Binary Search Trees

Ravi Chugh

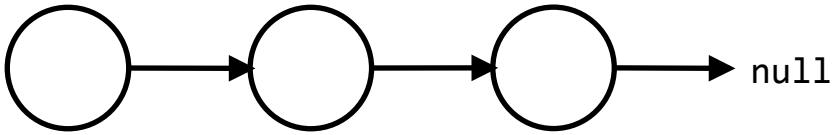
March 28, 2014

Review: Linked Lists

Goal: Program that keeps track of friends

Problem: Arrays have fixed length

Solution: **Linked Lists**



Review: Linked Lists

```
class LinkedList {
  str friend;
  LinkedList next;


  bool find(str s) {
    if (s == this.friend) return true
    else if (this.next == null) return false
    else return this.next.find(s)
  }


  ...
}
```

Questions for Today

```
class LinkedList {
  str friend;
  LinkedList next;

  bool find(str s) {
    if (s == this.friend) return true
    else if (this.next == null) return false
    else return this.next.find(s)
  }

  ...
}
```

- How "quickly" does `find` work?
- How can we do better?

find(s): How Many Nodes Traversed?

Best Case	Avg. Case	Worst Case
??	*??*	*??*

```
bool find(str s) {
  if (s == this.friend) return true
  else if (this.next == null) return false
  else return this.next.find(s)
}
```

find(s): How Many Nodes Traversed?

Best Case	Avg. Case	Worst Case
1	*≈ size*	*size*

```
bool find(str s) {
  if (s == this.friend) return true
  else if (this.next == null) return false
  else return this.next.find(s)
}
```

Idea: Keep Names Sorted!

```
class LinkedList {
  str friend;
  LinkedList next;


  bool find(str s) {
    if (s == this.friend) return true
    else if (this.next == null) return false
    else return this.next.find(s)
  }


  ...
}
```

Idea: Keep Names Sorted!

```
class LinkedList {
  str friend;
  LinkedList next; // all names in next
                   // are bigger than friend

  bool find(str s) {
    if (s == this.friend) return true
    else if (this.next == null) return false
    else return this.next.find(s)
  }


  ...
}
```

Idea: Keep Names Sorted!

```
class LinkedList {
  str friend;
  LinkedList next; // all names in next
                   // are bigger than friend

  bool find(str s) {
    if (s == this.friend) return true
    else if (this.next == null) return false
    else if (s < this.friend) return false
    else return this.next.find(s)
  }

  ...
}
```

```
class LinkedList {
  str friend;
  LinkedList next; // all names in next
                   // are bigger than friend
```

basic info about **data structure** is tracked **"inside the PL"**

more complex **invariants** are tracked **"outside the PL"**

- *comments*
- *test cases*
- *in your head!*

`find(s)`: How Many Nodes Traversed?

Data Structure	Best Case	Avg. Case	Worst Case
Lists	*1*	*≈ size*	*size*

`find(s)`: How Many Nodes Traversed?

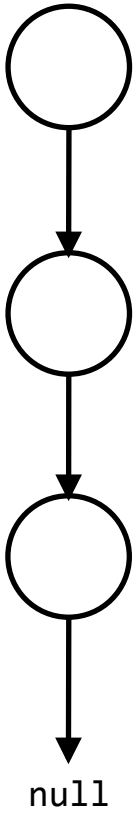
Data Structure + Invariants	Best Case	Avg. Case	Worst Case
Lists	*1*	*≈ size*	*size*
Lists + Sorted	*??*	*??*	*??*

`find(s)`: How Many Nodes Traversed?

Data Structure + Invariants	Best Case	Avg. Case	Worst Case
Lists	*1*	*≈ size*	*size*
Lists + Sorted	*1*	*≈ size/2*	*size*

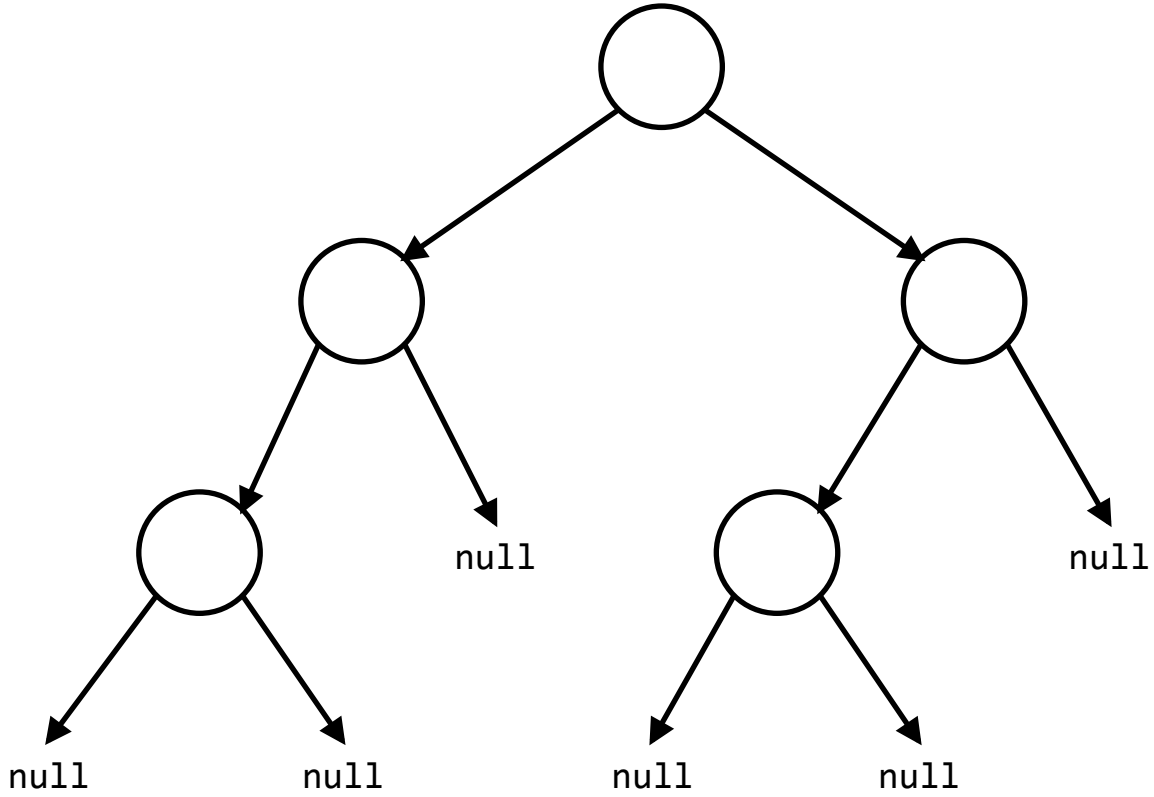
Linked Lists

node has 0 or 1
“next” pointers



Binary Trees

node has 0, 1, or 2
“next” pointers



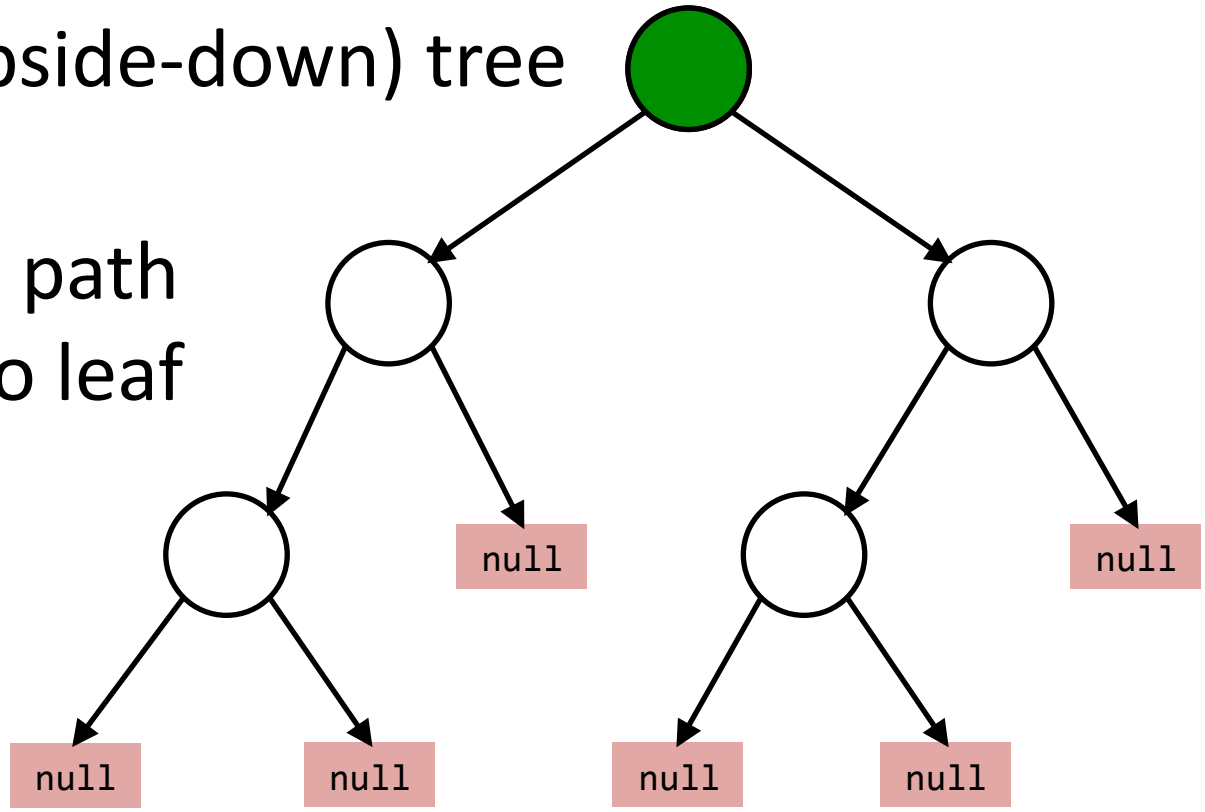
Binary Trees

node has 0, 1, or 2
"next" pointers

"**root**" of the (upside-down) tree

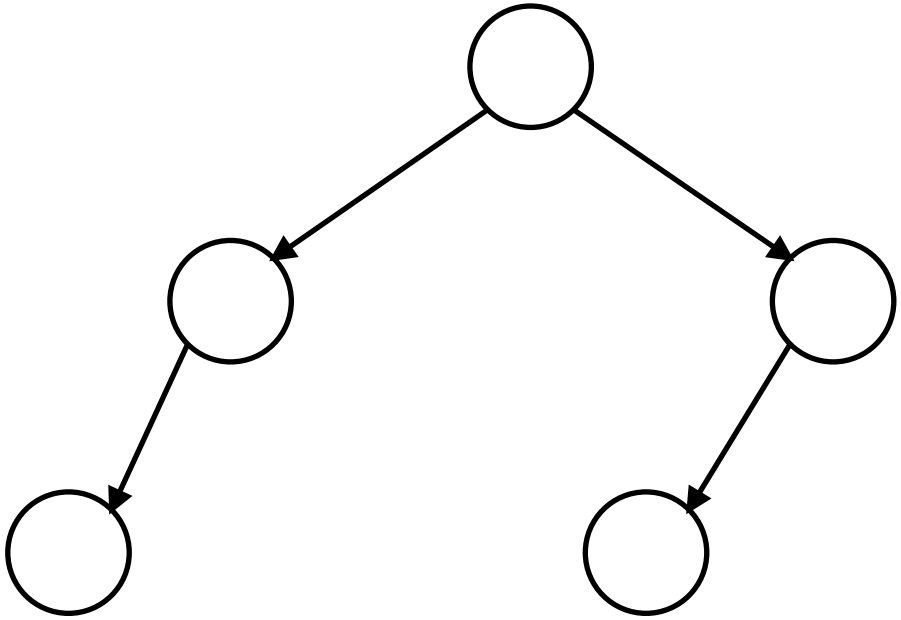
"**branch**" is path
from root to leaf

"**leaves**" of tree



Binary Trees

node has 0, 1, or 2
"next" pointers



```
class BinaryTree {
  str friend;
  BinaryTree left;
  BinaryTree right;

  bool find(str s) {

    ???
    ???
    ???

  }

  ...
}
```

```
class BinaryTree {
  str friend;
  BinaryTree left;
  BinaryTree right;

  bool find(str s) {
    if (s == this.friend) return true
    if (this.left == this.right == null) return false
    if (this.left == null) return this.right.find(s)
    if (this.right == null) return this.left.find(s)
    else return this.left.find(s) || this.right.find(s)
  }

  ...
}
```

Best Case	Avg. Case	Worst Case
??	*??*	*??*

`find(s)`: How Many Nodes Traversed?

Data Structure + Invariants	Best Case	Avg. Case	Worst Case
Lists	*1*	*≈ size*	*size*
Lists + Sorted	*1*	*≈ size/2*	*size*
Trees	*1*	*≈ size*	*size*

```
class BinaryTree {
  str friend;
  BinaryTree left;
  BinaryTree right;
```

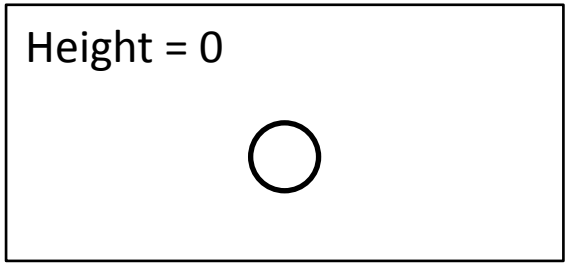
```
class BinaryTree {
  str friend;
  BinaryTree left;  // names smaller than friend
  BinaryTree right; // names bigger  than friend
```

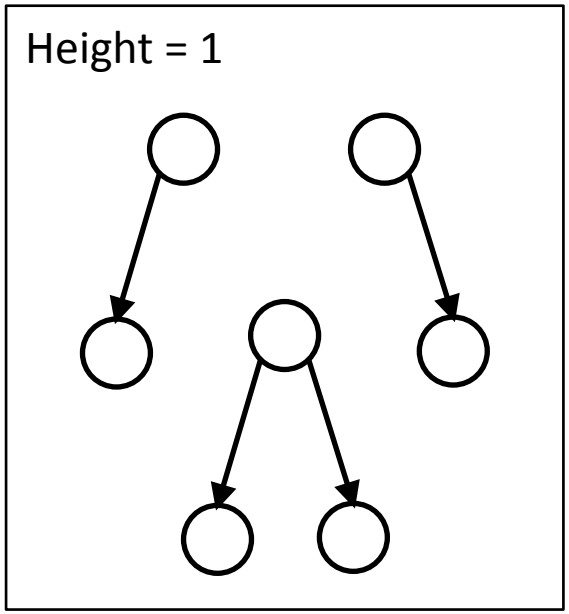
Called a
Binary "Search" Tree (BST)

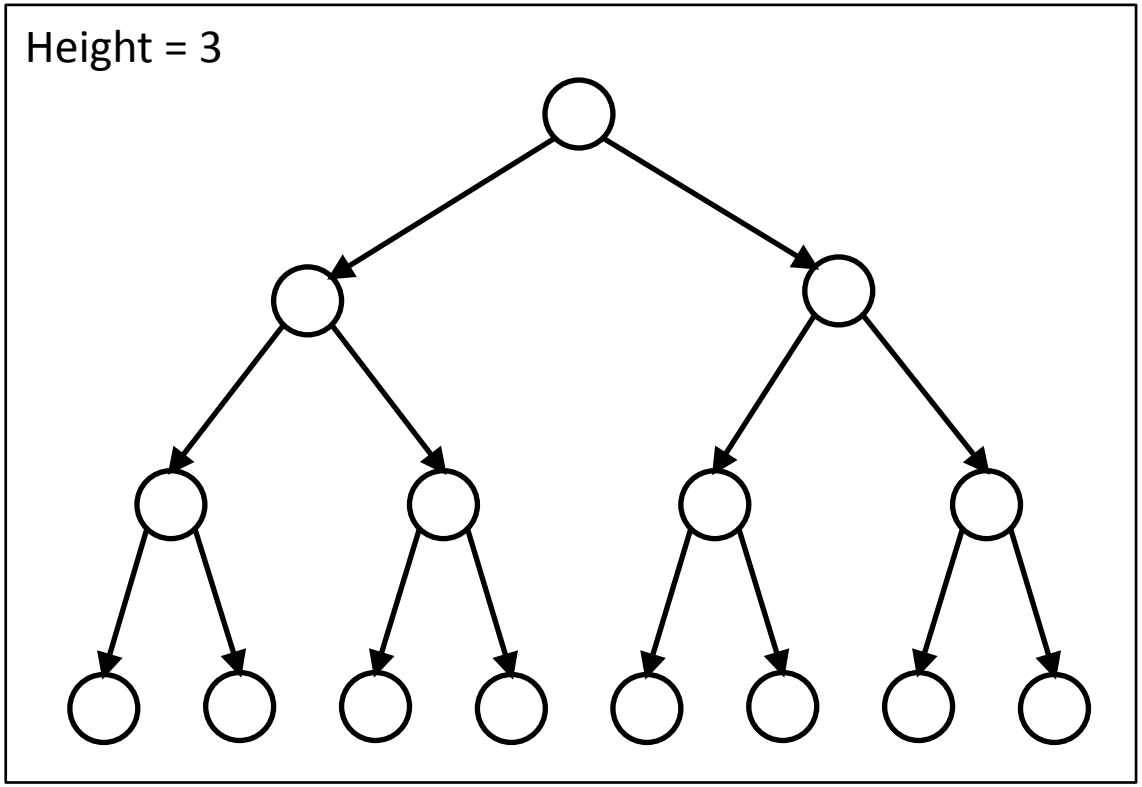
```
class BinaryTree {
  str friend;
  BinaryTree left;  // names smaller than friend
  BinaryTree right; // names bigger  than friend

  bool find(str s) {
    if (s == this.friend) return true
    else if (s < this.friend)
      if (this.left == null) return false
      else return this.left.find(s)
    else // if (s > this.friend)
      if (this.right == null) return false
      else return this.right.find(s)
  }
  ...
}
```

Tree Height

"**height**" = length of longest branch minus one



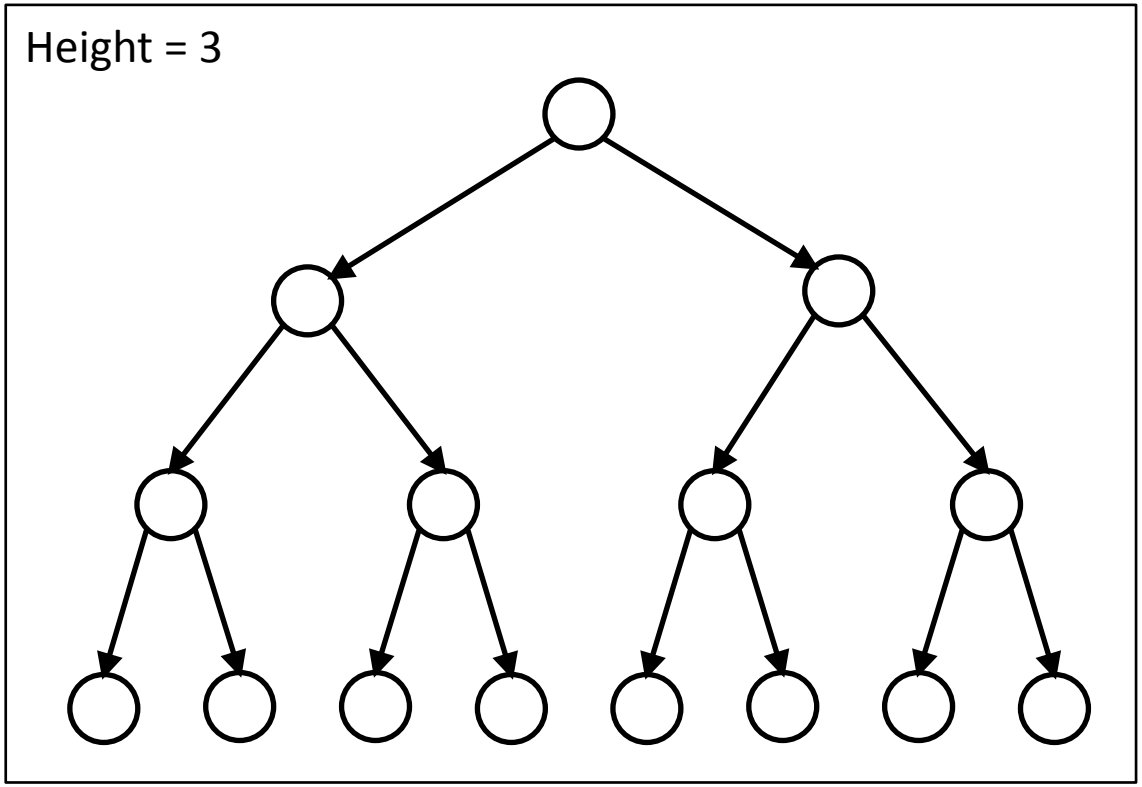




Tree Height

"**height**" = length of longest branch minus one

height	max size
0	1
1	3
2	7
3	15
4	31
5	63
6	127
…	…
h	$\mathbf{2^{(h+1)} - 1}$



Tree Height

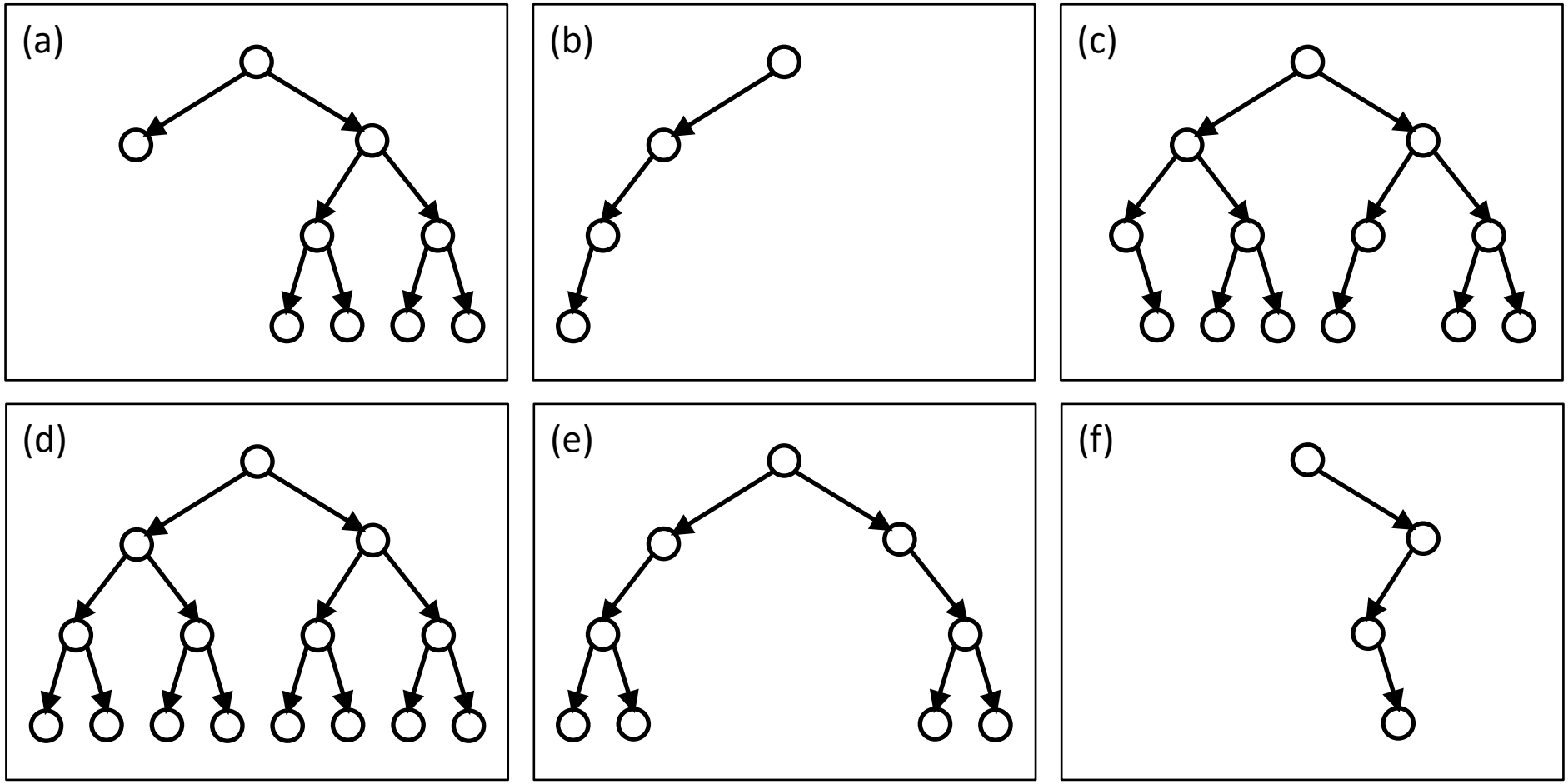
"**height**" = length of longest branch minus one

height	max size
0	1
1	3
2	7
3	15
4	31
5	63
6	127
…	…
h	$\mathbf{2^{(h+1)} - 1}$

- Can store up to ***n*** nodes in a tree of height $\boldsymbol{log_2(n) - 1}$
- This is the upper bound…

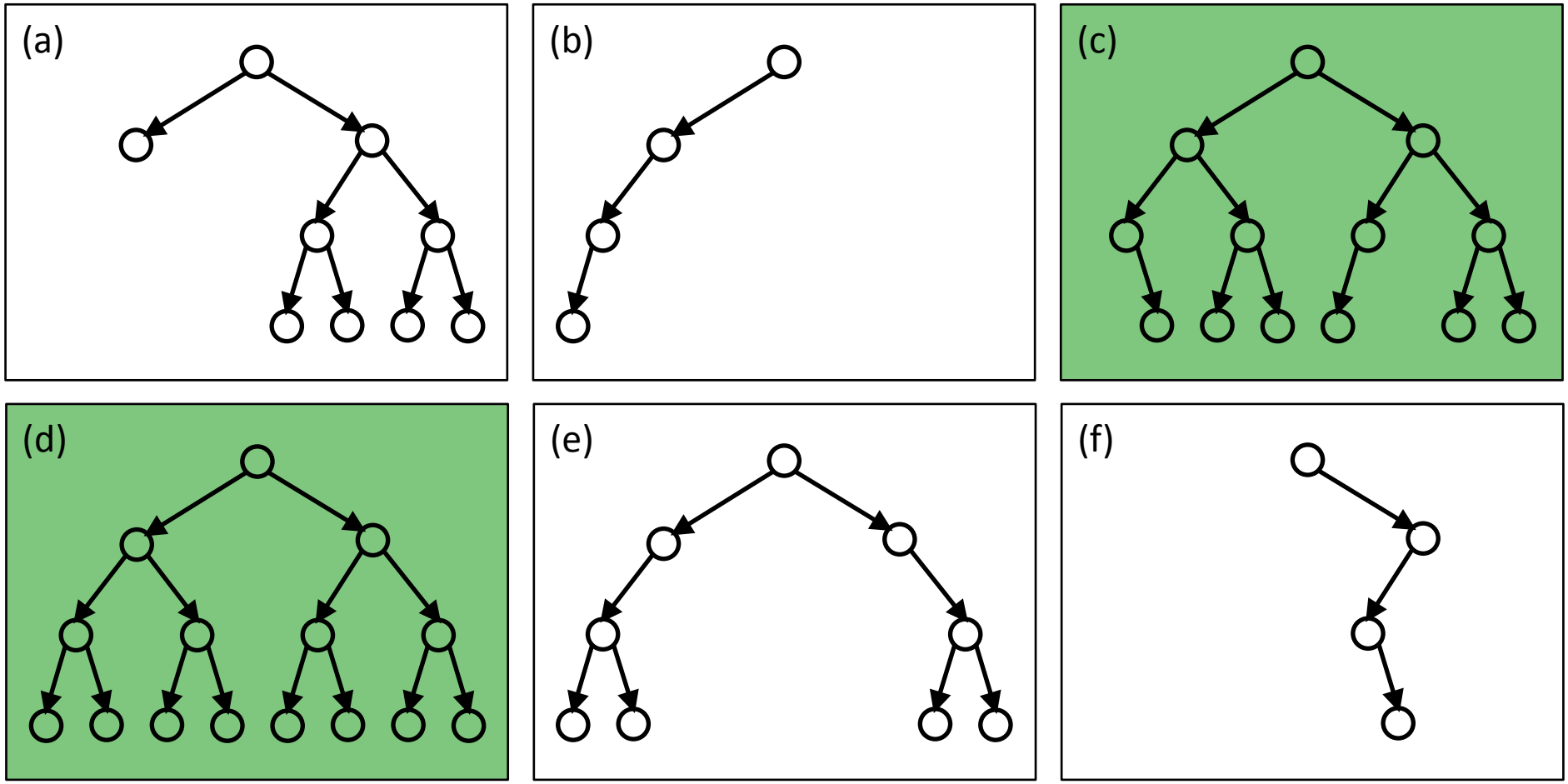
Group Exercise

Group the following trees according to characteristics you think are interesting



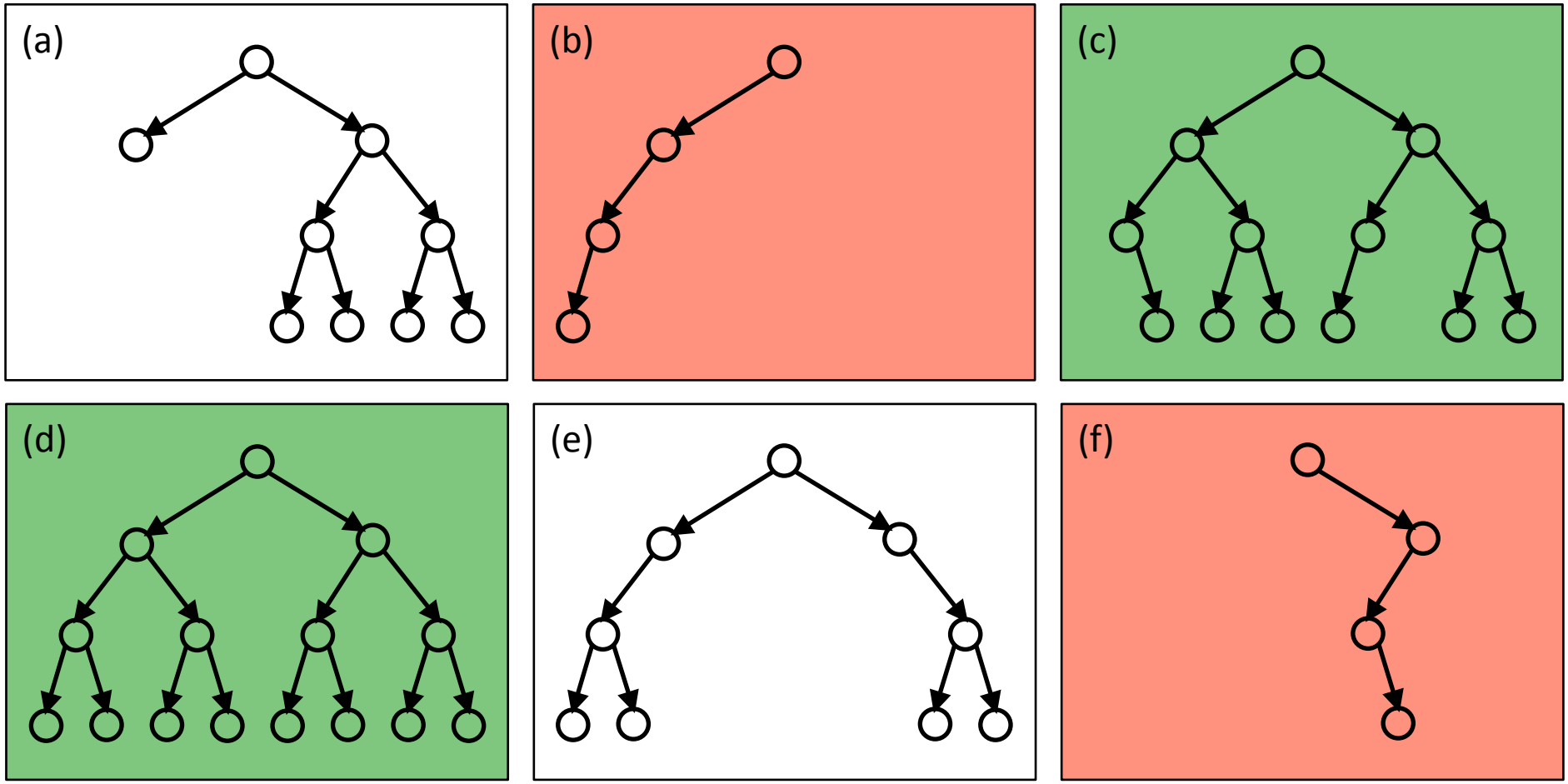
(Mostly) Balanced

branches differ by at most 1



(Mostly) Balanced branches differ by at most 1

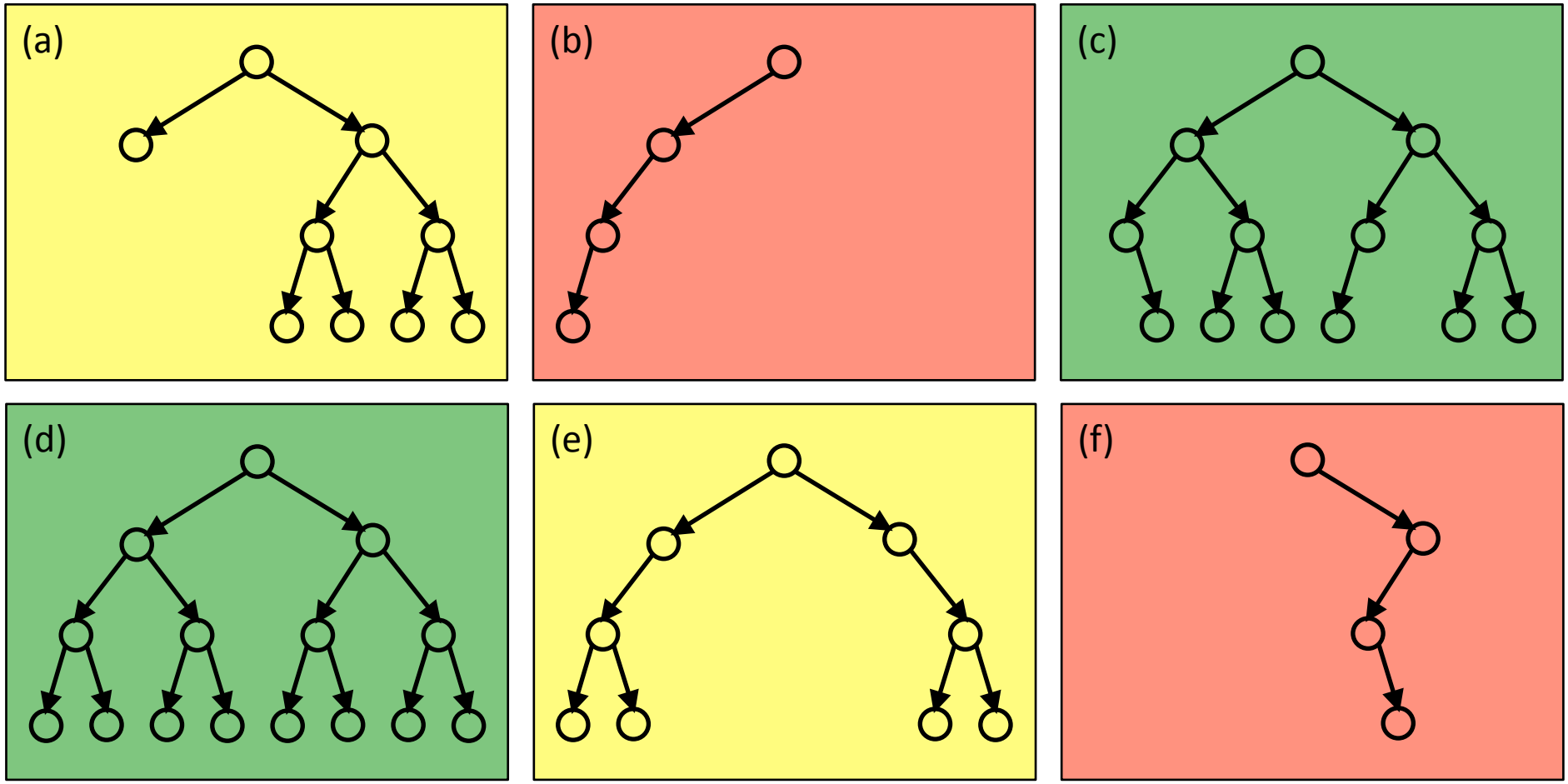
(Really) Unbalanced just crooked linked lists!



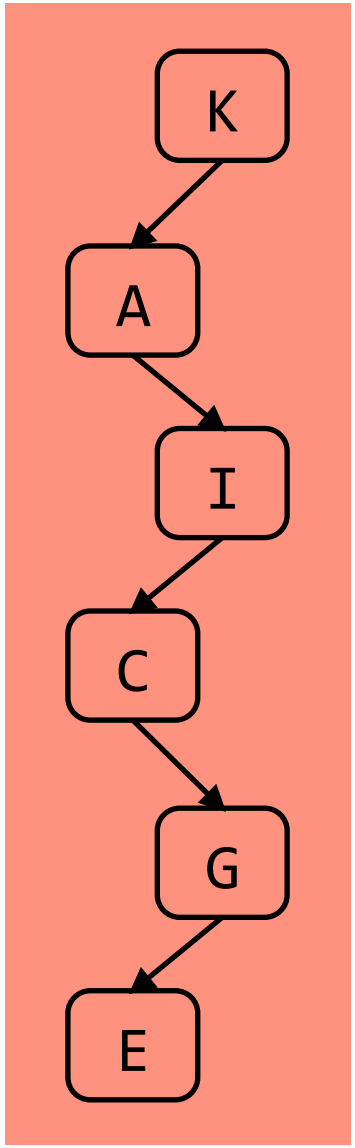
(Mostly) Balanced — branches differ by at most 1

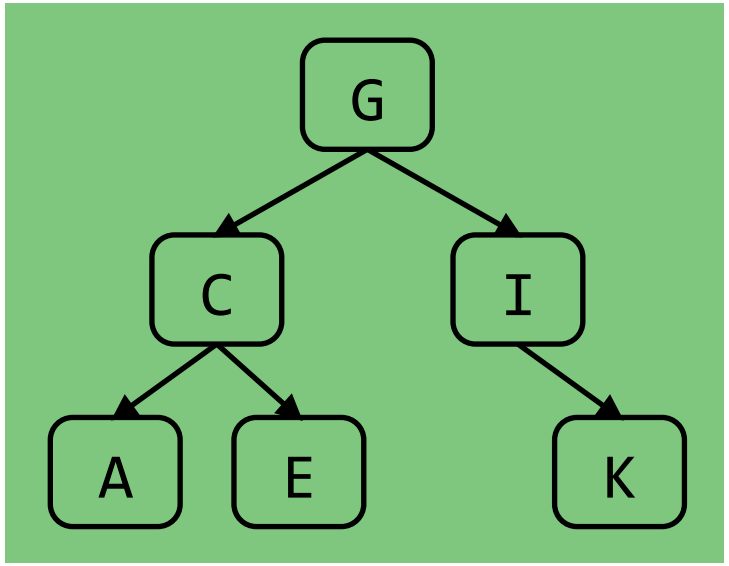
Unbalanced — in between two extremes

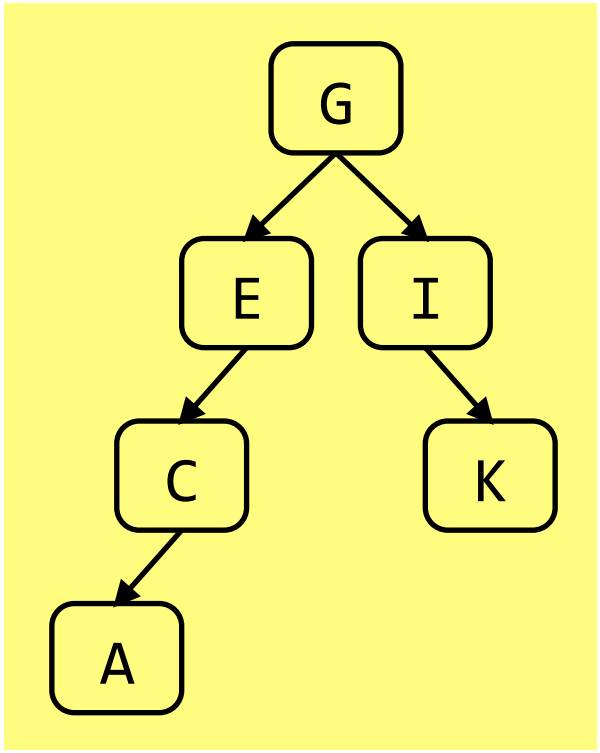
(Really) Unbalanced — just crooked linked lists!



Many Different BSTs for Same Data







Best Case	Avg. Case	Worst Case
??	*??*	*??*

```
bool find(str s) {
  if (s == this.friend) return true
  else if (s < this.friend)
    if (this.left == null) return false
    else return this.left.find(s)
  else // if (s > this.friend)
    if (this.right == null) return false
    else return this.right.find(s)
}
```

`find(s)`: How Many Nodes Traversed?

Data Structure + Invariants	Best Case	Avg. Case	Worst Case	
Lists	*1*	*≈ size*	*size*	
Lists + Sorted	*1*	*≈ size/2*	*size*	
Trees	*1*	*≈ size*	*size*	
Trees + Sorted	*1*	*≈ height*	*height*	

Height(Tree) <= Size(Tree)

If we're lucky: Height << Size

If we're not: Height == Size

`find(s)`: How Many Nodes Traversed?

Data Structure + Invariants	Best Case	Avg. Case	Worst Case	
Lists	*1*	*≈ size*	*size*	
Lists + Sorted	*1*	*≈ size/2*	*size*	
Trees	*1*	*≈ size*	*size*	
Trees + Sorted	*1*	*≈ height*	*height*	
Trees + Sorted + Balanced	*1*	*≈ height*	*height*	

Height(Balanced Tree) << Size(Balanced Tree) !

Recap: Introduction to Binary Trees

Binary Trees

- similar to linked lists but with two `next` pointers

Binary Search Trees

- smaller elements in `left` subtree
- bigger elements in `right` subtree

Balanced Binary Search Trees

- height of tree is much smaller than size
- average time to `find` is much faster than linked lists

We will learn how to talk about "efficiency" with mathematical rigor in CS 35 / CS 41

Recap: Introduction to Binary Trees

– combining data structure and invariants is the art of programming

– in CS 37 / CS 91, we'll see how PLs can help blend this spectrum!

Balanced Binary Search Trees

Food for Thought

- How to store data other than strings in a BST?
- How to define `insert/remove` for BSTs while maintaining balance?
- Relationship between binary search trees and the binary search algorithm?