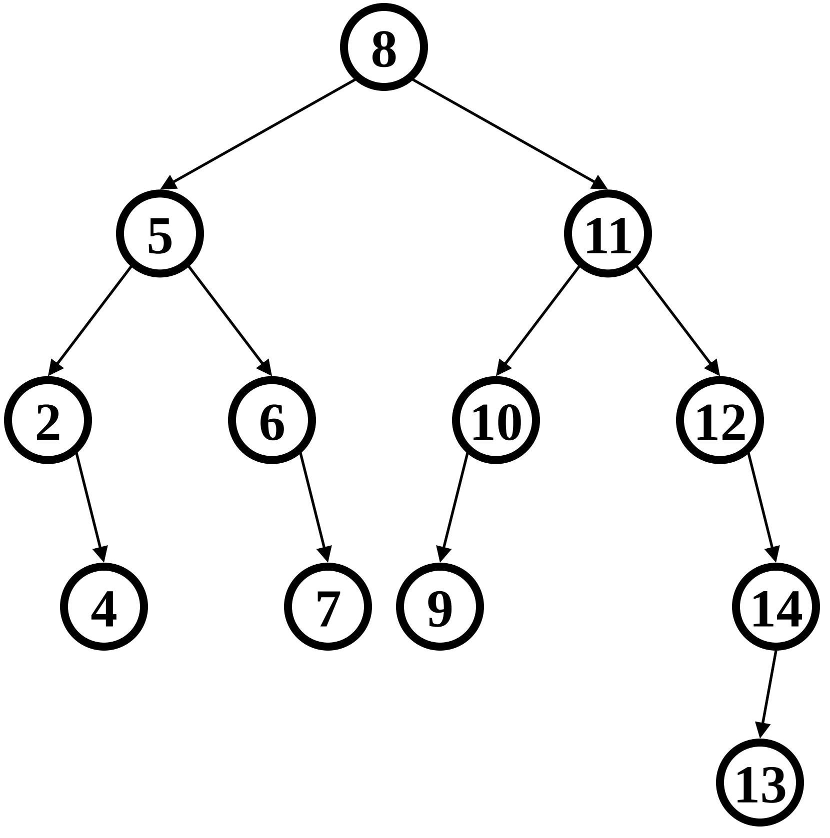


1	**3**	**4**	**5**	**7**	**8**	**9**	**10**

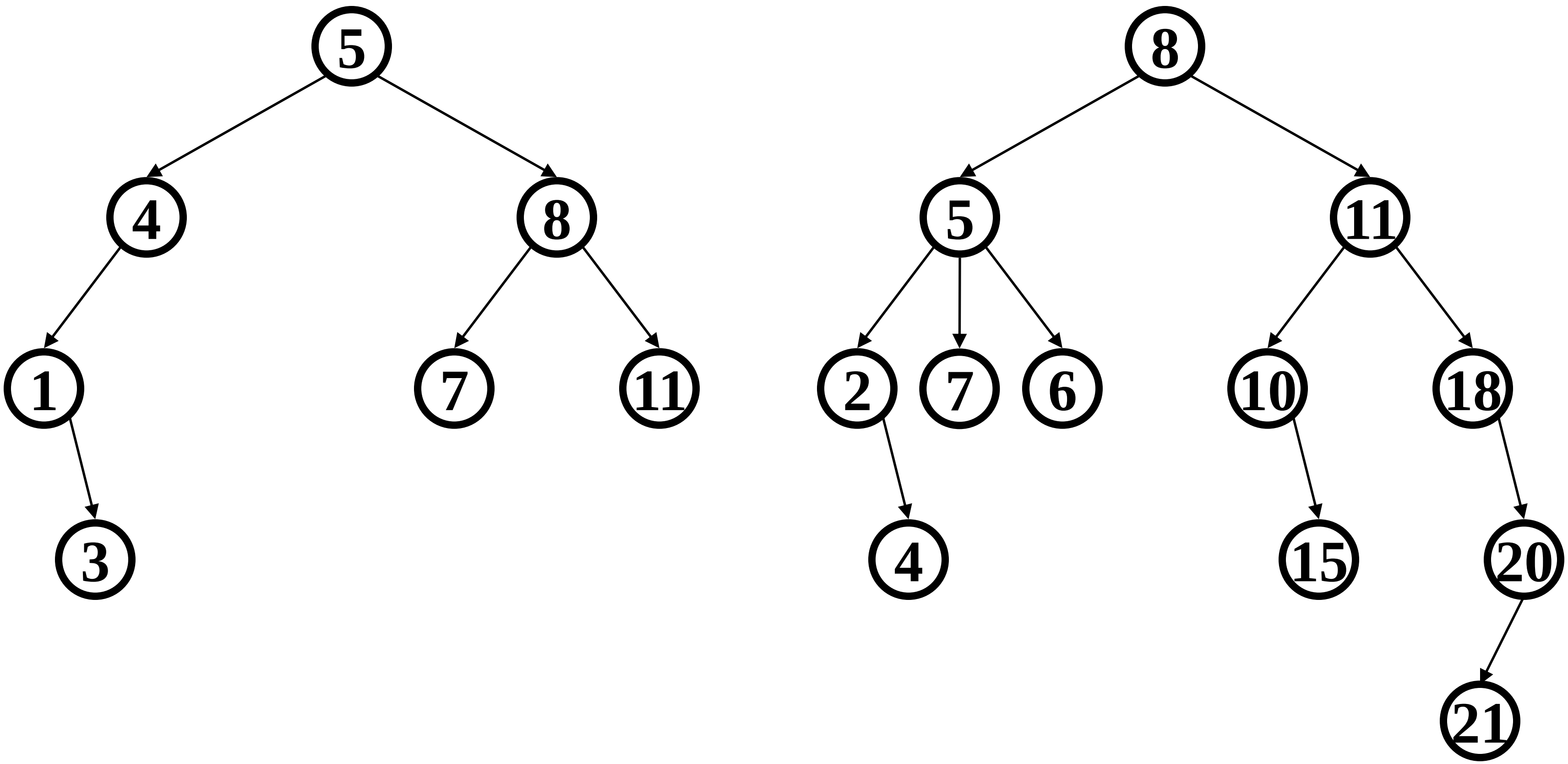
Our goal is the performance of binary search in a tree representation

Binary Search Tree

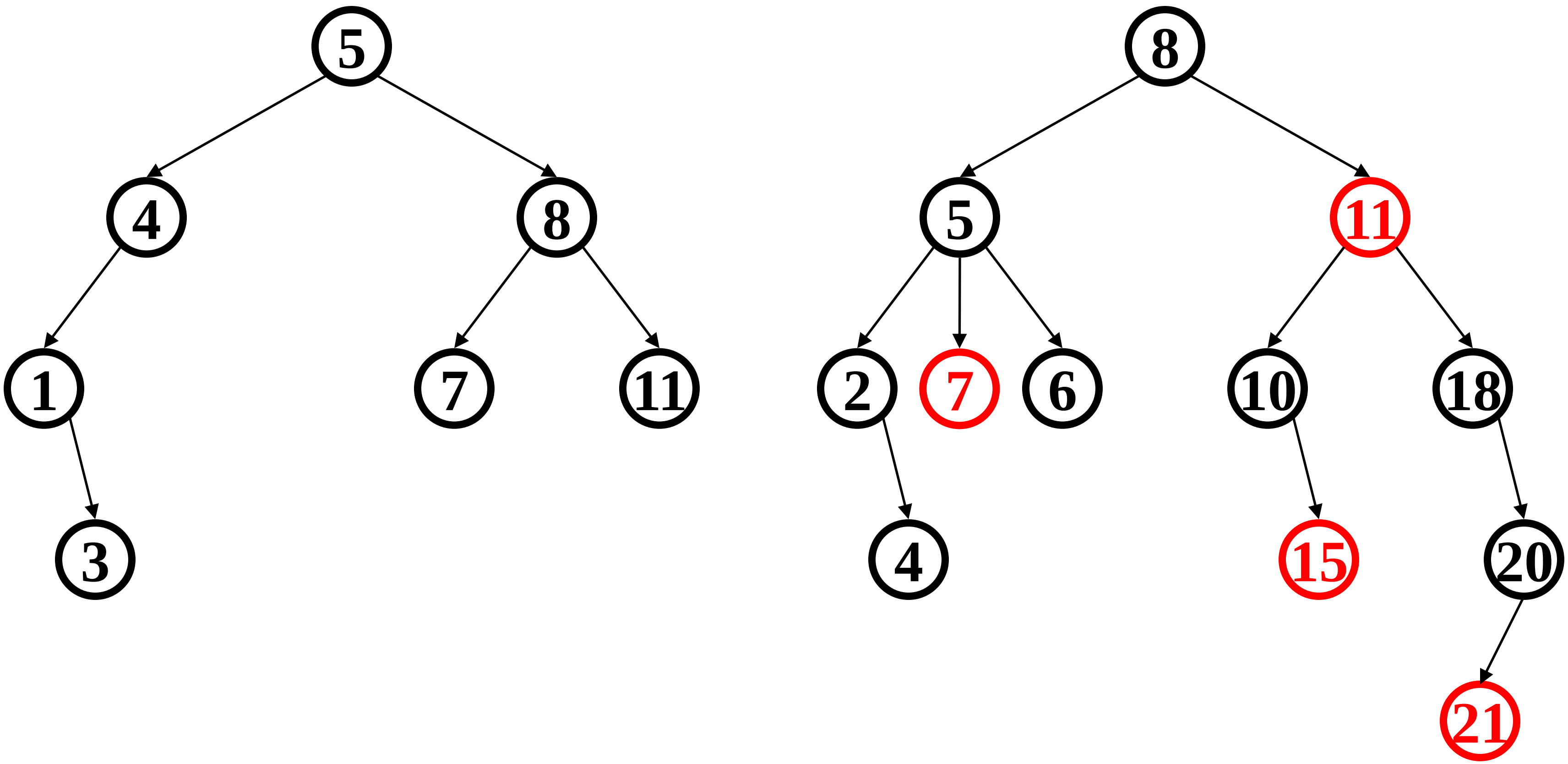
- Structure Property ("binary")
  - each node has ≤ 2 children
- Order Property
  - all keys in left subtree are smaller than node's key
  - all keys in right subtree are larger than node's key



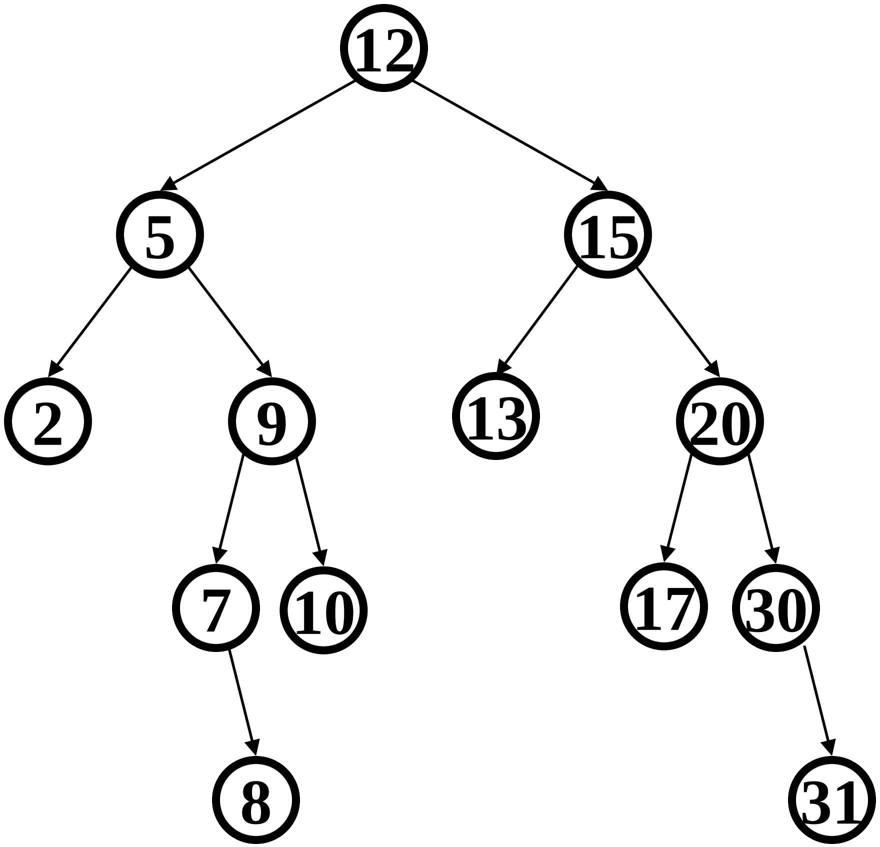
Are these BSTs?



Are these BSTs?



Insert and Find in BST



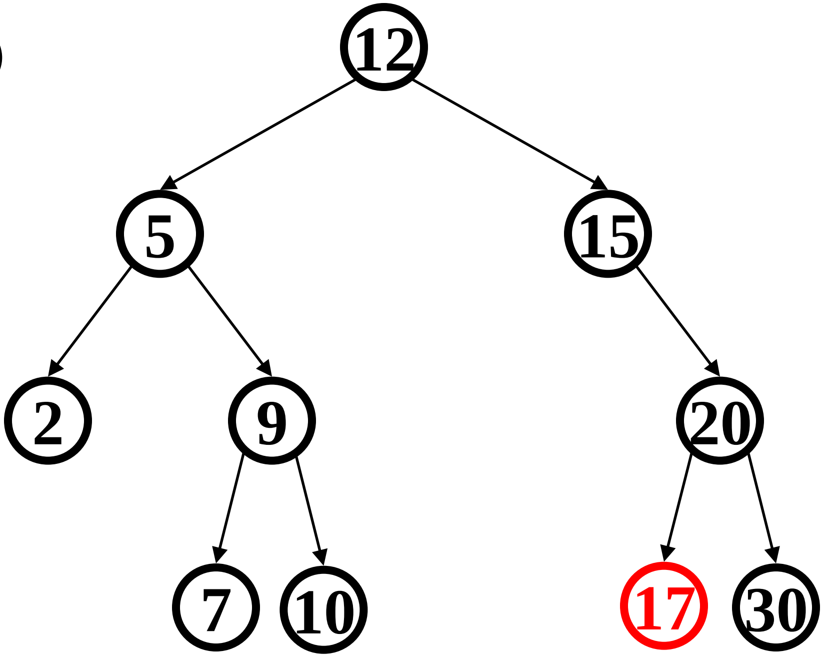

```
insert(13)
insert(8)
insert(31)
find(17)
find(11)
```

Insertion happens at leaves

Find walks down tree

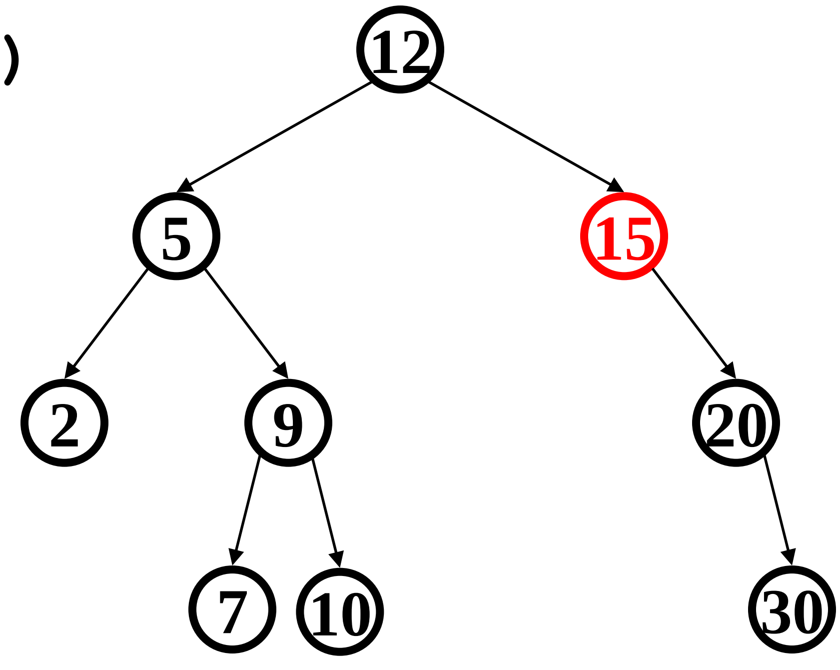
Deletion – The Leaf Case

`delete(17)`



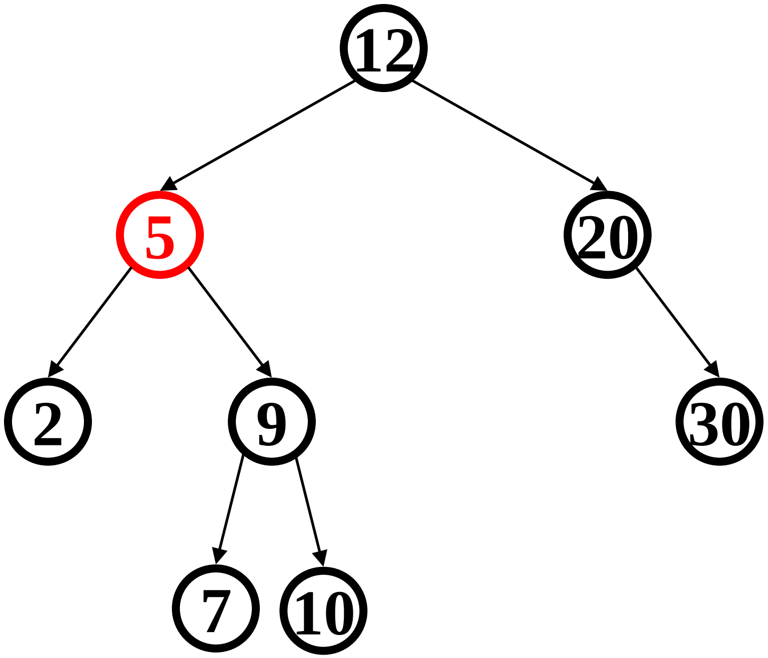
Deletion – The One Child Case

`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

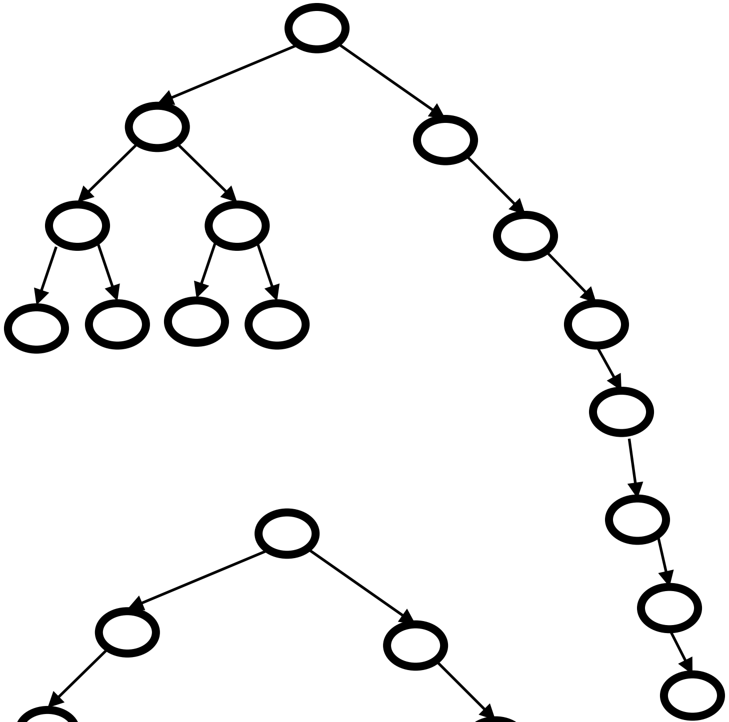
Require a **Balance Condition** that will:

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

Potential Balance Conditions

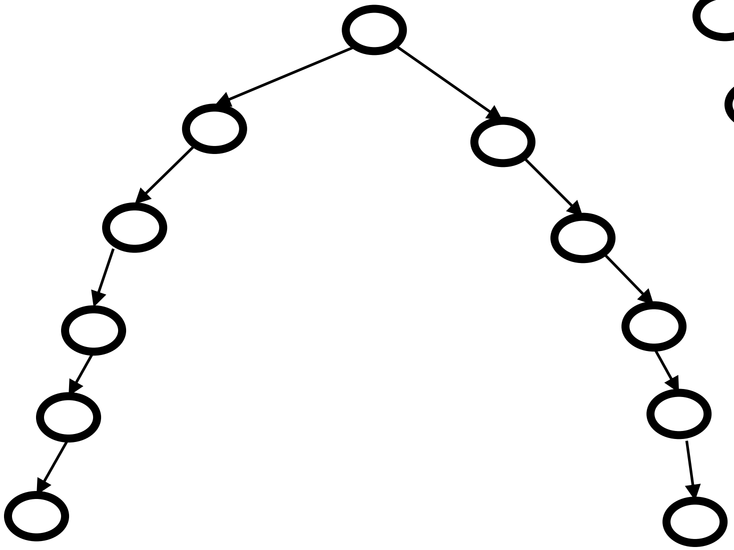
1. Left and right subtrees of the *root* have equal number of nodes

   Too weak!
   Height mismatch example:



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

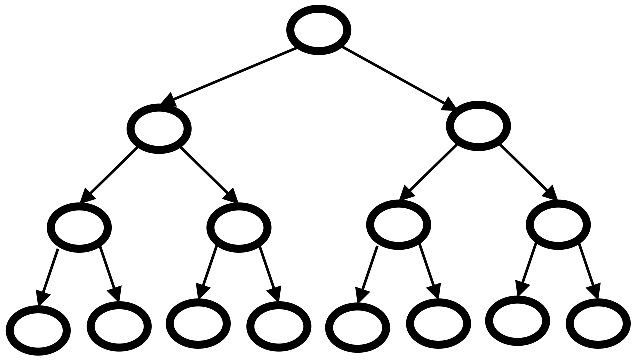
   Too weak!
   Double chain example:



Potential Balance Conditions

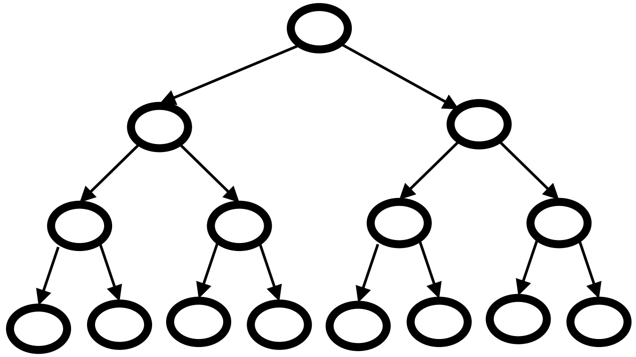
3. Left and right subtrees of every node have equal number of nodes

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)



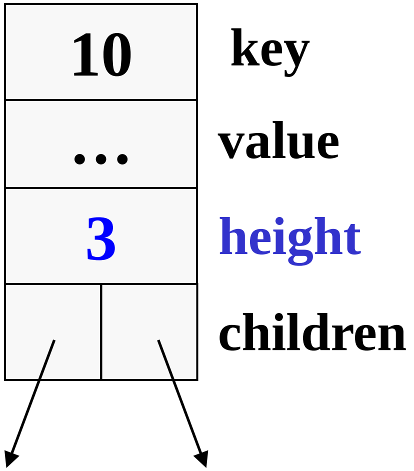
The AVL Balance Condition

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

Definition: **balance**(*node*) = height(*node*.left) – height(*node*.right)

AVL *property*: **for every node *x*, –1 ≤ balance(*x*) ≤ 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?

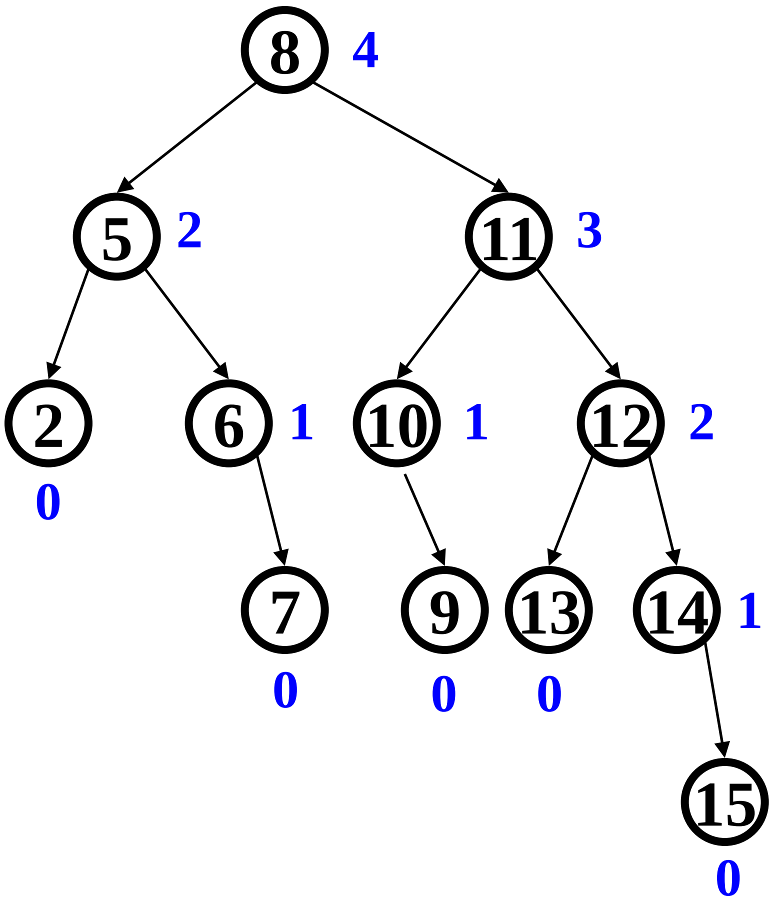
This is the minimum AVL tree of height 4

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

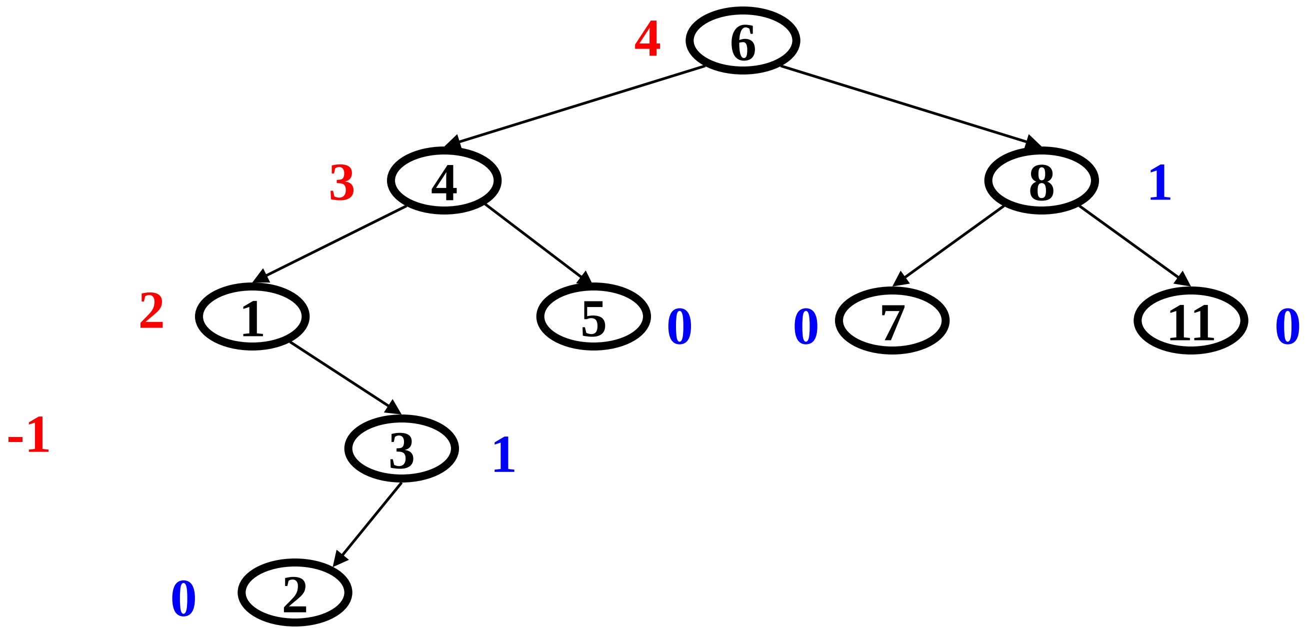
$S(h) = S(h-1) + S(h-2) + 1$

$S(-1) = 0$
$S(0) = 1$
$S(1) = 2$
$S(2) = 4$
$S(3) = 7$
$S(4) = 12$

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



An AVL Tree?



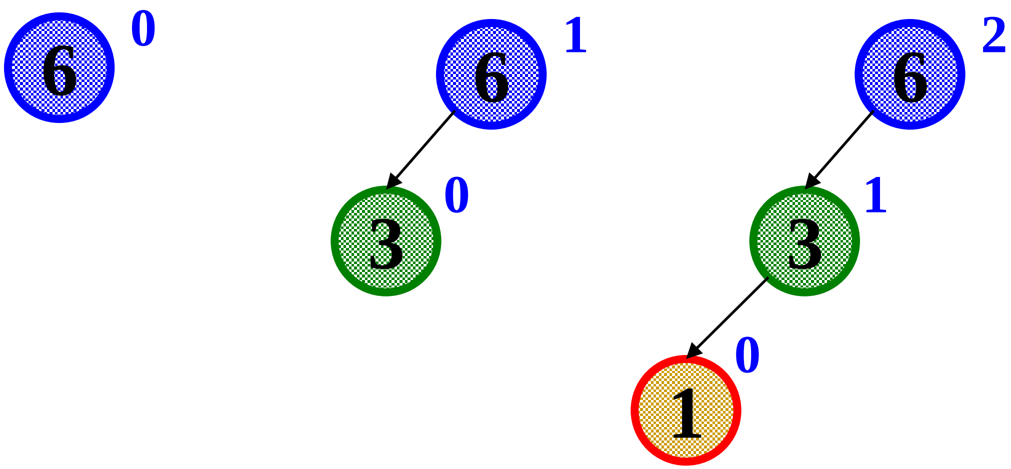
AVL Tree Operations

- **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)

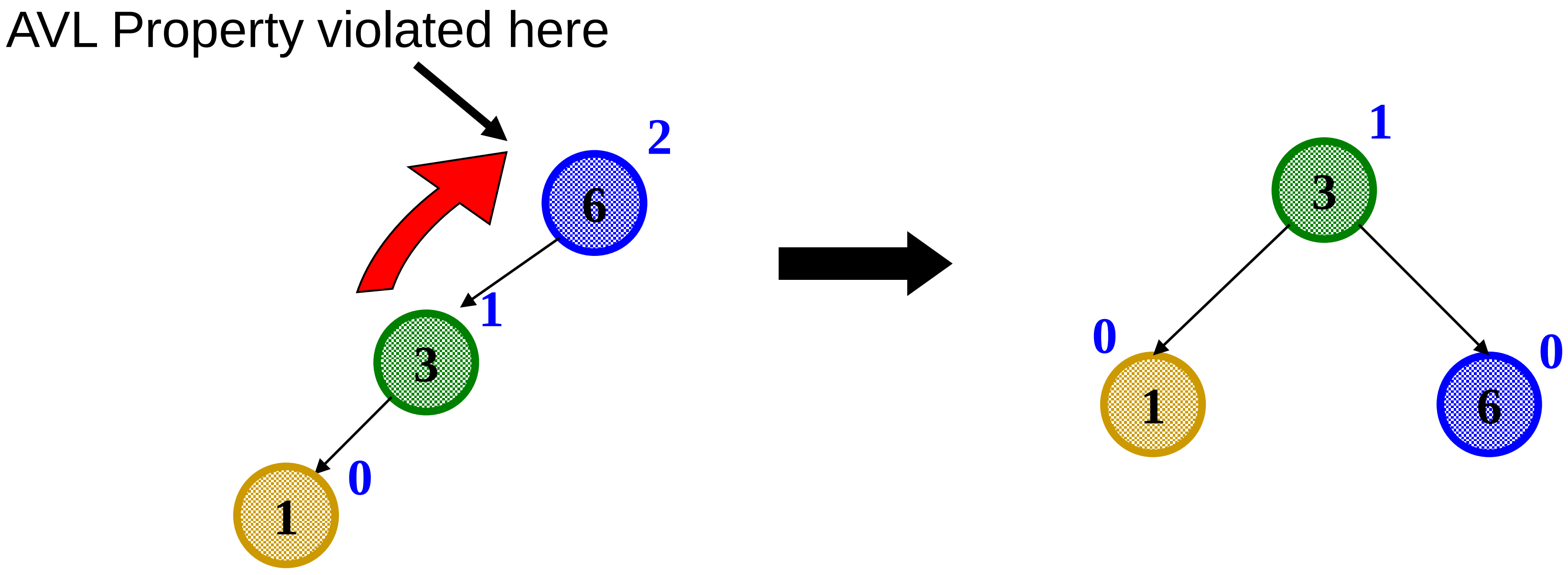
Insert(1)

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



Insert and Detect Potential Imbalance

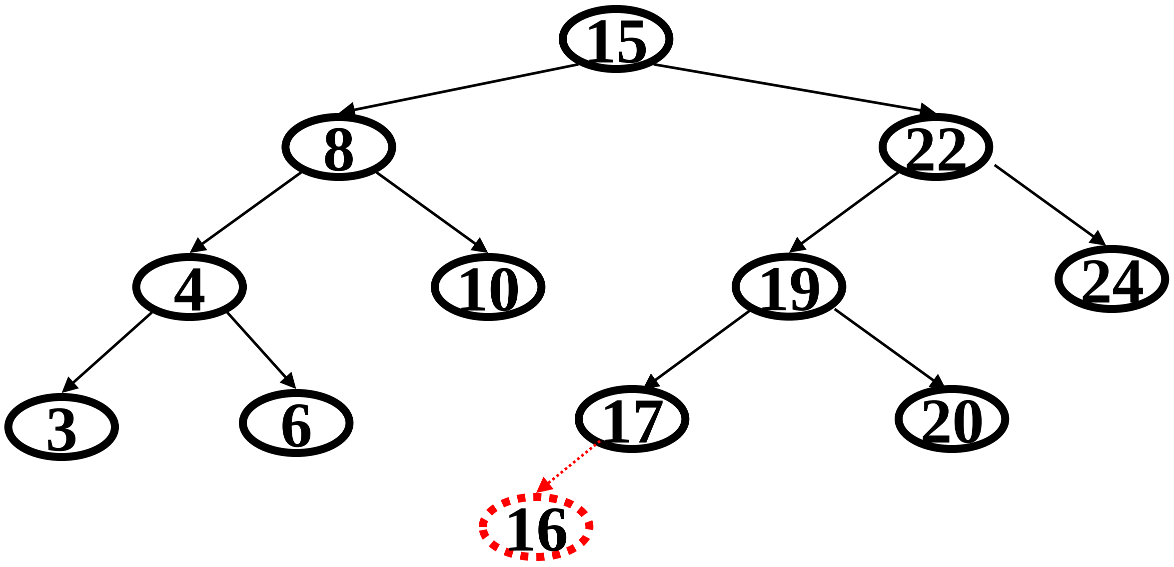
1. Insert the new node (at a leaf, as in a BST)
2. For each node on the path from the new leaf to the root
   the insertion may, or may not, have changed the node's height
3. After recursive insertion in a subtree
   detect height imbalance
   perform a *rotation* to restore balance at that node

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

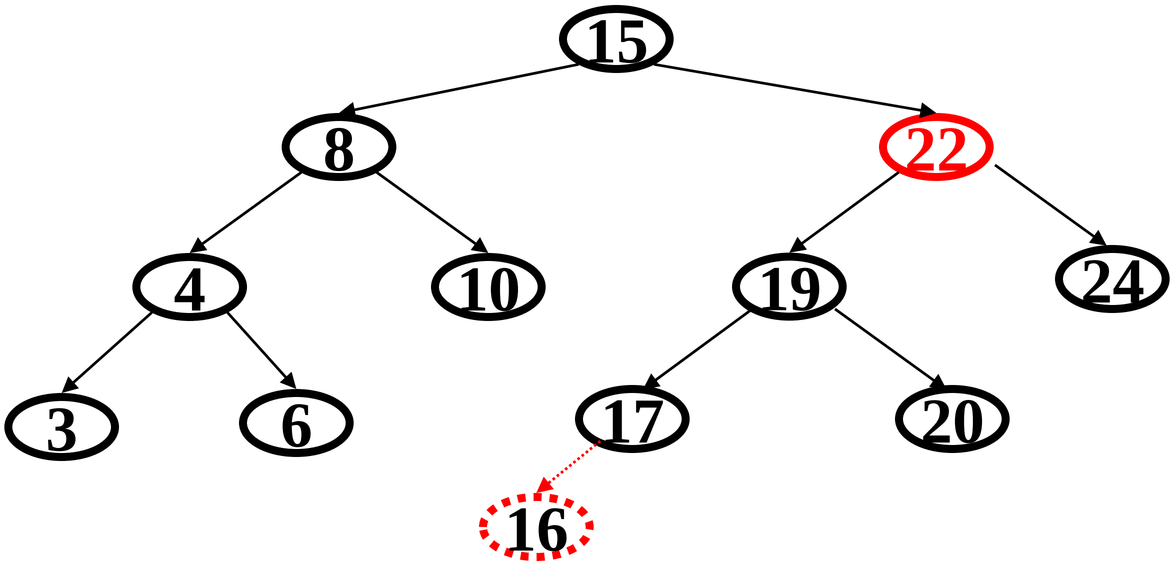
Fact that an implementation can ignore:

- 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

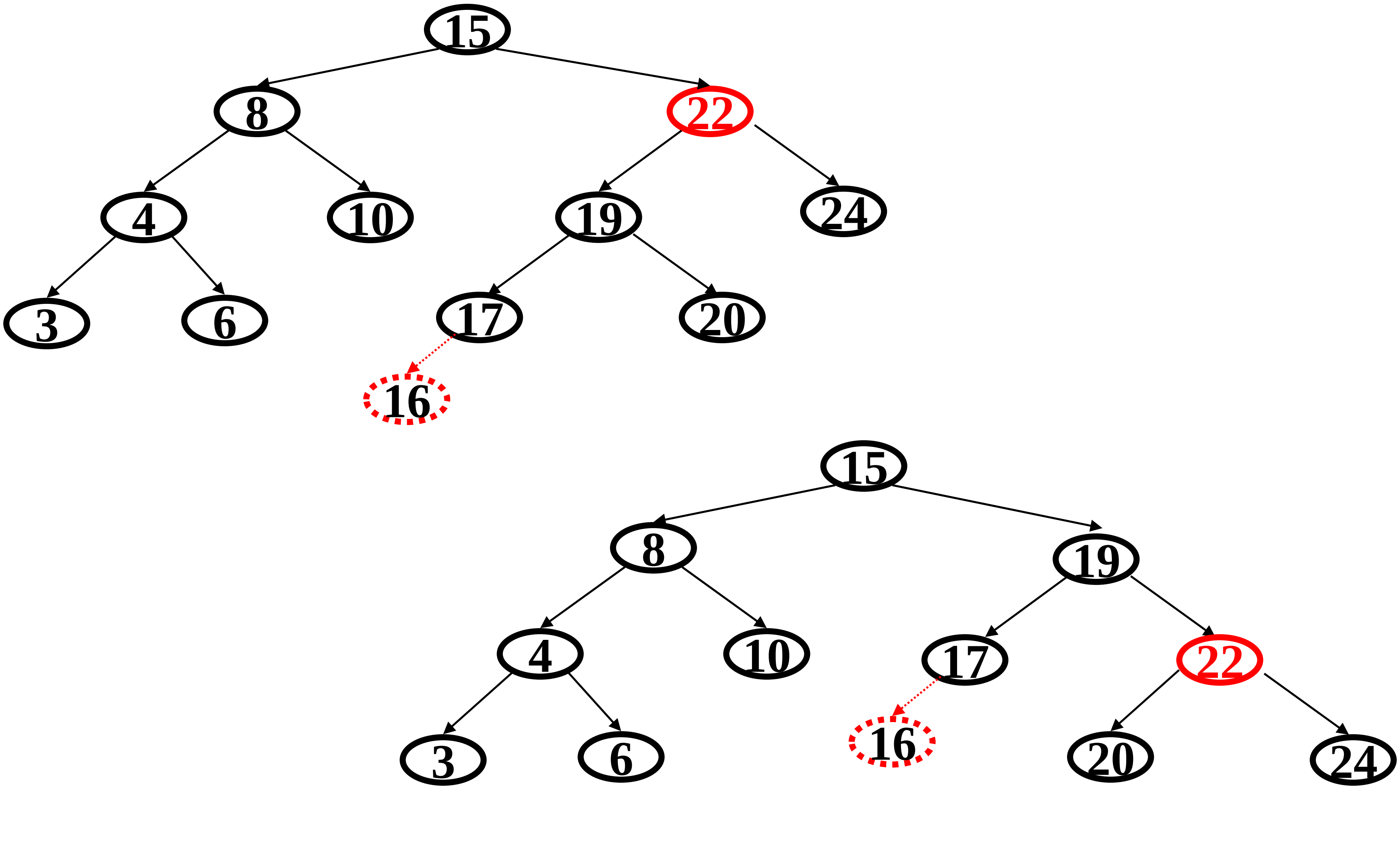
Single Rotation Example: Insert(16)



Single Rotation Example: Insert(16)

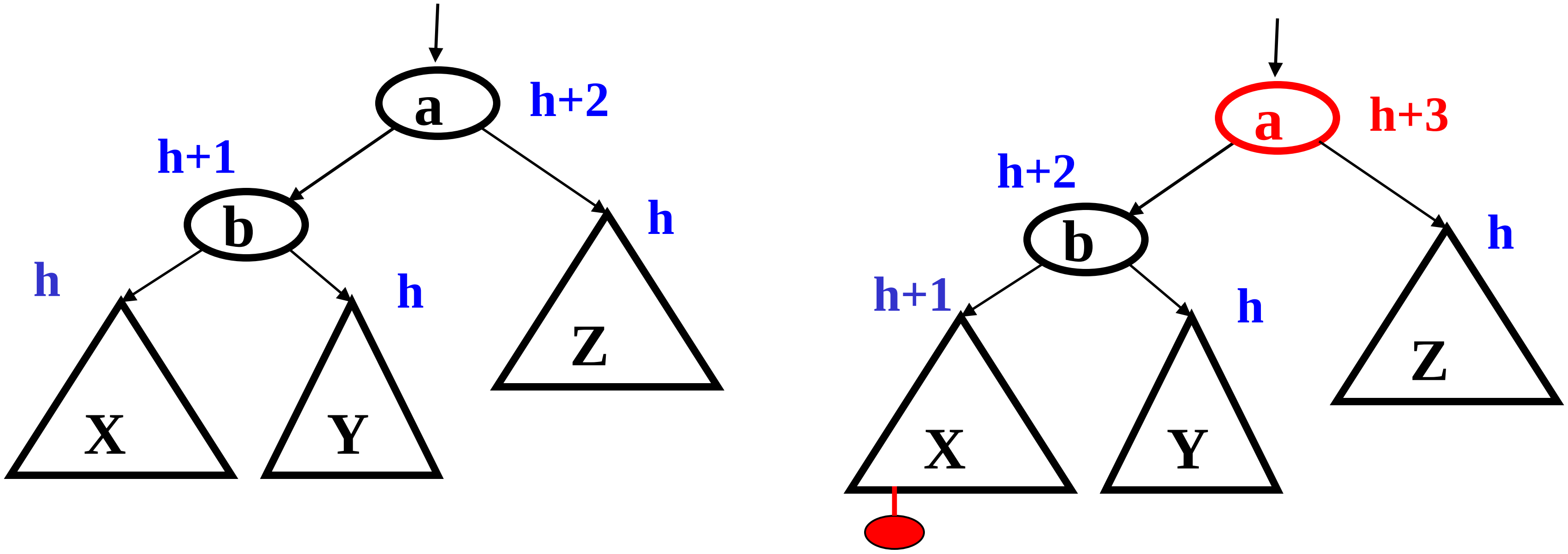


Single Rotation Example: Insert(16)



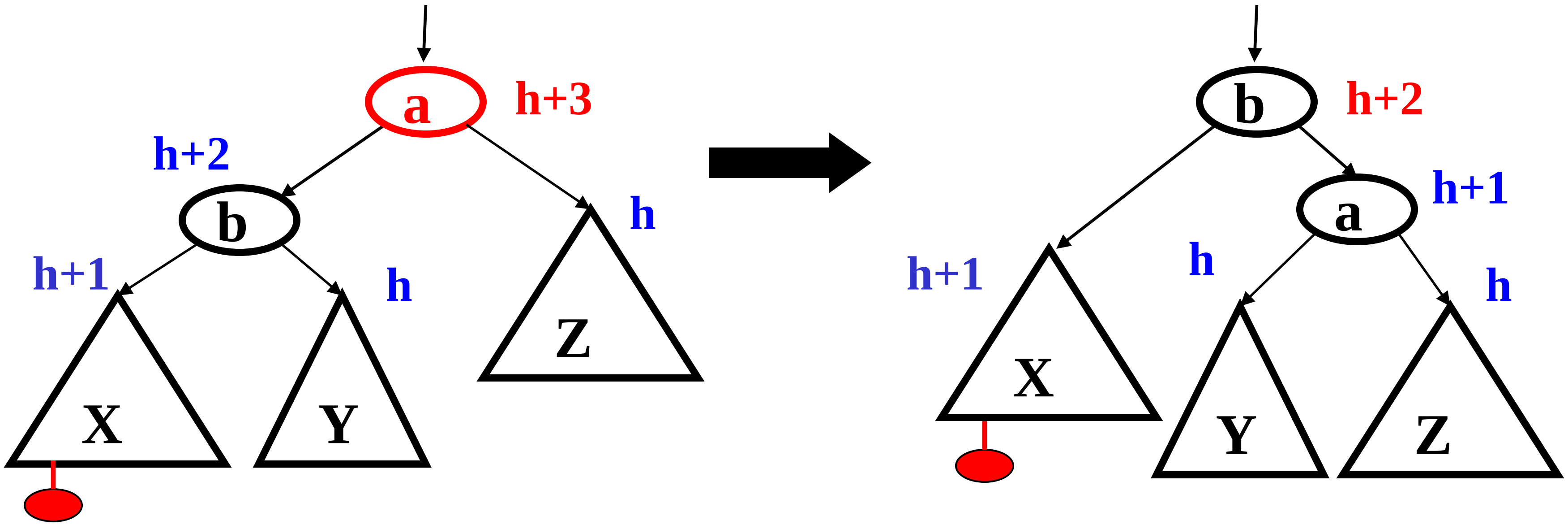
Left-Left Case

- 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

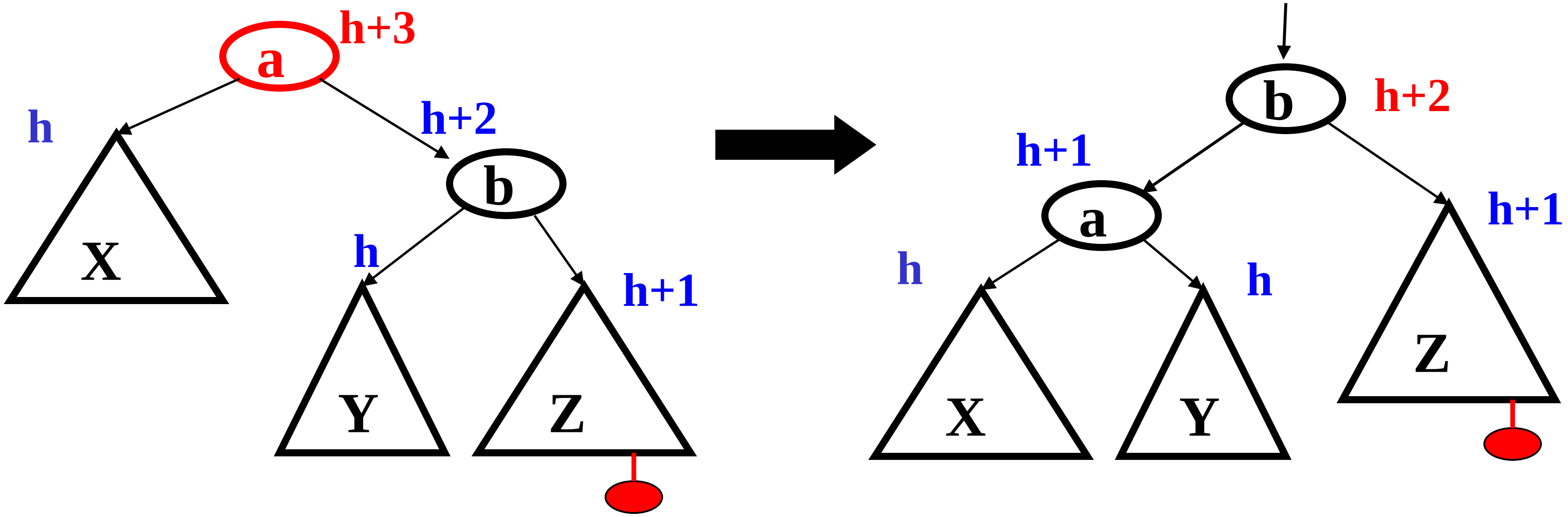
- So we rotate at ***a***, using BST facts: X < b < Y < a < Z



- 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

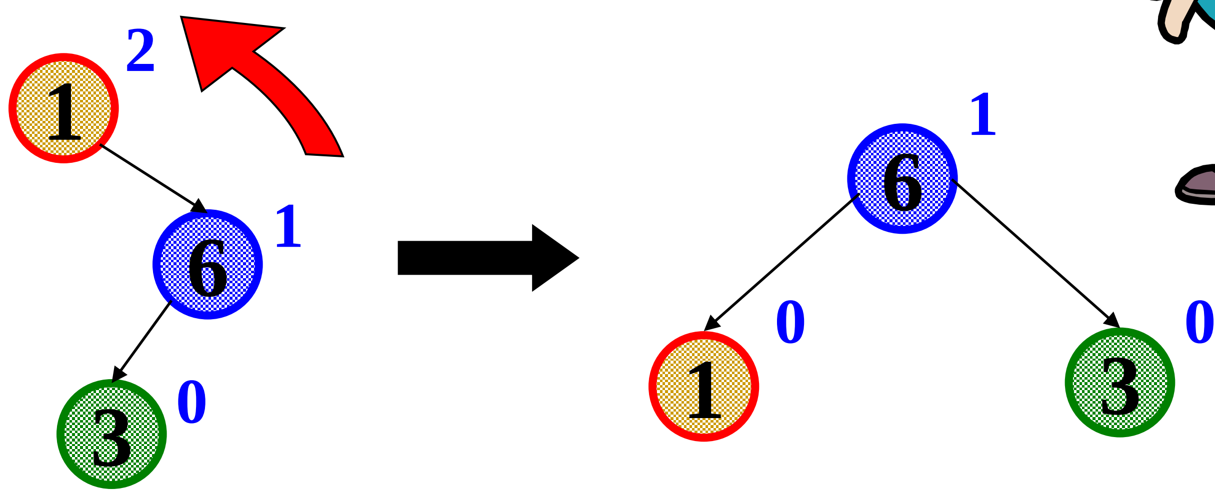


The Other Two Cases

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

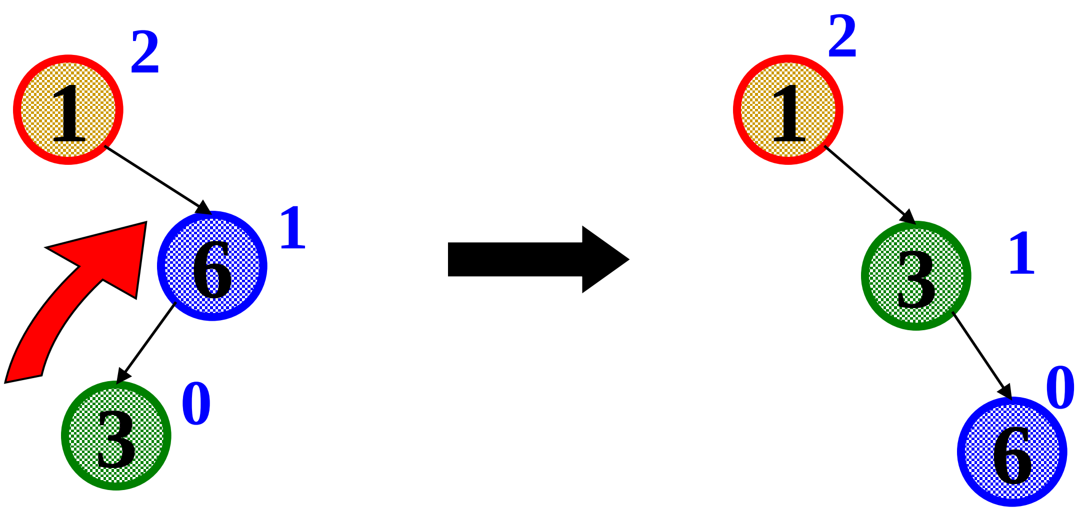


The Other Two Cases

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

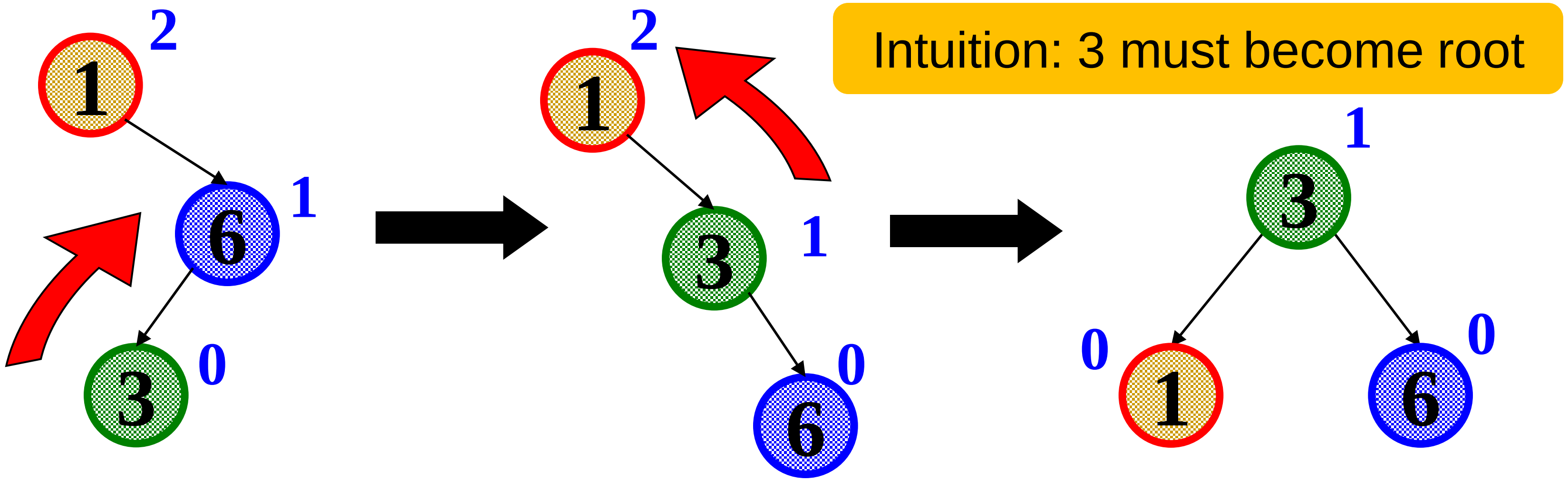


Double Rotation

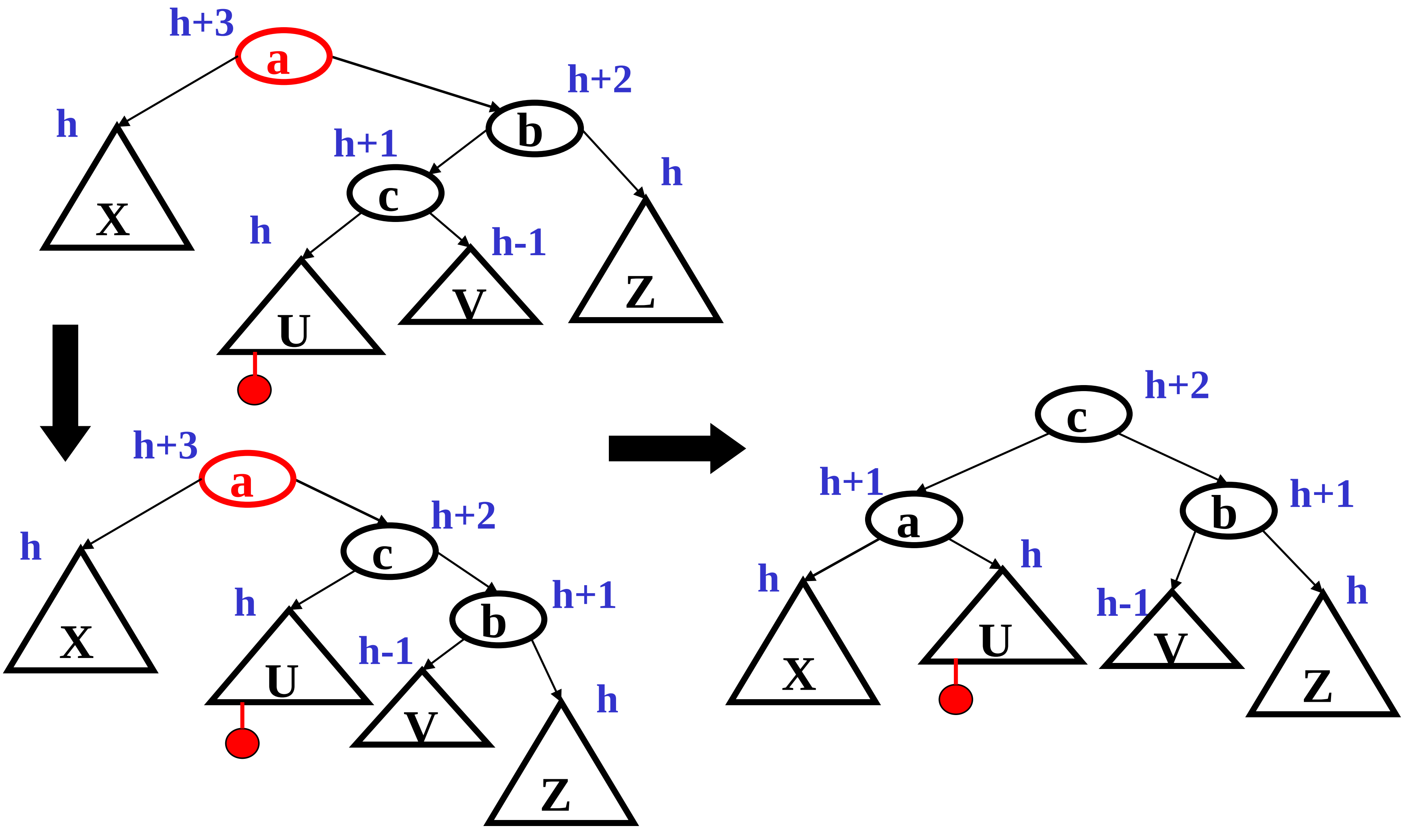
- First attempt at rotation violated the BST property
- Second attempt at rotation did not fix balance
- But if we do both, it works!

Double rotation:

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

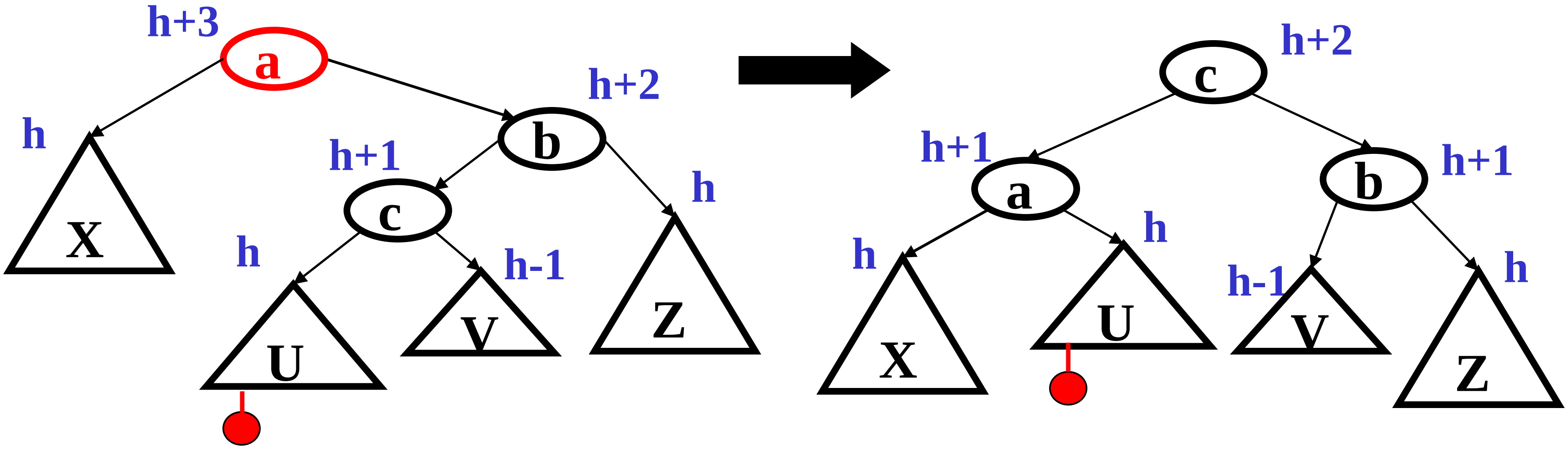


Right-Left Case



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:



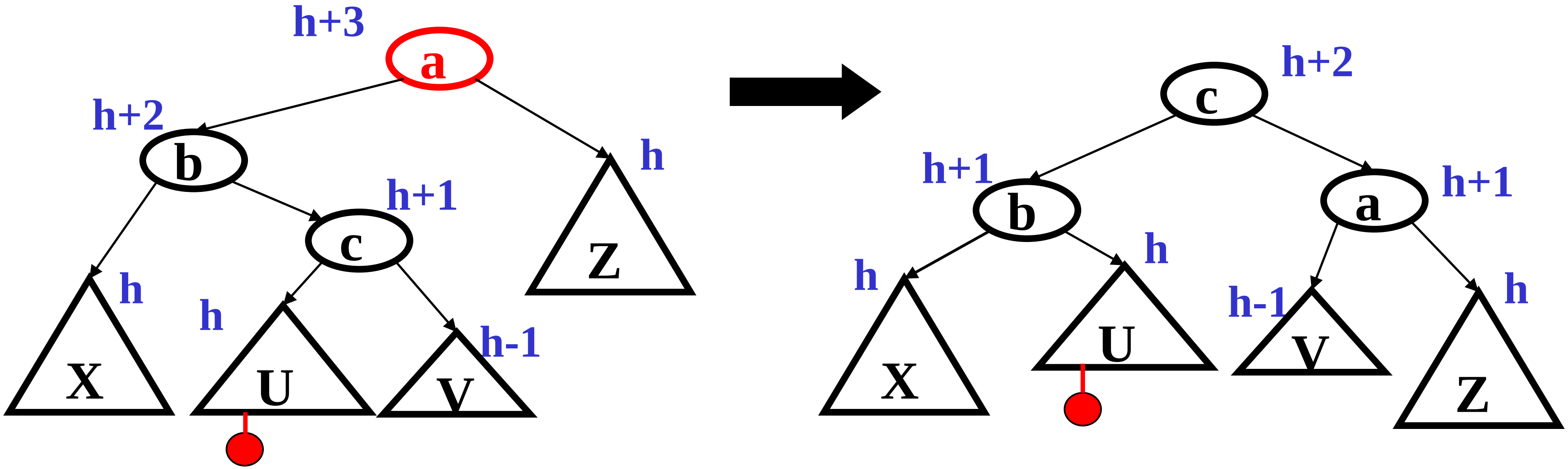
Easier to remember than you may think:

Move c to grandparent's position

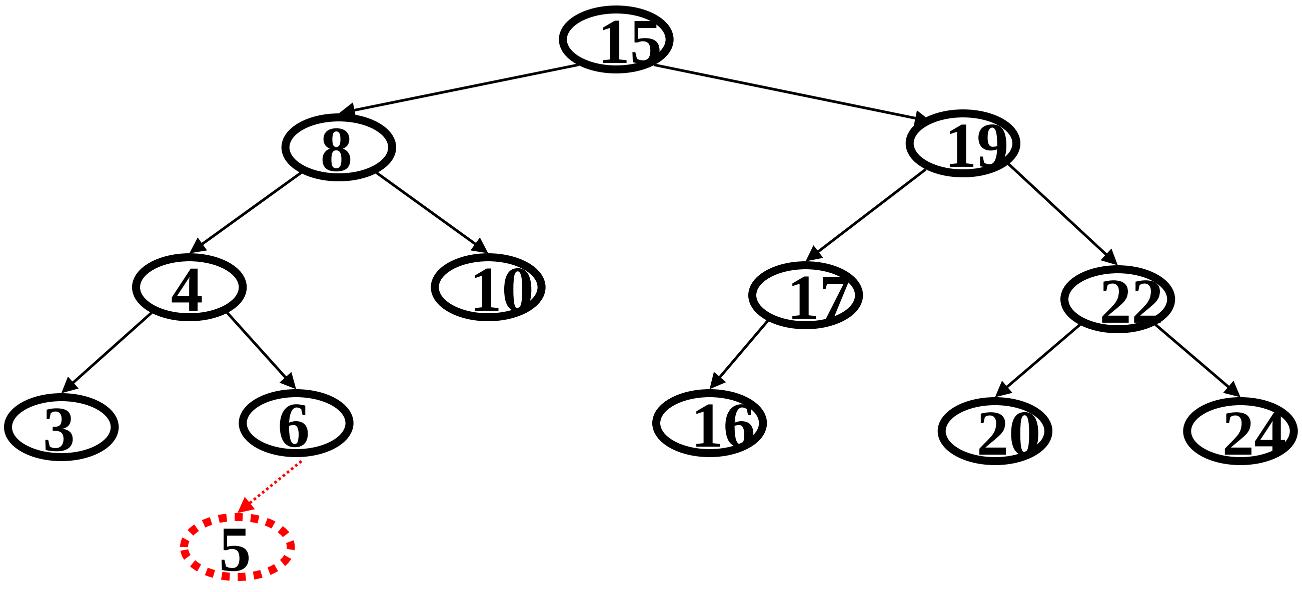
Put a, b, X, U, V, and Z in the only legal position for a BST

Left-Right Case

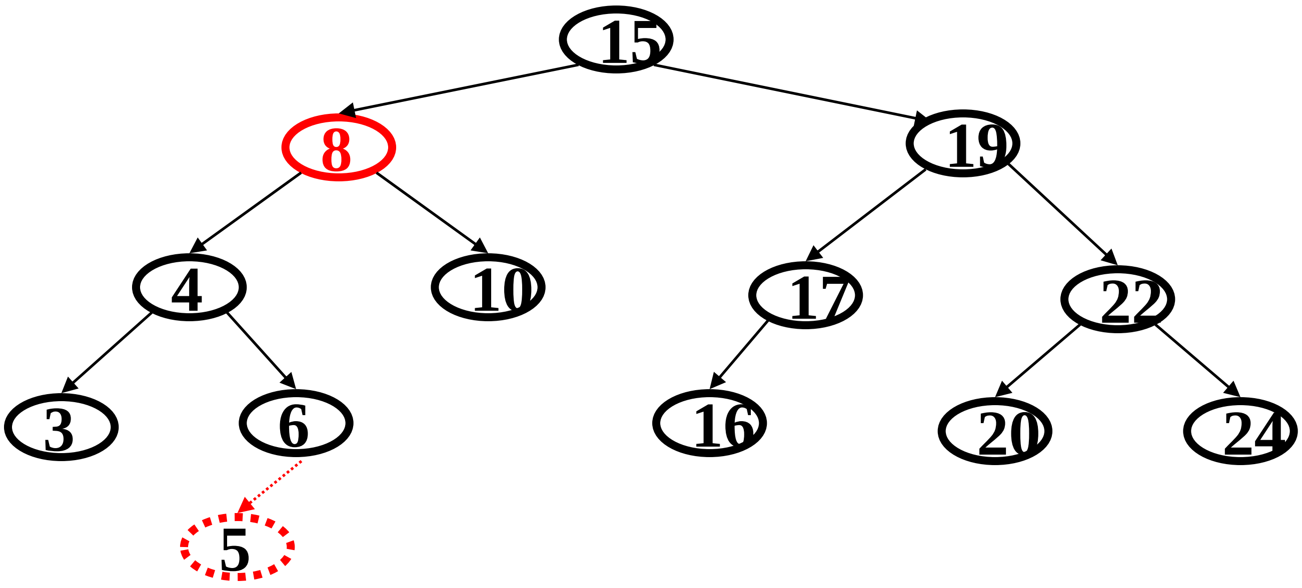
- Mirror image of right-left
  - No new concepts, just additional code to write



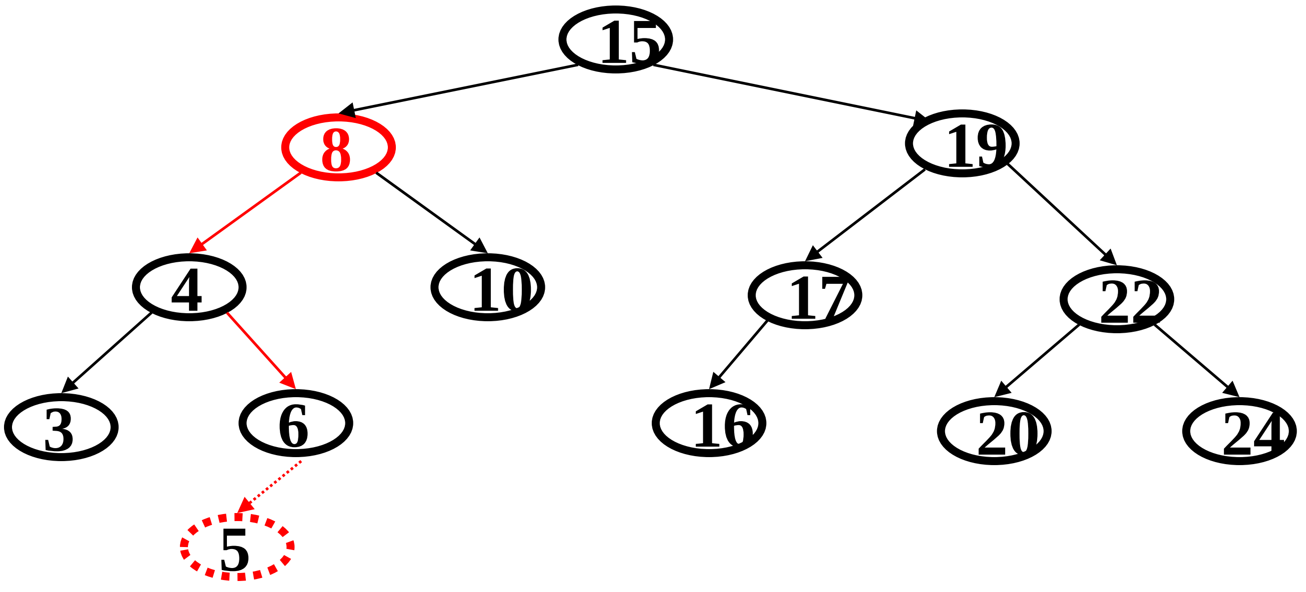
Double Rotation Example: Insert(5)



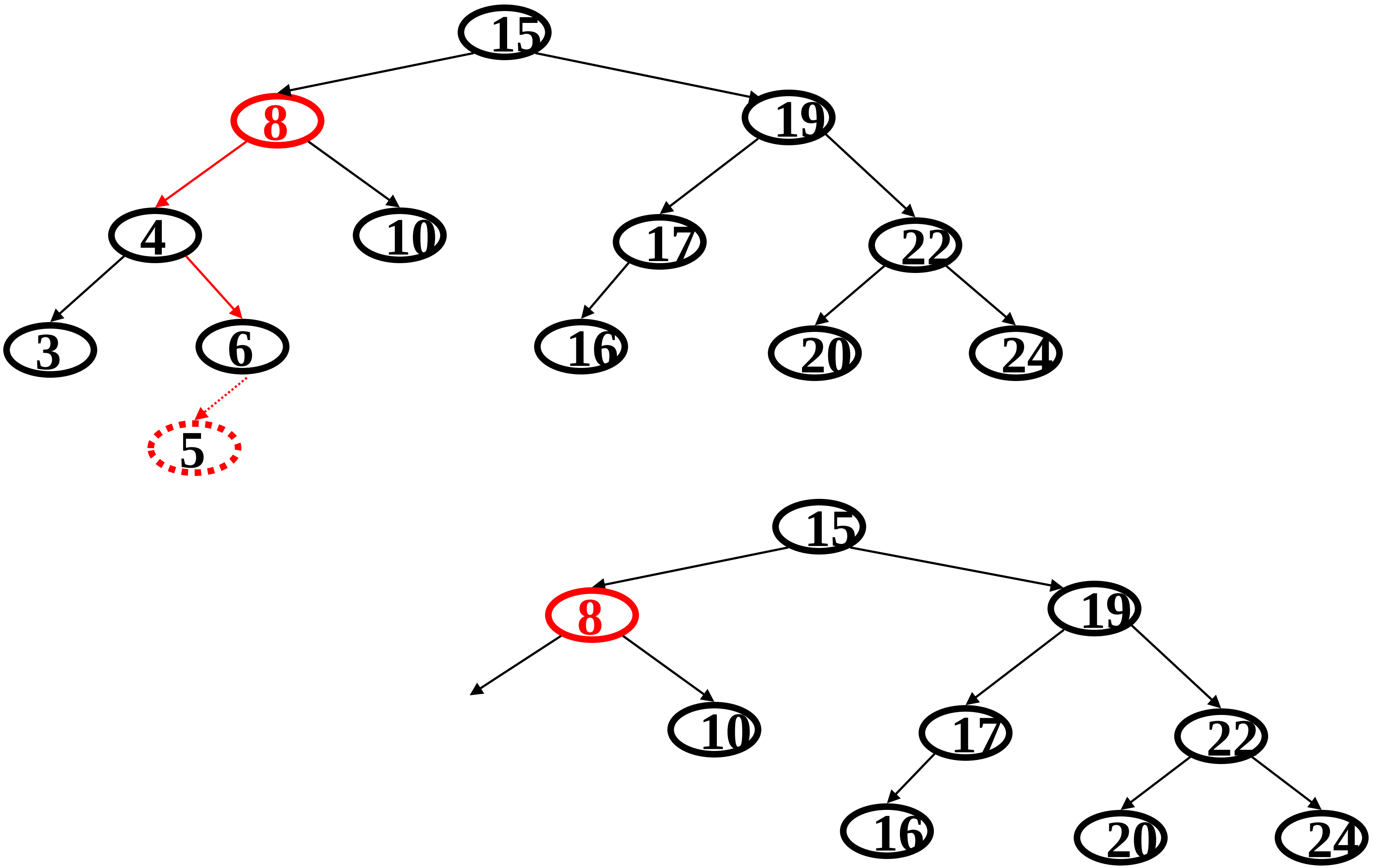
Double Rotation Example: Insert(5)



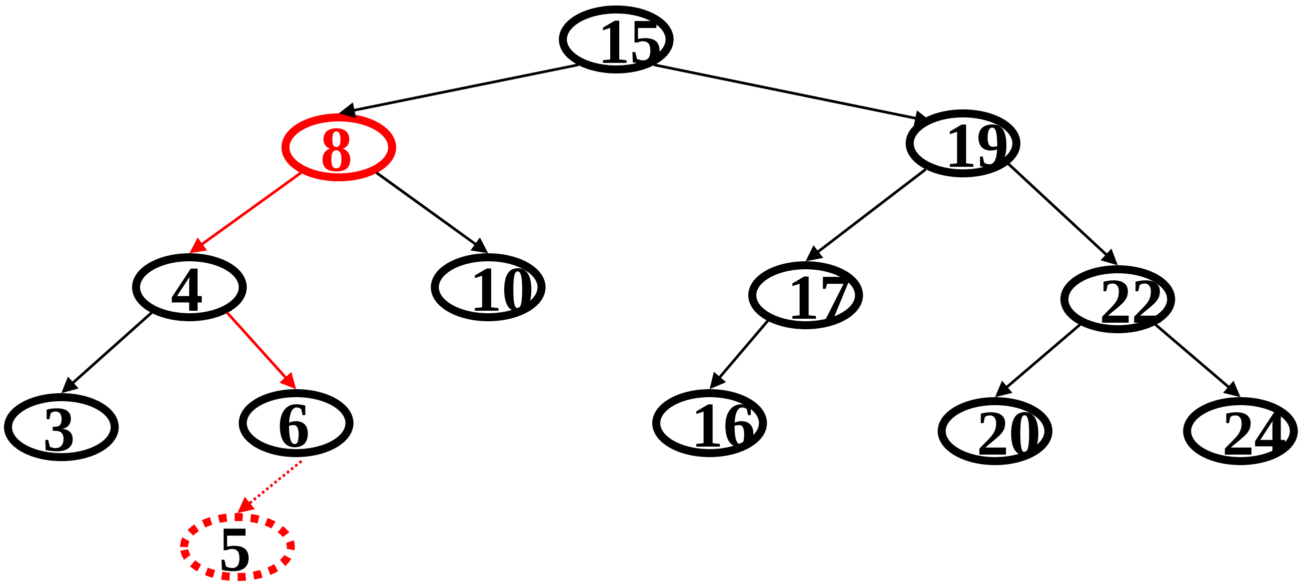
Double Rotation Example: Insert(5)

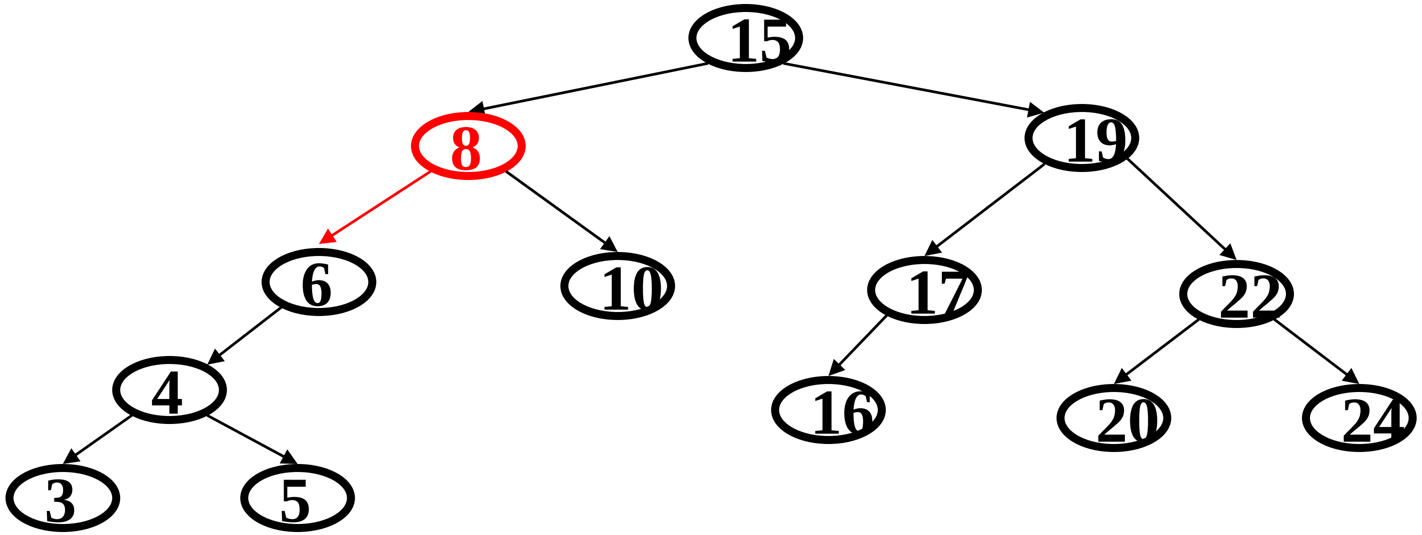


Double Rotation Example: Insert(5)

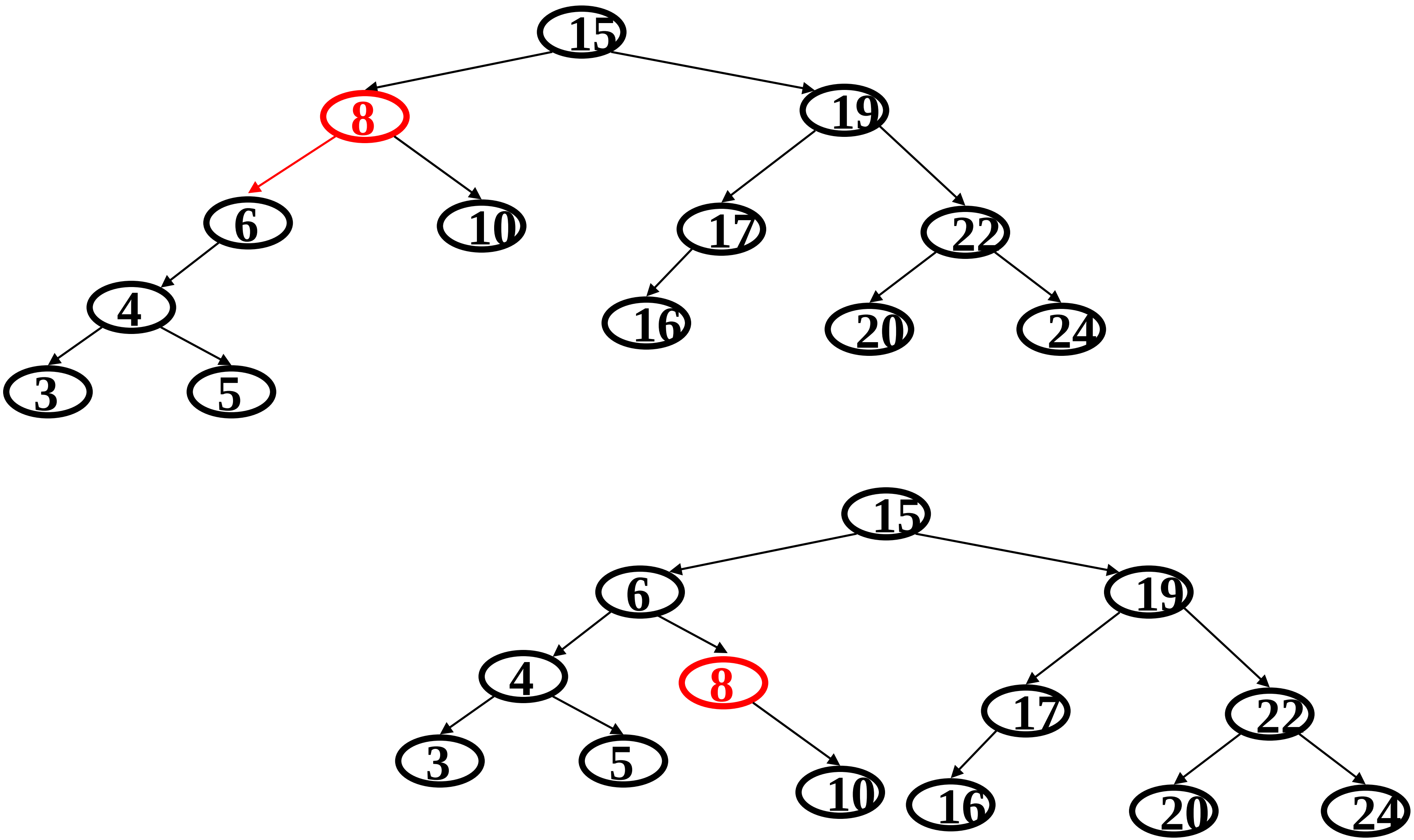


Double Rotation Example: Insert(5)





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`**: $O(\texttt{log}\ n)$

Worst-case complexity of **`insert`**: $O(\texttt{log}\ n)$

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

Worst-case complexity of **`buildTree`**: $O(n\ \texttt{log}\ n)$

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

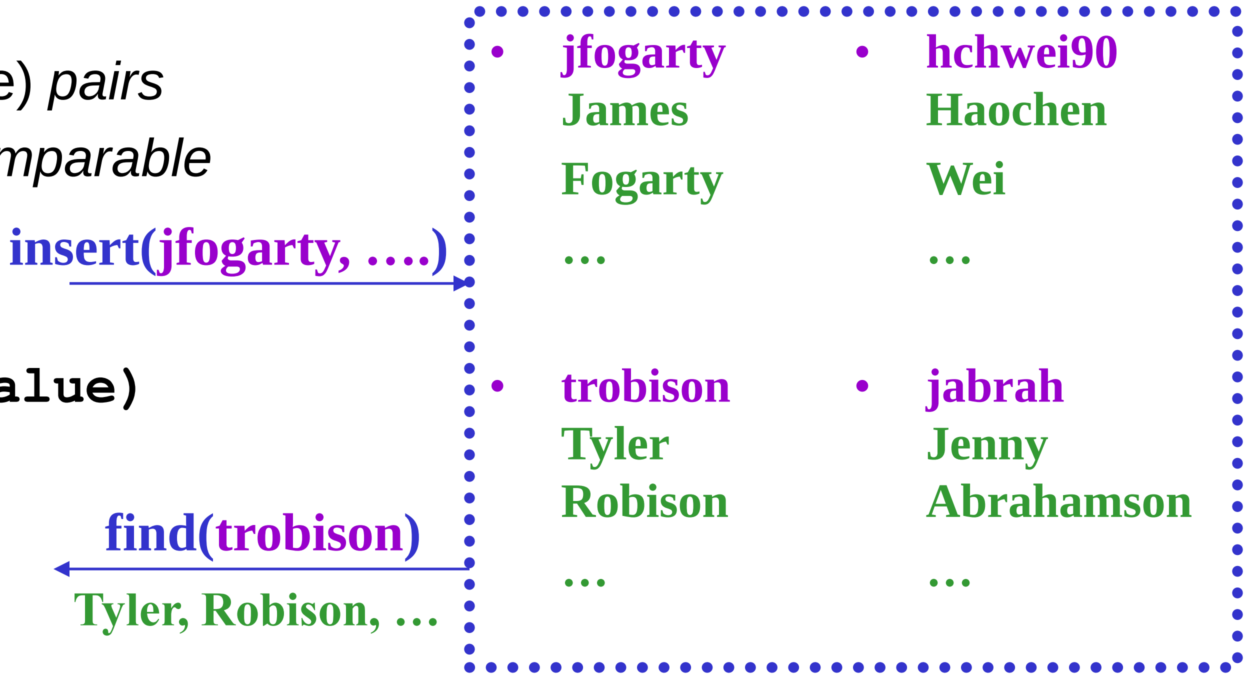
Lecture 7: B Trees

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)`**
  - …



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*

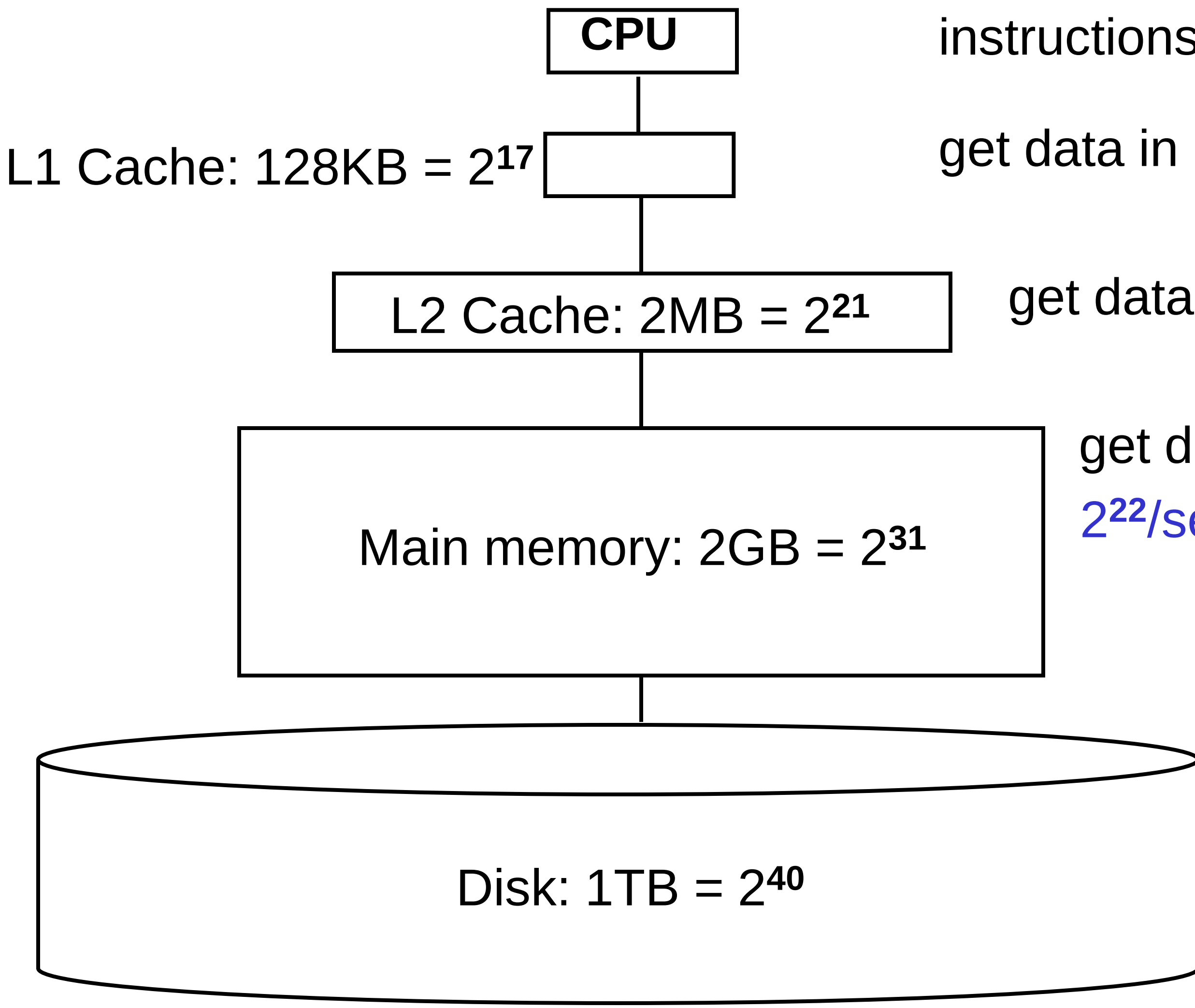
2. B-Trees
   – Also always balanced, but different and shallower

3. Hashtables
   – Not tree-like at all

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

A Typical Hierarchy

A plausible configuration …



instructions (e.g., addition): 2^{30}/sec

get data in L1: 2^{29}/sec = 2 insns

get data in L2: 2^{25}/sec = 30 insns

get data in main memory:
2^{22}/sec = 250 insns

get data from
"new place" on disk:
2^{7}/sec =8,000,000 insns

"streamed": 2^{18}/sec

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

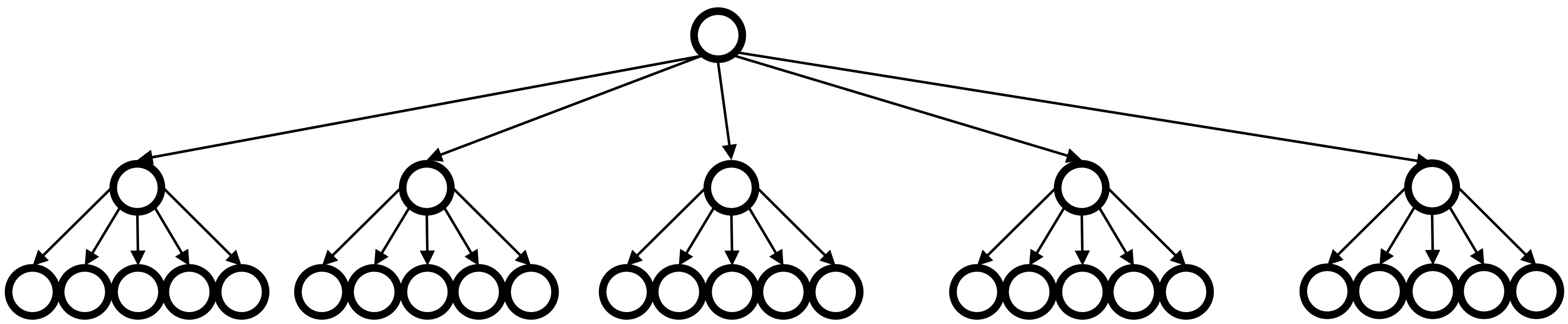
- Moving data up the memory hierarchy is slow because of *latency*
  - Might as well send more, just in case
  - Send nearby memory because:
    - It is easy, we are here anyways
    - And likely to be asked for soon (locality of reference)

- 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

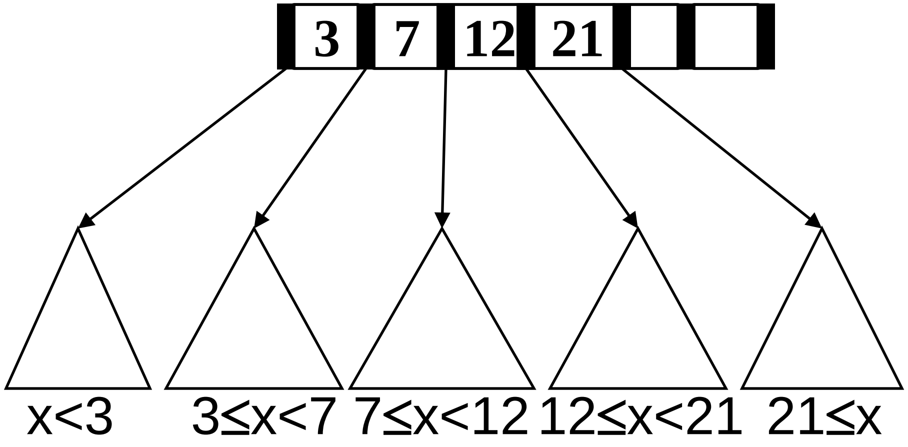
- What should the order property be?
- How would you rebalance (ideally without more disk accesses)?
- Any “useful” data at the internal nodes takes up disk-block space without being used by finds moving past it

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")*

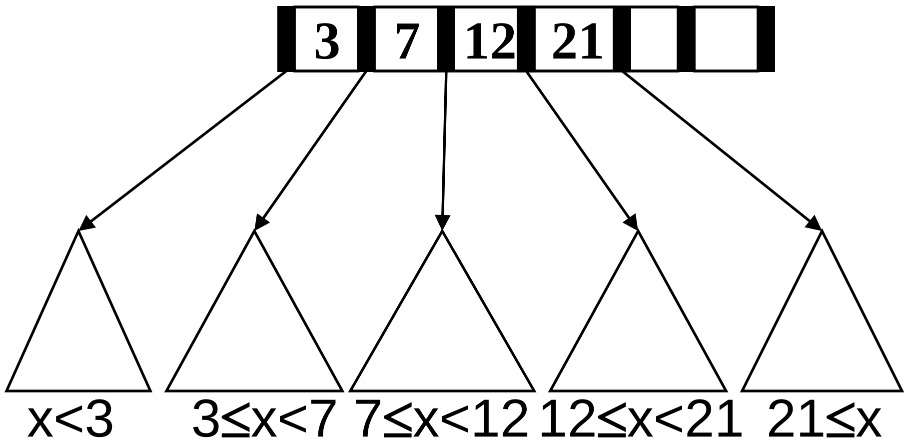
- Two types of nodes:
  - internal nodes and leaf nodes
- Each internal node has room for up to *M-1* keys and *M* children
  - no data; all data at the leaves!
- Order property:
  - Subtree between *x* and *y*
    - Data that is ≥ ***x*** and < ***y***
  - Notice the ≥
- Leaf has up to *L* sorted data 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 ≤ M-1 keys
  - At the leaf do binary search on the ≤ L data items
- To get logarithmic running time, we need a balance condition

Structure Properties

- **Root** (special case)
  - If tree has ≤ *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)

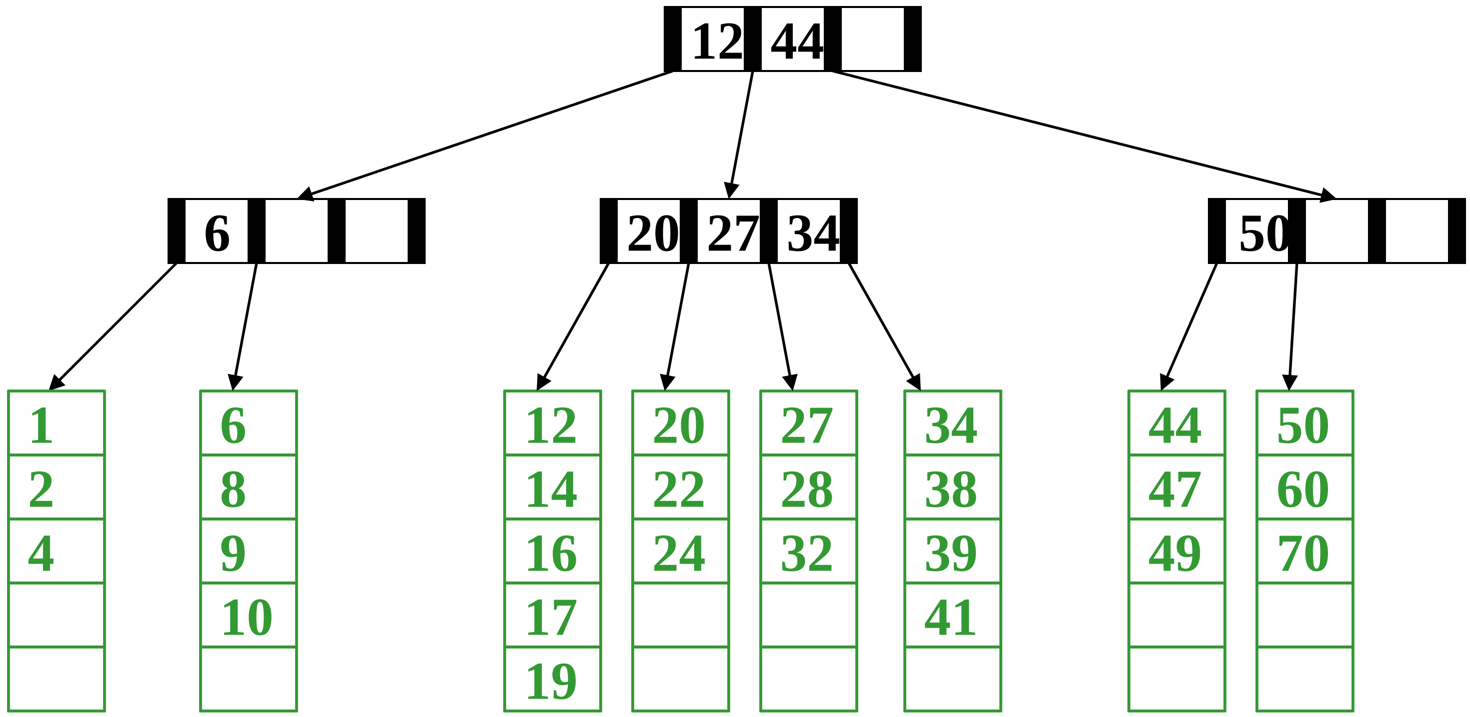
(Any *M* > 2 and *L* will work; ***picked based on disk-block size***)

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…

$$n \geq 2 \lceil M/2 \rceil^{h-1} \lceil L/2 \rceil$$

$2 \lceil M/2 \rceil^{h-1}$: minimum number of leaves

$\lceil L/2 \rceil$: minimum data per leaf

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

Disk Friendliness

What makes B trees so disk friendly?

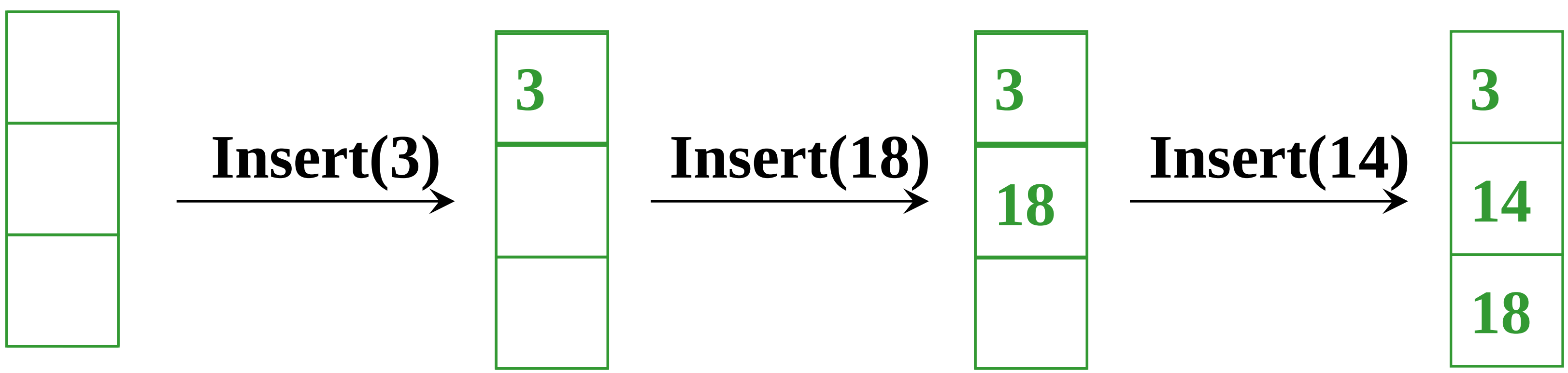
- Many keys stored in one **internal node**
  - All brought into memory in one disk access
  - But only if we pick *M* wisely
  - Makes the binary search over *M*-1 keys totally worth it (insignificant compared to disk access times)

- **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

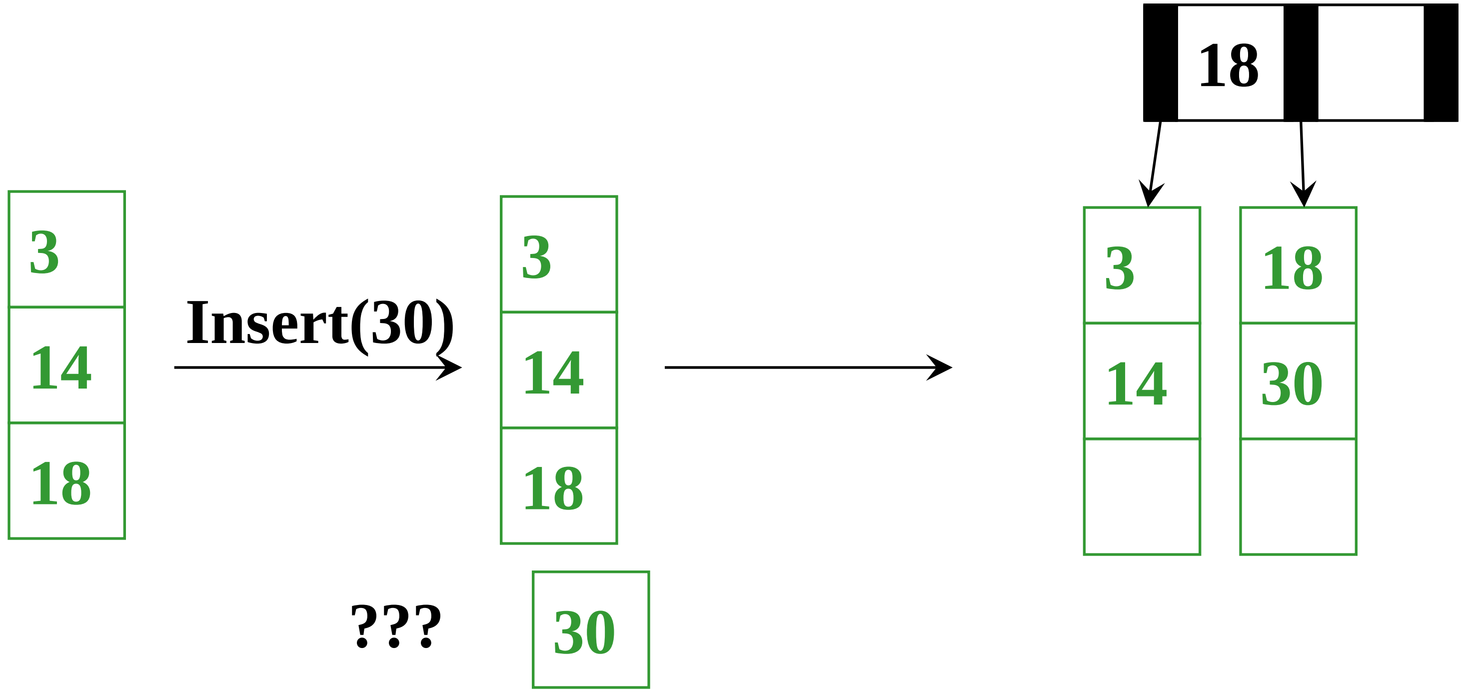


The empty B-Tree (the root will be a leaf at the beginning)

Simply need to keep data sorted

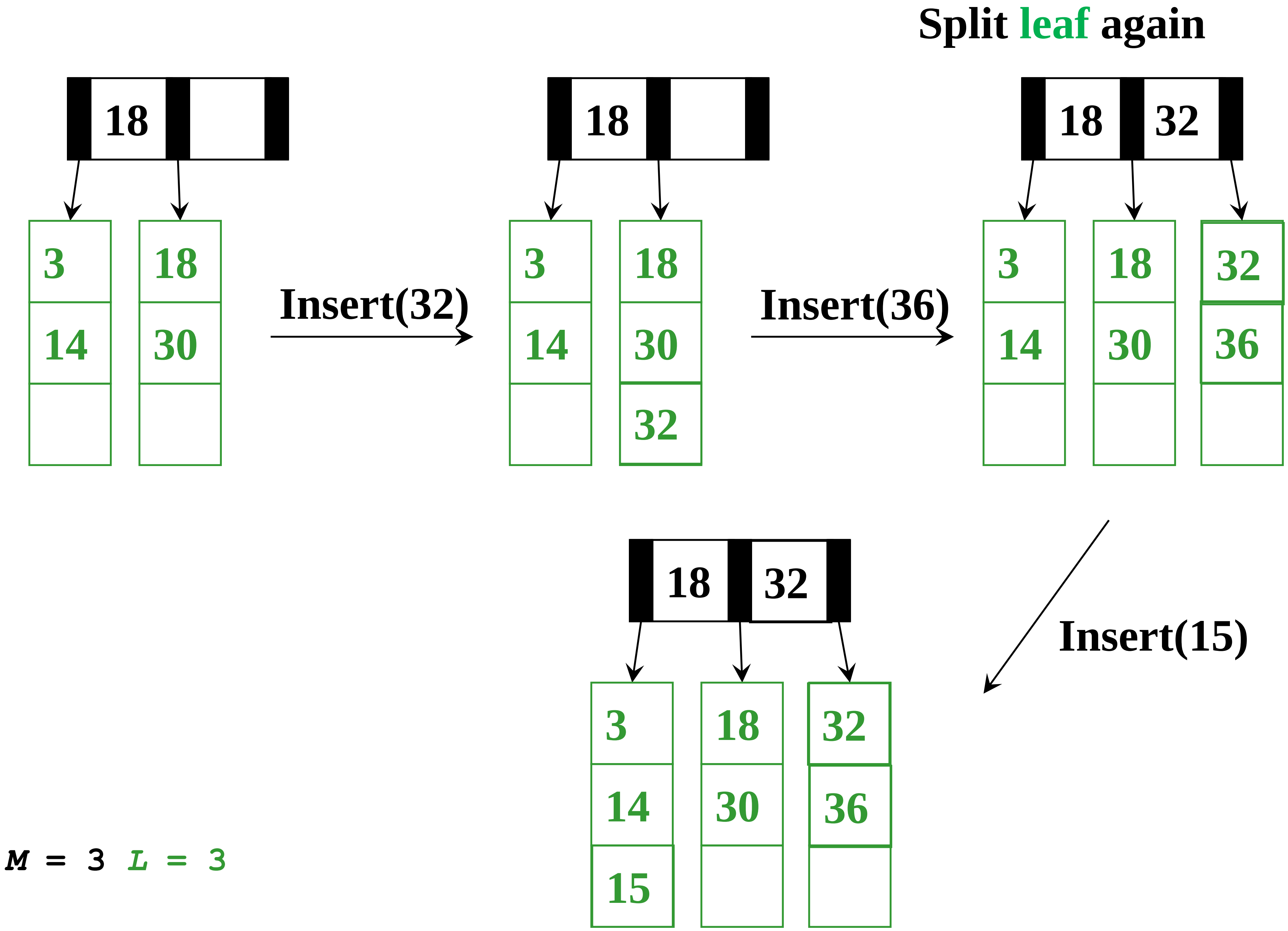
M **= 3** ***L*** **= 3**

$M = 3$ $L = 3$

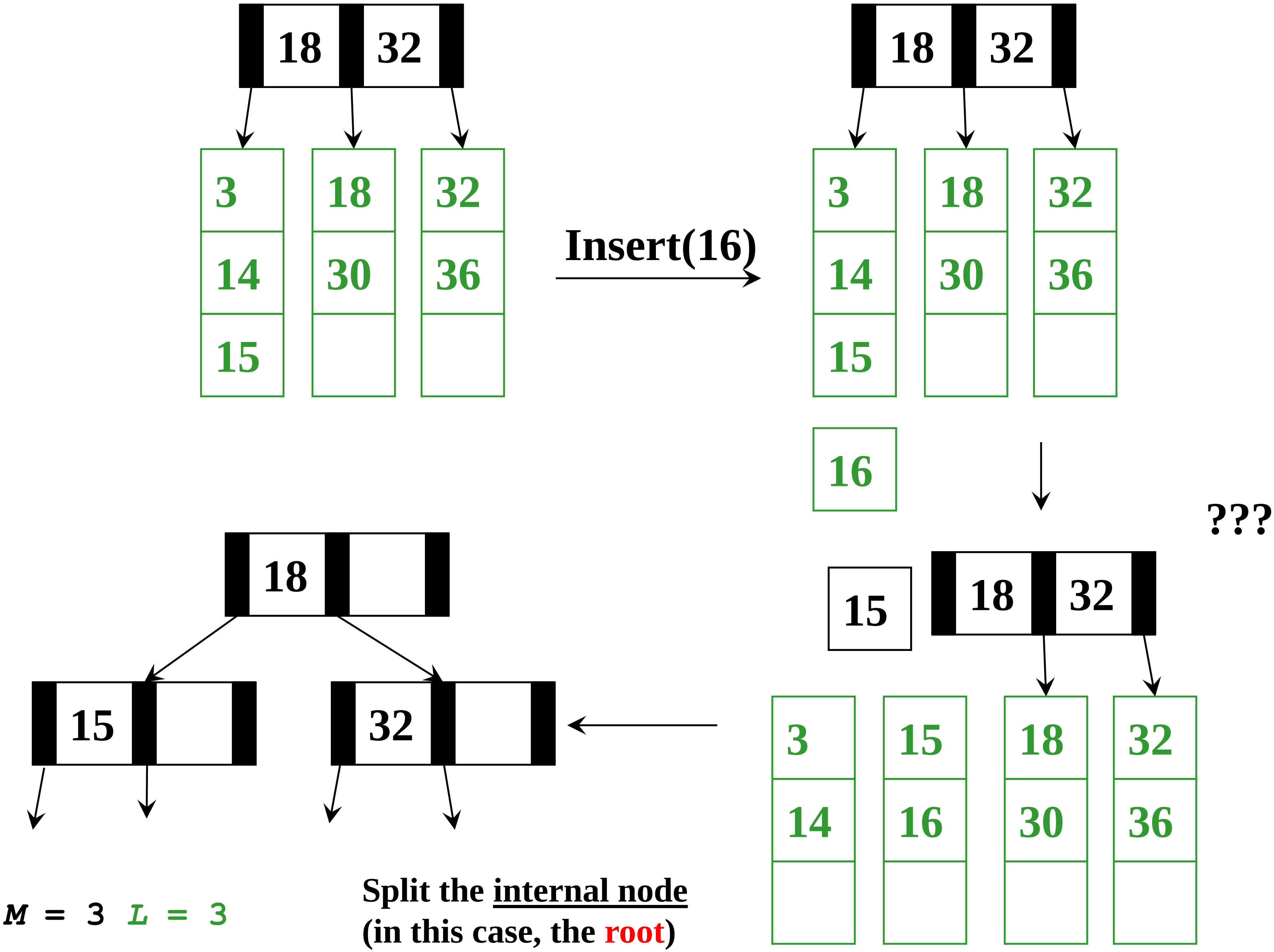


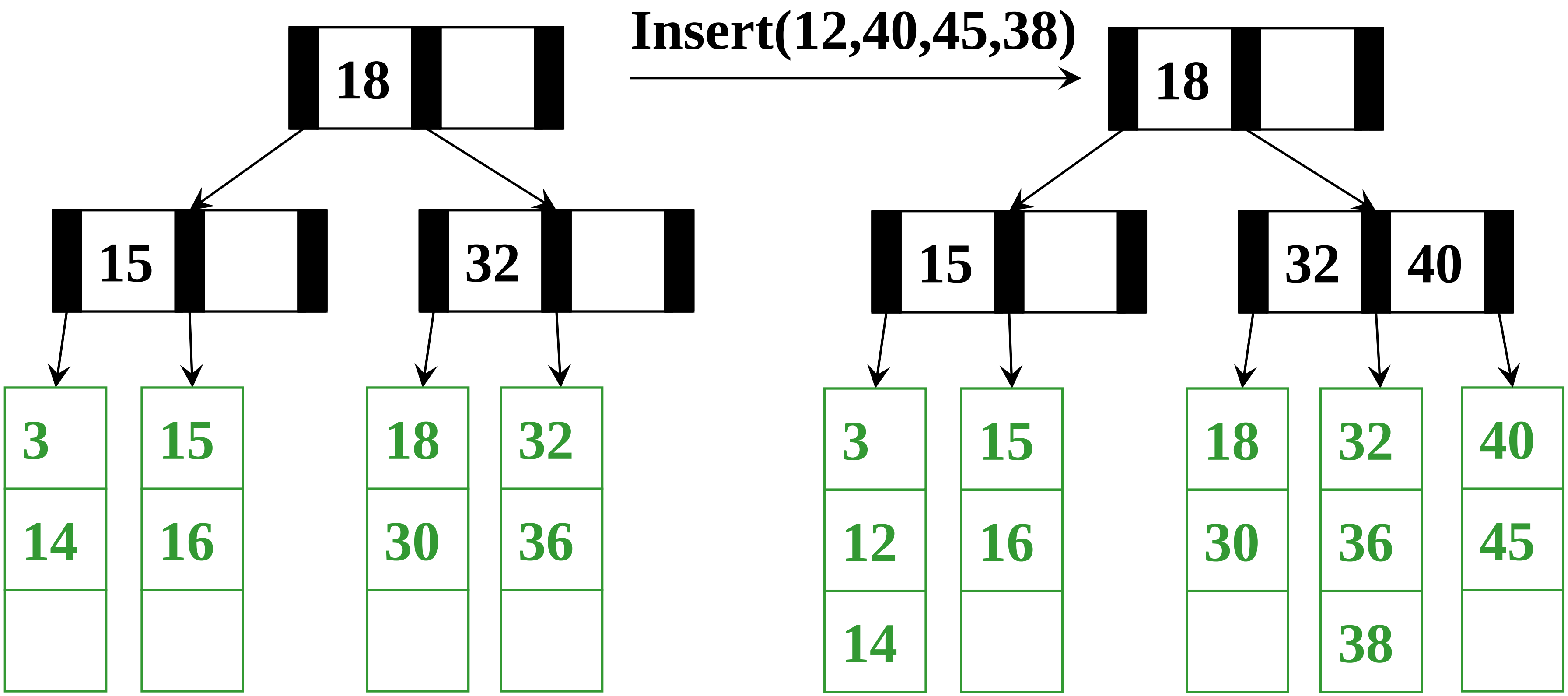
- When we ‘overflow’ a leaf, we split it into 2 leaves
- Parent gains another child
- If there is no parent, we create one

- How do we pick the new key?
  - Smallest element in right tree



Split leaf again
18
3
14
18
30
Insert(32)
18
3
14
18
30
32
Insert(36)
18
32
3
14
18
30
32
36
Insert(15)
18
32
3
14
15
18
30
32
36
M = 3 L = 3


18 32
3
14
15
18
30
32
36
Insert(16)
18 32
3
14
15
18
30
32
36
16
???
15
18 32
3
14
15
16
18
30
32
36
18
15
32
M = 3 L = 3
Split the internal node
(in this case, the root)



M = 3 *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 $\lceil \mathbf{(M+1)/2} \rceil$ smaller items
     - New **node** with $\lfloor \mathbf{(M+1)/2} \rfloor = \lceil \mathbf{M/2} \rceil$ 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

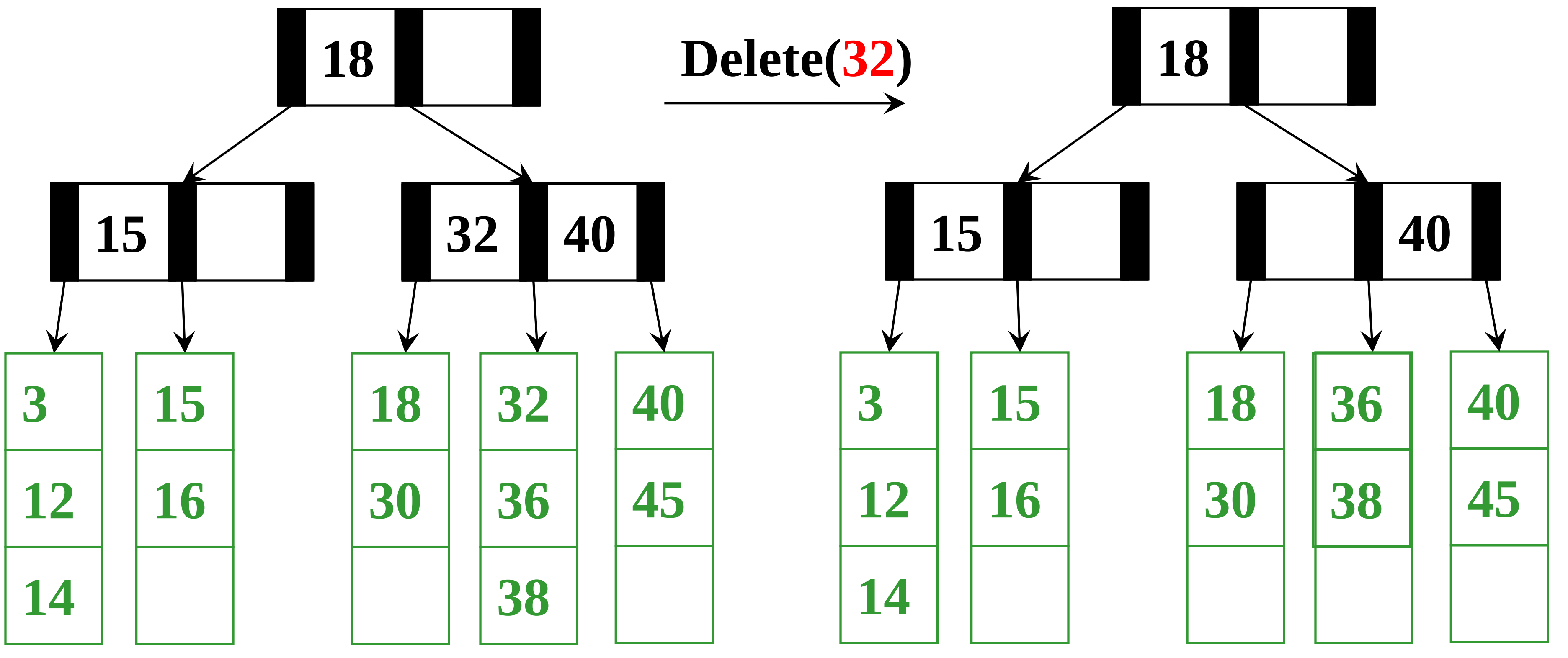
- Find correct leaf: $O(\log_2 M \log_M n)$
- Insert in leaf: $O(L)$
- Split leaf: $O(L)$
- Split parents all the way up to root: $O(M \log_M n)$

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

But it's not that bad:

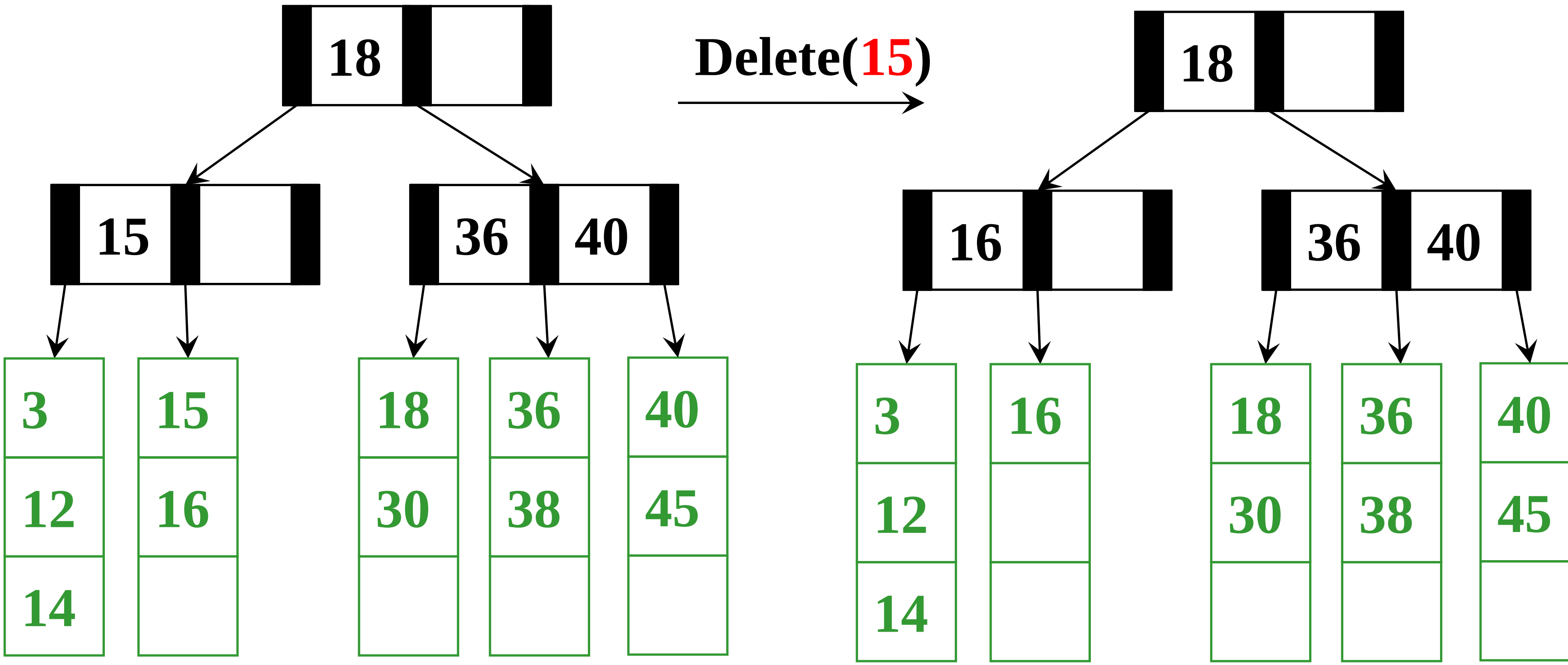
- Splits are not that common (only required when a node is FULL, M and L are likely to be large, and after a split will be half empty)
- Splitting the **root** is extremely rare
- Remember disk accesses is name of the game: $O(\log_M n)$

Deletion



Let them eat cake!

M = 3 *L* = 3

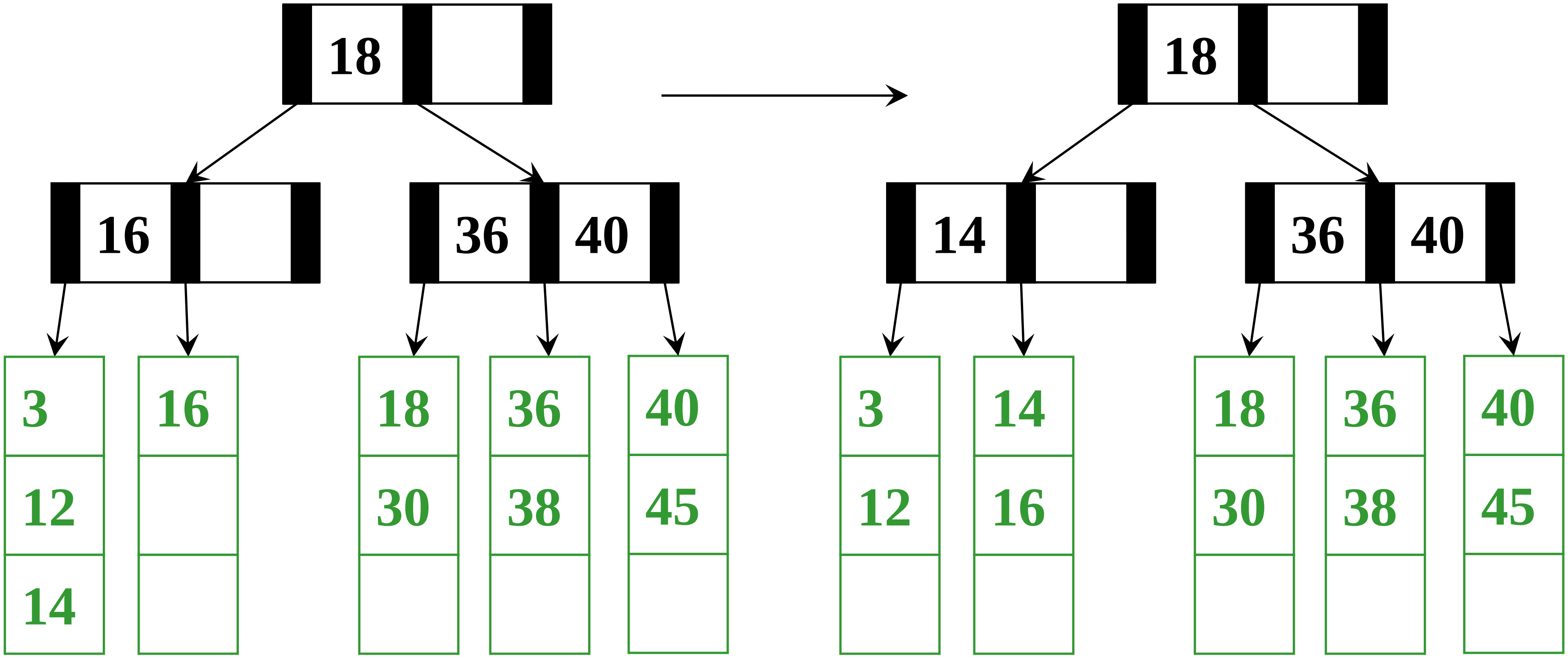


Are we okay?

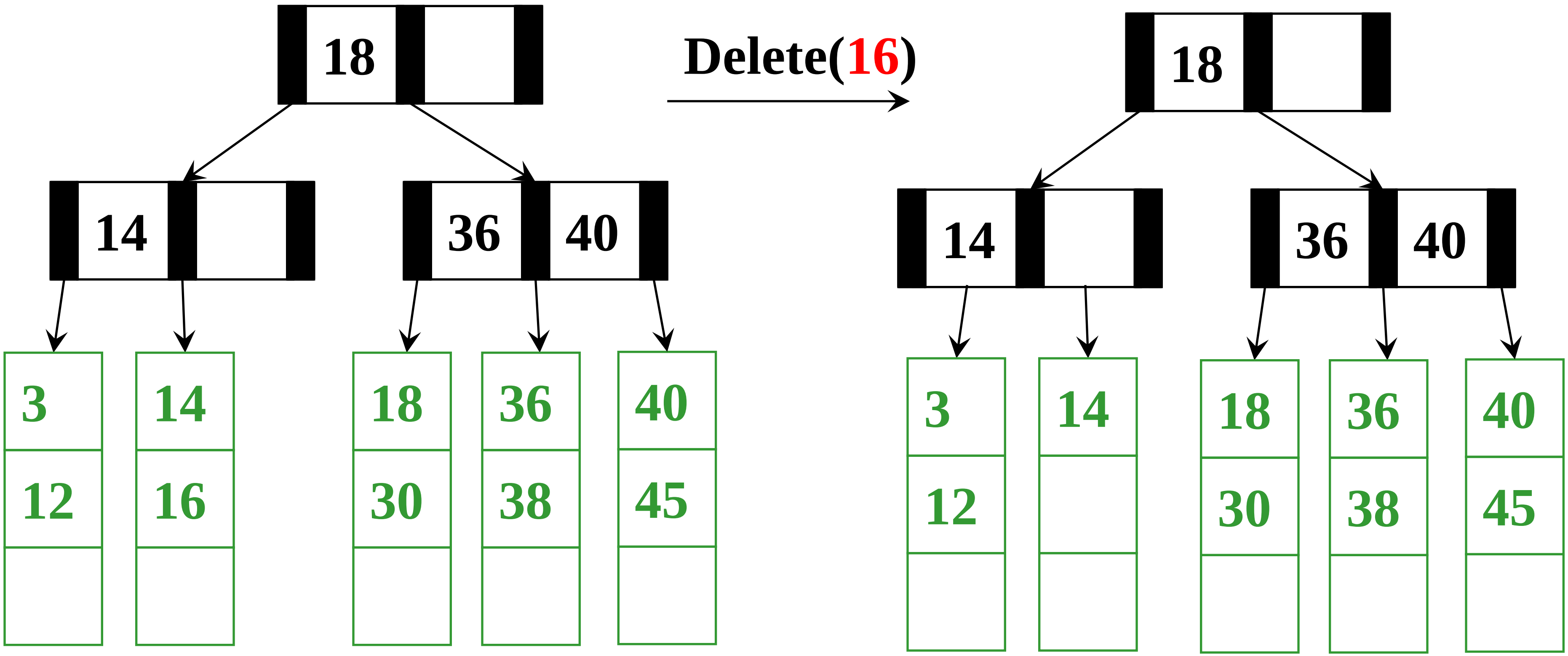
Are you using that 14?
Can I borrow it?

Dang, not half full

M = 3 *L* = 3


18
16
36
40
3
12
14
16
18
30
36
38
40
45
18
14
36
40
3
12
14
16
18
30
36
38
40
45

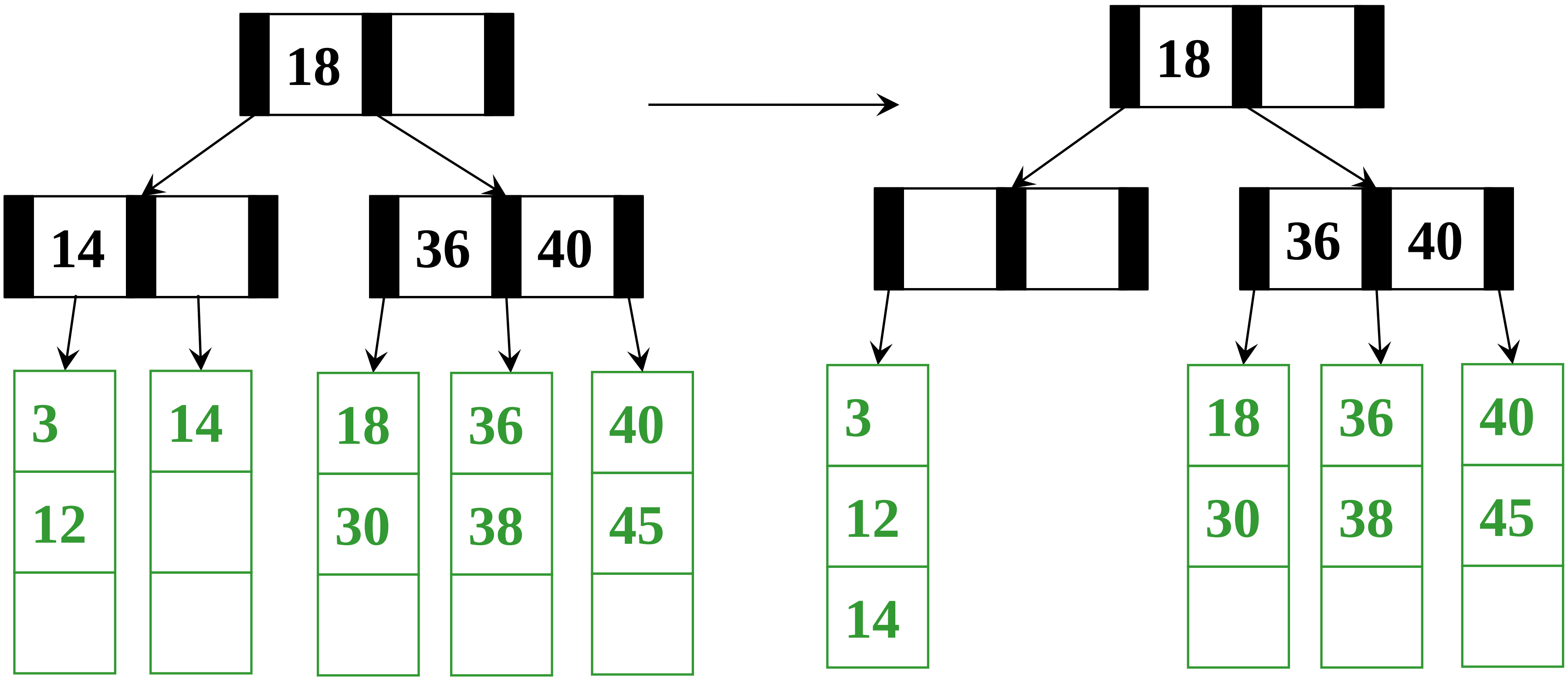
M = 3 *L* = 3



Are you using that 12?

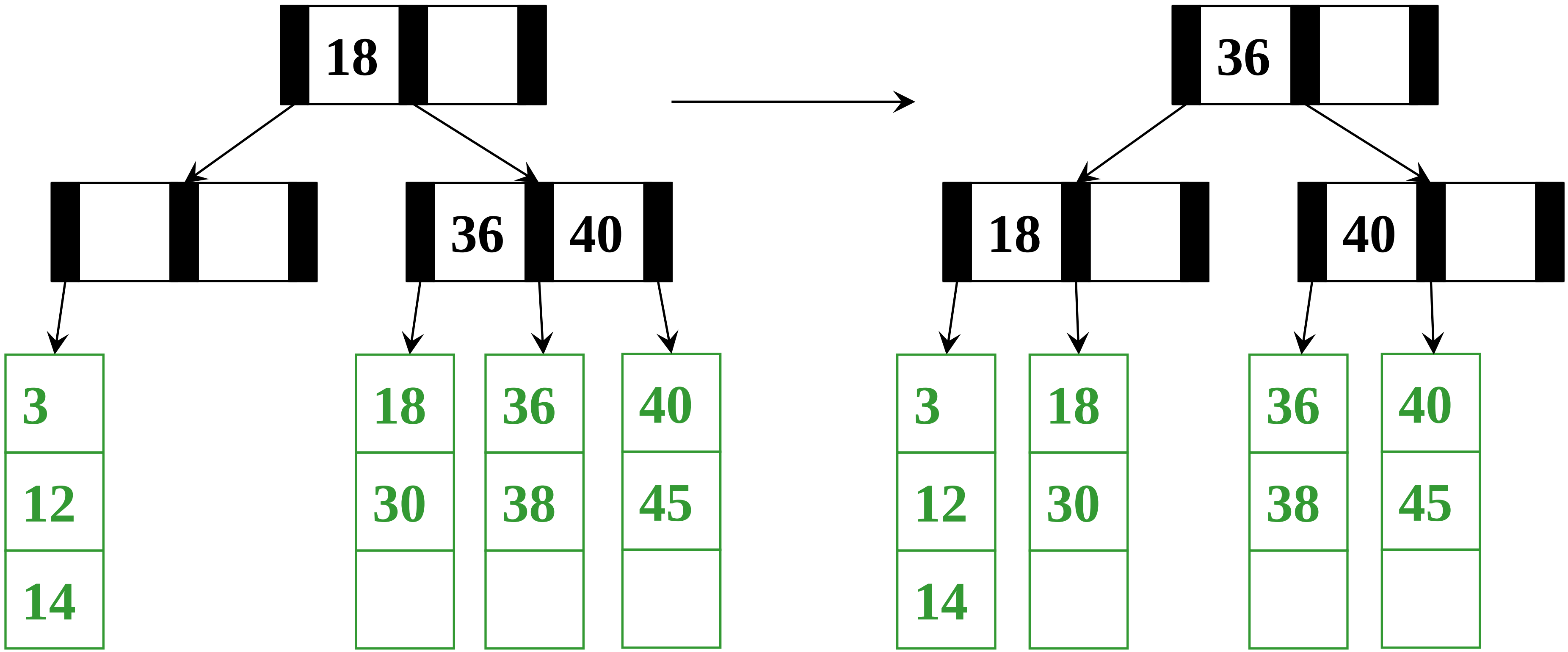
Are you using that 18?

`M = 3 L = 3`

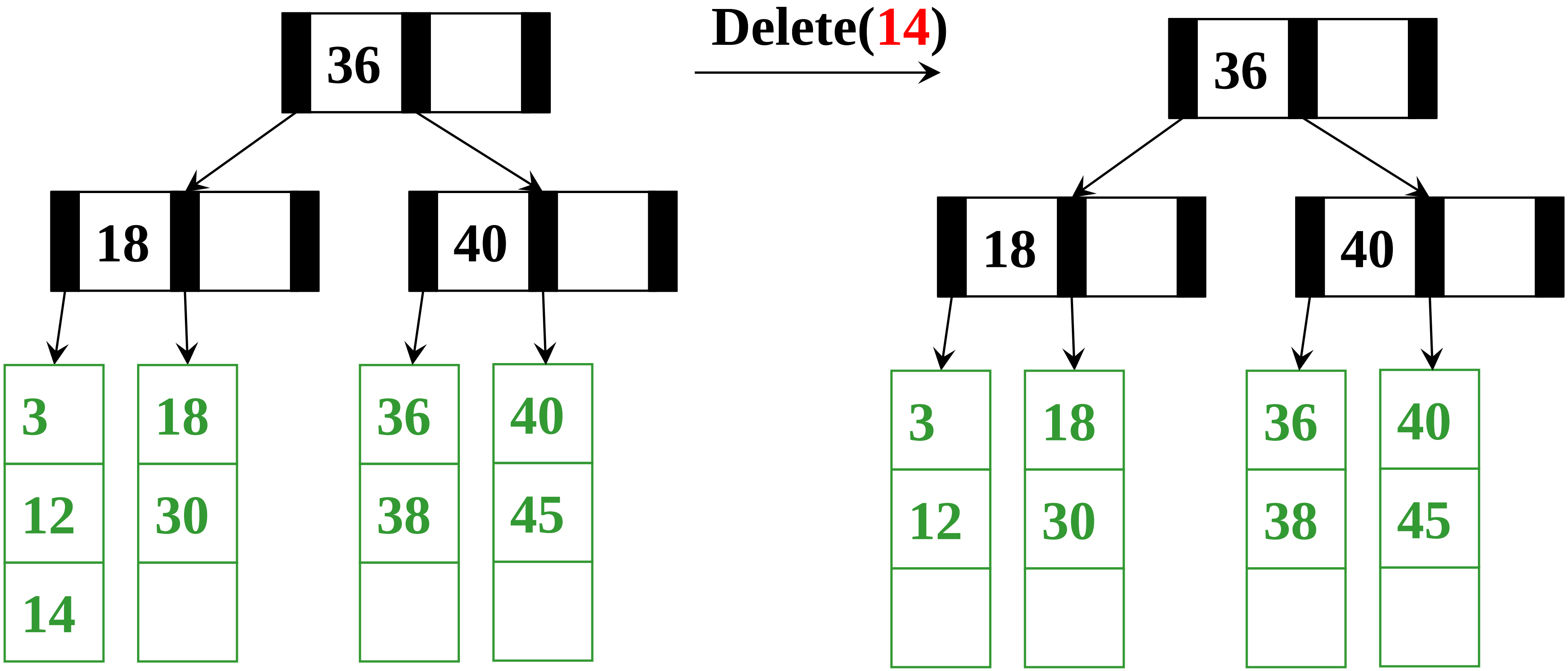


Are you using that 18/30?

$M = 3$ $L = 3$

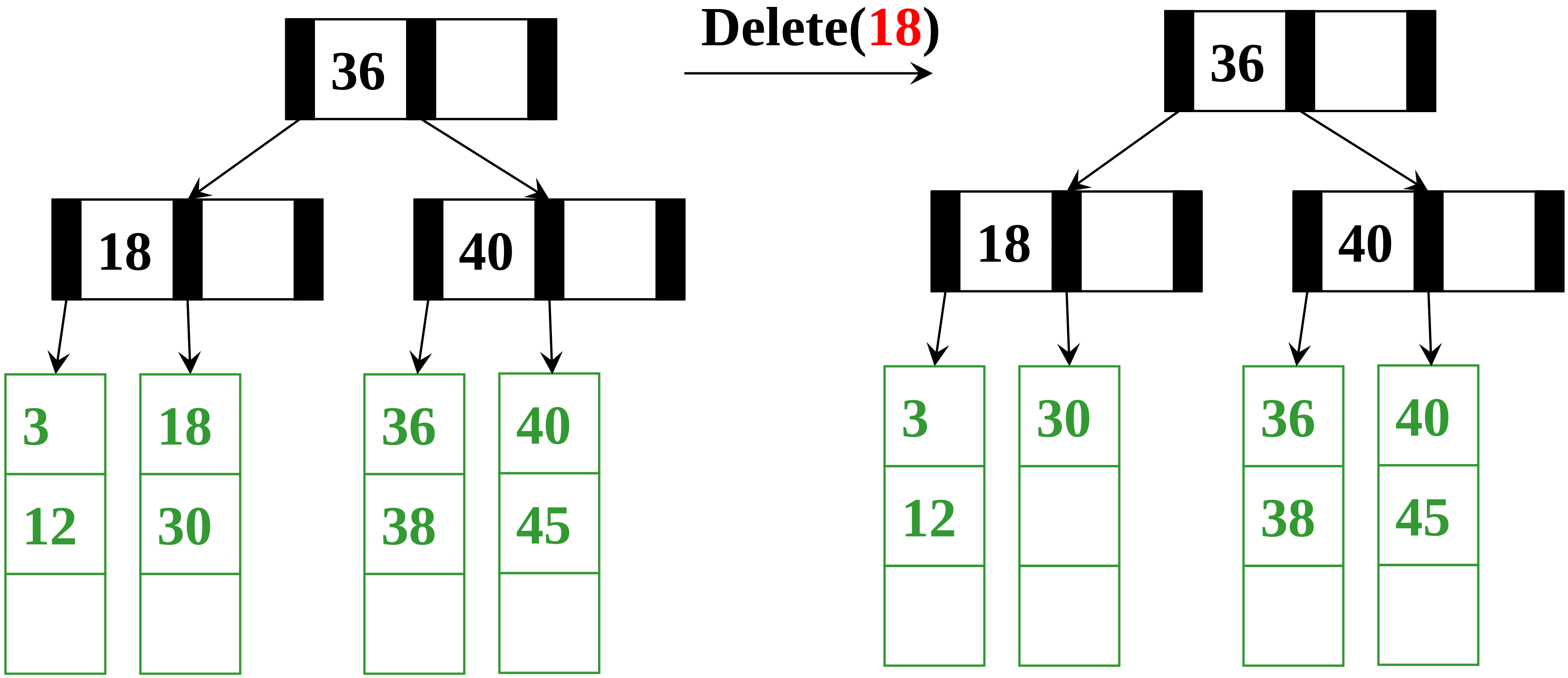


M = 3 *L* = 3

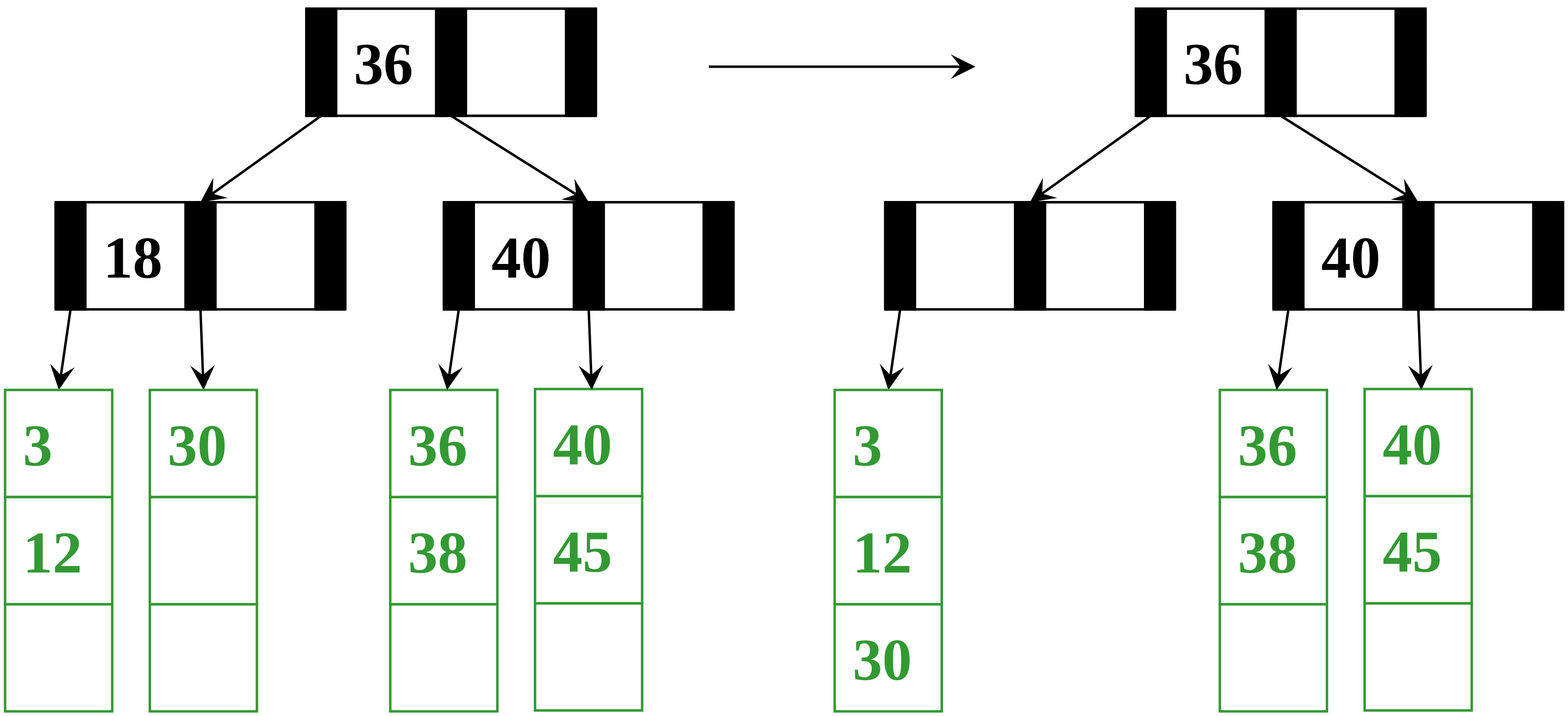

Delete(14)
36
18
40
3
12
14
18
30
36
38
40
45
36
18
40
3
12
18
30
36
38
40
45


easy

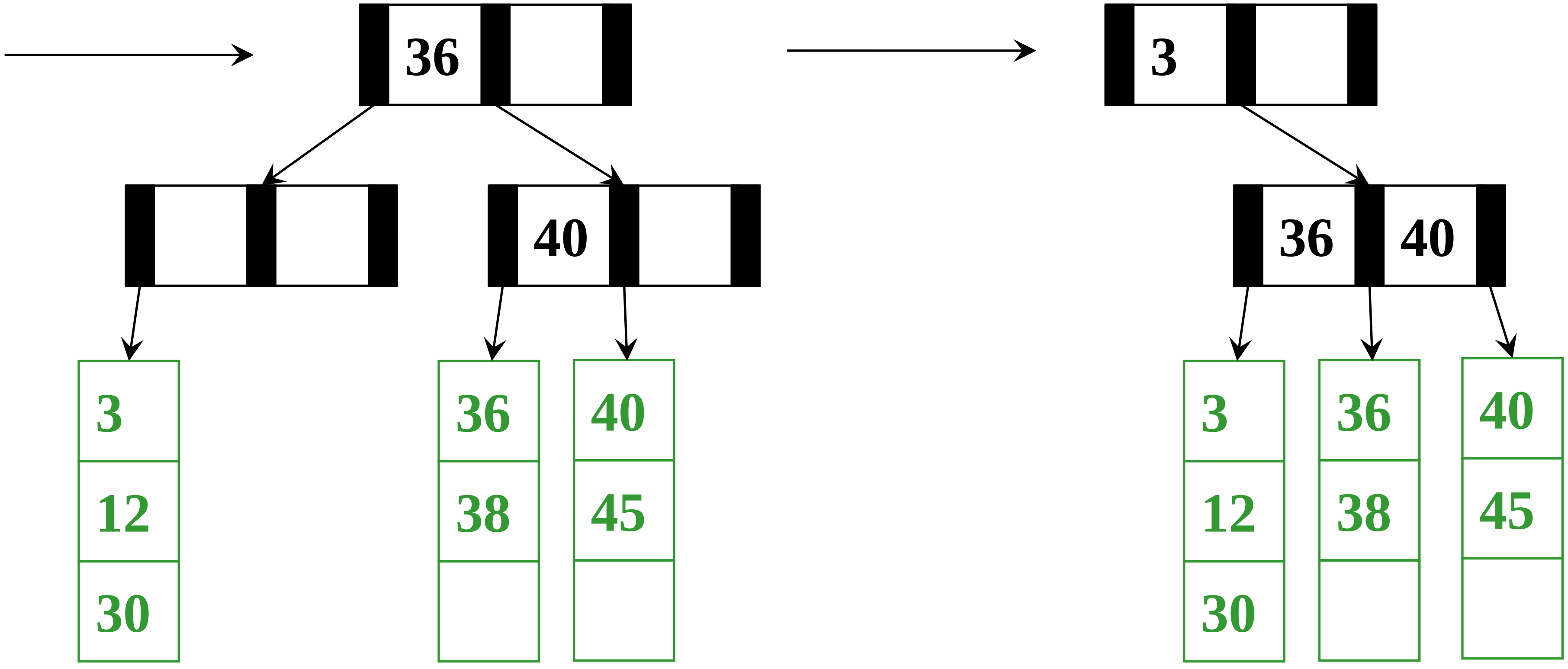
$M = 3$ $L = 3$



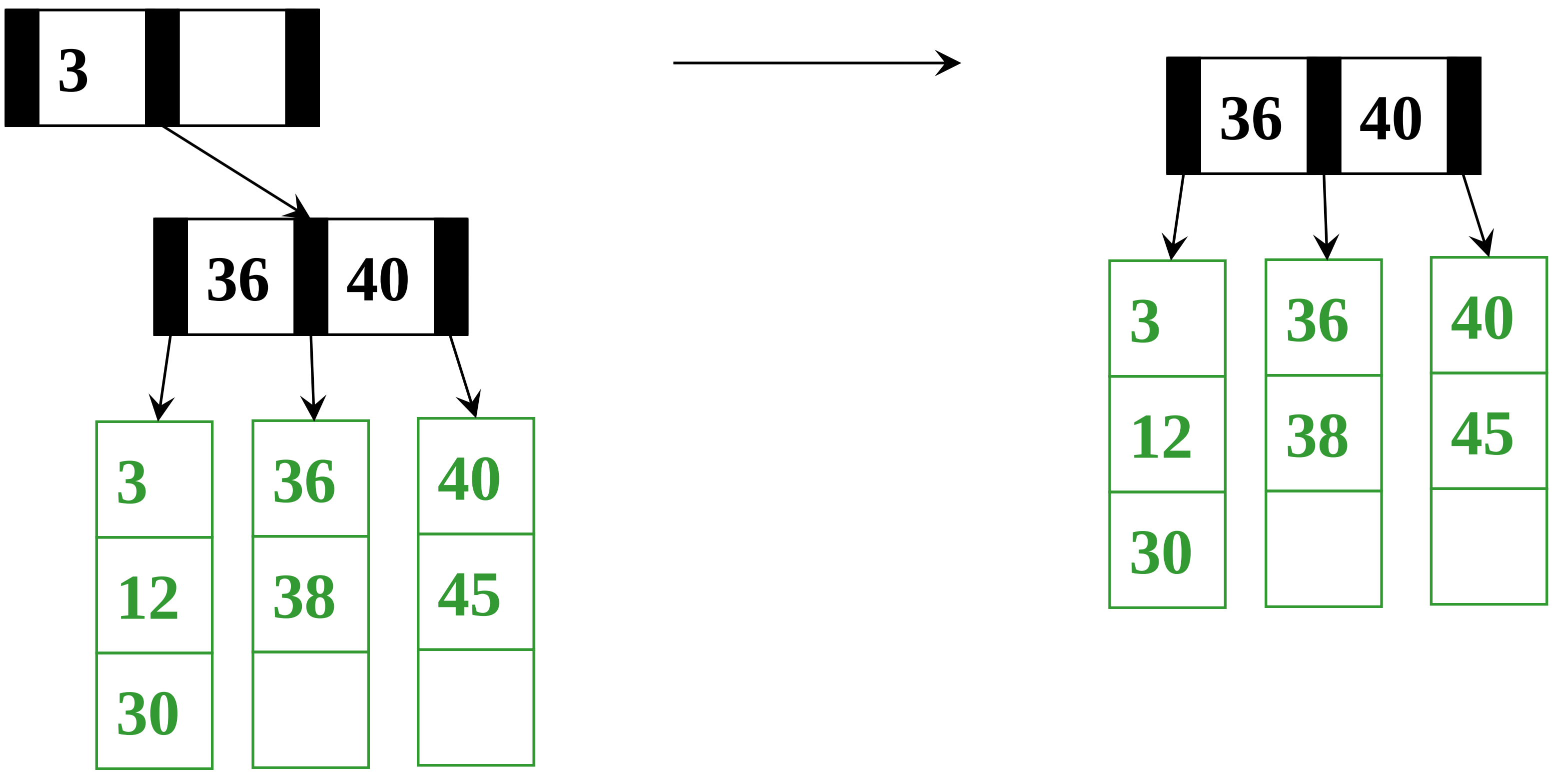
M = 3 *L* = 3



M = 3 L = 3



M = 3 **L** = 3



$M = 3$ $L = 3$

Deletion Algorithm

1. Remove the data from its leaf

2. If the leaf now has $\lceil \boldsymbol{L/2} \rceil$ **- 1**, *underflow!*
   - If a neighbor has > $\lceil \boldsymbol{L/2} \rceil$ 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 $\lceil \boldsymbol{M/2} \rceil$ **- 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:

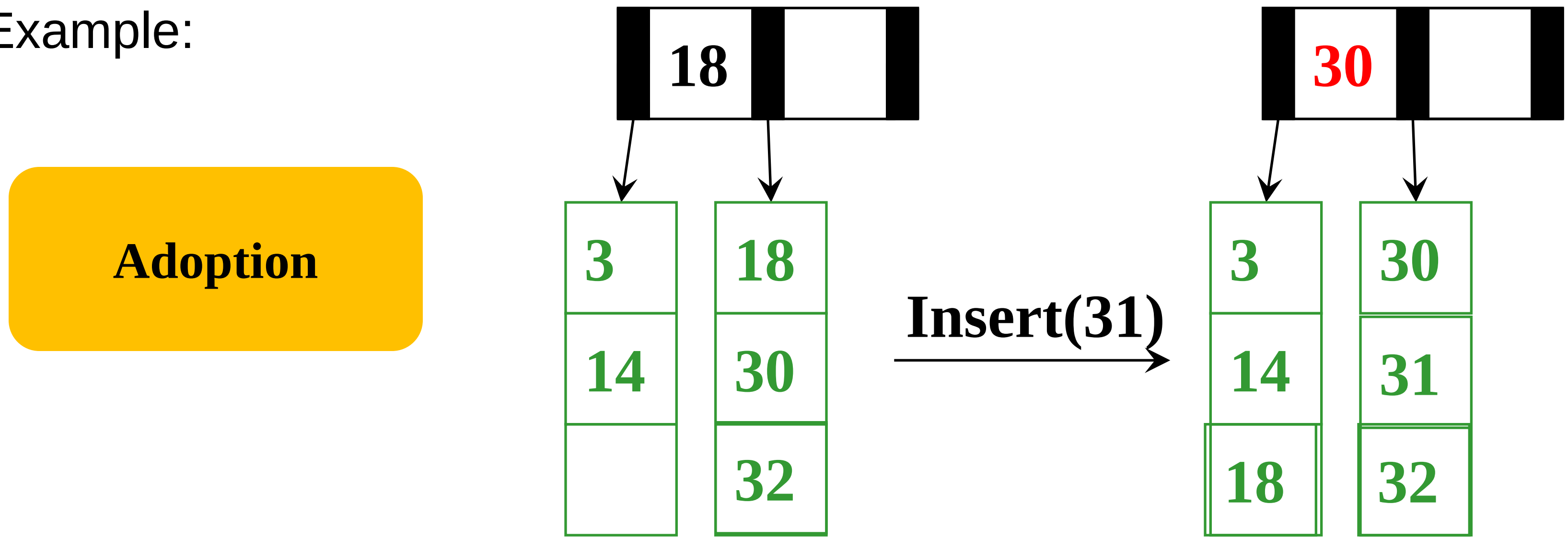
- Merges are not that common
- Remember disk access is the name of the game: $O(\log_M n)$

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:



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

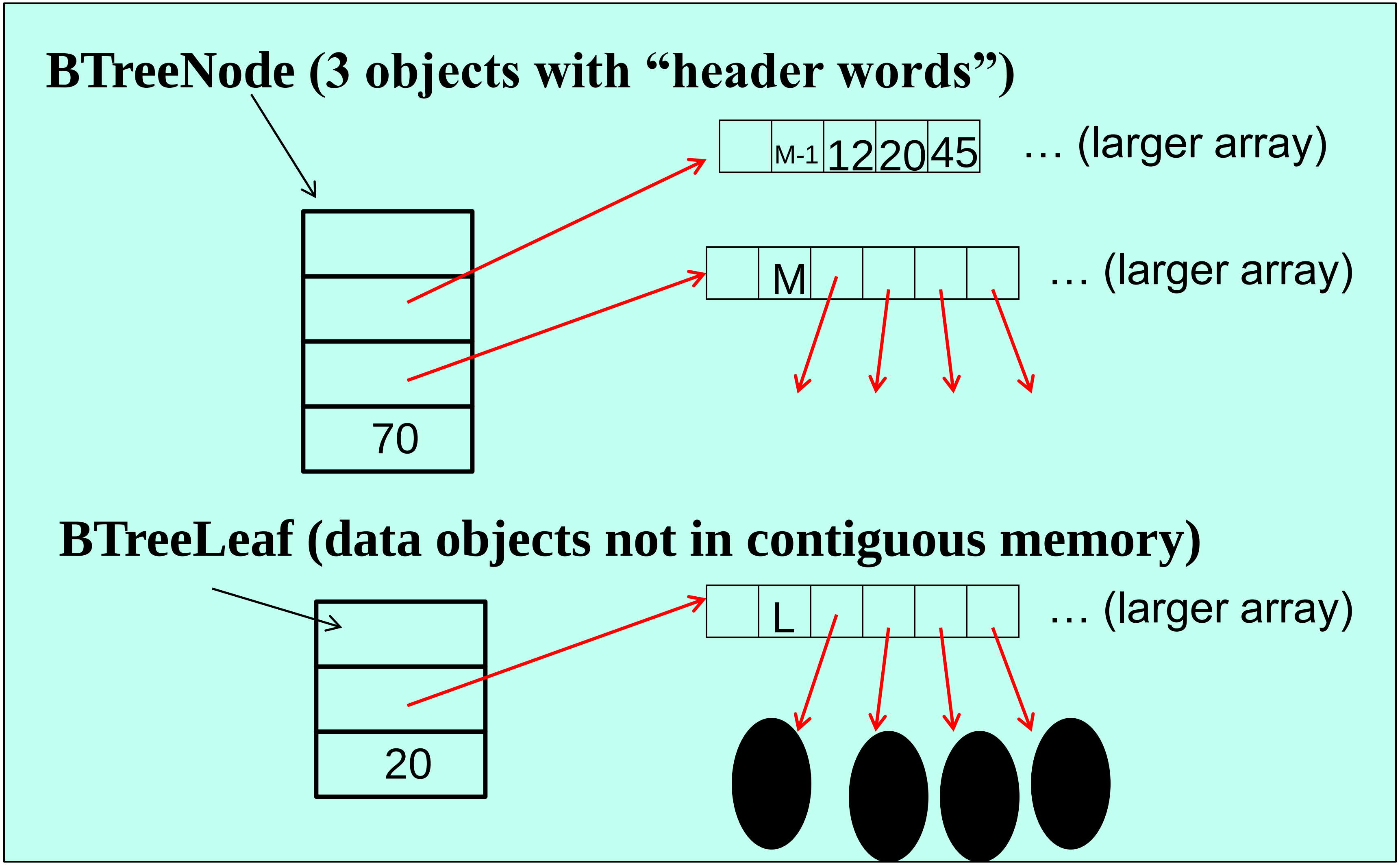
The key issue is extra *levels of indirection*…

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



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

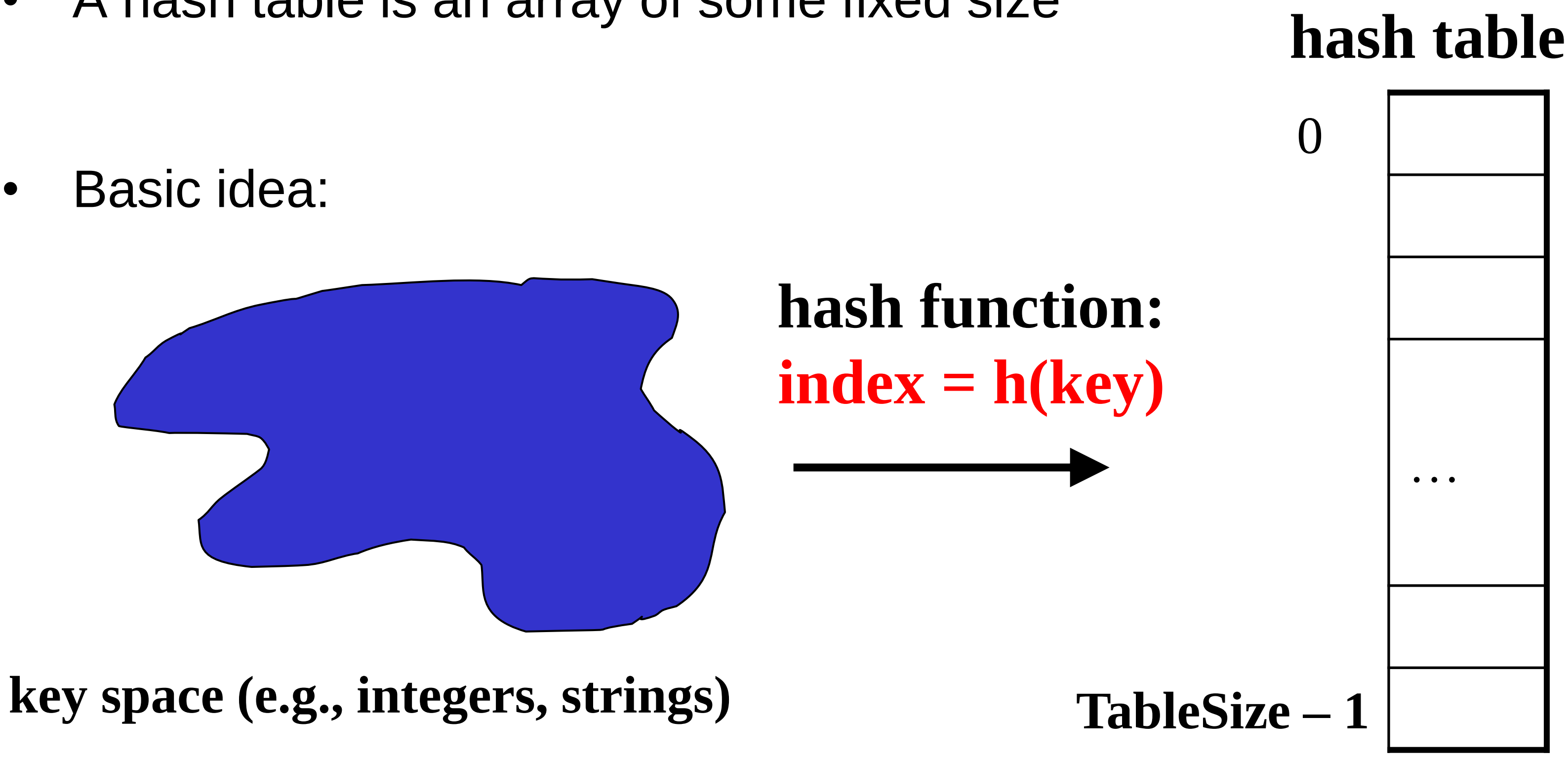
Simple Implementations

For dictionary with *n* key/value pairs

	`insert`	`find`	`delete`	
• 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* << *m*)

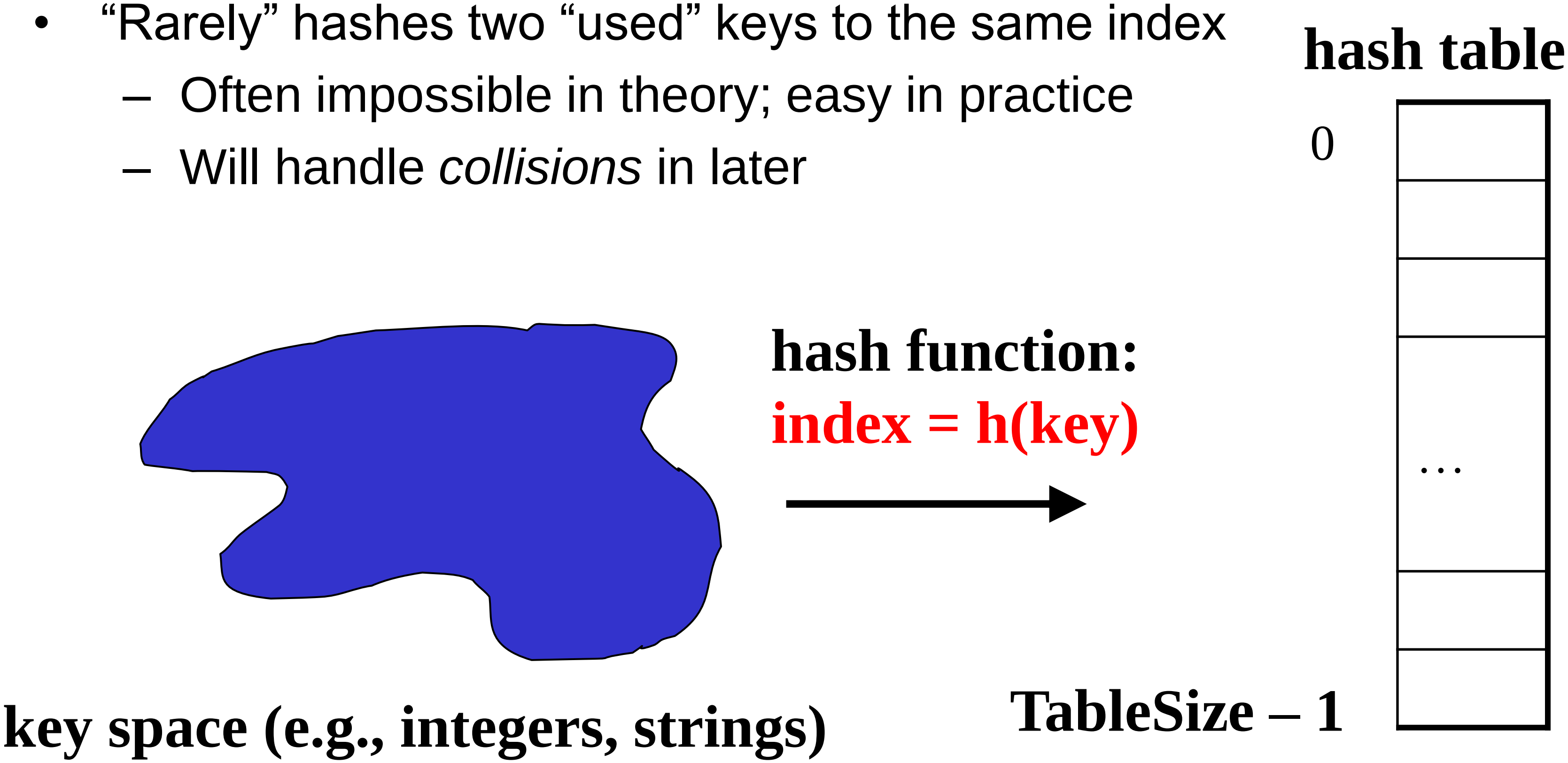
Many dictionaries have this property

- 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:

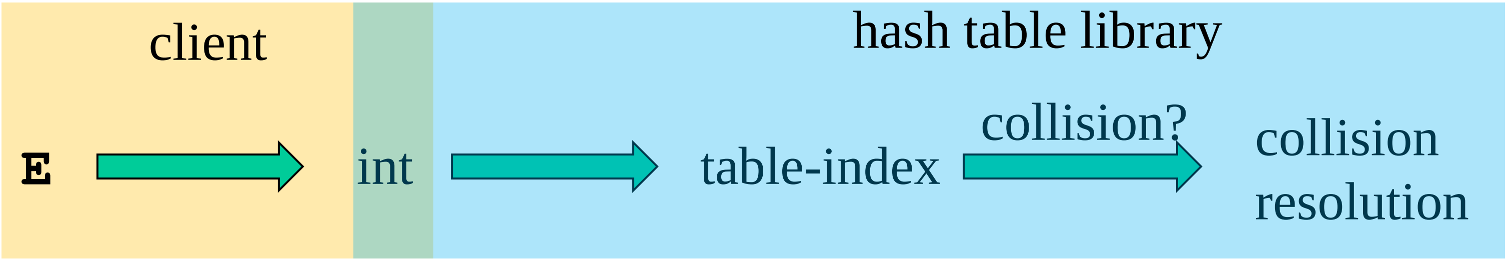
- Is fast to compute
- “Rarely” hashes two “used” keys to the same index
  - Often impossible in theory; easy in practice
  - Will handle *collisions* in later



Who Hashes What

- Hash tables can be generic
  - To store elements of type **`E`**, we just need **`E`** to be:
    1. Comparable: order any two **`E`** (as with all dictionaries)
    2. Hashable: convert any **`E`** to an **`int`**

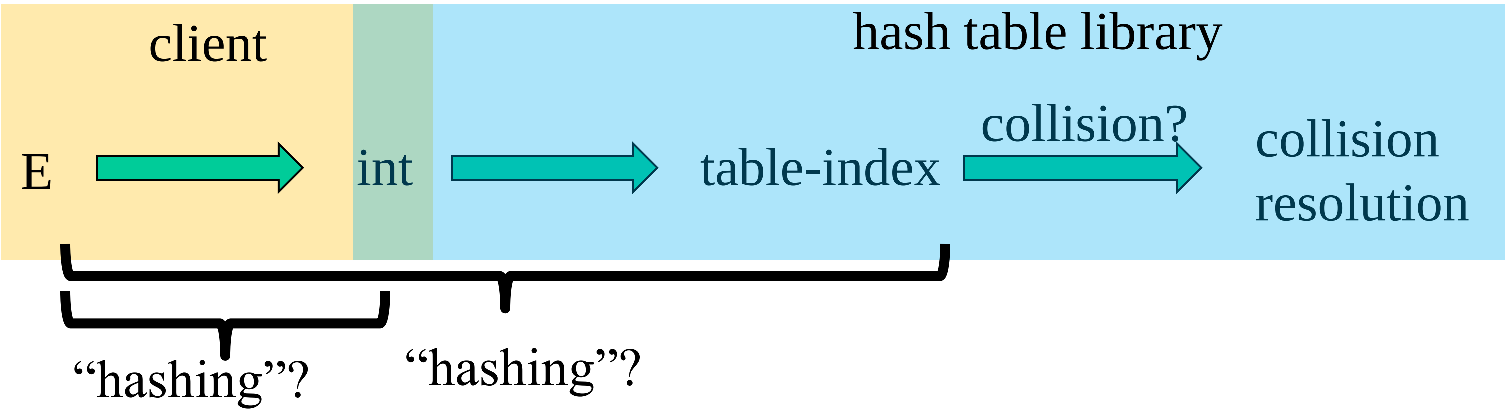
- When hash tables are a reusable library, the division of responsibility generally breaks down into two roles:



- 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" **`int`**s 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(key) = key % TableSize`

  – Client: **`f(x) = x`**

  – Library **`g(x) = f(x) % 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 **`gcd(x,y)==1`**),
then **`(a * x) % y == (b * x) % y`** if and only if **`a % y == b % 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 `int`s, the client must convert to an `int`
  - Trade-off: speed and distinct keys hashing to distinct `int`s

- Common and important example: Strings
  - Key space K = $s_0 s_1 s_2 \ldots s_{m-1}$
    - where s_i are chars: $s_i \in [0,256]$
  - Some choices: Which best avoid collisions?

1. h(K) = s_0 % TableSize
2. h(K) = $\left(\sum_{i=0}^{m-1} s_i \right)$ % TableSize
3. h(K) = $\left(\sum_{i=0}^{k-1} s_i \cdot 37^i \right)$ % 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

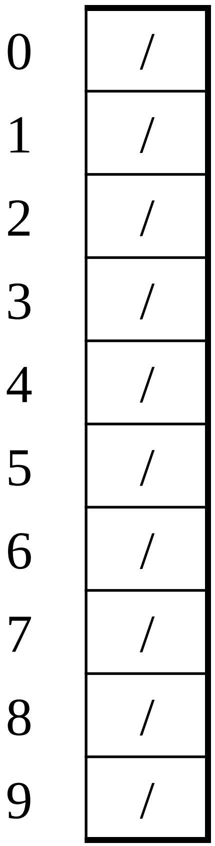
Collision:

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



Chaining:

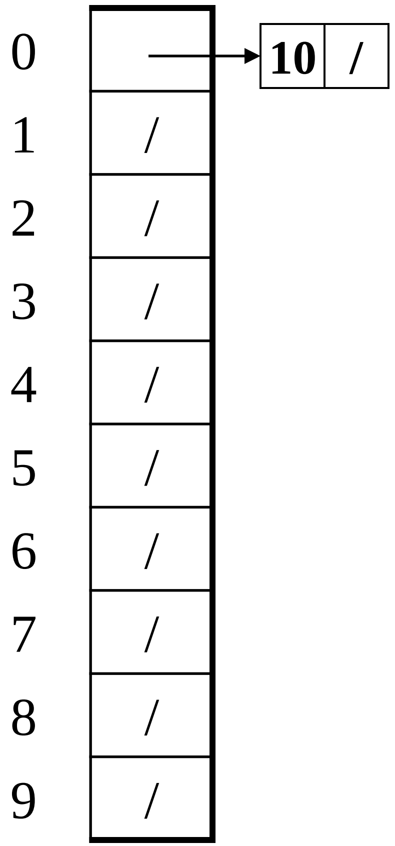
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

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

Separate Chaining



Chaining:

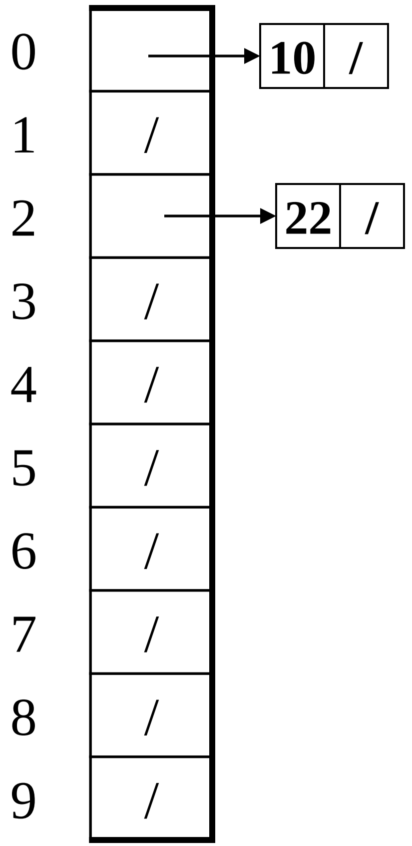
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

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

Separate Chaining



Chaining:

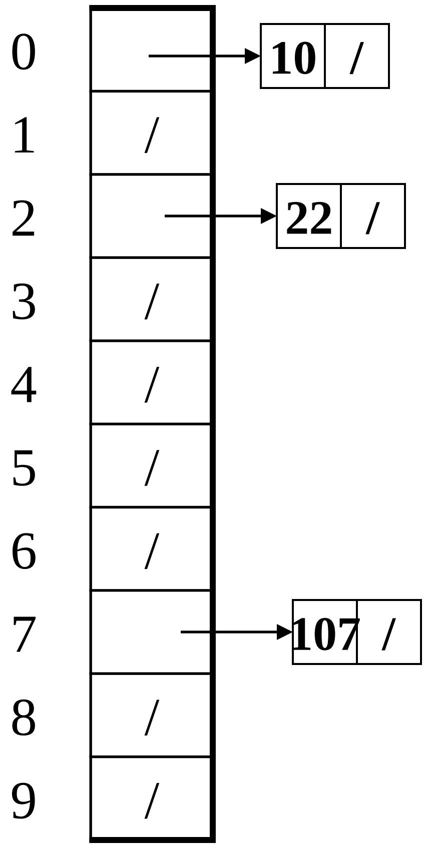
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

As easy as it sounds

Example:

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

Separate Chaining



Chaining:

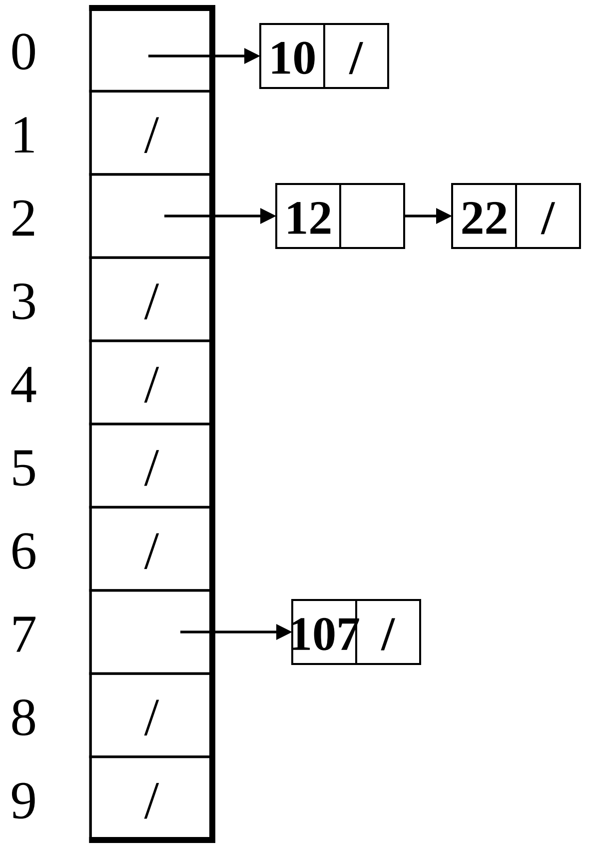
All keys that map to the same table location are kept in a list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

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

Separate Chaining

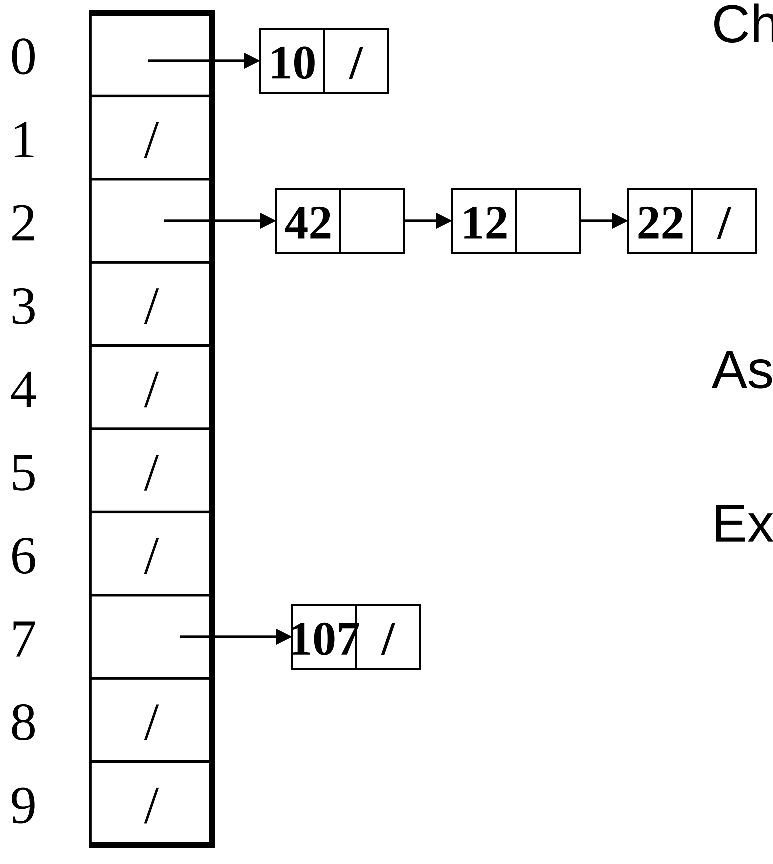


Chaining:
All keys that map to the same table location are kept in a list (a.k.a. a “chain” or “bucket”)

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, λ, of a hash table is

$$\lambda = \frac{\text{N}}{\text{TableSize}}$$ ← **number of elements**

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

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

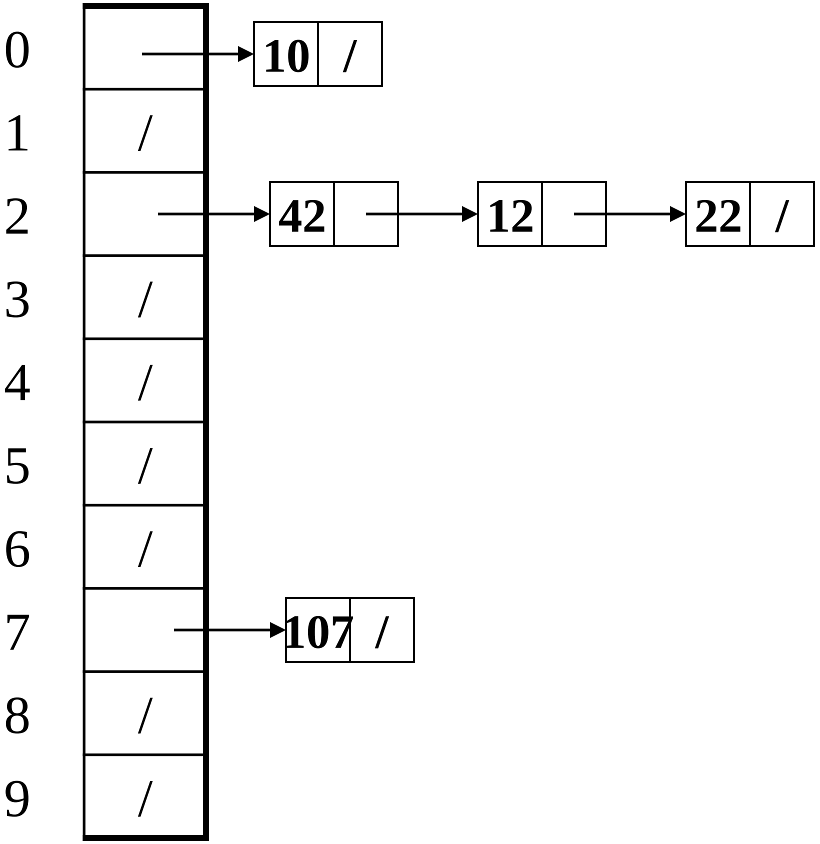
- Each unsuccessful **`find`** compares against λ items
- Each successful **`find`** compares against $\lambda / 2$ items
- If λ is low, find & insert likely to be O(1)
- We like to keep λ around 1 for separate chaining

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(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…
- Example: insert 38, 19, 8, 109, 10

0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	/
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(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…
- Example: insert 38, 19, 8, 109, 10

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

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(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…
- Example: insert 38, 19, 8, 109, 10

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

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(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…
- Example: insert 38, 19, 8, 109, 10

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

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(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…
- Example: insert 38, 19, 8, 109, 10

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

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(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…
- Example: insert 38, 19, 8, 109, 10

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

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(key) + i) % TableSize`
- In general have some probe function **`f`** and use
  `h(key) + f(i) % TableSize`

Open addressing does poorly with high load factor λ

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

Terminology

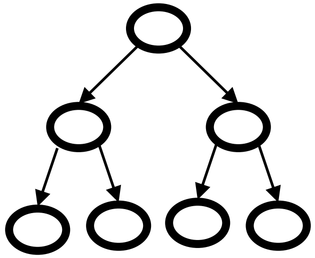
We and the book use the terms

- "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"

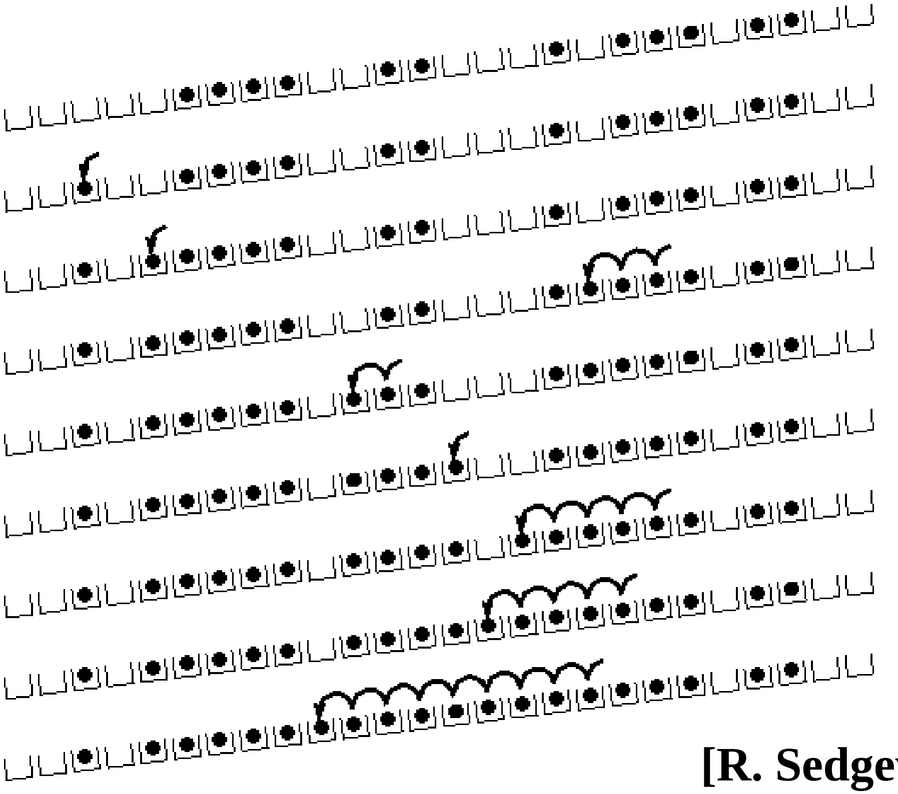
10	✘	/	23	/	/	16	✘	26

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

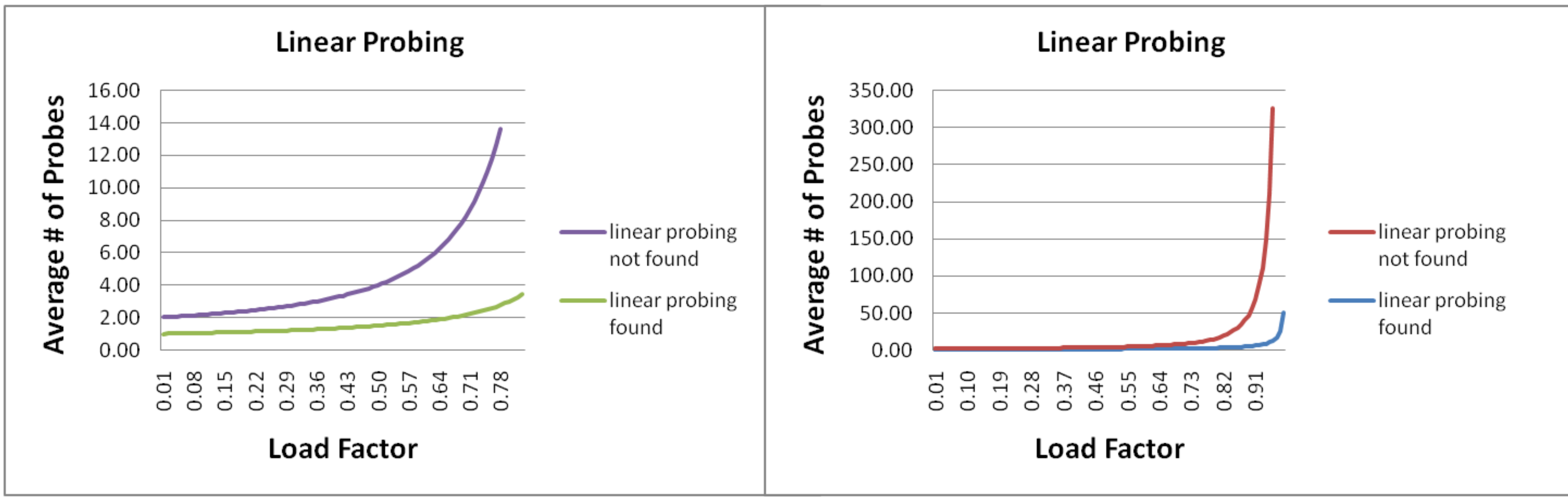
- Non-trivial facts we won’t prove:

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

- This is pretty bad: need to leave sufficient empty space in the table to get decent performance (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 λ and has no trouble with $\lambda > 1$

Open Addressing: Quadratic Probing

- We can avoid primary clustering by changing the probe function

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

  - For quadratic probing:

    `f(i) = i`^{2}

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

- Intuition: Probes quickly "leave the neighborhood"

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

TableSize=10

Insert:
89
18
49
58
79

Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

TableSize=10

Insert:
89
18
49
58
79

Quadratic Probing Example

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

TableSize=10

Insert:
89
18
49
58
79

Quadratic Probing Example

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

TableSize=10

Insert:
89
18
49
58
79

Quadratic Probing Example

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

TableSize=10

Insert:
89
18
49
58
79

Quadratic Probing Example

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

TableSize=10

Insert:
89
18
49
58
79

Another Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	

TableSize = 7

Insert:

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

Another Quadratic Probing Example

0	
1	
2	
3	
4	
5	
6	76

TableSize = 7

Insert:

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

Another Quadratic Probing Example

0	
1	
2	
3	
4	
5	40
6	76

TableSize = 7

Insert:

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

Another Quadratic Probing Example

0	48
1	
2	
3	
4	
5	40
6	76

TableSize = 7

Insert:

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

Another Quadratic Probing Example

0	48
1	
2	5
3	
4	
5	40
6	76

TableSize = 7

Insert:

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

Another Quadratic Probing Example

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

TableSize = 7

Insert:

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

Another Quadratic Probing Example

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

TableSize = 7

Insert:

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

Doh: For all *n*, **`(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 $0 \le i,j \le T/2$ where $i \ne j$,

    $$(h(key) + i^2)\ \%\ T \ne (h(key) + j^2)\ \%\ T$$

  - If **`T = TableSize`** is *prime* and $\lambda < ½$,
    quadratic probing will find an empty slot in at most **`T/2`** probes

  - If you keep $\lambda < ½$, 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^{th} probe: **`h(key) % TableSize`**
  - 1^{st} probe: **`(h(key) + g(key)) % TableSize`**
  - 2^{nd} probe: **`(h(key) + 2*g(key)) % TableSize`**
  - 3^{rd} probe: **`(h(key) + 3*g(key)) % TableSize`**
  - …
  - i^{th} probe: **`(h(key) + i*g(key)) % TableSize`**

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

Double Hashing

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

T = 10 (TableSize)

<u>Hash Functions</u>:

h(key) = key mod T

g(key) = 1 + ((key/T) mod (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(key) = key mod T

g(key) = 1 + ((key/T) mod (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:

$h(key) = key \bmod T$

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

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(key) = key mod T

g(key) = 1 + ((key/T) mod (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(key) = key mod T

g(key) = 1 + ((key/T) mod (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(key) = key mod T

g(key) = 1 + ((key/T) mod (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 + 5 = 8

3 + 10 = 13

3 + 15 = 18

3 + 20 = 23

3 + 25 = 28

Double Hashing Analysis

- Intuition:

  Because each probe is "jumping" by **`g(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(key) = key % p`**
  - **`g(key) = q – (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(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 λ depends on type of table

- Popular topic for job interview questions
  - Also many real-world applications

CSE332: Data Abstractions

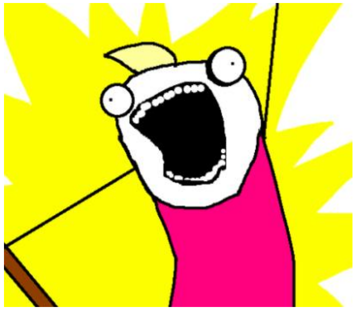
Lecture 10: Comparison Sorting

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`**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 **`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 **`A`** such that for any **`i`** and **`j`**,
  if **`i < j`** then **`A[i]`** ≤ **`A[j]`**
- Unspoken assumption: **`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`**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 2nd element in order
  - Now insert 3rd element in order
  - Now insert 4th 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`**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 2nd element in order
  - Now insert 3rd element in order
  - Now insert 4th 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 1st
  - Find next smallest element, put it 2nd
  - Find next smallest element, put it 3rd
  - …

- "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^{st}
  - Find next smallest element, put it 2^{nd}
  - Find next smallest element, put it 3^{rd}
  - …

- "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`**th element, put it at **`arr[n-i]`**

  • That array location is not part of the heap anymore!

4	7	5	9	**8**	**6**	**10**	**3**	**2**	**1**

heap part (4 … 10) — sorted part (3 2 1)

`arr[n-i]=`
`deleteMin()`

5	7	6	9	**8**	**10**	**4**	**3**	**2**	**1**

heap part (5 … 10) — sorted part (4 3 2 1)

"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

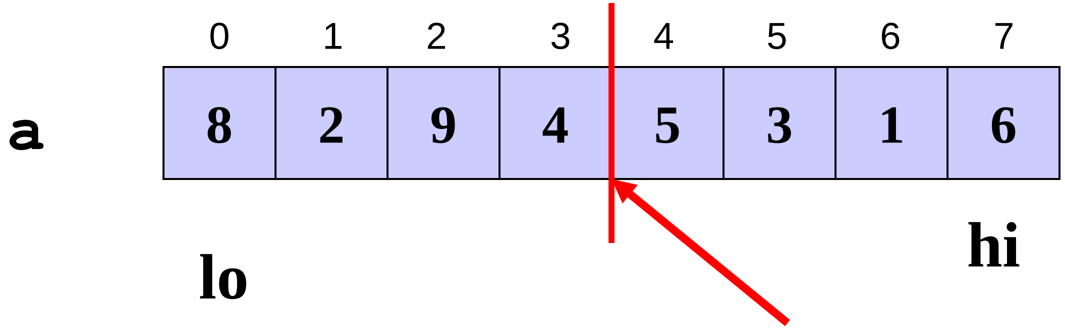
Divide-and-Conquer Sorting

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 **`(hi+lo)/2`**
    - Sort from **`(hi+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

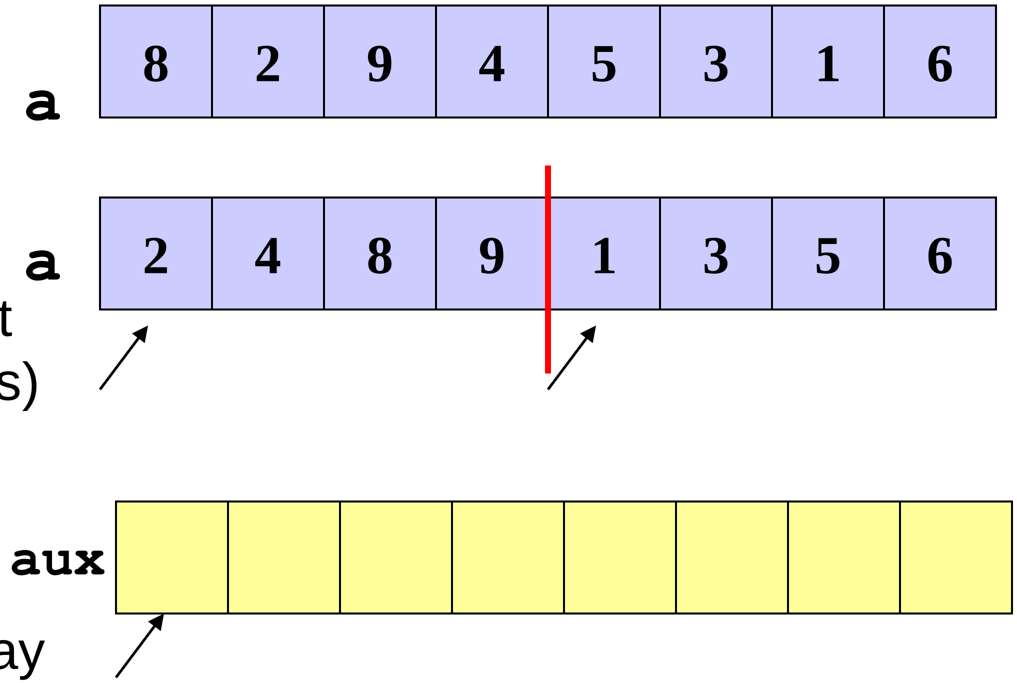
Start with:

After recursion:

(for now we just assume it works)

Merge:

Use 3 "fingers" and 1 more array



(After merge, copy back to original array)

Example: Focus on Merging

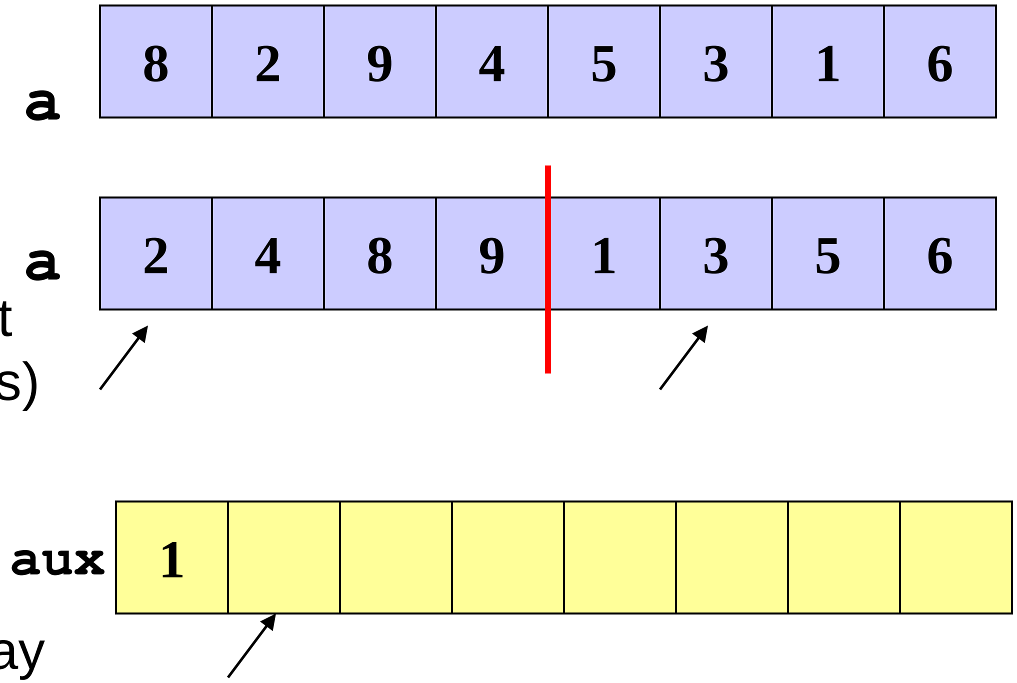
Start with:

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: Focus on Merging

Start with:

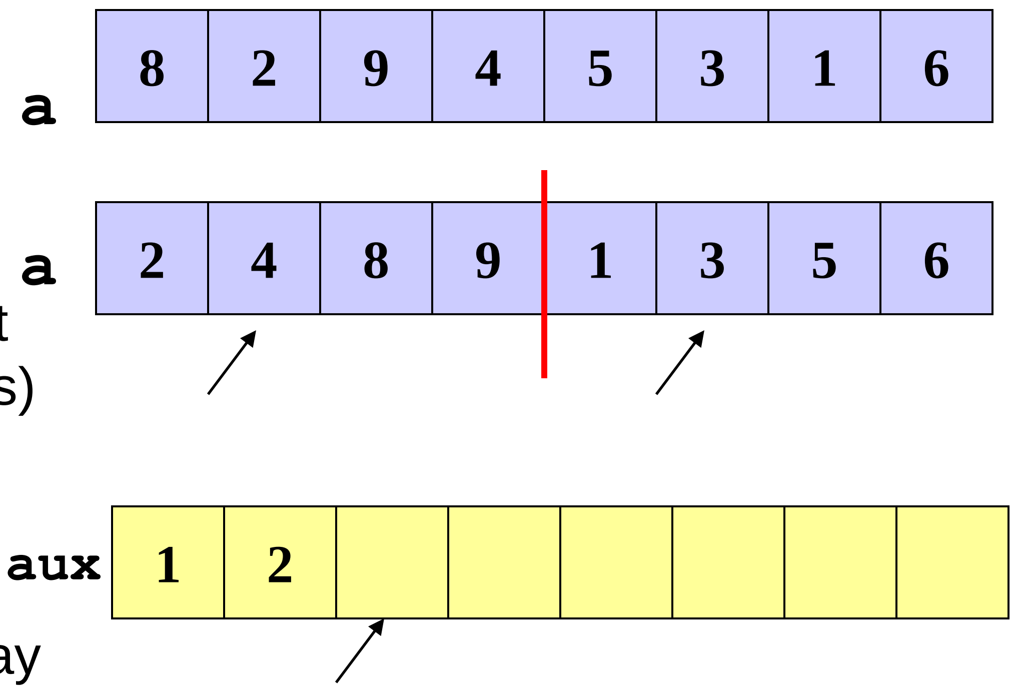
After recursion:

(for now we just assume it works)

Merge:

Use 3 "fingers"

and 1 more array



(After merge, copy back to original array)

Example: Focus on Merging

Start with:

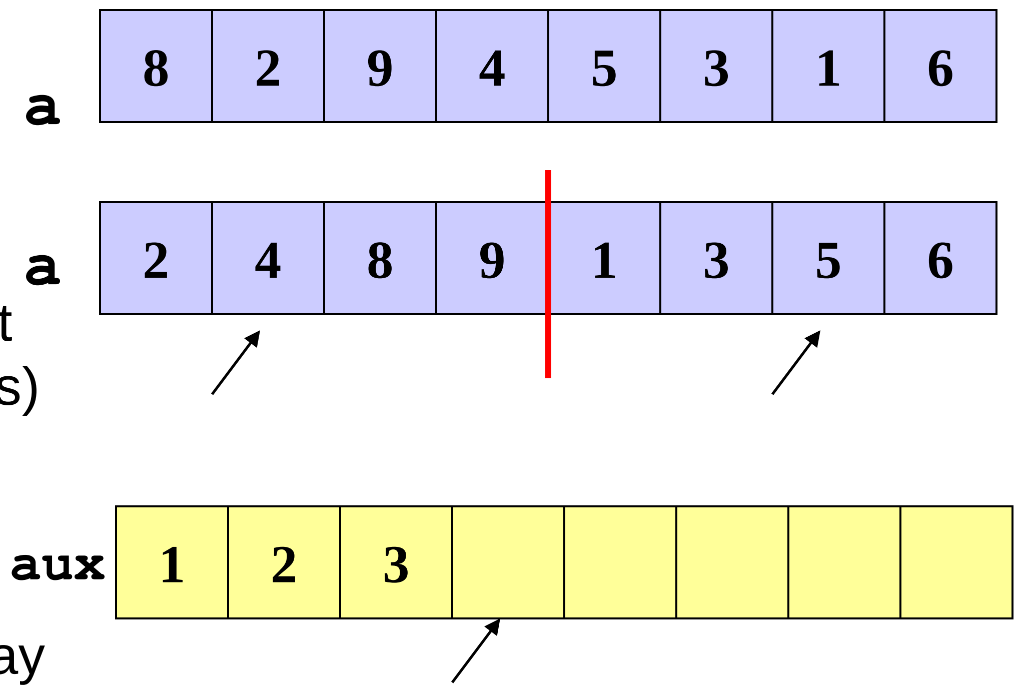
After recursion:

(for now we just assume it works)

Merge:

Use 3 “fingers”

and 1 more array



(After merge, copy back to original array)

Example: Focus on Merging

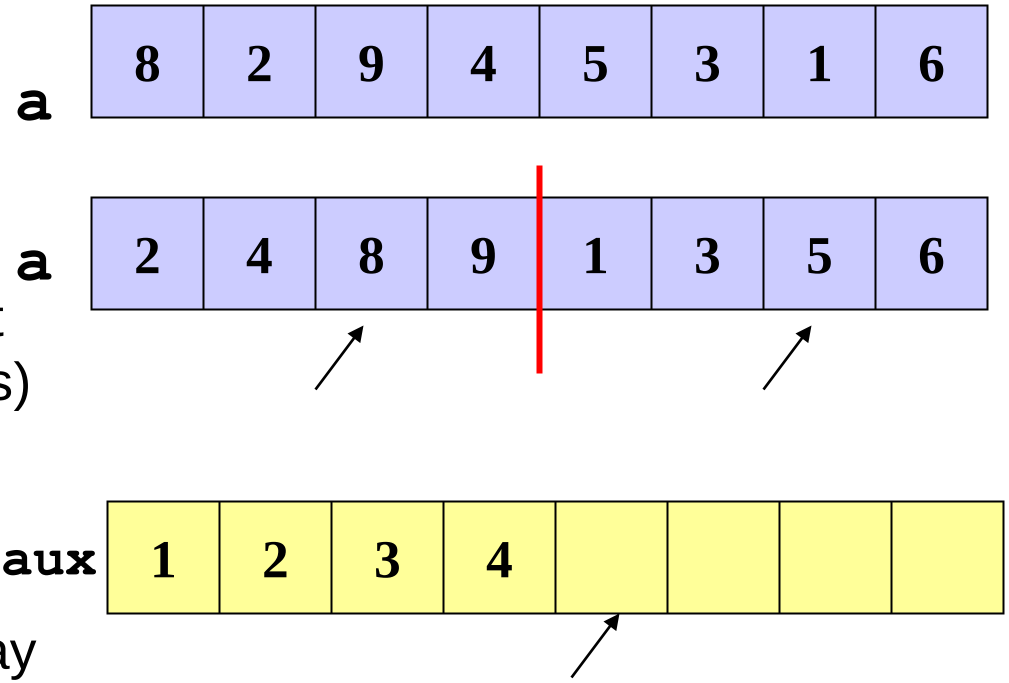
Start with:

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: Focus on Merging

Start with:

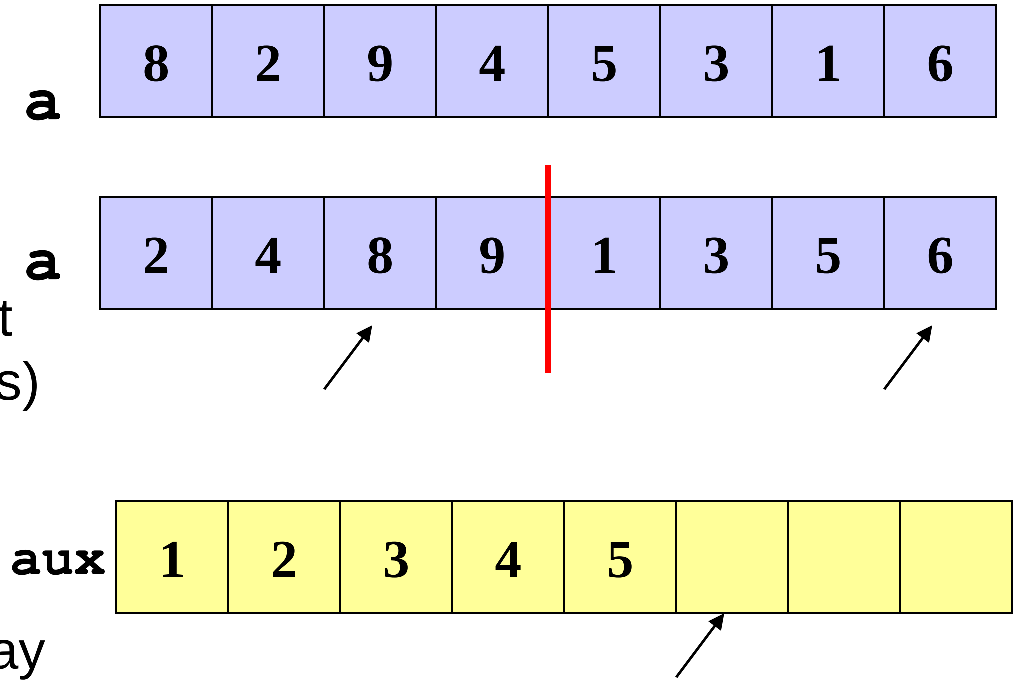
After recursion:

(for now we just assume it works)

Merge:

Use 3 "fingers"

and 1 more array



(After merge, copy back to original array)

Example: Focus on Merging

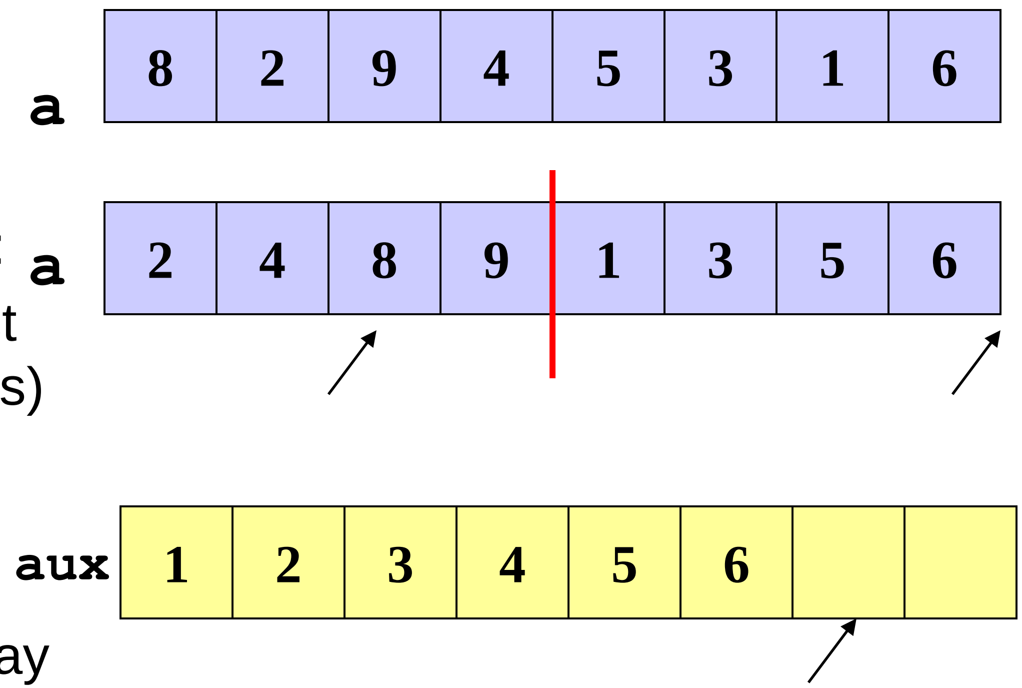
Start with:

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: Focus on Merging

Start with:

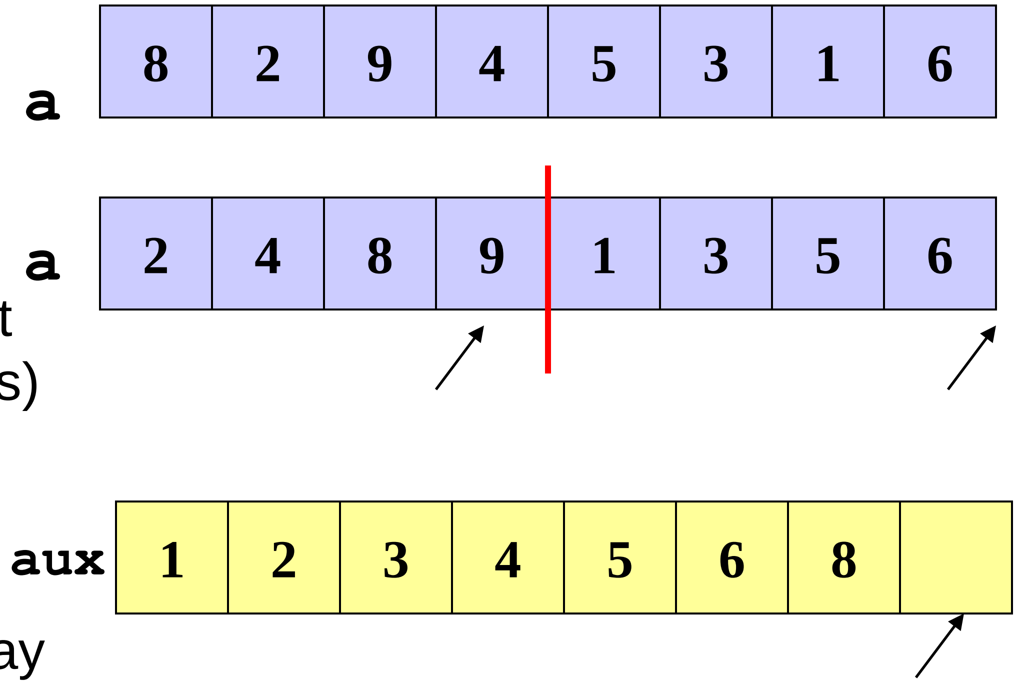
After recursion:

(for now we just assume it works)

Merge:

Use 3 “fingers”

and 1 more array



(After merge, copy back to original array)

Example: Focus on Merging

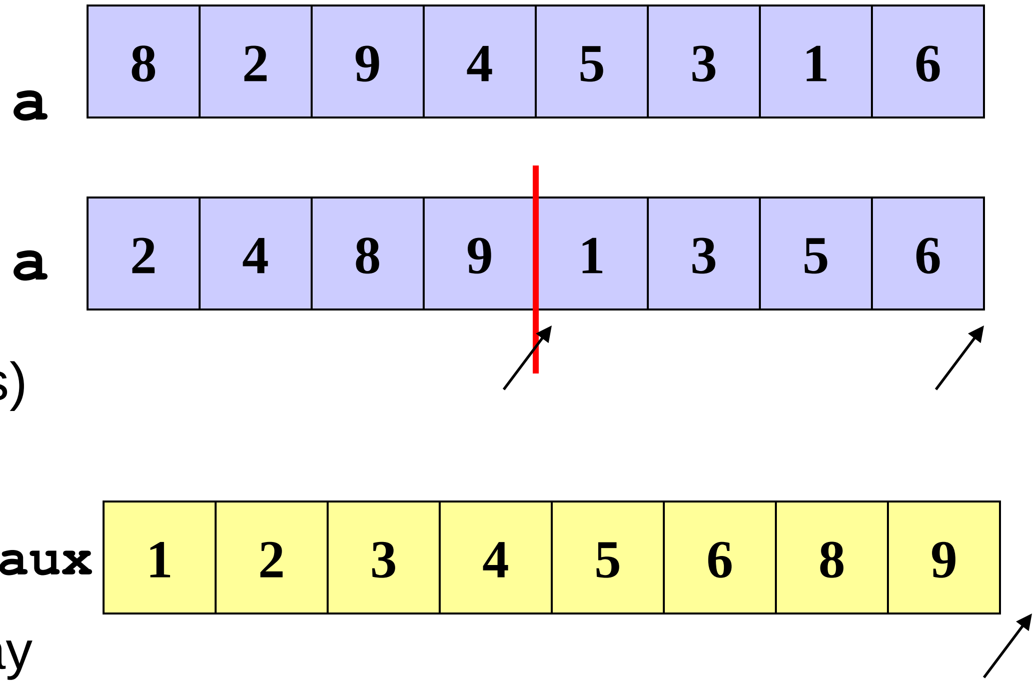
Start with:

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: Focus on Merging

Start with:

a

8	2	9	4	5	3	1	6

After recursion:

(for now we just assume it works)

a

2	4	8	9	1	3	5	6

Merge:

Use 3 “fingers” and 1 more array

aux

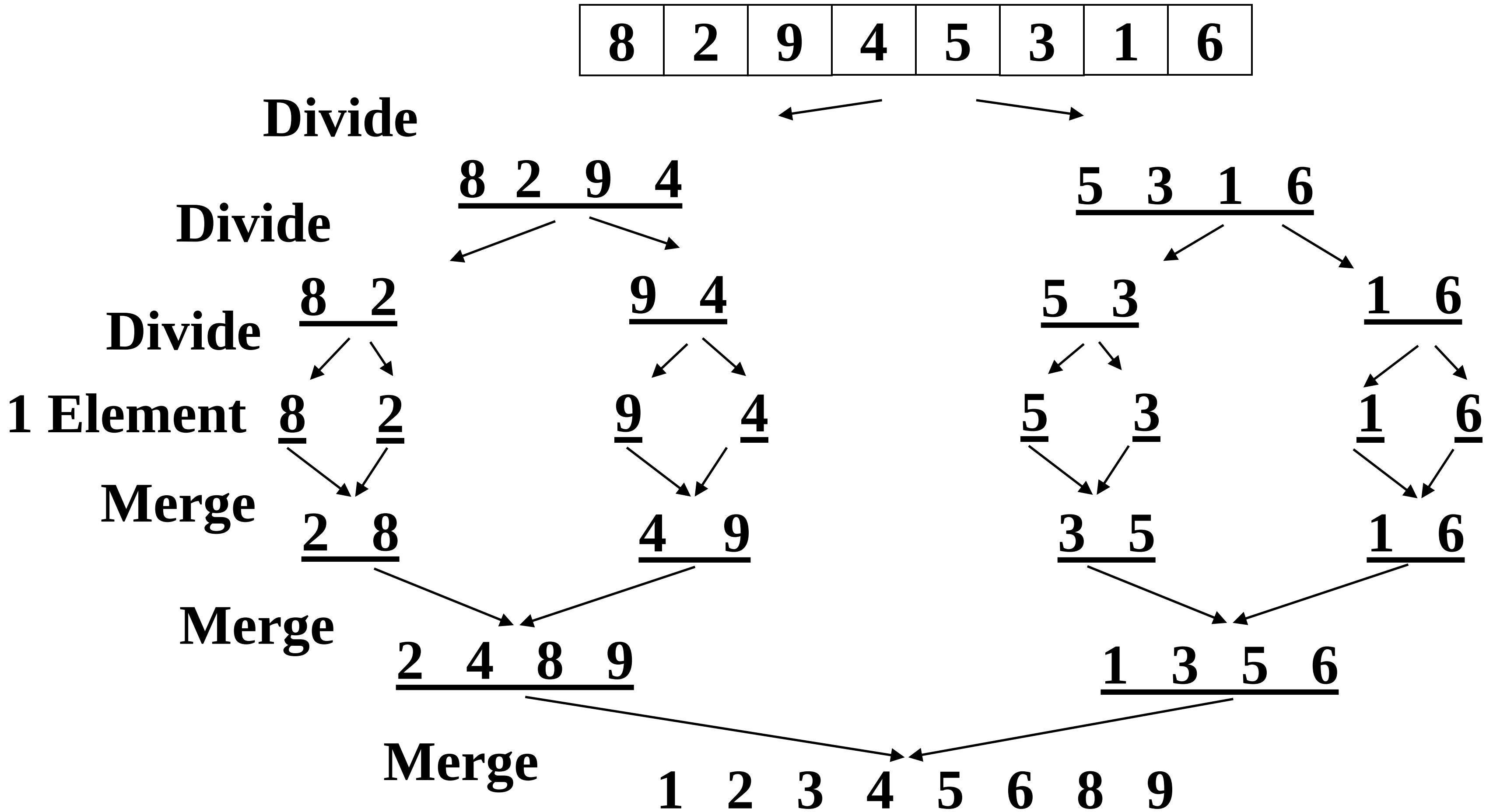
1	2	3	4	5	6	8	9

(After merge, copy back to original array)

a

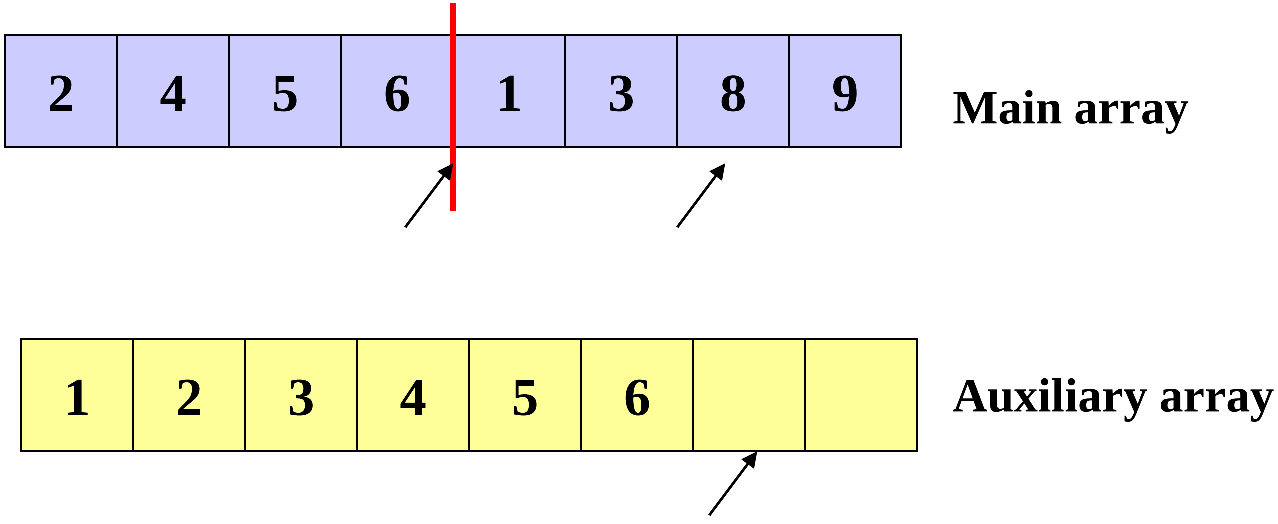
1	2	3	4	5	6	8	9

Example: Mergesort Recursion



Mergesort: Some Time Saving Details

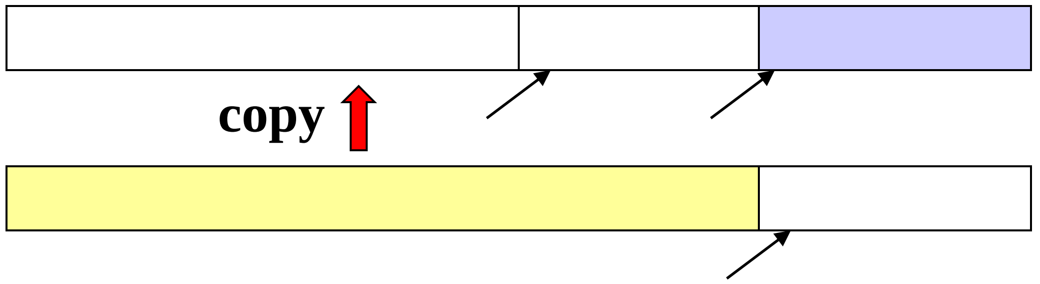
- What if the final steps of our merge looked like this:



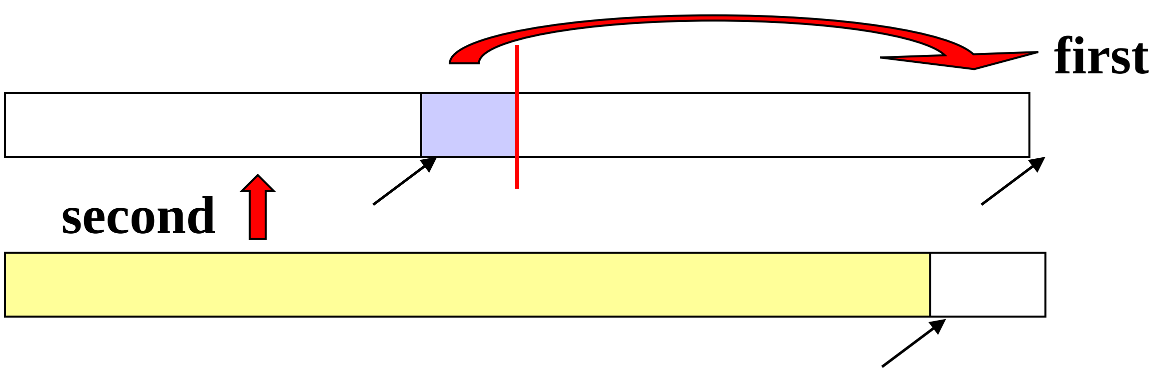
- Wasteful to copy to the auxiliary array just to copy back…

Mergesort: Some Time Saving Details

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



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



Mergesort: Saving Space and Copying

Simplest / Worst:

Use a new auxiliary array of size **`(hi-lo)`** 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

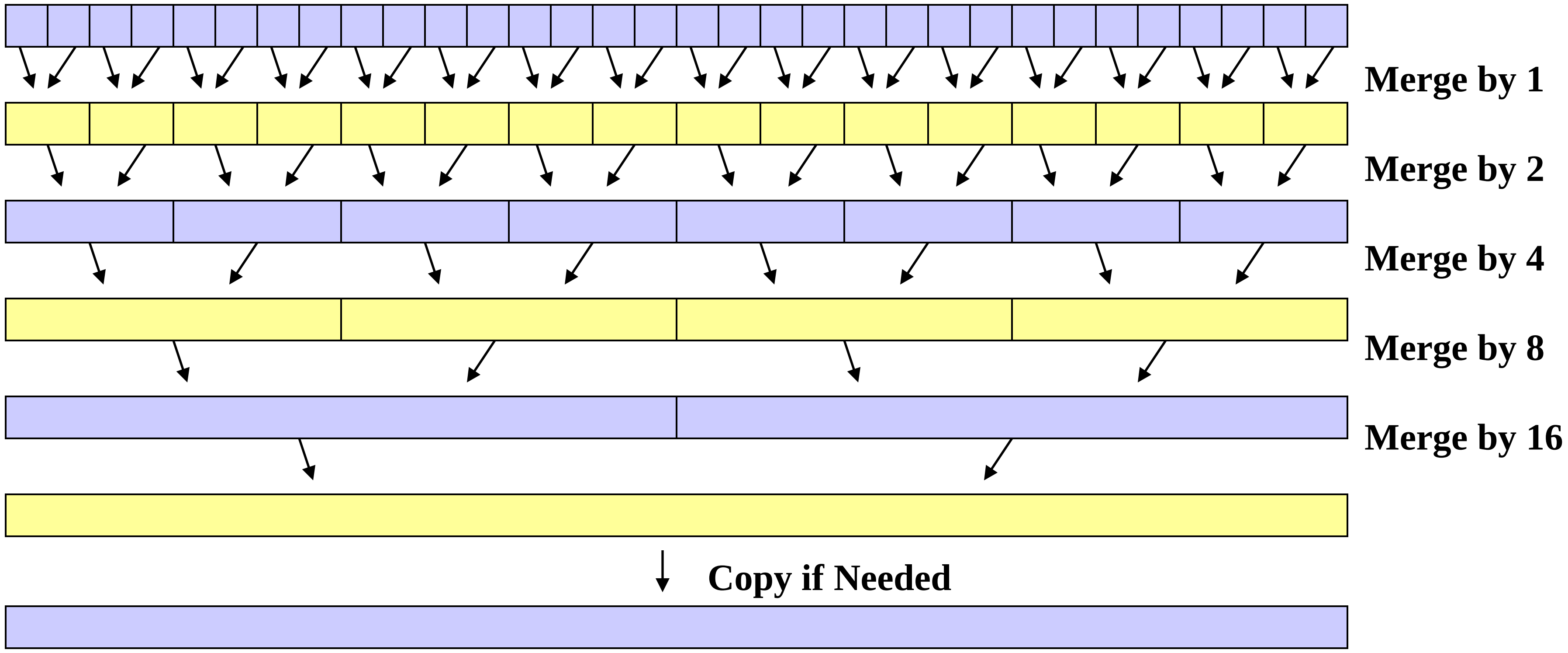
Best:

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:

$T(1) = c_1$

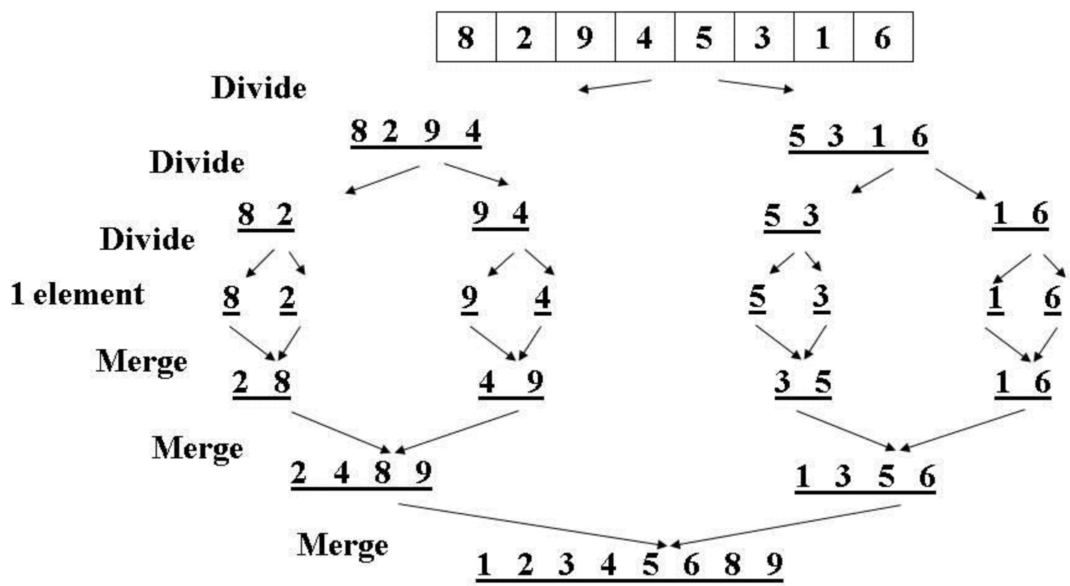
$T(n) = 2T(n/2) + c_2 n$

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

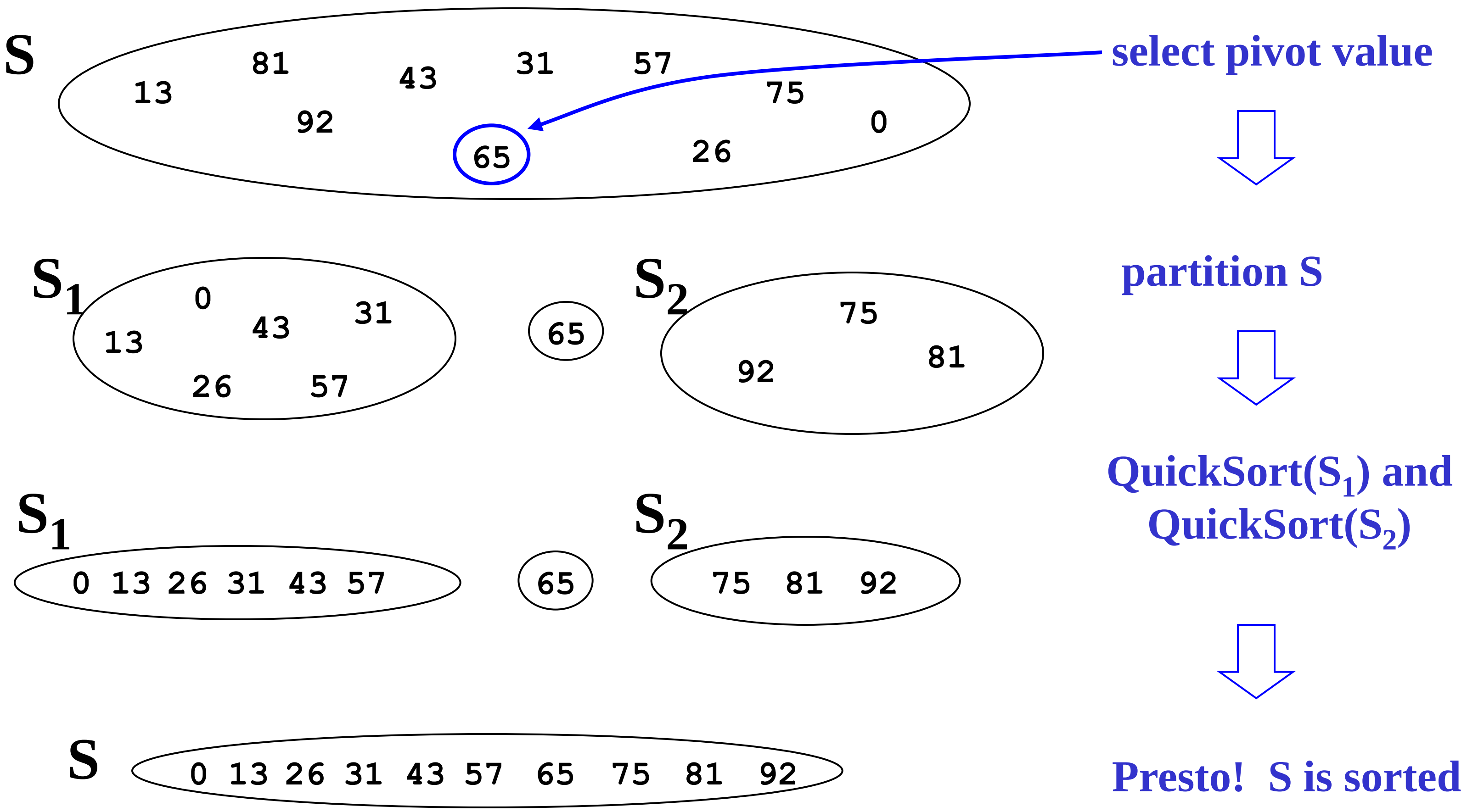
2. Partition all the data into:
   A. The elements less than the pivot
   B. The pivot
   C. The elements greater than the pivot

3. Recursively sort A and C

4. The answer is as simple as “A, B, C”

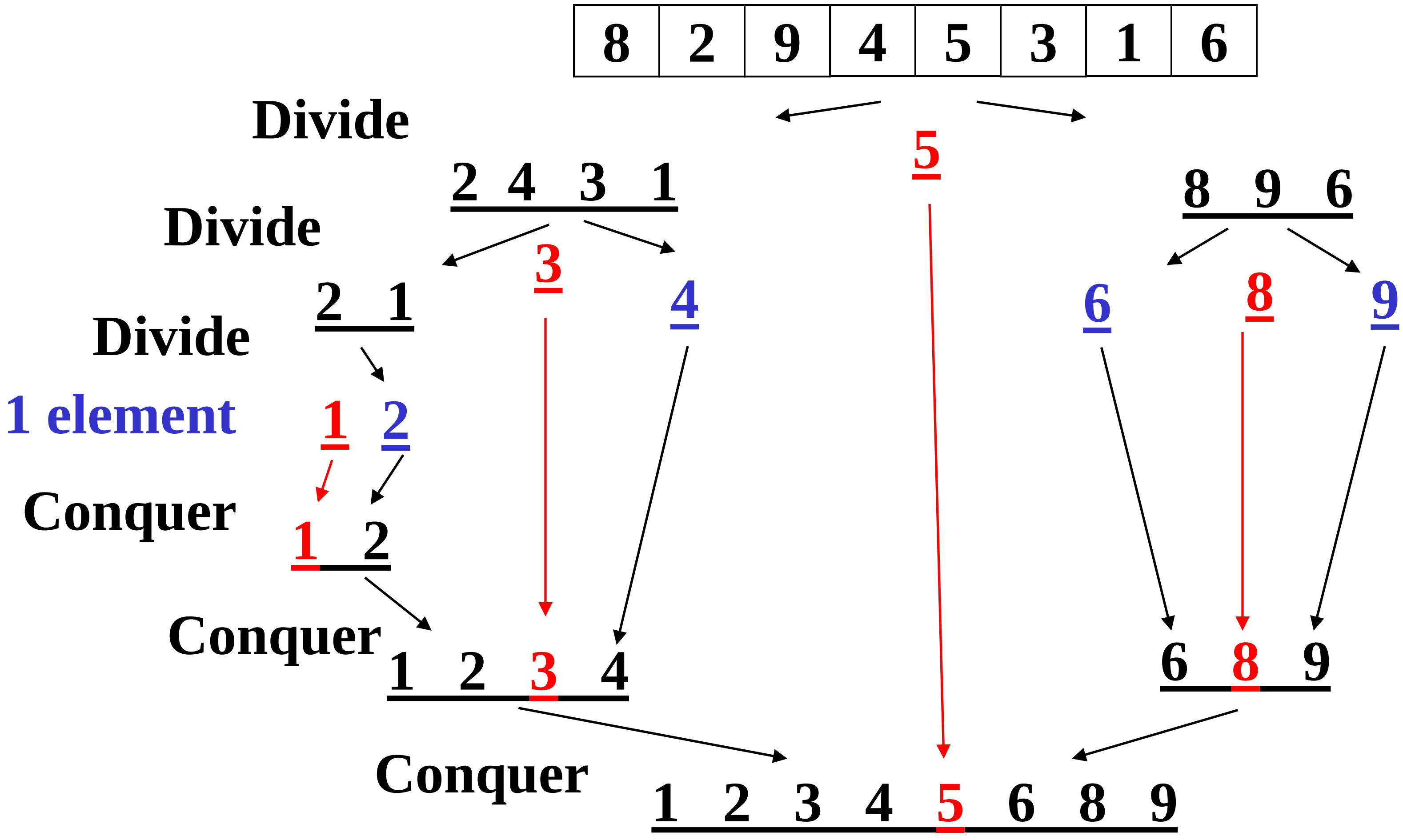
Alas, there are some details lurking in this algorithm

Quicksort: Think in Terms of Sets



[Weiss]

Example: Quicksort Recursion



Quicksort Details

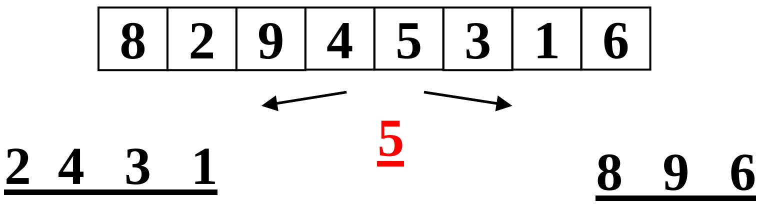
We have not explained:

- 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

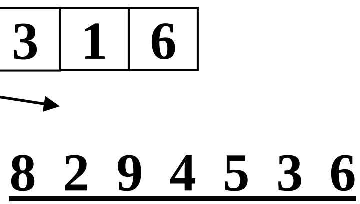
- How to implement partitioning
  - In linear time
  - In place

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

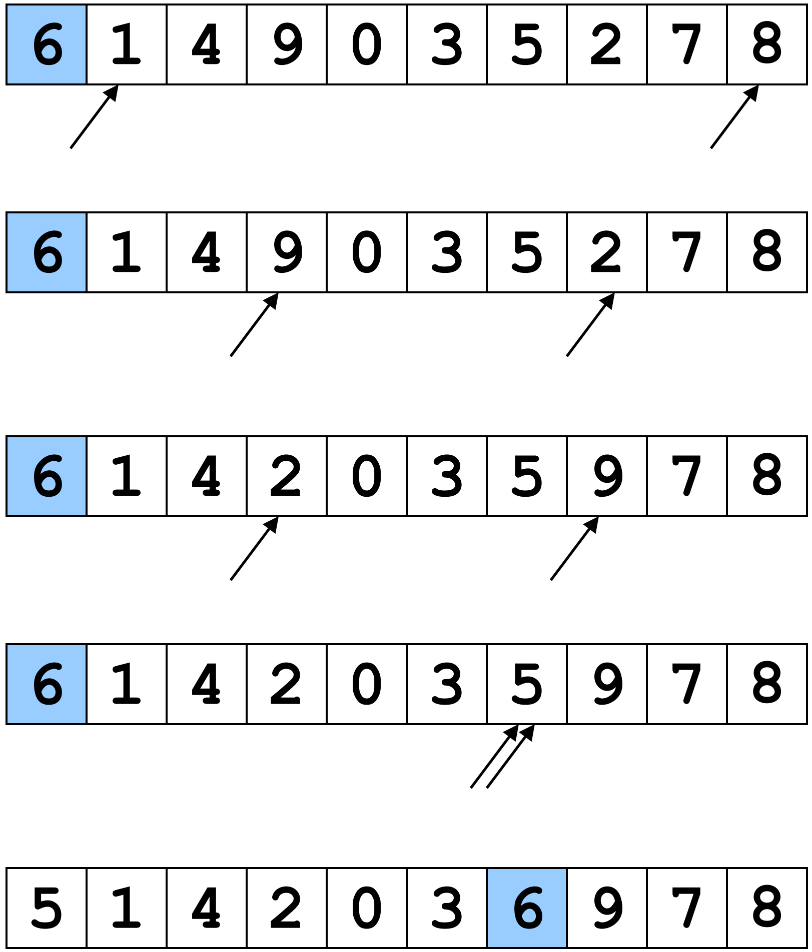
Now partition in place

Move fingers

Swap

Move fingers

Move pivot



Quicksort Analysis

- Best-case: Pivot is always the median

  T(0)=T(1)=1

  $T(n)=2T(n/2) + n$ -- 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)	**Fancier algorithms: O(n log n)**	**Comparison lower bound: Ω(n log n)**	**Specialized algorithms: O(n)**	**Handling huge data sets**
Insertion sort **Selection sort** **Shell sort** **…**	**Heap sort** **Merge sort** **Quick sort (avg)** **…**		**Bucket sort** **Radix sort**	**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 -- 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: $O(n \texttt{ log } n)$, 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.
  - $n(n-1)(n-2)\ldots(2)(1) = n!$ possible orderings

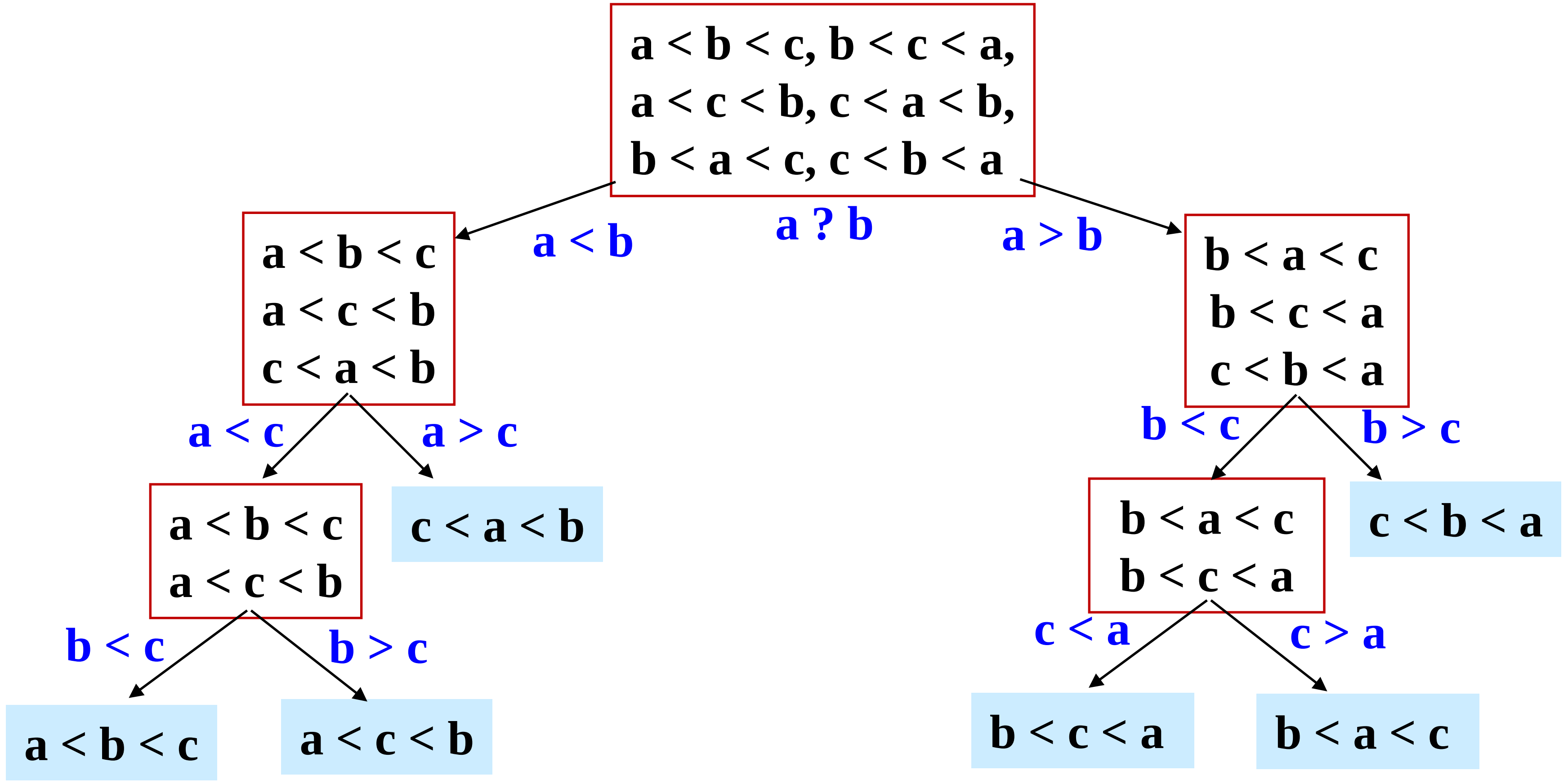
Representing Every Comparison Sort

- 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



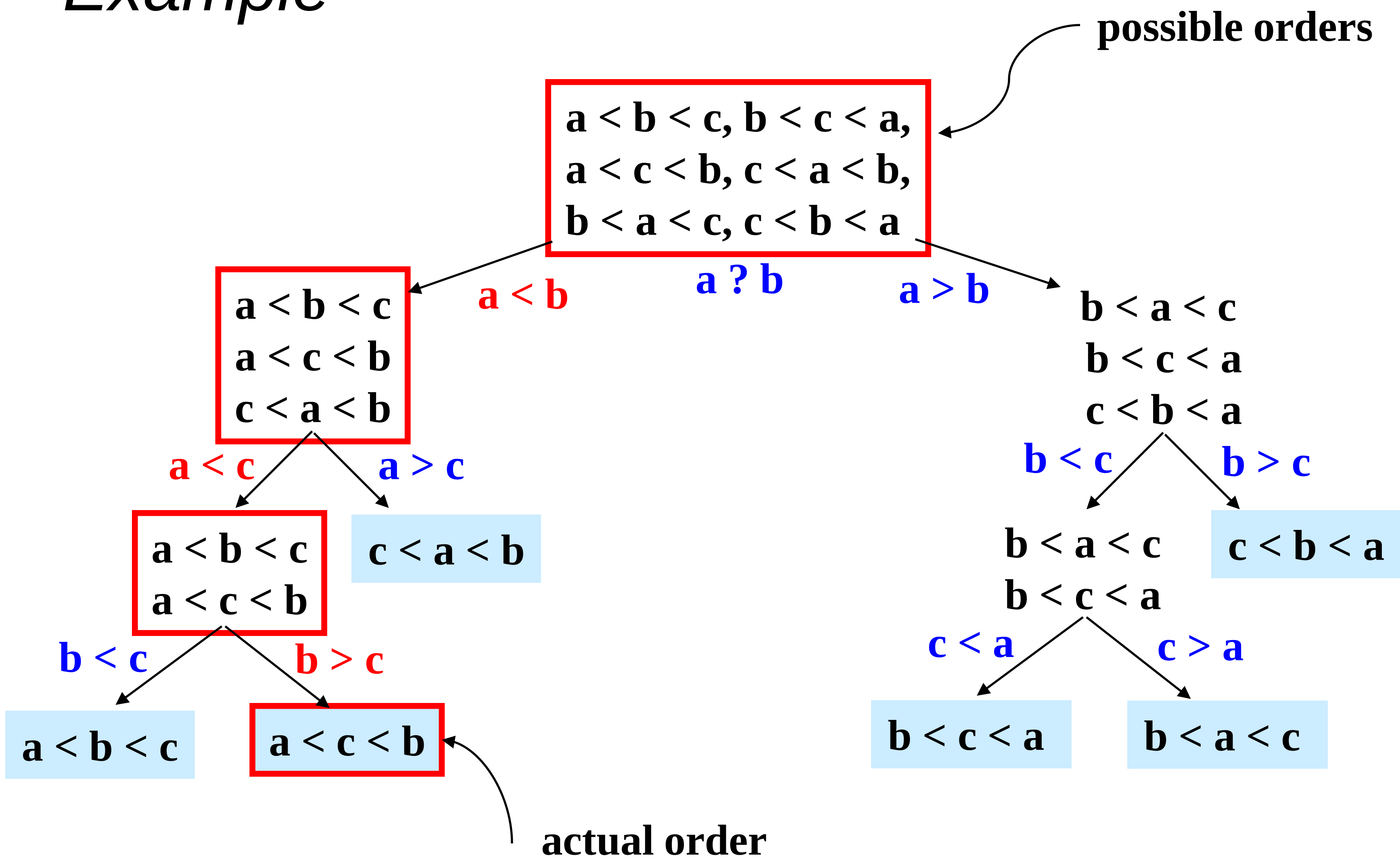
The leaves contain all the possible orderings of a, b, c

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


possible orders
a < b < c, b < c < a,
a < c < b, c < a < b,
b < a < c, c < b < a
a ? b
a < b
a > b
a < b < c
a < c < b
c < a < b
b < a < c
b < c < a
c < b < a
a < c
a > c
b < c
b > c
a < b < c
a < c < b
c < a < b
b < a < c
b < c < a
c < b < a
b < c
b > c
c < a
c > a
a < b < c
a < c < b
b < c < a
b < a < c
actual order

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?

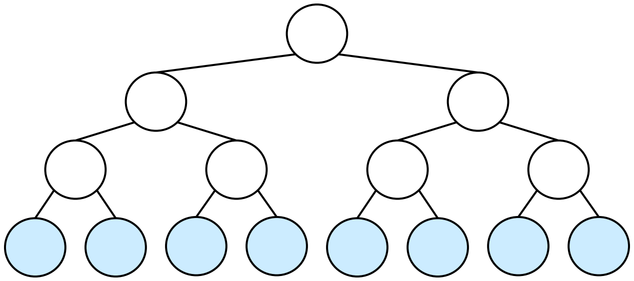
Now: Show that a binary tree with n! leaves has height $\Omega(n \log n)$

- 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 $\log_2 L$
- So the height of our decision tree, h:

$h \geq \log_2 (n!)$	property of binary trees
$= \log_2 (n*(n-1)*(n-2)\ldots(2)(1))$	definition of factorial
$= \log_2 n + \log_2 (n-1) + \ldots + \log_2 1$	property of logarithms
$\geq \log_2 n + \log_2 (n-1) + \ldots + \log_2 (n/2)$	keep first n/2 terms
$\geq (n/2) \log_2 (n/2)$	each of the n/2 terms left is $\geq \log_2 (n/2)$
$\geq (n/2)(\log_2 n - \log_2 2)$	property of logarithms
$\geq (1/2)n\log_2 n - (1/2)n$	arithmetic
"=" $\Omega (n \log n)$	

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: Ω(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.ka. 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.ka. 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.ka. 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

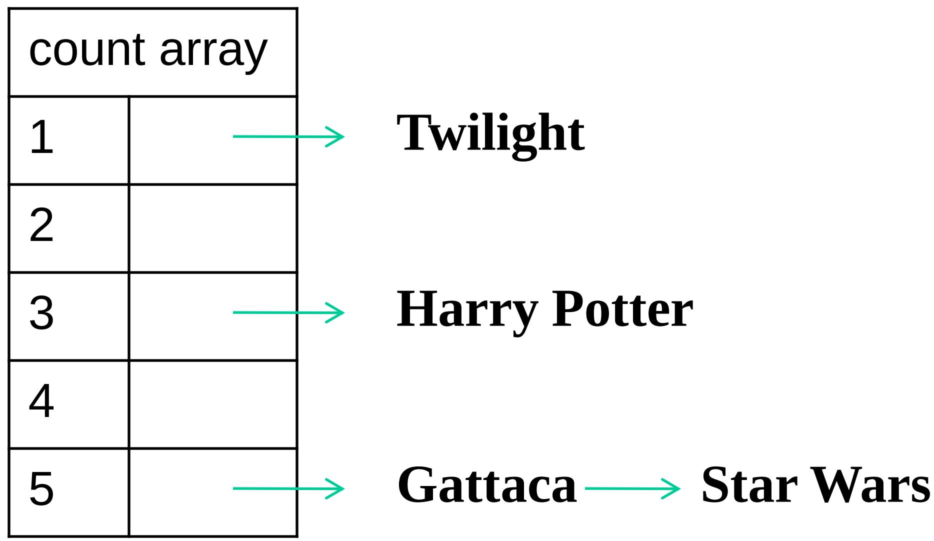
What is the running time?

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 1st 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

Bucket sort by 10's digit

After 1st pass

721
3
123
537
67
478
38
9

0	1	2	3	4	5	6	7	8	9
03 09		721 123	537 38			67	478		

After 2nd pass

3
9
721
123
537
38
67
478

Example: Radix Sort: Pass #3

After 2nd 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
003 **0**09 **0**38 **0**67	**1**23			**4**78	**5**37		**7**21		

After 3rd 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
- **Ω** $(n$ **`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!