Binary Search Trees

CSE 332 Spring 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Annie Mao	Howard Xiao	Lucy Jiang
Chris Choi	Jade Watkins	Nathan Lipiarski
Ethan Knutson	Khushi Chaudhari	Richard Jiang

Announcements

- Quiz 1 has been graded and should hit your inbox after lecture
  - Median group size was 3 students; median group grade was 82 pts
  - 31 students went solo; median solo grade was 74.5 pts
  - Regrade requests close before lecture on Friday
- P2 partner survey due Thursday afternoon
  - Convert your availability to PDT to save yourself some 😟😟😟
- Ex 5 is *super hard*. Start now!
- We are always available for 1:1 meetings! Let us know how we can help!

Learning Objectives

- Be able to use both the expansion method and the tree method, to find the closed-form of a recurrence relation
- Understand the difference between a binary tree and a binary search tree and, in particular, be able to apply the *BST ordering property*
- Implement recursive versions of binary tree traversals (ie, pre-/in-/post-order)
- Implement recursive versions of the BST find/contains, insert, and remove algorithms

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

Summing an Array, Again (1 of 5)

Two "obviously" linear algorithms:

```
int sum(int[] arr){
  int ans = 0;
  for(int i=0; i<arr.length; ++i)
     ans += arr[i];
  return ans;
}
```

Recursive:

```
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);
}
```

Summing an Array, Again (2 of 5)

- What about a binary version of **sum**?
  - Can we get a BinarySearch-like runtime?

```
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);
}
```

T(0) → `if(lo==hi)`

T(1) → `if(lo==hi-1)`

T(n) → `int mid = (hi+lo)/2;` `return help(arr,lo,mid) + help(arr,mid,hi);`

Summing an Array, Again (3 of 5)

Expansion Method

(1) Base case + Recurrence Relation

$$T(n) = \begin{cases} c_1 & \text{if } n=0 \\ c_2 & \text{if } n=1 \\ c_3 + 2T(\frac{n}{2}) \end{cases}$$

(2) Expansion + general form

$$T(n) = c_3 + 2T(\tfrac{n}{2})$$
$$= c_3 + (c_3 + c_3 + 4T(\tfrac{n}{4}))$$
$$= c_3 + c_3 + c_3 + (c_3 + c_3 + c_3 + c_3 + 8T(\tfrac{n}{8}))$$
$$\vdots$$
$$= \sum_{i=0}^{k-1} 2^i c_3 + 2^k T(\tfrac{n}{2^k})$$

1 expn
2 expns
3 expns
⋮
k expns

(3) Closed form (see slide 19)

Let $k = \log n$

$$T(n) = (n-1)c_3 + 2^{\log n} c_2$$
$$= (n-1)c_3 + c_2 n \in O(n)$$

Technique #2: Tree Method

- Idea: We'll do the same reasoning, but give ourselves a visual to make the organization easier

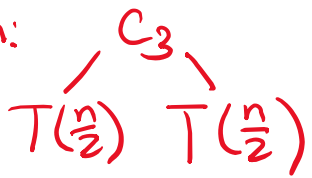
- We'll make a **tree**
  - Each node of the tree represents one recursive call
  - The children of that node are the new recursive calls made

Summing an Array, Again (4 of 5)

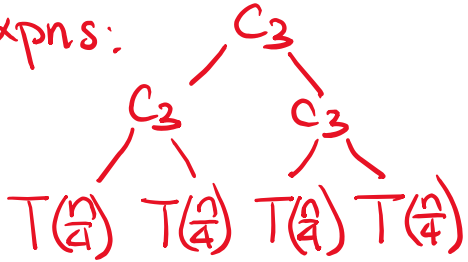
Tree Method

$$T(n) = \begin{cases} c_1 & \text{if } n=0 \\ c_2 & \text{if } n=1 \\ c_3 + 2T\left(\frac{n}{2}\right) & \end{cases}$$

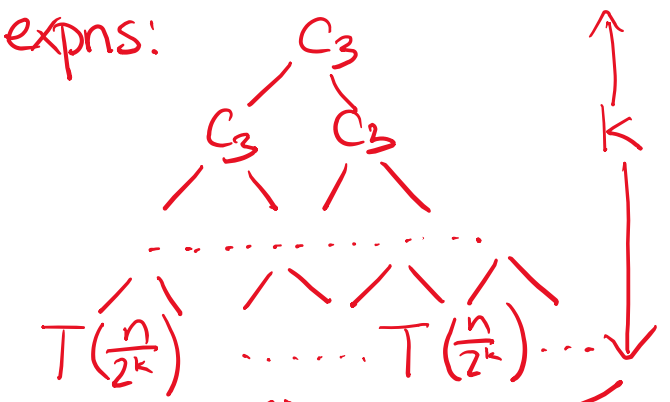
1 expn:



2 expns:



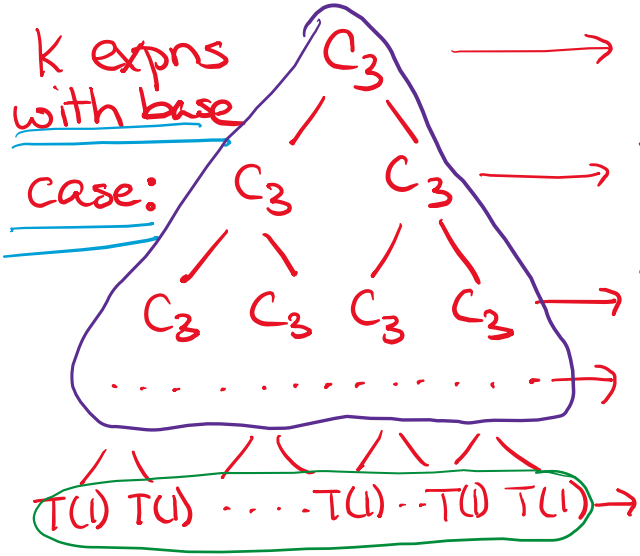
k expns:



when does the recursion stop; i.e., when do these become leaves?

$T(0)$ or $T(1)$!

Tree Method (cont.)



level 0: $c_3 = 2^0 c_3$

level 1: $2c_3 = 2^1 c_3$

level 2: $4c_3 = 2^2 c_3$

level i: $2^i c_3$

level $\log n$: $2^{\log n} \cdot T(1) = n c_2$

★ By inspection, we can see there are n leaves & $k = \log n$ levels ★

So $T(n) = \underbrace{c_2 n}_{\text{leaves}} + \underbrace{??}_{\text{internal nodes}}$

$$= c_2 n + \sum_{i=0}^{\log n - 1} 2^i c_3$$

$$= c_2 n + (n-1) c_3 \in O(n)$$

Summing an Array, Again (5 of 5)

- Runtime is: O(n)
- Observation: it adds each number once while doing little else
  - Can't do better than O(n); have to read whole array!

```
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);
}
```

Parallelism Teaser

- But suppose we could do two recursive calls *at the same time*

- If you have as much parallelism as needed, the recurrence becomes
  - T(n) = O(1) + **1** T(n/2)

```
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);
}
```

Really Common Recurrences

Recurrence Relation	*Closed Form*	*Name*	*Example*
T(n) = O(1) + T(n/2)	O(log n)	Logarithmic	Binary Search
T(n) = O(1) + T(n-1)	O(n)	Linear	Sum (v1: "Recursive Sum")
T(n) = O(1) + 2T(n/2)	O(n)	Linear	Sum (v2: "Recursive Binary Sum")
T(n) = O(n) + T(n/2)	O(n)	Linear	
T(n) = O(n) + 2T(n/2)	O(n log n)	Loglinear	MergeSort
T(n) = O(n) + T(n-1)	$O(n^2)$	Quadratic	
T(n) = O(1) + 2T(n-1)	$O(2^n)$	Exponential	Fibonacci

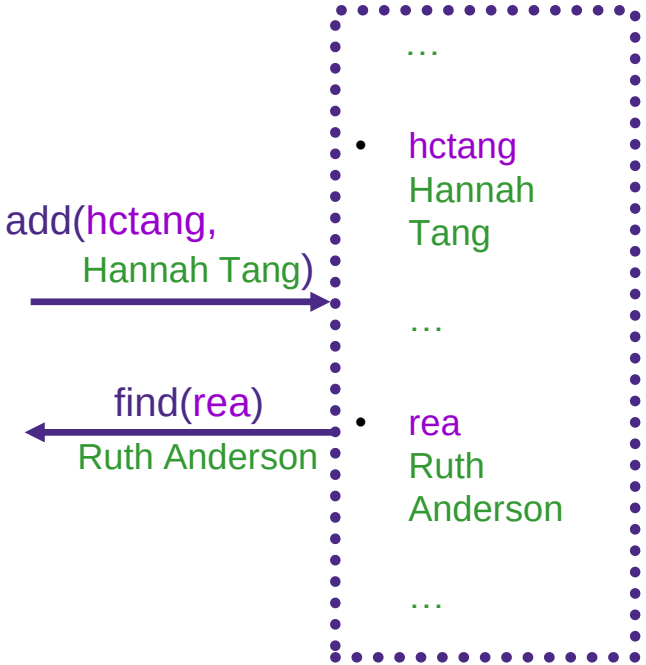
Lecture Outline

- 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

Dictionary ADT

- Also known as "**Map ADT**"

- Operations:
  - **`add(k, v)`:**
    - places (k,v) in dictionary
    - if key already present, typically overwrites existing item
  - **`find(k)`:**
    - Returns v associated with k
  - **`contains(k)`:**
    - Returns true if k is in the dictionary
  - **`remove(k)`:**
    - …



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

Dictionary ADT: Data Structures

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

	`insert`	`find`	`delete`
Unsorted linked list			
Unsorted array	O(1)	O(n)	O(n) find + O(n) shift
Sorted linked list			
Sorted array	O(n) find + O(n) shift	O(log n)	O(n) find + O(n) shift

- * *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

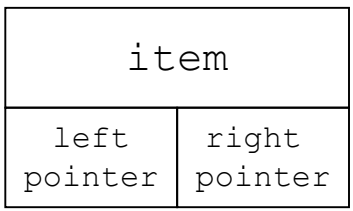
Lecture Outline

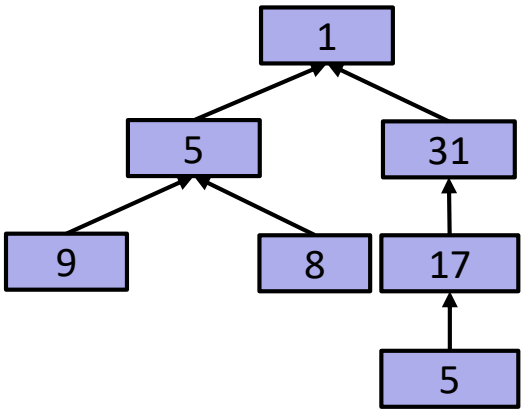
- Redo: Analyzing Recursive Code
- Review: Dictionary and Set ADTs
- **Binary Trees != Binary Search Trees**
  - Tree traversals
- Binary Search Trees as Dictionary/Set Data Structures
  - Find/Contains

Binary Tree

- 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

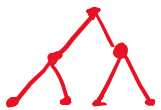
Binary Tree: Some Numbers

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

h=2

- 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

Calculating Tree Height

- What is the height of a tree with root `r`?
- What is the runtime for your algorithm? O(n)

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

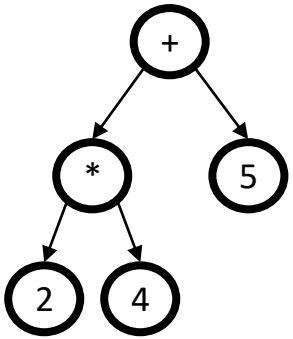
- *Note*: non-recursive is painful – need your own stack of pending nodes
  - Much easier to use recursion's call stack

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

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



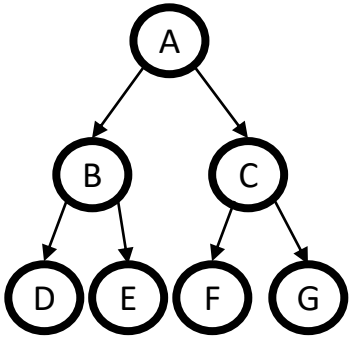
(an expression tree)

- 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)

```
PrintIndented:
+
  *
    2
    4
  5
```

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

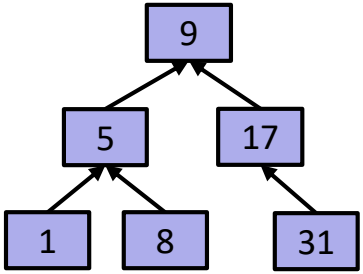
Lecture Outline

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

Binary *Search* Trees

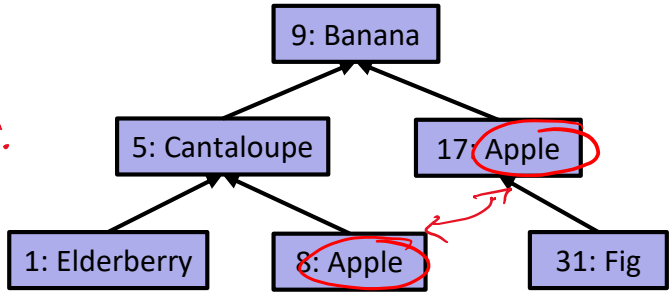
- A **Binary Search Tree** is a binary tree with the following invariant: for every node with key k in the BST:
  - The left subtree only contains keys <k
  - The right subtree only contains keys >k

Set:



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

Dictionary:



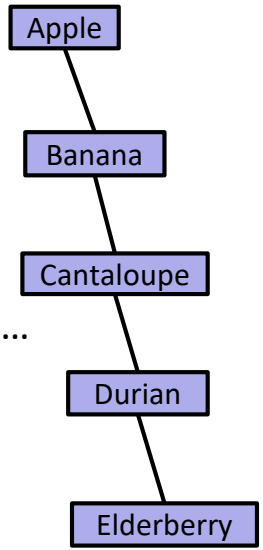
The BST ordering applies recursively to the entire subtree

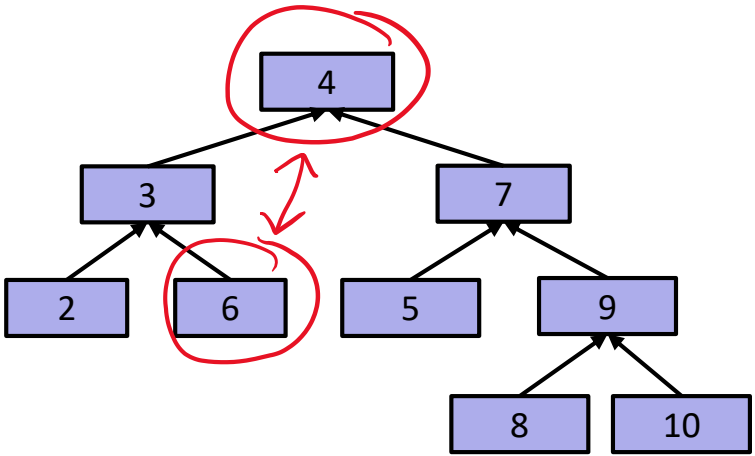
Poll Everywhere

pollev.com/cse332

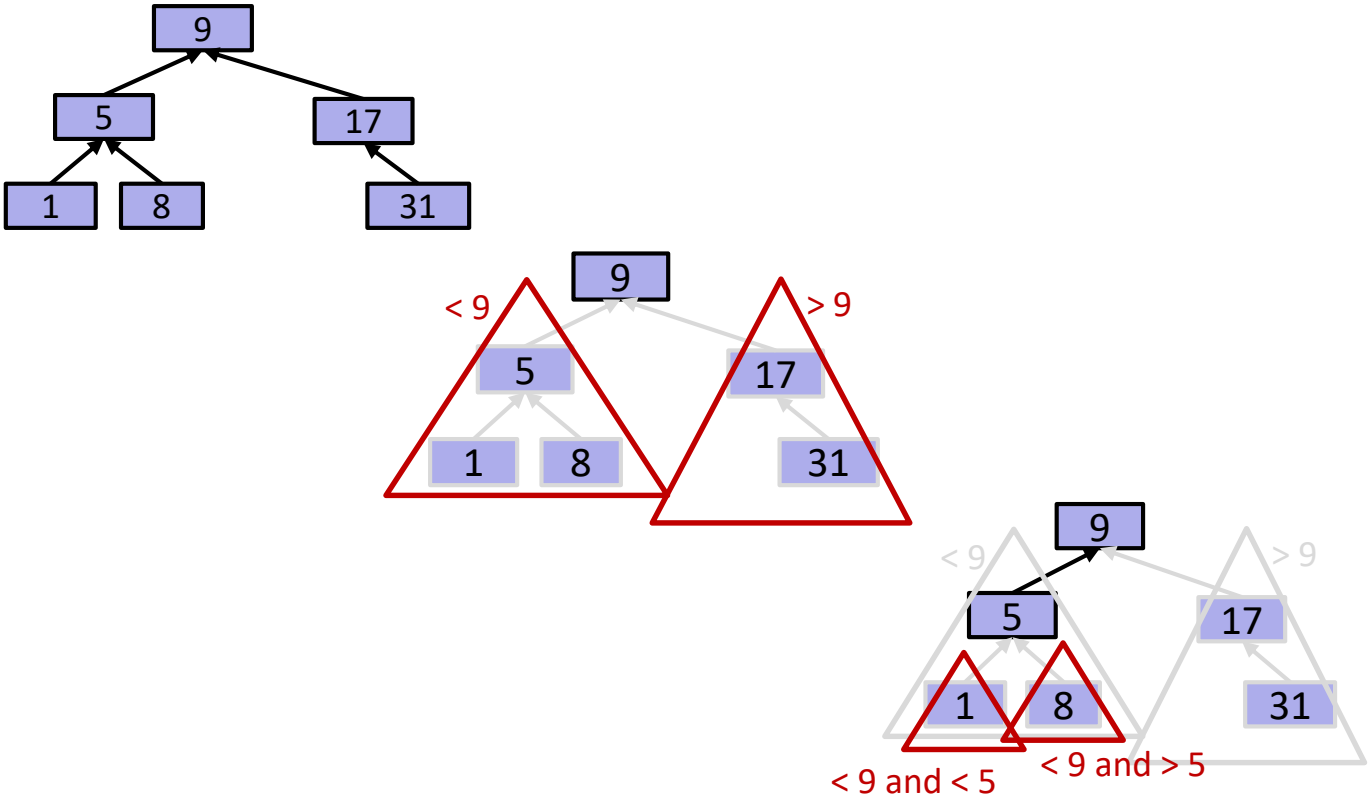
- Are these Binary Search Trees?

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**

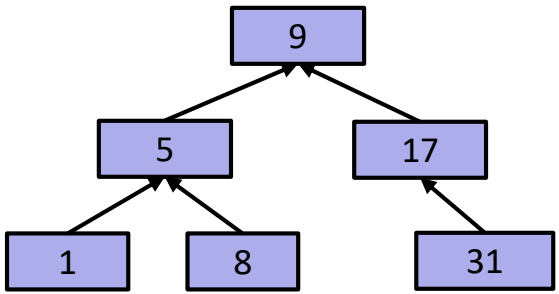
Binary Search Trees: Find/Contains

- 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?

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

}
```

ABCD
ABDC
both will work



If you want your algorithm to handle dups ≥ 2

A

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

B

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

C

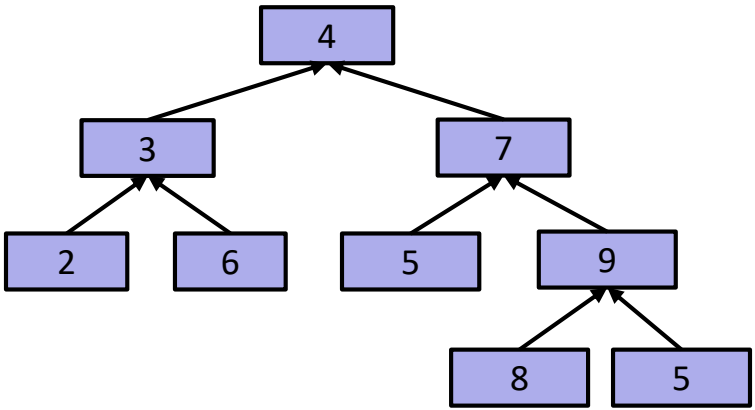
```
if (k < n.k) {
  return contains(
      n.left, k);
}
```

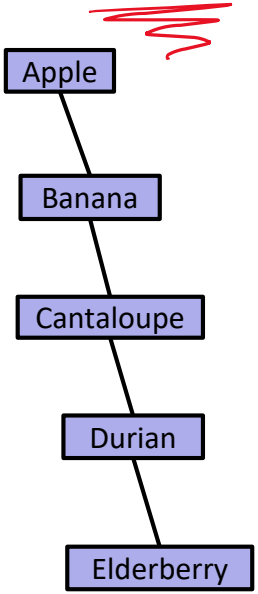
D

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

BST Find/Contains's runtime

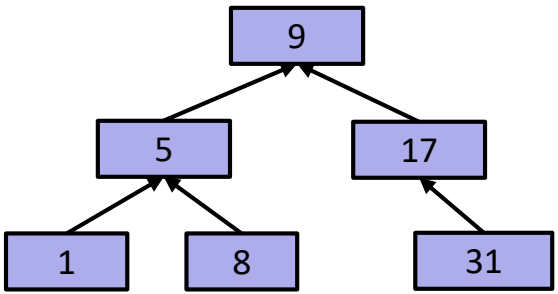
- What is find's worst-case runtime, as a function of *n*?
- What is find's worst-case runtime, as a function of *height*?





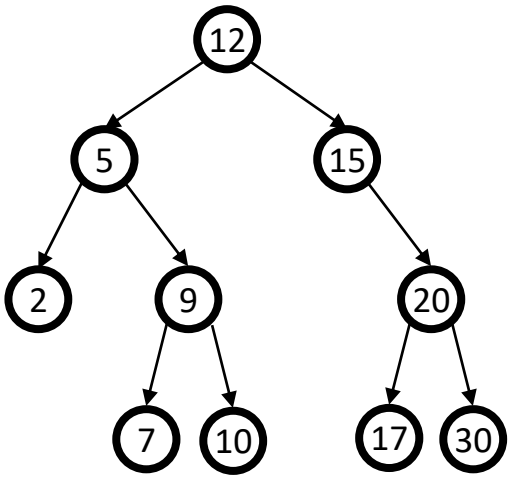
BST Find/Contains: Iterative

```
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;
}
```



Other "finding operations"

- Find *minimum* node
- Find *maximum* node



Summary

- The tree method is a visualization of the expansion method of finding a closed-form recursively-defined runtime expression
- Binary Search Trees have a (recursively-defined) ordering property