Set and Map ADTs: Binary Search Trees

CSE 373 Winter 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Aaron Johnston
Amanda Park
Anish Velagapudi
Brian Chan
Elena Spasova

Ethan Knutson
Farrell Fileas
Howard Xiao
Jade Watkins
Lea Quan

Nathan Lipiarski
Sam Long
Yifan Bai
Yuma Tou

Poll Everywhere

pollev.com/uwcse373

About how long did Homework 2 take?

A. 0-2 Hours
B. 2-4 Hours
C. 4-6 Hours
D. 5-10 Hours
E. 10-14 Hours
F. 14+ Hours
G. I haven’t finished yet / I don’t want to say

Announcements

- Homework 3: Autocomplete is released
  - We’ve started to implement a rate-limiting / token-saving policy to encourage you to write your own tests and to start early.
  - Thresholds are “reasonable”
  - Hint: If you implemented a unittest that tested the exact thing the autograder described, you could run the autograder’s test in the debugger (and also not have to use your tokens).
  - Hint: MatchResult takes an *inclusive* start but an *exclusive* end index

- HW2 feedback survey
  - Similar to HW1; help us improve our homeworks

- Extra DITs added Monday morning
  - 11-12:30, CSE 4th floor breakout

Questions from Reading Quiz

- Do map values need to be unique as well?
- Is the Java TreeMap a BST?
- What if the map keys aren’t numbers?

Lecture Outline

- **Binary Search and Binary Range Search**
- ADTs: Sets and Maps
- Binary Search Trees as Sets and Maps
- BST Operations:
    - Find/Contains
    - Add
    - Remove

Case Analysis != Asymptotic Analysis

- Case analysis deals with a specific input or a specific class of inputs

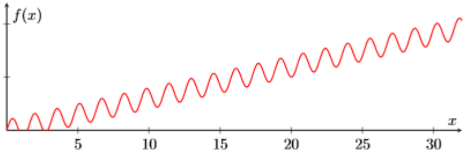
- Asymptotic analysis deals with “the shape of the curve near infinity”

- Demos (each case has their own O, Θ, and Ω bounds):
  - Best: https://www.desmos.com/calculator/uovi22xfwq
  - Worst: https://www.desmos.com/calculator/v3u5hviyqe
  - Overall: https://www.desmos.com/calculator/huqfxcwu05

"Shapes Near Infinity"

has Θ bound for overall case

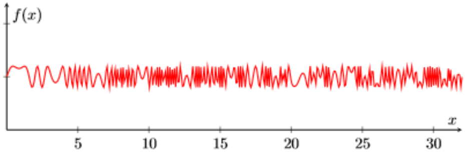
(b)



Solution:

$\mathcal{O}(n)$, $\Omega(n)$, and $\Theta(n)$.

(c)

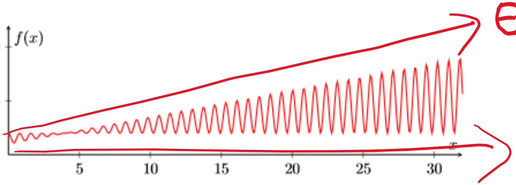


Solution:

$\mathcal{O}(1)$, $\Omega(1)$, and $\Theta(1)$.

no Θ-bound for overall case

(e)



Θ-bound for worst case

Θ-bound for best case

Solution:

$\mathcal{O}(n)$ and $\Omega(1)$. This function does not have a big-Θ, because the tightest upper and lower bounds are not the same.

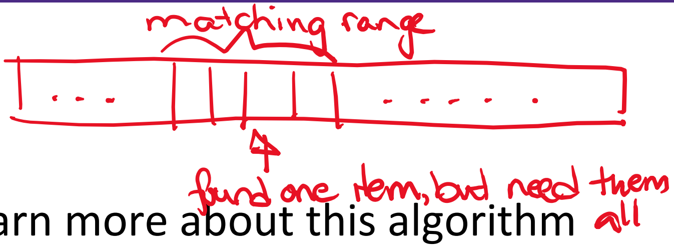
Binary Search Runtime

```java
public static boolean binarySearch(int[] sorted, int findMe) {
  if (sorted.length == 0)
    return false;

  int mid = sorted.length / 2;
  if (findMe < sorted[mid])
    int[] subrange = Arrays.copyOfRange(sorted, 0, mid);
    return binarySearch(subrange, findMe);
  else if (x > sorted[mid])
    int[] subrange = Arrays.copyOfRange(sorted, mid, sorted.length);
    return binarySearch(subrange, findMe);
  else
    return true;
}
```

early exit causes $\Theta(1)$ best case runtime

Case	Big-O	Big-Theta	Big-Omega
Best	$O(1)$	$\Theta(1)$	$\Omega(1)$
Worst	$O(\log_2 N)$	$\Theta(\log_2 N)$	$\Omega(\log_2 N)$
Overall	$O(\log_2 N)$	DNE	$\Omega(1)$

Binary *Range* Search



- You are highly encouraged to learn more about this algorithm with a websearch or by talking with a friend
  - Remember: Don’t copy-n-paste other people’s (or your) code

- Basic idea is that you’re looking for the *first* and *last* elements of a range
  - Which means, unlike binary search, there’s no early exit when you’ve found a matching item

Case	Big-O	Big-Theta	Big-Omega
Best	$O(\log_2 N)$	$\Theta(\log_2 N)$	$\Omega(\log_2 N)$
Worst	$O(\log_2 N)$	$\Theta(\log_2 N)$	$\Omega(\log_2 N)$
Overall			

Lecture Outline

- Binary Search vs Binary Range Search
- **ADTs: Sets and Maps**
- Binary Search Trees as Sets and Maps
- BST Operations:
  - Find/Contains
  - Add
  - Remove

ADTs So Far

List ADT. A collection storing an ordered sequence of elements.

- Each element is accessible by a zero-based index.
- A list has a size defined as the number of elements in the list.
- Elements can be added to the front, back, *or any index in the list*.
- Optionally, elements can be removed from the front, back, *or any index in the list*.

- Data structures that implemented the List ADT include LinkedList and ArrayList
- When we restrict List's functionality, we end up with the 3 other ADTs we've seen so far

ADTs So Far

Deque ADT. A collection storing an ordered sequence of elements.

- Each element is accessible by a zero-based index.
- A deque has a size defined as the number of elements in the deque.
- Elements can be added to the front or back.
- Optionally, elements can be removed from the front or back.

Stack ADT. A collection storing an ordered sequence of elements.

- A stack has a size defined as the number of elements in the stack.
- Elements can only be added and removed from the top (“LIFO”)

Queue ADT. A collection storing an ordered sequence of elements.

- A queue has a size defined as the number of elements in the queue.
- Elements can only be added to one end and removed from the other (“FIFO”)

- Data structures that implemented these ADTs are LinkedList and ArrayList variants

Set ADT

Set ADT. A collection of values.

- A set has a size defined as the number of elements in the set.
- You can add and remove values.
- Each value is accessible via a “get” or “contains” operation.

- Naïve implementation: a list of items
  - add(v):
  - contains(v):
  - remove(v):

```
class Item<Value> {
  Value v;
}

LinkedList<Item> set;
```

Map ADT

Map ADT. A collection of keys, each associated with a value.

- A map has a size defined as the number of elements in the map.
- You can add and remove (key, value) pairs.
- Each value is accessible by its key via a "get" or "contains" operation.

- Also known as "**Dictionary ADT**"
- Naïve implementation: a set of (key, value) pairs
  - add(k, v): $\Theta(1)$
  - find(k):
  - contains(k): $\Theta(N)$
  - remove(k):

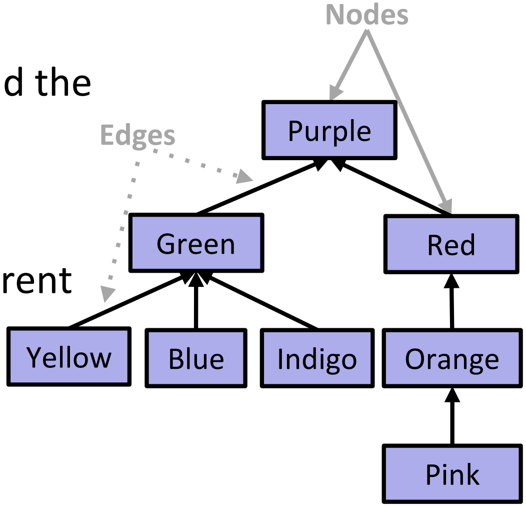
```
class KVPair<Key, Value> {
  Key k;
  Value v;
}

LinkedList<KVPair> map;
```

Lecture Outline

- Binary Search vs Binary Range Search
- ADTs: Sets and Maps
- **Binary Search Trees as Sets and Maps**
- BST Operations:
  - Find/Contains
  - Add
  - Remove

Tree Data Structure

- A **Tree** is a collection of nodes; each node has <= 1 parent and >= 0 children
  - **Root node**: the "top" of the tree and the only node with no parent
  - **Leaf node**: a node with no children
  - **Edge**: the connection between a parent and child
  - There is exactly one path between any pair of nodes

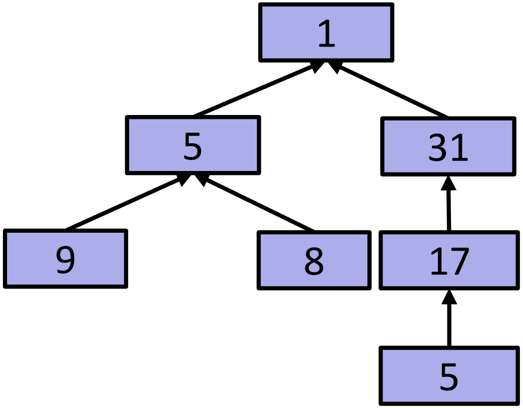


- **Subtree**: a node and all of its descendants
  - Trees are defined recursively!

```
class Node<Value> {
  Value v;
  List<Node> children;
}
```

Binary Tree Data Structure

- A **Binary Tree** is a tree where each node has 0 <= children <= 2



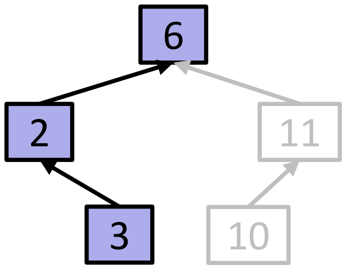
```
class BinaryNode<Value> {
  Value v;
  BinaryNode left;
  BinaryNode right;
}
```

Review: Binary Search

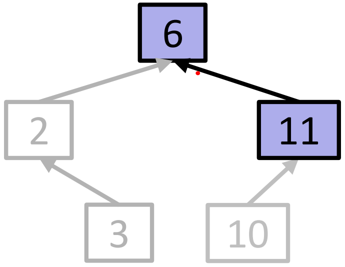
- Remember Binary Search's "function call tree"?

2	3	6	10	11

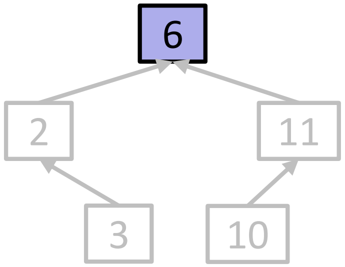
binarySearch(3)



binarySearch(11)

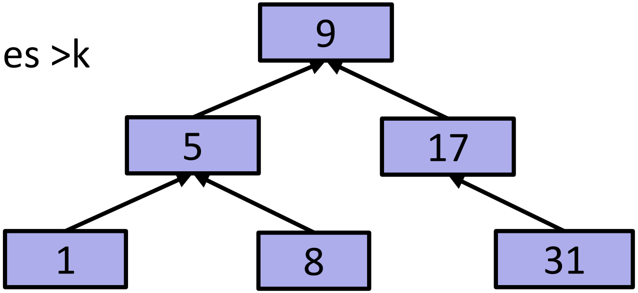


binarySearch(6)



Binary *Search* Trees

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



```
class BSTNode<Value> {
  Value v;
  BSTNode left;
  BSTNode right;
}
```

Reminder: the BST ordering applies recursively to the entire subtree

Poll Everywhere

pollev.com/uwcse373

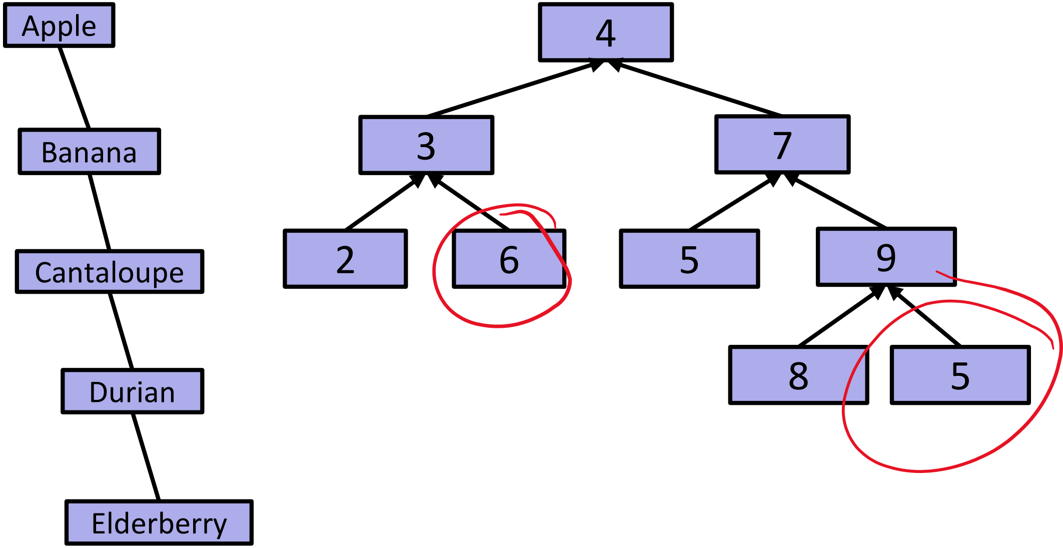
- Are these Binary Search Trees?

A. Yes / Yes

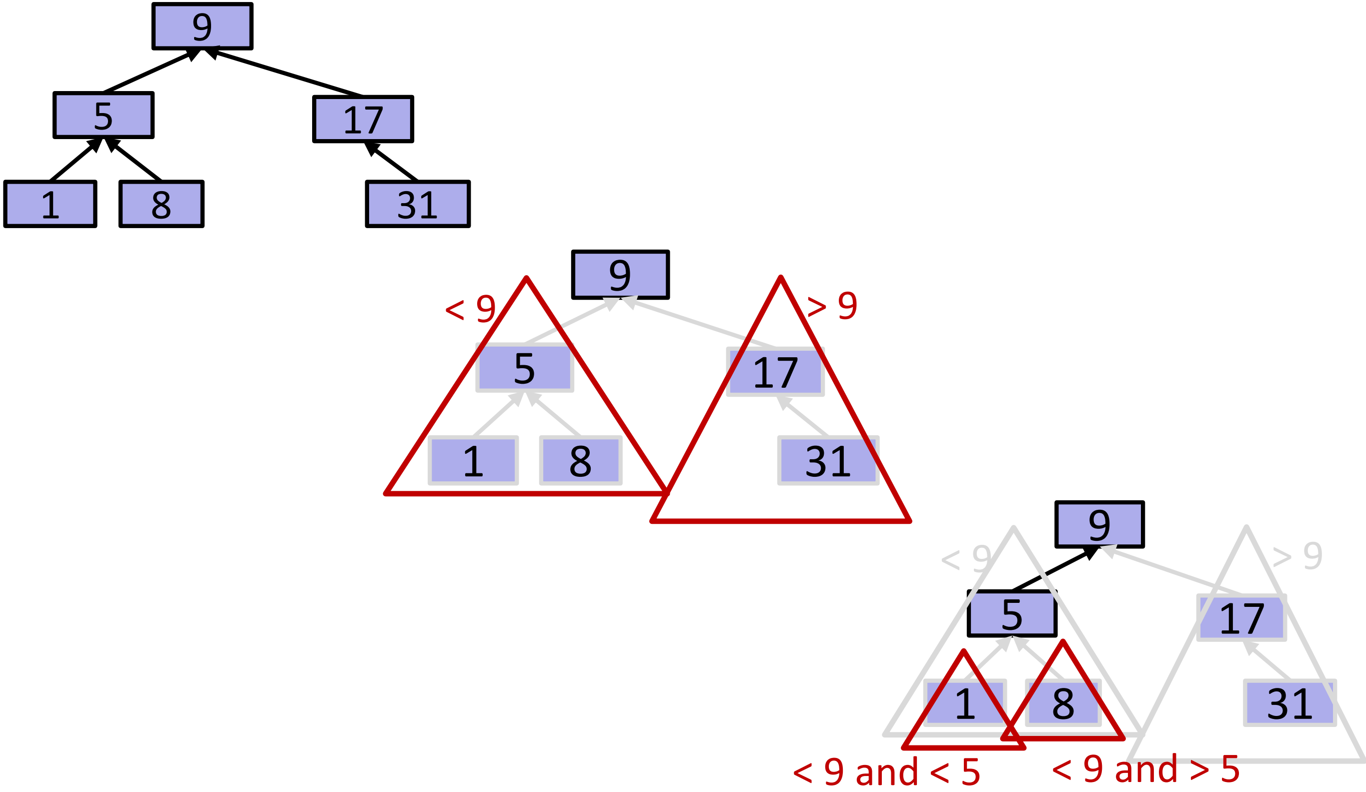
[✓ B. Yes / No]

C. No / Yes

D. No / No

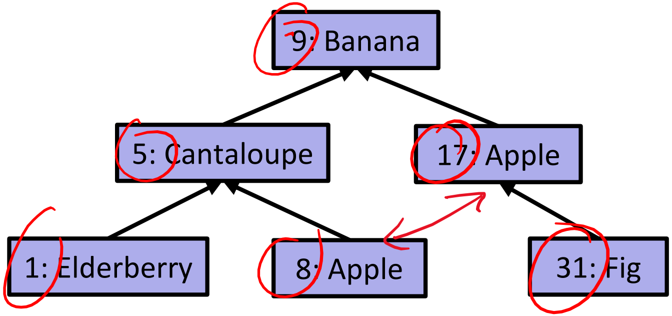


BST Ordering Applies *Recursively*



Binary Search Trees as Maps

- Since BSTs contain keys, they can also contain (key, value) pairs



```
class BSTNode<Key, Value> {
  Key k;
  Value v;
  BSTNode left;
  BSTNode right;
}
```

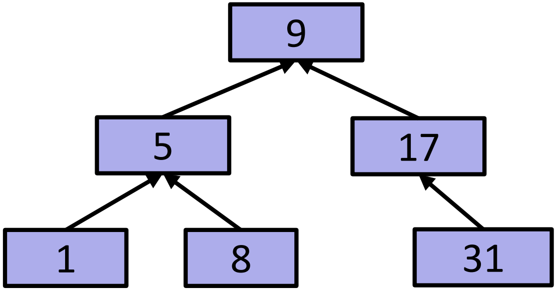
- Sorted by keys, not values
- Values can be duplicated

Lecture Outline

- Binary Search vs Binary Range Search
- ADTs: Maps and Sets
- Binary Search Trees as Maps and Sets
- BST Operations:
  - **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?



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

}
```

A,B,C,D } either will work
A,B,D,C

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

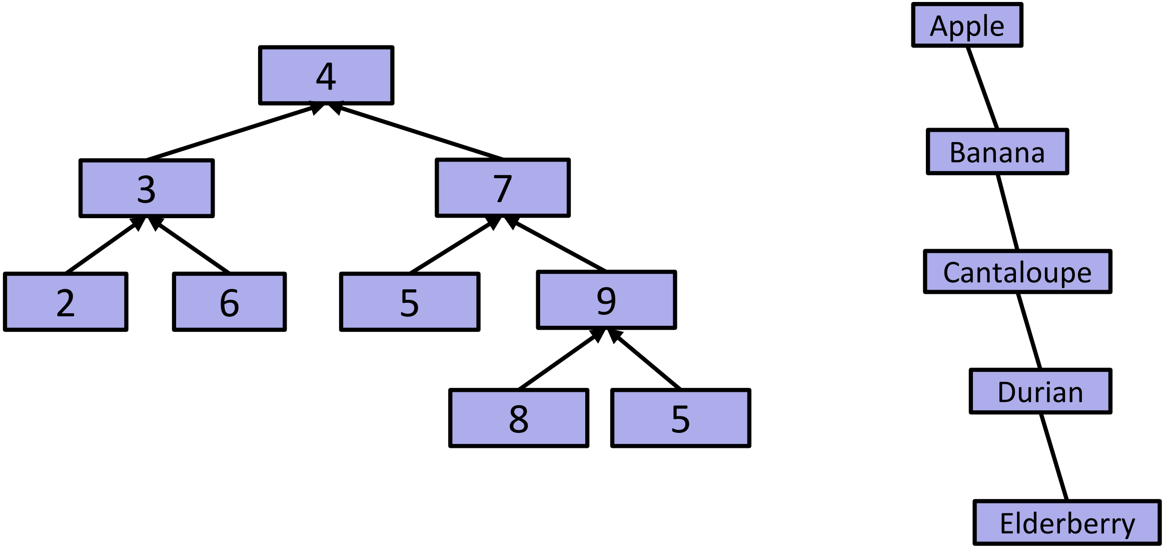
B
```
if (k.equals(n.key))
  return true;
```

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

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

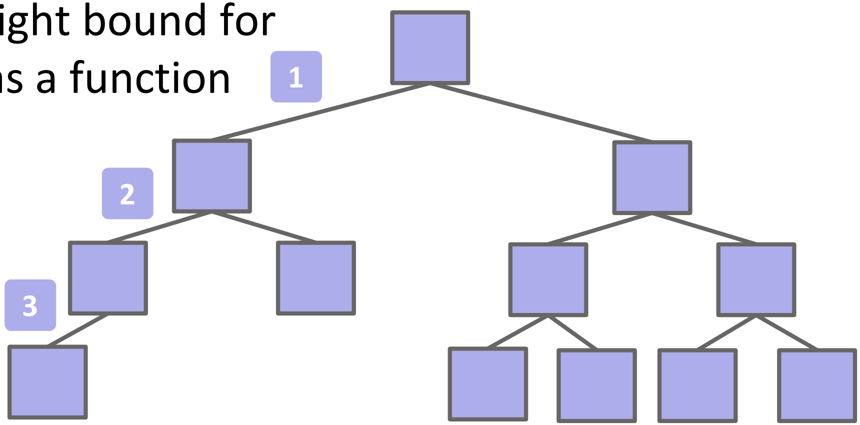
BST Find/Contains's runtime

- What is find's *worst-case* runtime, as a function of n? } don't know yet!
- What is find's worst-case runtime, as a function of *height*? } $\Theta(h)$



BST Height (or depth)

- The **height** of a binary search tree is the number of edges on *the longest path* between the root node and any leaf
  - A **path** is a connected sequence of edges that join parent-child nodes
  - The height of this tree is **3**
- We don't have a tight bound for the tree's height as a function of its size!



Lecture Outline

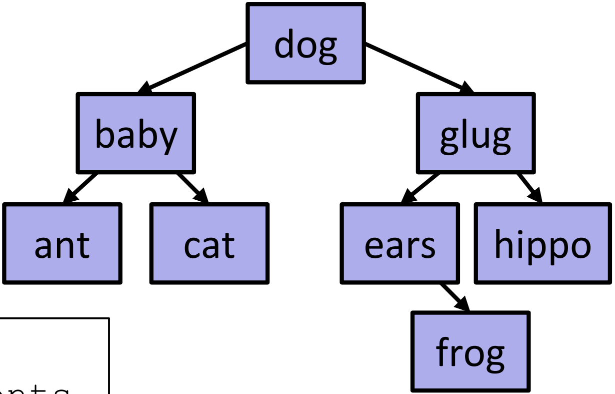
- Binary Search vs Binary Range Search
- ADTs: Maps and Sets
- Binary Search Trees as Maps and Sets
- BST Operations:
  - Find/Contains
  - **Add**
  - Remove

Binary Search Trees: Add

- Where does the new key belong?

Easiest to add to a leaf

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



```
BSTNode add(BST t, Key k) {
  // Implement by putting statements
  // in the correct order
}
```

D, B, C, A } either will work
D, C, B, A

Same runtime as find: Θ(h)

A
```
return t;
```

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

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

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

Lecture Outline

- Binary Search vs Binary Range Search
- ADTs: Maps and Sets
- Binary Search Trees as Maps and Sets
- BST Operations:
  - Find/Contains
  - Add
  - **Remove (to be continued Friday)**