# Binary Search Trees

CSE 332 Summer 2021

**Instructor:** Kristofer Wong

**Teaching Assistants:**

| | | |
|---|---|---|
| Alena Dickmann | Arya GJ | Finn Johnson |
| Joon Chong | Kimi Locke | Peyton Rapo |
| Rahul Misal | Winston Jodjana | |

# Announcements

- Exercise 2 due!
- Project 1 due Tuesday – how to turn in?
- Surveys from first day
- No lecture Monday!!
- Recording for Friday

# Lecture Outline

- **Review: Dictionary and Set ADTs**
- Working with Binary Trees
- Binary Search Trees as Dictionary/Set Data Structures
  - Find/Contains
  - Add/Remove

# Dictionary ADT: Known 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(n) | O(n) |
| Sorted array | O(n) | O(log n) | O(n) |

**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
  - Hash Tables
    - Not tree-like at all

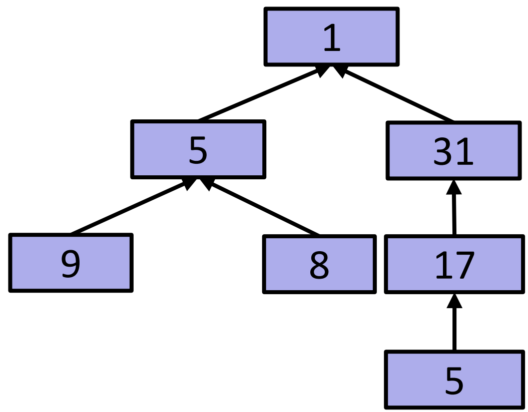
- Skipping: Other balanced binary search trees
  - Eg, red-black tree (and LLRBs), splay tree

# Lecture Outline

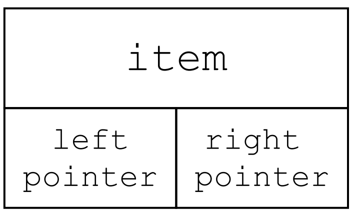
- Review: Dictionary and Set ADTs
- **Working With Binary Trees**
- Binary Search Trees as Dictionary/Set Data Structures
  - Find/Contains
  - Add/Remove

# Binary Trees: Quick Review

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




- Representation:




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

# Binary Trees: Quick Review

- 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: $2^{h+1} - 1$
  - min # of leaves: **1**
  - min # of nodes: **h+1**

# Recursion Example: Calculating Tree Height

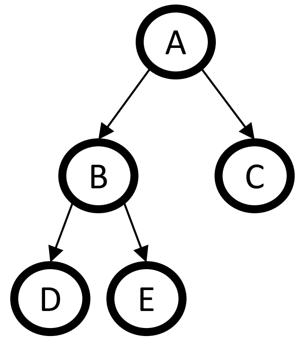
- What is the height of a tree with root **`r`**?
- What is the runtime for your algorithm?

```
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

# Tree Traversals

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

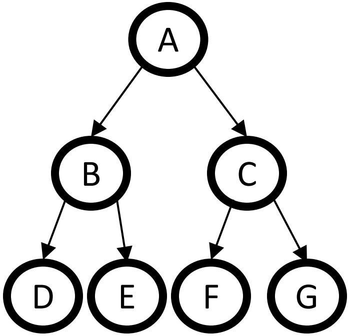


(an expression tree)

- 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)

# Traversals: Recursive Implementation

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




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

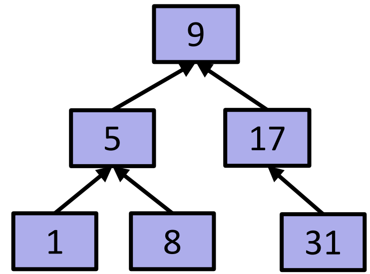
- Again, non-recursive implementation is painful

# Lecture Outline

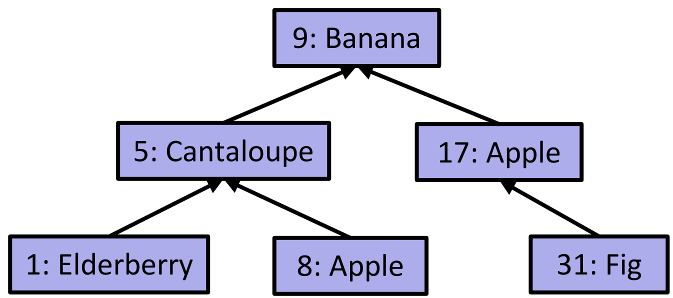
- Review: Dictionary and Set ADTs
- Working With Binary Trees
- **Binary Search Trees**
  - BST != Binary Trees
  - Find/Contains
  - Add/Remove

# 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



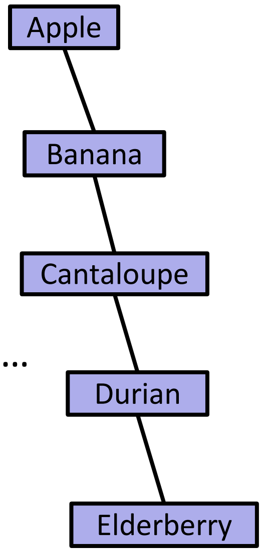

- Reminder: BSTs can also contain (key, value) pairs

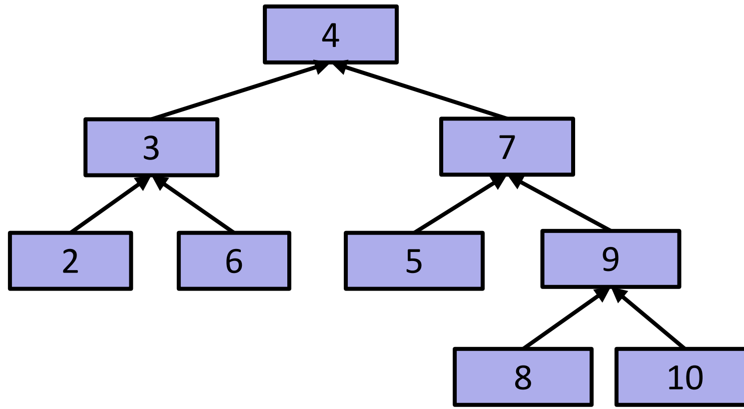



*The BST ordering applies recursively to the entire subtree* gradescope gradescope.com/courses/275833

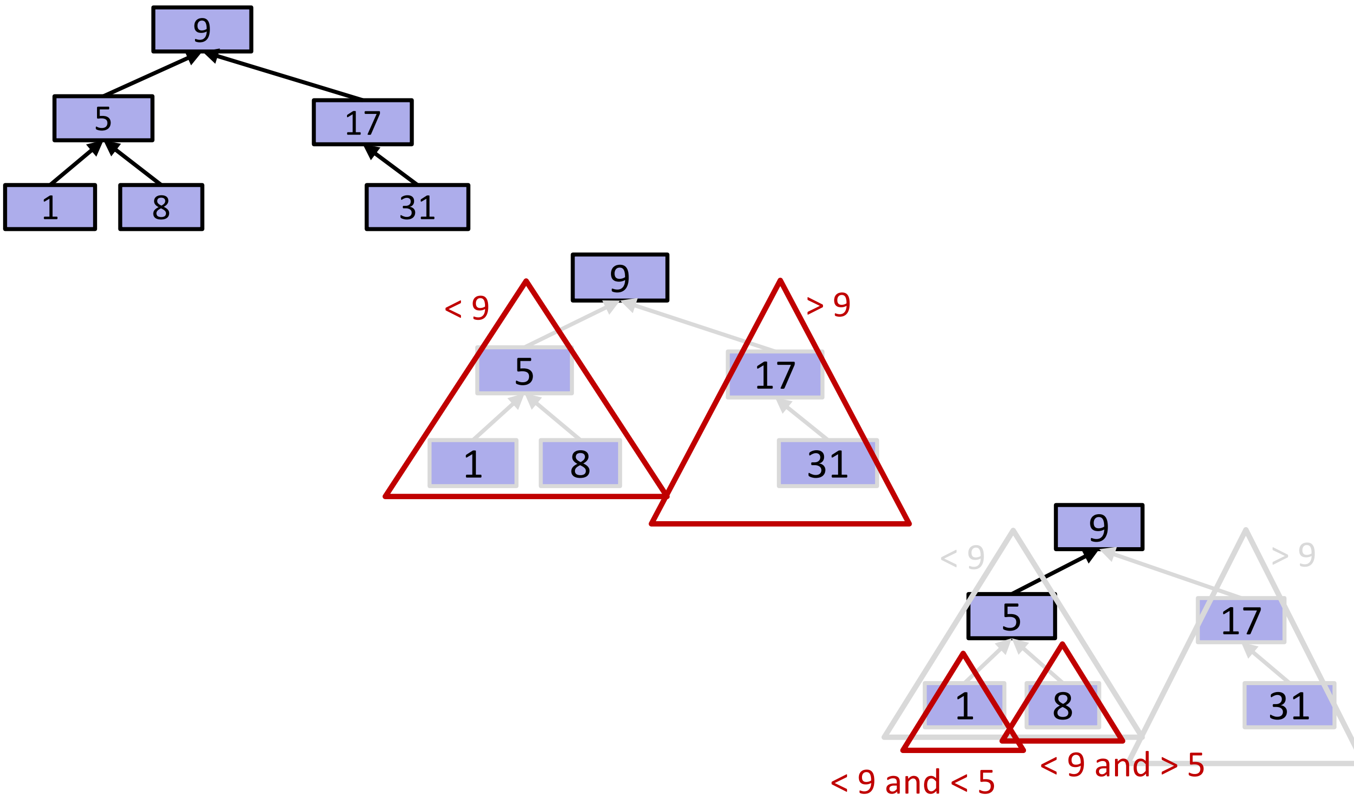
- Are these Binary Search Trees?

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



# BST Ordering Applies *Recursively*



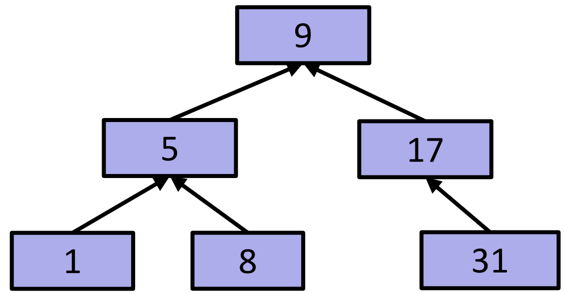
# Lecture Outline

- 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

```
boolean contains(BSTNode n,
                 Key k) {
A, B, {C, D}
}
```




A

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

B

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

C

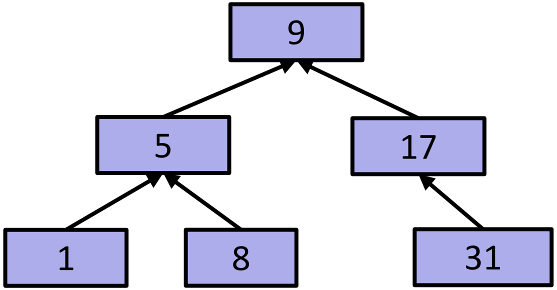
```
if (k < n.k) {
  return contains(
       n.left, k);
}
```

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;
}
```



# gradescope

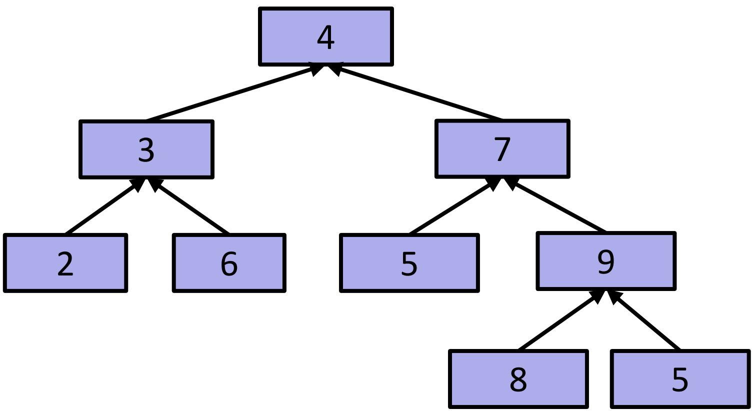
gradescope.com/courses/275833

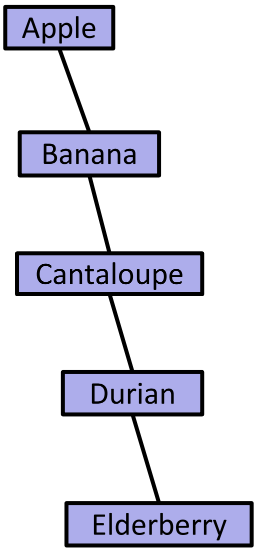
What do you expect the worst-case Big O runtime to be for Find and Contains?

O(N)

# BST Find/Contains's runtime

- What is find's worst-case runtime, as a function of *n*?
- What is find's worst-case runtime, as a function of *height*?

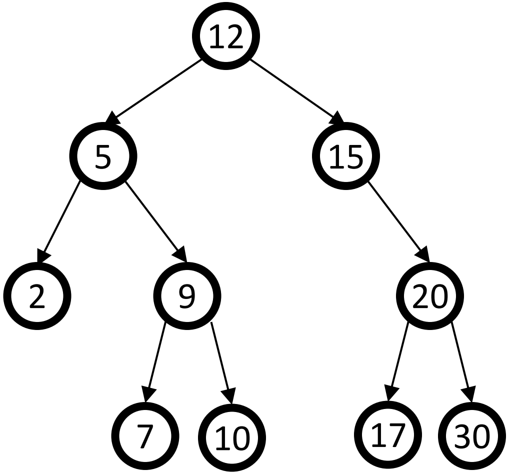


# Other "finding operations"

- Find *minimum* node

- Find *maximum* node



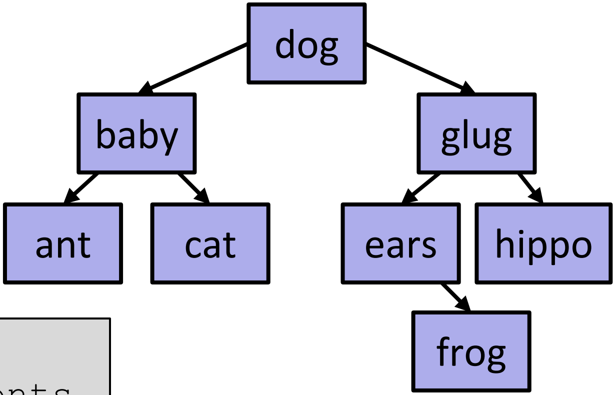

```
BSTNode largest(BSTNode n) {
  while (n.right != null) {
    n = n.right;
  }
  return n;
}
```

# Lecture Outline

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

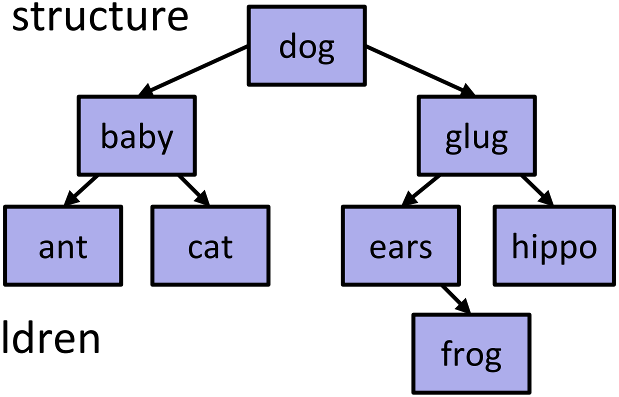



```
BSTNode add(BSTNode t, Item i) {
  // Implement by putting statements
  // in the correct order

  D, {B, C}, A
}
```

| A | B | C | D |
|---|---|---|---|
| `return t;` | `if (k < i.key)) { t.left = add(t.left, i); }` | `if (k > i.key) { t.right = add(t.right, i); }` | `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

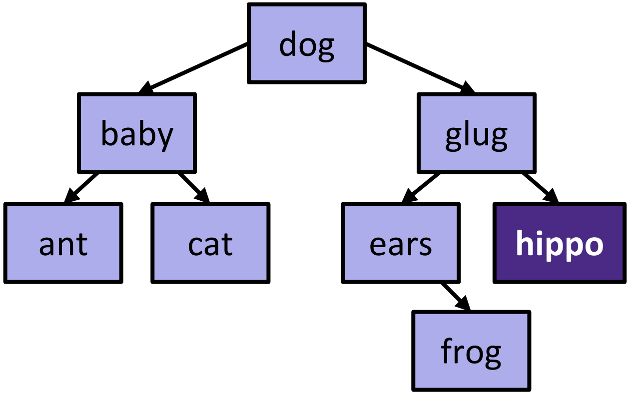
- In each case, we must maintain the **BST Ordering**!

# BST Remove: Case #1: Leaf

- Remove the node with the key **hippo**

- Runtime?

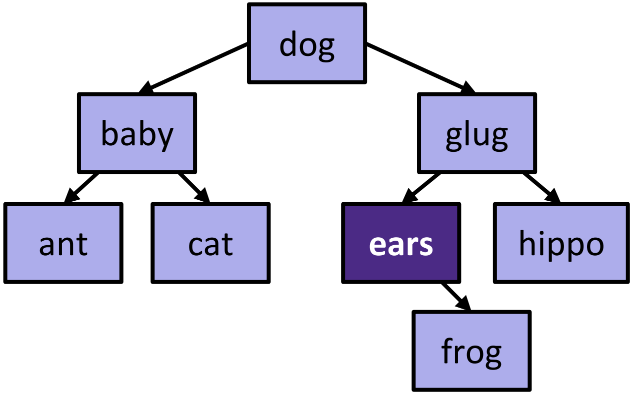
```
BSTNode remove(BSTNode n) {


find(n)
Set parent's pointer to n to
null (remove from tree)






}
```



# 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?

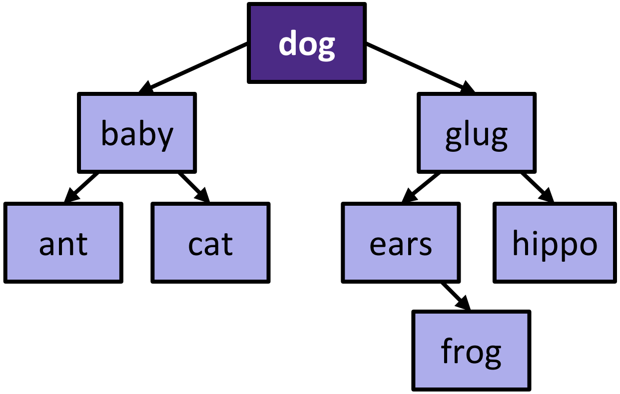
```
BSTNode remove(BSTNode n) {

find(n)
set n's parent's pointer to
frog instead of n.

}
```



# BST Remove: Case #3: Two Children

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

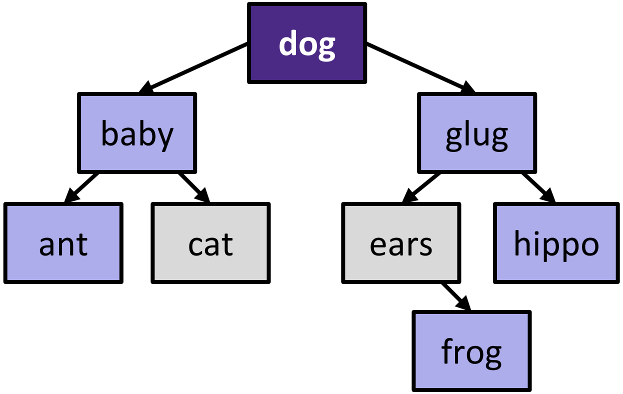
- The replacement node's key:
  - Must be ≻ than all keys in left subtree
  - Must be ≺ than all keys in right subtree



# BST Remove: Case #3: Two Children

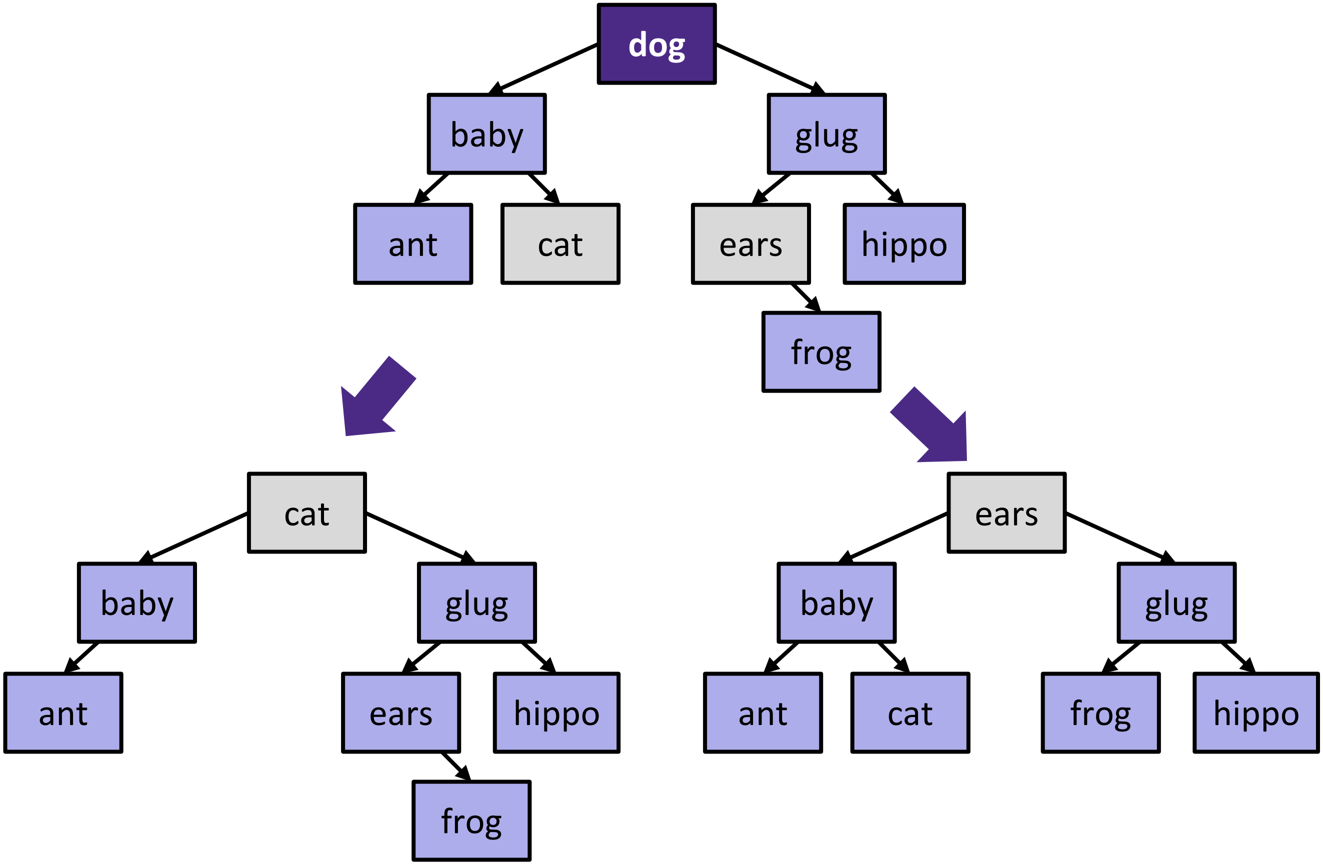
- Remove the node with the key **dog**

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




- The predecessor or successor has either 0 or 1 children

# BST Remove: Case #3: Two Children

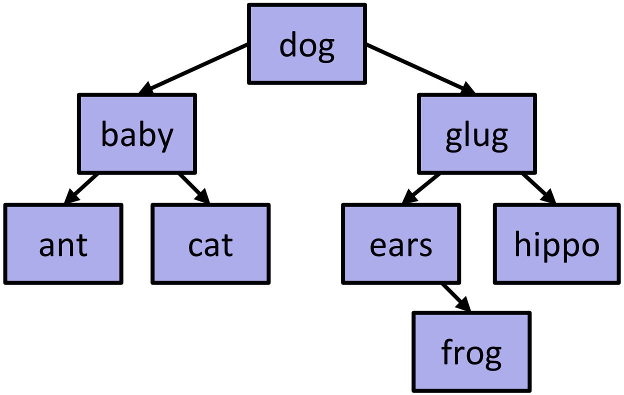


# Aside: Finding the largest (or smallest) node

- 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) |