CSCI 241

Lecture 10

Binary Search Trees: Removal, Balanced BSTs

Announcements

- Reminder: today is the deadline to declare the major!
  - To be eligible to apply students must be in the last of (241, 247, 301) and submit an application and major declaration card—both are available from the CS Advising Office, CF 459.

Happenings

Monday, 2/4 – CSCI Faculty Candidate: Research Talk – 4 pm in CF 316

Tuesday, 2/5 – CSCI Faculty Candidate: Teaching Talk – 4 pm in CF 316

Tuesday, 2/5 – ACM Research Talk: Nick Majeske! – 5 pm in CF 316

Wednesday, 2/6 – PNNL Info Table – 11 am – 3 pm in the CF 4th Floor Foyer

Wednesday, 2/6 – Tech Talk: PNNL – 5 pm in CF 105

Wednesday, 2/6 – Peer Lecture Series: Debugging Workshop – 5 pm in CF 420

Thursday, 2/7 – Winter Career Fair w/ STEM Focus – 11 am – 3 pm in the MAC Gym

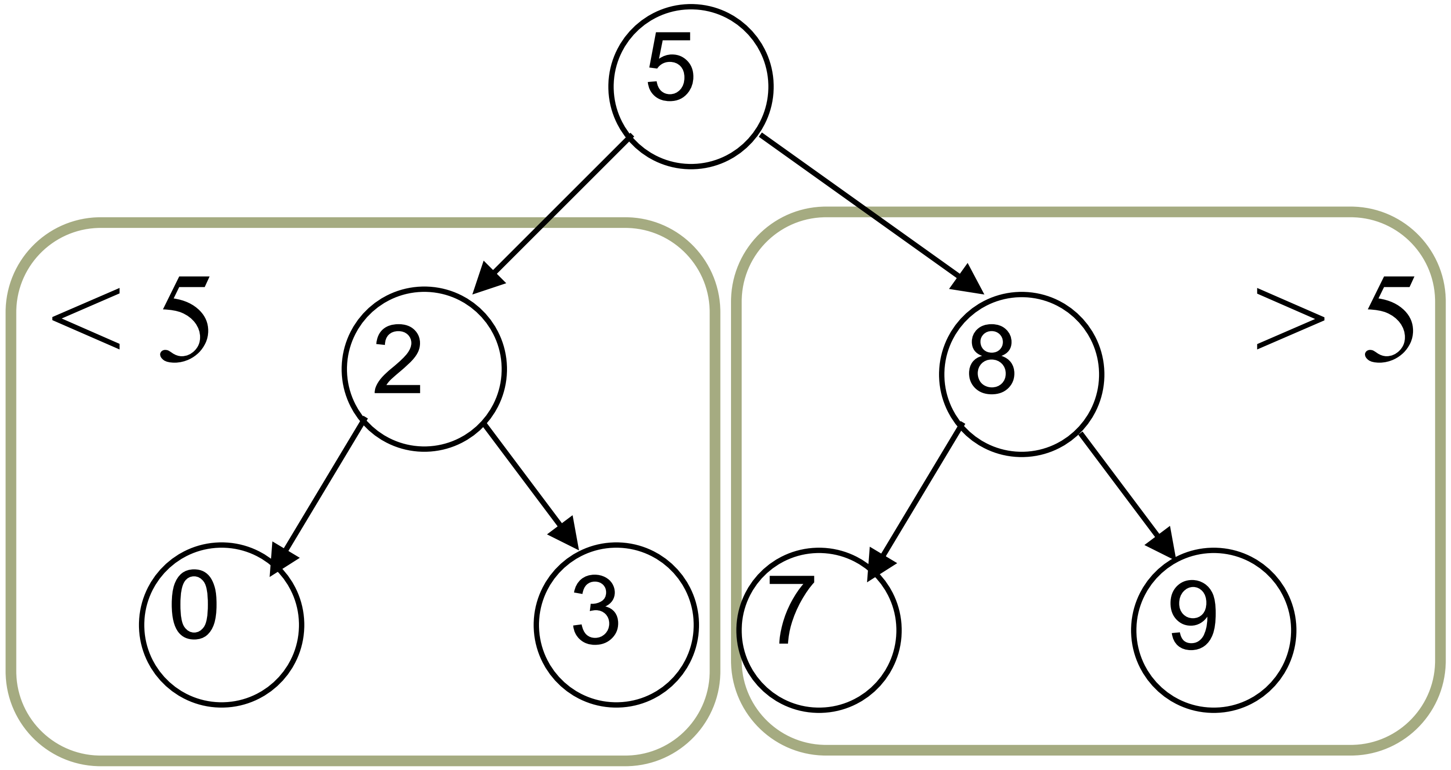
Goals (Wednesday and Today):

- 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.

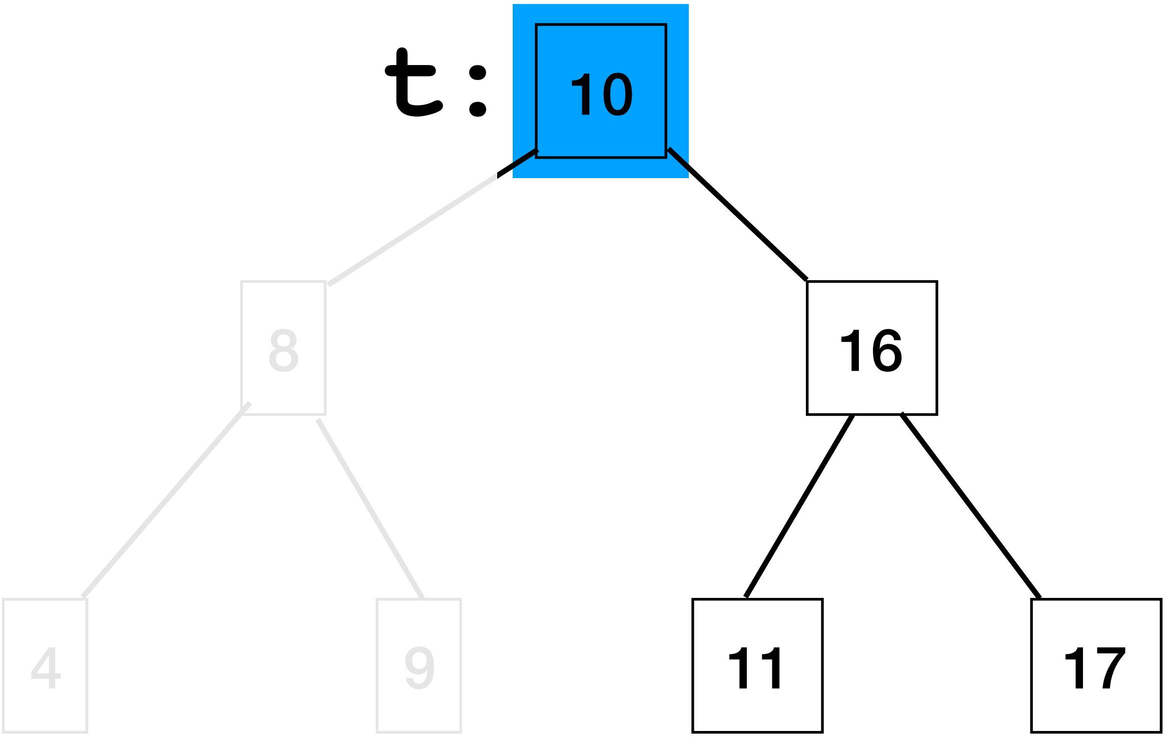
Binary Search Tree

```
/** 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



Searching a BST

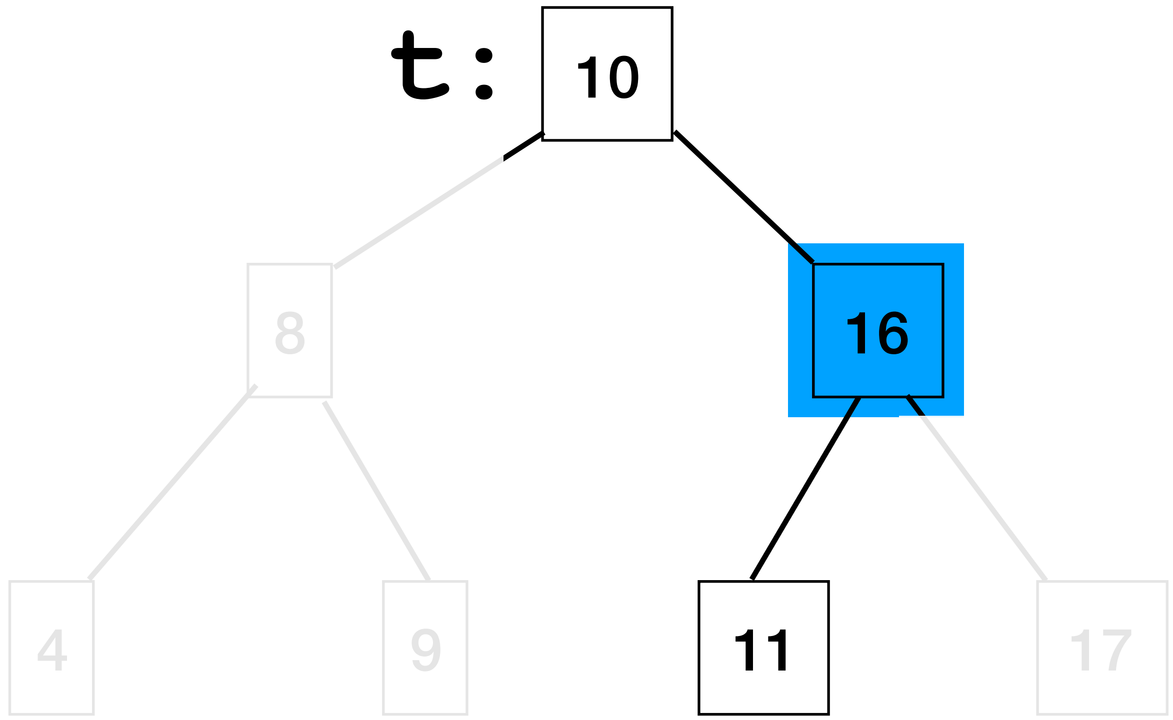


`search(t, 11)`

11 > 10

`search(right, 11)`

Searching a BST



`search(t, 11)`

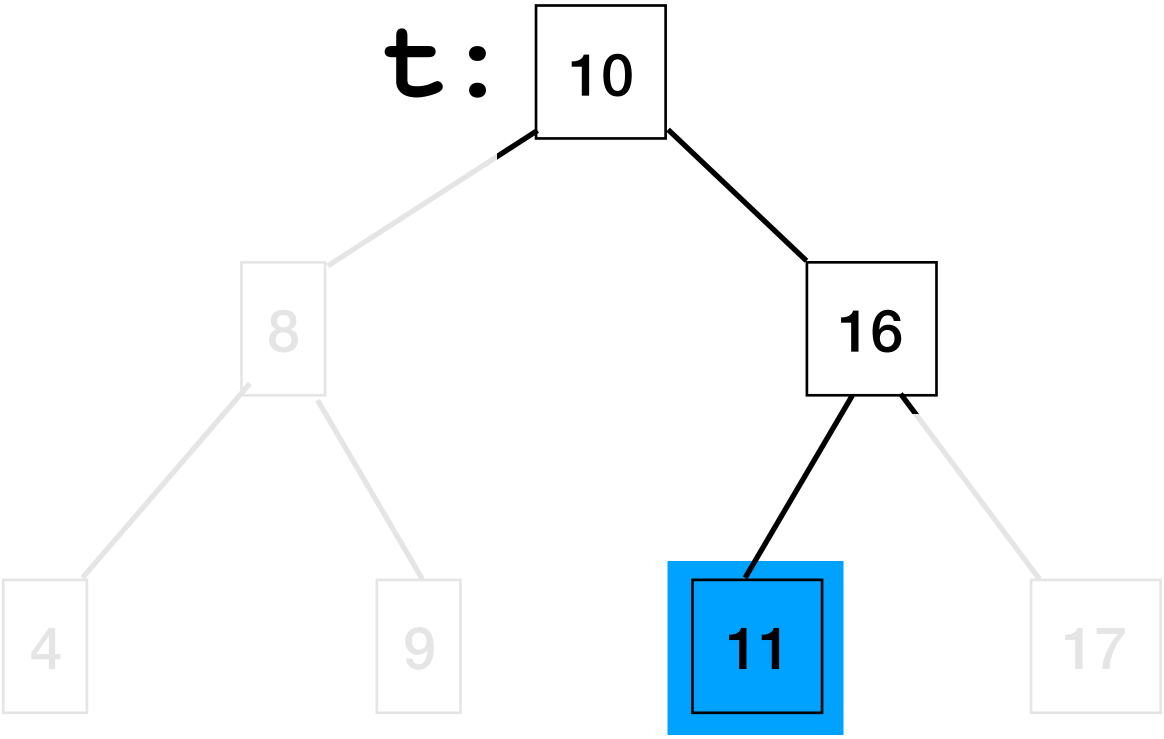
11 > 10

`search(right, 11)`

11 < 16

`search(left, 11)`

Searching a BST



`search(t, 11)`

11 > 10

`search(right, 11)`

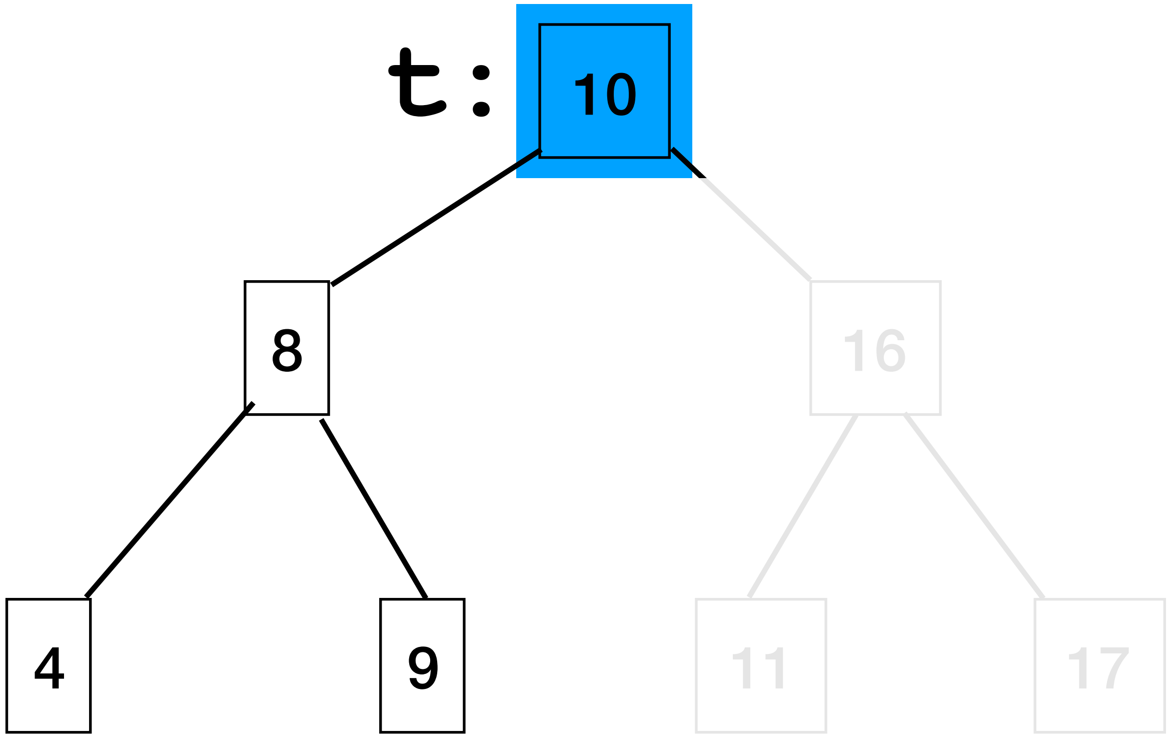
11 < 16

`search(left, 11)`

11 == 11

`found it! return.`

Searching a BST - the nonexistent case

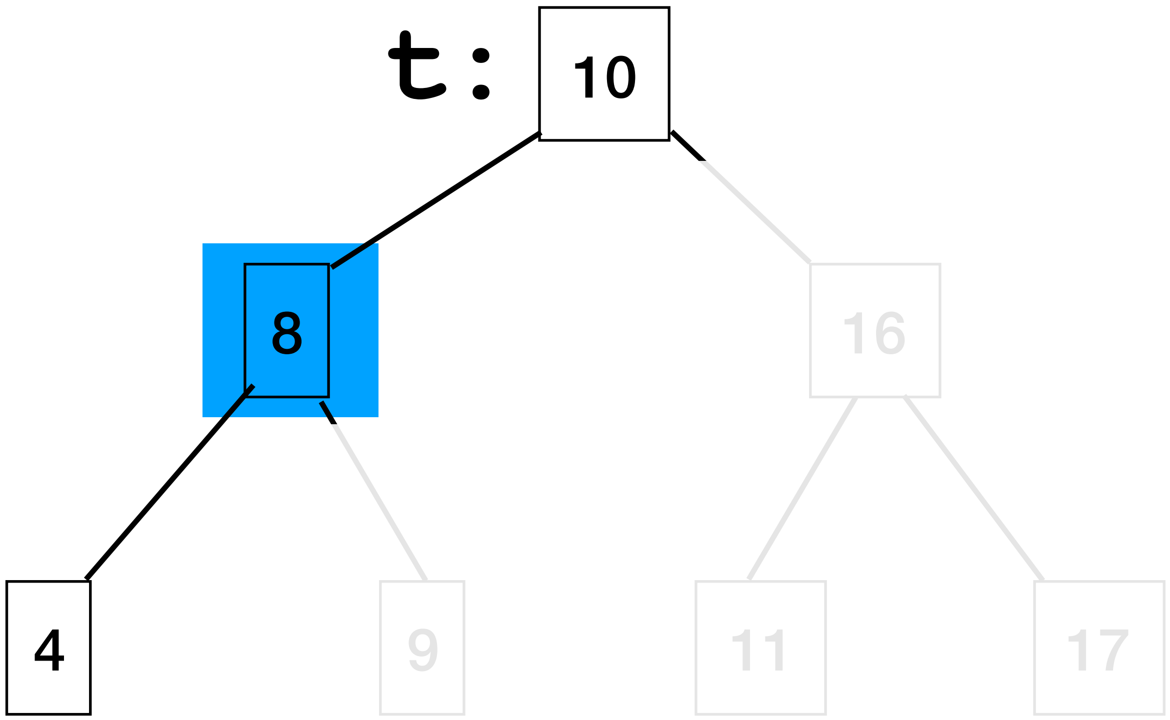


`search(t, 5)`

5 < 10

`search(left, 5)`

Searching a BST - the nonexistent case



`search(t, 5)`

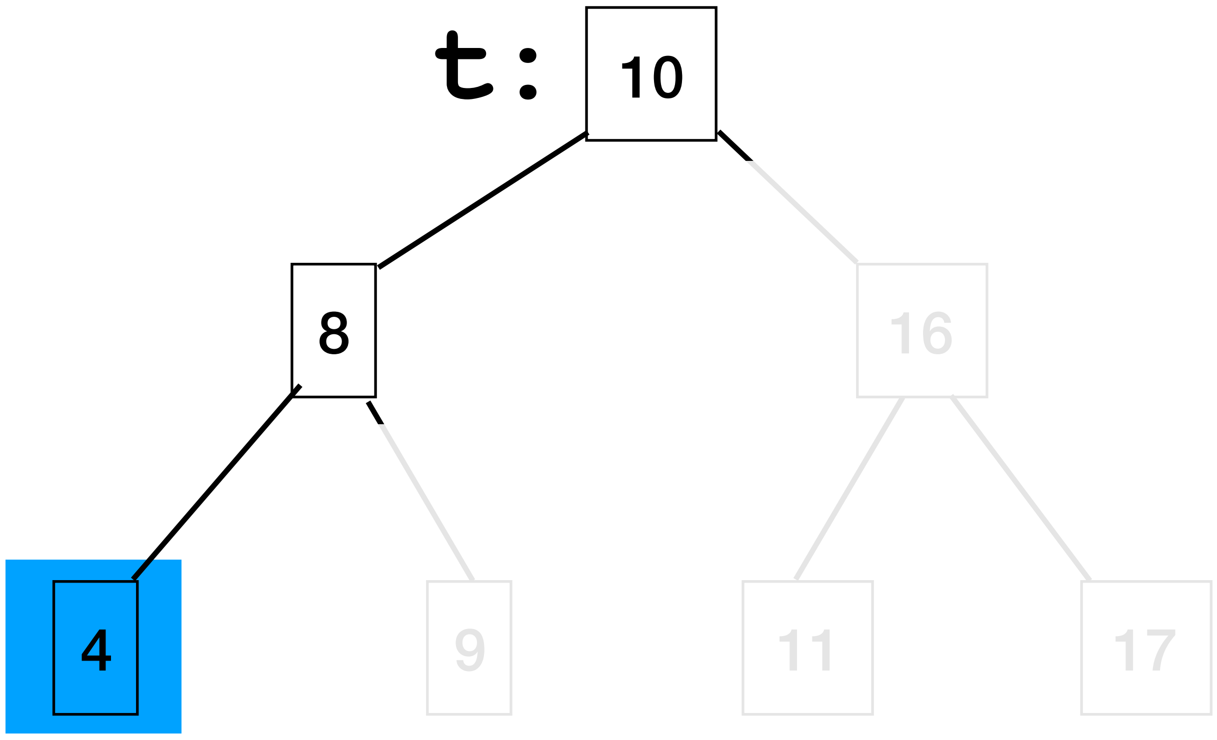
$5 < 10$

`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

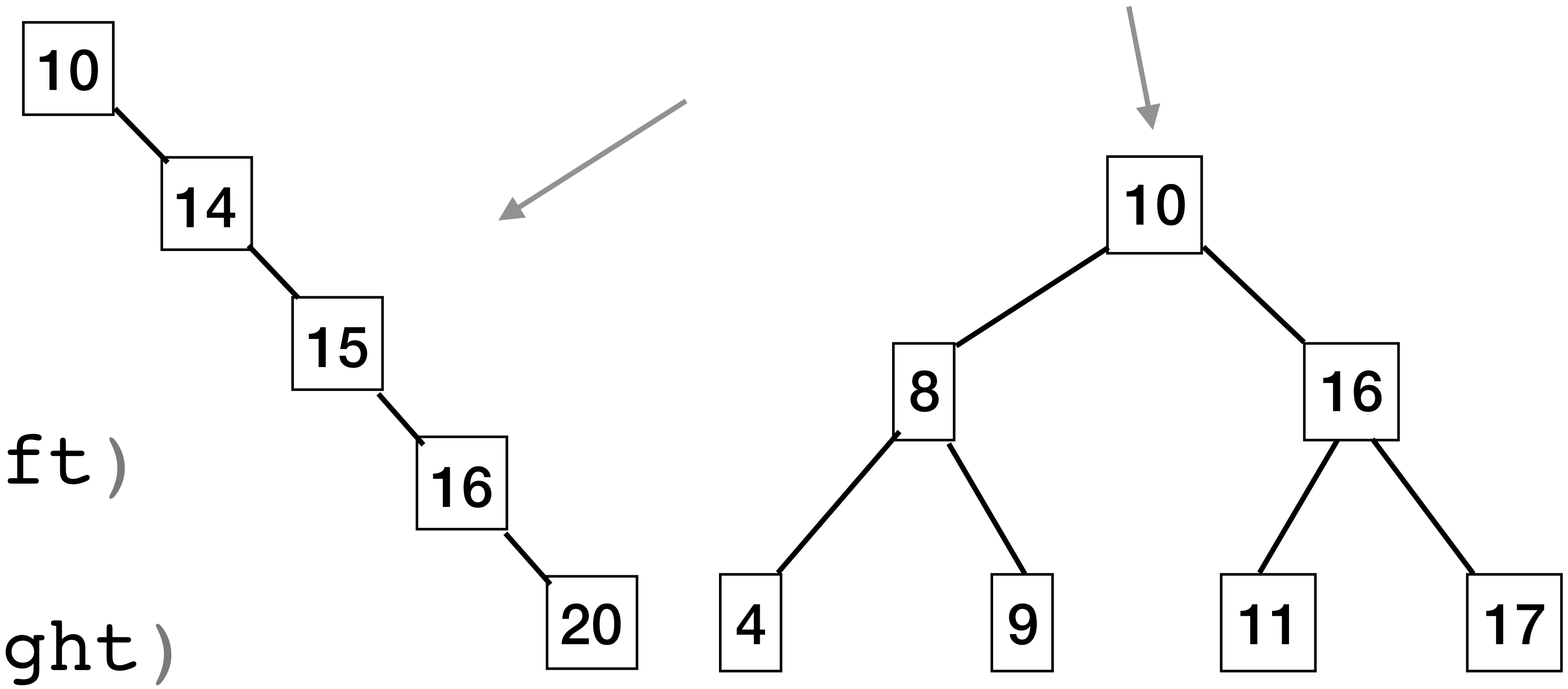
`search(right, 5)`

`null - not found!`

Searching 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)
```

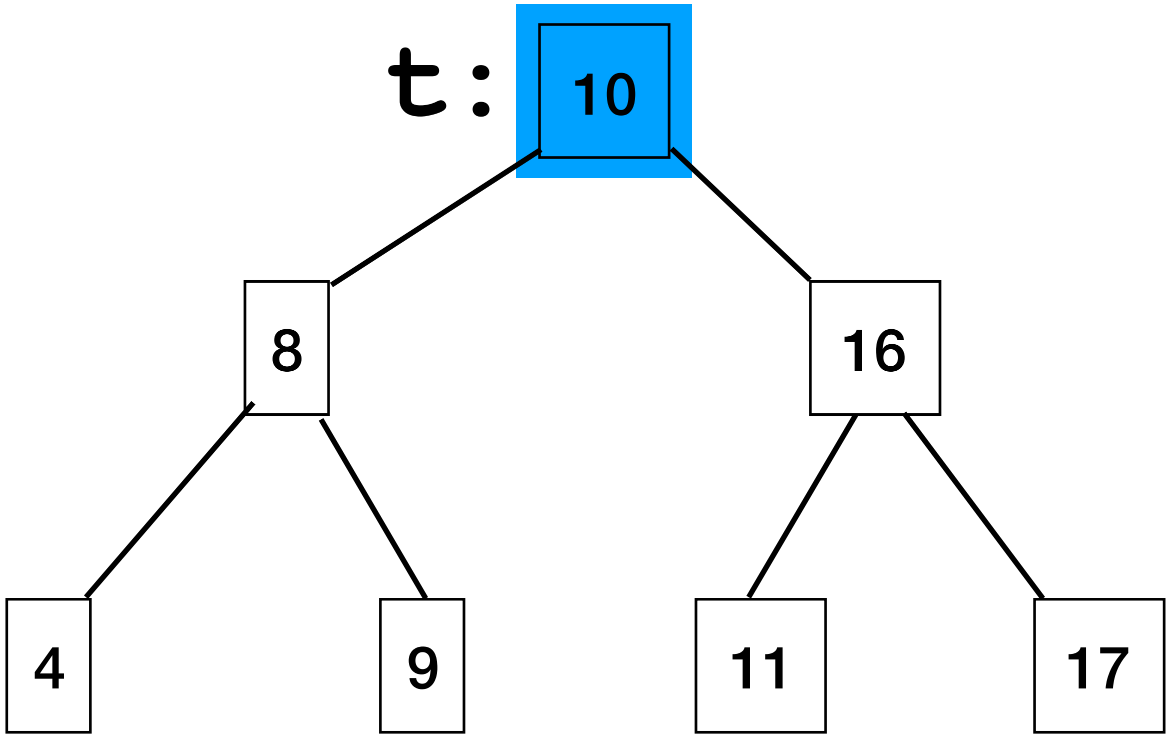
We want our trees to look more like this



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

Inserting into a BST

Inserting into a BST

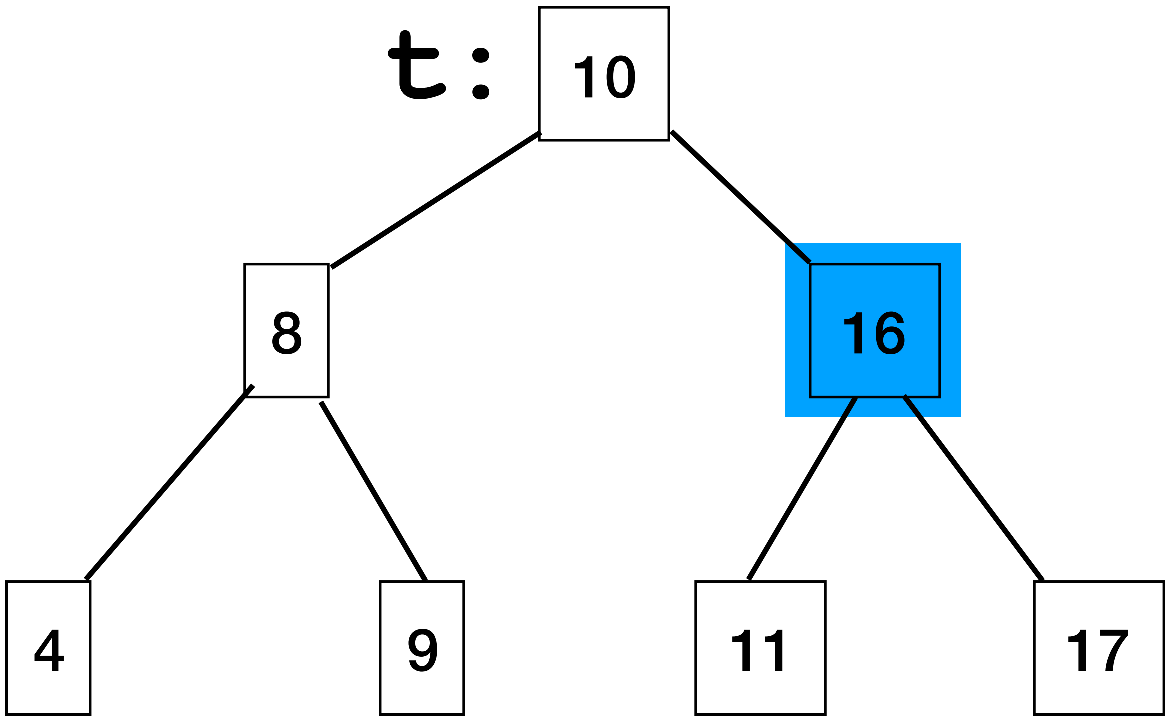


`insert(t, 11)`

11 > 10

`insert(right, 11)`

Inserting into a BST



`insert(t, 11)`

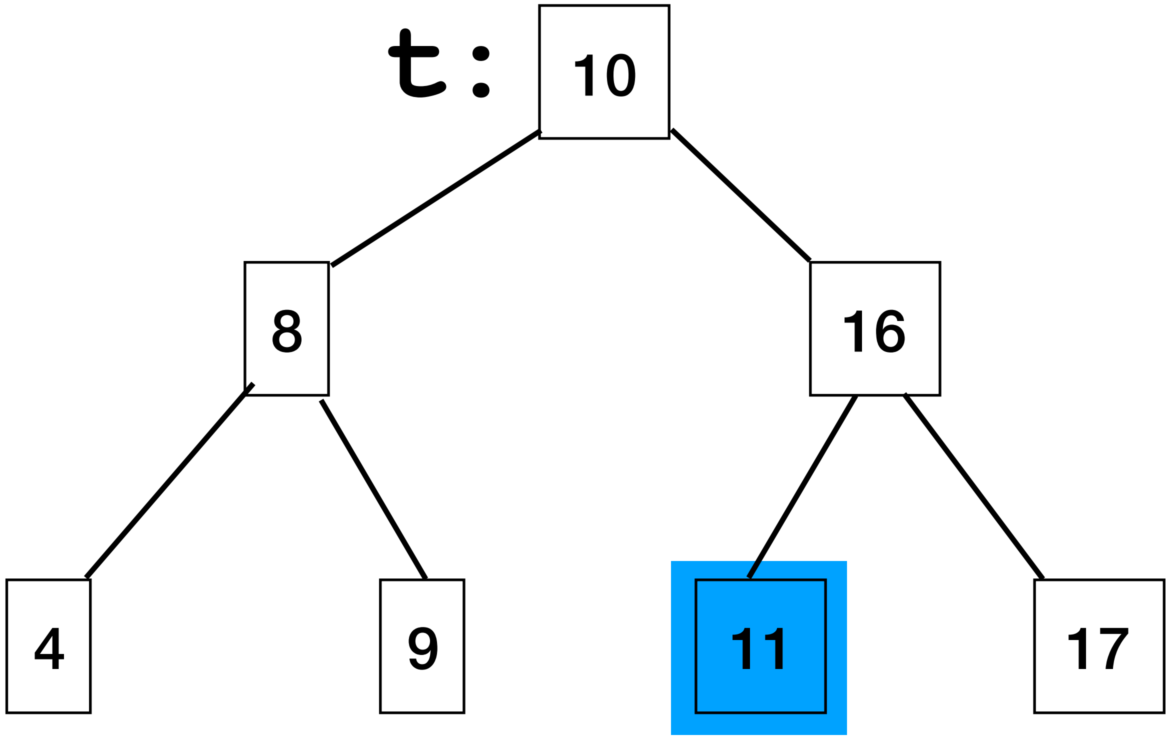
11 > 10

`insert(right, 11)`

11 < 16

`insert(left, 11)`

Inserting into a BST



`insert(t, 11)`

11 > 10

`insert(right, 11)`

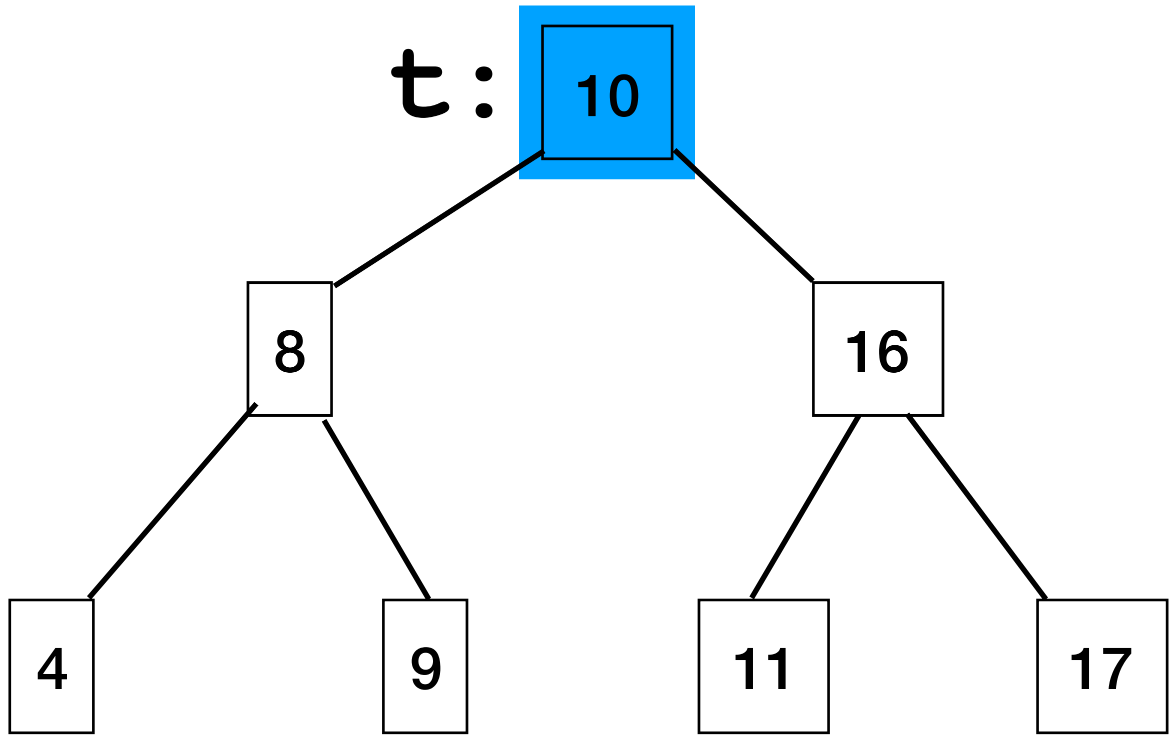
11 < 16

`insert(left, 11)`

11 == 11

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

Inserting into a BST - the nonexistent case

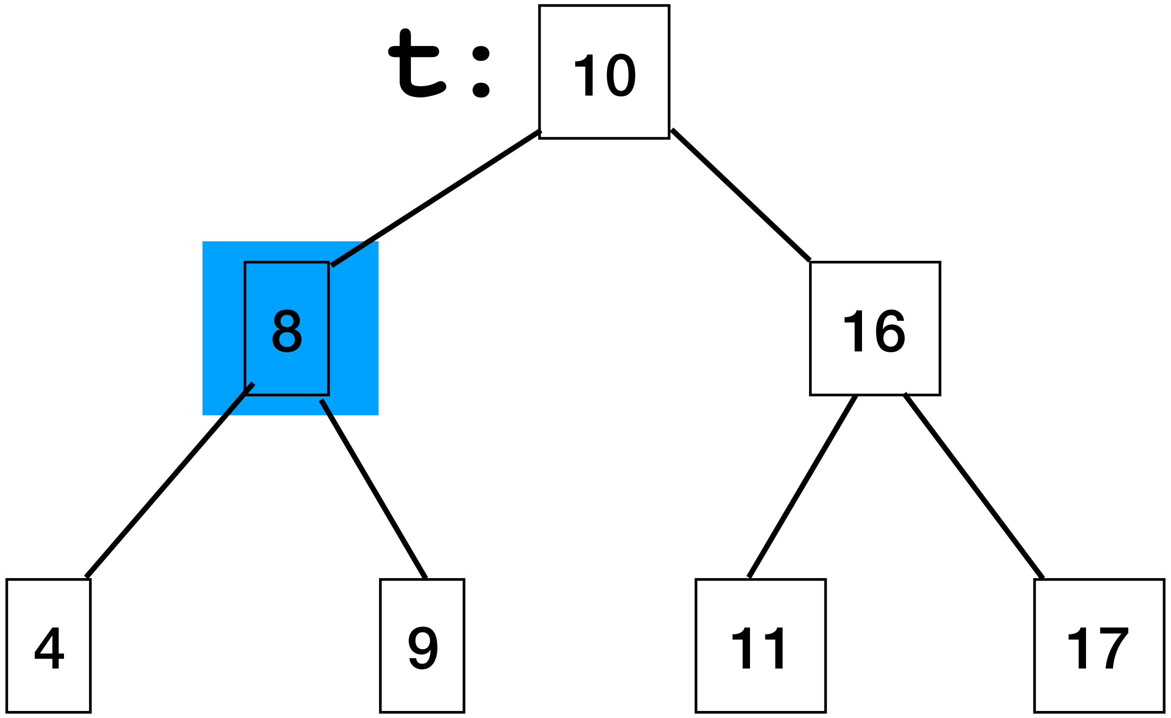


`insert(t, 5)`

5 < 10

`insert(left, 5)`

Inserting into a BST - the nonexistent case



`insert(t, 5)`

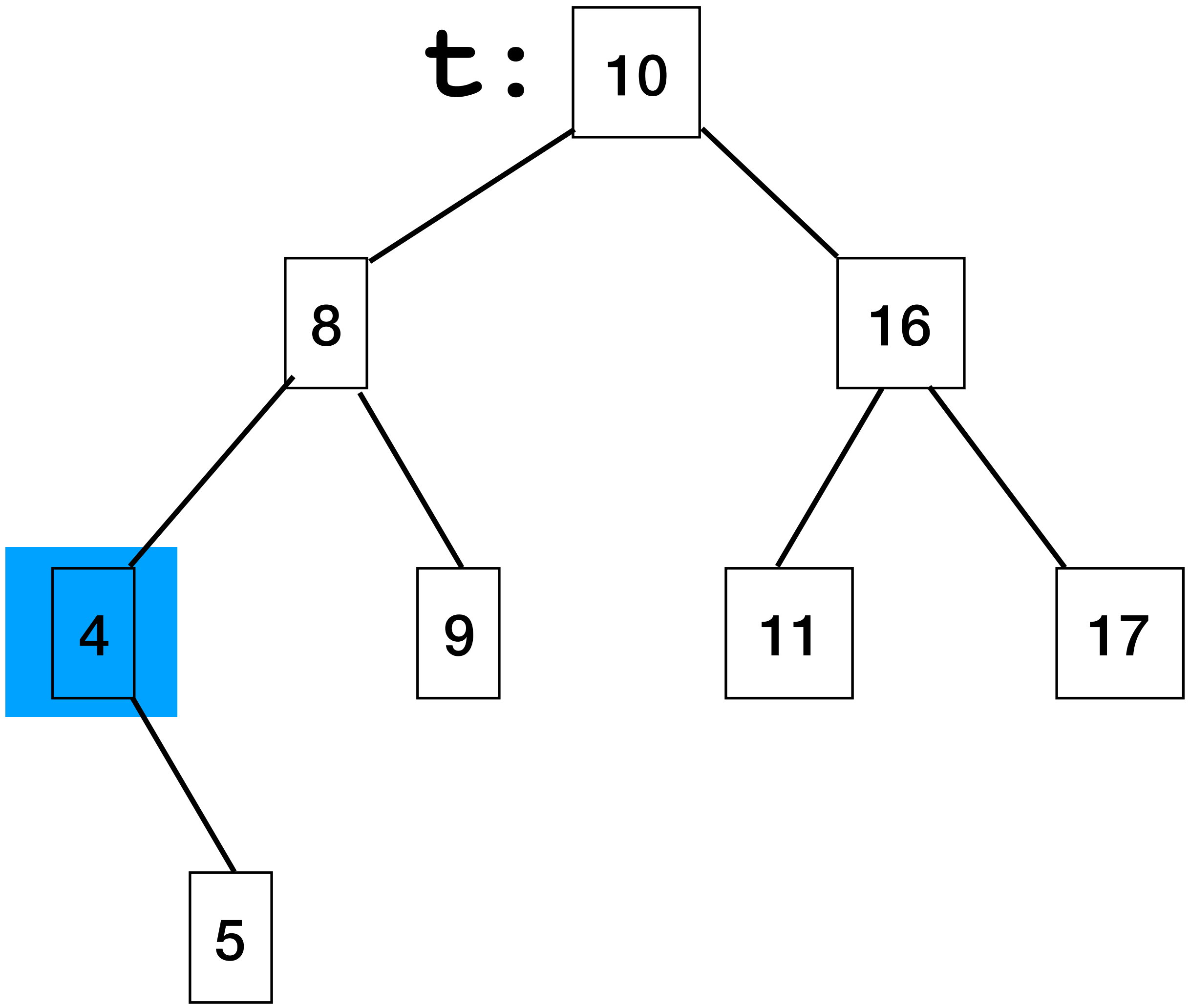
$5 < 10$

`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!`

Warm-up

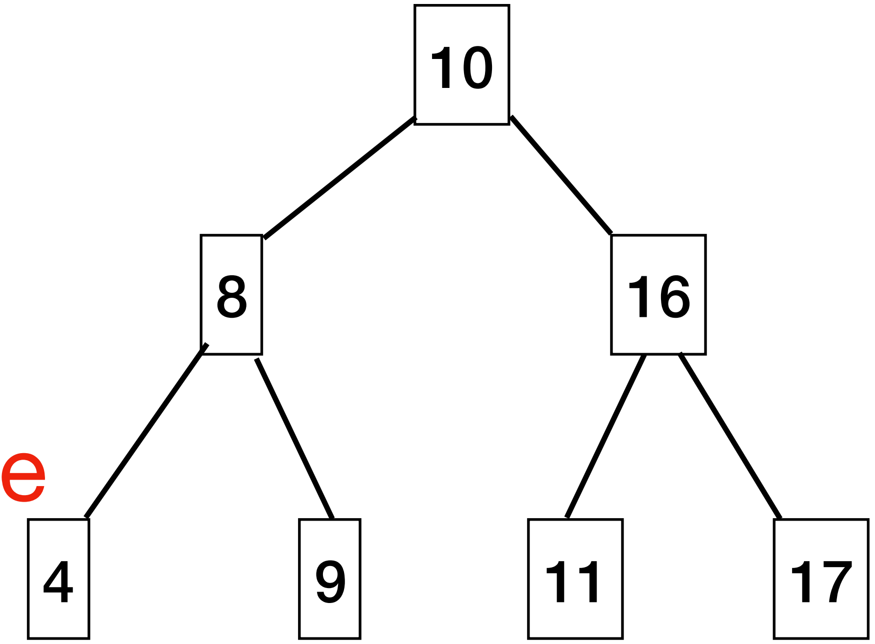
Write a method to find the smallest value in a BST:

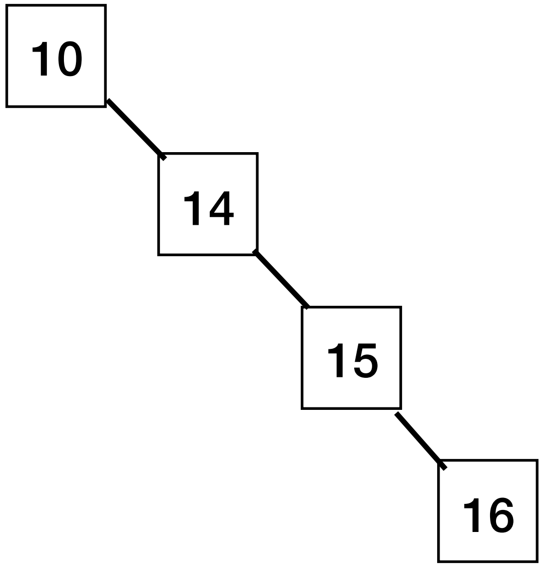
1. Spec

```
/** Returns min value in BST n.
  * pre: n is not null */
public int minimum(Node n) {
```

2. Base case

```
}
```





Warm-up

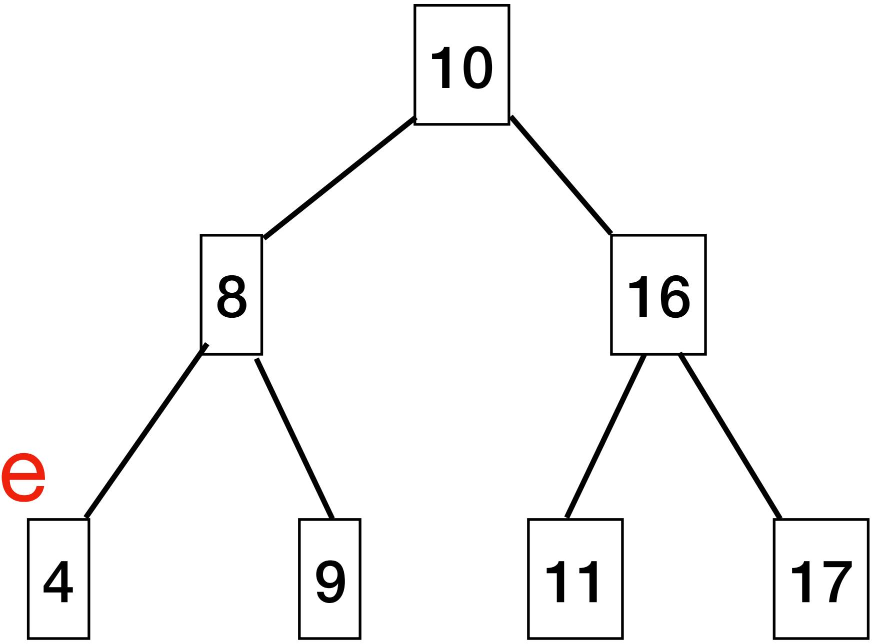
Write a method to find the smallest value in a BST:

1. Spec

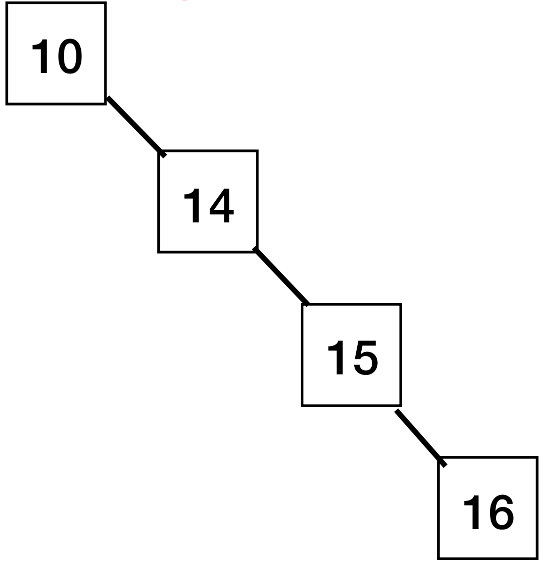
```
/** Returns min value in BST n.
  * pre: n is not null */
public int minimum(Node n) {
  if (n.left == null)
    return n.value;

}
```

2. Base case



4. Implement using recursive call



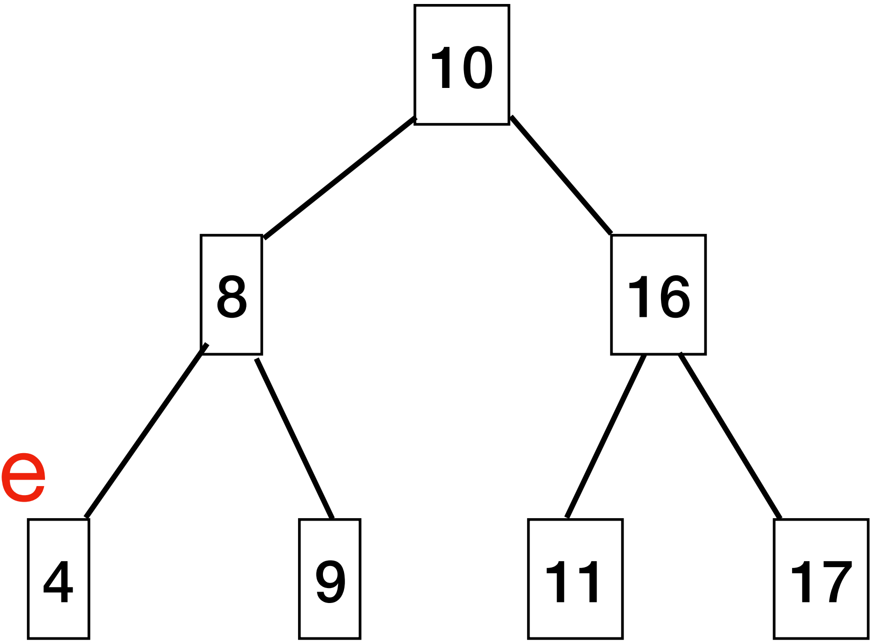
3. Recursive definition:

If n has a left child, smallest(n) is

- the smallest value in the left subtree

Warm-up

Write a method to find the smallest value in a BST:



1. Spec

```
/** Returns min value in BST n.
  * pre: n is not null */
public int minimum(Node n) {
  if (n.left == null)
    return n.value;
  return minimum(n.left);
}
```

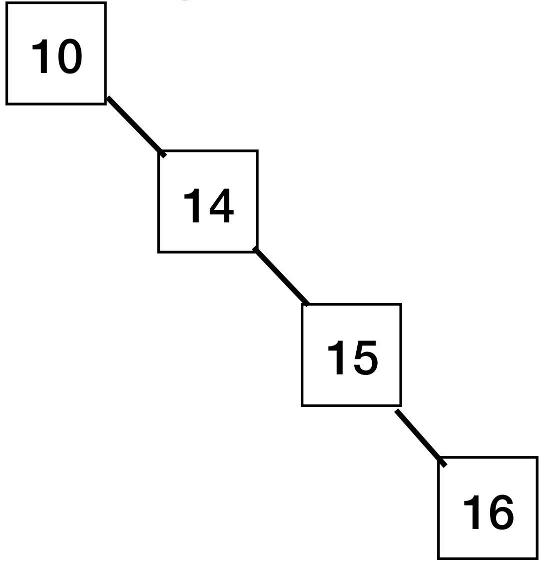
2. Base case

3. Recursive definition:

Smallest(n) is:

- the smallest value in the left subtree, or
- n.value if no left subtree exists.

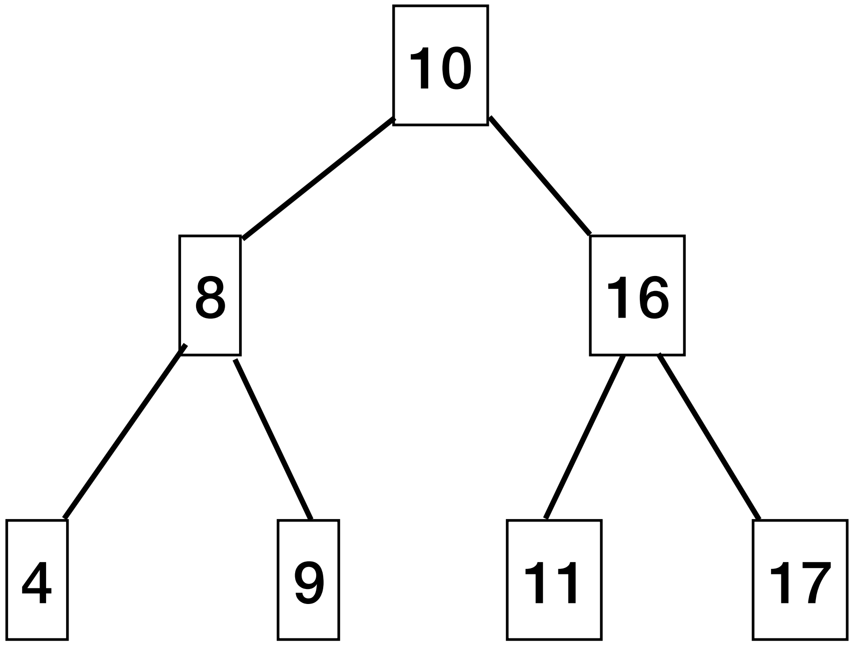
4. Implement using recursive call

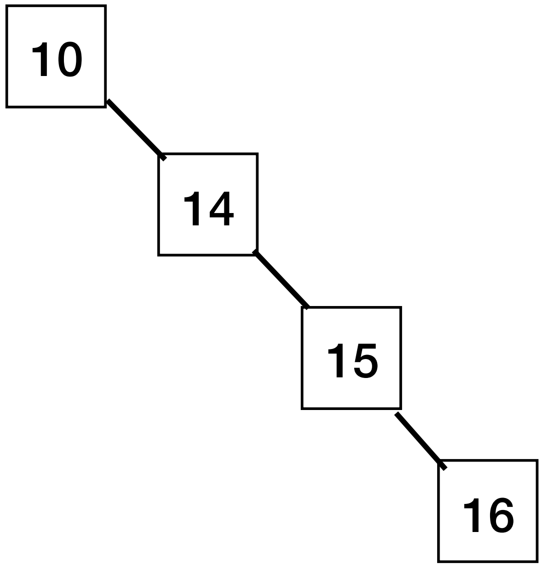


Warm-up

Write a method to find the smallest value in a BST:

```
/** Returns min value in BST n.
  * pre: n is not null */
public int minimum(Node n) {
  if (n.left == null)
    return n.value;
  return minimum(n.left);
}
```

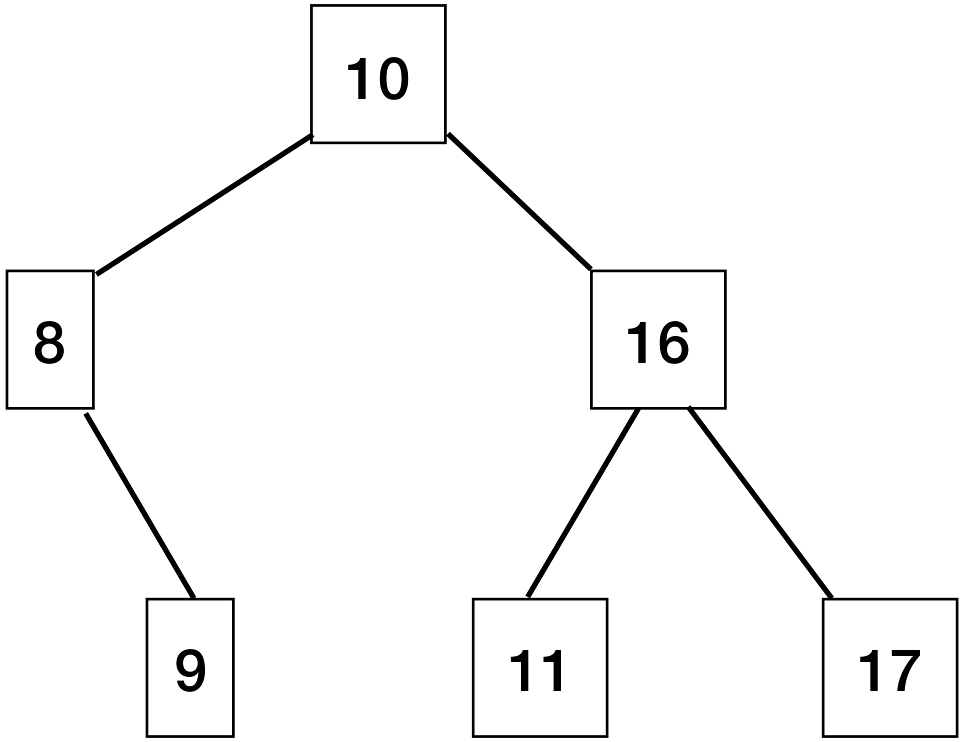




Deleting a node from a BST

Three possible cases:

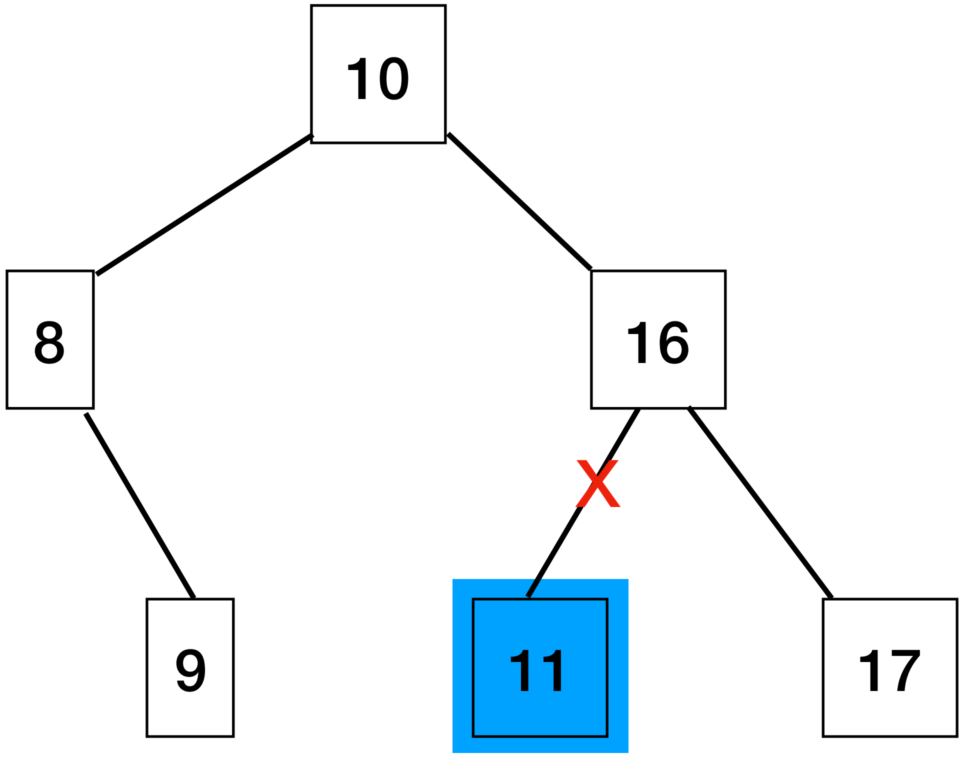
1. n has no children (is a leaf)
2. n has one child
3. n has two children



Deleting a node from a BST: Case 1

Three possible cases:

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

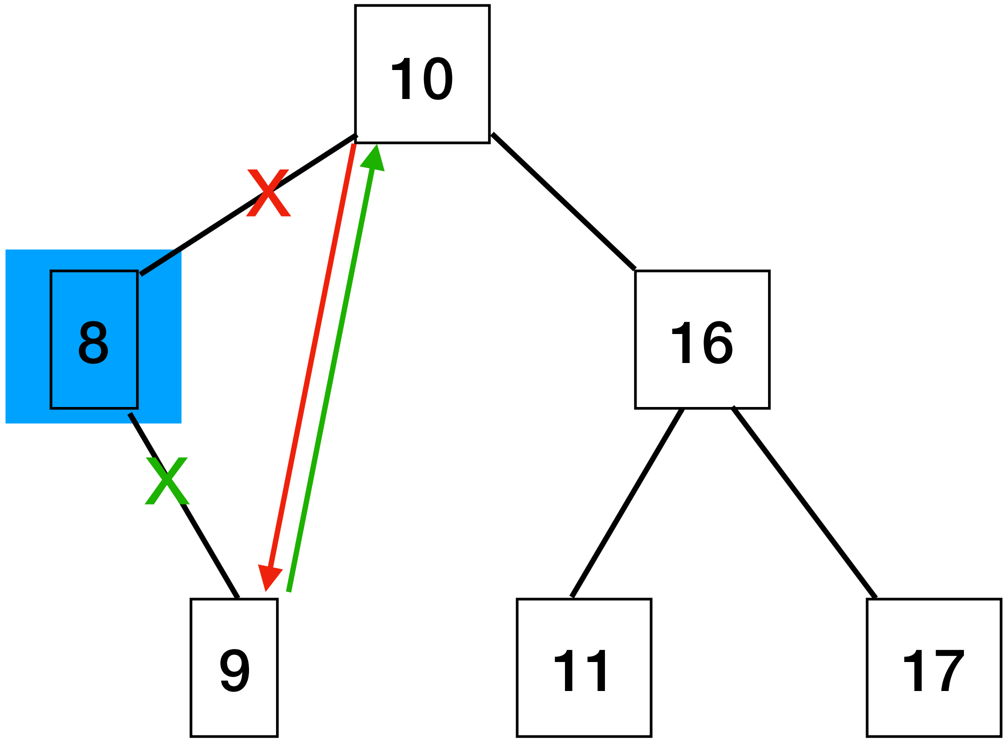


```
if (n is a leaf)
   replace parent's child with null
```

Deleting a node from a BST: Case 2

Three possible cases:

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

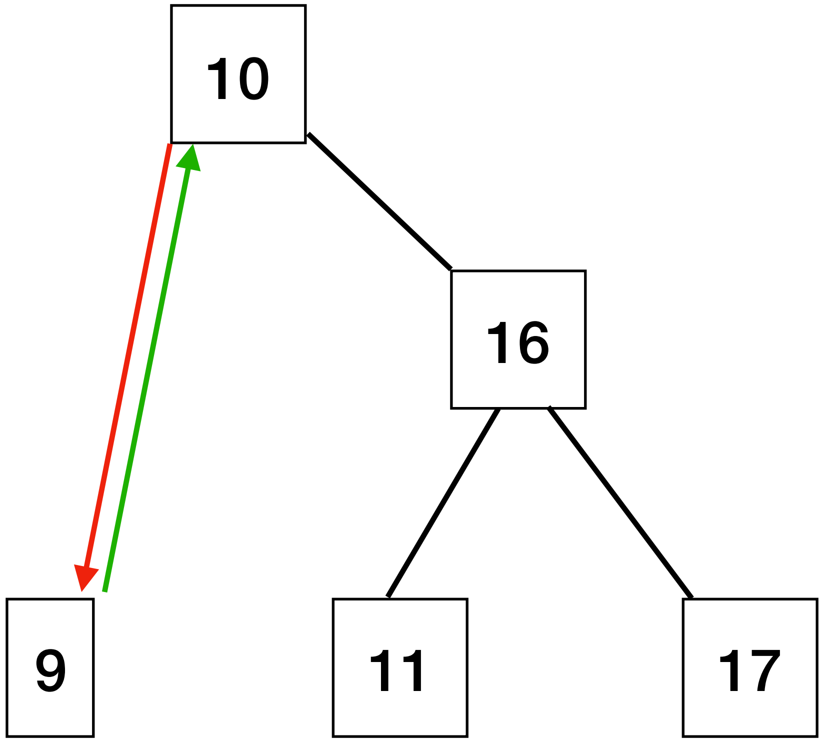


```
if (n has exactly one child)
   replace parent's child with n's child
   replace n's child's parent with n's parent
```

Deleting a node from a BST: Case 2

Three possible cases:

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

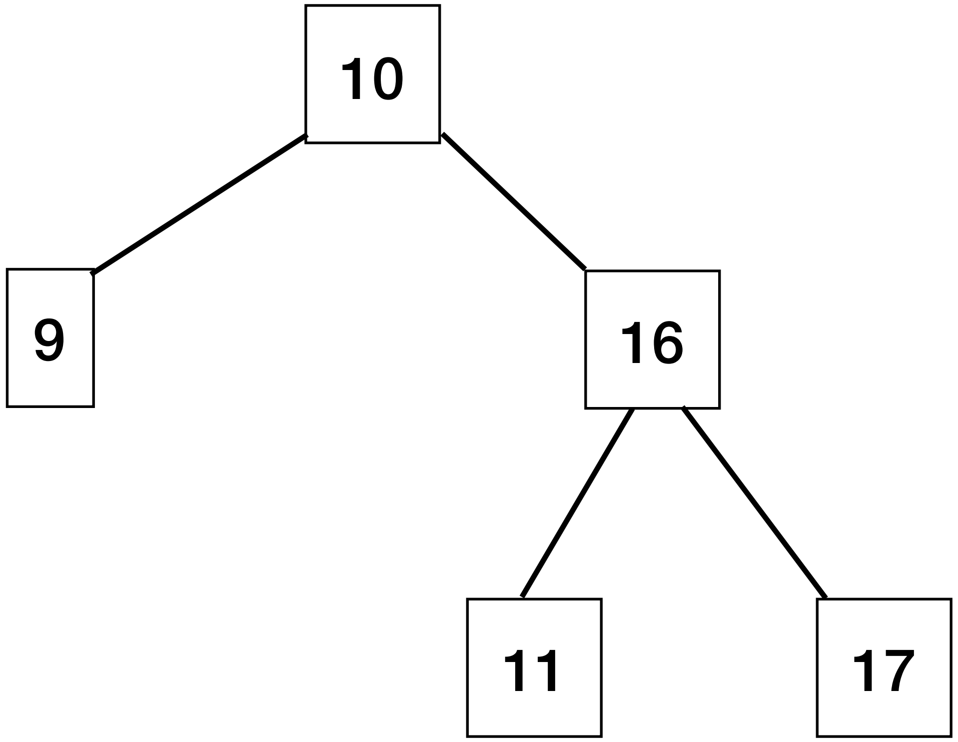


```
if (n has exactly one child)
   replace parent's child with n's child
   replace n's child's parent to n's parent
```

Deleting a node from a BST: Case 2

Three possible cases:

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



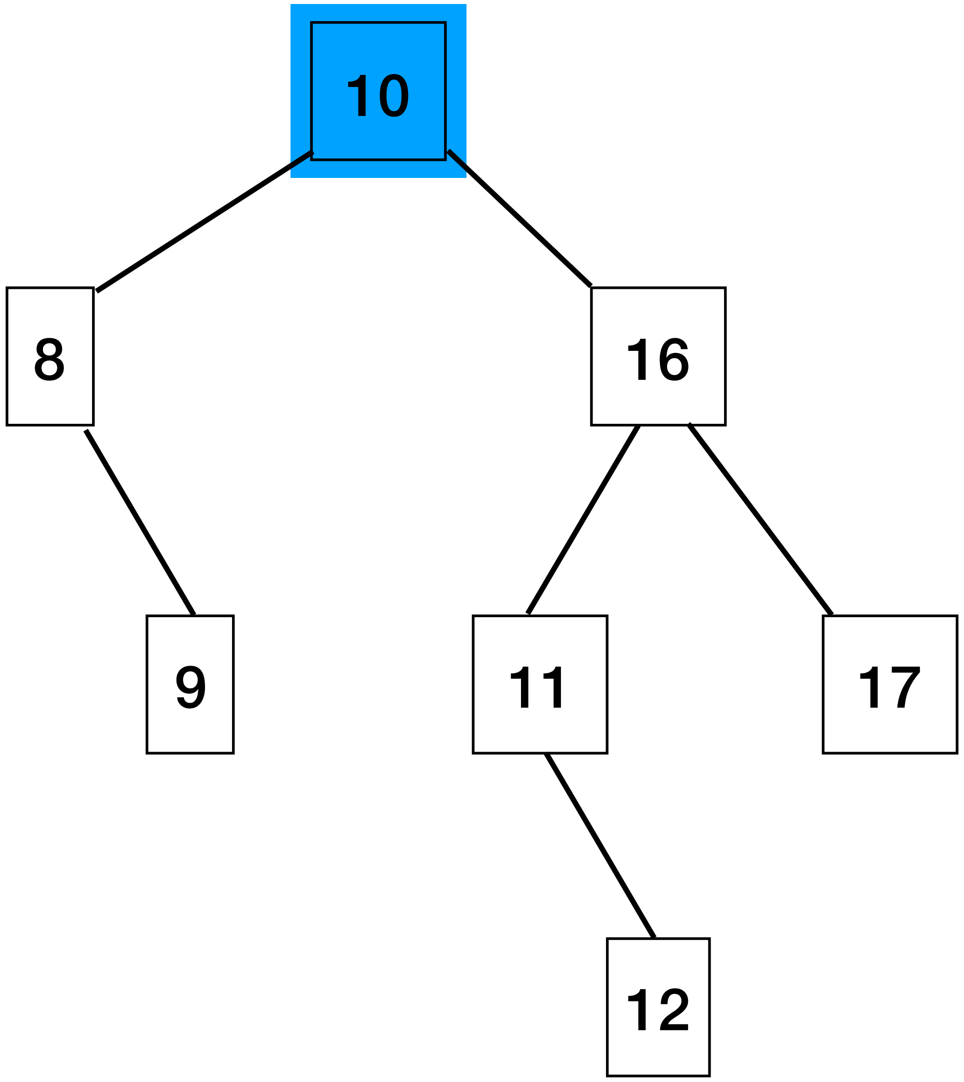
```
if (n has exactly one child)
   replace parent's child with n's child
   replace n's child's parent to n's parent
```

Deleting a node from a BST: Case 3

Three possible cases:

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

```
if (n has two children)
```

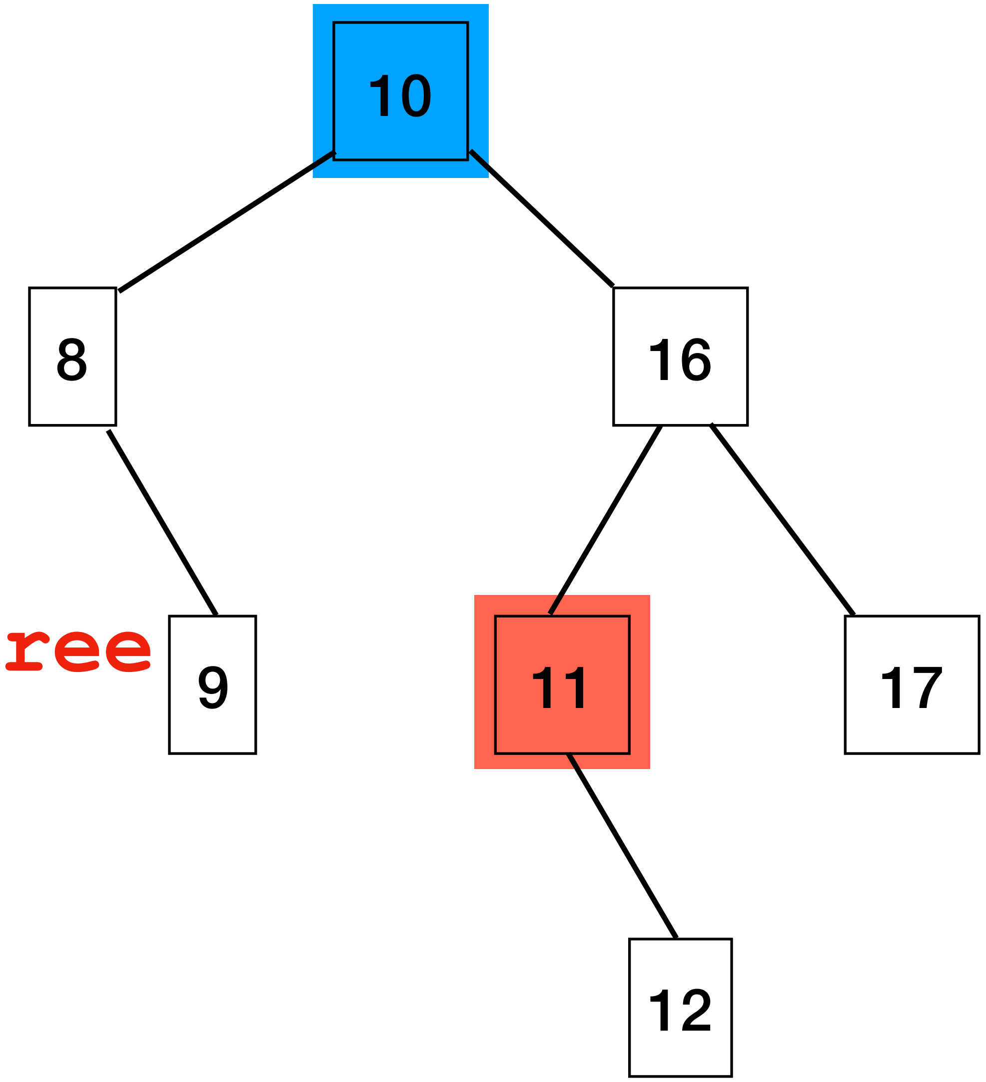


Deleting a node from a BST: Case 3

Three possible cases:

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

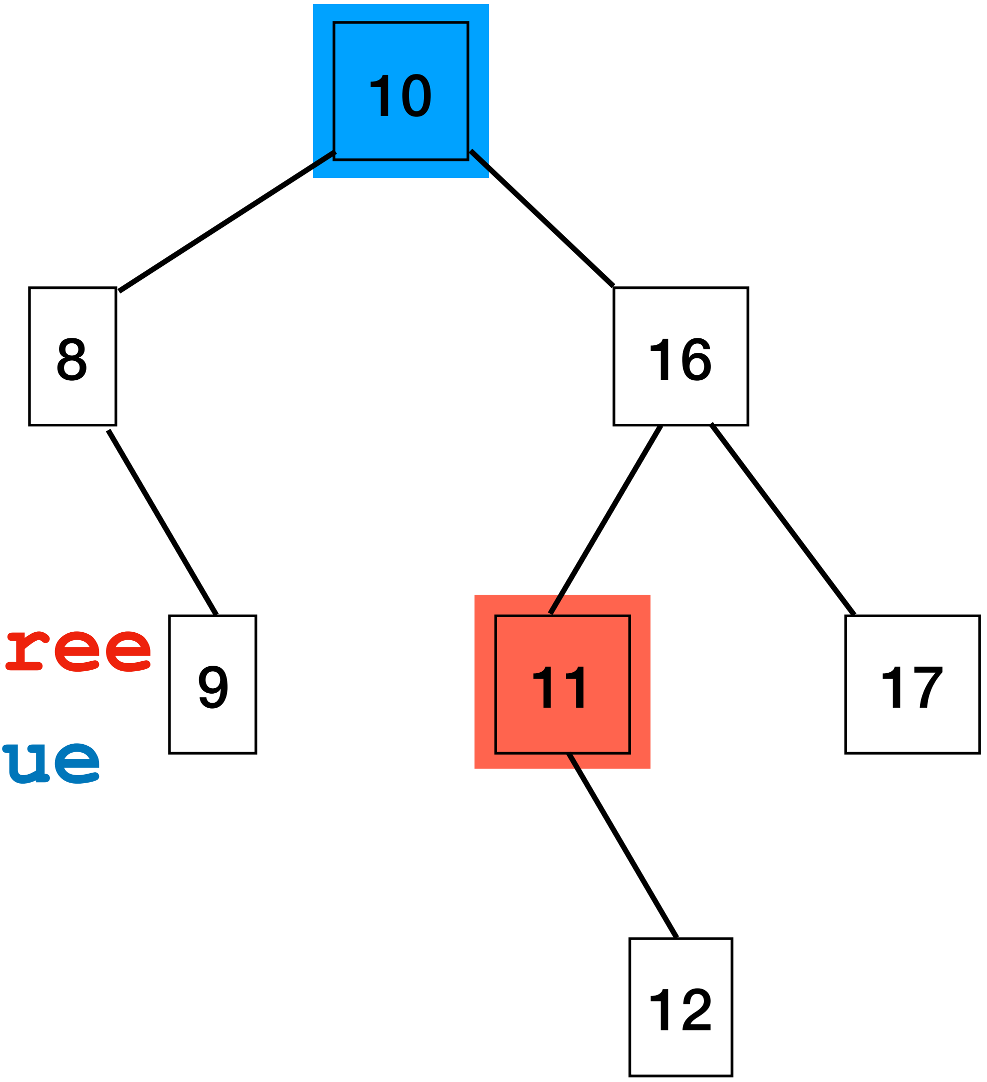
```
if (n has two children)
   let k = min node in right subtree
```



Deleting a node from a BST: Case 3

Three possible cases:

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

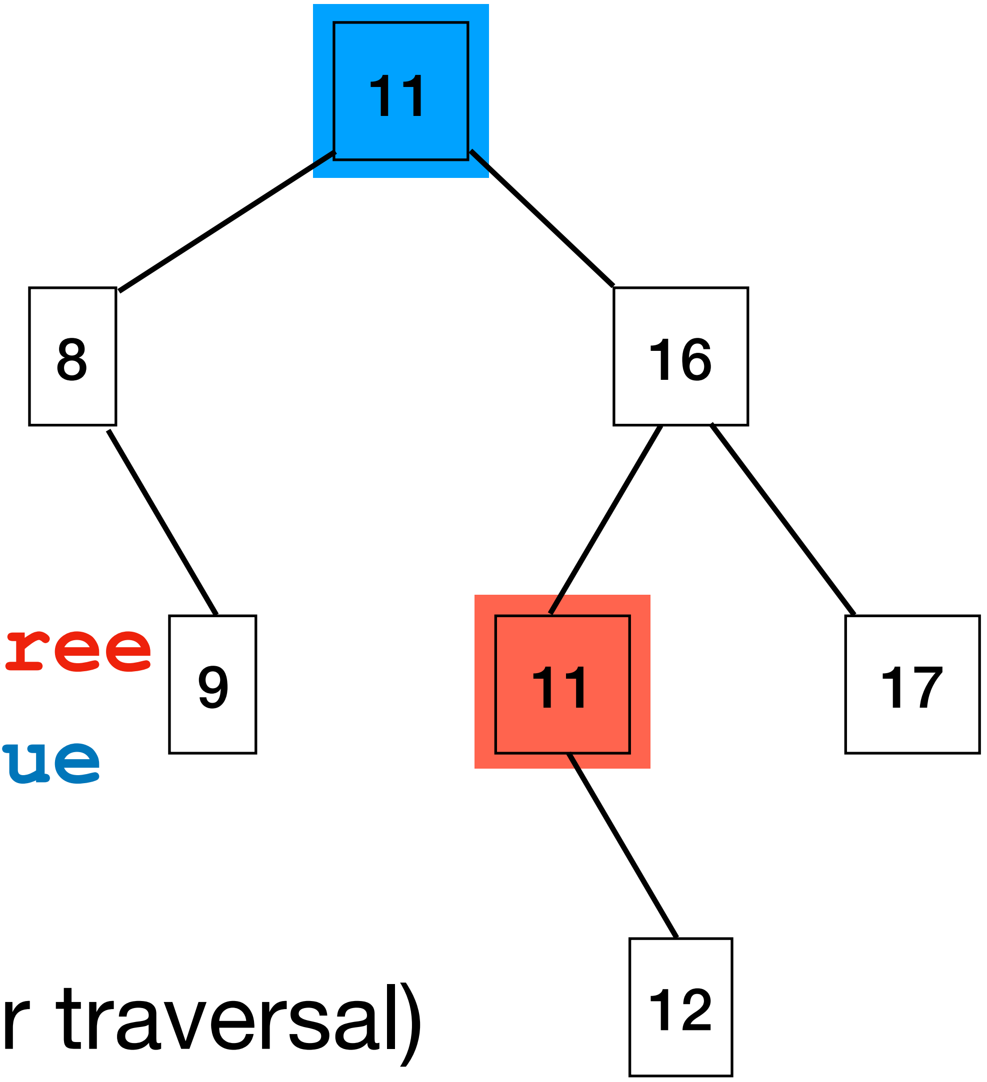


```
if (n has two children)
   let k = min node in right subtree
   replace n's value with k's value
```

Deleting a node from a BST: Case 3

Three possible cases:

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



```
if (n has two children)
  let k = min node in right subtree
  replace n's value with k's value
```

Can we do that?

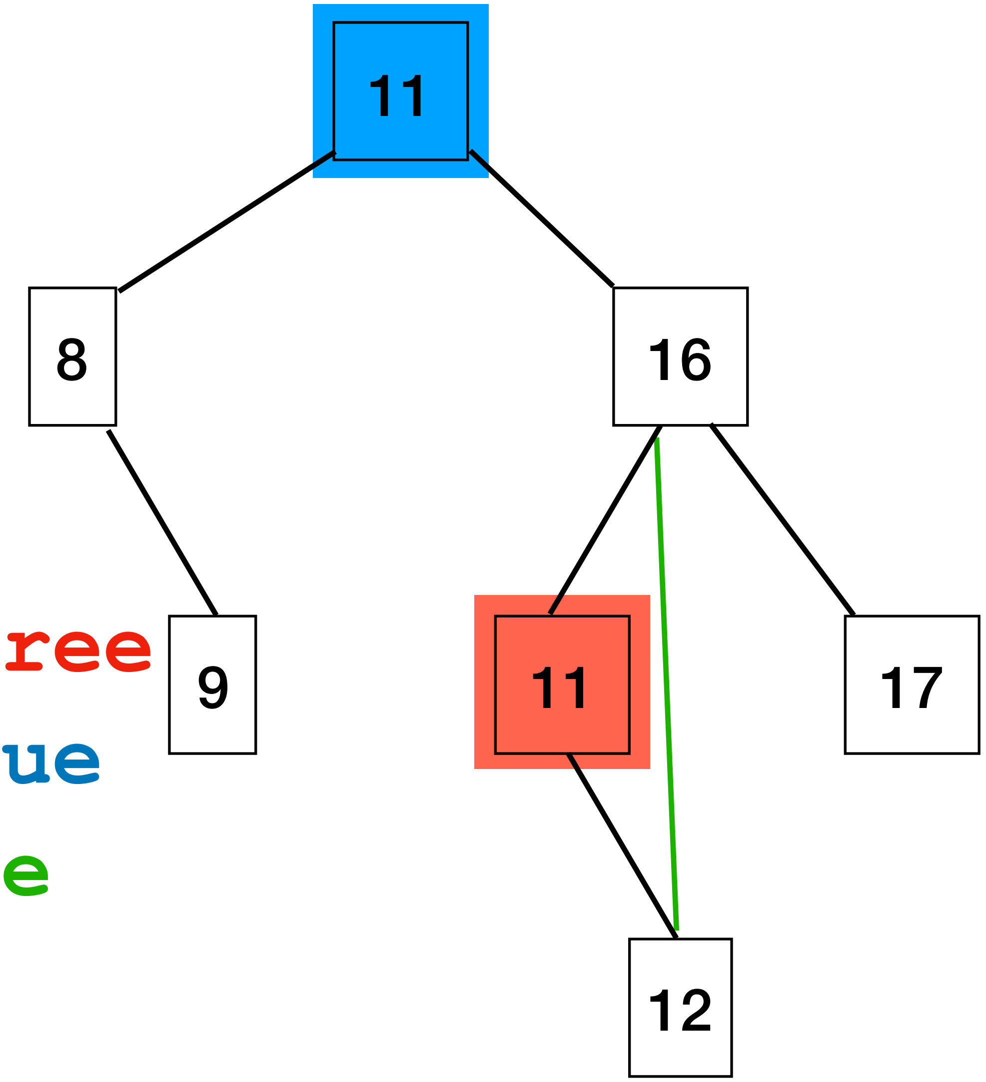
- k is n's **successor** (next in an in-order traversal)
- Everything *else* in n's right subtree is bigger than it
- Everything in n's left subtree is smaller than it
- k's value can safely replace n's…but now we have a duplicate.

Deleting a node from a BST: Case 3

Three possible cases:

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

```
if (n has two children)
   let k = min node in right subtree
   replace n's value with k's value
   remove k from n's right subtree
```

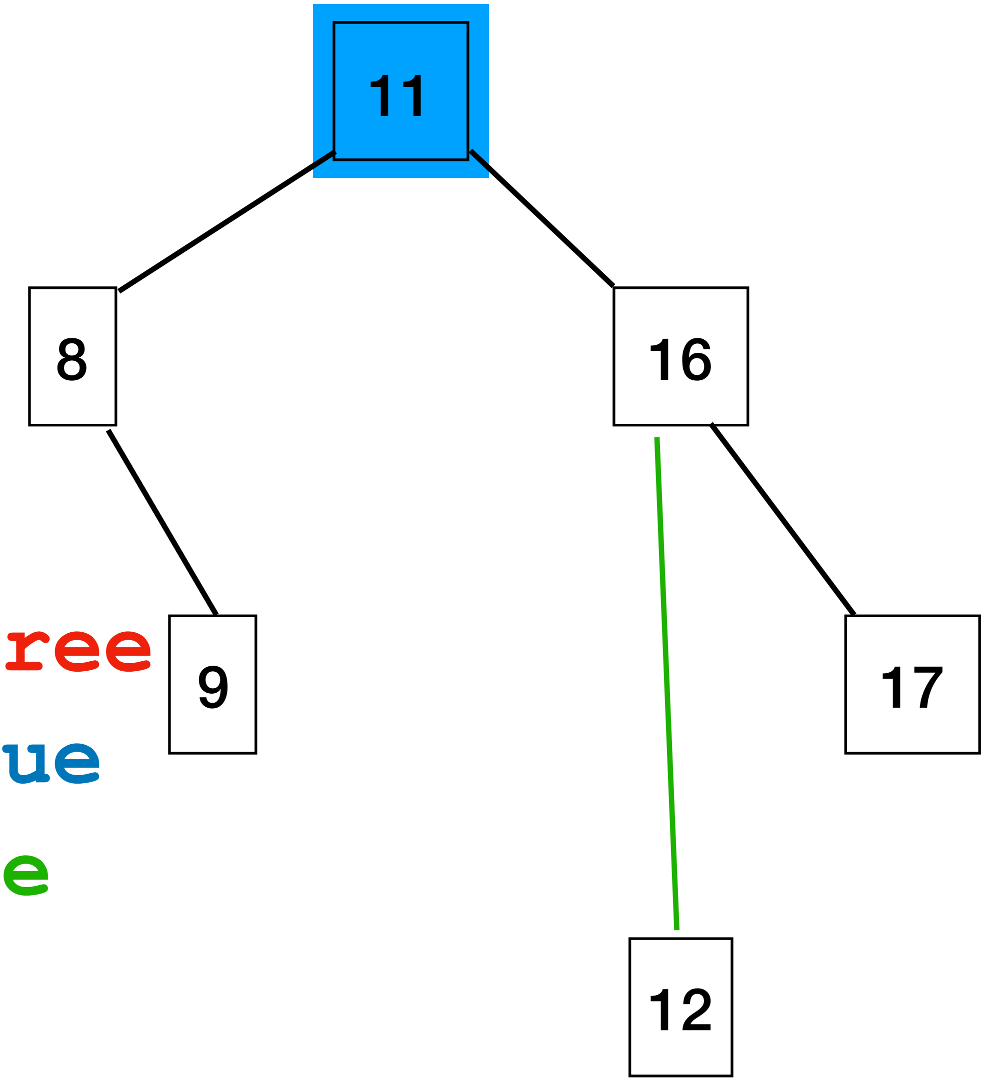


Deleting a node from a BST: Case 3

Three possible cases:

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

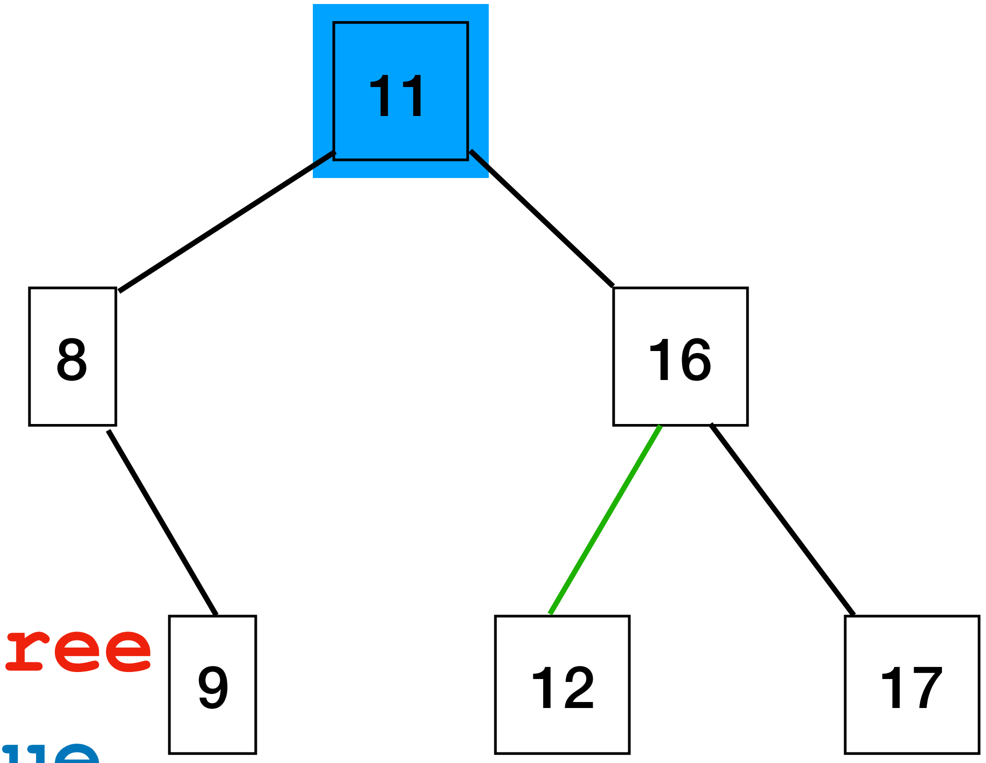
```
if (n has two children)
   let k = min node in right subtree
   replace n's value with k's value
   remove k from n's right subtree
```



Deleting a node from a BST: Case 3

Three possible cases:

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

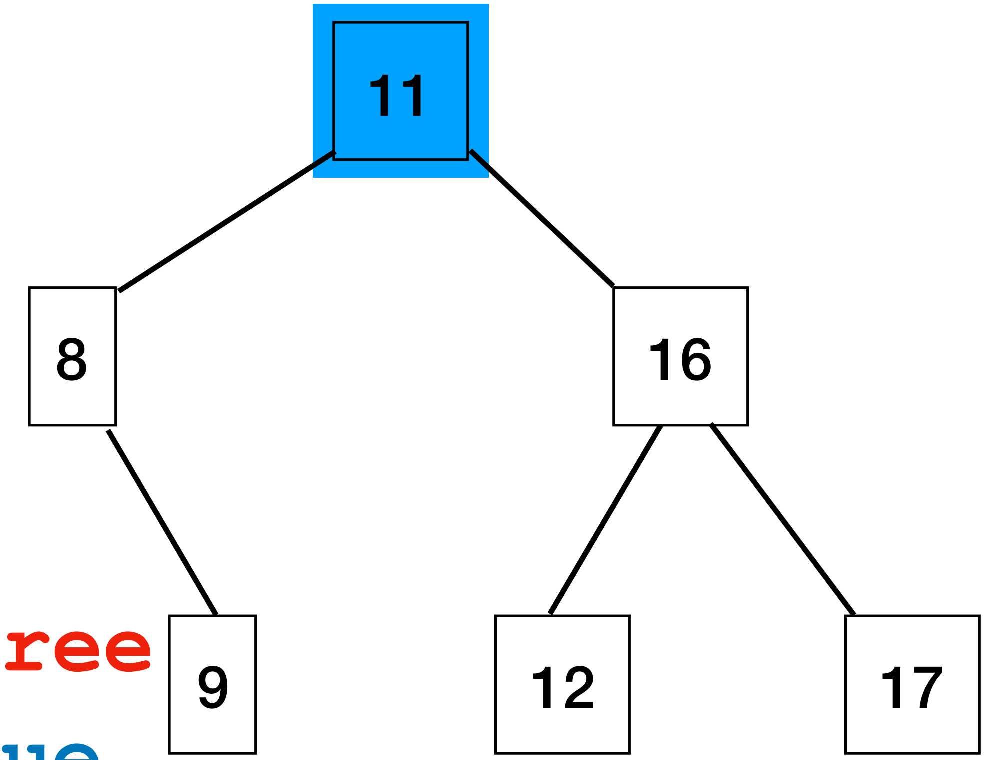


```
if (n has two children)
   let k = min node in right subtree
   replace n's value with k's value
   remove k from n's right subtree (recursively!)
```

Deleting a node from a BST: Case 3

Three possible cases:

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



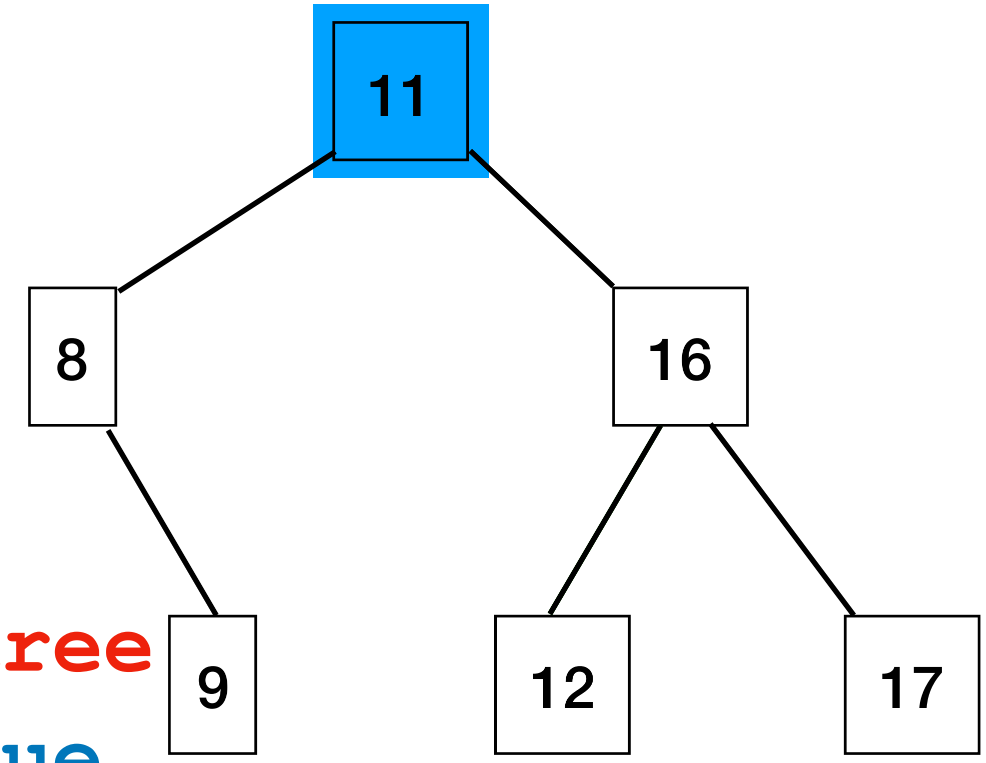
```
if (n has two children)
   let k = min node in right subtree
   replace n’s value with k’s value
   remove k from n’s right subtree
```

this **has to be** either Case 1 or Case 2!

Deleting a node from a BST: Case 3

Three possible cases:

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



```
if (n has two children)
   let k = min node in right subtree
   replace n's value with k's value
   remove k from n's right subtree
```

this **has to be** either Case 1 or Case 2!

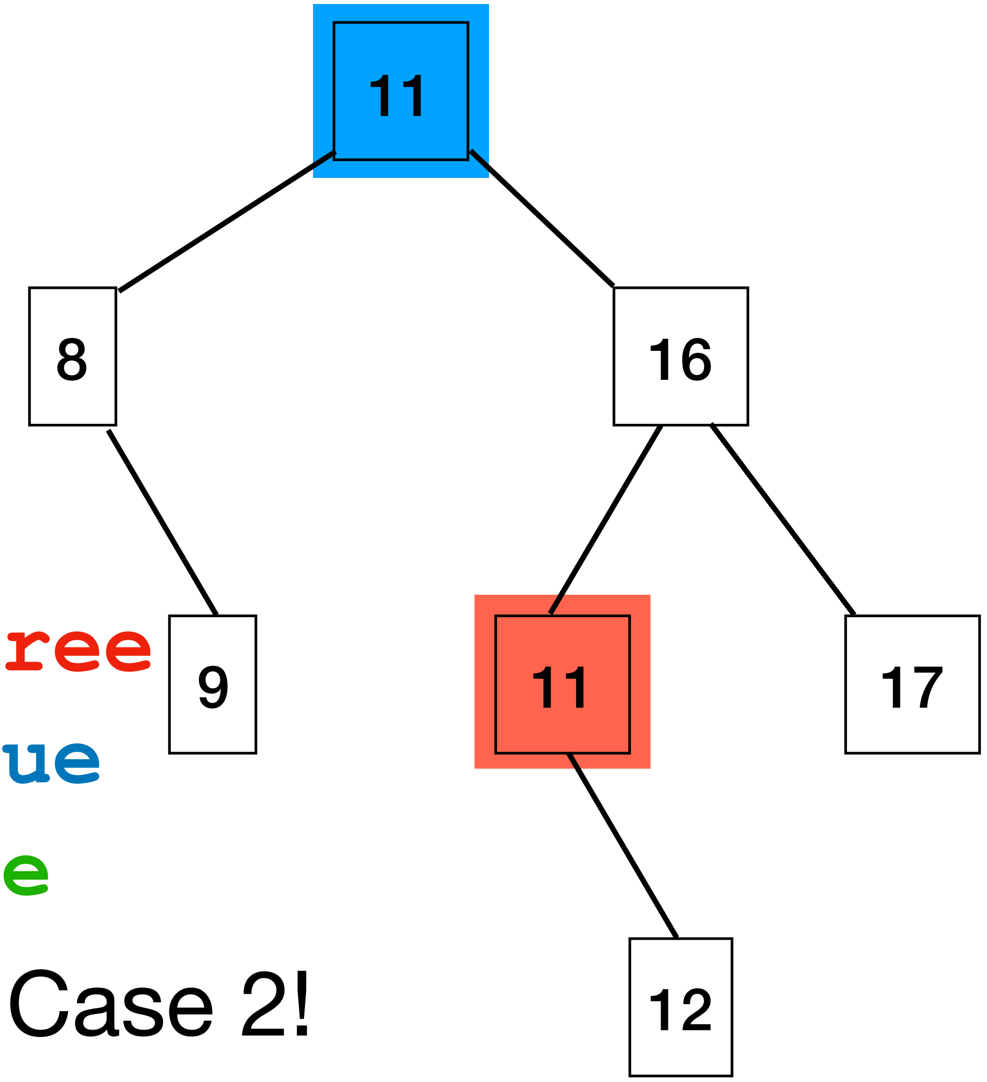
Why?

Deleting a node from a BST: Case 3

Three possible cases:

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

```
if (n has two children)
   let k = min node in right subtree
   replace n's value with k's value
   remove k from n's right subtree
```



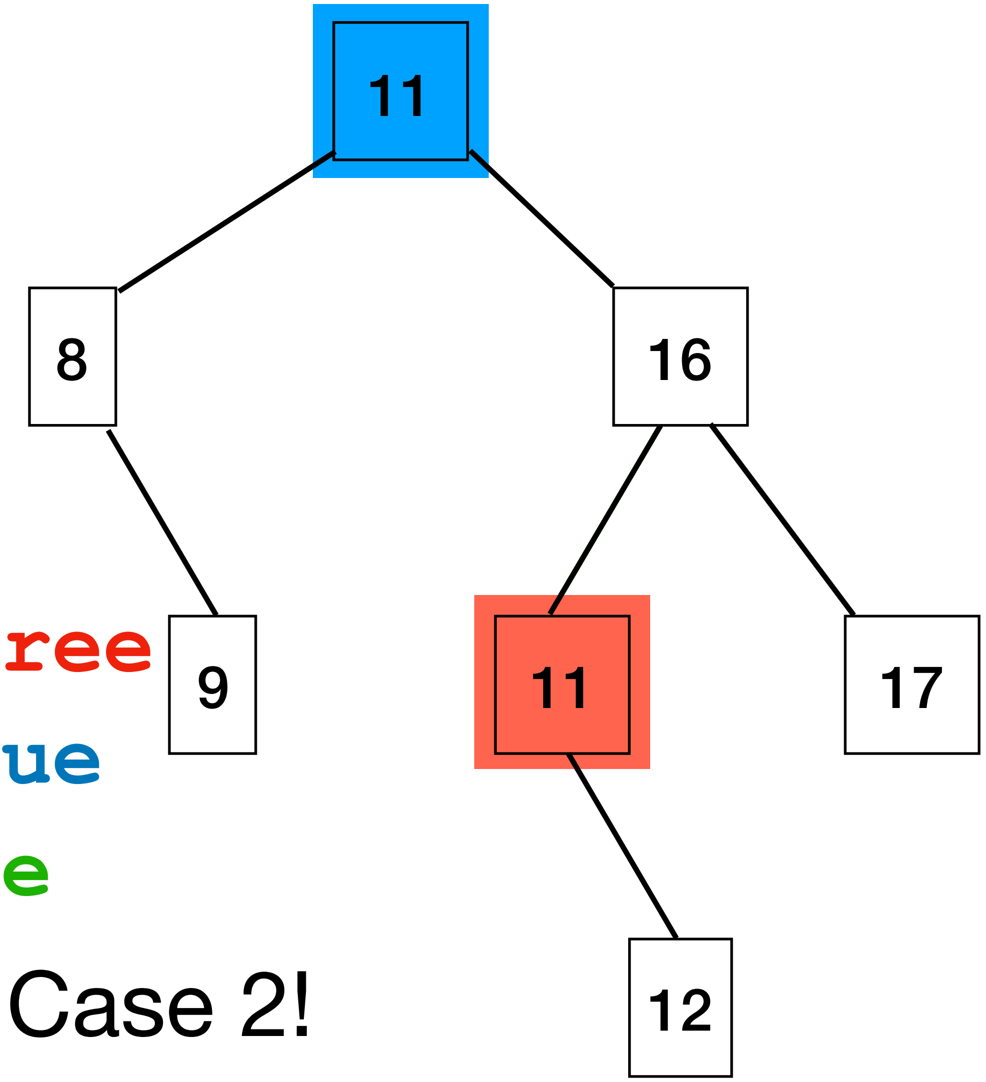
this **has to be** either Case 1 or Case 2!

Why? Rewind to before we removed it:

Deleting a node from a BST: Case 3

Three possible cases:

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



```
if (n has two children)
   let k = min node in right subtree
   replace n's value with k's value
   remove k from n's right subtree
```

this **has to be** either Case 1 or Case 2!

Why? Rewind to before we removed it:

- **k** is the smallest node in **n**'s right subtree.
- if it had a left child, that child would have to be smaller!

Details

- Need to update root pointer if root is removed.
- Often can't assume n.parent isn't null - n may be root
- To update parent's child pointer, you need to know which (L or R) child pointer to update.
- The approach presented differs from that in CLRS and some other resources.