Binary Search Trees

CSE 332 Spring 2021

Instructor: Hannah C. Tang

Teaching Assistants:

Aayushi Modi	Khushi Chaudhari	Patrick Murphy
Aashna Sheth	Kris Wong	Richard Jiang
Frederick Huyan	Logan Milandin	Winston Jodjana
Hamsa Shankar	Nachiket Karmarkar	

Announcements

- Quiz 1 is due 11am PDT (not midnight!) tomorrow
- Also tomorrow: P2 partner matching survey
  - Must fill out even if you keep the same partner
- Thanks to your TAs, P1 is now due FRIDAY at 8pm PDT
  - Late policy is percentage-off, not late days
- We are always available for 1:1 meetings! Let us know how we can help!

gradescope

gradescope.com/courses/256241

- For a binary tree of height *h*:
  - max # of leaves:
  - max # of nodes:
  - min # of leaves:
  - min # of nodes:

- Bonus question: What is the difference between a plain *binary tree*, a *binary* ***search*** tree, and a *binary* ***min-heap*** *tree*?

Lecture Outline

- **Redo: Floyd's buildHeap**
- Review: Dictionary and Set ADTs
- Binary Trees != Binary Search Trees
  - Tree traversals
- Binary Search Trees as Dictionary/Set Data Structures
  - Find/Contains
  - Add/Remove

buildHeap

- `buildHeap`() takes an array of size N and applies the heap-ordering principle to it
  - Faster than naïve call-`add()`-N-times algorithm $\Theta(n \log n)$
- Intuition:
  - Start in the *middle* of the array (ie, the first non-leaf node) and work *backwards* (ie, up the tree)
  - Percolate *down* to fix each node's position relative to its valid subheaps
- Correctness and Efficiency:
  - Can prove correctness inductively
  - MostpercolateDown calls don't "go far", summation shows Θ(n)

Lecture Outline

- Redo: Floyd’s buildHeap
- **Review: Dictionary and Set ADTs**
- Binary Trees != Binary Search Trees
  - Tree traversals
- Binary Search Trees as Dictionary/Set Data Structures
  - Find/Contains
  - Add/Remove

Dictionary ADT: Data Structures

- For a dictionary with *n* key/value pairs, what is the runtime for:

	`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(log n)	O(n)
Sorted array	O(n)	O(log n)	O(n)

- * *Note*: If we allow duplicates keys to be inserted, you could do these in O(1) because you do not need to check for a key's existence before insertion

Reminder: a dictionary maps *keys* to *values*; an *item* or *data* refers to the (key, value) pair

Dictionary ADT: Better Data Structures

- We will spend the next several lectures looking at dictionaries:
  - Binary Search Trees
  - AVL trees
    - Binary search trees with guaranteed balancing
  - B-Trees
    - Also always balanced, but different and shallower
    - “B” != “Binary”; B-Trees generally have large branching factor
  - Hash Tables
    - Not tree-like at all

- Skipping: Other balanced binary search trees
  - Eg, red-black tree (and LLRBs), splay tree

Lecture Outline

- Redo: Floyd's buildHeap
- Review: Dictionary and Set ADTs
- **Binary Trees != Binary Search Trees**
  - Tree traversals
- Binary Search Trees as Dictionary/Set Data Structures
  - Find/Contains
  - Add/Remove

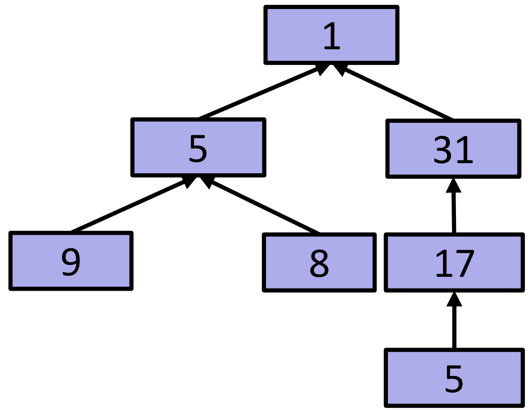
Binary Tree

recursive defn!

- A **Binary Tree** is empty or
  - a root *(with item)*
  - a left subtree *(maybe empty)*
  - a right subtree *(maybe empty)*

- Representation:

item	
left pointer	right pointer



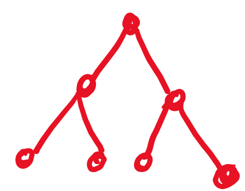
- For a dictionary, `item` will include a key and a value

Binary Tree: Some Numbers

- Recall: height of a tree = longest path from root to leaf
  - Count # of edges!

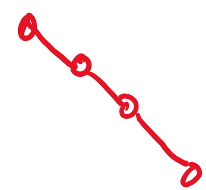
- For a binary tree of height *h*:
  - max # of leaves: 2^h
  - max # of nodes: $2^{h+1} - 1$

perfect tree



  - min # of leaves: **1**
  - min # of nodes: **h+1**

degenerate tree
linked list



Calculating Tree Height

- What is the height of a tree with root **`r`**?
- What is the runtime for your algorithm?

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

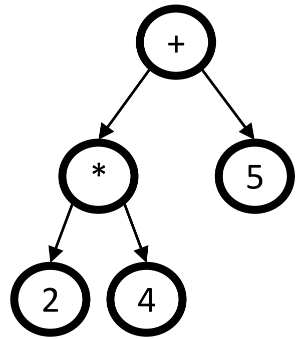
- *Note*: non-recursive is painful – need your own stack of pending nodes
  - Much easier to use recursion's call stack

Lecture Outline

- Redo: Analyzing Recursive Code
- Review: Dictionary and Set ADTs
- Binary Trees != Binary Search Trees
  - **Tree traversals**
- Binary Search Trees as Dictionary/Set Data Structures
  - Find/Contains
  - Add/Remove

Tree Traversals

- A *traversal* is an order for visiting all the nodes of a tree
  - *Pre-order*: **root**, left subtree, right subtree
    - + * 2 4 5
  - *In-order*: left subtree, **root**, right subtree
    - 2 * 4 + 5
  - *Post-order*: left subtree, right subtree, **root**
    -



(an expression tree)

- Sometimes order doesn't matter
  - Eg: sum all elements
  - Eg: find an element
- Sometimes order matters
  - Eg: print tree with indented children (pre-order)
  - Eg: evaluate an expression tree (post-order)

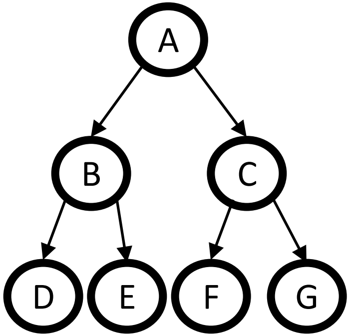
PrintIndented:

```
+
  *
    2
    4
  5
```

Traversals: Recursive Implementation

```
void inOrdertraversal(Node t) {
  if (t != null) {
    traverse(t.left);
    process(t.element);
    traverse(t.right);
  }
}
```

becomes postOrder!



Post order: D E B F G C A

- The difference between the 3 traversals (in their recursive implementations) is *when process() gets called*

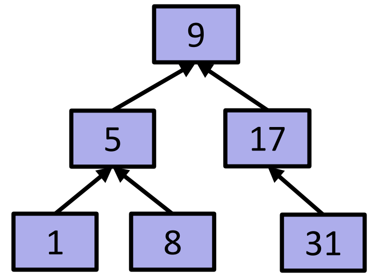
- Again, non-recursive implementation is painful

Lecture Outline

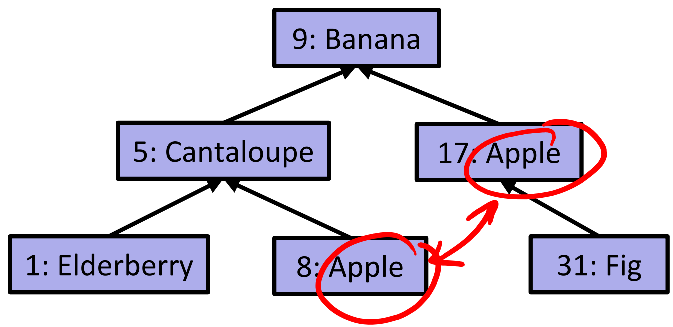
- Redo: Analyzing Recursive Code
- Review: Dictionary and Set ADTs
- Binary Trees != Binary Search Trees
  - Tree traversals
- **Binary Search Trees as Dictionary/Set Data Structures**
  - Find/Contains
  - Add/Remove

Binary *Search* Trees

- A **Binary Search Tree** is a binary tree with the following invariant: for every node with key k in the BST:
  - The left subtree only contains keys <k
  - The right subtree only contains keys >k



- Reminder: BSTs can also contain (key, value) pairs



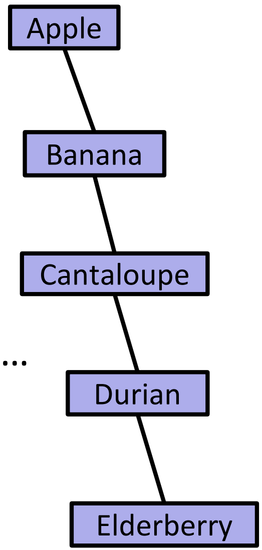
The BST ordering applies recursively to the entire subtree

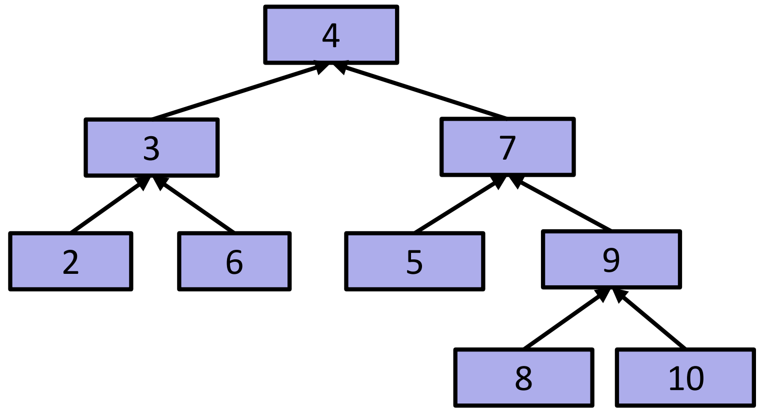
gradescope

gradescope.com/courses/256241

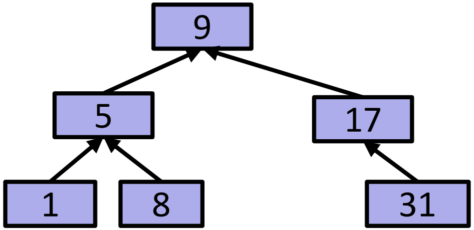
- Are these Binary Search Trees?

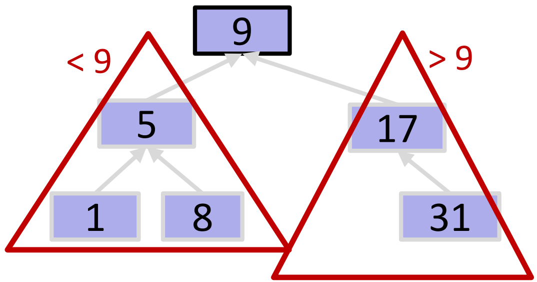
A. Yes / Yes
B. Yes / No
C. No / Yes
D. No / No
E. I’m not sure …

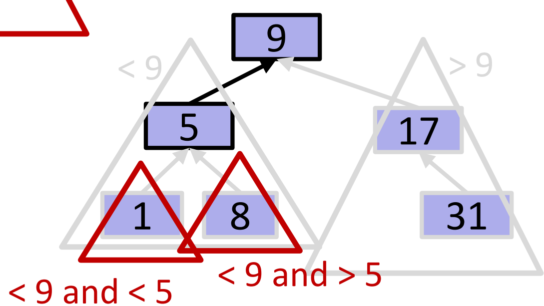




BST Ordering Applies *Recursively*





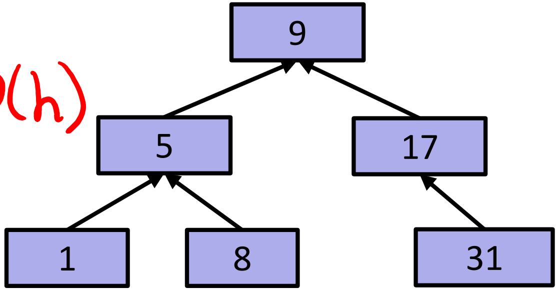


Lecture Outline

- Redo: Analyzing Recursive Code
- Review: Dictionary and Set ADTs
- Binary Trees != Binary Search Trees
  - Tree traversals
- Binary Search Trees as Dictionary/Set Data Structures
  - **Find/Contains**
  - Add/Remove

Binary Search Trees: Find/Contains

- Unsurprisingly, this looks a lot like binary search
- Can you implement contains() by putting the following statements in the correct order?
  - Hint: remember BST's invariants
- What is find's worst-case runtime? $\Theta(h)$




```
boolean contains(BSTNode n,
                 Key k) {

}
```

A

```
if (n == null)
  return false;
```

B

```
if (k.equals(n.key))
  return true;
```

C

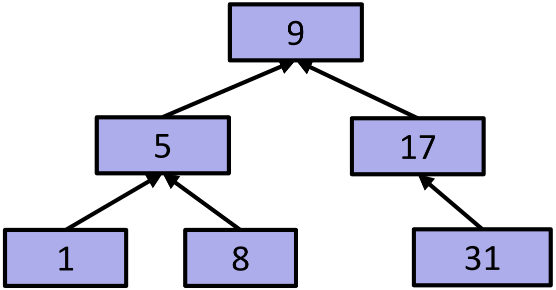
```
if (k < n.k) {
  return contains(
         n.left, k);
}
```

D

```
if (k >= n.k) {
  return contains(
         n.right, k);
}
```

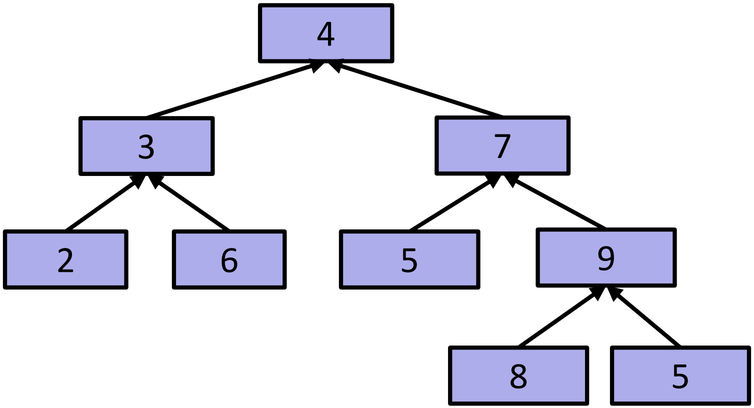
BST Find/Contains: Iterative

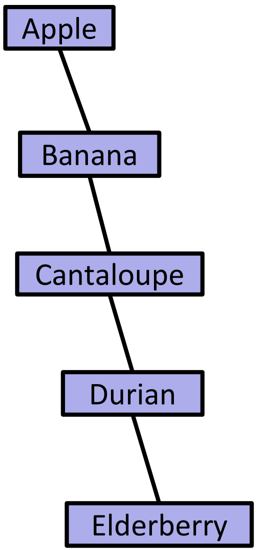
```
boolean contains(BSTNode n,
                 Key k) {
 while (n != null
       && n.key != k) {
  if (k < n.key)
    n = n.left;
  else( k > n.key)
    n = n.right;
 }
 if (n == null)
    return false;
 return true;
}
```



BST Find/Contains's runtime

- What is find's worst-case runtime, as a function of *n*? $\Theta(n)$
- What is find's worst-case runtime, as a function of *height*? $\Theta(h)$

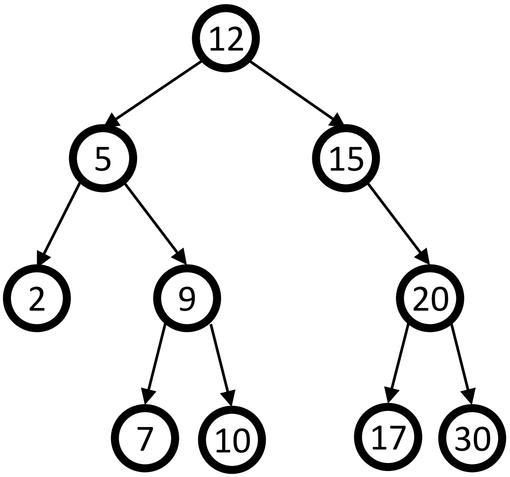




Other "finding operations"

- Find *minimum* node

- Find *maximum* node




```
BSTNode largest(BSTNode n) {
  while (n.right != null) {
    n = n.right;
  }
  return n;
}
```

Lecture Outline

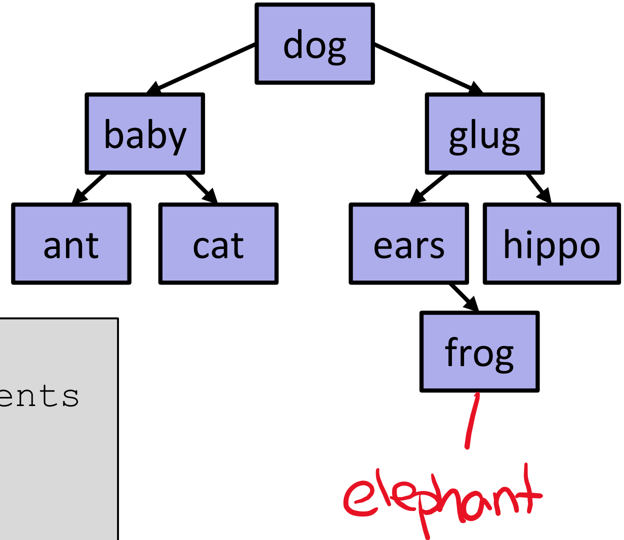
- Redo: Analyzing Recursive Code
- Review: Dictionary and Set ADTs
- Binary Trees != Binary Search Trees
  - Tree traversals
- Binary Search Trees as Dictionary/Set Data Structures
  - Find/Contains
  - **Add/Remove**

Binary Search Trees: Add

- Where does the new item belong?

  at a leaf!

- How do we use BST invariants to ensure the leaf is added correctly?




```
BSTNode add(BSTNode t, Item i) {
  // Implement by putting statements
  // in the correct order
}
```

D, {B,C / C,B}, A

A

```
return t;
```

B

```
if (k < i.key)) {
  t.left
    = add(t.left, i);
}
```

C

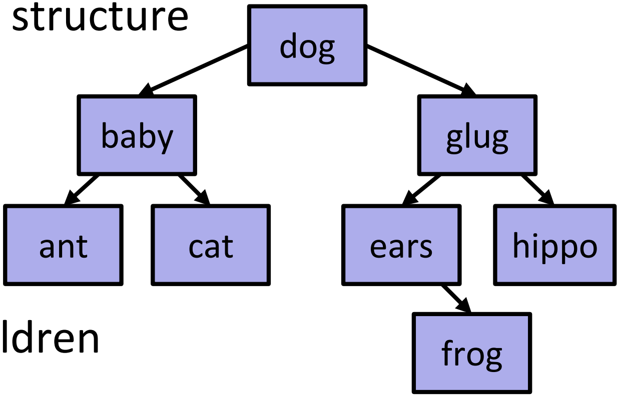
```
if (k > i.key) {
  t.right
    = add(t.right, i);
}
```

D

```
if (t == null){
  return
    new BSTNode(i);
}
```

Binary Search Trees: Remove

- Removing an item disrupts the tree structure
  - `find` the node to be removed
  - Remove it
  - "Fix" the tree so that it is still a BST



- 3 cases based on the number of children
  1. Node has no children
  2. Node has one child
  3. Node has two children

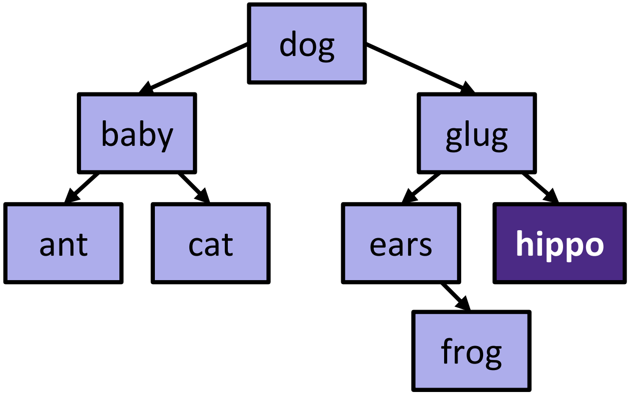
Reminder: a dictionary maps *keys* to *values*; an *item* or *data* refers to the (key, value) pair

- In each case, we must maintain the **BST Ordering**!

BST Remove: Case #1: Leaf

- Remove the node with the key **hippo**
- Runtime? $\Theta(h)$

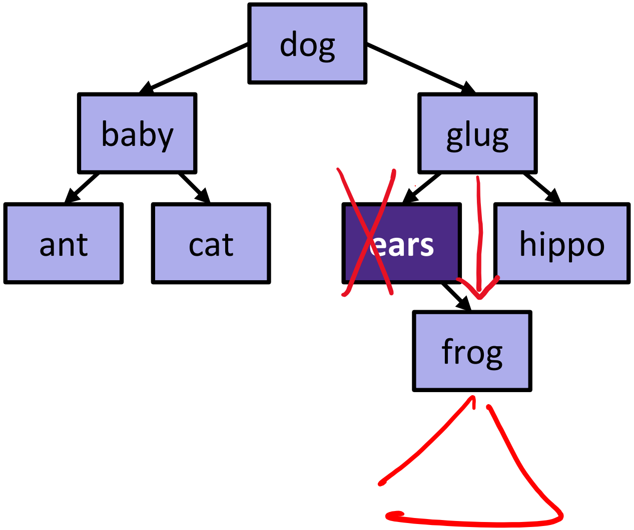
```
BSTNode remove(BSTNode n) {









}
```



BST Remove: Case #2: One Child

- Remove the node with the key **ears**
  - What does the BST invariant say about the descendant's keys?
- Runtime? $\Theta(h)$

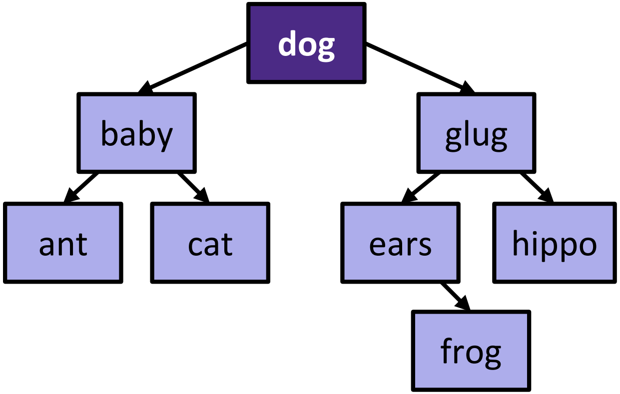
```
BSTNode remove(BSTNode n) {

}
```



BST Remove: Case #3: Two Children

- Remove the node with the key **dog**

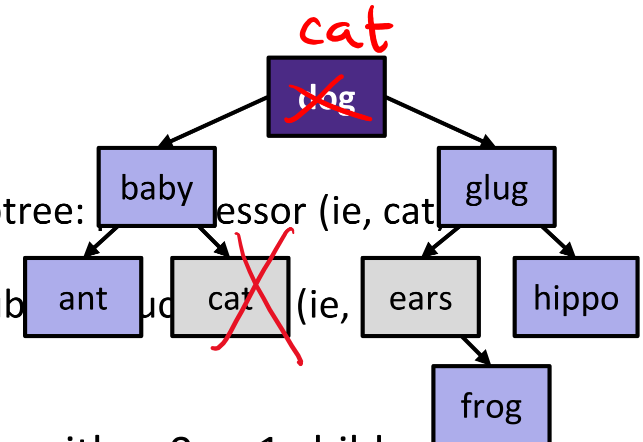
- The replacement node's key:
  - Must be ≻ than all keys in left subtree cat
  - Must be ≺ than all keys in right subtree ears



BST Remove: Case #3: Two Children

- Remove the node with the key dog

- The replacement node's key:
  - Must be $\succ$ than all keys in left subtree: essor (ie, cat
  - Must be $\prec$ than all keys in right sub uc (ie,



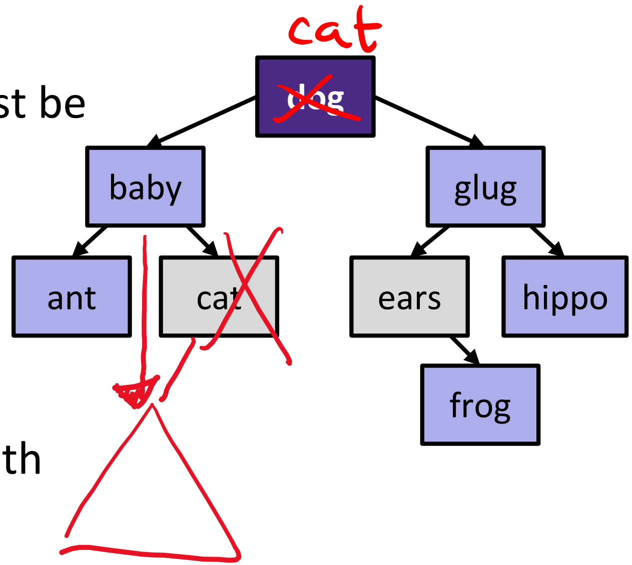
- The predecessor or successor has either 0 or 1 children
  - Why?

BST Remove: Case #3: Two Children

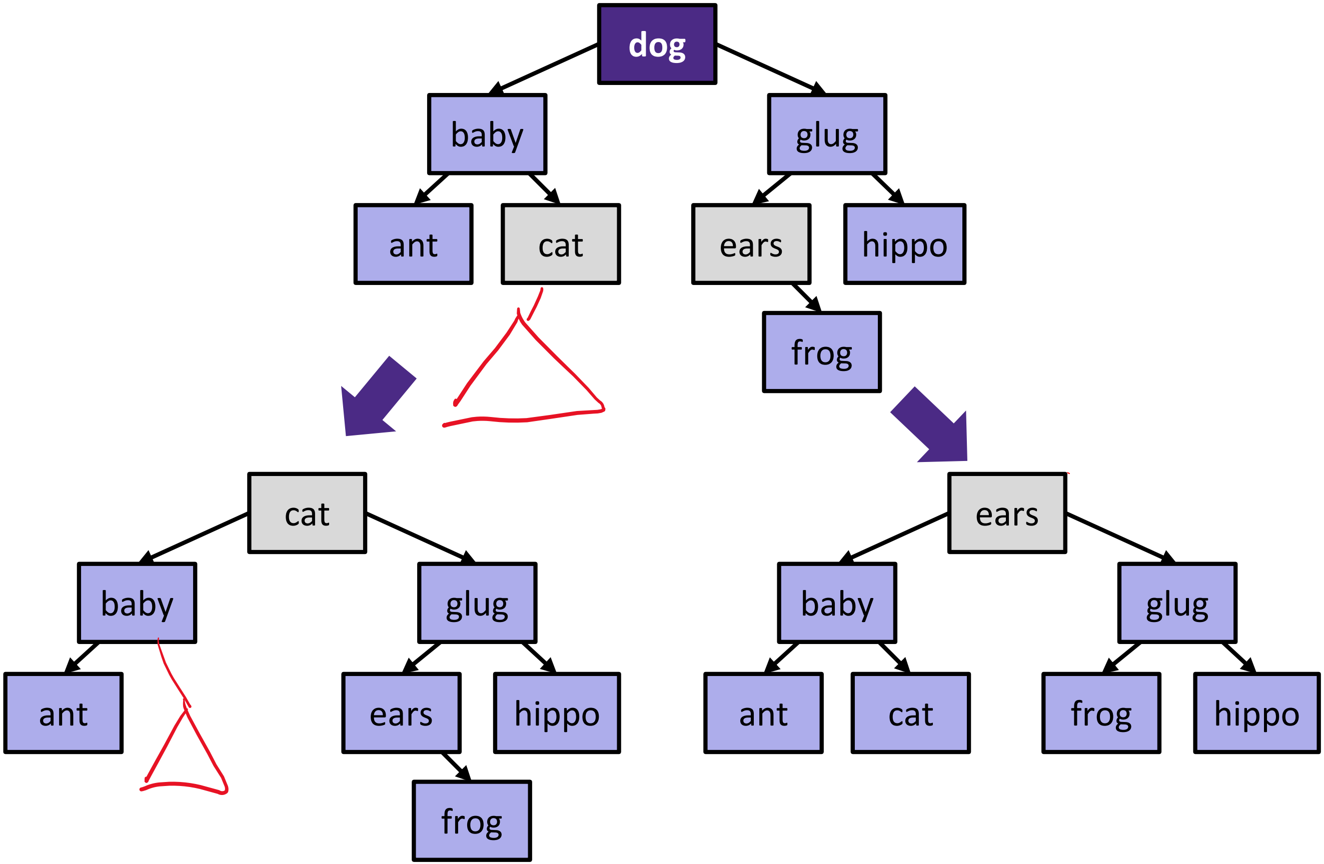
- Remove the node with the key **dog**

- The replacement node's key must be
  - > all keys in the left subtree (ie, predecessor **cat**), or
  - < all keys in the right subtree (ie, successor **ears**)

- The predecessor or successor both have <2 children
  - Why?



BST Remove: Case #3: Two Children

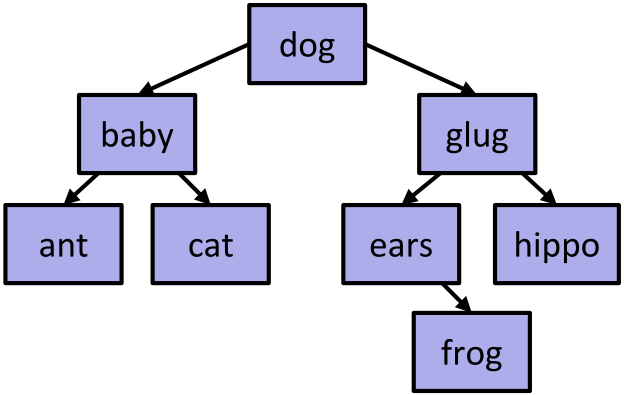


Aside: Finding the largest (or smallest) node

- The **predecessor** is the largest item in the left subtree
- The **successor** is the smallest item in the right subtree

- How do you find the largest (and smallest) item in a tree?
  - Remember that subtrees are trees too

```
BSTNode largest(BSTNode n) {
  while (n.right != null) {
    n = n.right;
  }
  return n;
}
```



BST Summary

- Binary Search Trees implement both Set and Dictionary ADTs
- Binary Search Trees are *recursively defined*
- There is no bound on the BST's height as a function of its size

	LinkedList Dictionary, Worst Case	BST Dictionary, Average Case	BST Dictionary, Worst Case
Find	Θ(N)	Θ(h) aka Θ(log N)	Θ(h) aka Θ(N)
Add	Θ(N)	Θ(h) aka Θ(log N)	Θ(h) aka Θ(N)
Remove	Θ(N)	Θ(h) aka Θ(log N)	Θ(h) aka Θ(N)

BST Summary

- Binary Search Trees implement both Set and Dictionary ADTs
- Binary Search Trees are *recursively defined*
- There is no bound on the BST's height as a function of its size