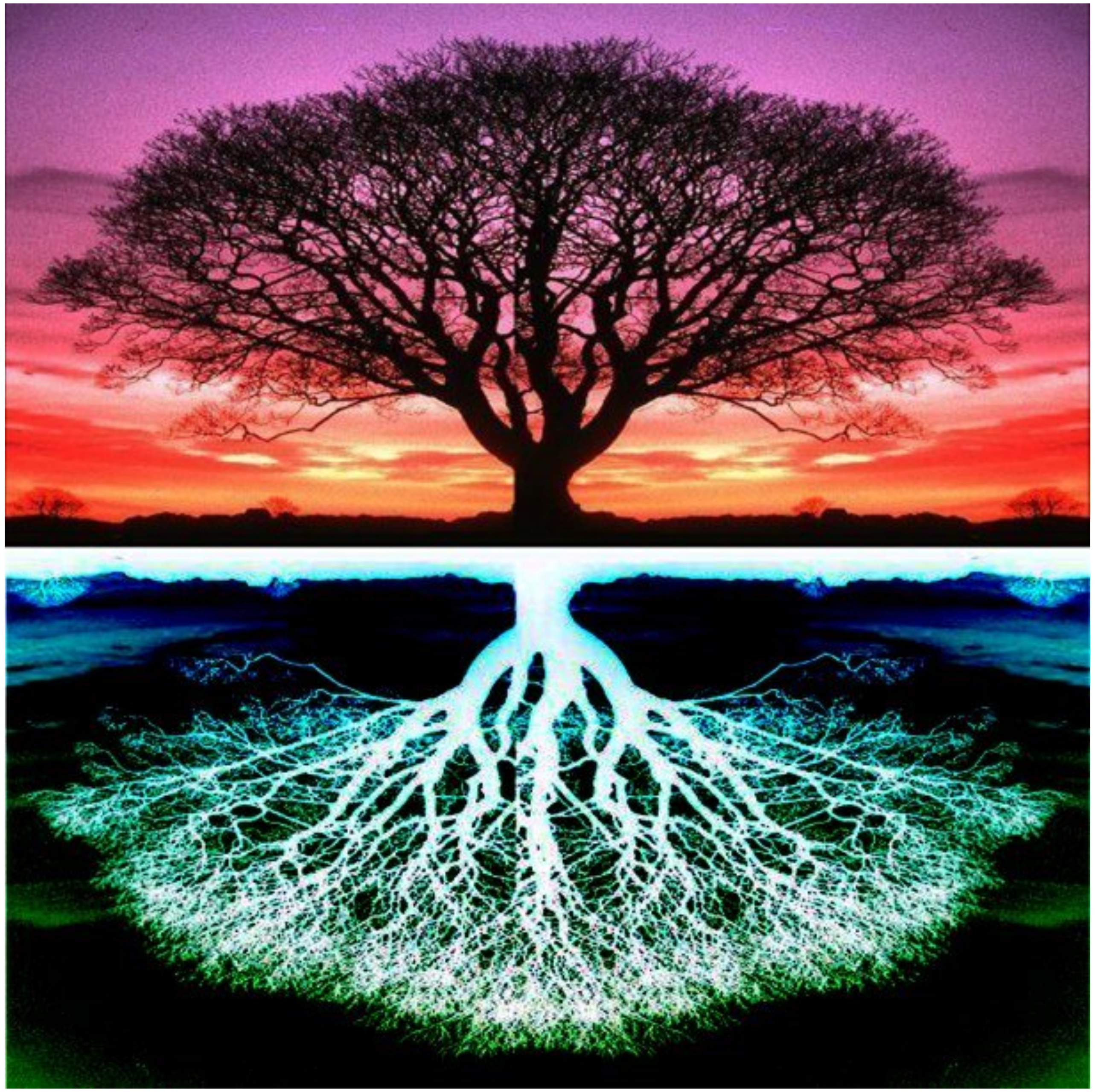


CSCI 241

Lecture 9

Binary Search Trees

Announcements

- A1 is in!
- Reminder again: if you submit late, you need to email me after you submit so I can pull your latest changes from Github.
- Aiming for Friday A2 release

Goals

- Know the definition and uses of a binary search tree.
- Be prepared to implement, and know the runtime of, the following BST operations:
  - searching
  - inserting
  - deleting

Tree Terminology

M is the ***root*** of this tree

G is the ***root*** of the ***left subtree*** of M

B, H, J, N, S are ***leaves*** *(have no children)*

N is the ***left child*** of P

S is the **right *child*** of P

P is the **parent** of N

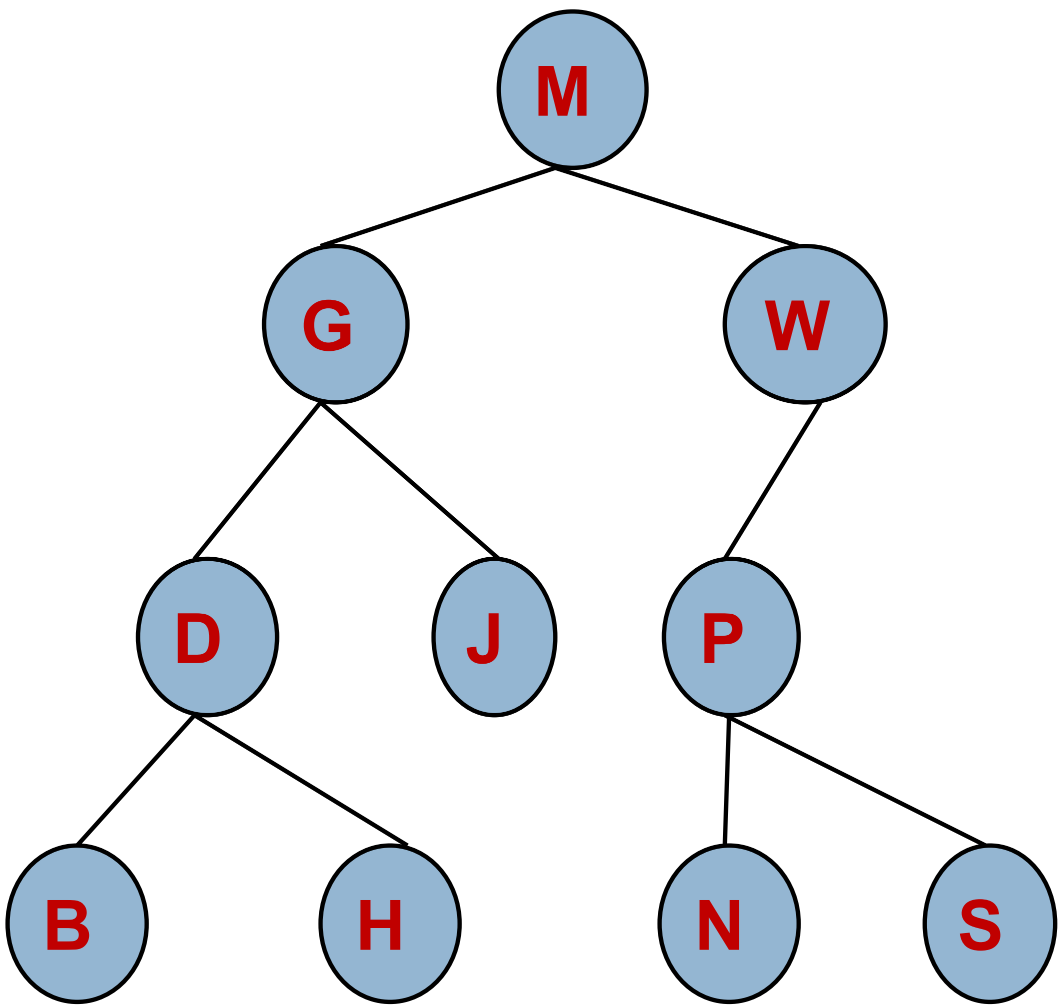
M and G are ***ancestors*** of D

P, N, S are ***descendants*** of W

J is at ***depth*** 2 (length of path from root)

The subtree rooted at W has ***height*** (length of longest path to a leaf) of 2

A collection of several trees is called a ____?



Tree Terminology: Lighting Round!

ABCD:

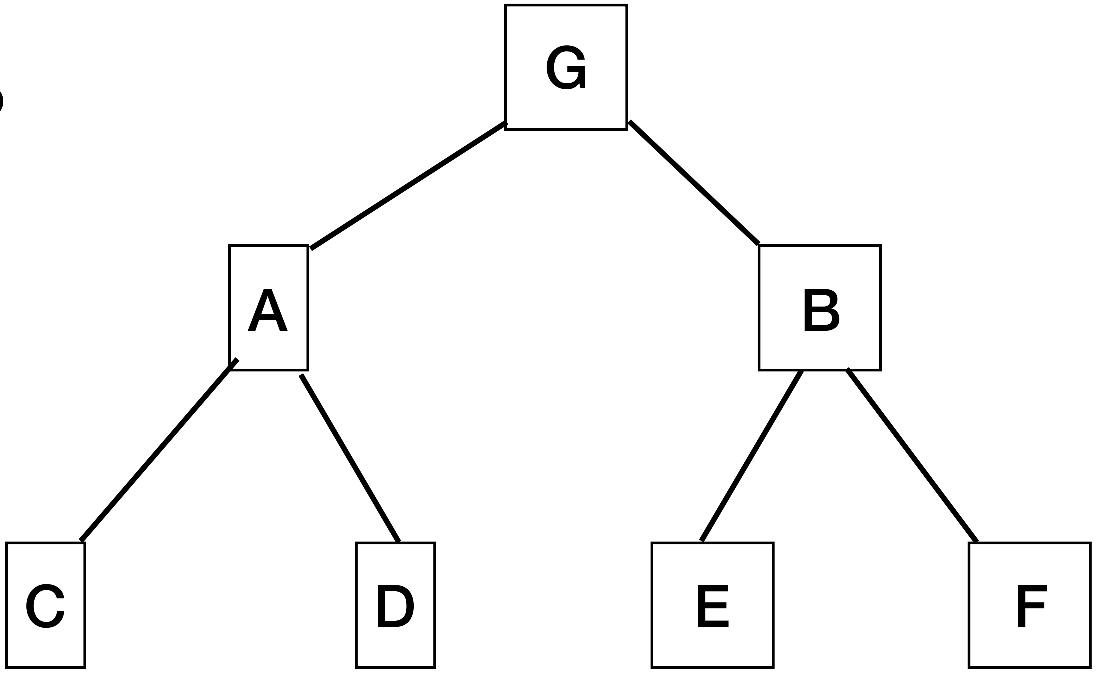
What's the root of G's right subtree?

What's an ancestor of F?

What's C's parent?

What's a node at depth 1?

What's a node at the root of a subtree of height 0?

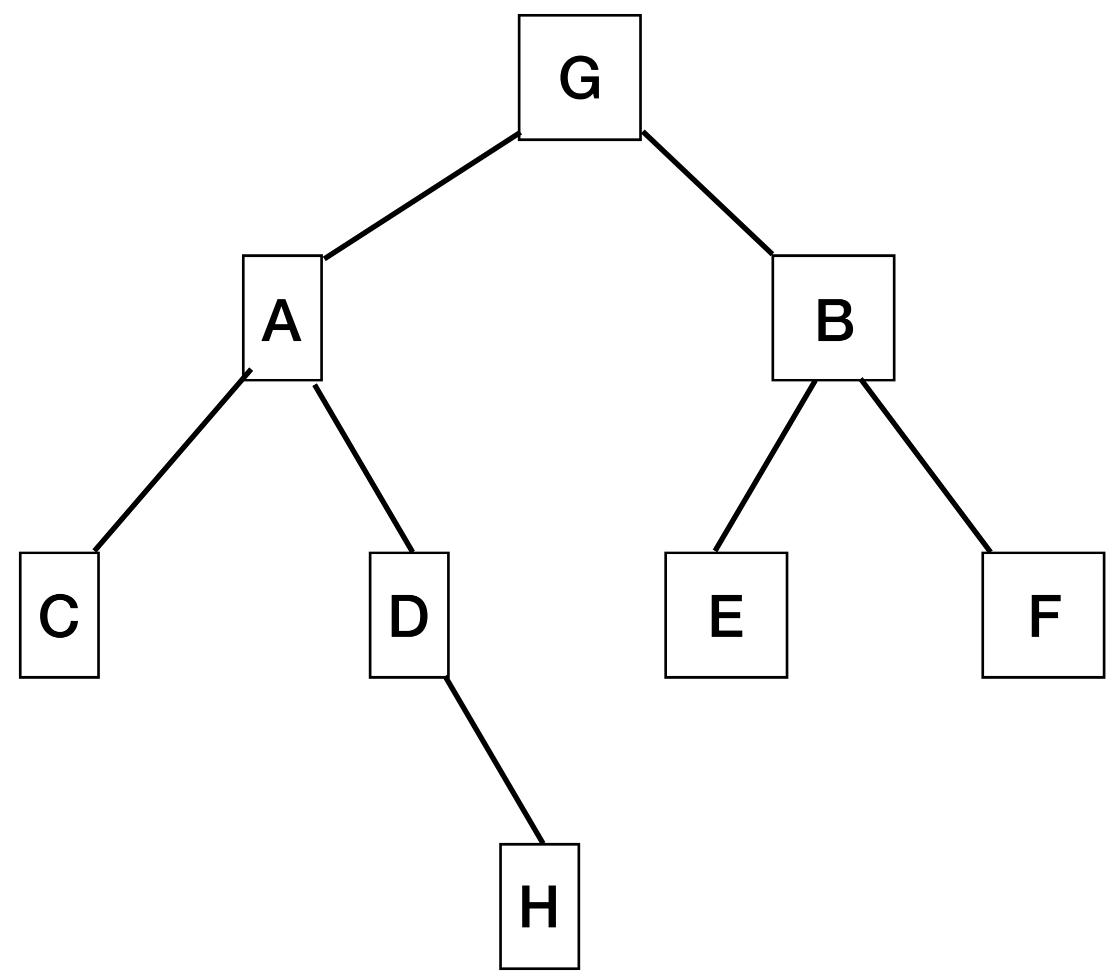


Tree Terminology: Lighting Round!

ABCD:

What’s the height of the tree rooted at G?

A. 1
B. 2
C. 3
D. 4



Tree Terminology: Lighting Round!

ABCD:

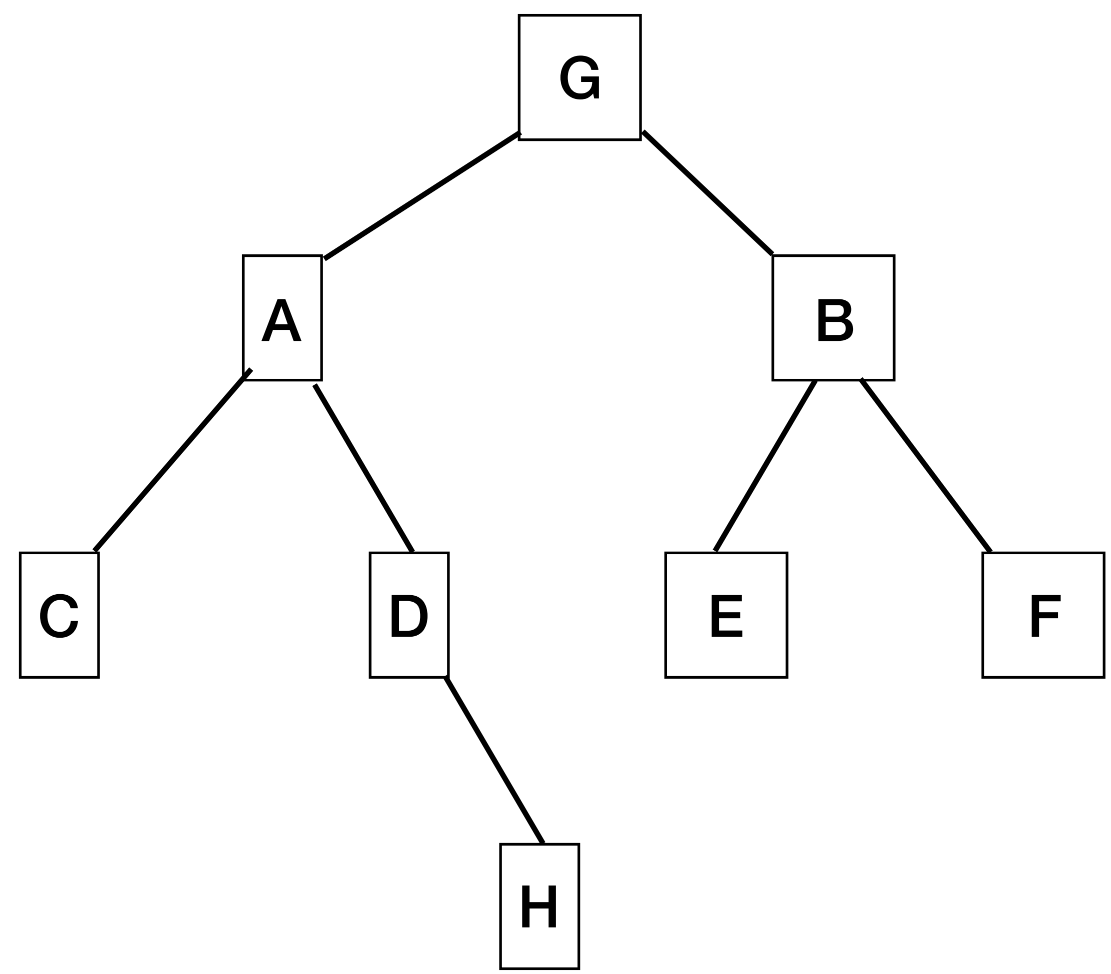
What's the depth of node D?

A. 1

B. 2

C. 3

D. 4



Binary Tree

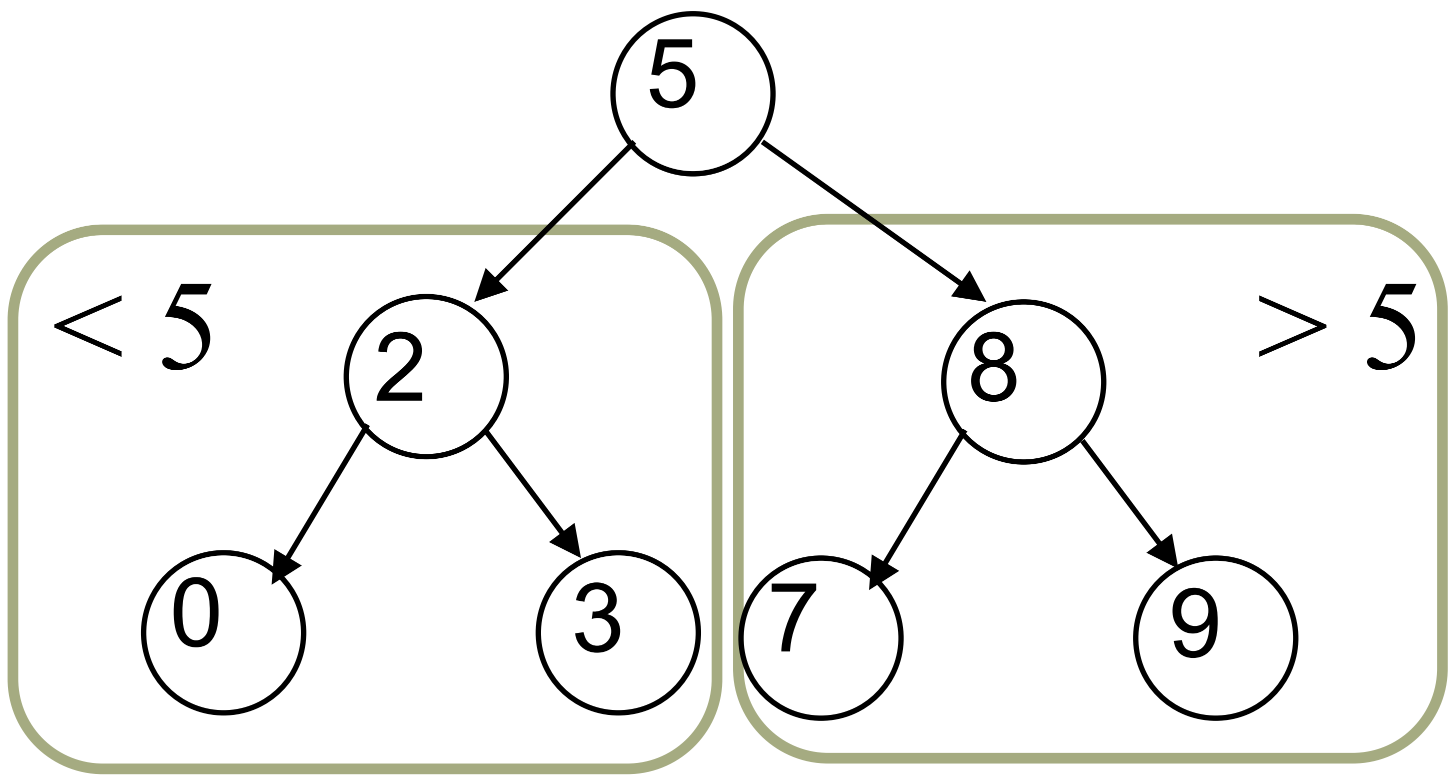
```
public class Tree {
  int value;
  Tree parent;
  Tree left;
  Tree right;
}
```

Binary Search Tree

```
/** BST: a binary tree, in which:
 * -all values in left are < value
 * -all values in right are > value
 * -left and right are BSTs */
public class BST {
  int value;
  BST parent;
  BST left;
  BST right;
}
```

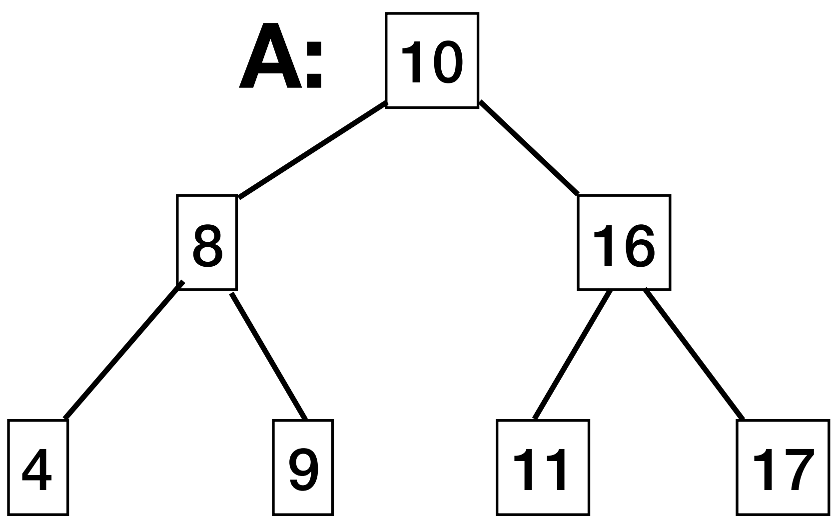
consequence: no duplicates!

Binary Search Tree

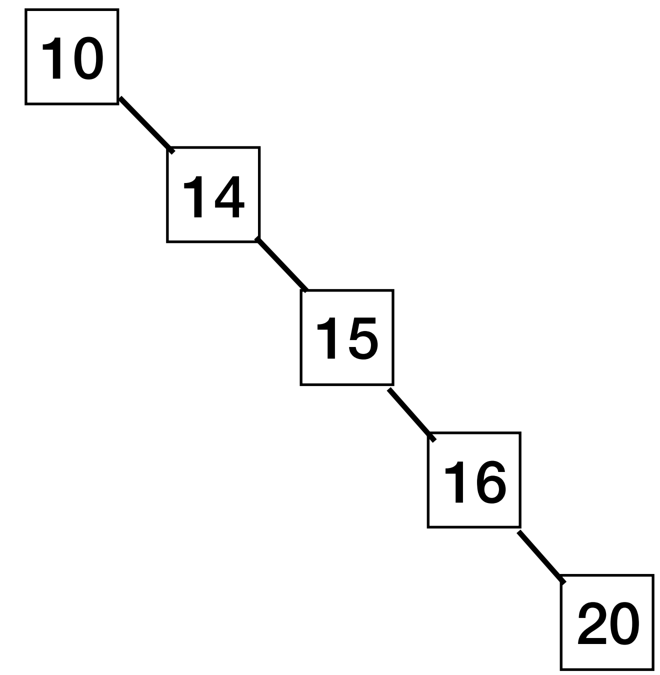


ABCD: Which of these is **not** a binary search tree?

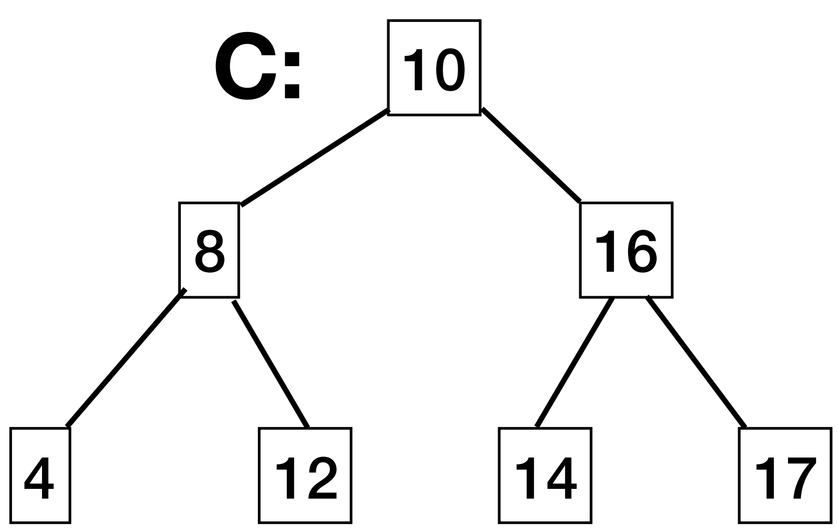
A:



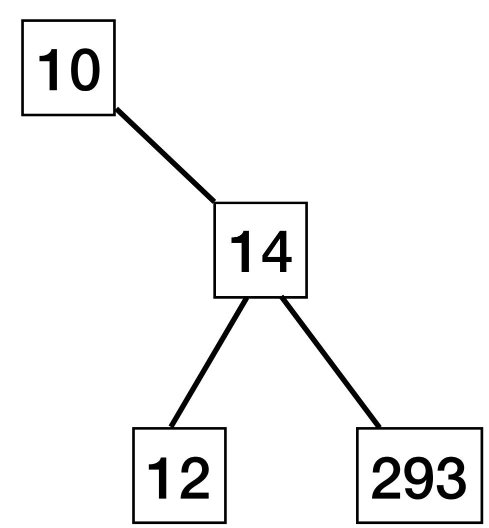
B:



C:



D:



Traversing a BST

pre-order traversal:

1. Process root
2. Process left subtree
3. Process right subtree

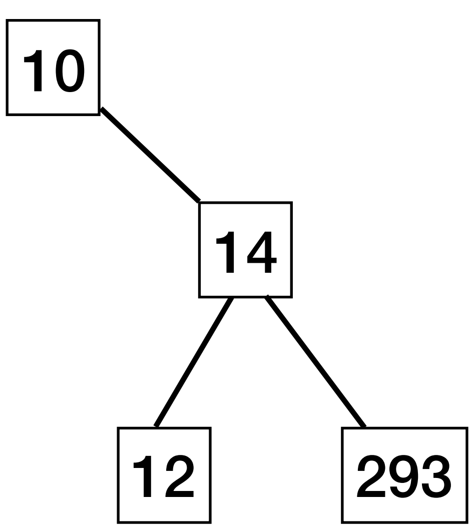
in-order traversal:

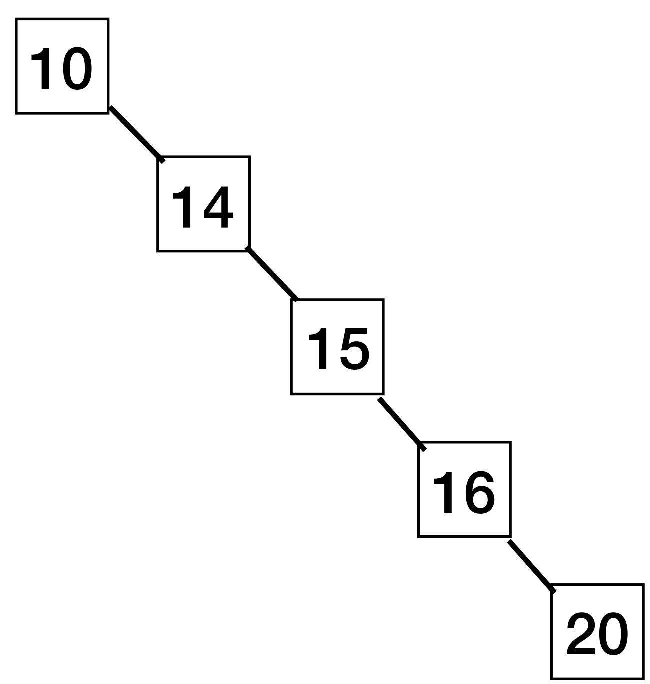
1. Process left subtree
2. Process root
3. Process right subtree

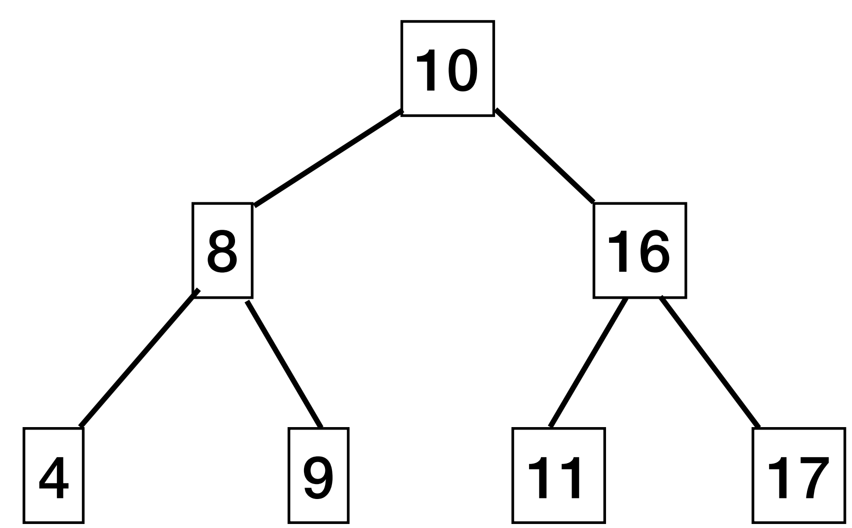
post-order traversal:

1. Process left subtree
2. Process right subtree
3. Process root

Write the values printed by an **in-order** traversal of each of the following BSTs:







Searching a Binary Tree

(not Search!)

- A **binary tree** is
  - Empty, or
  - Three things:
    - value
    - a left **binary tree**
    - a right **binary tree**

(not BST!)

Find v in a binary tree:

```
boolean findVal(Tree t, int v):
```

(base case - not found!)

```
    if t == null:
        return false
```

(base case - is this v?)

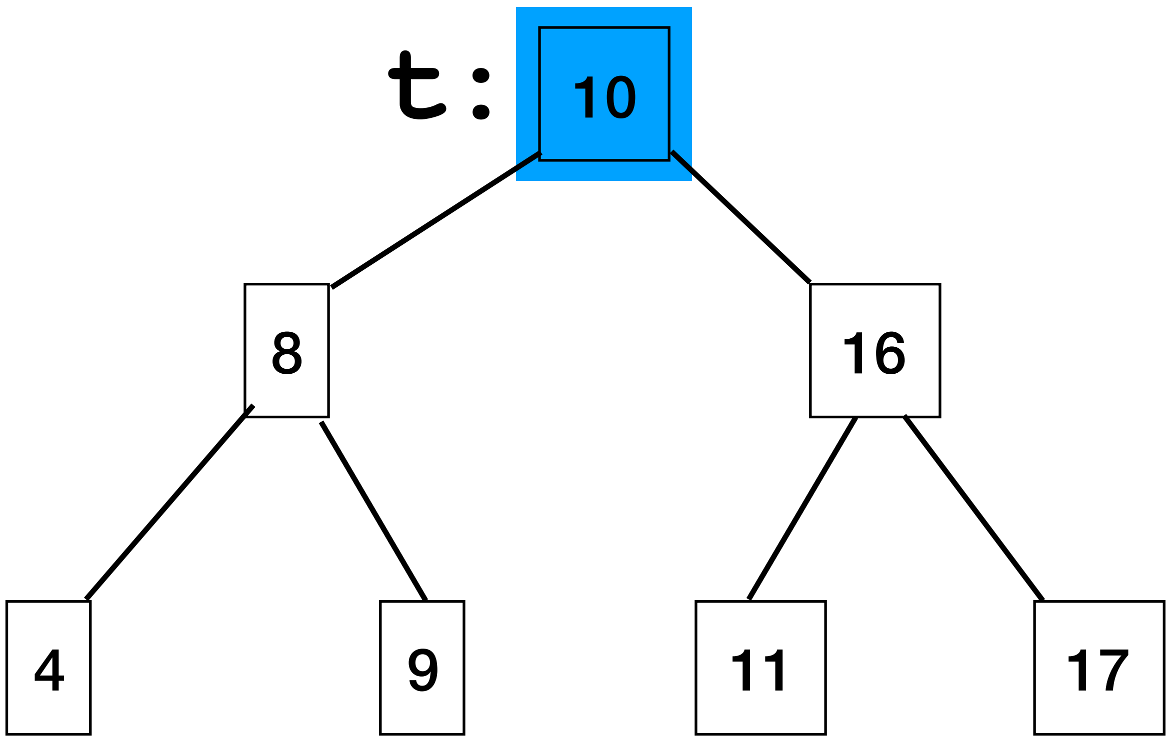
```
    if t.value == v: return true
```

(recursive call - is v in left?)

```
    return findVal(t.left)
        || findVal(t.right)
```

(recursive call - is v in right?)

Searching a BST

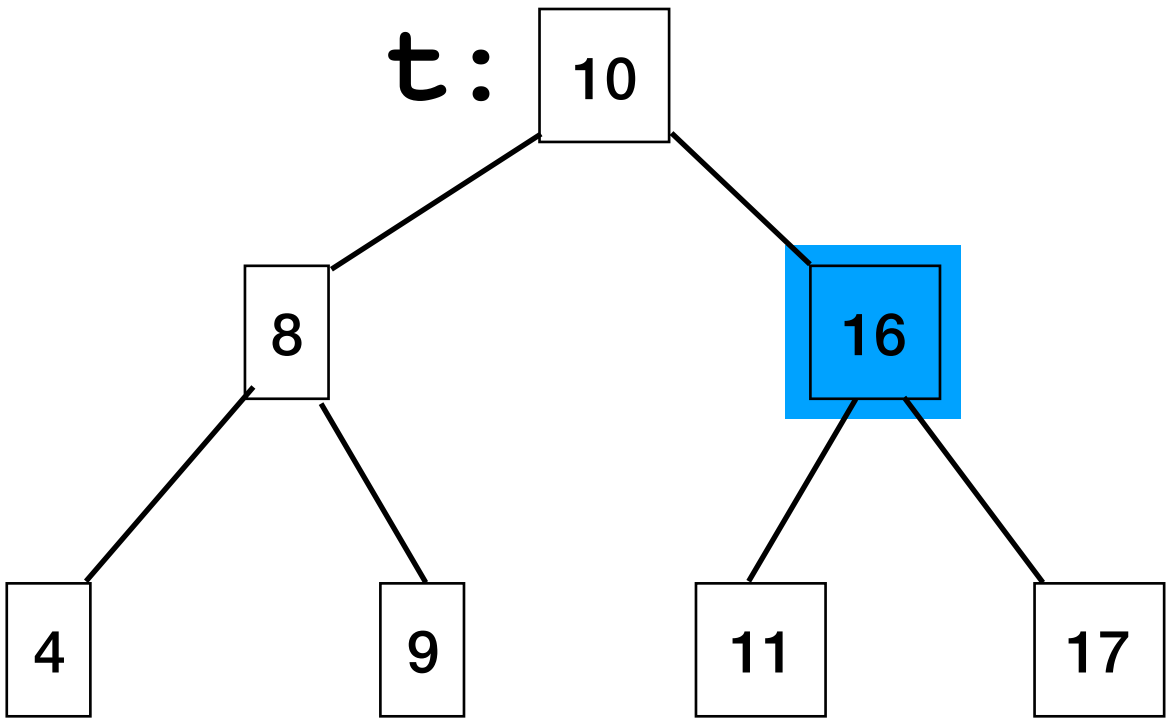


`search(t, 11)`

11 > 10

`search(right, 11)`

Searching a BST



`search(t, 11)`

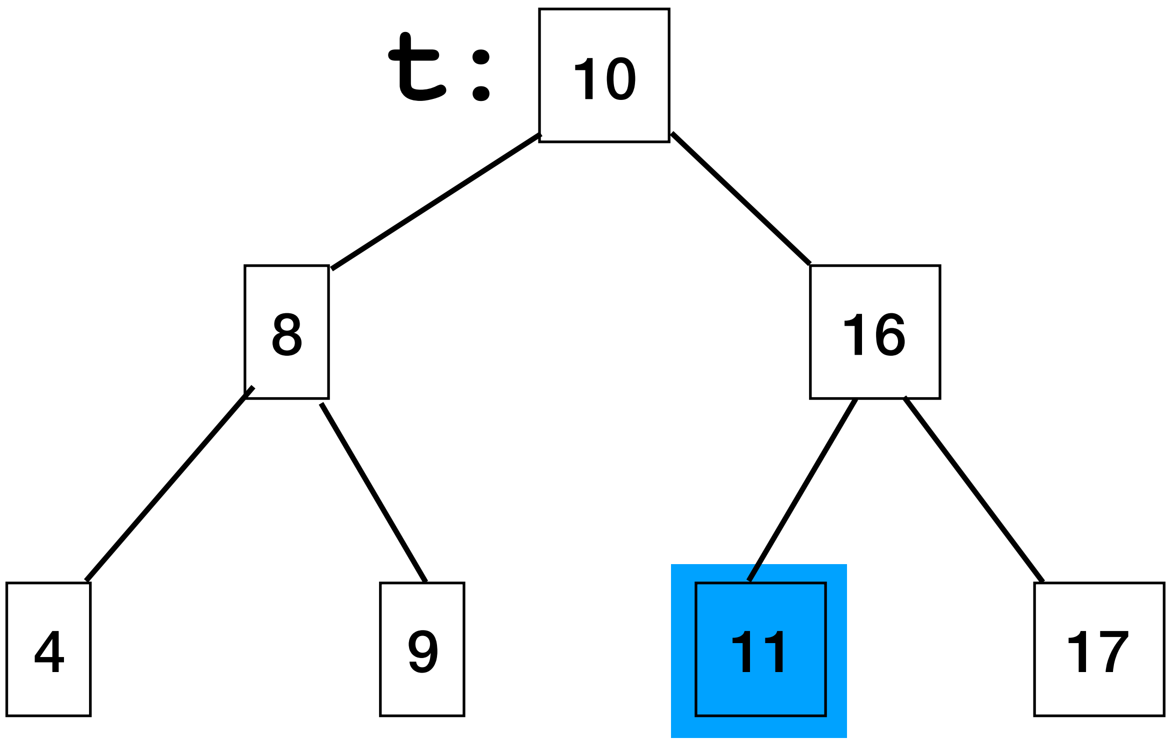
11 > 10

`search(right, 11)`

11 < 16

`search(left, 11)`

Searching a BST




```
search(t, 11)
```

11 > 10

```
search(right, 11)
```

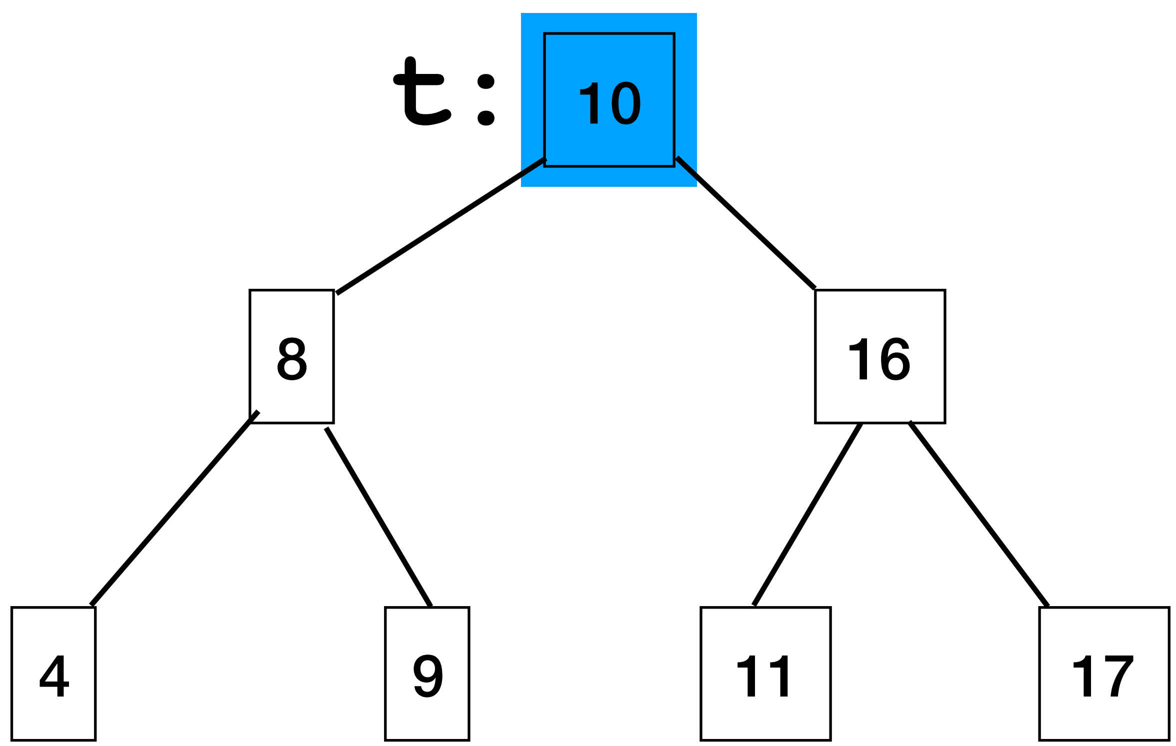
11 < 16

```
search(left, 11)
```

11 == 11

```
found it! return.
```

Searching a BST - the nonexistent case

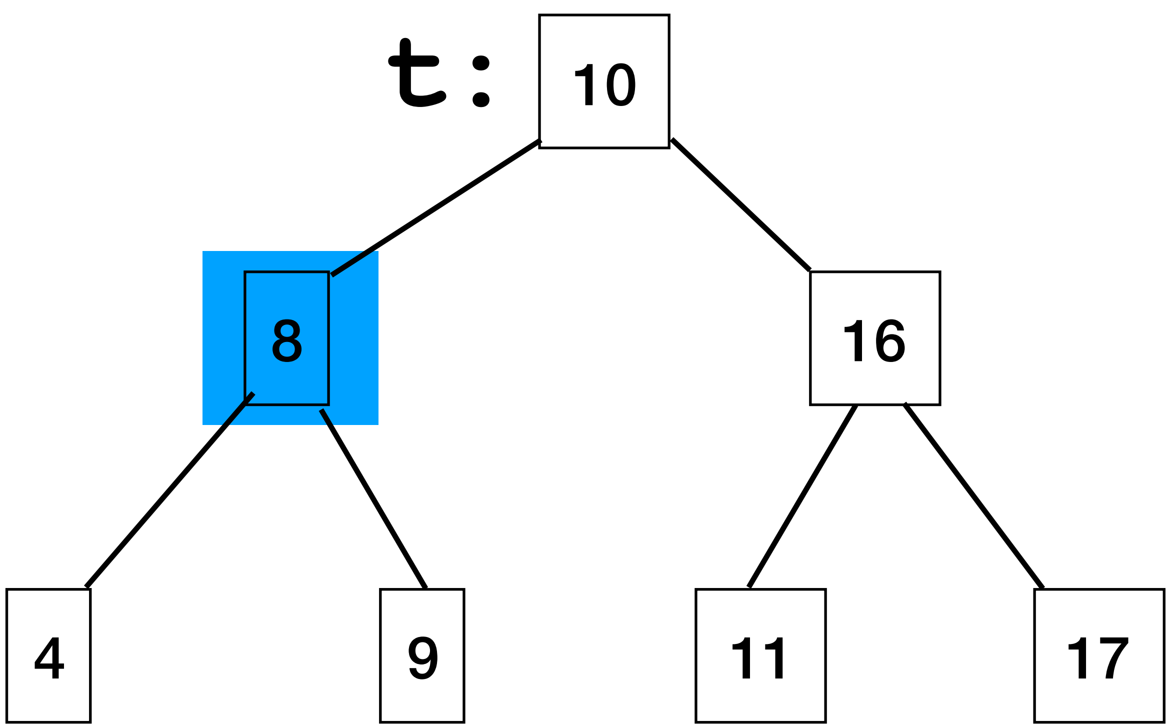


`search(t, 5)`

5 < 10

`search(left, 5)`

Searching a BST - the nonexistent case



`search(t, 5)`

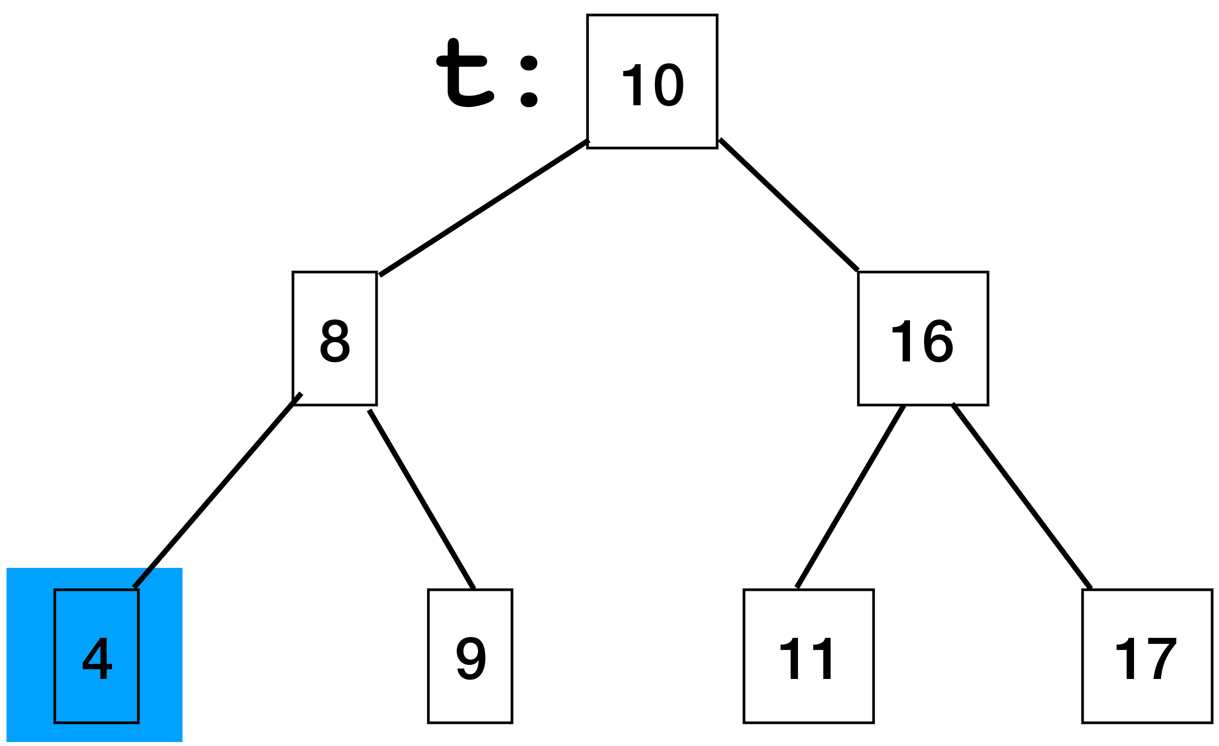
5 < 10

`search(left, 5)`

5 < 8

`search(left, 5)`

Searching a BST - the nonexistent case



`search(t, 5)`

5 < 10

`search(left, 5)`

5 < 8

`search(left, 5)`

5 > 4

`search(right, 5)`

`null - not found!`

Searching: BT vs BST

```java
/** Searches the binary tree
 * rooted at n for value v,
 * returning true iff it is
 * in the tree. */
boolean srchBT(n, v) {
  if (n == null) {
    return false;
  }
  if (n.v == v) {
    return true;
  }
  return srchBT(n.left,  v)
      || srchBT(n.right, v);
}
```

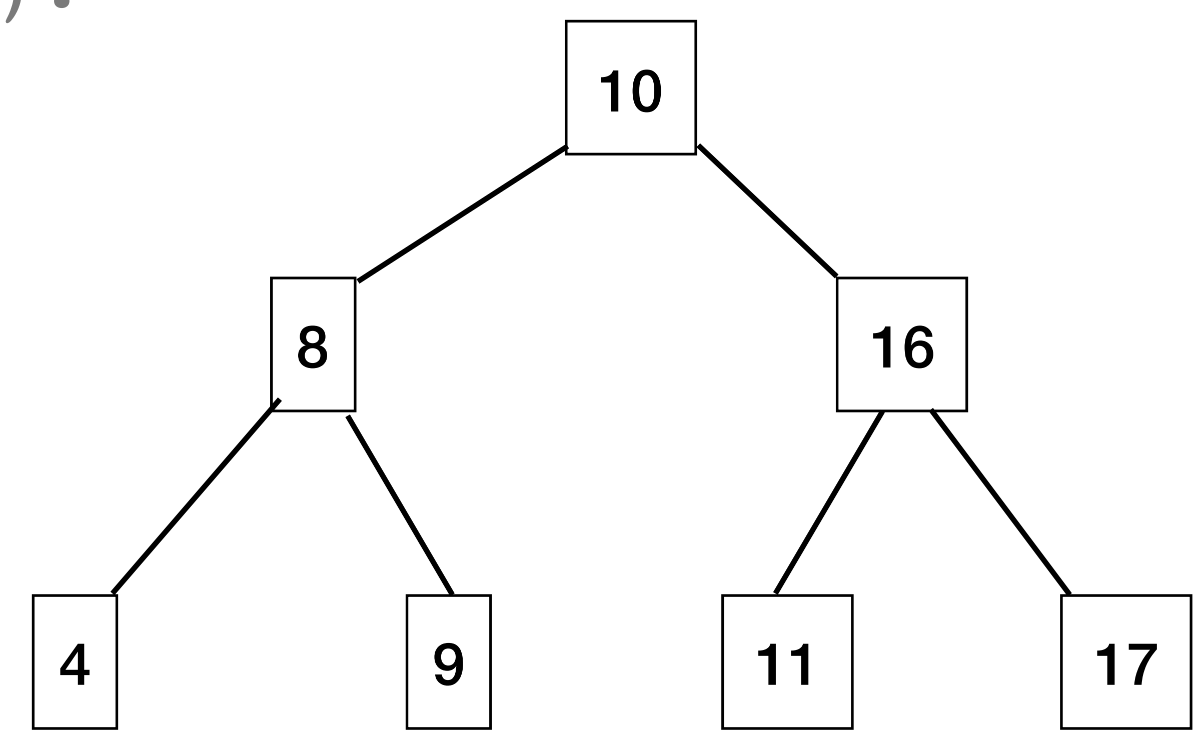
Two recursive calls

```java
/** Searches the binary *search*
 * tree rooted at n for value v,
 * returning true iff it is in
 * the tree. */
public srchBST(n, v) {
  if (n == null) {
    return false;
  }
  if (n.v == v) {
    return true;
  }
  if (v < n.v) {
    return srchBST(n.left, v);
  } else {
    return srchBST(n.right, v);
  }
}
}
```

One recursive call!

Searching a BST: What's the runtime?

```
boolean search(BST t, int v):
  if t == null:
    return false
  if t.value == v:
    return true
  if v < t.value:
    return search(t.left)
  else:
    return search(t.right)
```



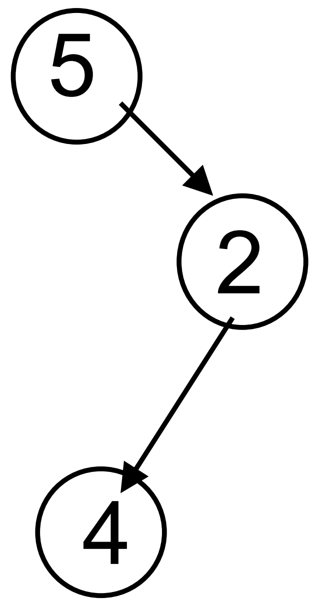
If h is the tree's **height**, search can visit at most h+1 nodes!

Runtime of search is **O(h)**.

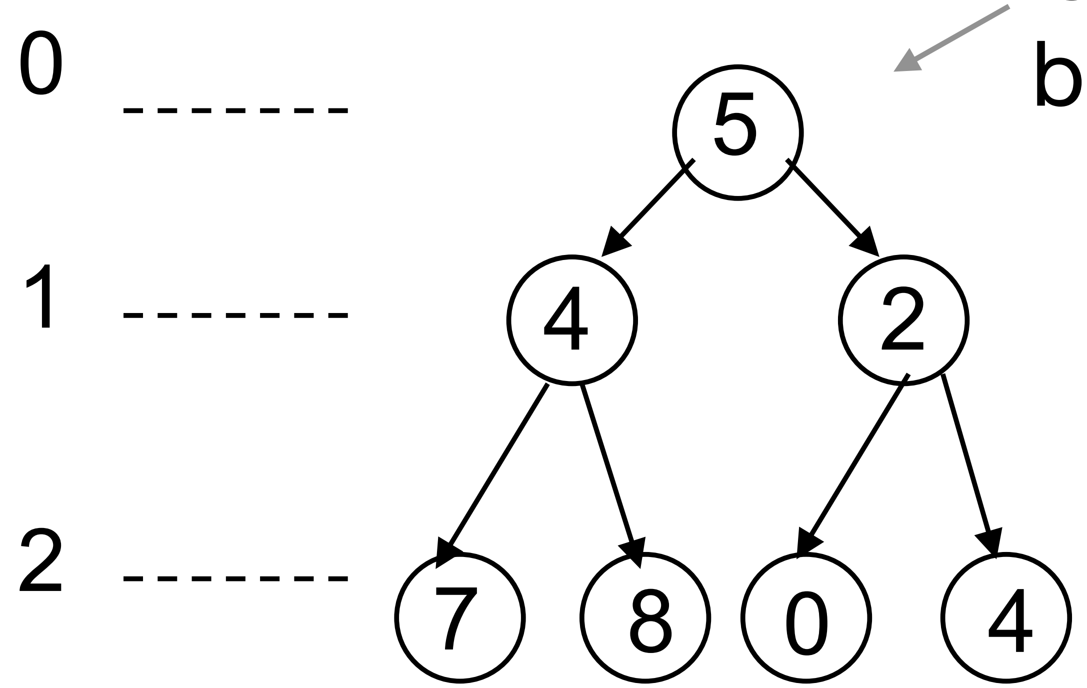
That's great, but how does h relate to n, the number of nodes?

How many nodes does a tree with height h have?

Consider h = 2:



depth



Complete binary tree

Fewest possible:

$n = h+1$

n is $O(h)$

h is O(n)

Most possible:

At depth d: 2^d nodes possible.

At all depths: $2^0 + 2^1 + \ldots + 2^h$

$= 2^{h+1} - 1$

$n = 2^{h+1} - 1$

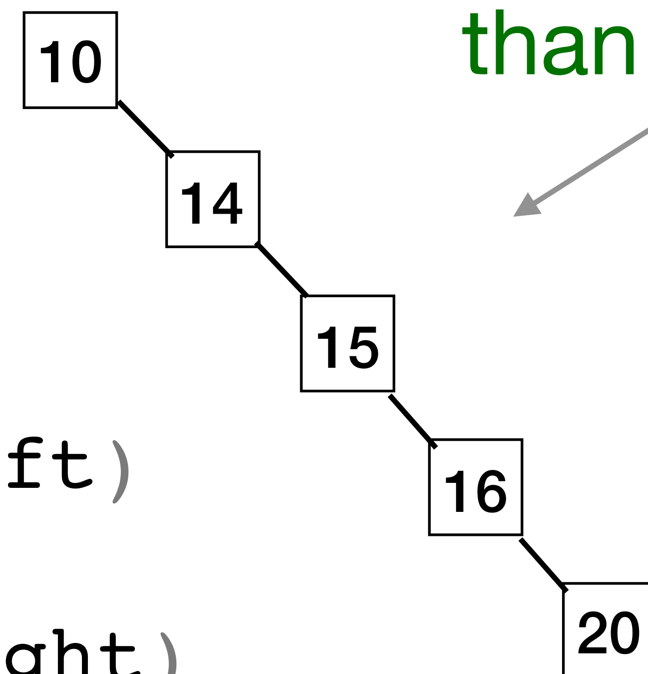
n is $O(2^h)$

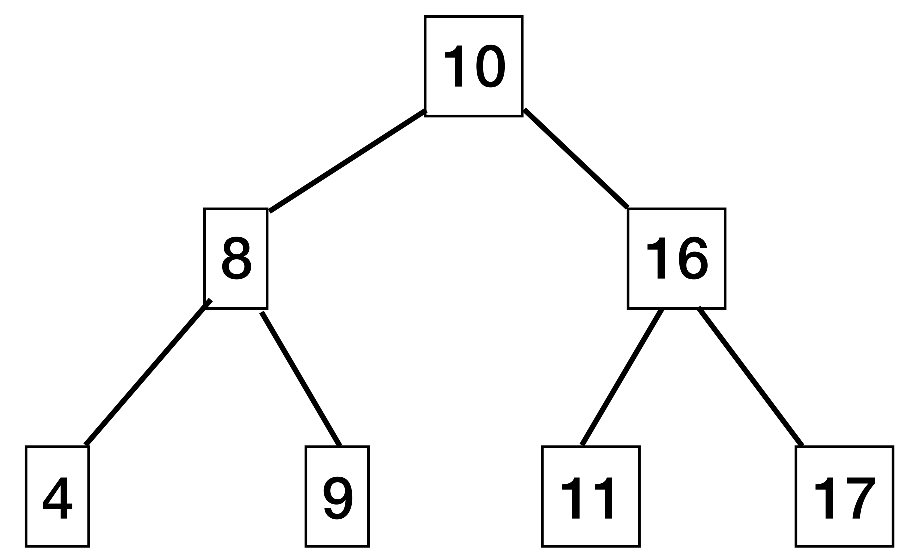
h is O(log n)

Searching a BST: What's the runtime?

```
boolean search(BST t, int v):
  if t == null:
    return false
  if t.value == v:
    return true
  if t.value < v:
    return search(t.left)
  else:
    return search(t.right)
```

We want our trees to look more like this than this

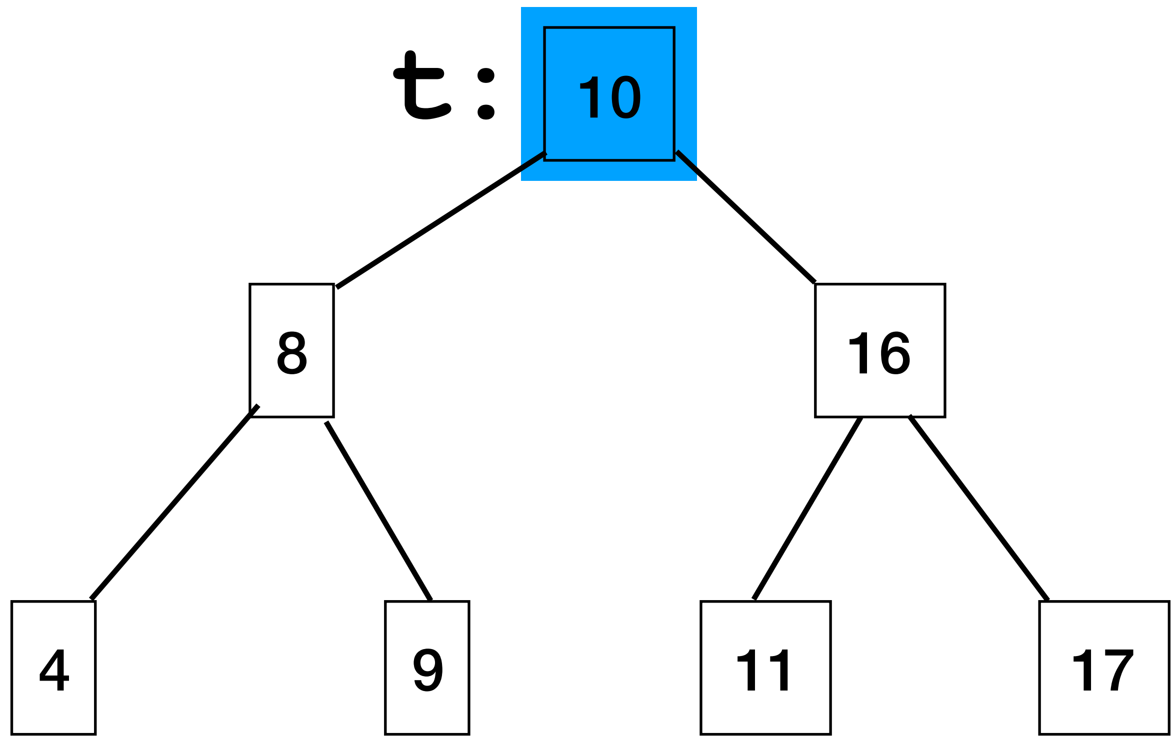




Runtime of search is O(h). Worst: O(n) Best: O(log n)

Inserting into a BST

Inserting into a BST

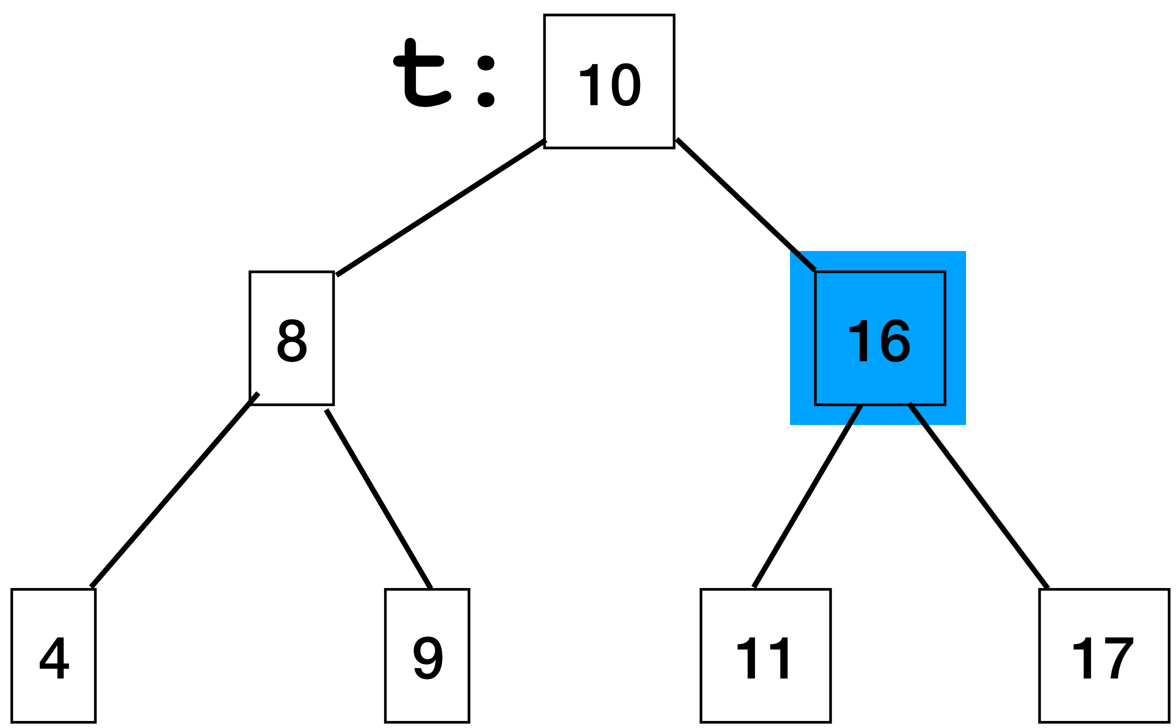


`insert(t, 11)`

11 > 10

`insert(right, 11)`

Inserting into a BST



`insert(t, 11)`

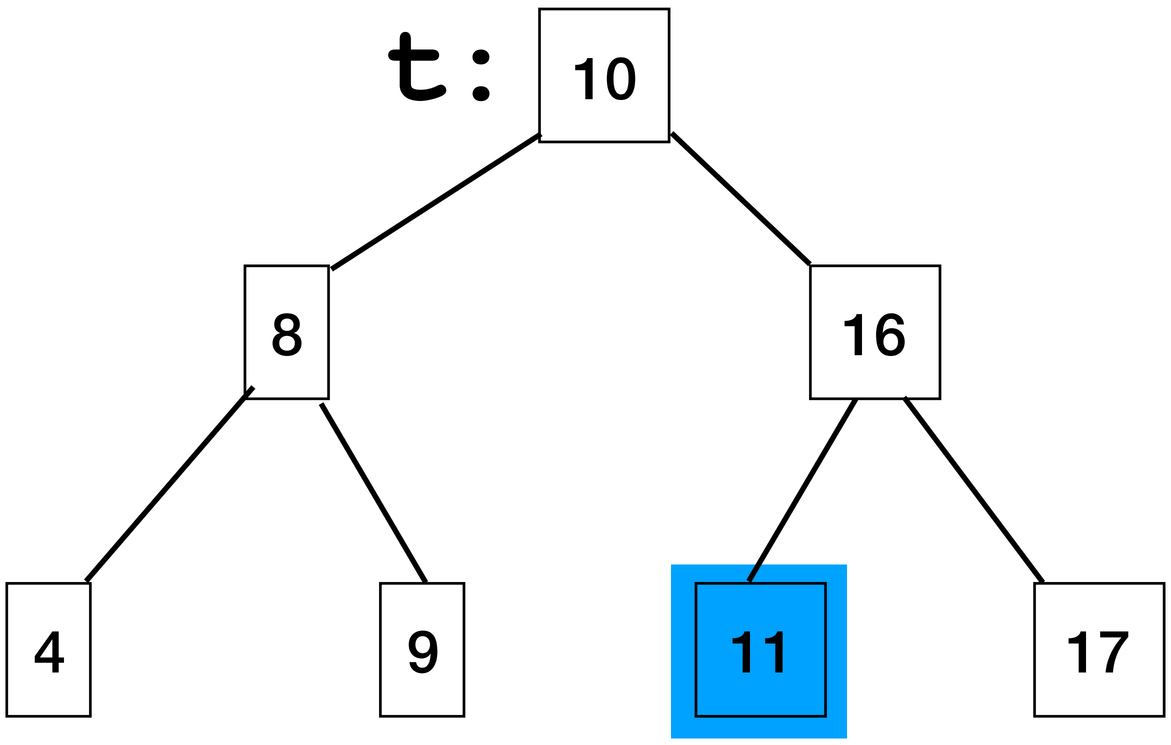
11 > 10

`insert(right, 11)`

11 < 16

`insert(left, 11)`

Inserting into a BST



`insert(t, 11)`

11 > 10

`insert(right, 11)`

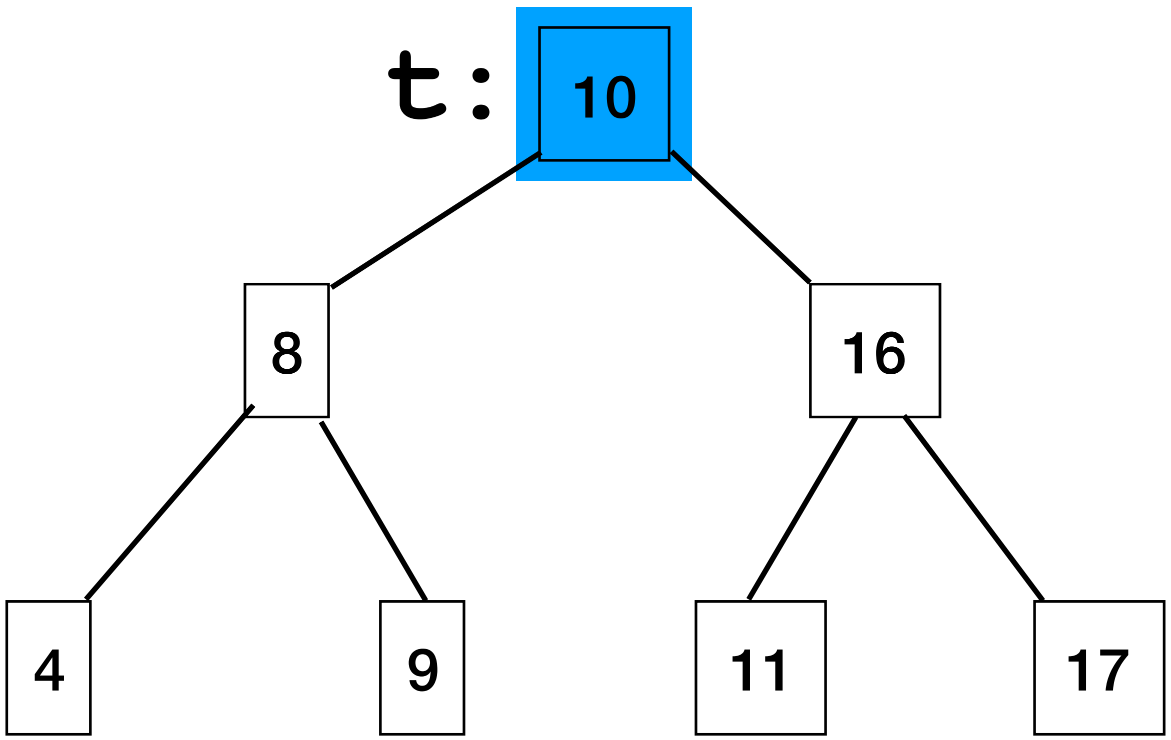
11 < 16

`insert(left, 11)`

11 == 11

`found it! no duplicates, allowed; nothing to do. return.`

Inserting into a BST - the nonexistent case

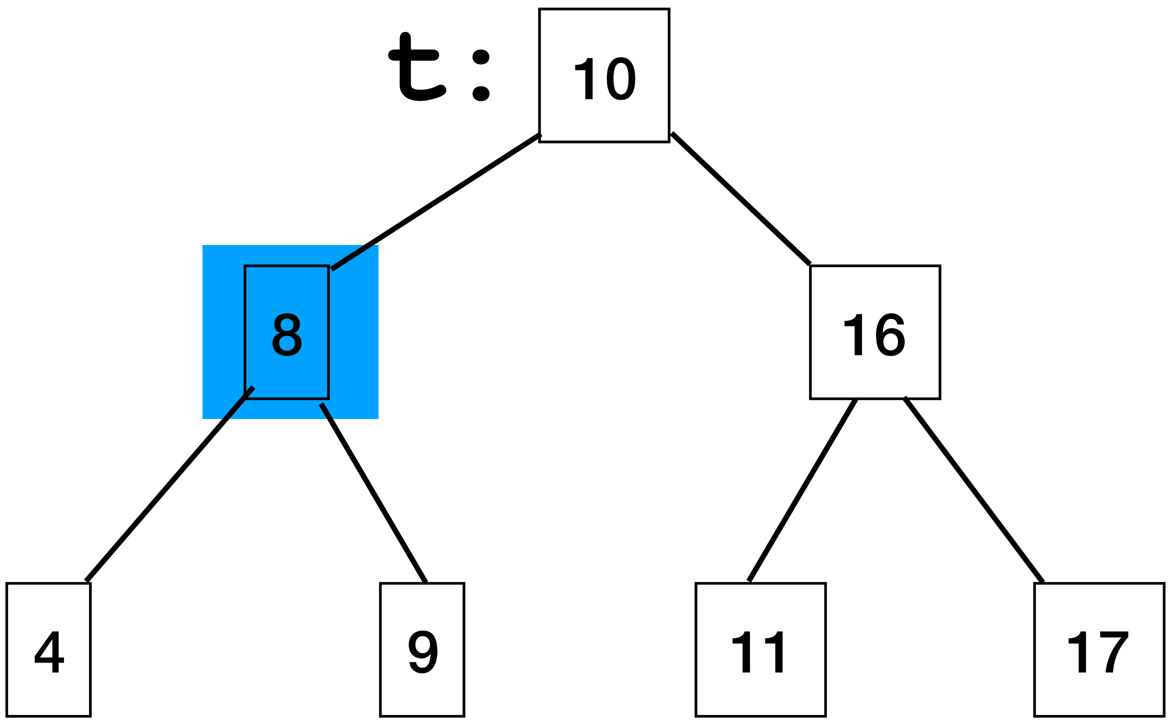



```
insert(t, 5)
```

5 < 10

```
insert(left, 5)
```

Inserting into a BST - the nonexistent case



`insert(t, 5)`

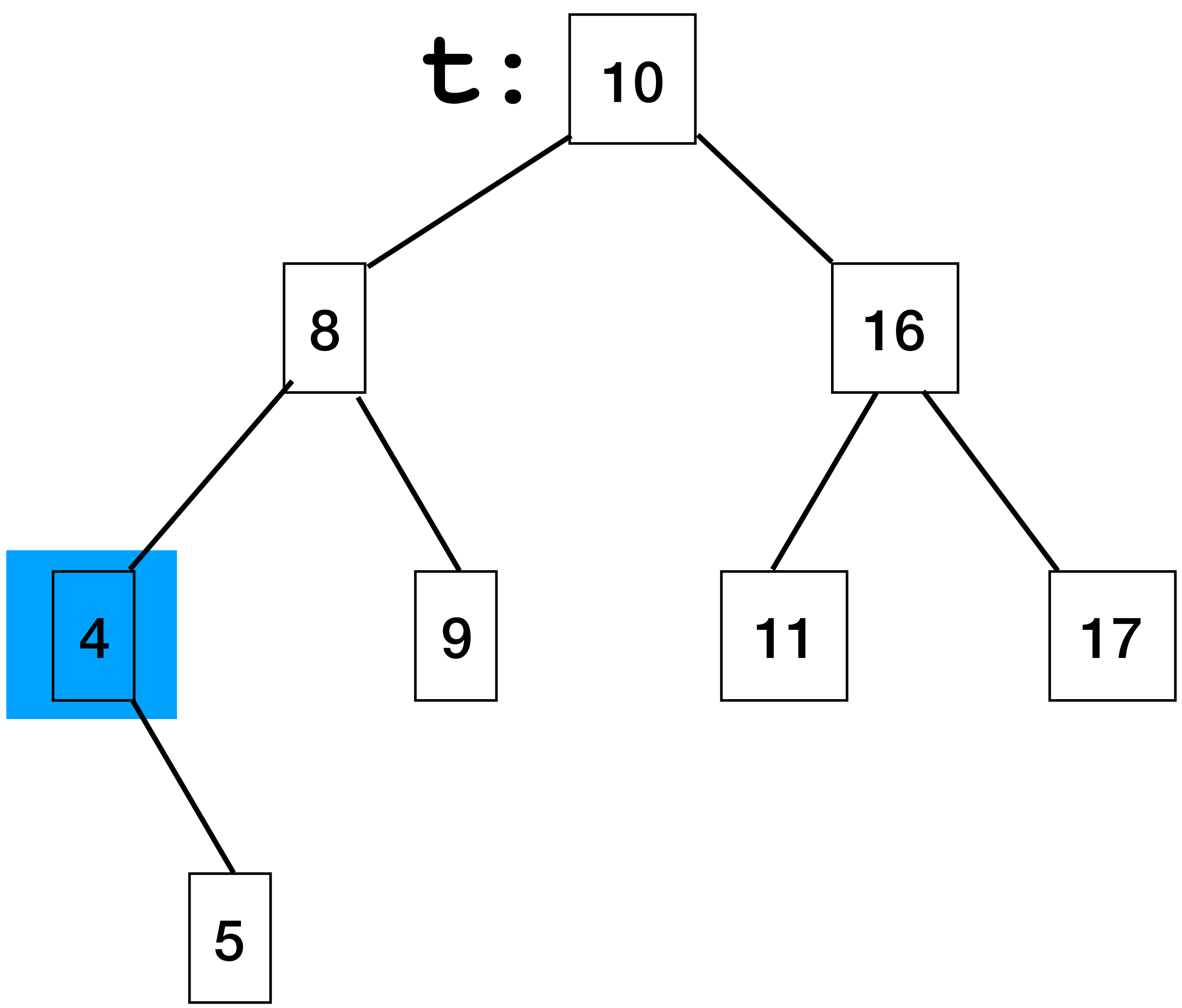
5 < 10

`insert(left, 5)`

5 < 8

`insert(left, 5)`

Inserting into a BST - the nonexistent case



`insert(t, 5)`

5 < 10

`insert(left, 5)`

5 < 8

`insert(left, 5)`

5 > 4

`insert(right, 5)`

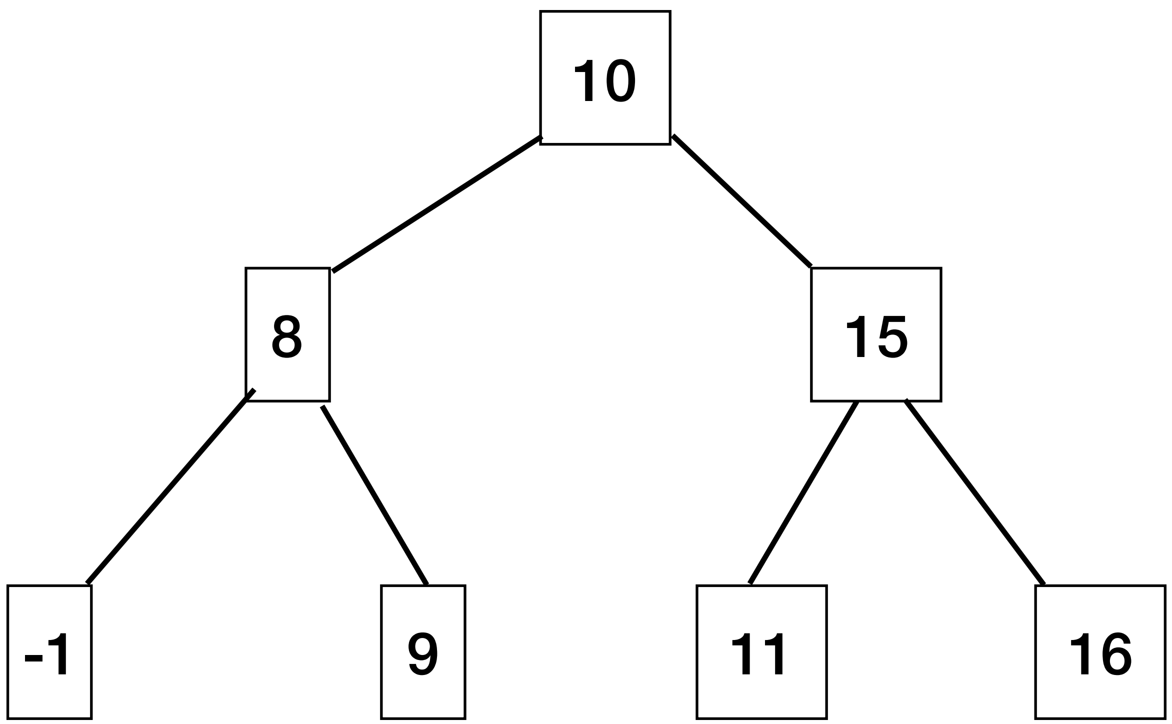
`null - not found. insert it here!`

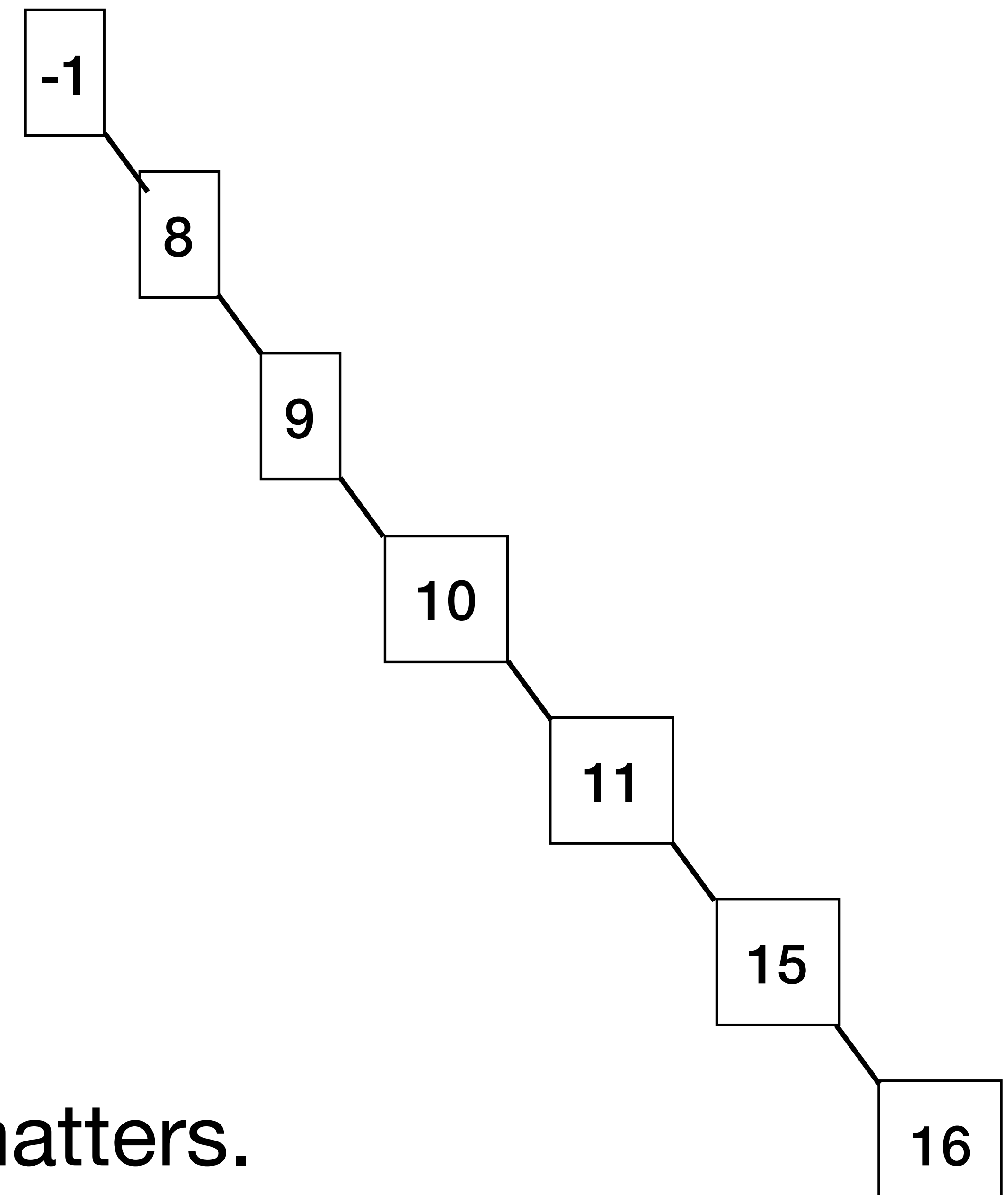
Let's Build Some Trees

```
t = new BST();
t.insert(10)
t.insert(15)
t.insert(16)
t.insert(8)
t.insert(16)
t.insert(9)
t.insert(11)
t.insert(-1)
```

```
t = new BST();
t.insert(-1)
t.insert(8)
t.insert(9)
t.insert(10)
t.insert(11)
t.insert(15)
t.insert(16)
t.insert(16)
```

Let’s Build Some Trees





Insertion order matters.
We can’t always control it.