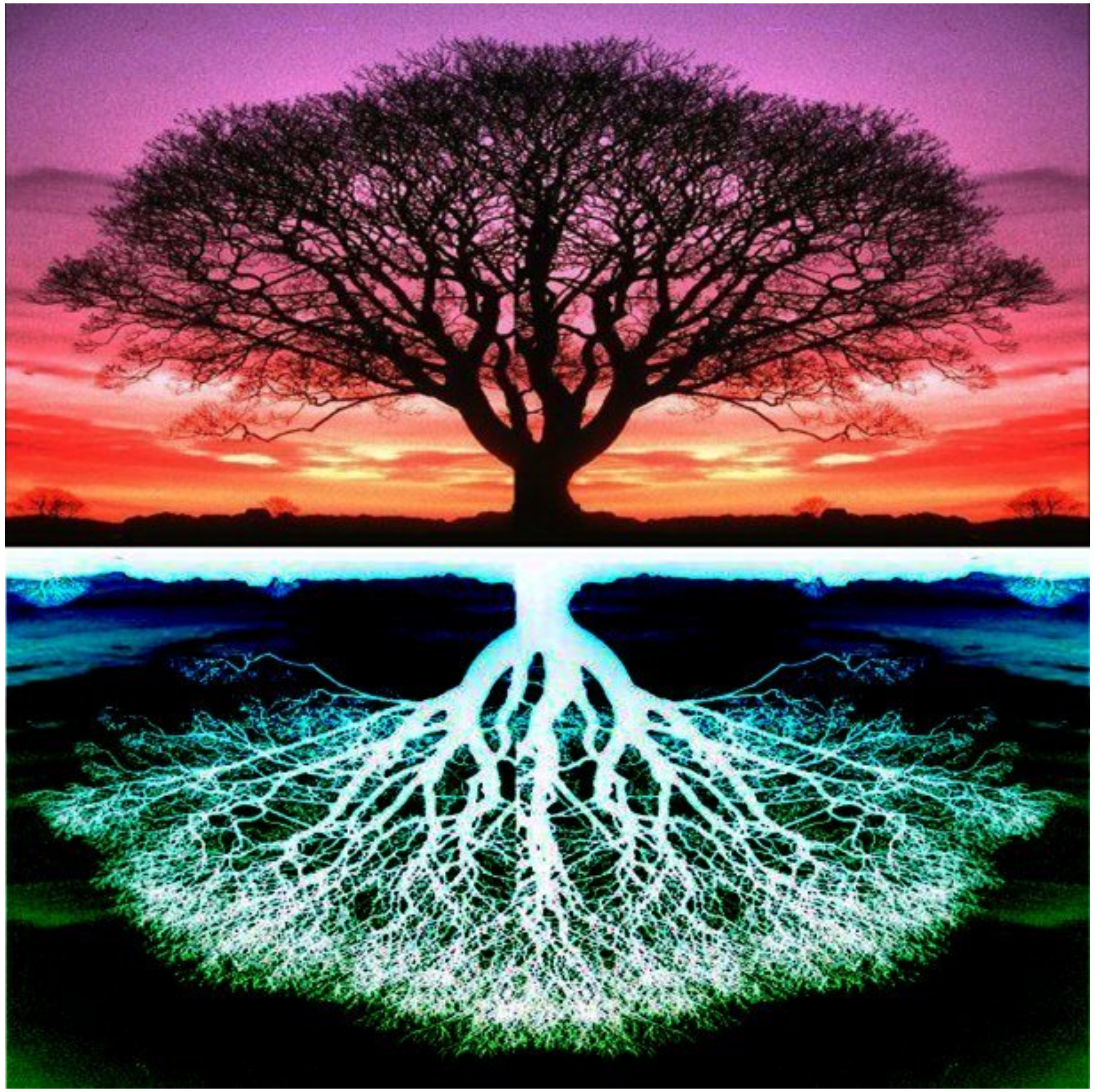


CSCI 241

Lecture 9
Binary Search Trees

Announcements

- Performance bug alert: in merge(), don't make a copy of the whole array if you're only merging part of it!
- Quiz grades will be released via GradeScope
- How's A1 going?

Goals (Today and Wednesday):

- 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
- Know what a balanced BST is and why we want it.

Tree Terminology: Lighting Round!

ABCD:

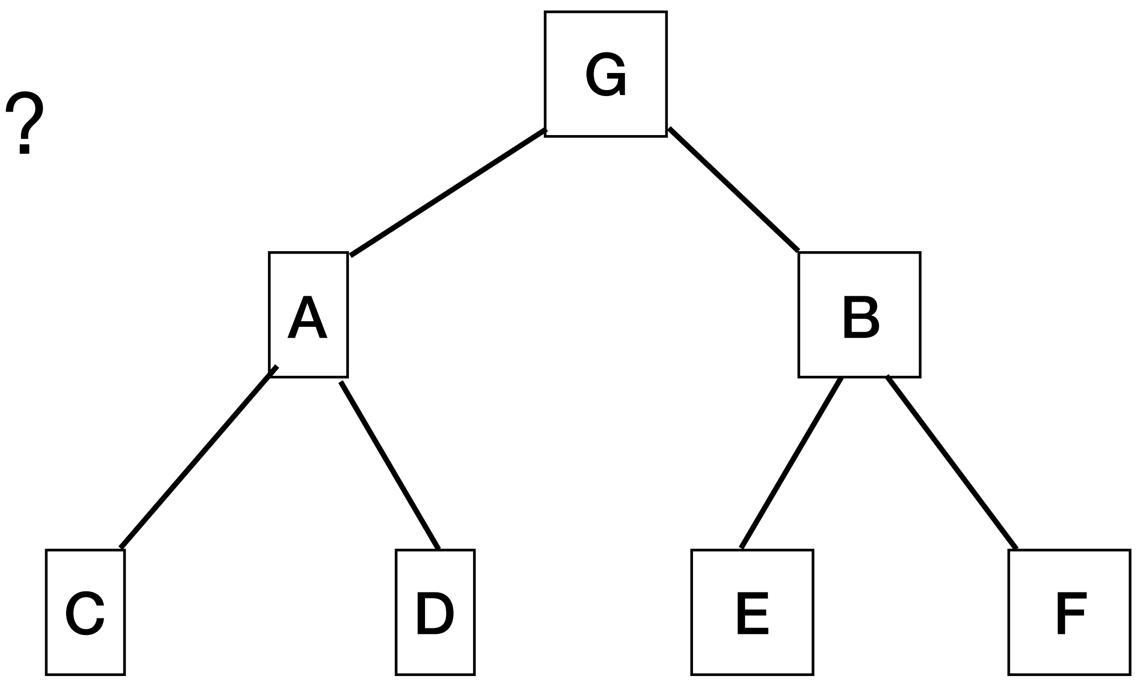
What’s the root of G’s right subtree?

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?



Binary Tree

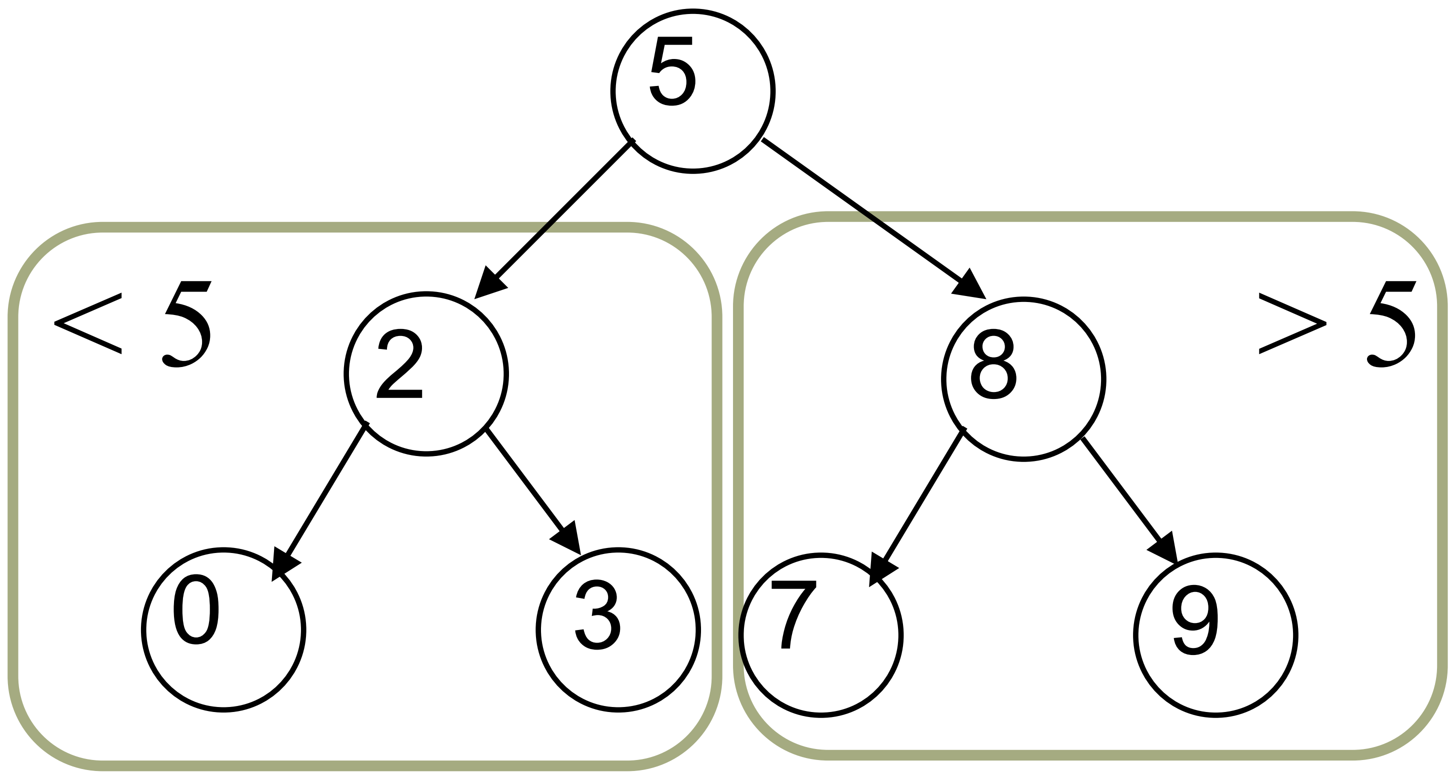
```
public class Tree {
    int value;
    Tree parent;
    Tree left;
    Tree right;
}
```

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;
}
```

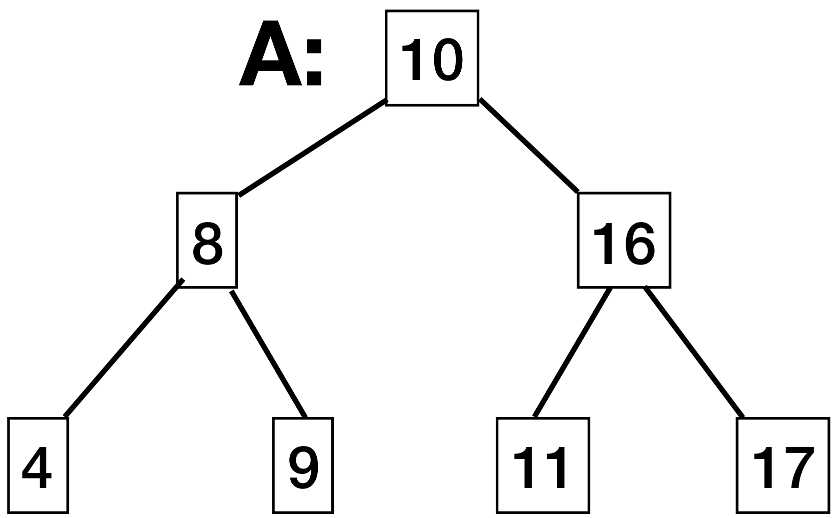
consequence: no duplicates!

Binary Search Tree

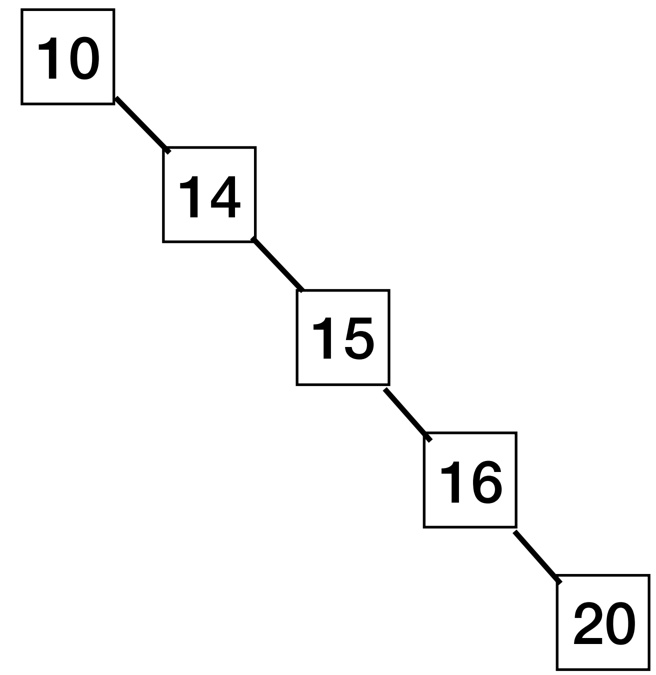


ABCD: Which of these is **not** a binary search tree?

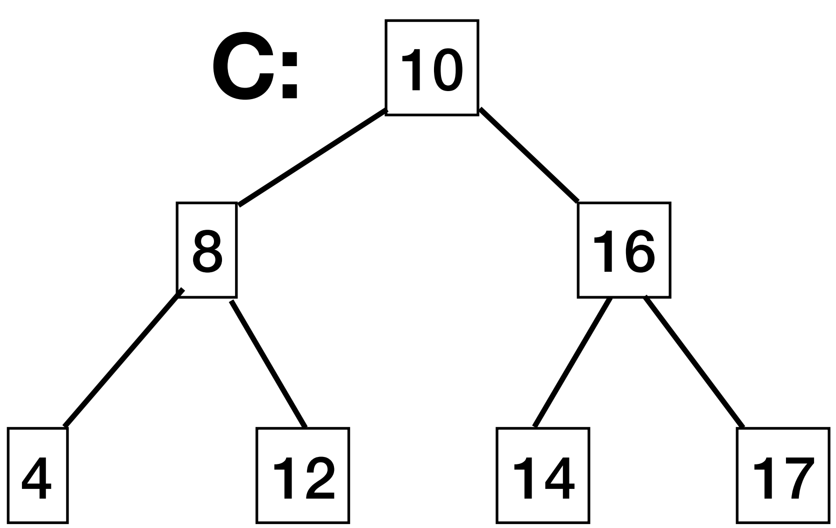
A:



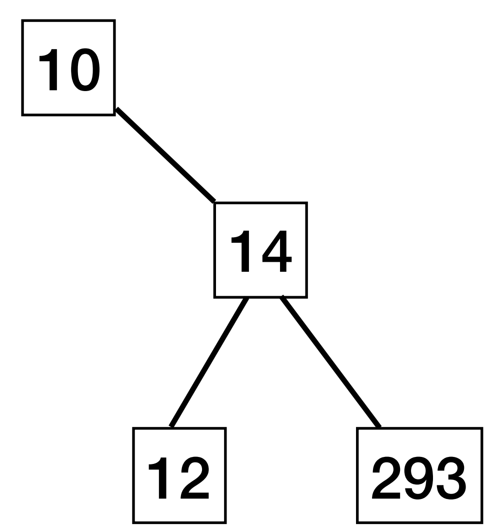
B:



C:



D:



Traversing a BST

pre-order traversal:

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

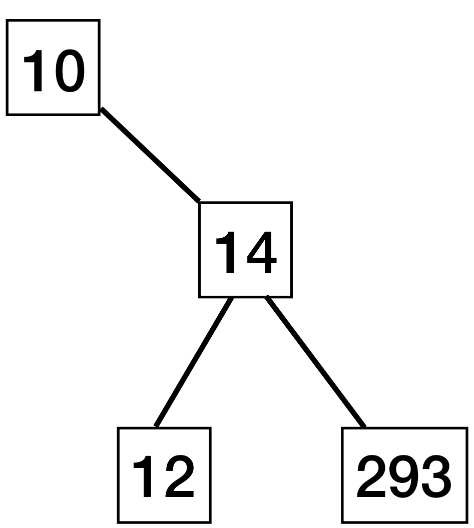
in-order traversal:

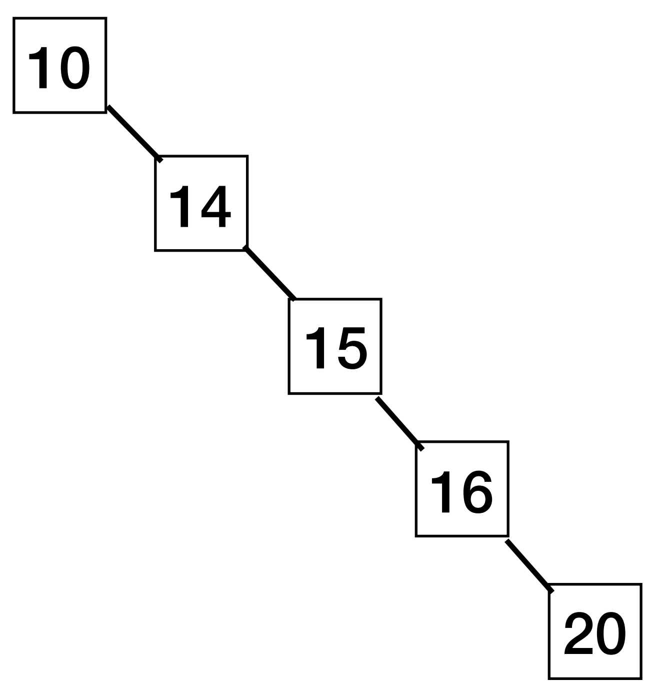
1. Process left subtree
2. Process root
3. Process right subtree

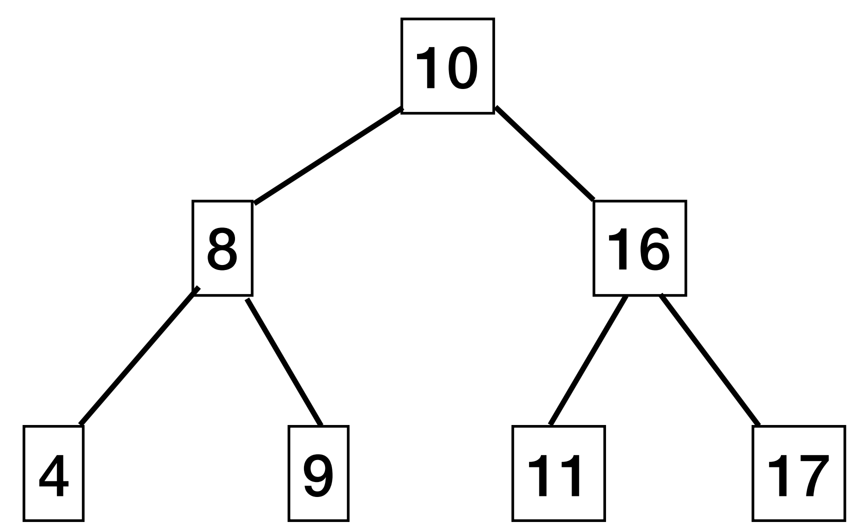
post-order traversal:

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

- 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?)

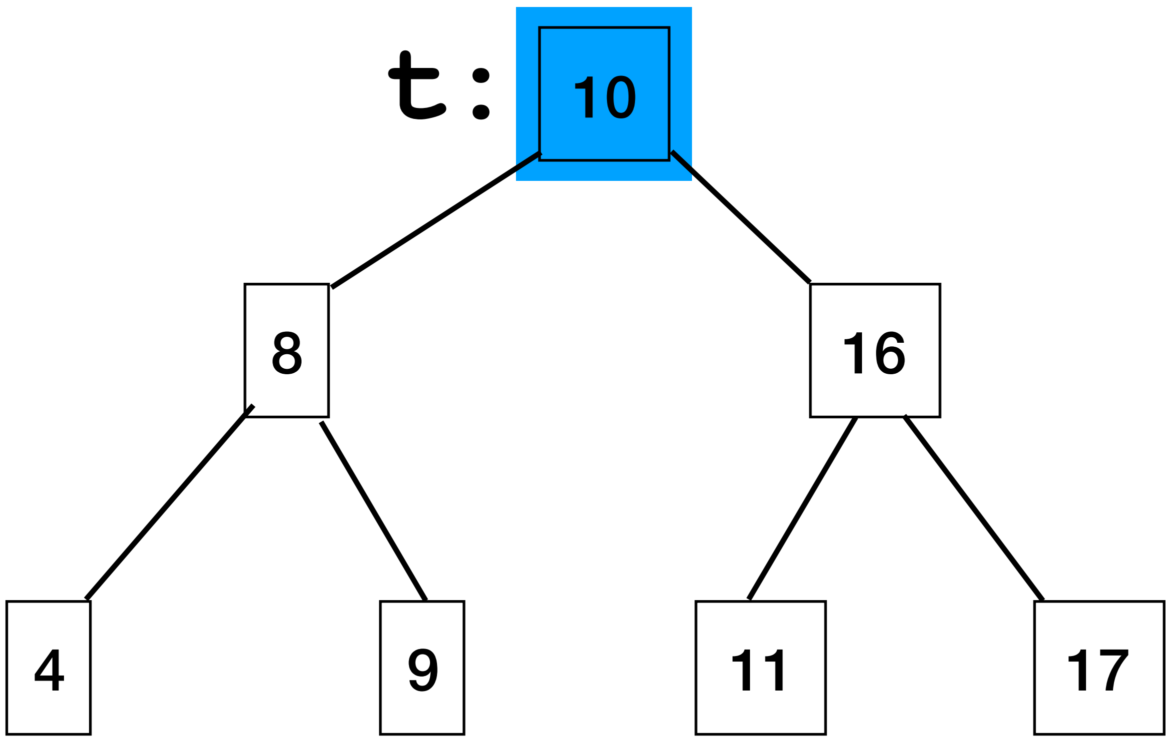
```
if t.value == v: return true
```

(recursive call - is v in left?)

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

(recursive call - is v in right?)

Searching a BST

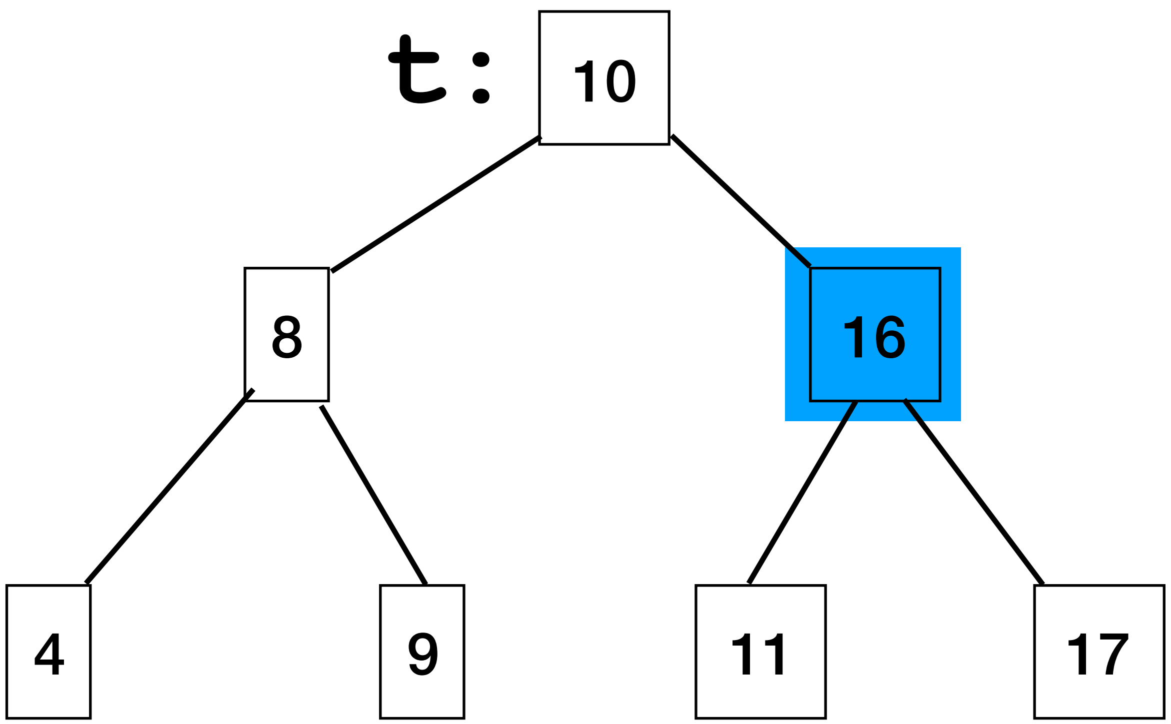


`search(t, 11)`

$11 > 10$

`search(right, 11)`

Searching a BST




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

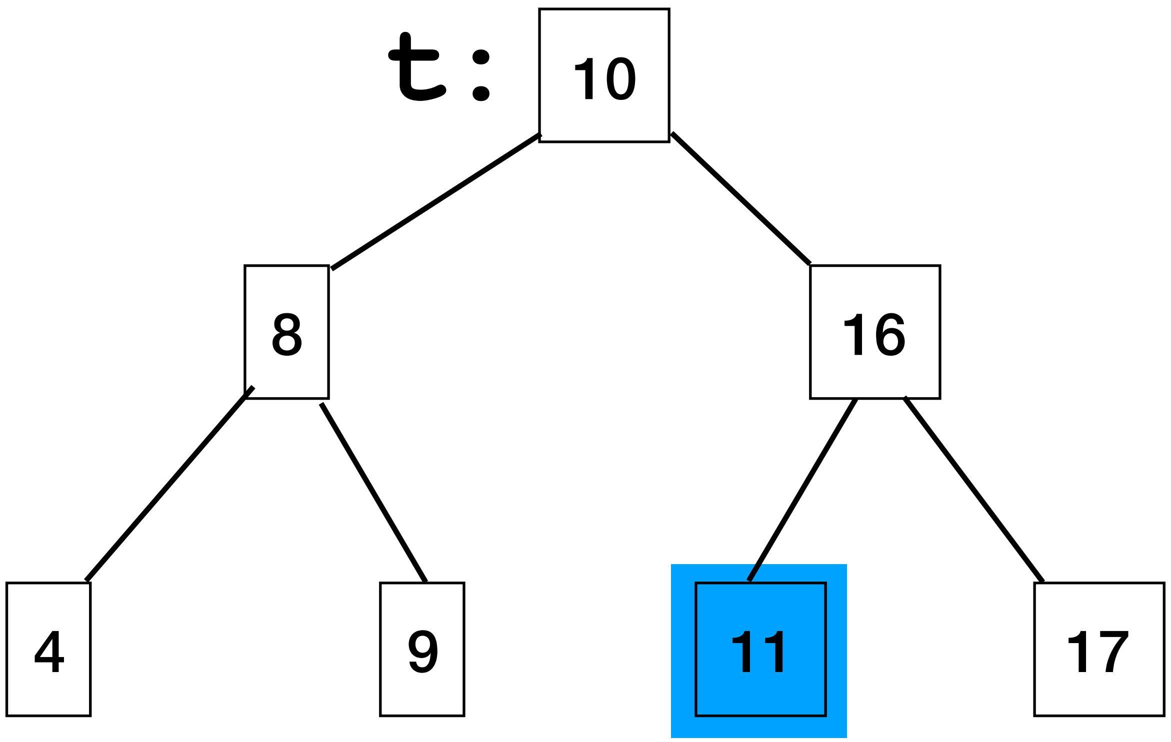
11 > 10

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

11 < 16

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

Searching a BST



search(t, 11)

11 > 10

search(right, 11)

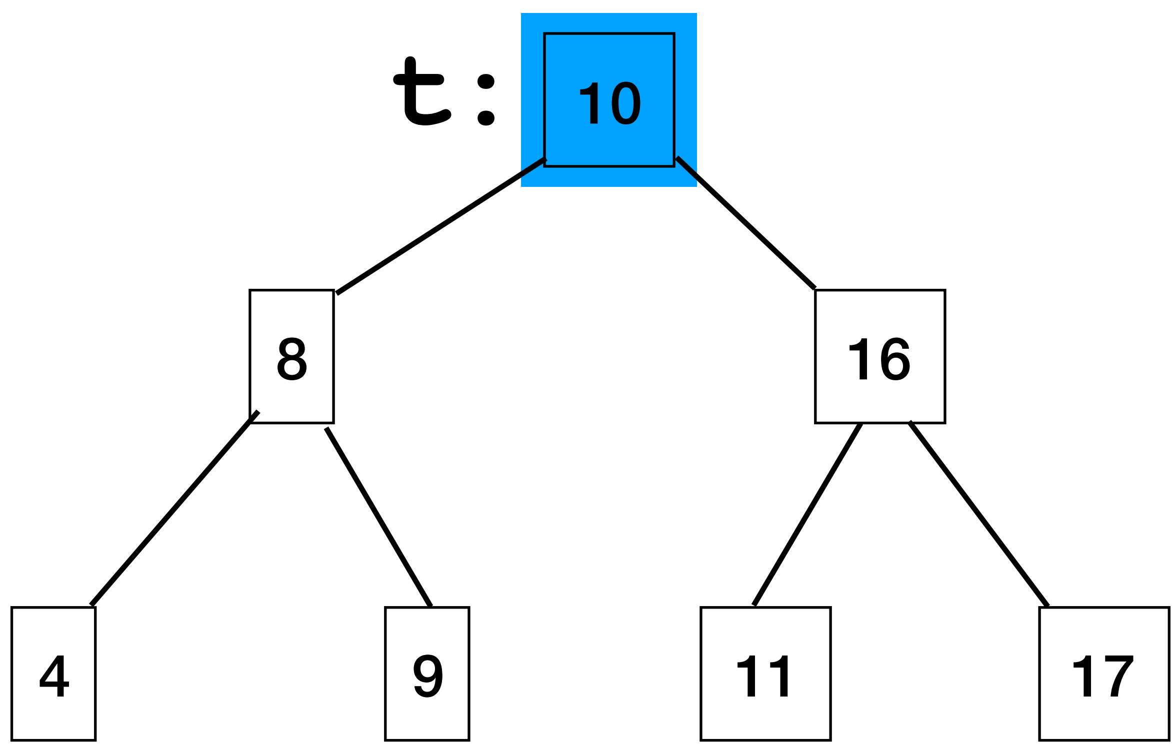
11 < 16

search(left, 11)

11 < 16

found it! return.

Searching a BST - the nonexistent case

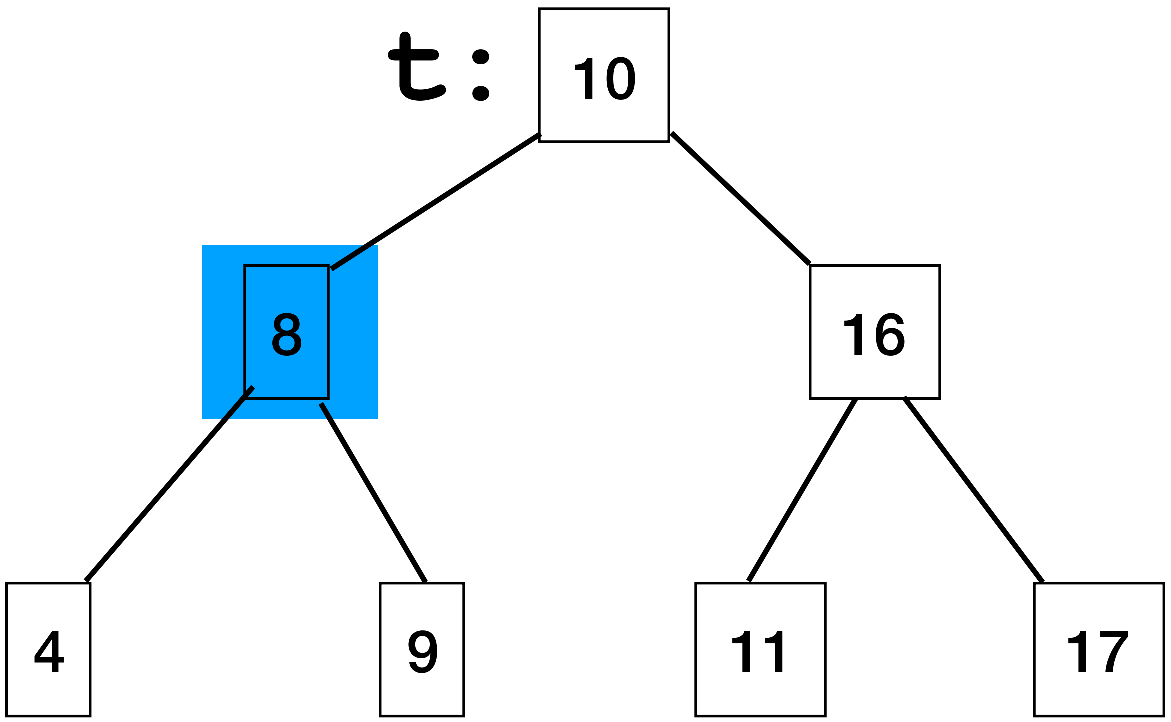


`search(t, 5)`

5 < 10

`search(left, 5)`

Searching a BST - the nonexistent case



`search(t, 5)`

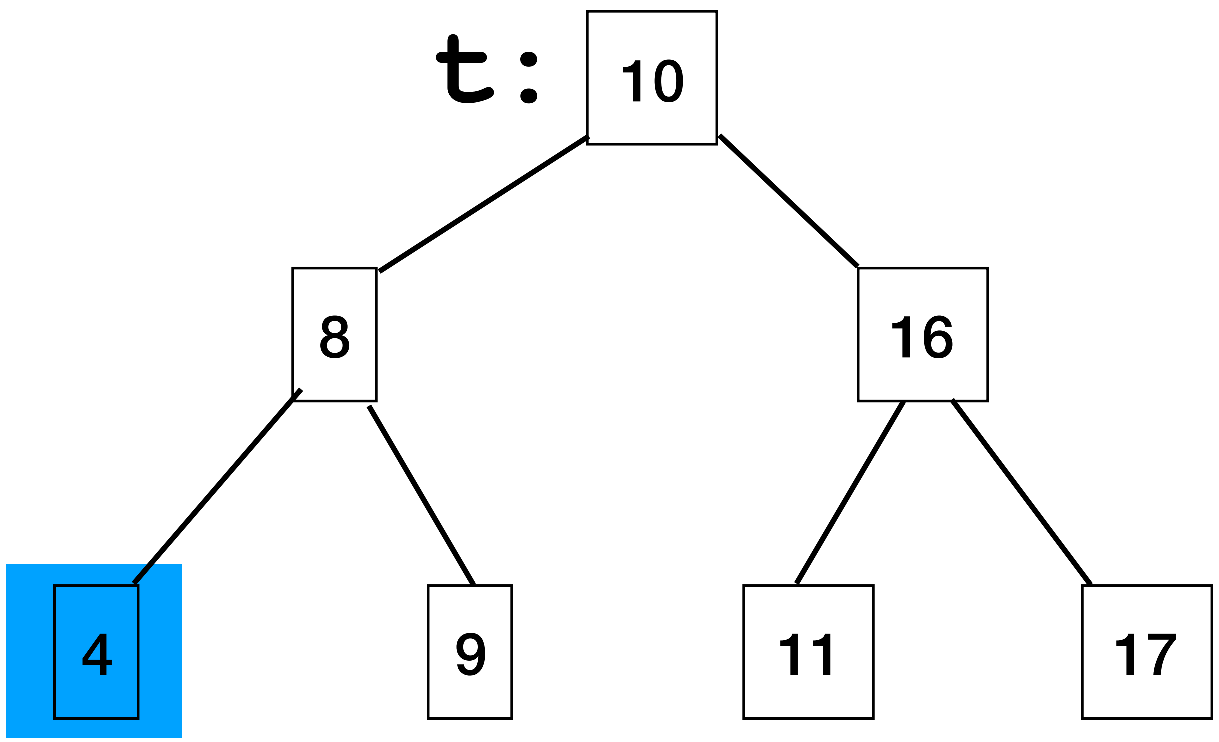
5 < 10

`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

Compare binary tree to binary search tree:

```
boolean searchBT(n, v):
  if n==null, return false
  if n.v == v, return true
  return searchBST(n.left, v)
      || searchBST(n.right, v)
```

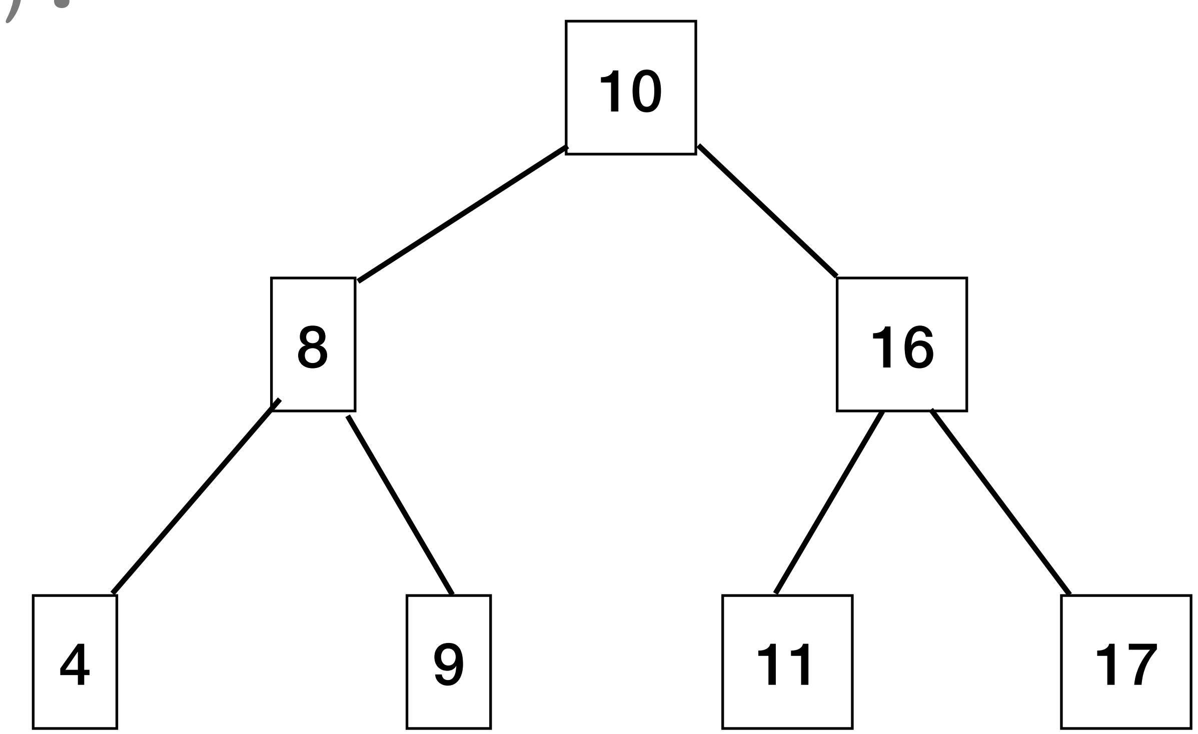
2 recursive calls

```
boolean searchBST(n, v):
  if n==null, return false
  if n.v == v, return true
  if v < n.v
    return searchBST(n.left, v)
  else
    return searchBST(n.right, v)
```

1 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)
```



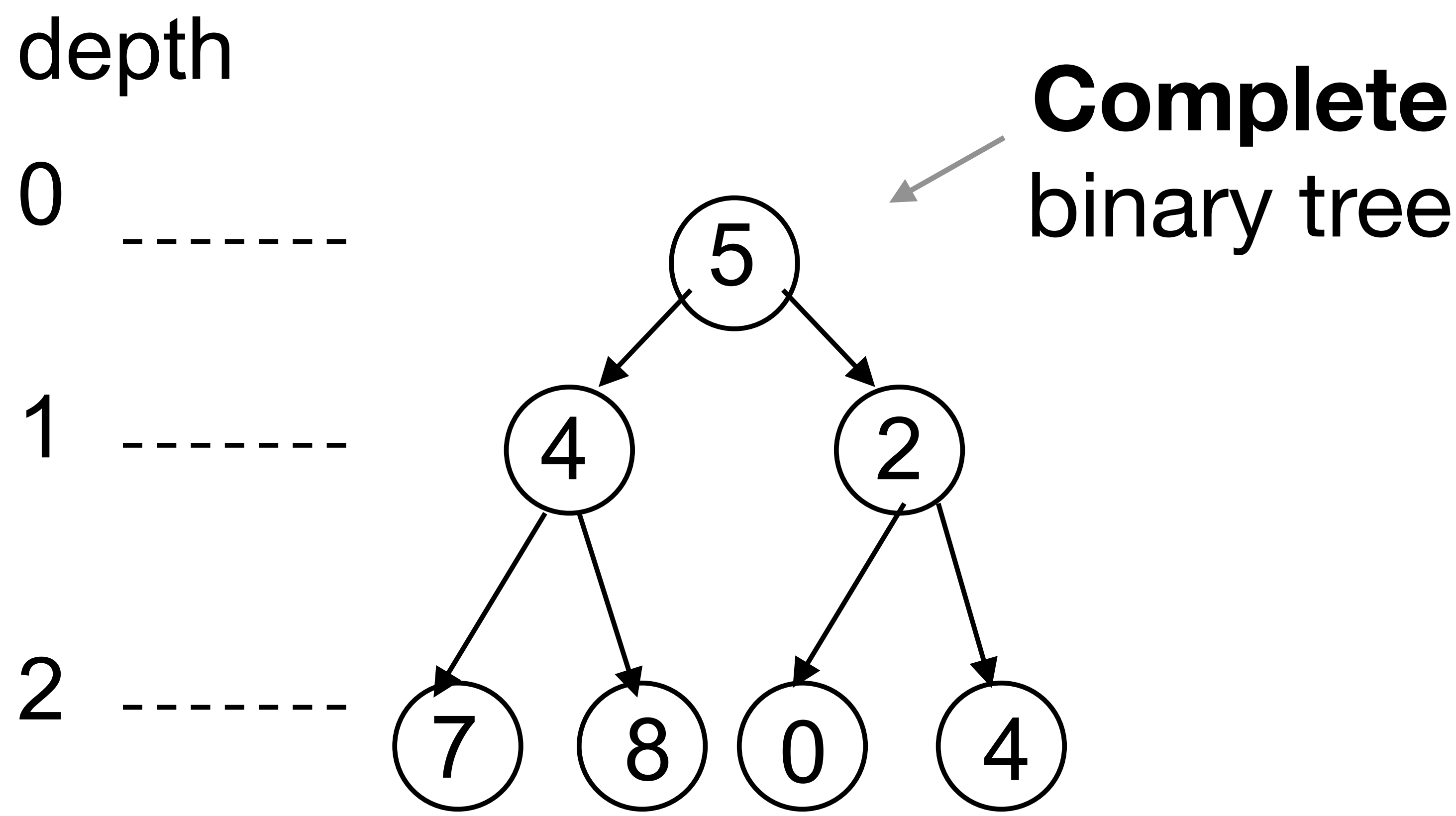
If h is the tree's **height**, search can visit at most h+1 nodes!

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 $h = 2$:



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: $2^0 + 2^1 + \ldots + 2^h$

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

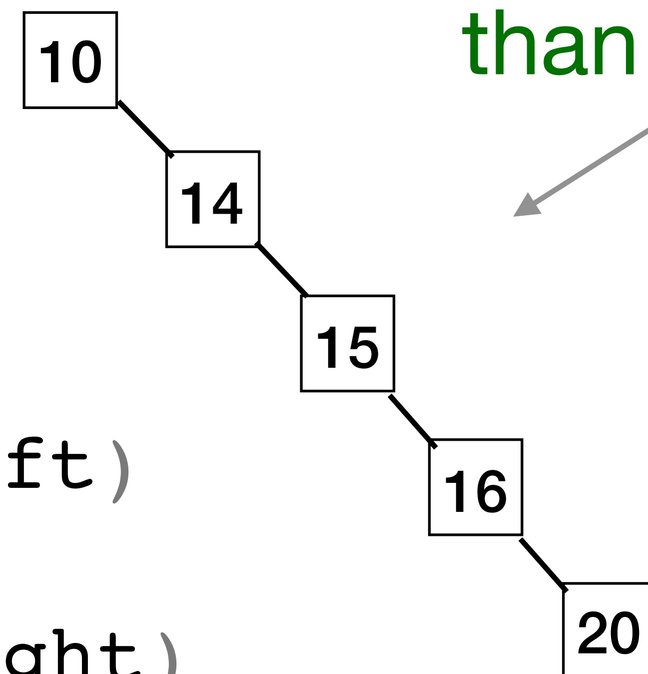
$n = 2^{h+1} - 1$

n is $O(2^h)$

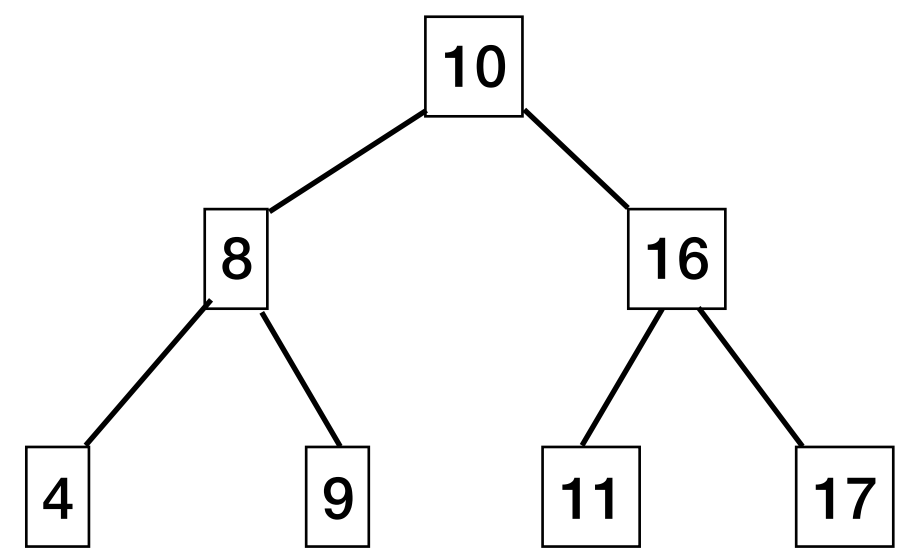
h is O(log n)

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)
```



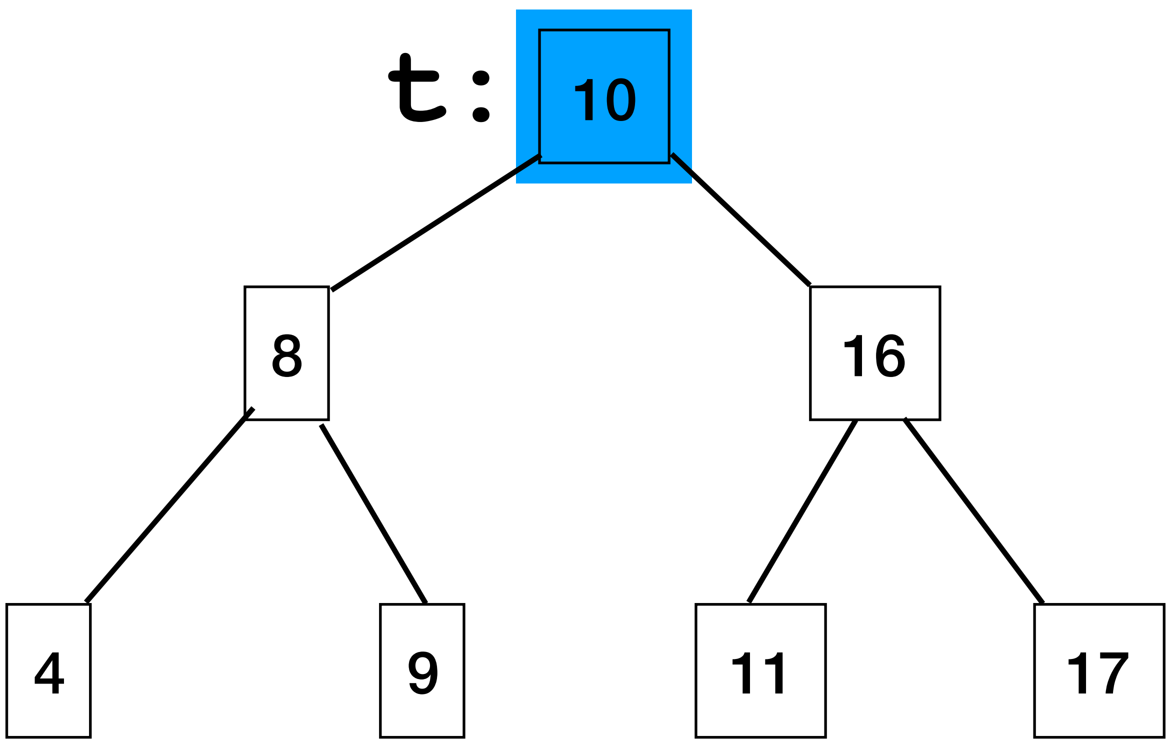
We want our trees to look more like this than this



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

Inserting into a BST

Inserting into a BST

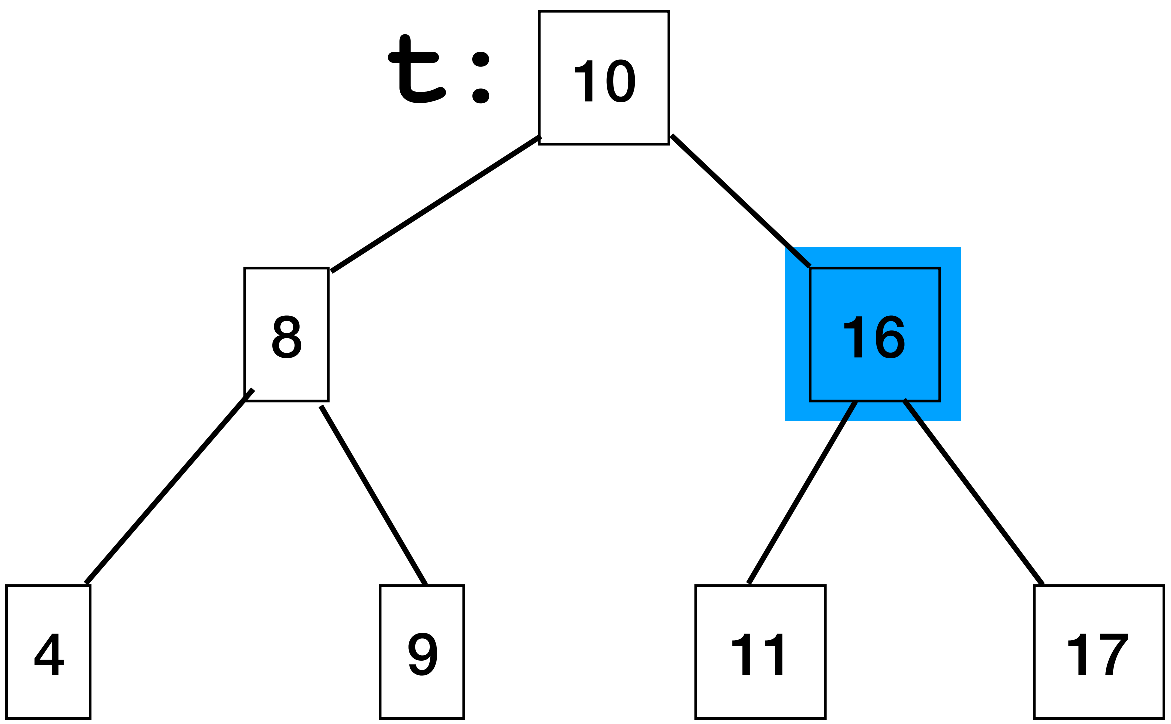


`insert(t, 11)`

$11 > 10$

`insert(right, 11)`

Inserting into a BST



`insert(t, 11)`

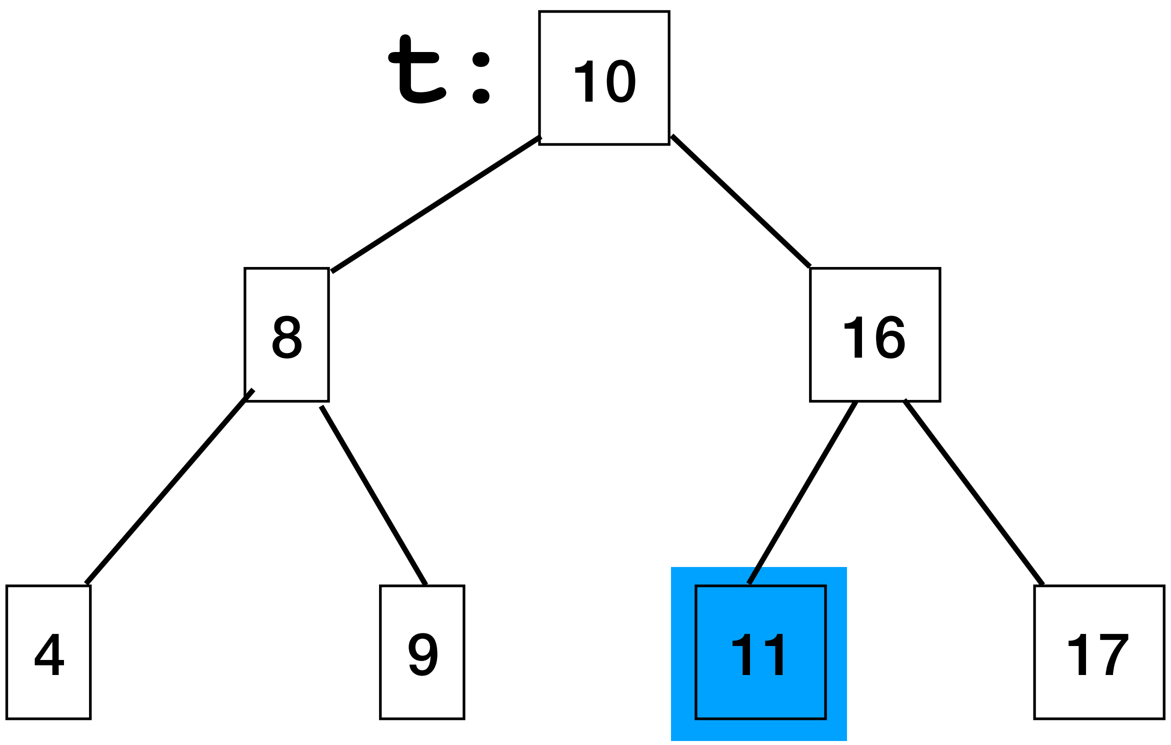
11 > 10

`insert(right, 11)`

11 < 16

`insert(left, 11)`

Inserting into a BST



`insert(t, 11)`

11 > 10

`insert(right, 11)`

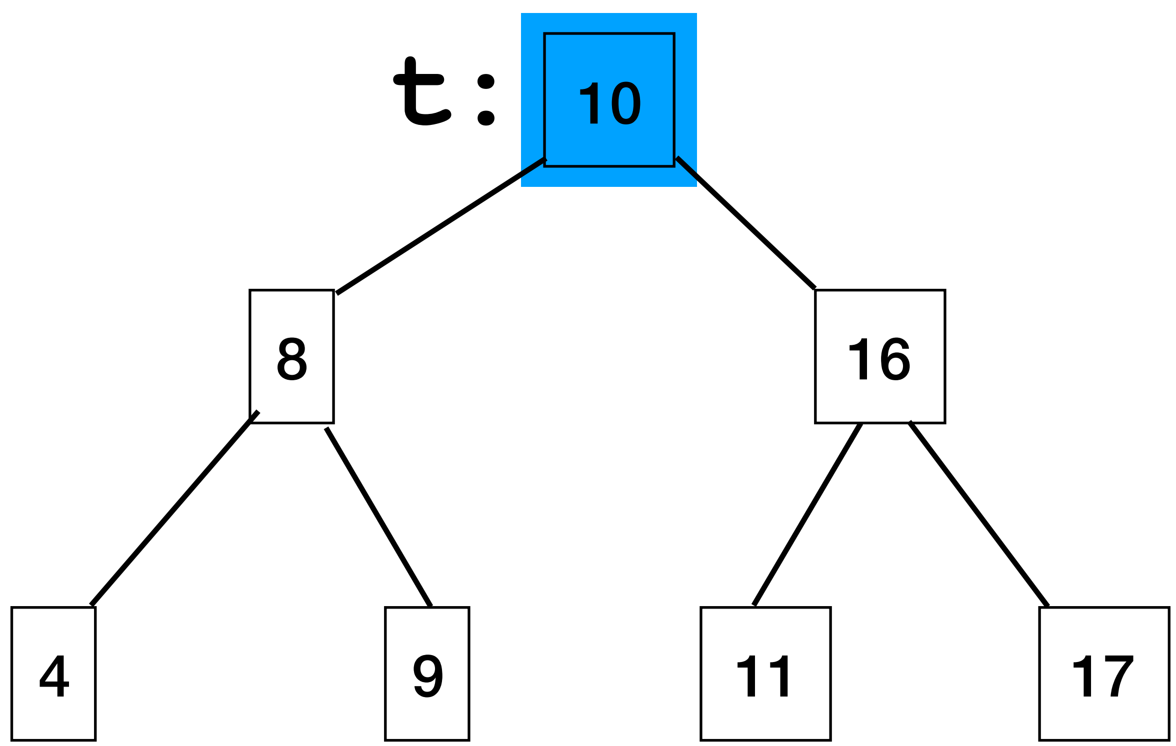
11 < 16

`insert(left, 11)`

11 < 16

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

Inserting into a BST - the nonexistent case

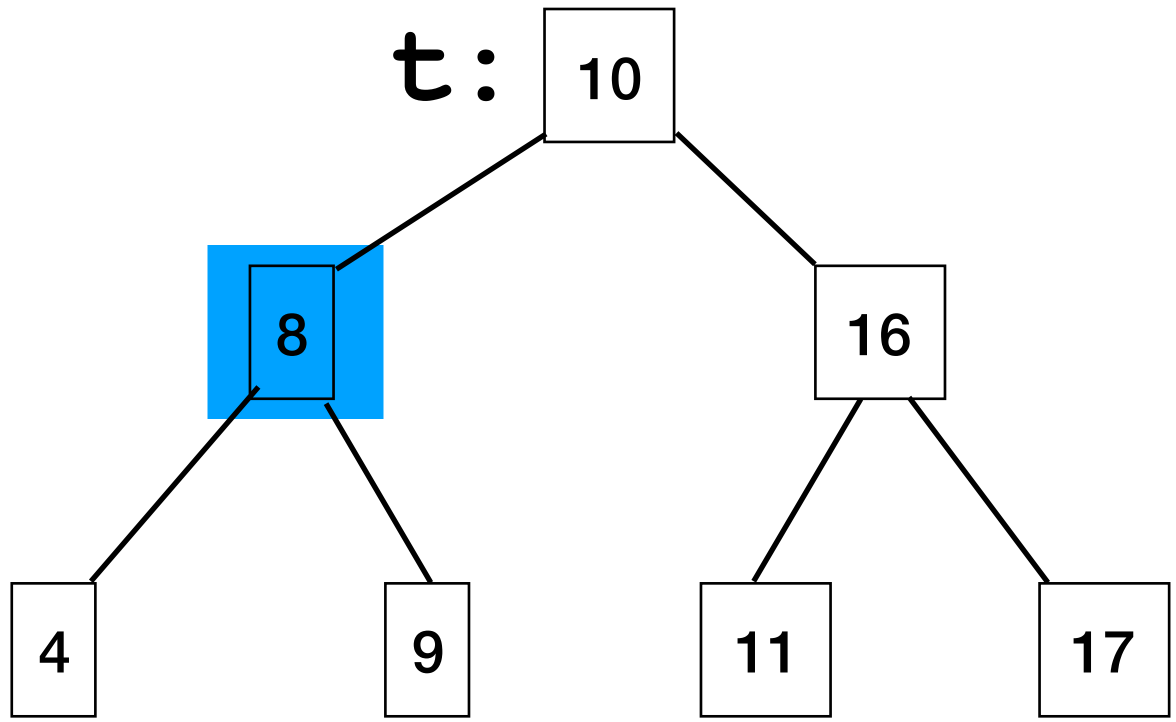



```
insert(t, 5)
```

5 < 10

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

Inserting into a BST - the nonexistent case



`insert(t, 5)`

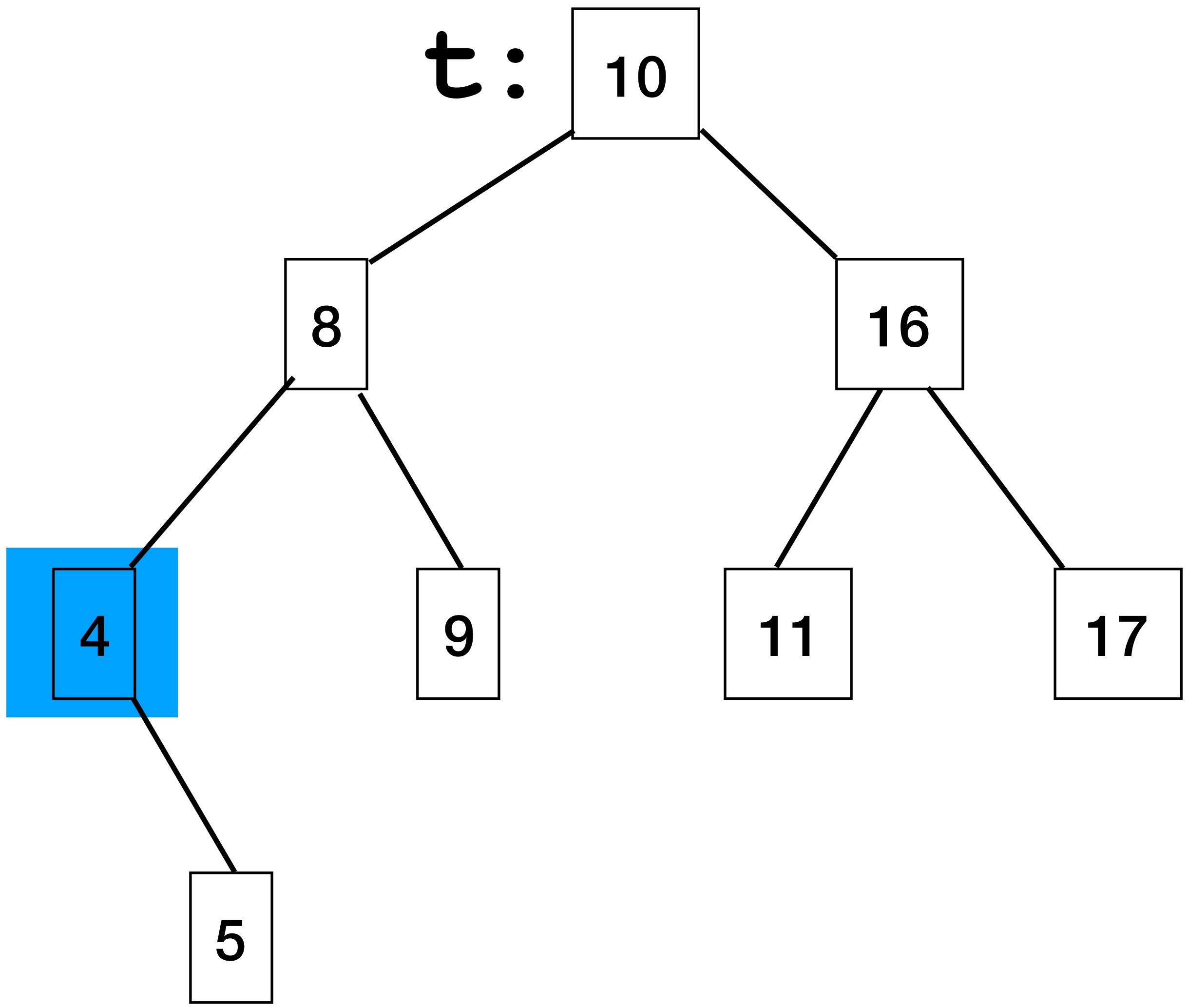
5 < 10

`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)`

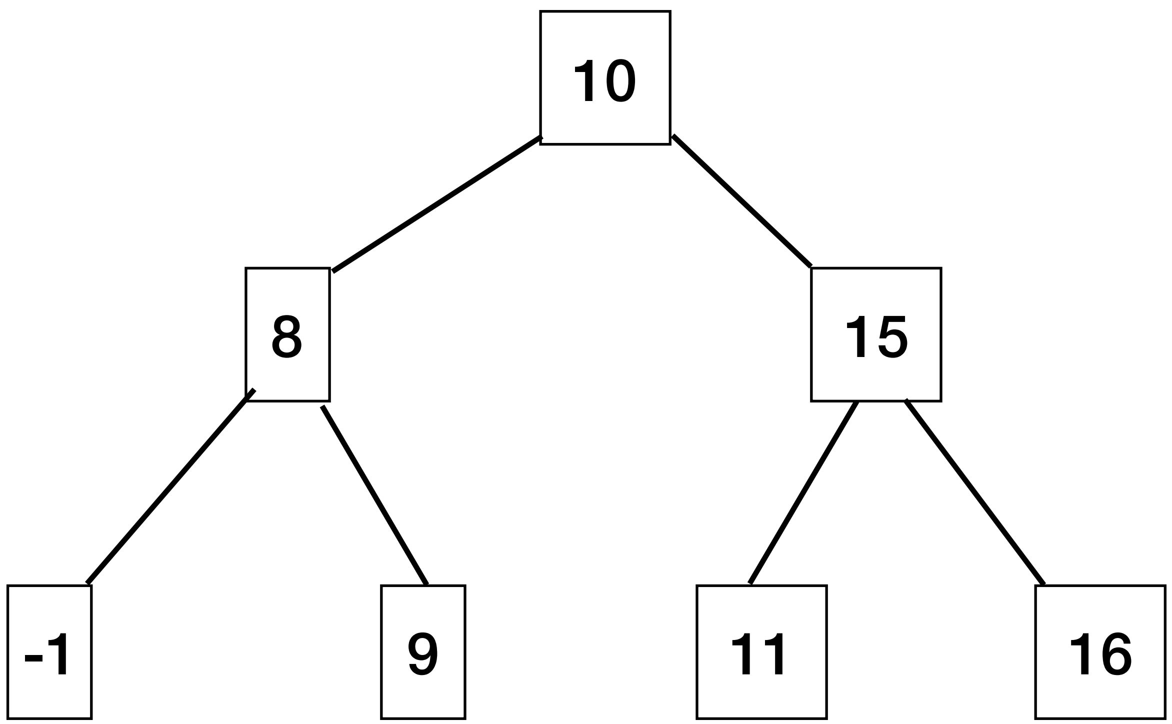
`null - not found. insert it here!`

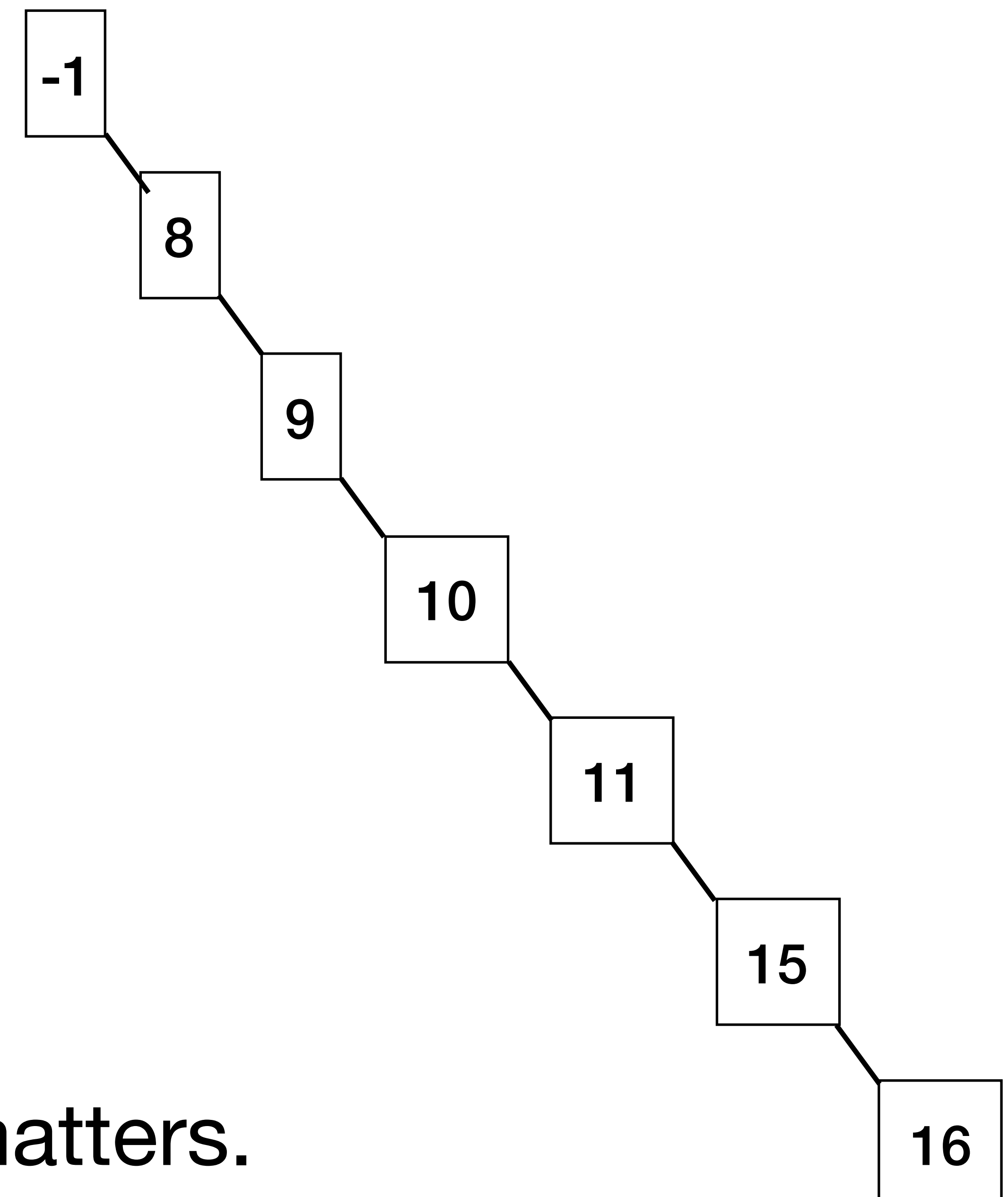
Let's Build Some Trees

```
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(11)
t.insert(10)
t.insert(15)
t.insert(16)
t.insert(16)
```

Let's Build Some Trees





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

Deleting a node from a BST

Three possible cases:

1. n has no children (is a leaf)
2. n has one child
3. n has two children