http://yellkey.com/subject

To what degree do you believe the following statement: Nearly anybody enrolled at Berkeley could achieve an A in CS61B.

- Assume they can spend the entire summer preparing beforehand, hire a tutor during the semester, etc.

a. Strongly disagree
b. Disagree
c. Neutral
d. Agree
e. Strongly agree

Growth vs. Fixed Mindset

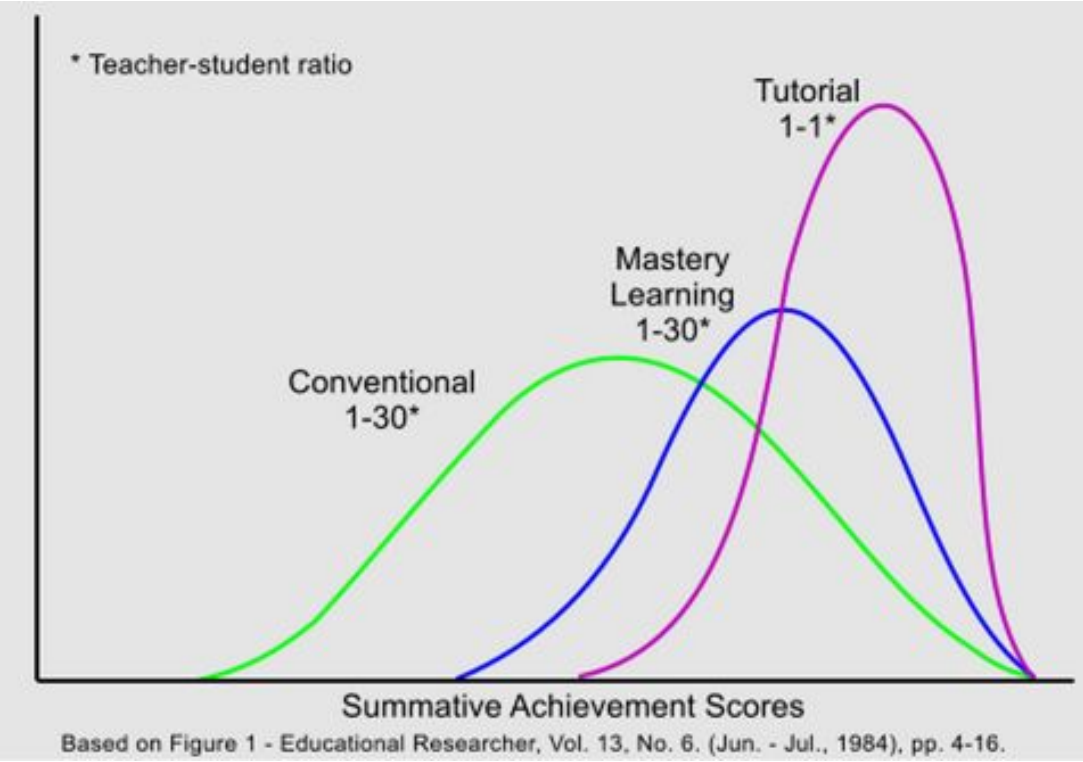
Students can be thought of as having either a “growth” mindset or a “fixed” mindset (based on research by Carol Dweck).

- “In a **fixed mindset** students believe their basic abilities, their intelligence, their talents, are just fixed traits. They have a certain amount and that's that, and then their goal becomes to look smart all the time and never look dumb.”
  - Perhaps most damningly, having to put in effort is a sign of weakness!
- “In a **growth mindset** students understand that their talents and abilities can be developed through effort, good teaching and persistence. They don't necessarily think everyone's the same or anyone can be Einstein, but they believe everyone can get smarter if they work at it.”

Growing up and even through undergrad I was very much part of the “fixed mindset” camp.

A Brief Aside

Bloom's Two Sigma Problem: In one experiment, student randomly picked for 1-on-1 teaching performed similarly to the top 2% of a simultaneous lecture course.



A Supporting Experiment

In Sp16, I gave students the option to fail intentionally.

- On average, students were 2.5 letter grades higher the second time, e.g. someone in the middle of the B range got an A the second time.

Possible interpretation: There exists some sort of preparation for 61B that helps students do much better.

Announcements

HW2 due next Wednesday.

- Will be out late tonight.
- Application of disjoint sets towards a cool physics problem.

Roundtable discussions about project 2 approximately next Thursday: (Link)

- If demand is high, will select people randomly.

Announcements

Project 2. Some lessons:

- API design is hard. Your first design is probably going to be ugly.
- Modular design is important.
- If something is ugly and works, you can refactor your code so that it still works and is less ugly.
- Your choice of classes and underlying data structures make a critical difference in how hard your code is to implement.
- Working on a team has a complex human dimension and can be anywhere between infuriating and wonderful.

For more software engineering, see CS169.

- Future HWs and project 3 will not feel quite like this one.

Announcements

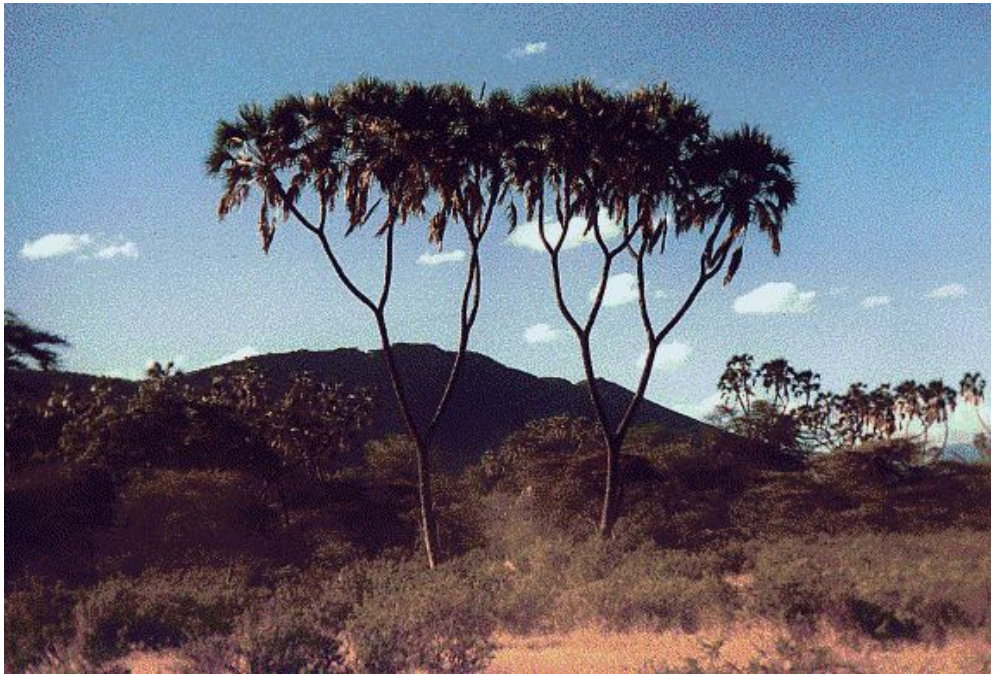
How should you do well on midterm 2?

- Go through the study guides soon after the relevant lecture.
- Work on study guide problems independently, and discuss solutions with others.

CS61B

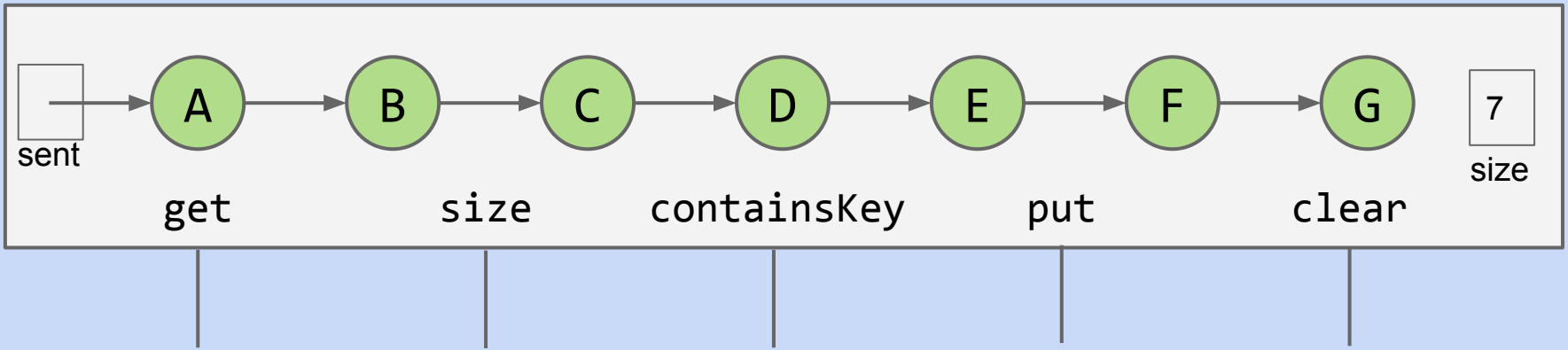
Lecture 21: Binary Search Trees

- Binary Search Tree (intro)
- BST Definitions
- BST Operations
- Performance



Analysis of an OLLMap<Character, ?>

In an earlier lecture, we implemented a map based on unordered arrays. For the ***order linked list*** map implementation below, name an operation that takes worst case linear time, i.e. Θ(N).

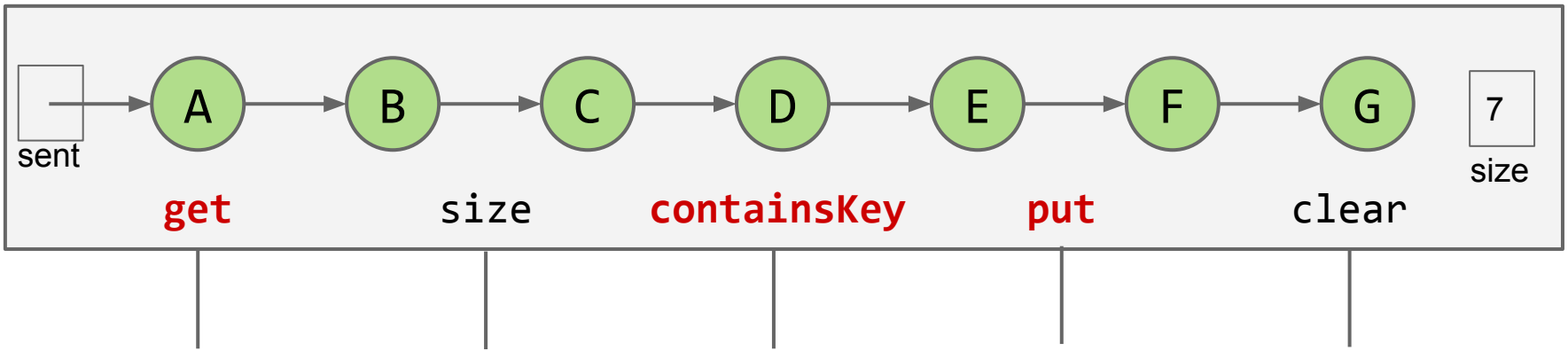


For space reasons, only keys are shown!

- Values are not relevant to the discussion today.

Analysis of an OLLMap<Character, ?>

In an earlier lecture, we implemented a map based on unordered arrays. For the ***order linked list*** map implementation below, name an operation that takes worst case linear time, i.e. Θ(N).



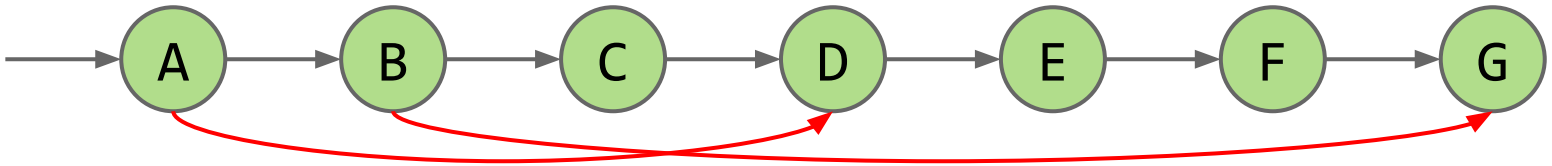
For space reasons, only keys are shown!

- Values are not relevant to the discussion today.

Optimization: Extra Links

Fundamental Problem: Slow search, even though it's in order.

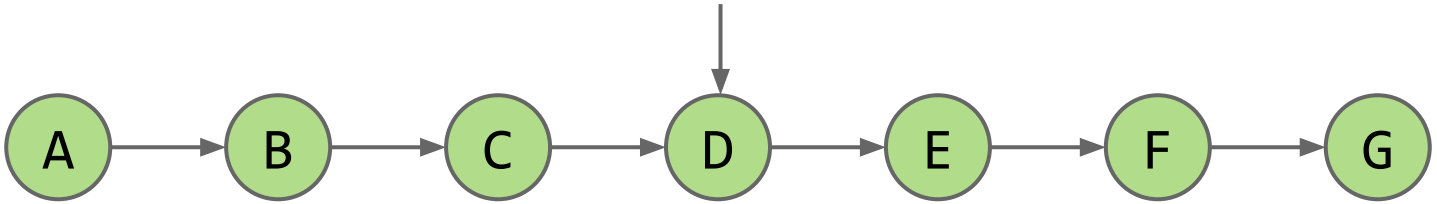
- Add (random) express lanes. Skip List (won't discuss in 61B)



Optimization: Change the Entry Point

Fundamental Problem: Slow search, even though it's in order.

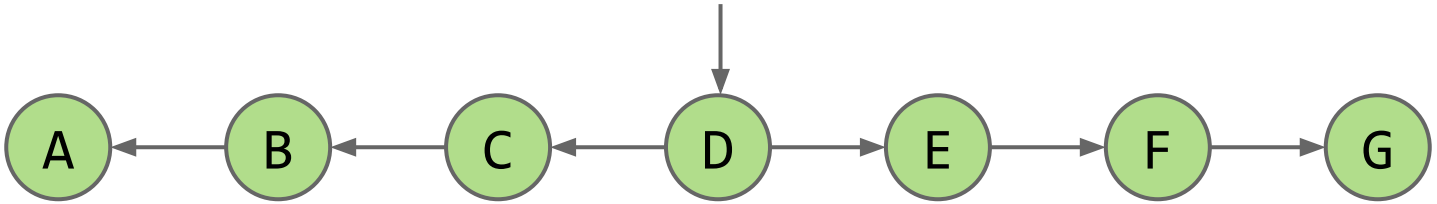
- Move pointer to middle.



Optimization: Change the Entry Point, Flip Links

Fundamental Problem: Slow search, even though it's in order.

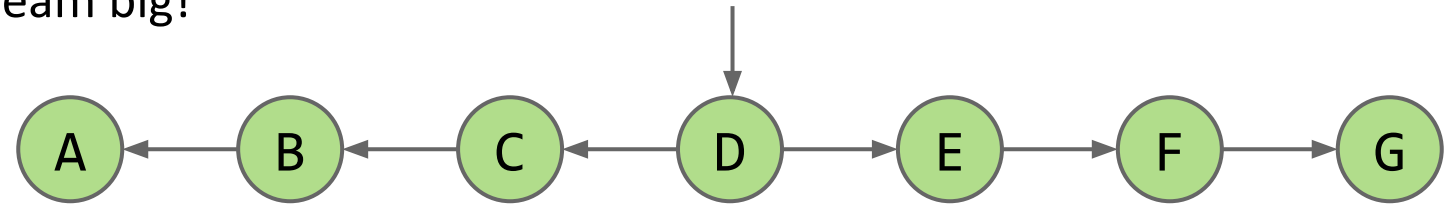
- Move pointer to middle and flip left links. Halved search time!



Optimization: Change the Entry Point, Flip Links

Fundamental Problem: Slow search, even though it's in order.

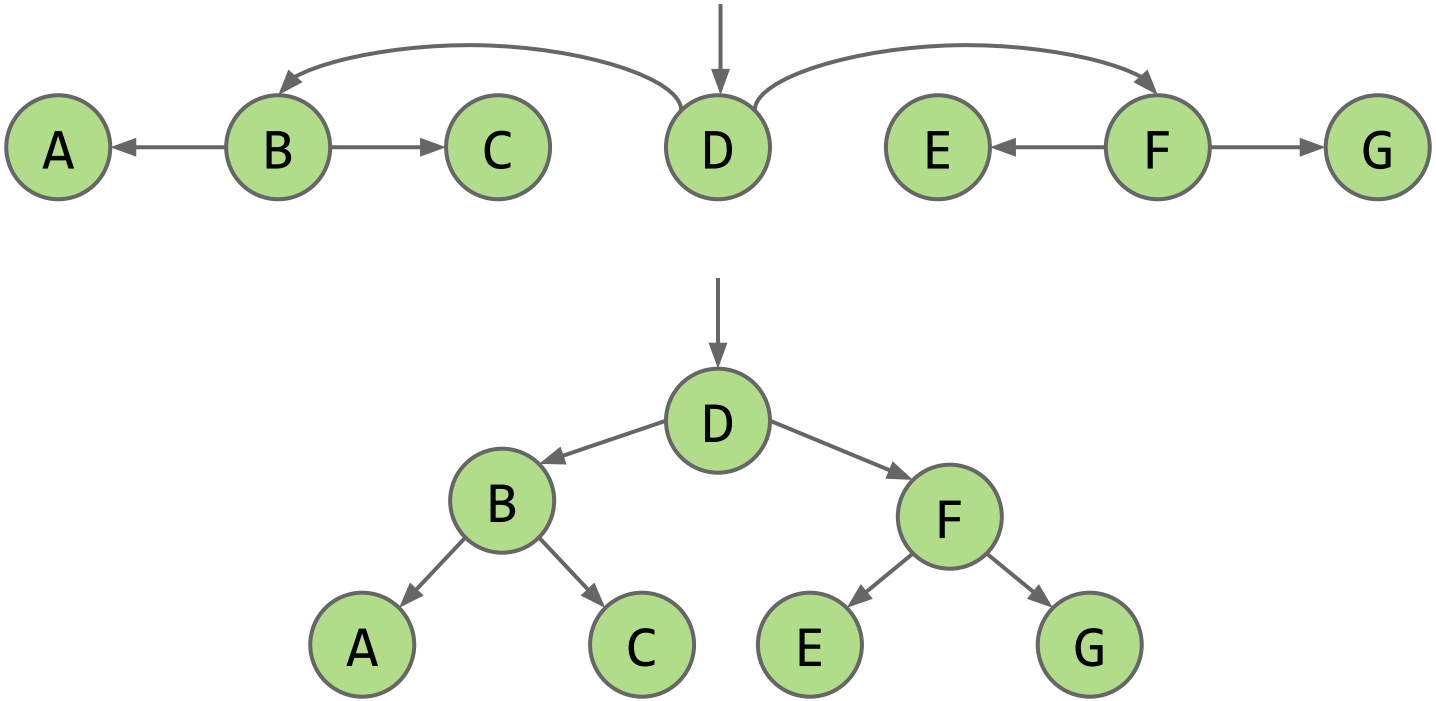
- How do we do even better?
- Dream big!



Optimization: Change Entry Point, Flip Links, Allow Big Jumps

Fundamental Problem: Slow search, even though it's in order.

- How do we do better?



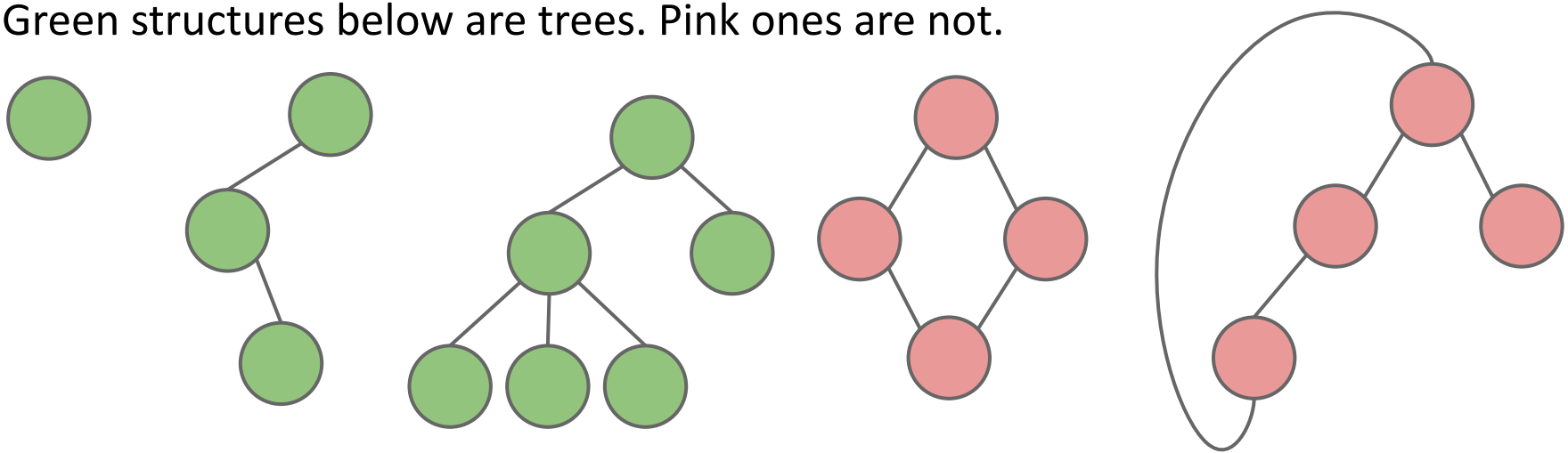
BST Definitions

Tree

A tree consists of:

- A set of nodes.
- A set of edges that connect those nodes.
  - Constraint: There is exactly one path between any two nodes.

Green structures below are trees. Pink ones are not.



Rooted Trees and Rooted Binary Trees

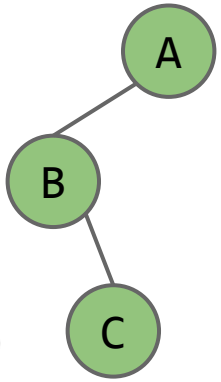
In a rooted tree, we call one node the root.

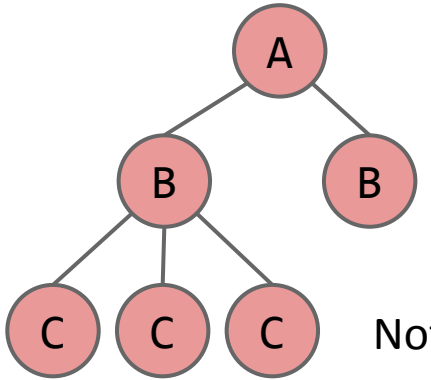
- Every node N except the root has exactly one parent, defined as the first node on the path from N to the root.
- Unlike (most) real trees, the root is usually depicted at the top of the tree.
- A node with no child is called a leaf.

In a rooted binary tree, every node has either 0, 1, or 2 children (subtrees).

For each of these:

- A is the root.
- B is a child of A. (and C of B)
- A is a parent of B. (and B of C)





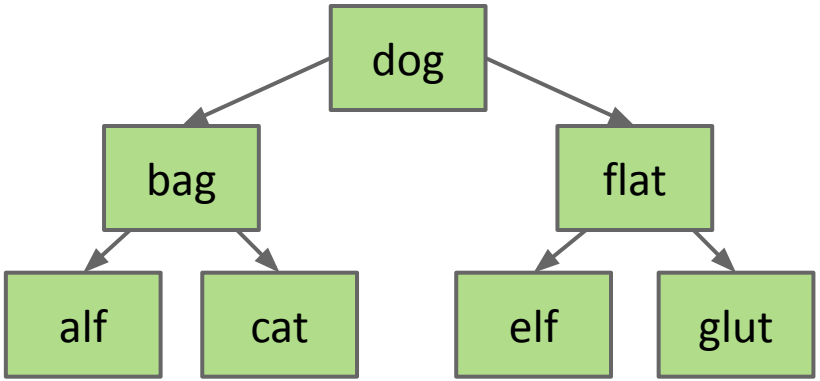
Not binary!

Binary Search Trees

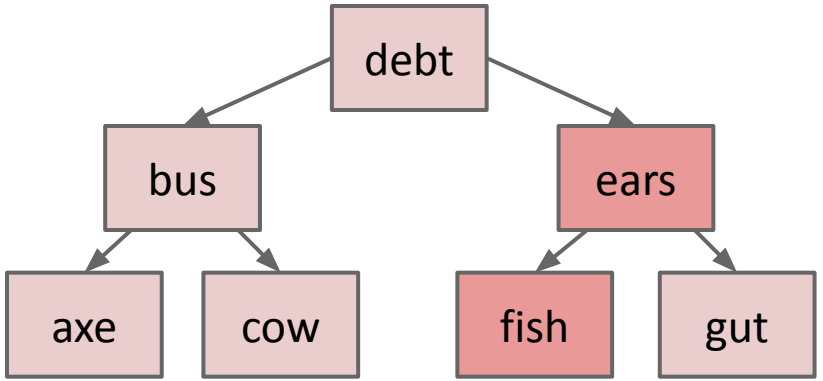
A binary search tree is a rooted binary tree with the BST property.

BST Property. For every node X in the tree:

- Every key in the **left** subtree is **less** than X's key.
- Every key in the **right** subtree is **greater** than X's key.



Binary Search Tree



Binary Tree, but not a Binary Search Tree

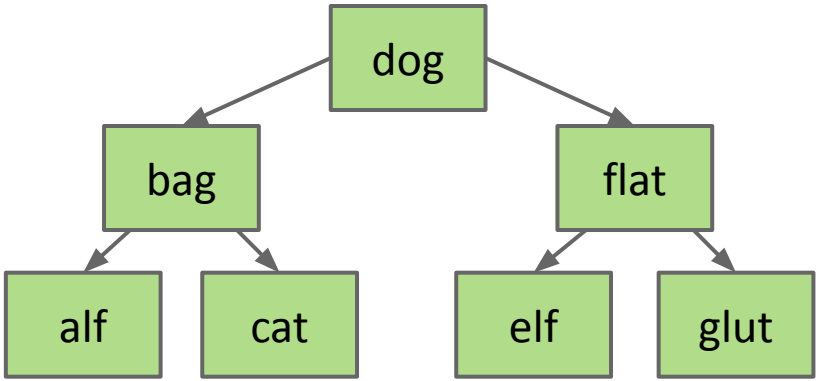
Binary Search Trees

Ordering must be complete, transitive, and antisymmetric. Given keys p and q:

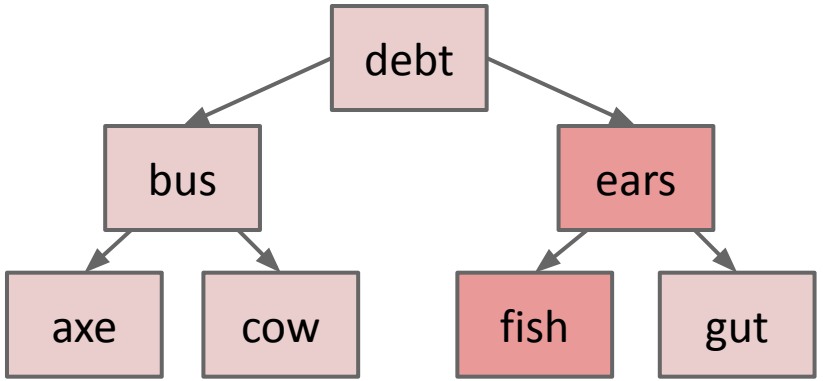
- Exactly one of $p \prec q$ and $q \prec p$ are true.
- $p \prec q$ and $q \prec r$ imply $p \prec r$.

One consequence of these rules: No duplicate keys allowed!

- Keeps things simple. Most real world implementations follow this rule.



Binary Search Tree



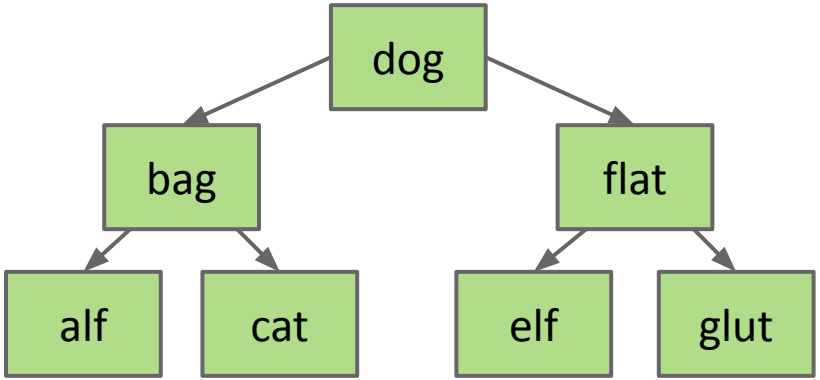
Binary Tree, but not a Binary Search Tree

BST Operations

Finding a searchKey in a BST (come back to this for the BST lab)

If searchKey equals label, return.

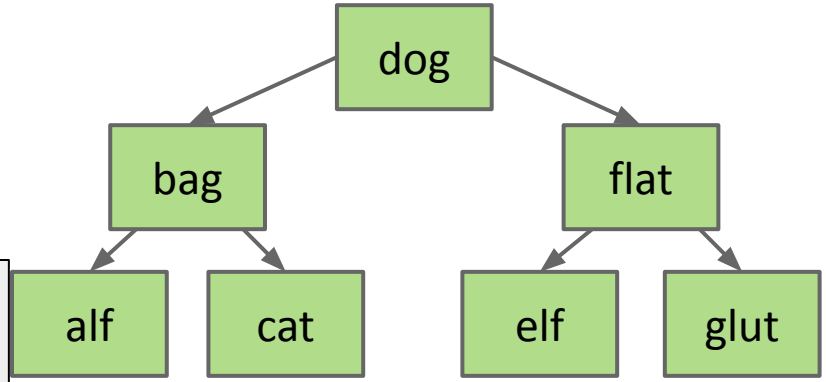
- If searchKey < label, search left.
- If searchKey > label, search right.



Finding a searchKey in a BST

If searchKey equals label, return.

- If searchKey < label, search left.
- If searchKey > label, search right.



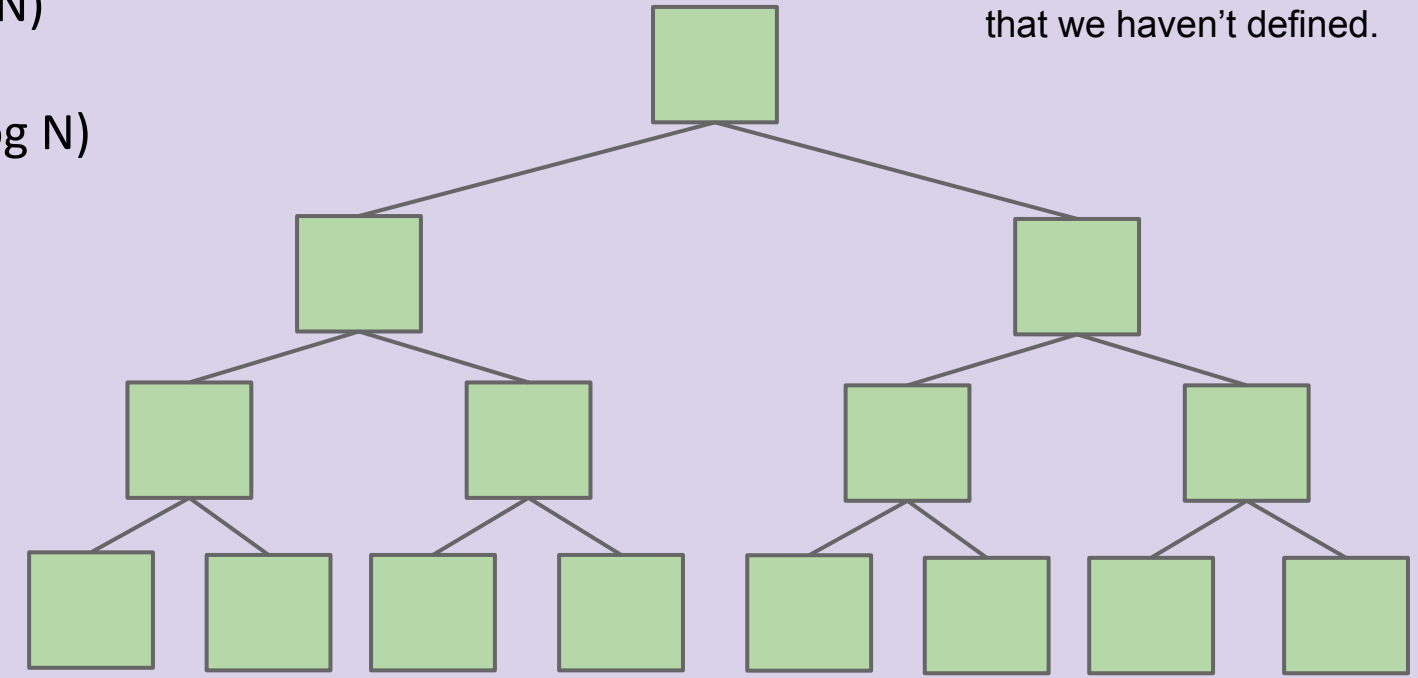

```
static BST find(BST T, Key sk) {
   if (T == null)
      return null;
   if (sk.keyequals(T.label()))
      return T;
   else if (sk < T.label())
      return find(T.left, sk);
   else
      return find(T.right, sk);
}
```

BST Search: http://yellkey.com/position

What is the runtime to complete a search on a "bushy" BST in the worst case, where N is the number of nodes.

A. $\Theta(\log N)$
B. $\Theta(N)$
C. $\Theta(N \log N)$
D. $\Theta(N^2)$
E. $\Theta(2^N)$

"bushiness" is an intuitive concept that we haven't defined.

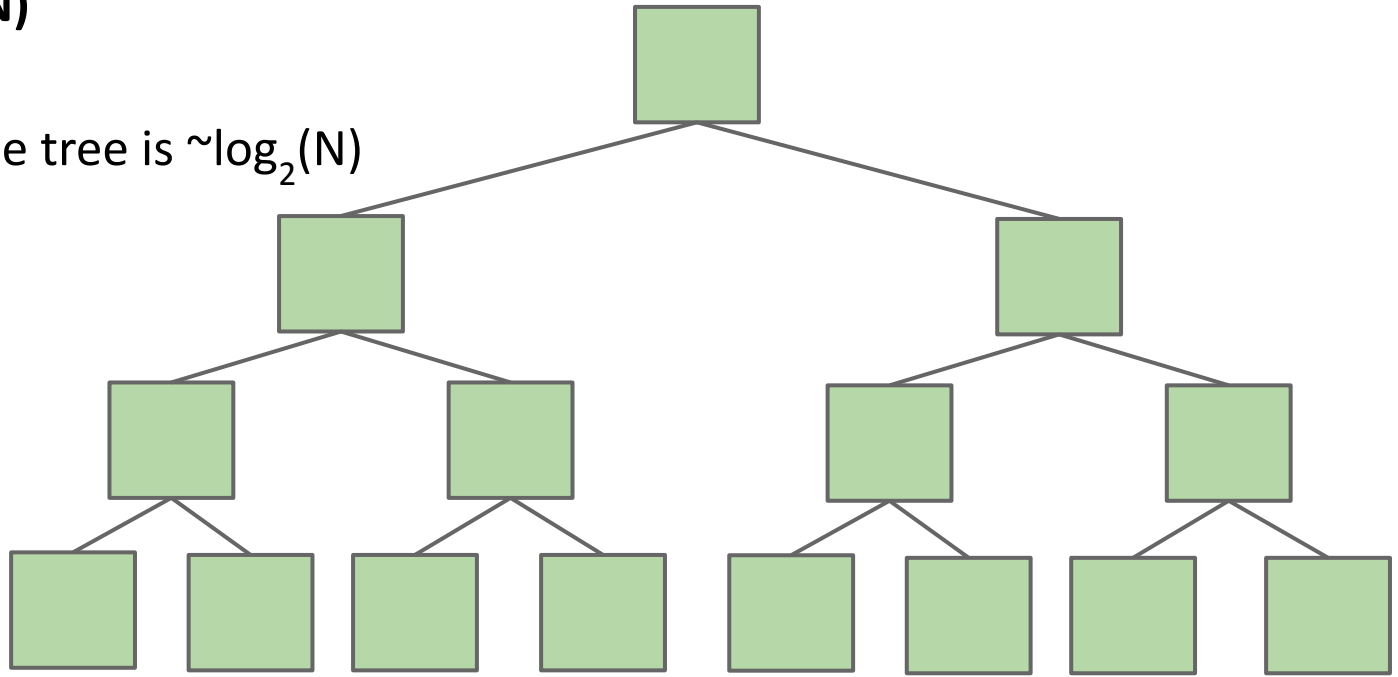


BST Search

What is the runtime to complete a search on a “bushy” BST in the worst case, where N is the number of nodes.

A. Θ(log N)

Height of the tree is ~$\log_2(N)$



BSTs

Bushy BSTs are extremely fast.

- At 1 microsecond per operation, can find something from a tree of size 10^{300000} in one second.

Much (perhaps most?) computation is dedicated towards finding things in response to queries.

- It's a good thing that we can do such queries almost for free.

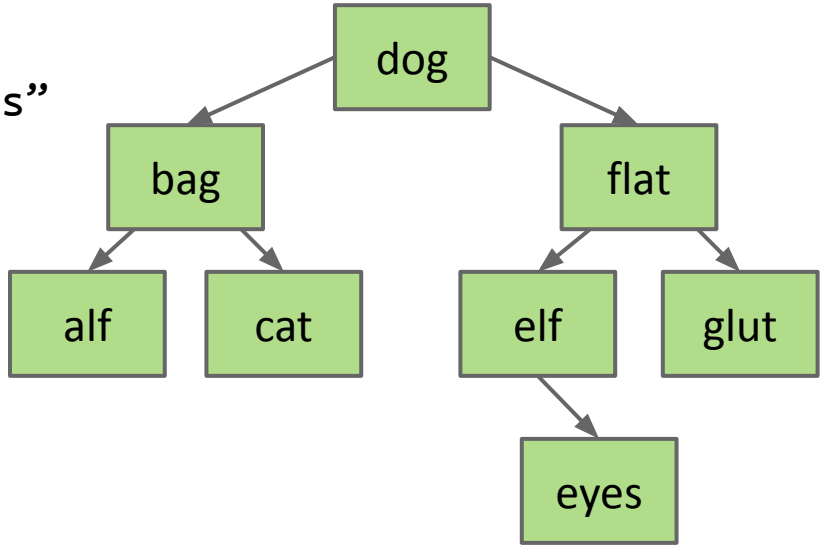
Inserting a new key into a BST

Search for key.

- If found, do nothing.
- If not found:
  - Create new node.
  - Set appropriate link.

Example:

`insert “eyes”`

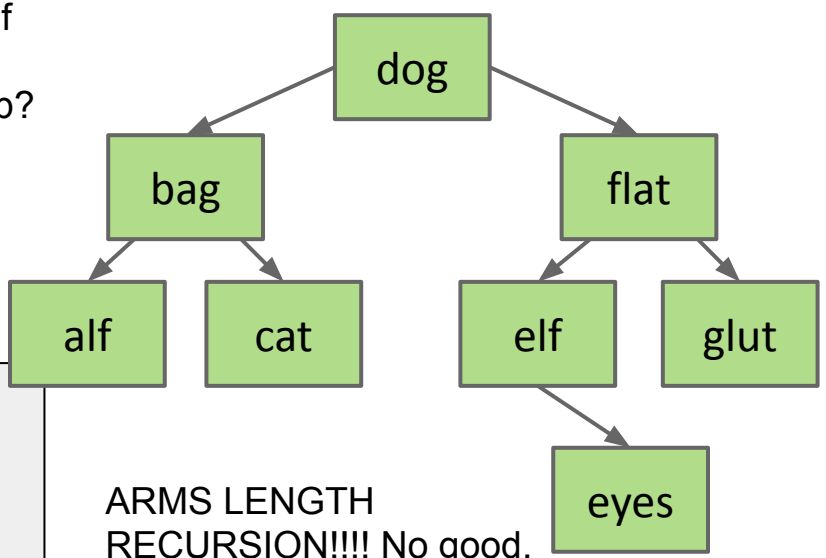


Inserting a new key into a BST

Search for key.

- If found, do nothing.
- If not found:
  - Create new node.
  - Set appropriate link.

What would we do if the BST was implementing a map?



ARMS LENGTH RECURSION!!!! No good.

```
static BST insert(BST T, Key ik) {
  if (T == null)
    return new BST(ik);
  if (ik < T.label()))
    T.left = insert(T.left, ik);
  else if (ik > T.label())
    T.right = insert(T.right, ik);
  return T;
}
```

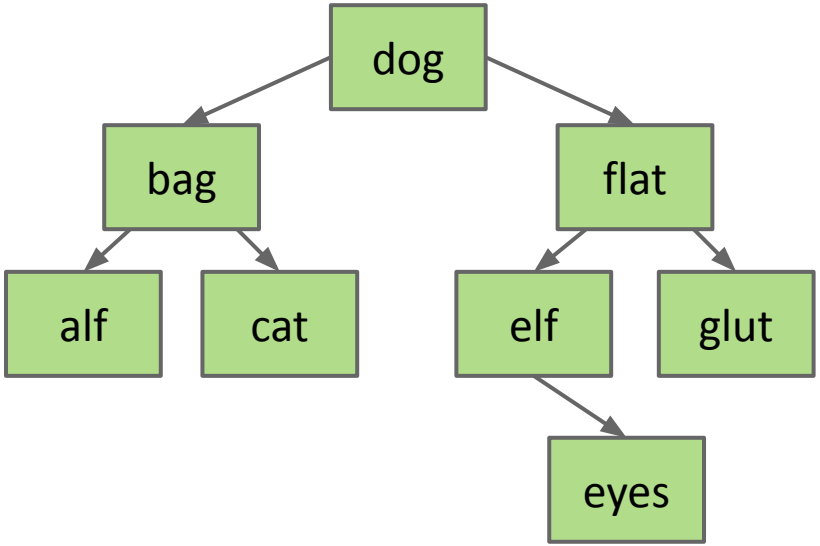
A common rookie bad habit to avoid:

```
if (T.left == null)
  T.left = new BST(ik);
else if (T.right == null)
  T.right = new BST(ik);
```

Deleting from a BST

3 Cases:

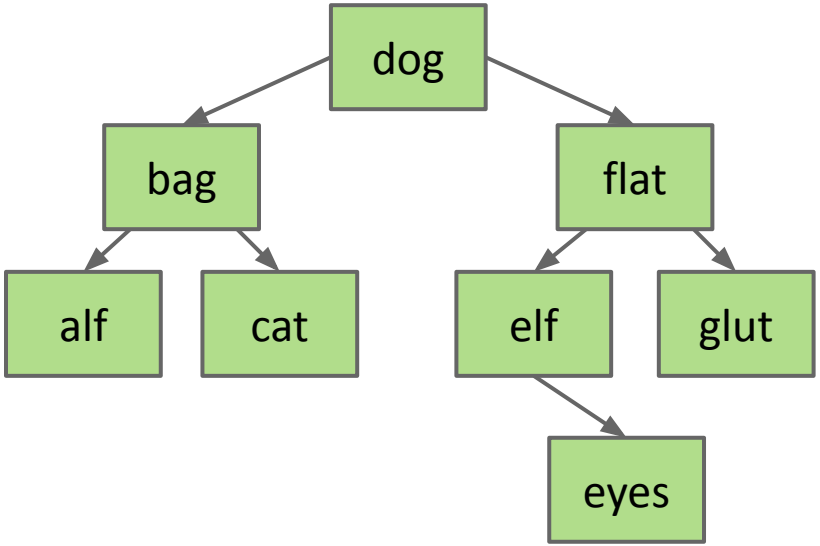
- Deletion key has no children.
- Deletion key has one child.
- Deletion key has two children.



Case 1: Deleting from a BST: Key with no Children

Deletion key has no children (“glut”):

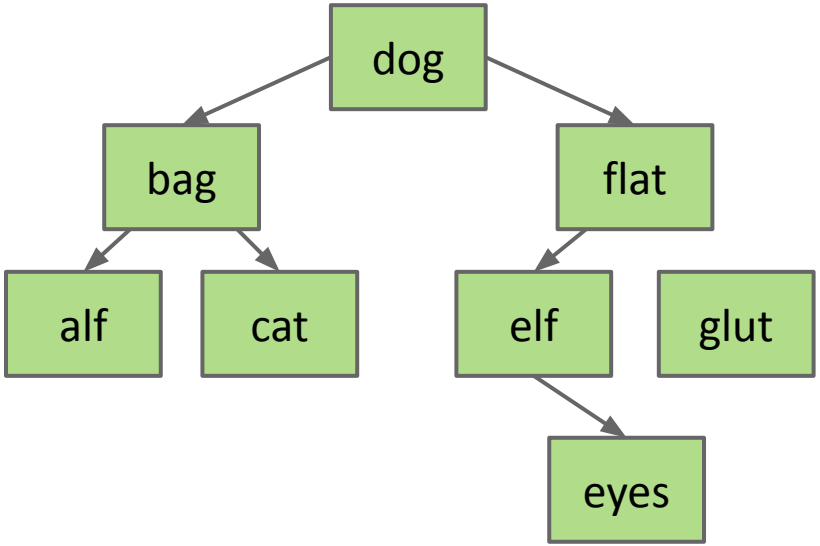
- Just sever the parent’s link.
- What happens to “glut” node?



Case 1: Deleting from a BST: Key with no Children

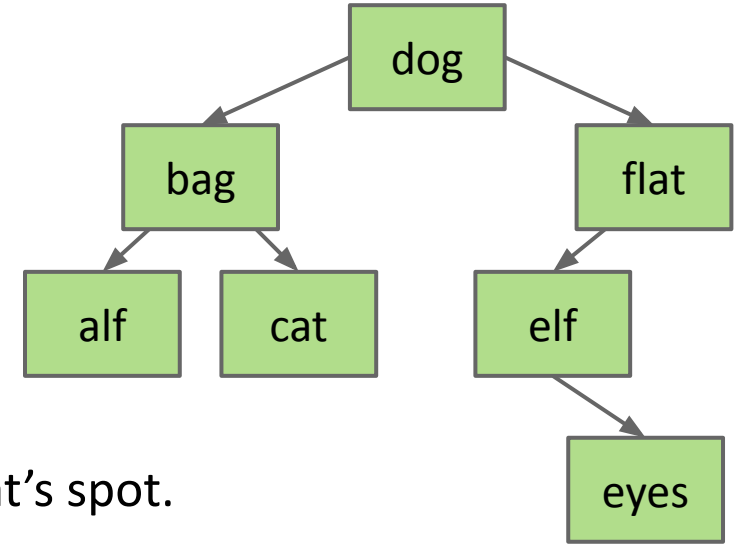
Deletion key has no children (“glut”):

- Just sever the parent’s link.
- What happens to “glut” node?
  - Garbage collected.



Case 2: Deleting from a BST: Key with one Child

Example: delete("flat"):



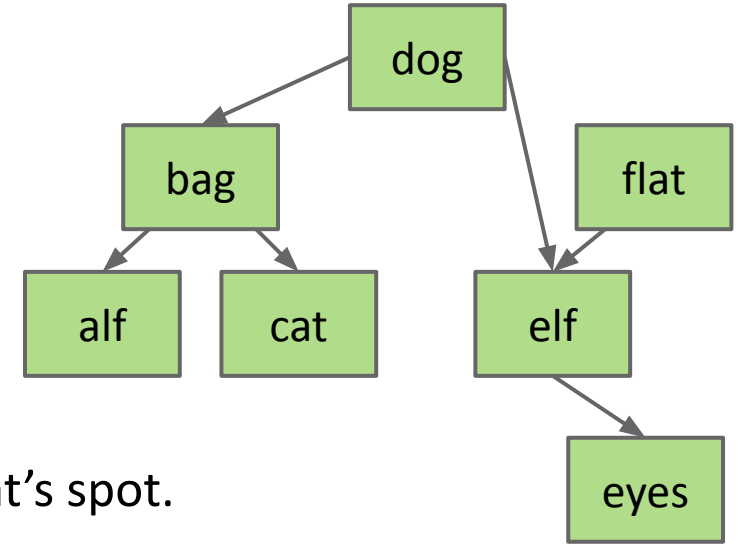
Goal:

- Maintain BST property.
- Flat's child definitely larger than dog.
  - Safe to just move that child into flat's spot.

Thus: Move flat's parent's pointer to flat's child.

Case 2: Deleting from a BST: Key with one Child

Example: delete("flat"):



Goal:

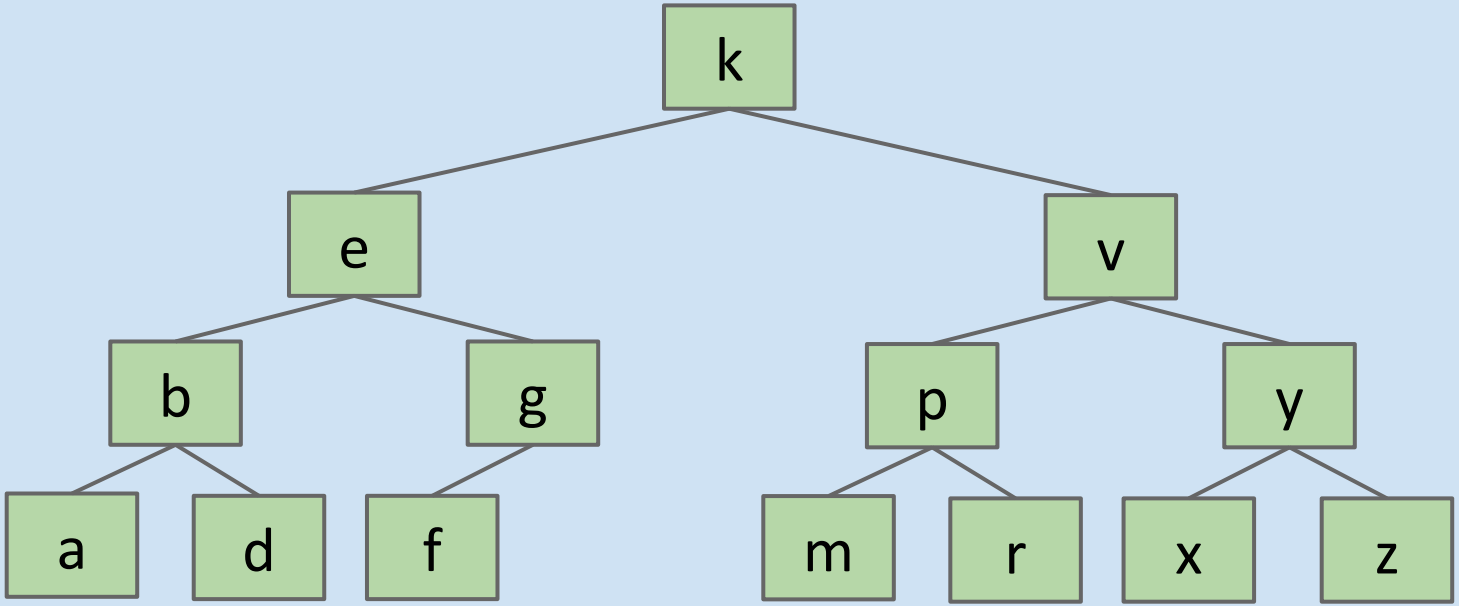
- Maintain BST property.
- Flat's child definitely larger than dog.
  - Safe to just move that child into flat's spot.

Thus: Move flat's parent's pointer to flat's child.

- Flat will be garbage collected (along with its instance variables).

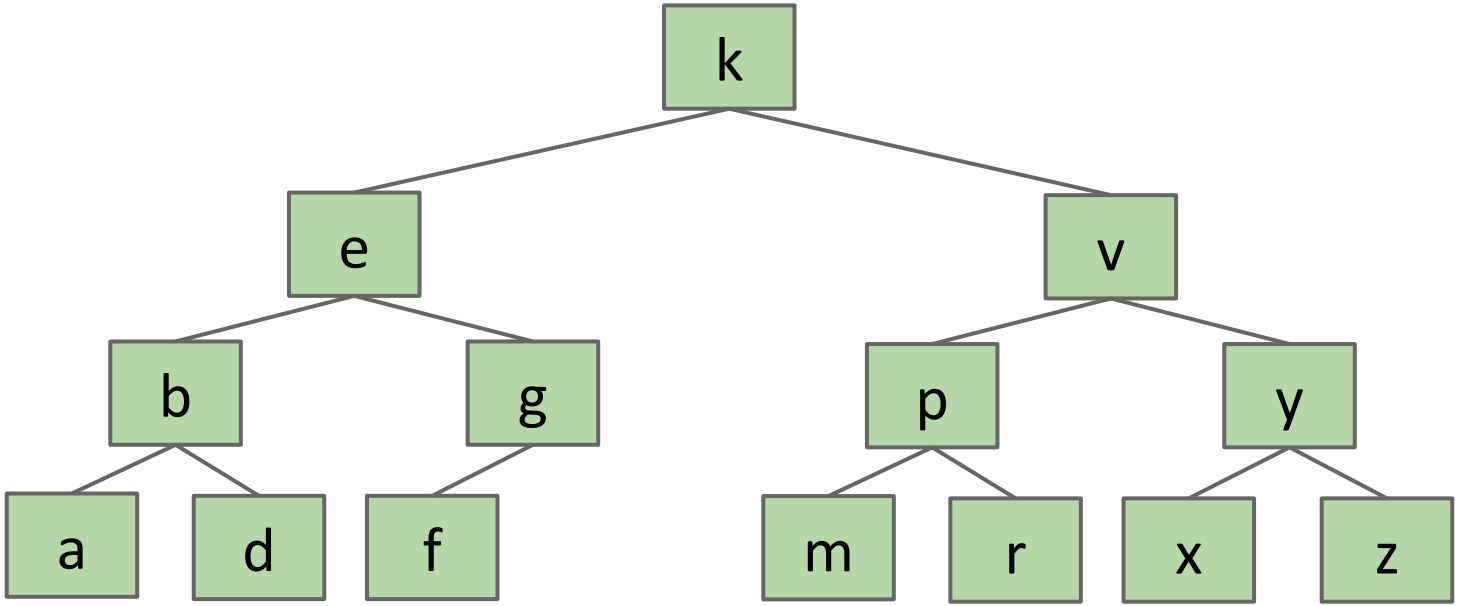
Hard Challenge

Delete k.



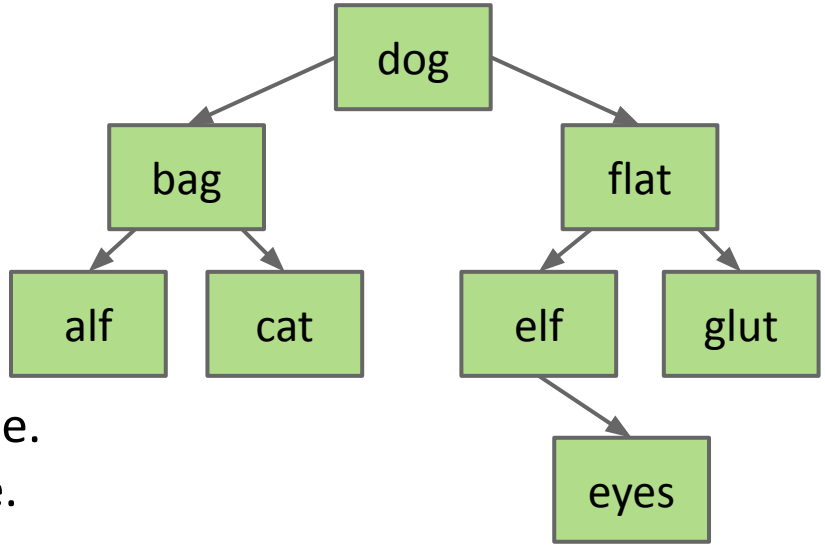
Hard Challenge

Delete k.



Case 3: Deleting from a BST: Deletion with two Children (Hibbard)

Example: delete(“dog”)



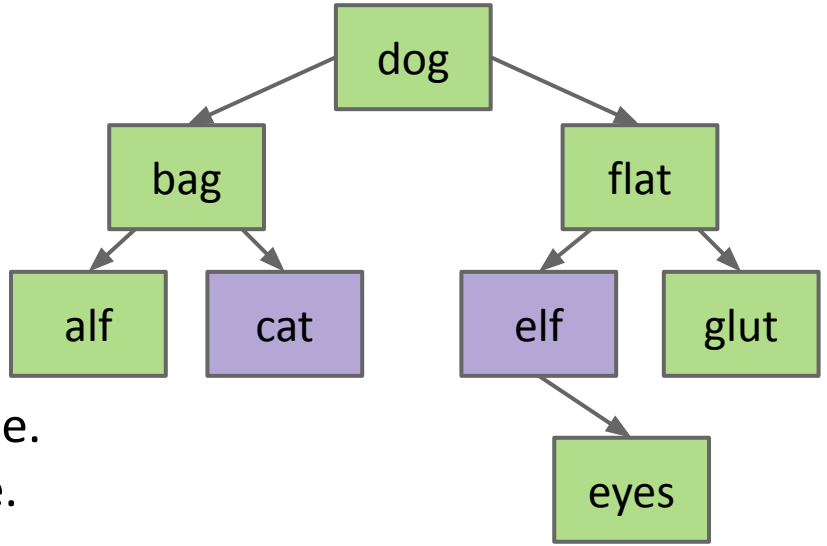
Goal:

- Find a new root node.
- Must be > than everything in left subtree.
- Must be < than everything right subtree.

Would bag work?

Case 3: Deleting from a BST: Deletion with two Children (Hibbard)

Example: delete(“dog”)



Goal:

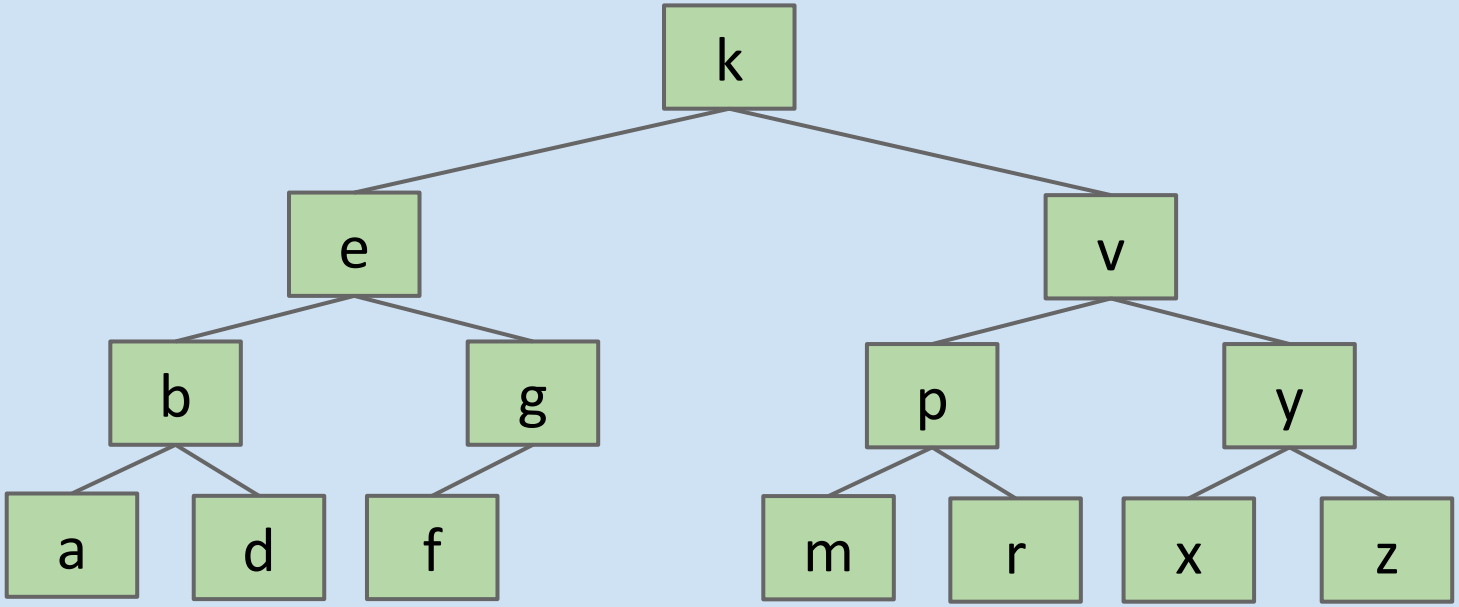
- Find a new root node.
- Must be > than everything in left subtree.
- Must be < than everything right subtree.

Choose either predecessor (“cat”) or successor (“elf”).

- Delete “cat” or “elf”, and stick new copy in the root position:
  - This deletion guaranteed to be either case 1 or 2. Why?
- This strategy is sometimes known as “Hibbard deletion”.

Hard Challenge (Hopefully Now Easy)

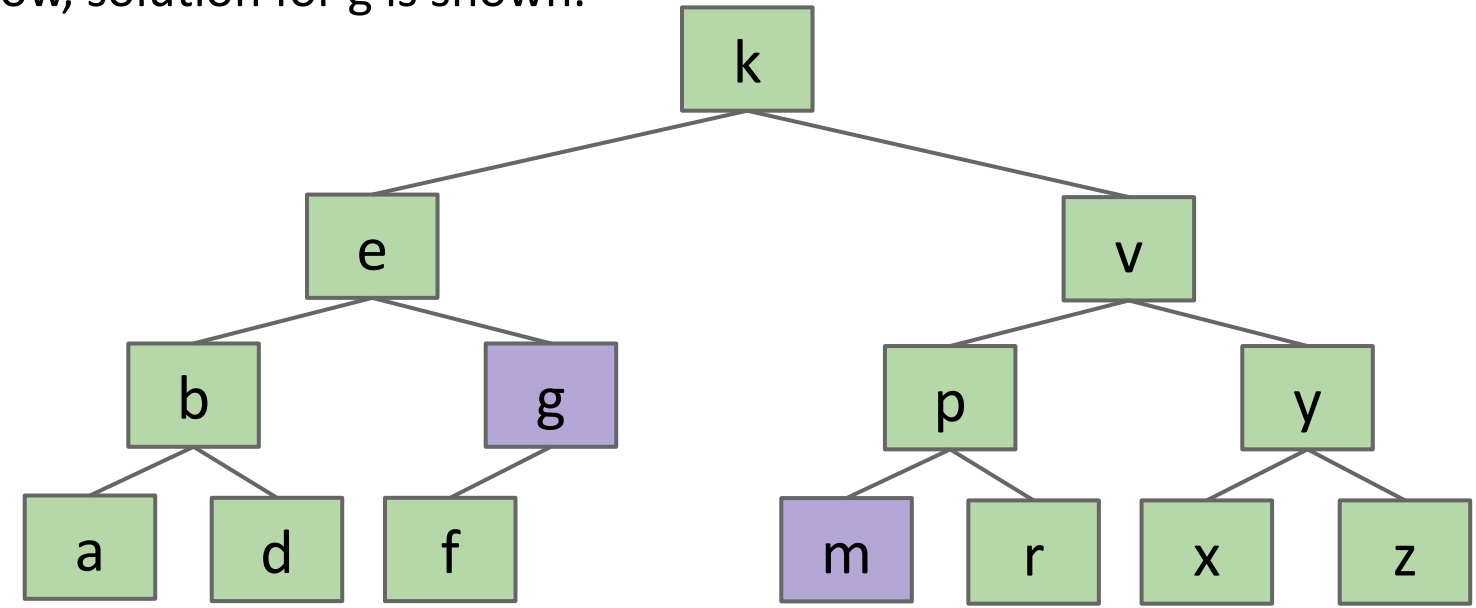
Delete k.



Hard Challenge (Hopefully Now Easy)

Delete k. Two solutions: Either promote g or m to be in the root.

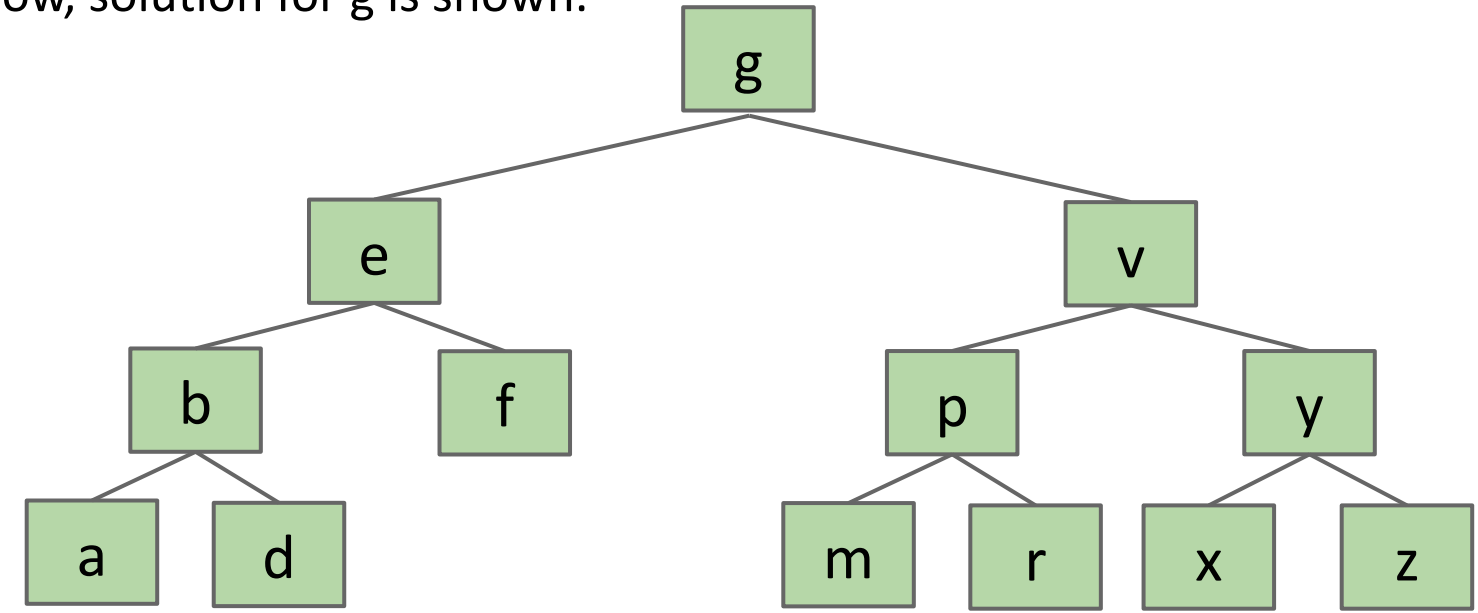
- Below, solution for g is shown.



Hard Challenge (Hopefully Now Easy)

Two solutions: Either promote g or m to be in the root.

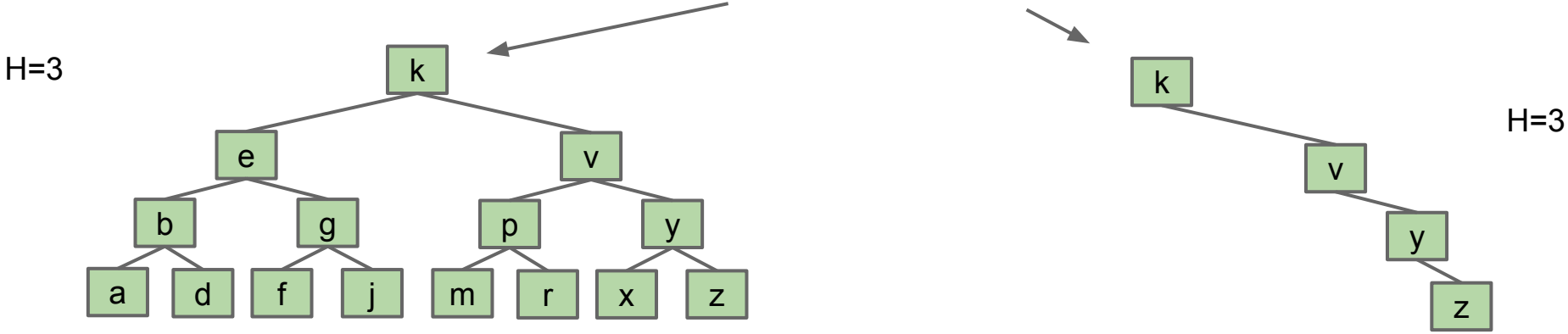
- Below, solution for g is shown.



BST Performance

Tree Height

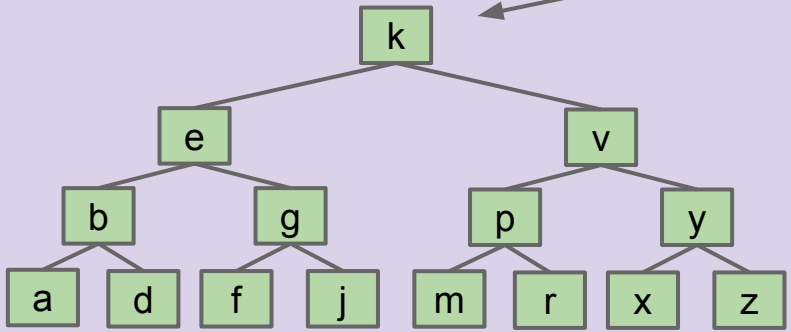
Height varies dramatically between “bushy” and “spindly” trees.

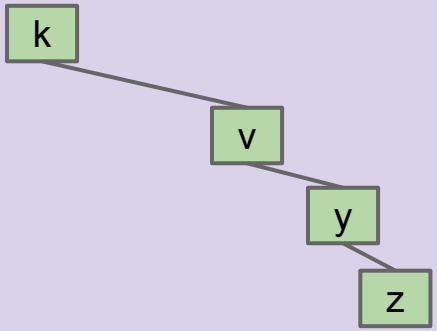


Tree Height: http://yellkey.com/type

Height varies dramatically between “bushy” and “spindly” trees.

H=3





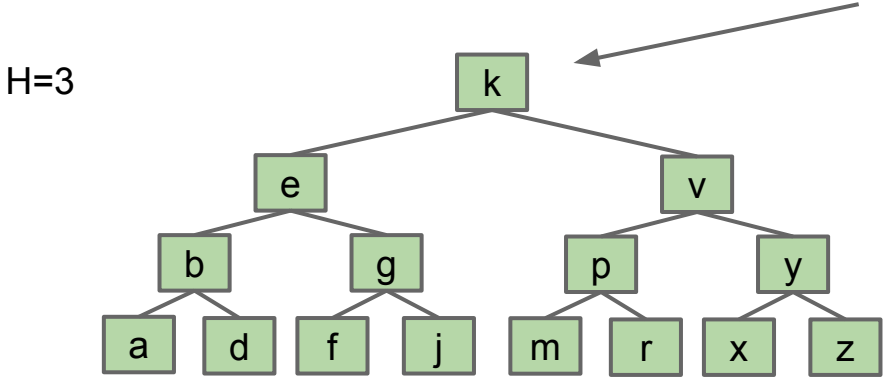
H=3

Let H(N) be the height of a tree with N nodes. Give H(N) in Big-Theta notation for “bushy” and “spindly” trees, respectively:

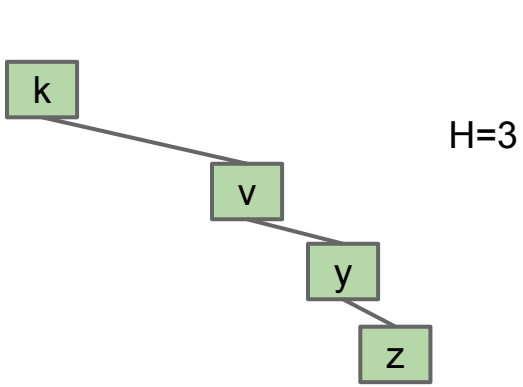
A. Θ(log(N)), Θ(log(N))
B. Θ(log(N)), Θ(N)
C. Θ(N), Θ(log(N))
D. Θ(N), Θ(N)

Tree Height

Height varies dramatically between “bushy” and “spindly” trees.



H = Θ(log(N))

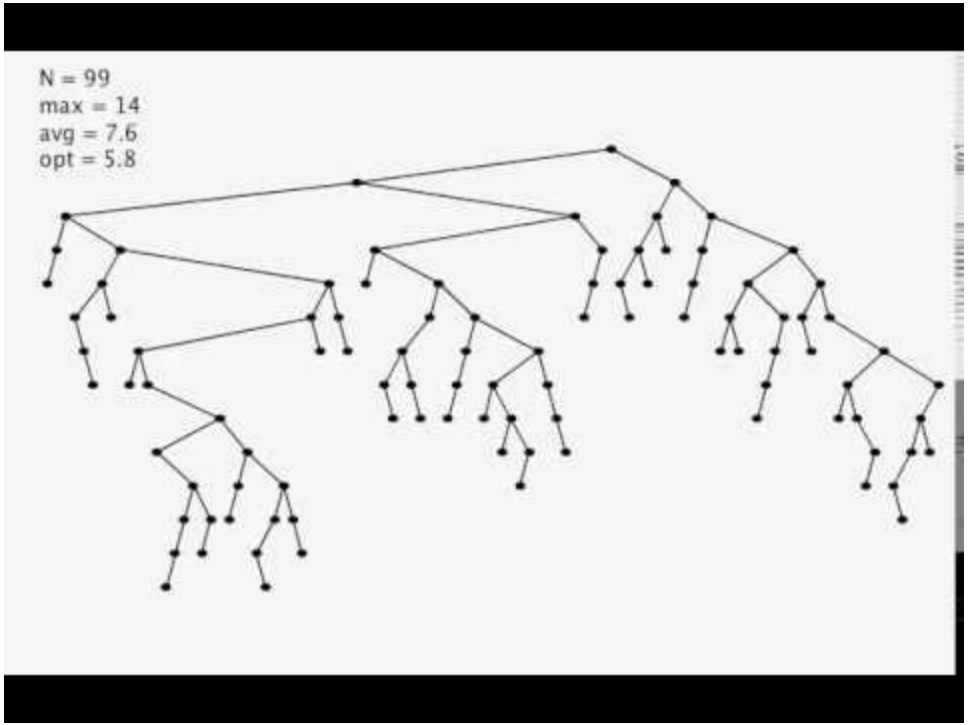


H = Θ(N)

Performance of spindly trees can be just as bad as a linked list!

- Example: containsKey(“z”) would take linear time.

BST Insertion Demo



Nice Property. Random inserts take on average only Θ(log N) each.

Video courtesy of Kevin Wayne (Princeton University)

BST: Mathematical Analysis

Comparison Counting. If N distinct keys are inserted into a BST, the expected average number of compares per insert is $C(N) \sim 2 \ln N = \Theta(\log N)$

- Will discuss this proof briefly towards the end of this course.

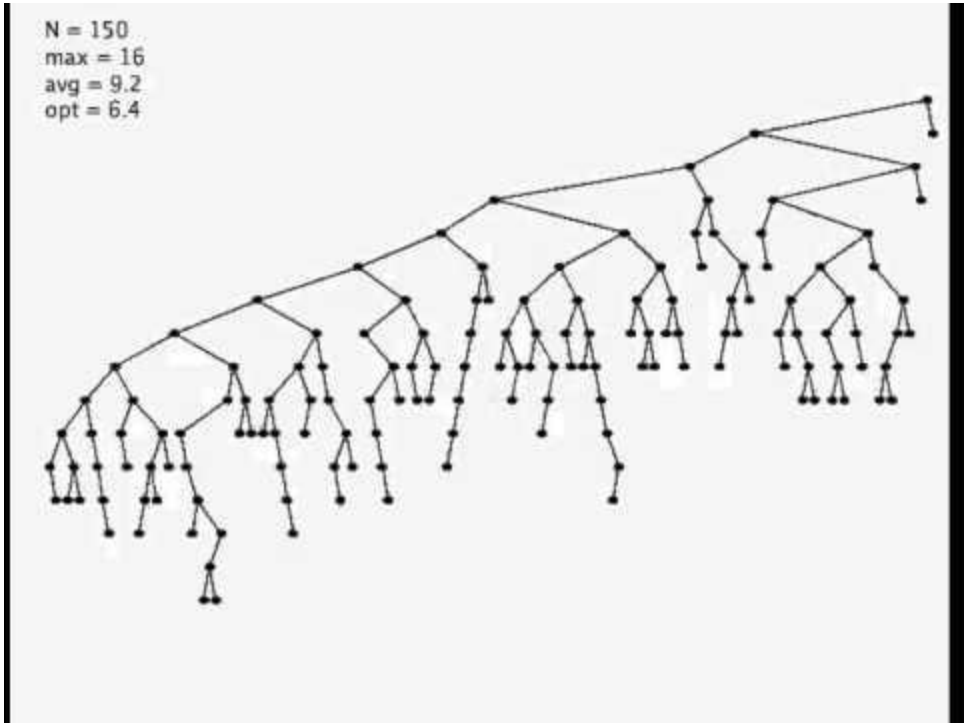
Tree Height. If N distinct keys are inserted in random order, expected tree height $H(N) \sim 4.311 \ln N$ (see Reed, 2003).

Recall tilde notation from Asymptotics 3 lecture:

- Similar to BigTheta, but don't throw away the multiplicative constant.

Formal definition: $f(x) \sim g(x)$ means that $\lim_{x \to \infty} \frac{f(x)}{g(x)} = 1$

BST Deletion Demo



Surprising Fact. Trees not balanced! C(N) ~ sqrt(N) per operation.

Open Problem. Find a simple and efficient delete for BSTs.

Video courtesy of Kevin Wayne (Princeton University)

Summary

Binary search trees: Efficient data structures for supporting insertion and search.

- Operations on “Bushy” BSTs are logarithmic time.
- Insertion of random data yields a bushy BST.
  - On random data, order of growth for get/put operations is logarithmic.

Performance issues:

- “Spindly” trees have linear performance.
- Hibbard deletion results in order of growth that is sqrt(N).
  - ~~Nobody knows how to do better on simple BSTs.~~ Can improve to log(N) by having Hibbard Deletion that randomly picks successor vs. predecessor.

Lab this week: Implementing a BSTMap.

Next time: Fixing these performance issues.

BST Implementation Tips

Tips for BST Lab

- Code from class was "naked recursion". Your BSTMap will not be.
- For each public method, e.g. `put(K key, V value)`, create a private recursive method, e.g. `put(K key, V value, Node n)`
- When inserting, always set left/right pointers, even if nothing is actually changing.
- Avoid "arms length base cases". Don't check if left or right is null!

```
static BST insert(BST T, Key ik) {
  if (T == null)
    return new BST(ik);
  if (ik < T.label()))
    T.left = insert(T.left, ik);
  else if (ik > T.label())
    T.right = insert(T.right, ik);
  return T;
}
```

Always set, even if nothing changes!

Avoid "arms length base cases".

```
if (T.left == null)
  T.left = new BST(ik);
else if (T.right == null)
  T.right = new BST(ik);
```

Citations

Probably photoshopped binary tree: http://cs.au.dk/~danvy/binaries.html

Demo movies for binary search tree operations: Kevin Wayne (Princeton University)