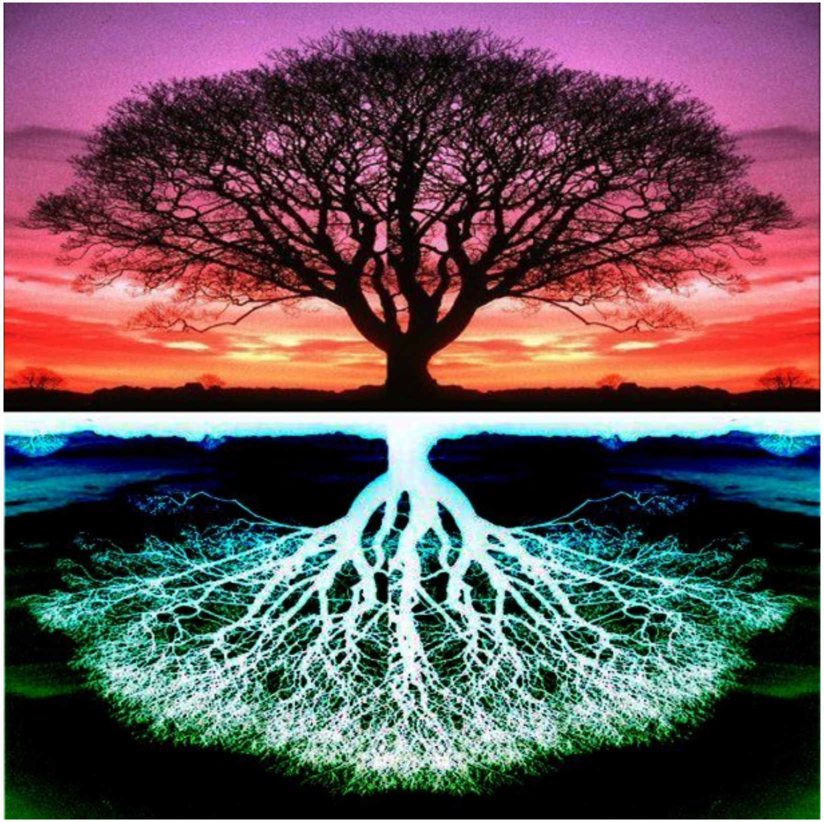


CSCI 241

Lecture 11

Binary Search Trees

please get logged into Socrative now!
socrative.com
room name: CSCI241

Announcements

Goals

- Know how to perform (and code) three tree traversals: pre-order, in-order, and post-order.
- 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

M is the **root** of this tree

N is the **left child** of P

S is the **right child** of P

P is the **parent** of N

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

B, H, J, N, S are **leaves**

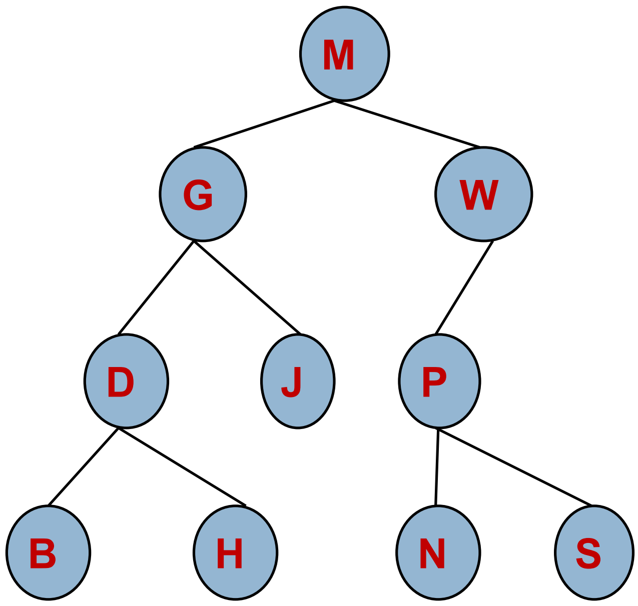
M and G are **ancestors** of D

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

J is at **depth** 2

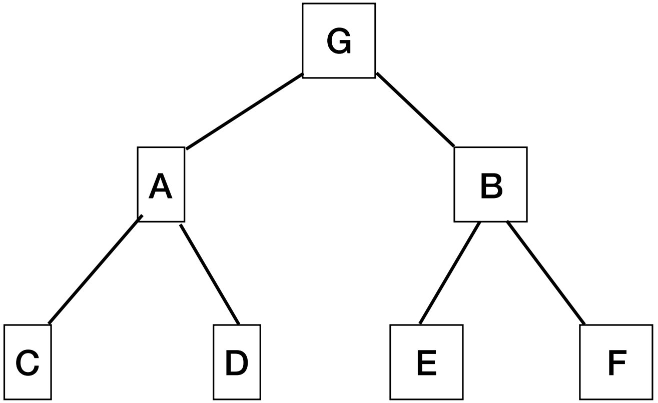
The subtree rooted at W has **height** 2

A collection of several trees is called a forest.



Tree Terminology: Lighting Round!

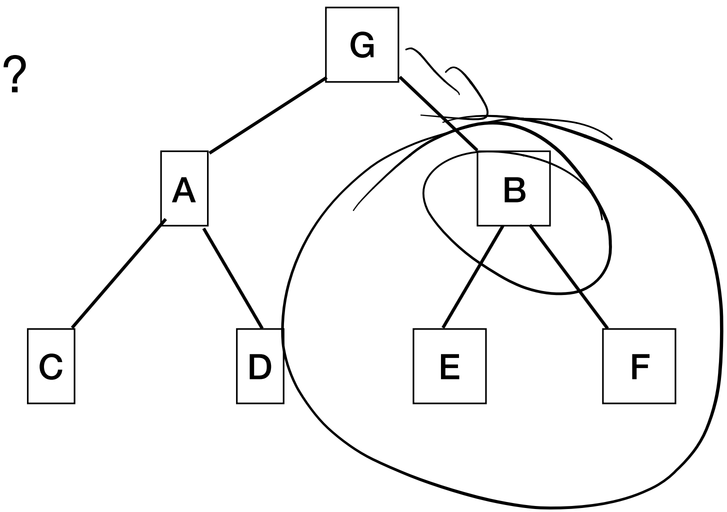
ABCD (name the node!):



Tree Terminology: Lighting Round!

ABCD (name the node!):

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

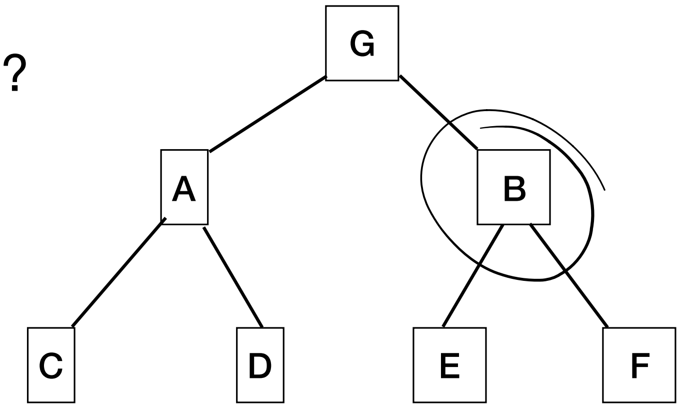


Tree Terminology: Lighting Round!

ABCD (name the node!):

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

What's an ancestor of F?



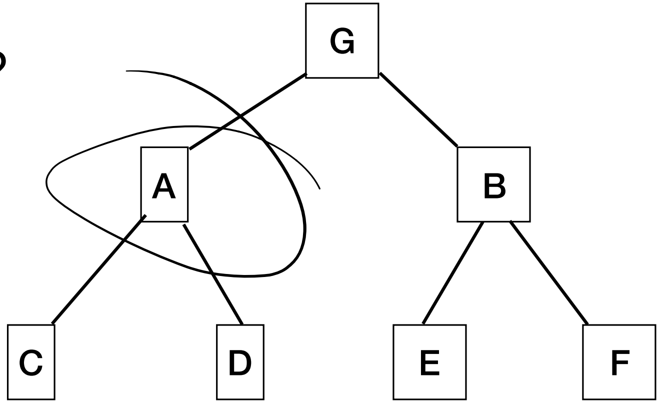
Tree Terminology: Lighting Round!

ABCD (name the node!):

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

What's an ancestor of F?

What's C's parent?



Tree Terminology: Lighting Round!

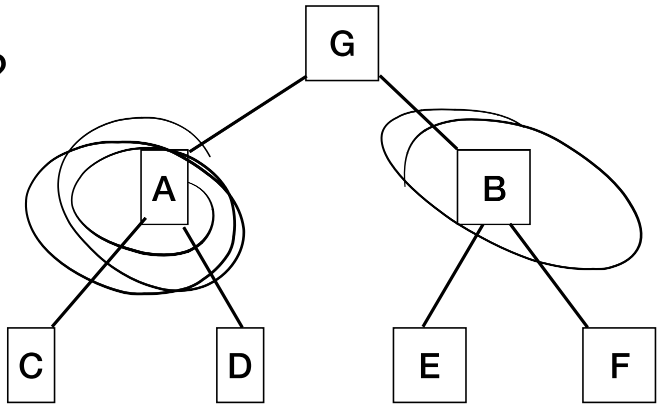
ABCD (name the node!):

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?



Tree Terminology: Lighting Round!

ABCD (name the node!):

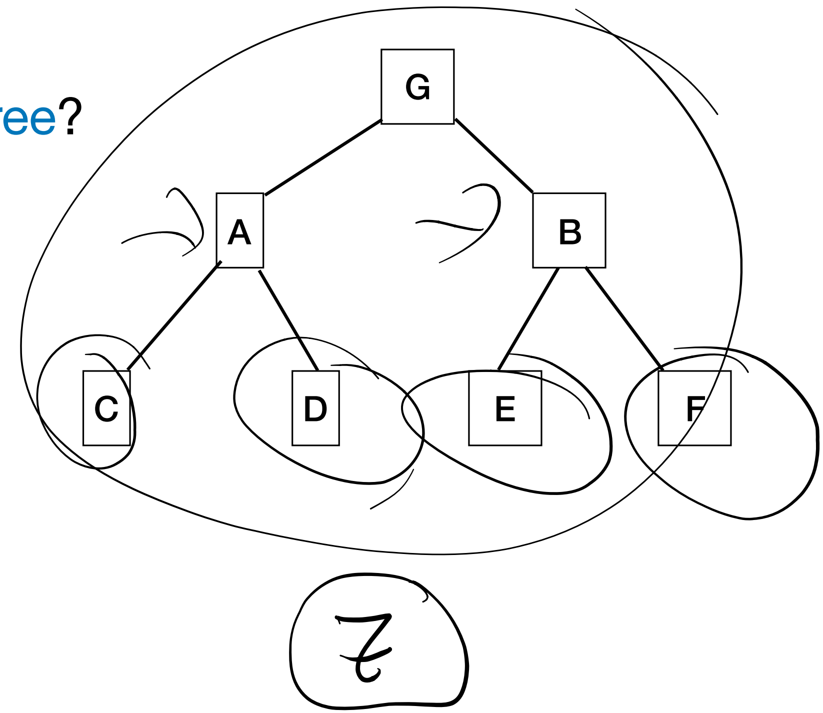
What's the root of G's right subtree?

What's an ancestor of F?

What's C's parent?

What's a node at depth 1?

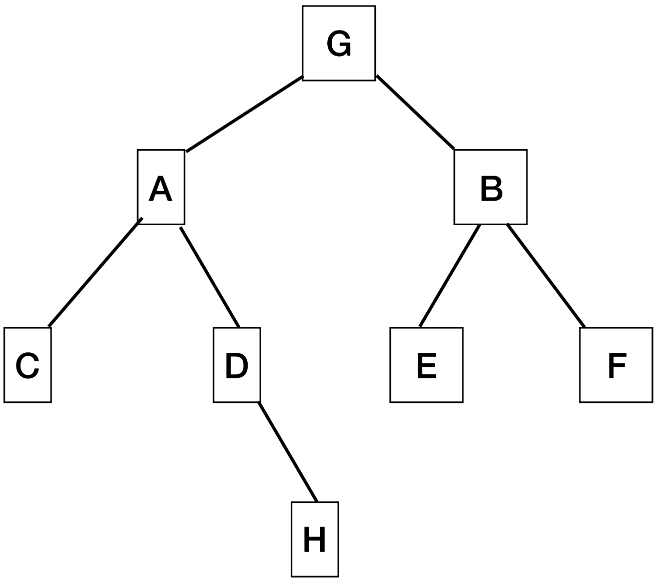
What's a node at the root of a subtree of height 0?



Tree Terminology: Lighting Round!

ABCD:

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

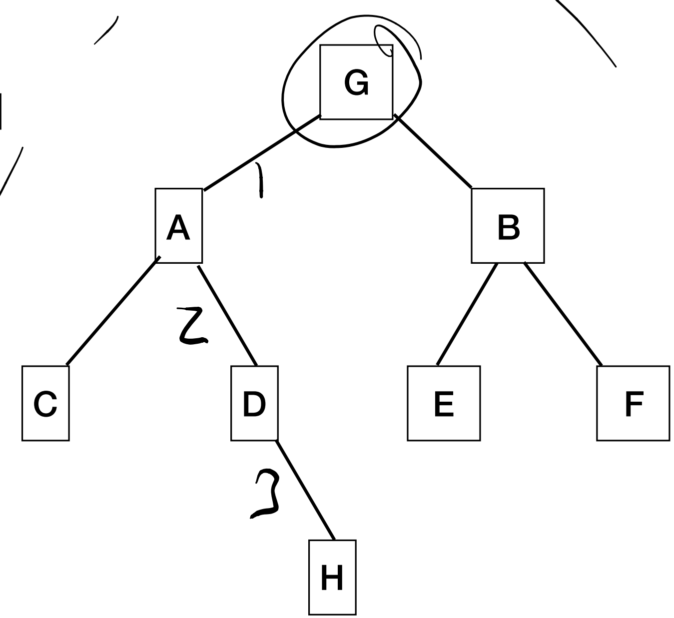


Tree Terminology: Lighting Round!

ABCD:

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

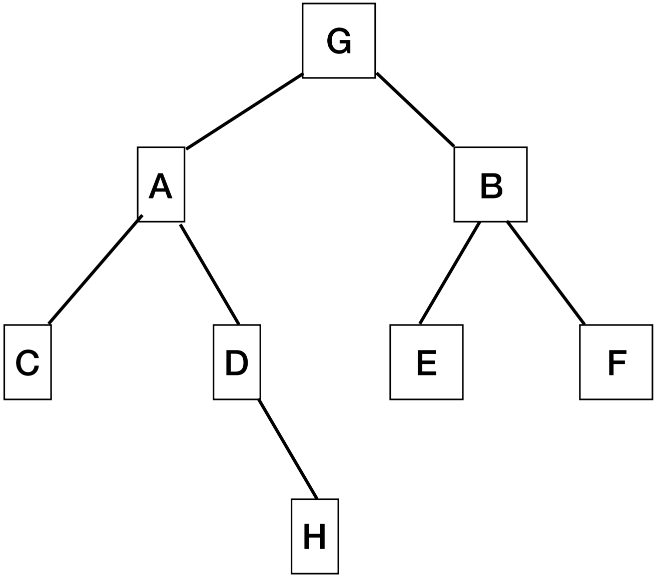
A. 1
B. 2
C. 3
D. 4



Tree Terminology: Lighting Round!

ABCD:

What's the depth of node D?

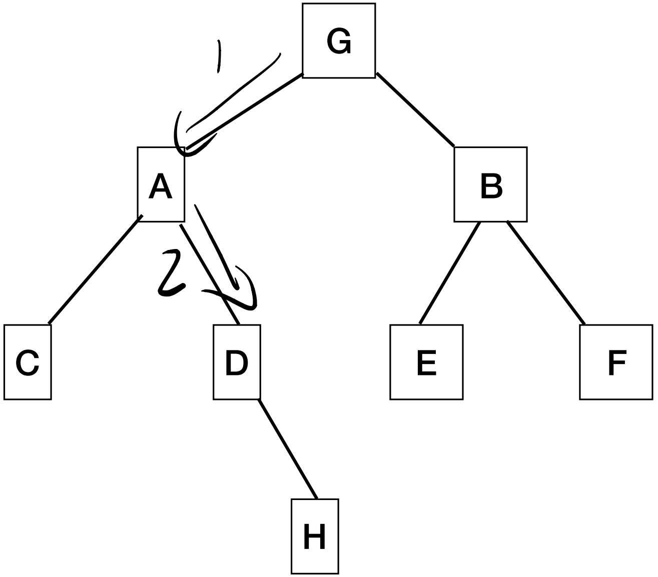


Tree Terminology: Lighting Round!

ABCD:

What's the depth of node D?

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



Tree Traversals

Print (or otherwise process) every node in a tree:

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

Tree Traversals

Print (or otherwise process) every node in a tree:

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

Print all nodes in a binary tree:

```
boolean printTree(Tree t):
```

(base case - nothing to print)

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

(print this node's value)

```
System.out.println(t.value)
```

(recursive call - print left subtree)

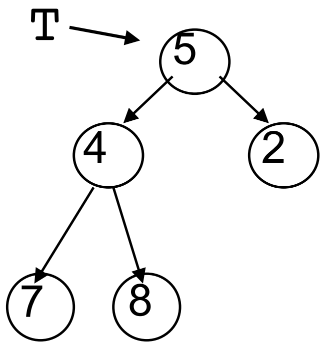
```
printTree(t.left)
```

(recursive call - print left subtree)

```
printTree(t.right)
```

Tree Traversals

Print (or otherwise process) every node in a tree:



Print all nodes in a binary tree:

```
boolean printTree(Tree t):
```

(base case - nothing to print)

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

(print this node's value)

```
System.out.println(t.value)
```

(recursive call - print left subtree)

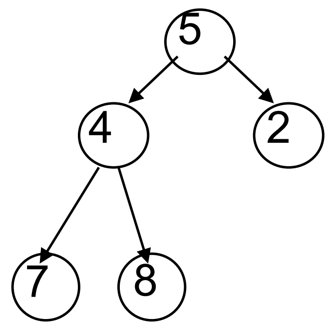
```
printTree(t.left)
```

(recursive call - print left subtree)

```
printTree(t.right)
```

Practice Exercise

- Write the values printed by a:
  - pre-order
  - in-order
  - post-order



traversal of this (or any other) binary tree.

Tree Traversals

"Walking" over the whole tree is called a tree traversal
This is done often enough that there are standard names.
Previous example was a **pre-order traversal**:

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

Other common traversals:

in-order traversal:

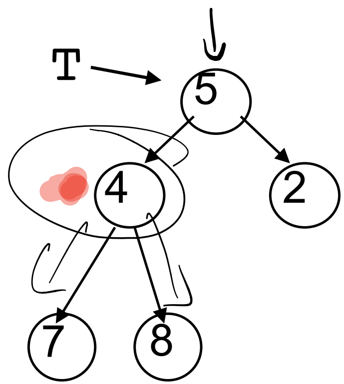
1. Process left subtree
2. Process root
3. Process right subtree

post-order traversal:

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

Tree Traversals

Print (or otherwise process) every node in a tree:



ABCD: `T` is a reference to the node with value 5. What is printed by the call `preorder(T)`?

A. 5 4 2 7 8
B. 7 4 8 5 2
C. 7 8 4 2 5
D. 5 4 7 8 2

Print all nodes in a binary tree:

```
boolean preorder(Tree t):
```

(base case - nothing to print)

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

(print this node's value)

```
System.out.println(t.value)
```

(recursive call - print left subtree)

```
printTree(t.left)
```

(recursive call - print left subtree)

```
printTree(t.right)
```

Binary Tree

```java
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;    < value
  BST right;   > value
}
```

Binary Search Tree

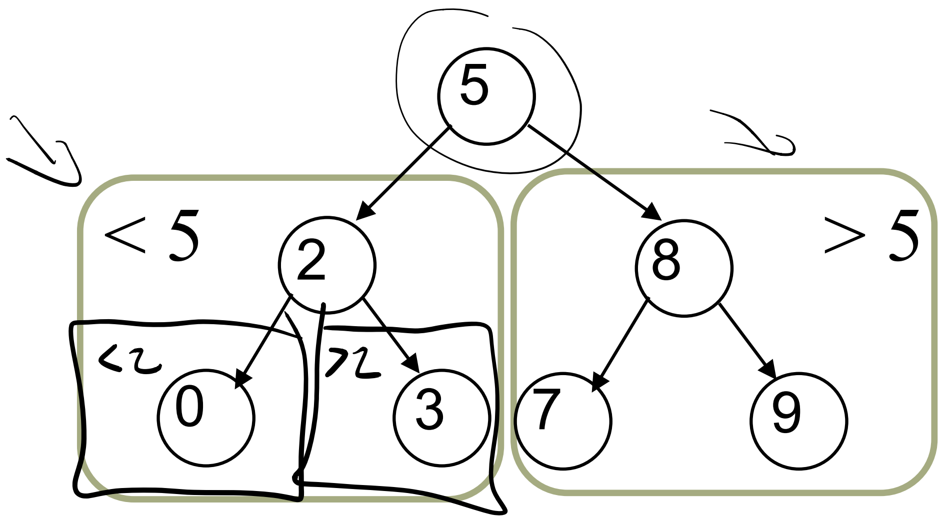
```
/** BST: a binary tree, in which:
 * -all values in left are < value
 * -all values in right are > value
 * -left and right are BSTs */
public class BST {
  int value;
  BST parent;
  BST left;
  BST right;
}
```

Binary Search Tree

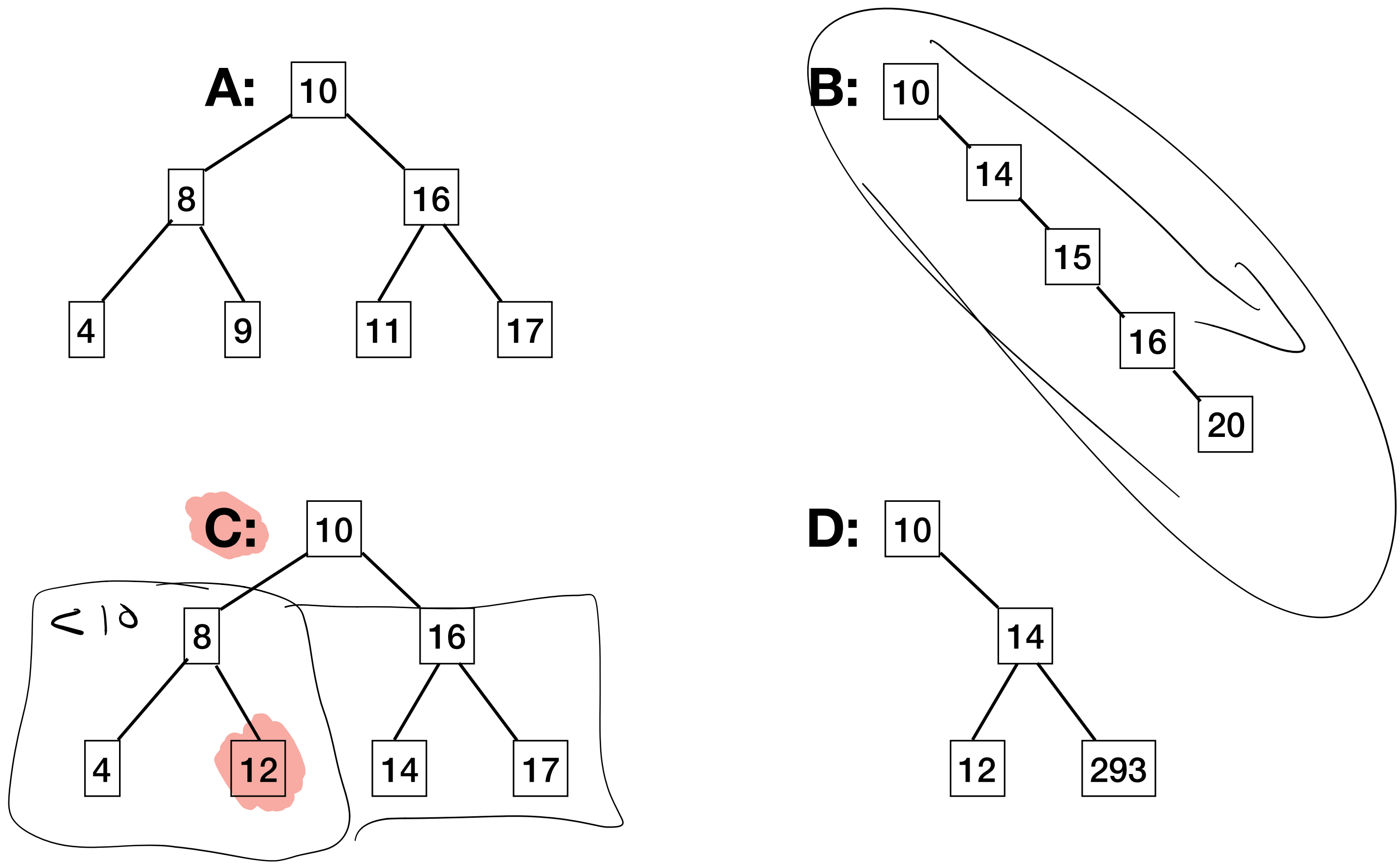
```
/** BST: a binary tree, in which:
 * -all values in left are < value
 * -all values in right are > value
 * -left and right are BSTs */
public class BST {
  int value;
  BST parent;
  BST left;
  BST right;
}
```

consequence: no duplicates!

Binary Search Tree



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



Traversing a BST

pre-order traversal:

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

in-order traversal:

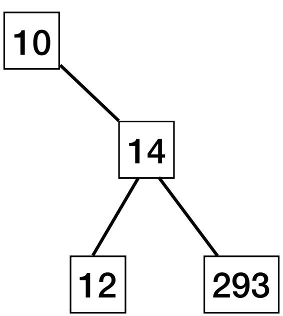
1. Process left subtree
2. Process root
3. Process right subtree

post-order traversal:

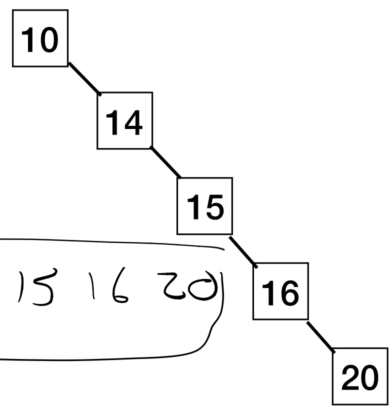
1. Process left subtree
2. Process right subtree
3. Process root

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

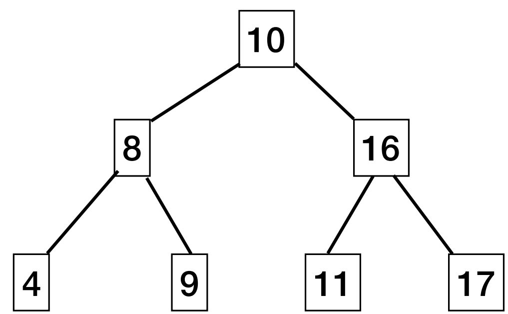
10 12 14 293



10 14 15 16 20



4 8 9 10 11 16 17



(not Search!)

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:

(base case - not found!)

(base case - is this v?)

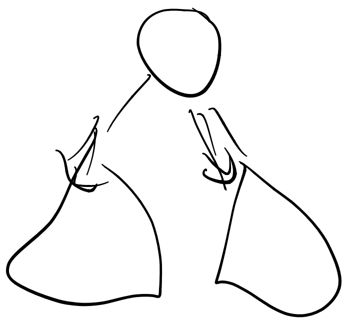
(recursive call - is v in left?)

(recursive call - is v in right?)

(not Search!)

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?)

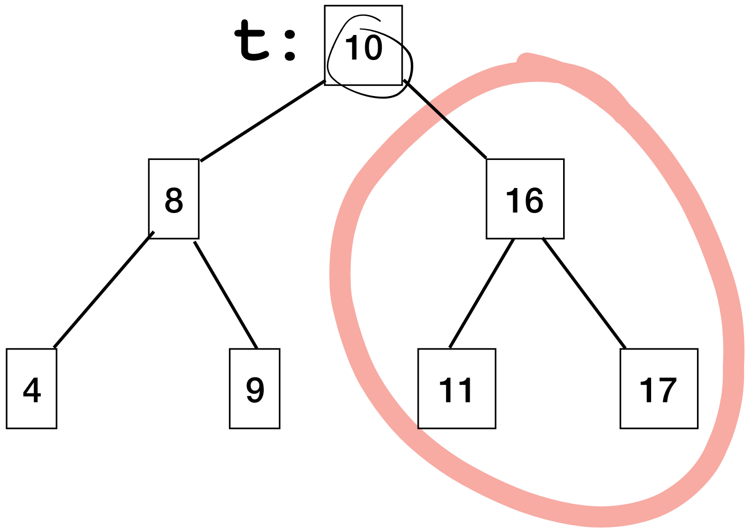
```
    if t.value == v: return true
```

(recursive call - is v in left?)

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

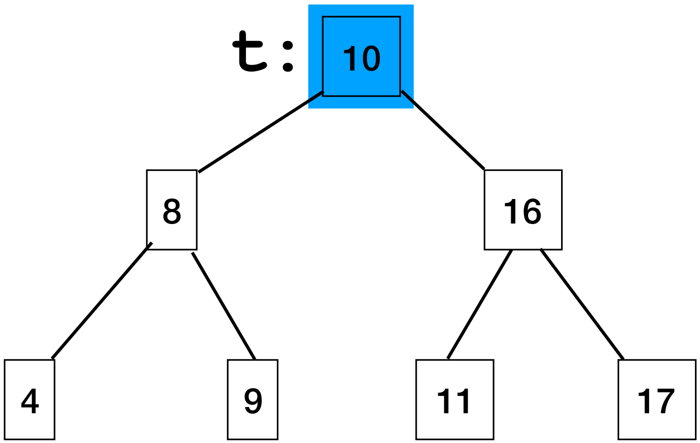
(recursive call - is v in right?)

Searching a BST



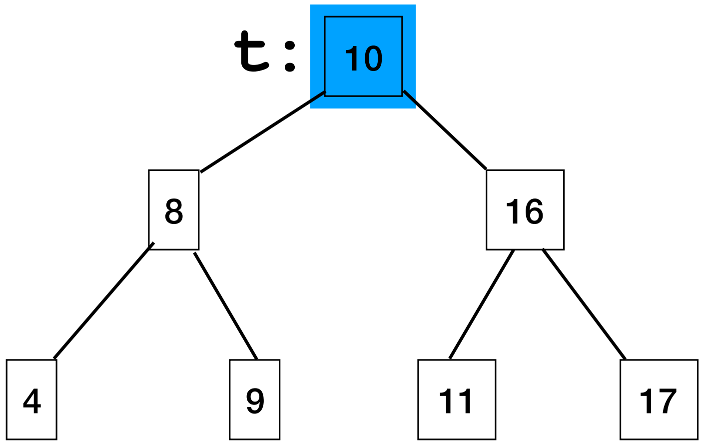
`search(t, 11)`

Searching a BST




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

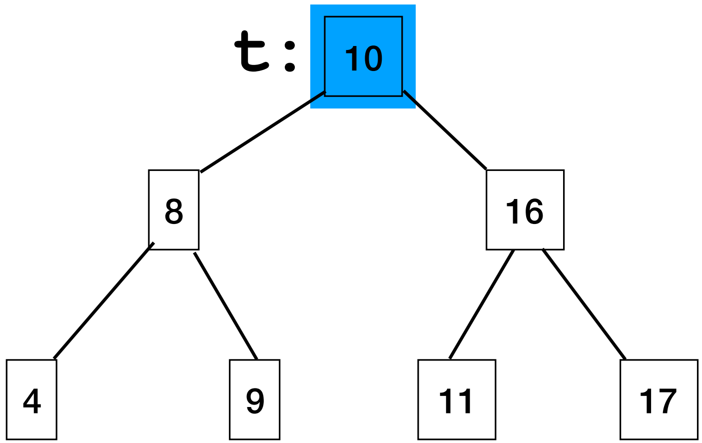
Searching a BST



`search(t, 11)`

11 > 10

Searching a BST

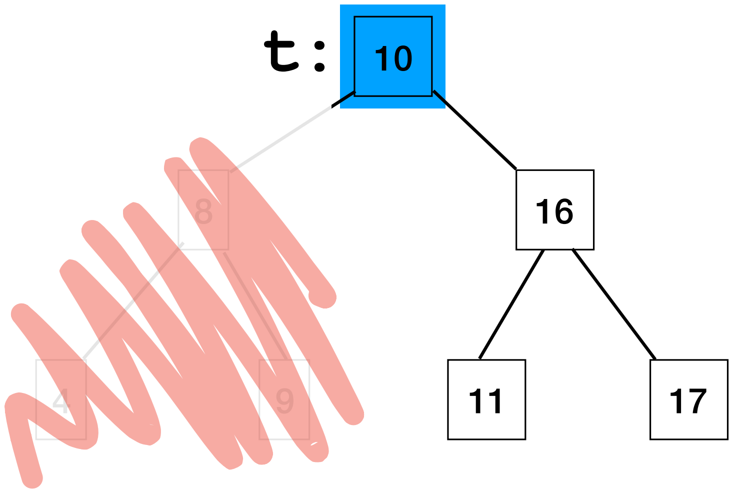



```
search(t, 11)
```

11 > 10

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

Searching a BST

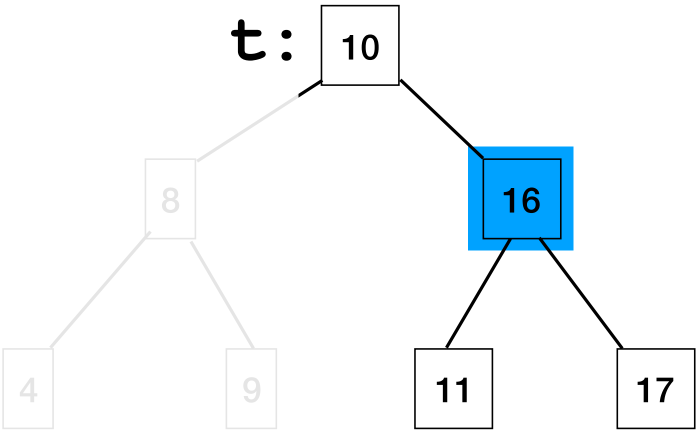


`search(t, 11)`

11 > 10

`search(right, 11)`

Searching a BST

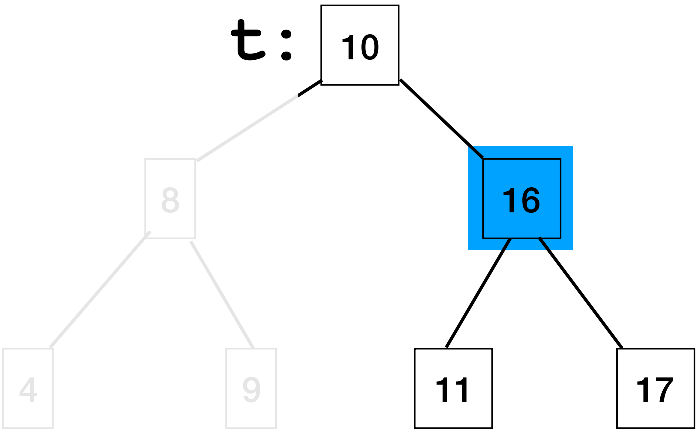



```
search(t, 11)
```

11 > 10

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

Searching a BST



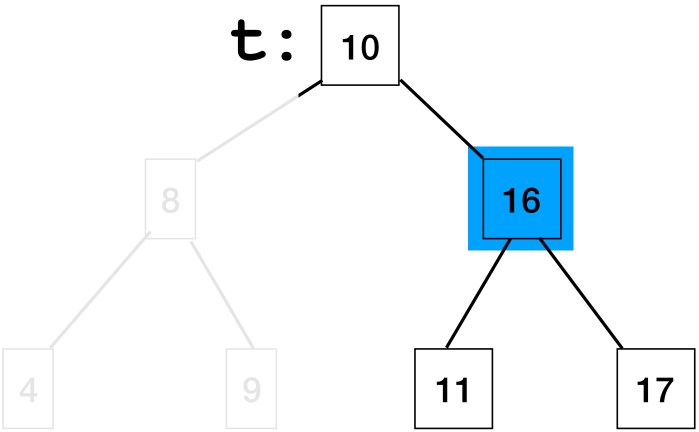
`search(t, 11)`

11 > 10

`search(right, 11)`

11 < 16

Searching a BST



`search(t, 11)`

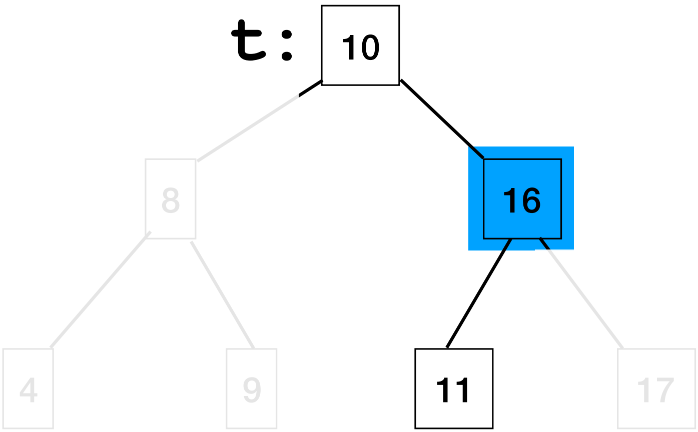
11 > 10

`search(right, 11)`

11 < 16

`search(left, 11)`

Searching a BST



`search(t, 11)`

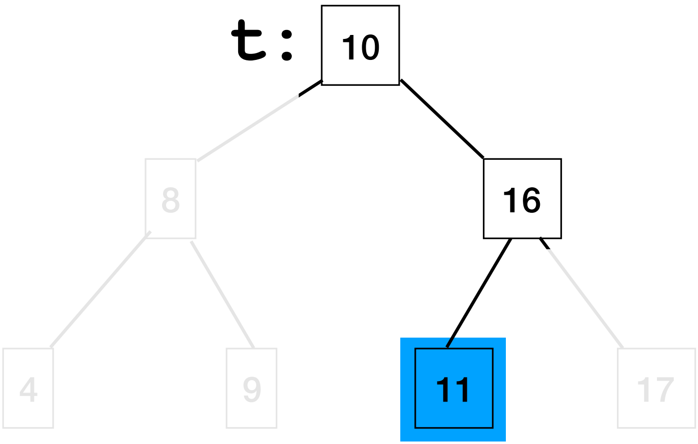
11 > 10

`search(right, 11)`

11 < 16

`search(left, 11)`

Searching a BST



`search(t, 11)`

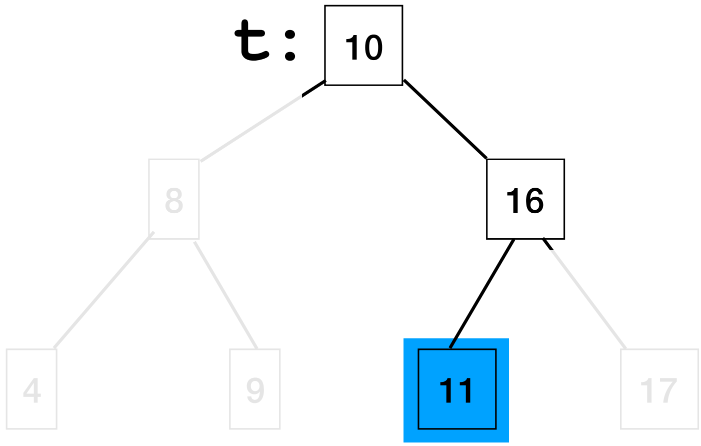
11 > 10

`search(right, 11)`

11 < 16

`search(left, 11)`

Searching a BST



`search(t, 11)`

11 > 10

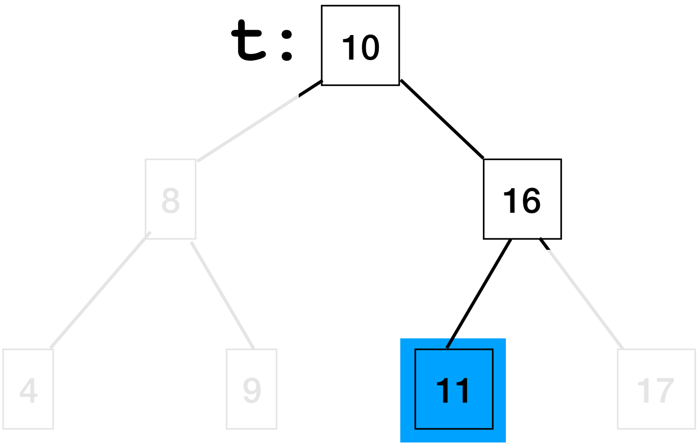
`search(right, 11)`

11 < 16

`search(left, 11)`

11 == 11

Searching a BST




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

11 > 10

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

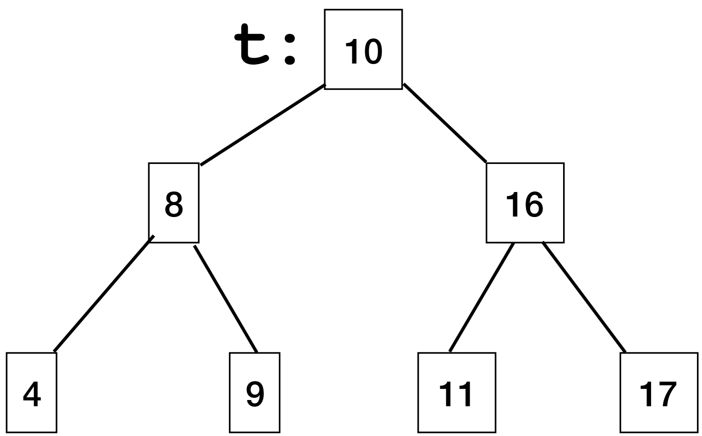
11 < 16

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

11 == 11

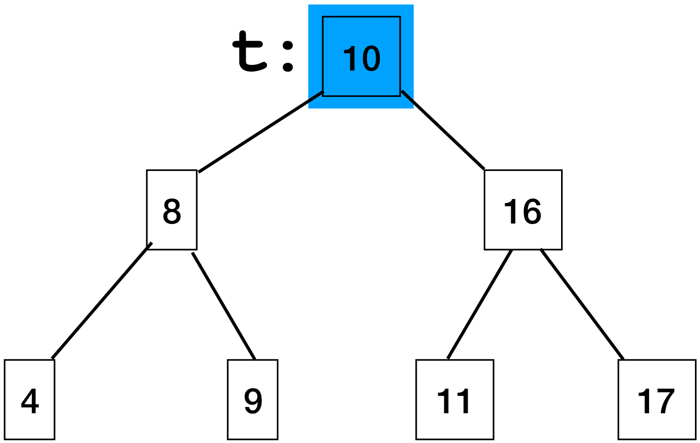
```
found it! return.
```

Searching a BST - the nonexistent case



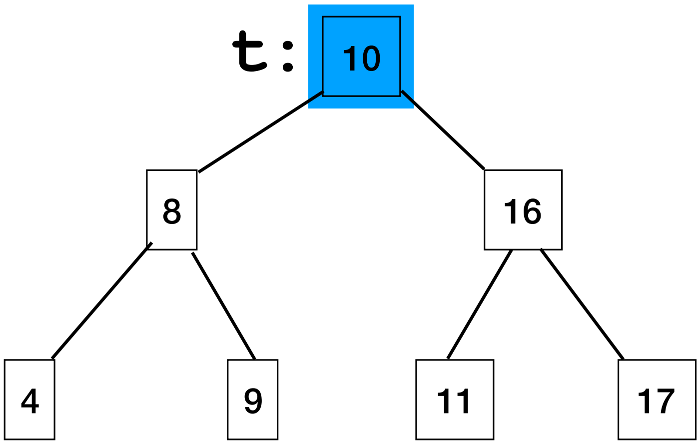

```
search(t, 5)
```

Searching a BST - the nonexistent case



`search(t, 5)`

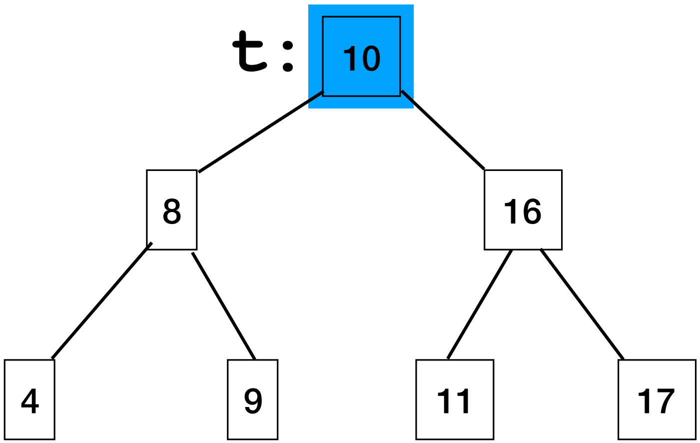
Searching a BST - the nonexistent case




```
search(t, 5)
```

5 < 10

Searching a BST - the nonexistent case

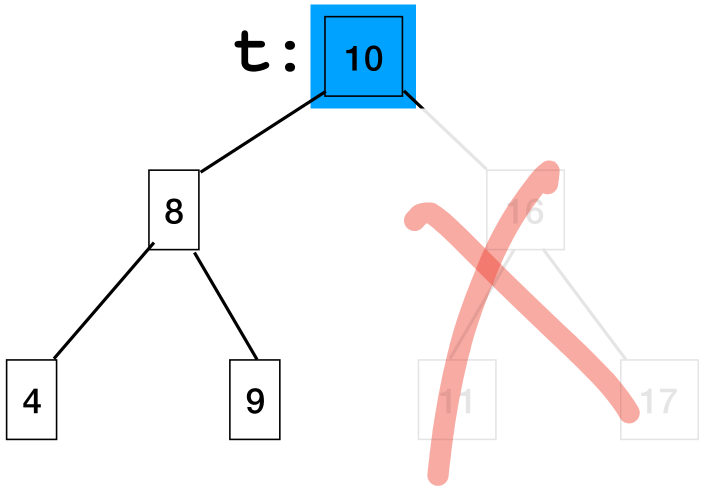


`search(t, 5)`

5 < 10

`search(left, 5)`

Searching a BST - the nonexistent case

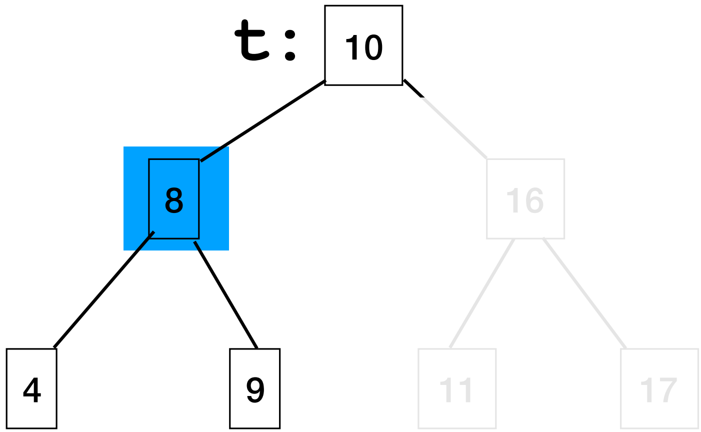



```
search(t, 5)
```

5 < 10

```
search(left, 5)
```

Searching a BST - the nonexistent case

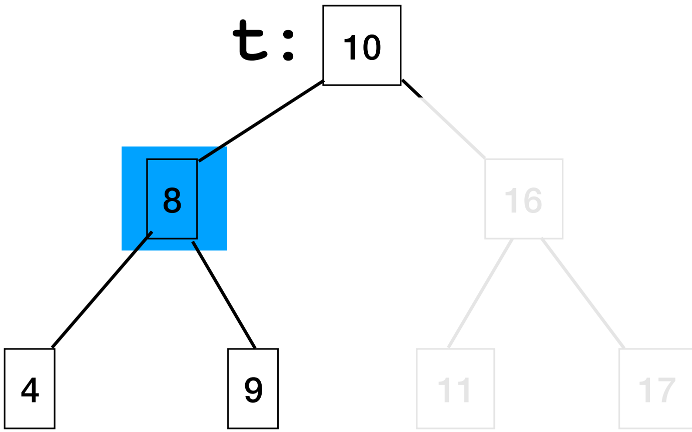



```
search(t, 5)
```

5 < 10

```
search(left, 5)
```

Searching a BST - the nonexistent case



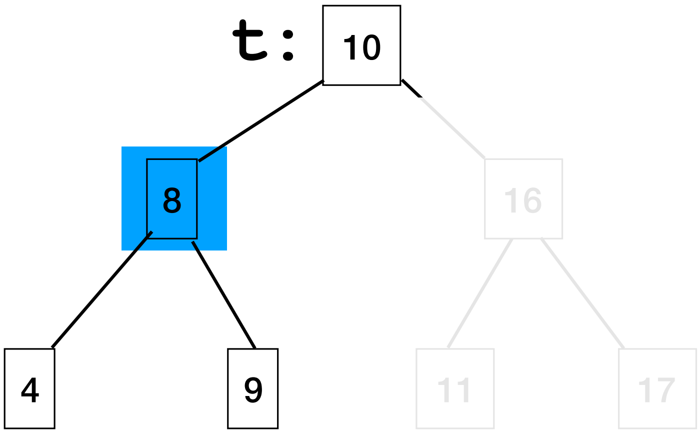
`search(t, 5)`

$5 < 10$

`search(left, 5)`

$5 < 8$

Searching a BST - the nonexistent case



`search(t, 5)`

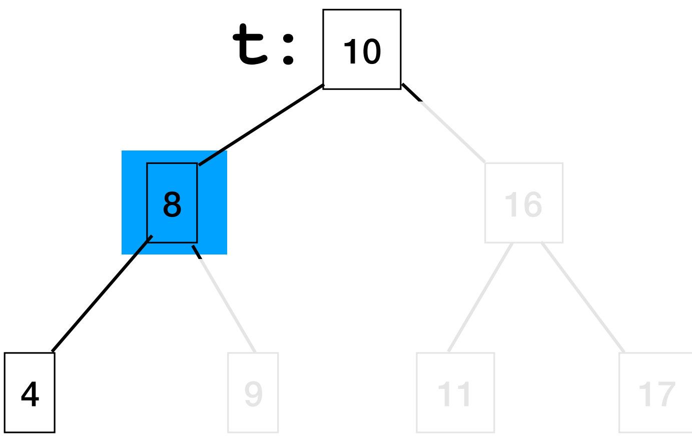
5 < 10

`search(left, 5)`

5 < 8

`search(left, 5)`

Searching a BST - the nonexistent case



`search(t, 5)`

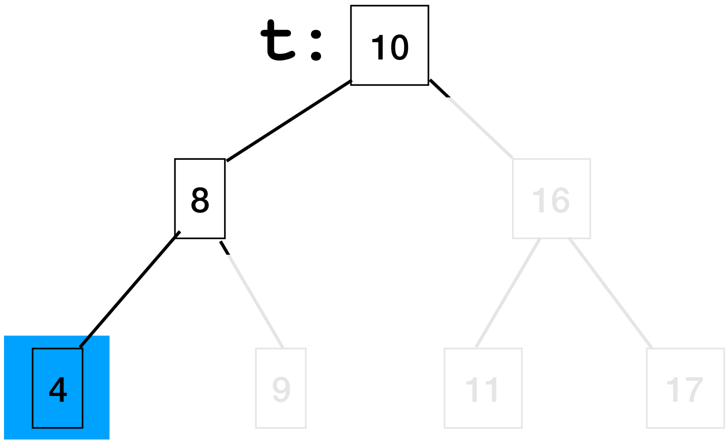
$5 < 10$

`search(left, 5)`

$5 < 8$

`search(left, 5)`

Searching a BST - the nonexistent case



`search(t, 5)`

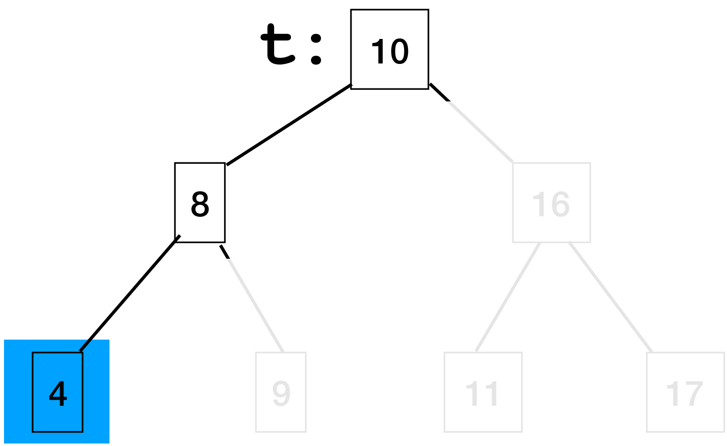
5 < 10

`search(left, 5)`

5 < 8

`search(left, 5)`

Searching a BST - the nonexistent case




```
search(t, 5)
```

5 < 10

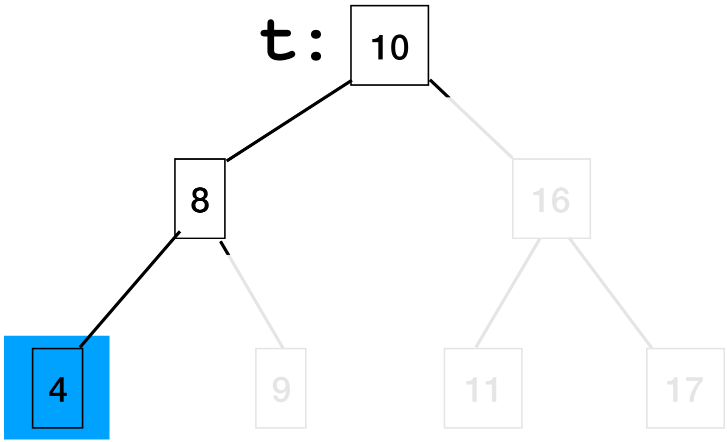
```
search(left, 5)
```

5 < 8

```
search(left, 5)
```

5 > 4

Searching a BST - the nonexistent case




```
search(t, 5)
```

5 < 10

```
search(left, 5)
```

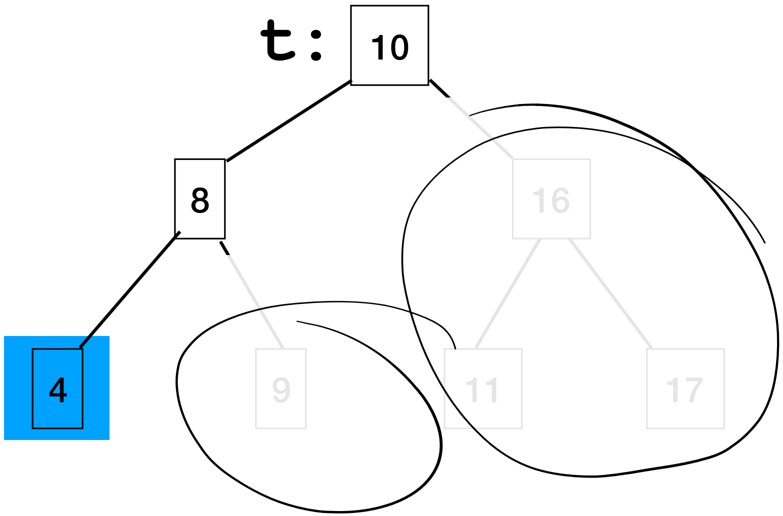
5 < 8

```
search(left, 5)
```

5 > 4

```
search(right, 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!`

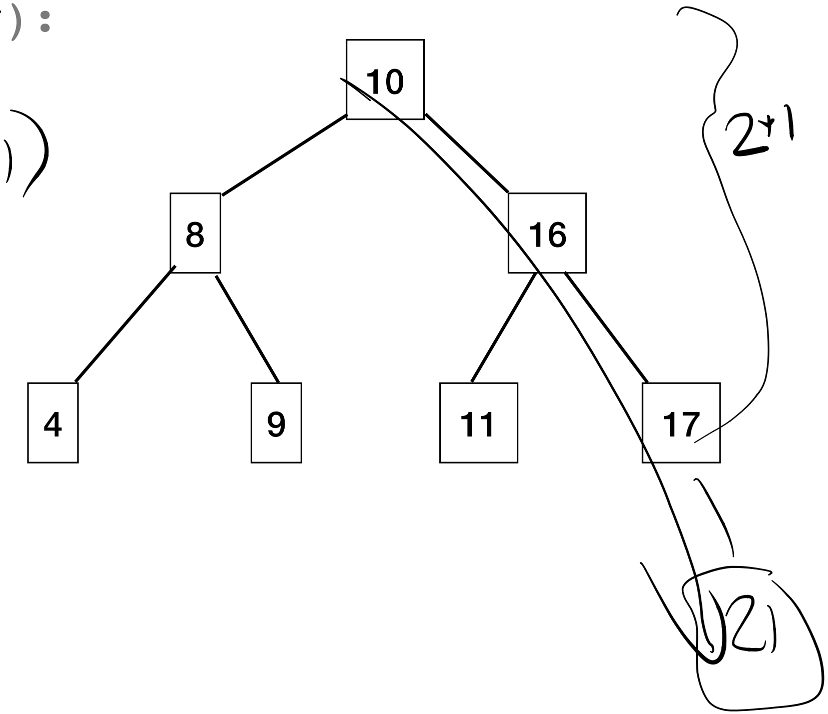
Searching: BT vs BST

```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);
}
```

```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);
  }
}
}
```

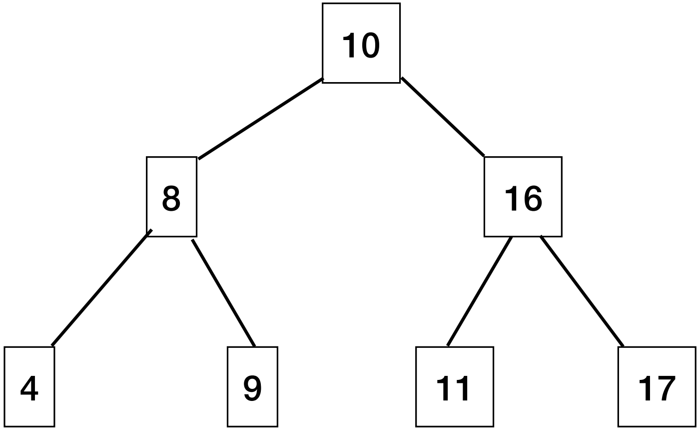
Searching a BST: What's the runtime?

```
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)
```



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)
```

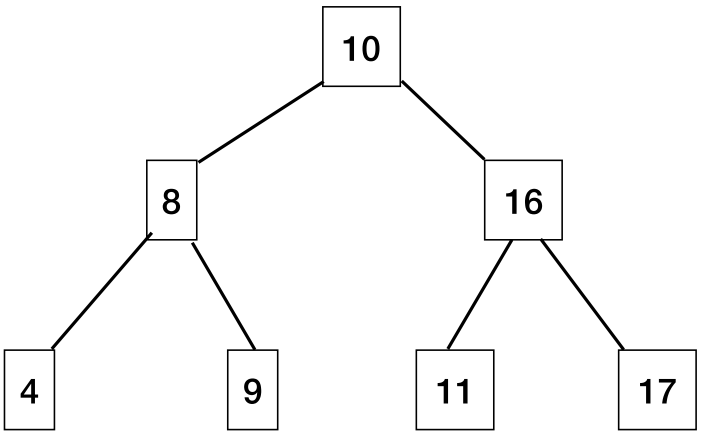


If h is the tree's **height**, search can visit at most h+1 nodes!

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

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)**.

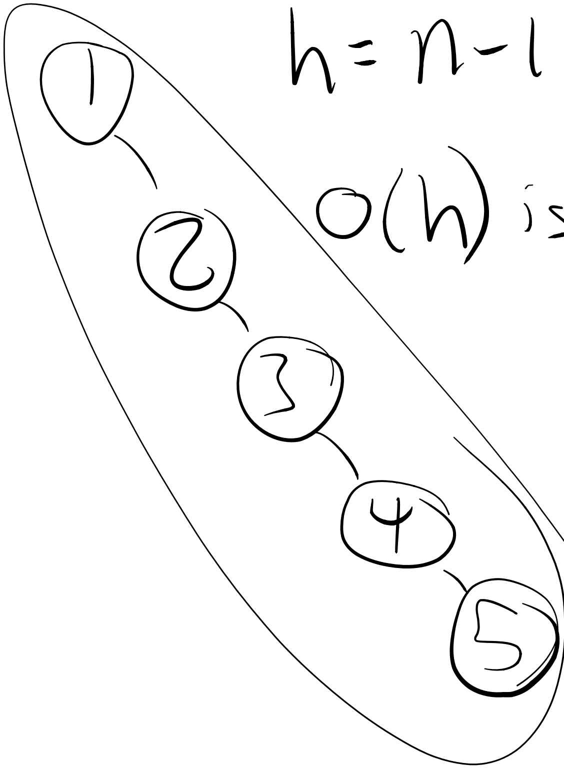
That's great, but how does h relate to n, the number of nodes?

Suppose we have n nodes. What's the

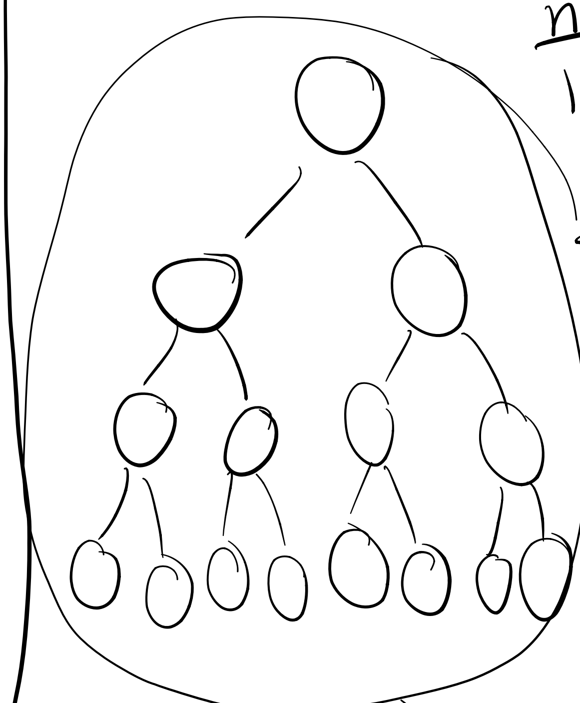
Max height?

$h = n - 1$

$O(h)$ is $O(n)$



Min height?



n_{ard}	d
1	0
2	1
4	2
8	3
2^d	d

$h = O(\log n)$

2^h at leaf level

$$n = 2^h + 2^{h-1} + 2^{h-2} + \dots 2^0$$

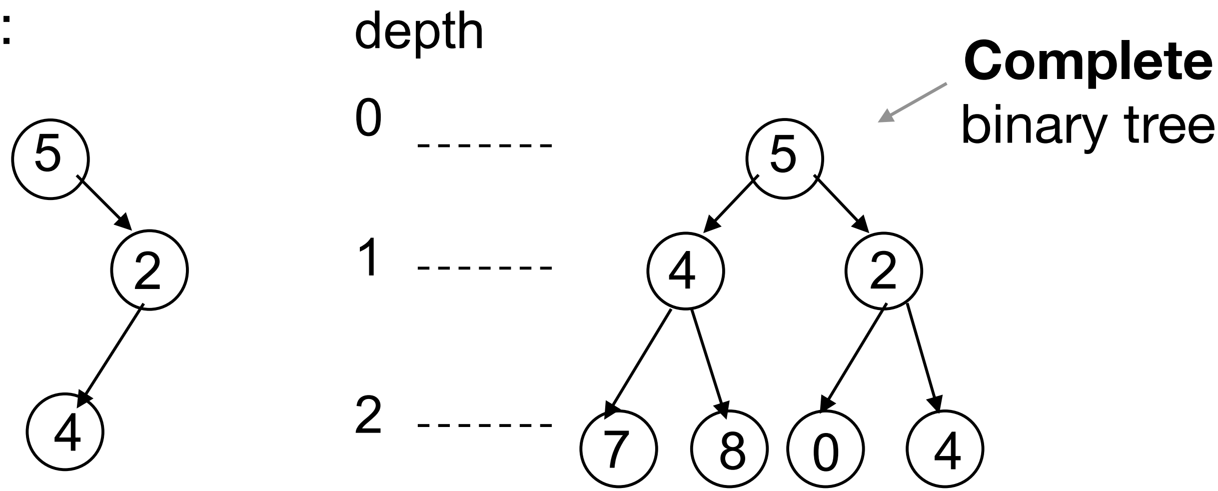
$$n = 2^h + 2^h - 1$$

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

$$O(\log_2 n) = h$$

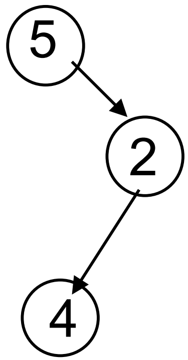
How many nodes does a tree with height h have?

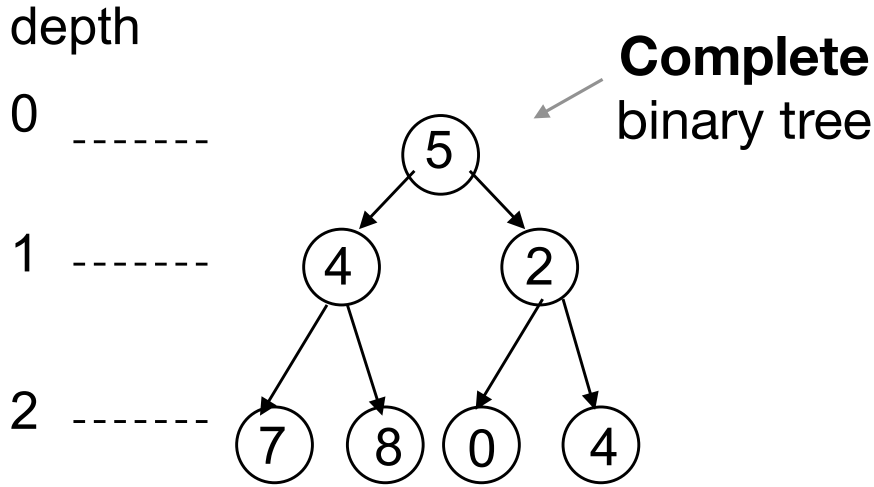
Consider h = 2:



How many nodes does a tree with height h have?

Consider h = 2:





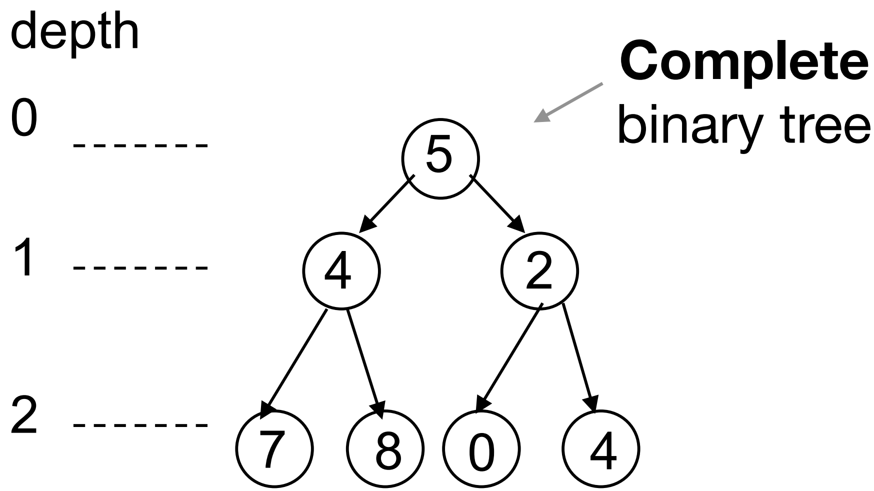
Fewest possible:

n = h+1

n is O(h)

How many nodes does a tree with height h have?

Consider h = 2:



Fewest possible:

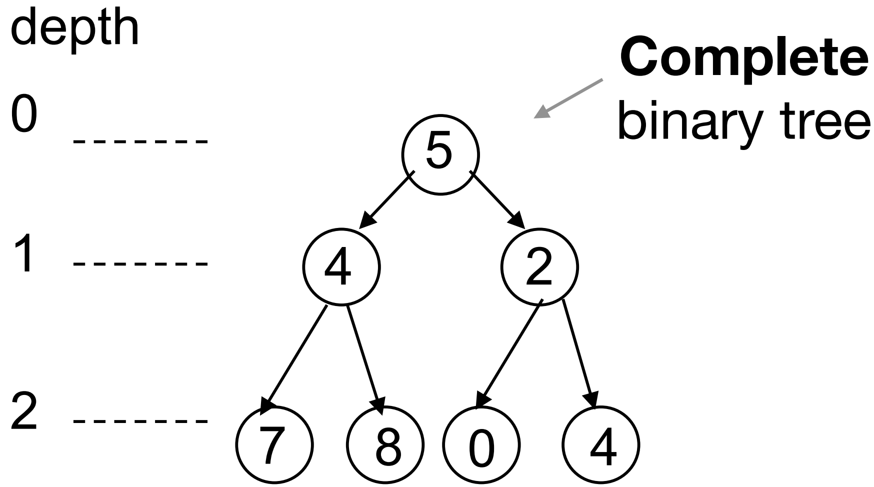
n = h+1

n is O(h)

h is O(n)

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:

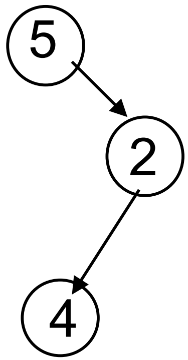
At depth d: 2^d nodes possible.

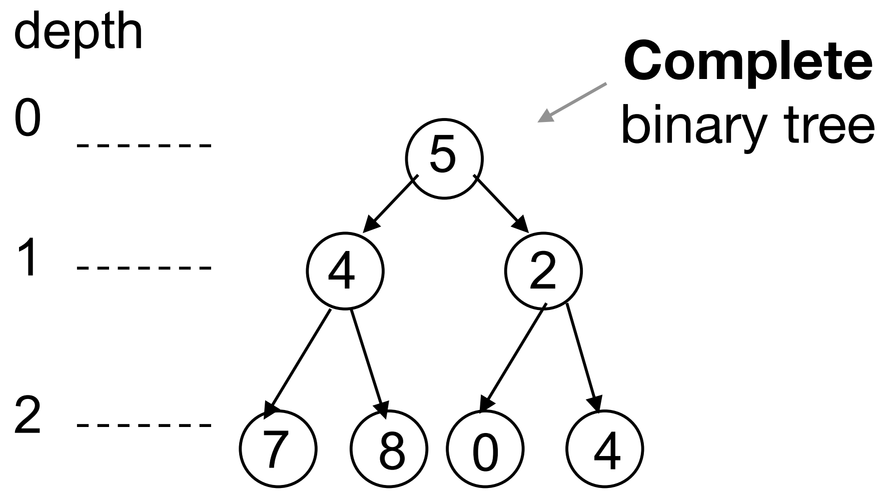
At all depths: $2^0 + 2^1 + \ldots + 2^h$

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

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.

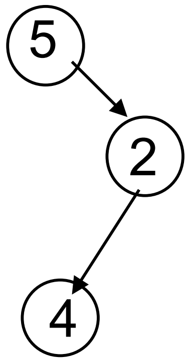
At all depths: $2^0 + 2^1 + \ldots + 2^h$

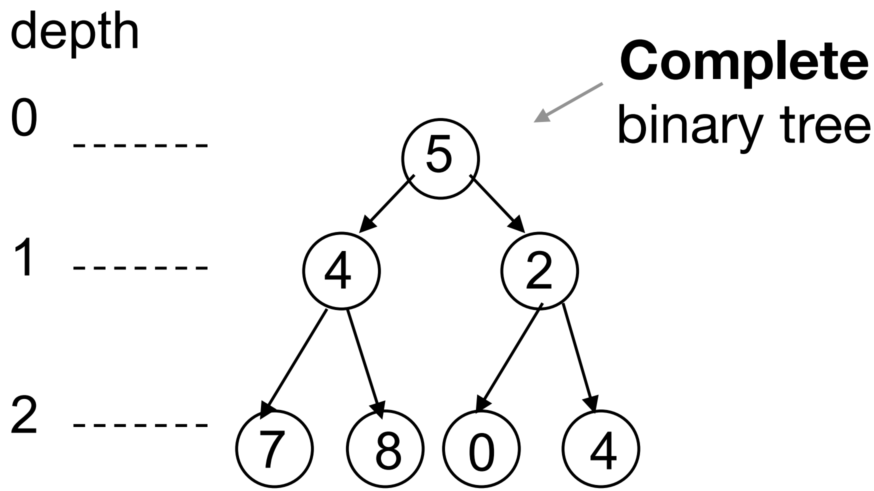
$= 2^{h+1} - 1$

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

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

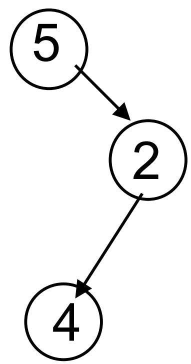
$= 2^{h+1} - 1$

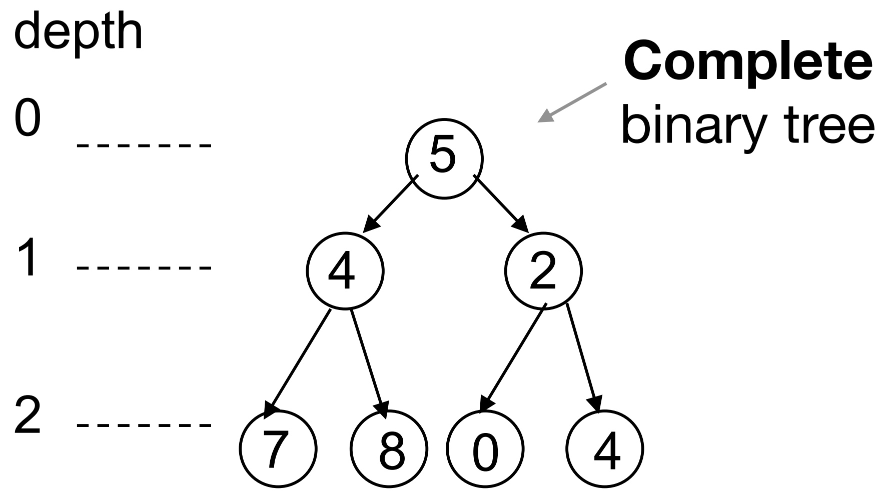
$n = 2^{h+1} - 1$

n is $O(2^h)$

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

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

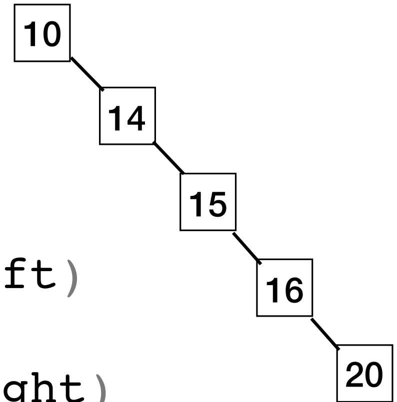
$n = 2^{h+1} - 1$

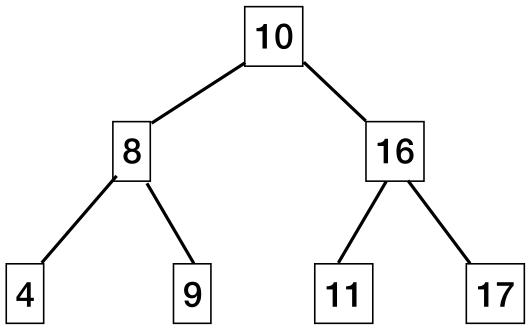
n is $O(2^h)$

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)
```



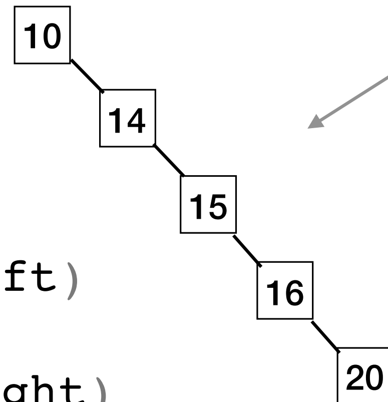


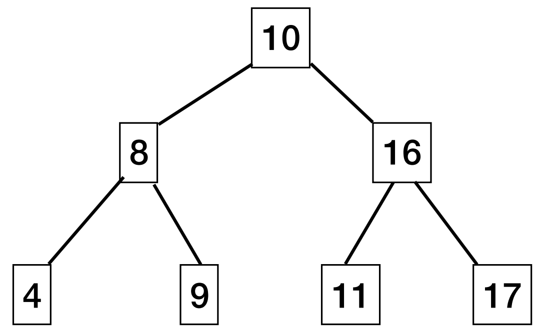
Runtime of search is O(h). Worst: **O(n)** Best: **O(log n)**

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





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