CSCI 241

Lecture 12

Binary Search Trees: insertion and removal

Announcements

- As usual:
    - Quiz 3 today
    - Lab 4 due Sunday
    - Lab 5 out Monday
- A2 out today! Due Monday 5/11.

Announcements

- Feedback survey results

Next week: Experiment!

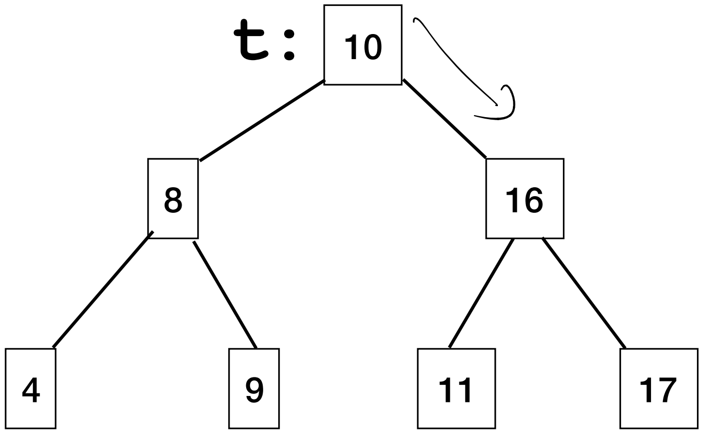
- Videos for Monday and Wednesday's lecture topics will be posted over the weekend (all out by end of Monday).
- About 5 video segments cover two lectures, not totaling more than 100 minutes.
- I will also provide practice exercises for each segment.
- M&W class periods (attendance optional): Q&A, exercise solutions, more exercises.
    - It's not office hours though - no code help.

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:
    - search
    - insert
    - remove

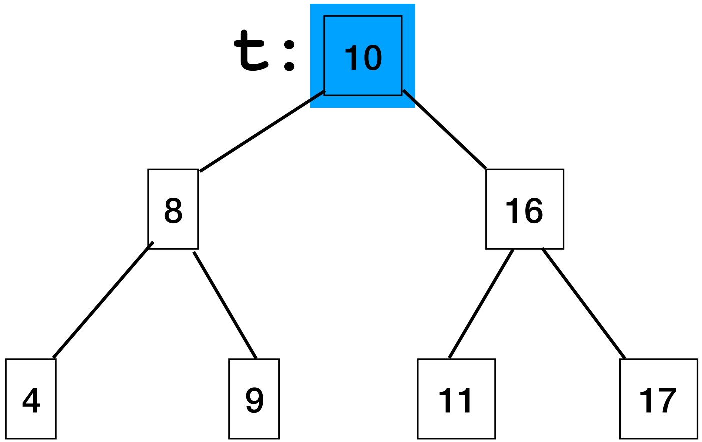
Inserting into a BST

Inserting into a BST



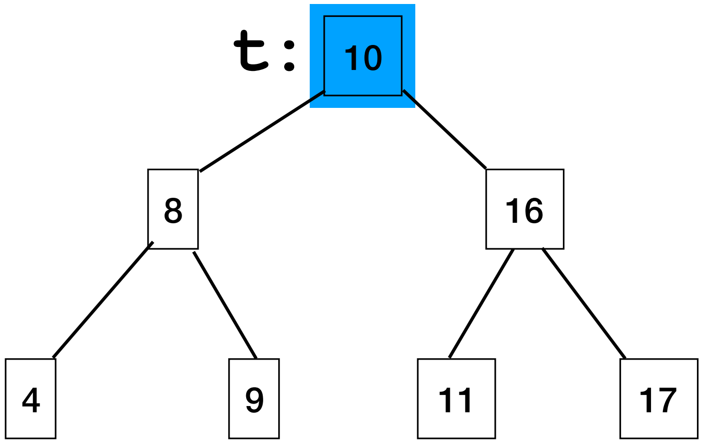
`insert(t, 11)`

Inserting into a BST



`insert(t, 11)`

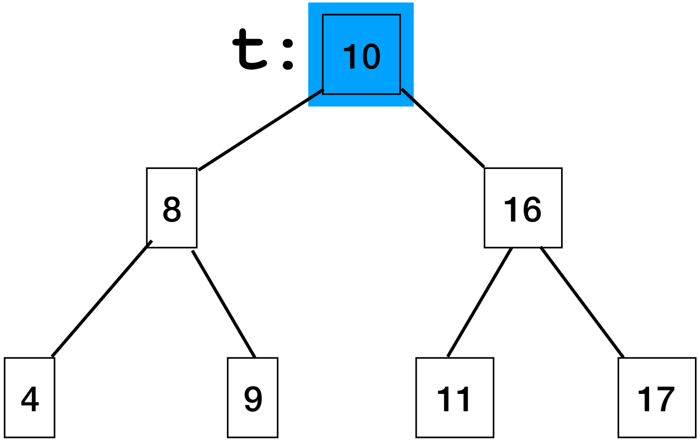
Inserting into a BST



`insert(t, 11)`

11 > 10

Inserting into a BST

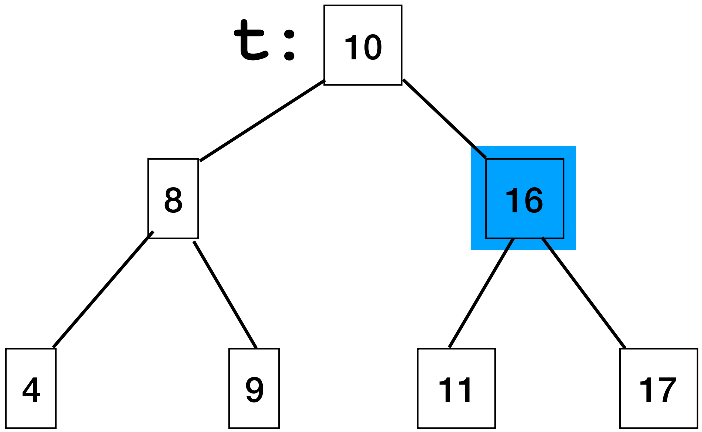


`insert(t, 11)`

11 > 10

`insert(right, 11)`

Inserting into a BST

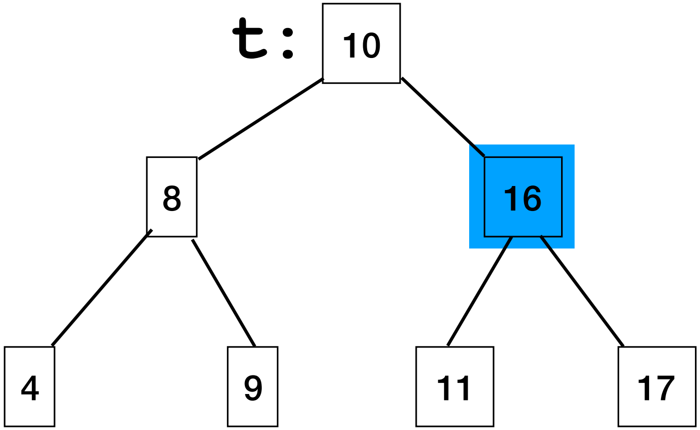


`insert(t, 11)`

11 > 10

`insert(right, 11)`

Inserting into a BST



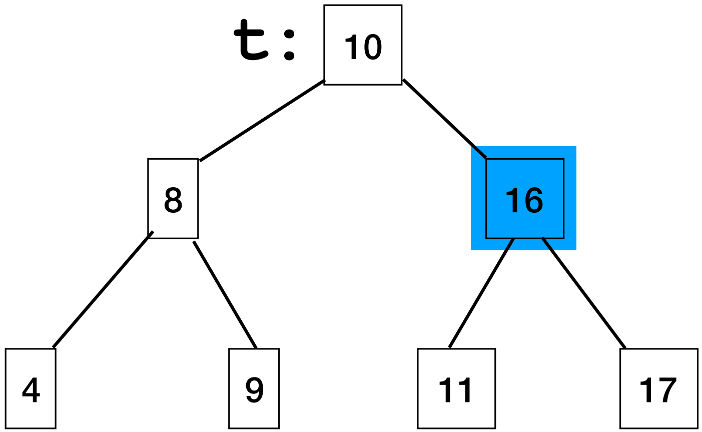
`insert(t, 11)`

11 > 10

`insert(right, 11)`

11 < 16

Inserting into a BST



`insert(t, 11)`

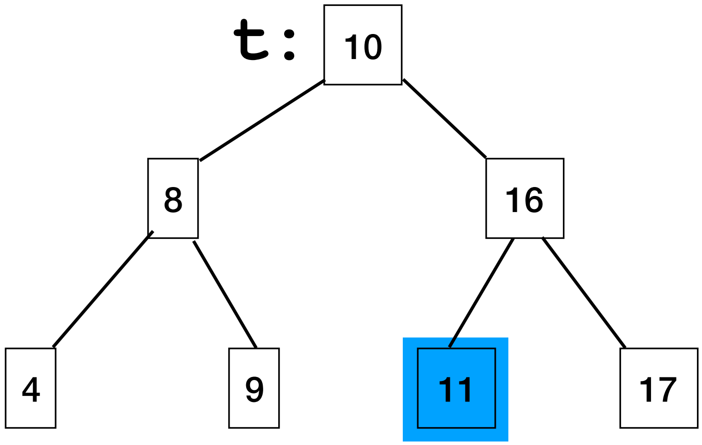
11 > 10

`insert(right, 11)`

11 < 16

`insert(left, 11)`

Inserting into a BST



`insert(t, 11)`

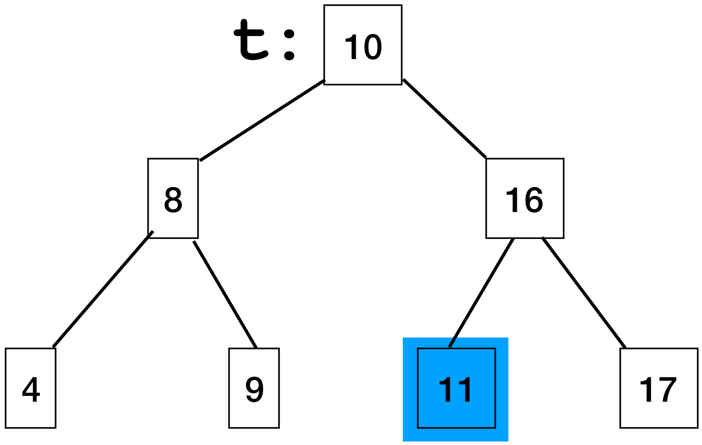
11 > 10

`insert(right, 11)`

11 < 16

`insert(left, 11)`

Inserting into a BST



`insert(t, 11)`

11 > 10

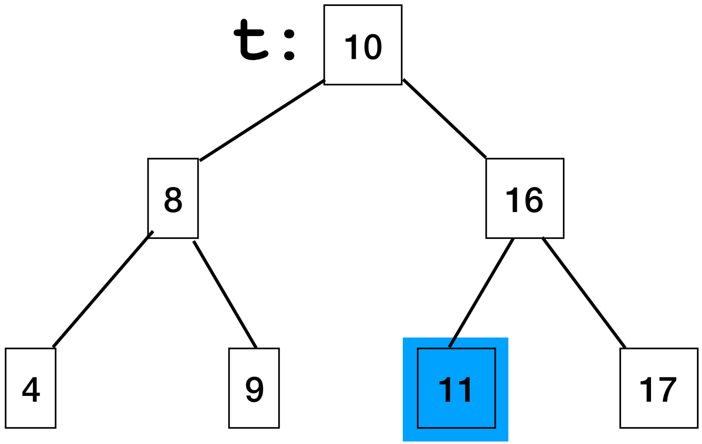
`insert(right, 11)`

11 < 16

`insert(left, 11)`

11 == 11

Inserting into a BST



`insert(t, 11)`

11 > 10

`insert(right, 11)`

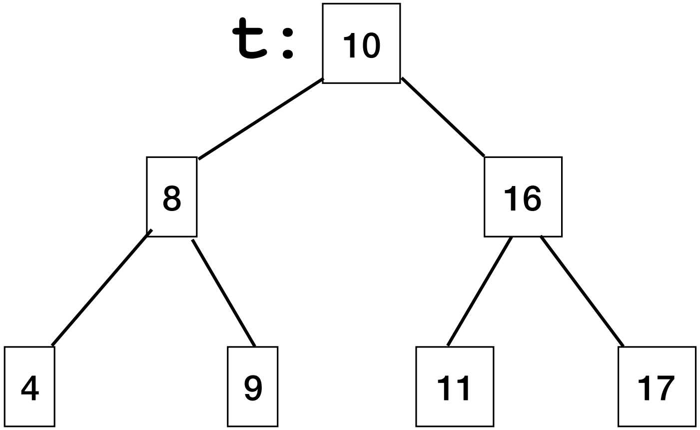
11 < 16

`insert(left, 11)`

11 == 11

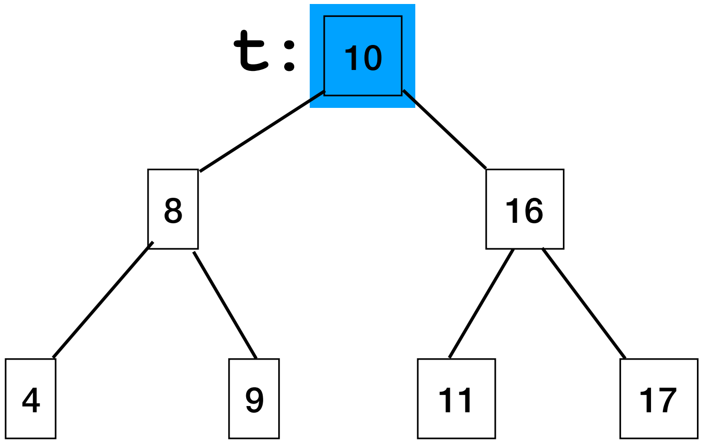
`found it! no duplicates, allowed; nothing to do. return.`

Inserting into a BST - the nonexistent case



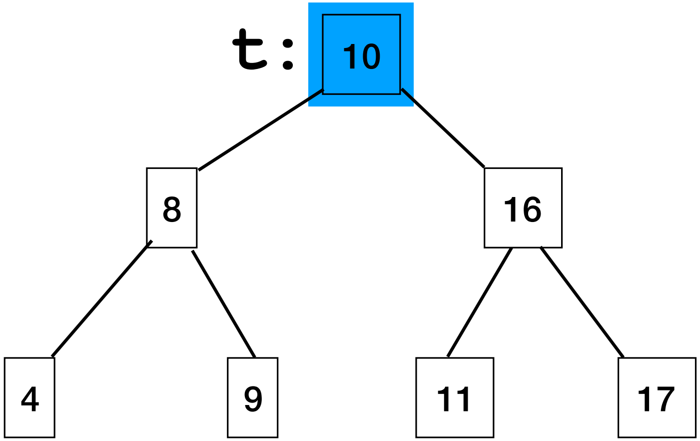
`insert(t, 5)`

Inserting into a BST - the nonexistent case



`insert(t, 5)`

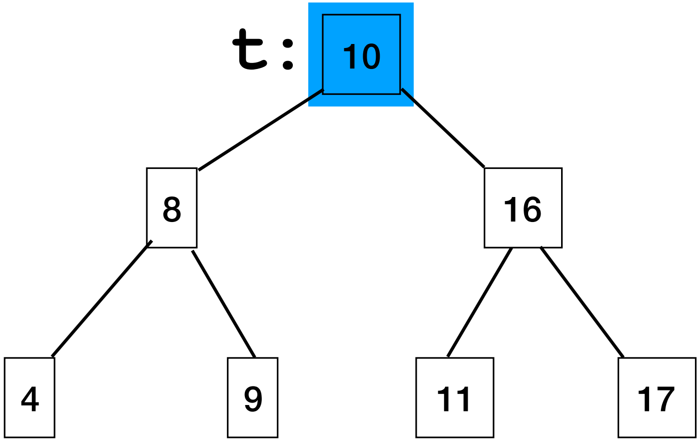
Inserting into a BST - the nonexistent case



`insert(t, 5)`

5 < 10

Inserting into a BST - the nonexistent case

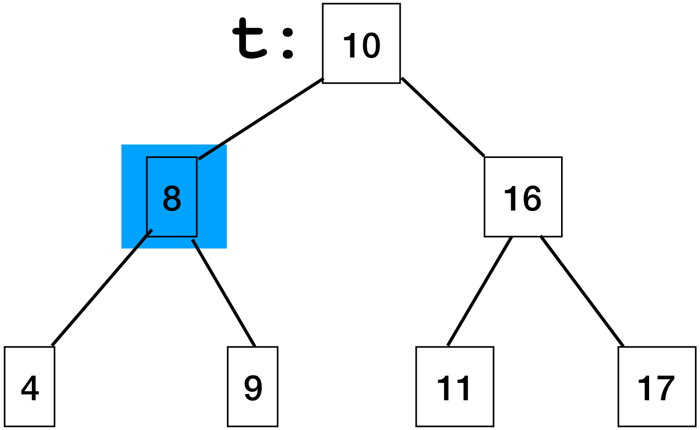


`insert(t, 5)`

$5 < 10$

`insert(left, 5)`

Inserting into a BST - the nonexistent case

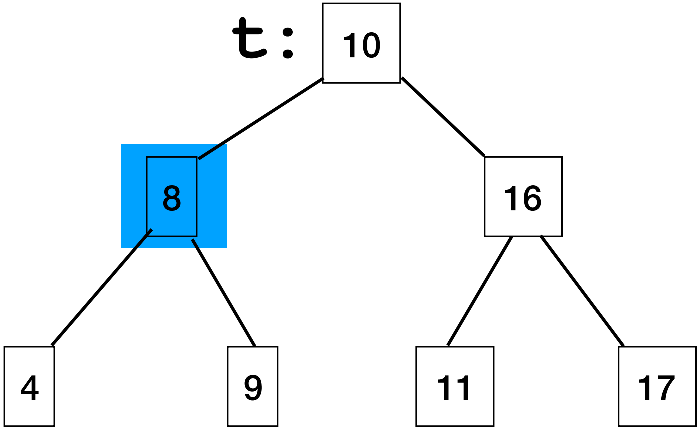


`insert(t, 5)`

$5 < 10$

`insert(left, 5)`

Inserting into a BST - the nonexistent case



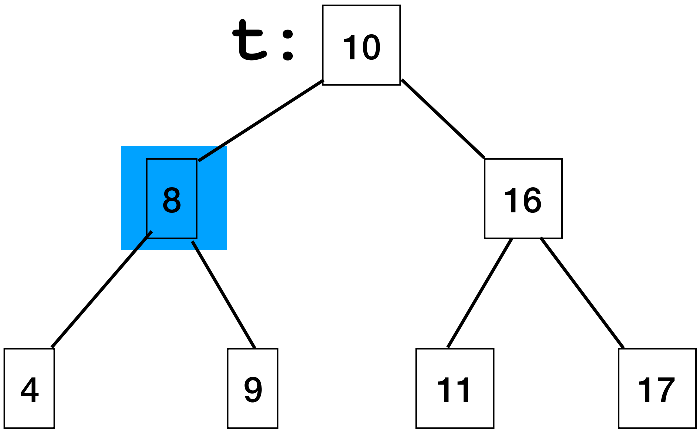
`insert(t, 5)`

5 < 10

`insert(left, 5)`

5 < 8

Inserting into a BST - the nonexistent case



`insert(t, 5)`

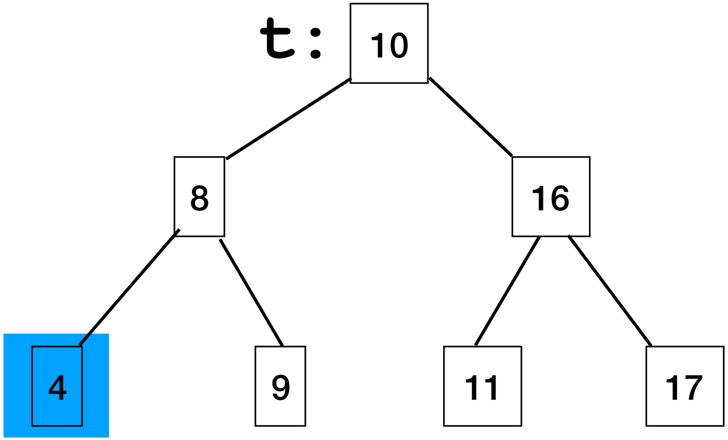
5 < 10

`insert(left, 5)`

5 < 8

`insert(left, 5)`

Inserting into a BST - the nonexistent case



`insert(t, 5)`

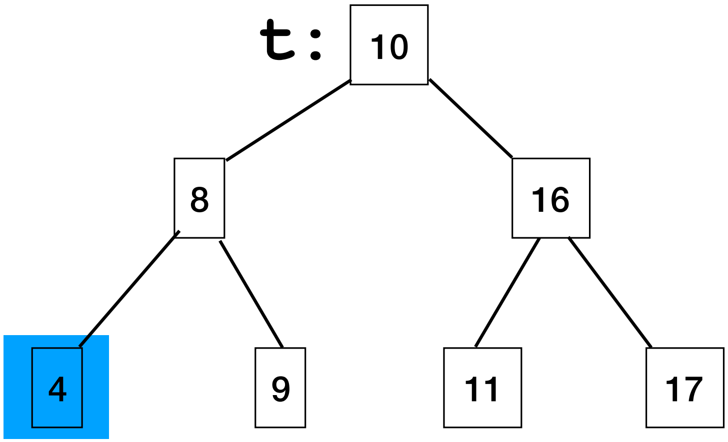
$5 < 10$

`insert(left, 5)`

$5 < 8$

`insert(left, 5)`

Inserting into a BST - the nonexistent case



`insert(t, 5)`

5 < 10

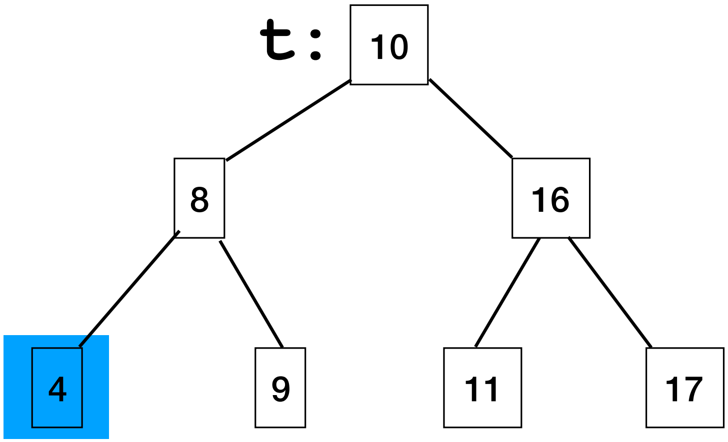
`insert(left, 5)`

5 < 8

`insert(left, 5)`

5 > 4

Inserting into a BST - the nonexistent case



`insert(t, 5)`

$5 < 10$

`insert(left, 5)`

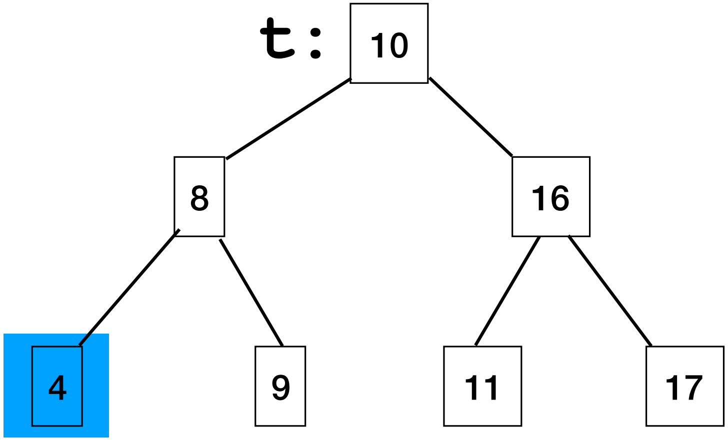
$5 < 8$

`insert(left, 5)`

$5 > 4$

`insert(right, 5)`

Inserting into a BST - the nonexistent case



`insert(t, 5)`

5 < 10

`insert(left, 5)`

5 < 8

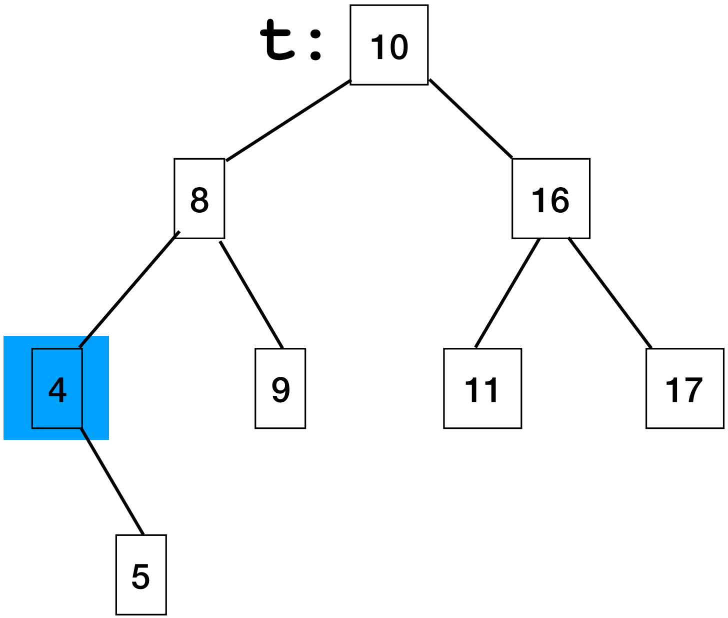
`insert(left, 5)`

5 > 4

`insert(right, 5)`

`null - not found. insert it here!`

Inserting into a BST - the nonexistent case



`insert(t, 5)`

$5 < 10$

`insert(left, 5)`

$5 < 8$

`insert(left, 5)`

$5 > 4$

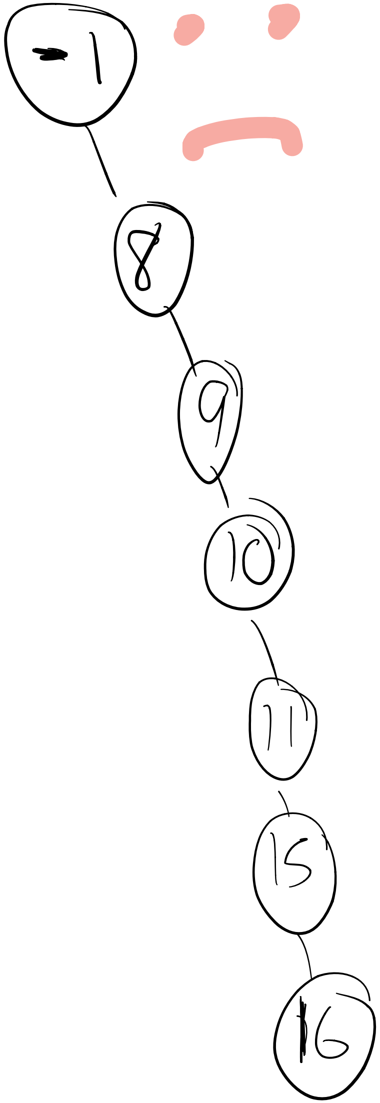
`insert(right, 5)`

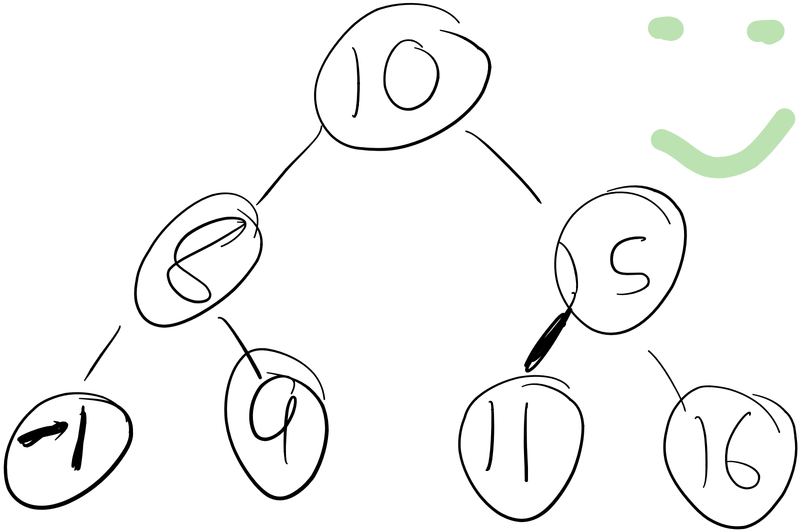
`null - not found. insert it here!`

Let’s Build Some Trees

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

```
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)
```


-1
8
9
10
11
15
16


10
8
15
-1
9
11
16

✓1. Spec

2. Base case

3. Recursive definition

4. Implement 3 with recursive calls.

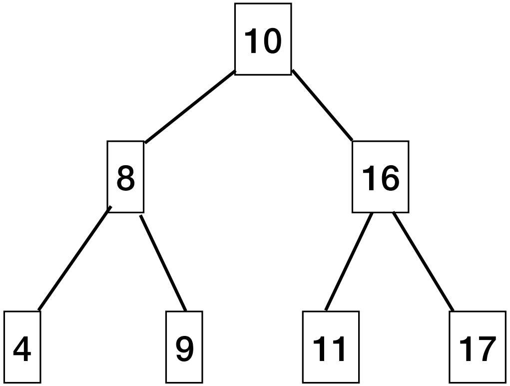
Warm-up

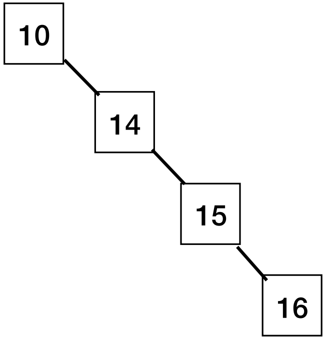
Write a method to find the smallest value in a BST:

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



}
```





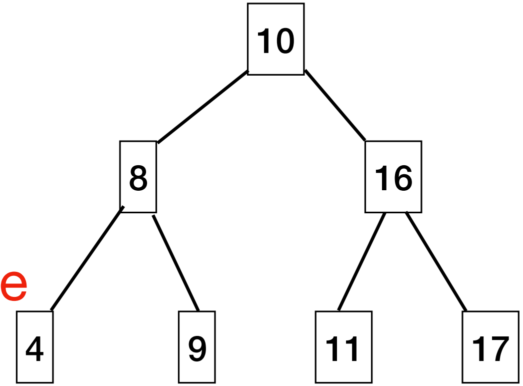
Warm-up

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

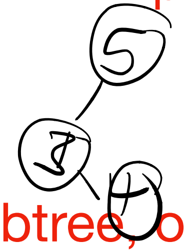
1. Spec

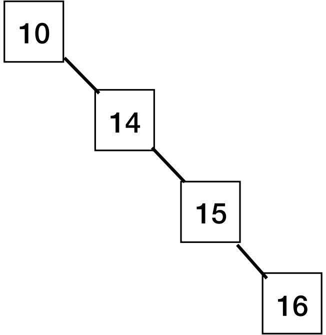
```
/** Returns min value in BST n.
  * pre: n is not null */
public int minimum(Node n) {
  if (n.left == null)
    return n.value;
  return minimum(n.left);
}
```

2. Base case



4. Implement using recursive call





3. Recursive definition:
Smallest(n) is:

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

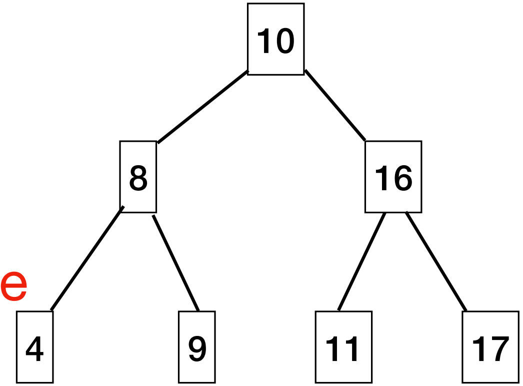
Warm-up

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

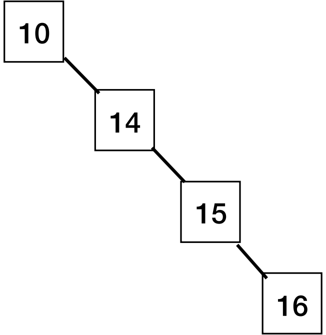
1. Spec

```
/** 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



4. Implement using recursive call



3. Recursive definition:

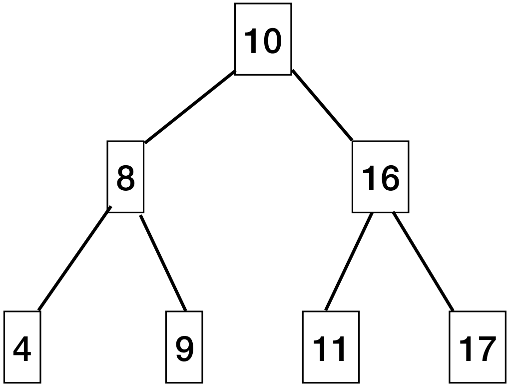
Smallest(n) is:

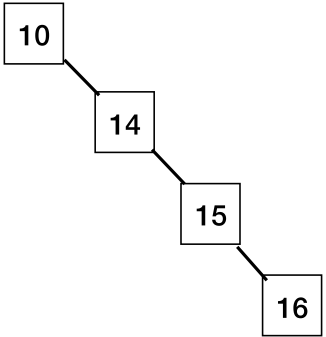
- the smallest value in the left subtree, or
- n.value if no left subtree exists.

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);
}
```

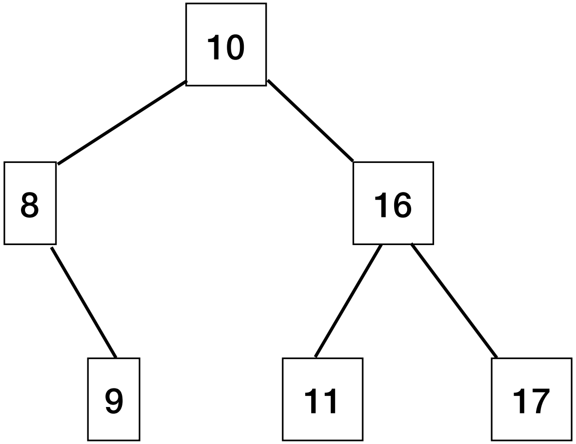




Deleting a node from a BST

Three possible cases:

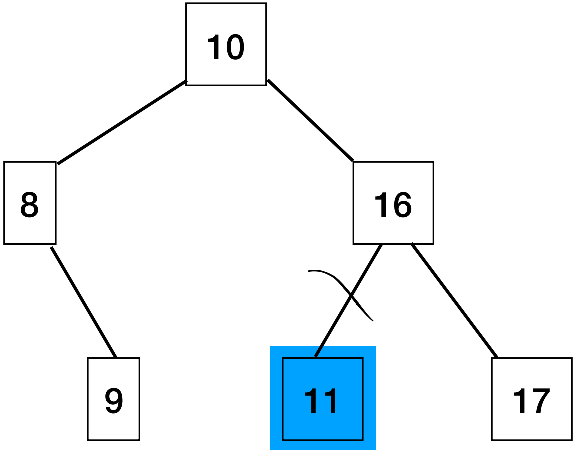
1. n has no children (is a leaf)
2. n has one child
3. n has two children



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

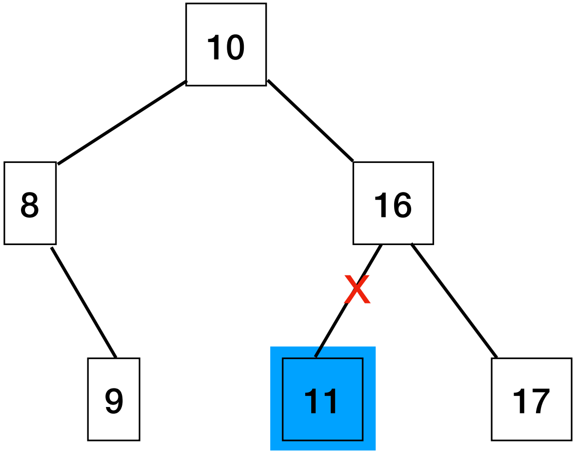


```
if (n is a leaf)
   replace parent’s child with null
```

Deleting a node from a BST: Case 1

Three possible cases:

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

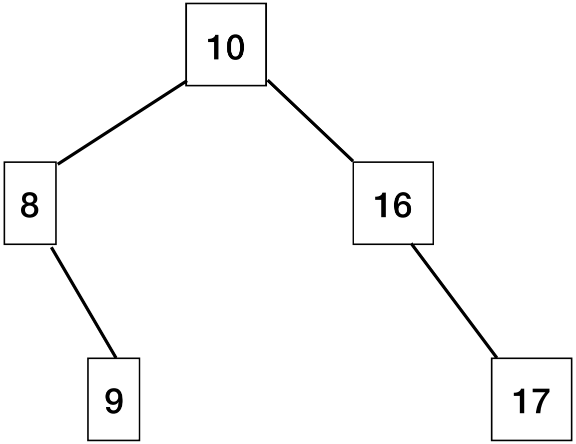


```
if (n is a leaf)
   replace parent's child with null
```

Deleting a node from a BST: Case 1

Three possible cases:

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

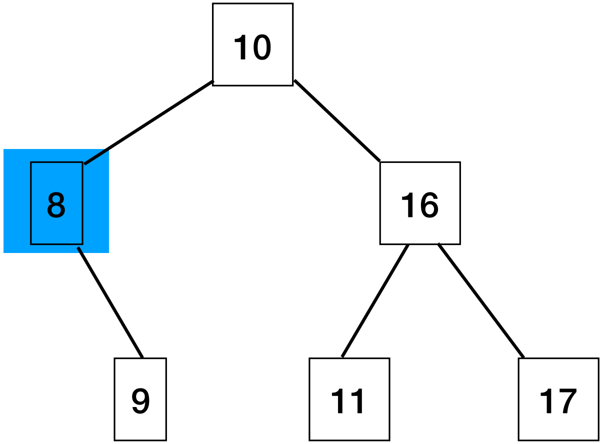


```
if (n is a leaf)
   replace parent’s child with null
```

Deleting a node from a BST: Case 2

Three possible cases:

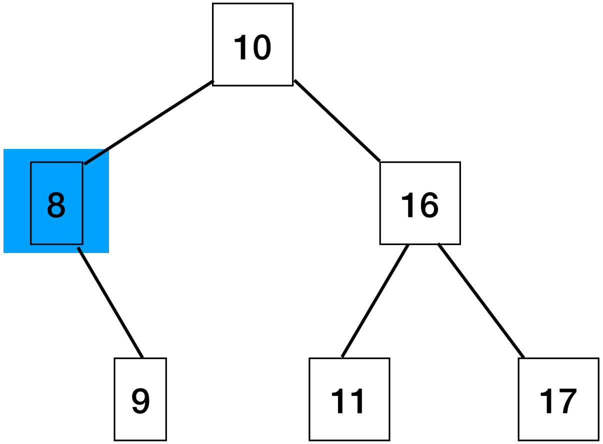
1. n has no children (is a leaf)
2. **n has one child**
3. n has two children



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

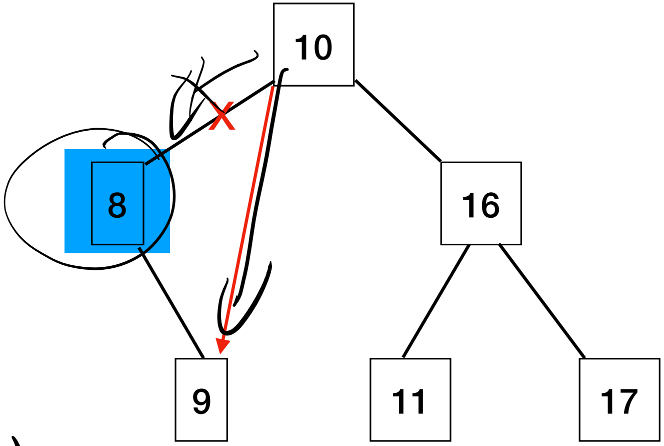


```
if (n has exactly one child)
```

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

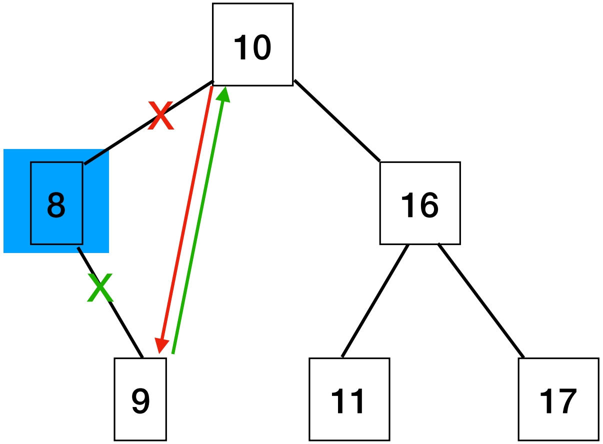


```
if (n has exactly one child)
   replace parent's child with n's child
```

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

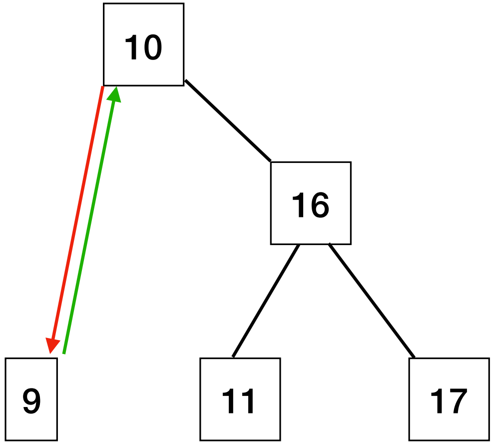


```
if (n has exactly one child)
  replace parent's child with n's child
  replace n's child's parent with n's parent
```

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

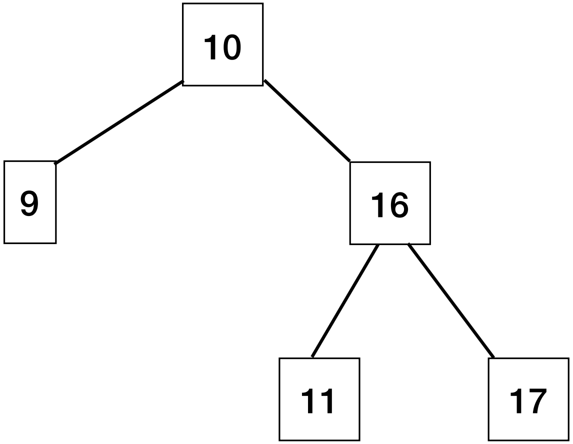


```
if (n has exactly one child)
   replace parent's child with n's child
   replace n's child's parent to n's parent
```

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



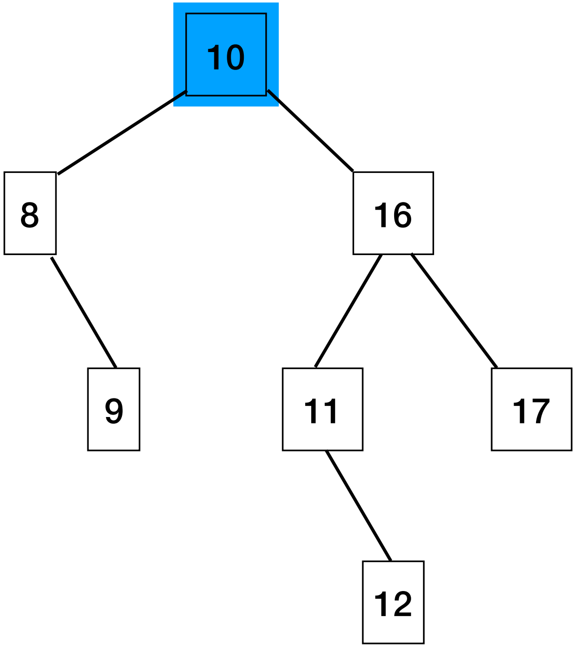
```
if (n has exactly one child)
   replace parent’s child with n’s child
   replace n’s child’s parent to n’s parent
```

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)
```

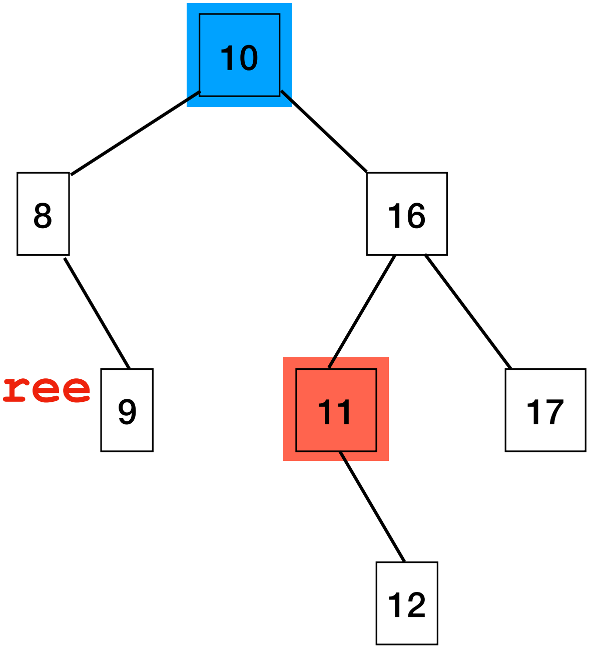


Deleting a node from a BST: Case 3

Three possible cases:

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

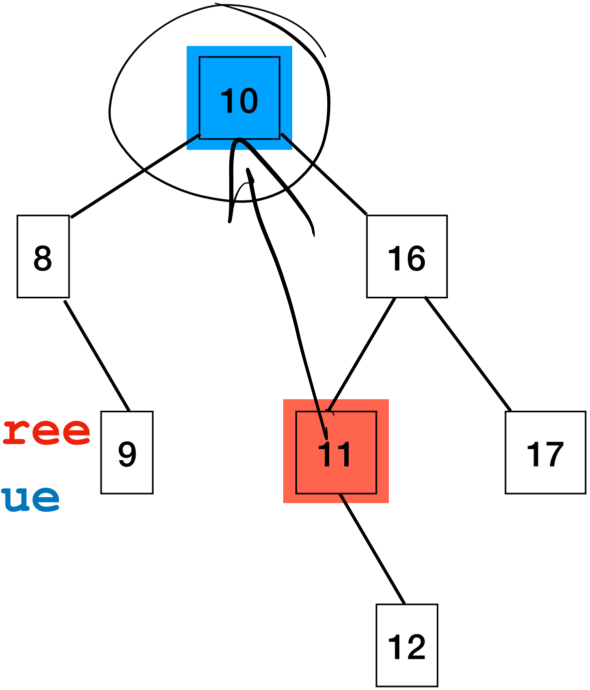
```
if (n has two children)
   let k = min node in right subtree
```



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**



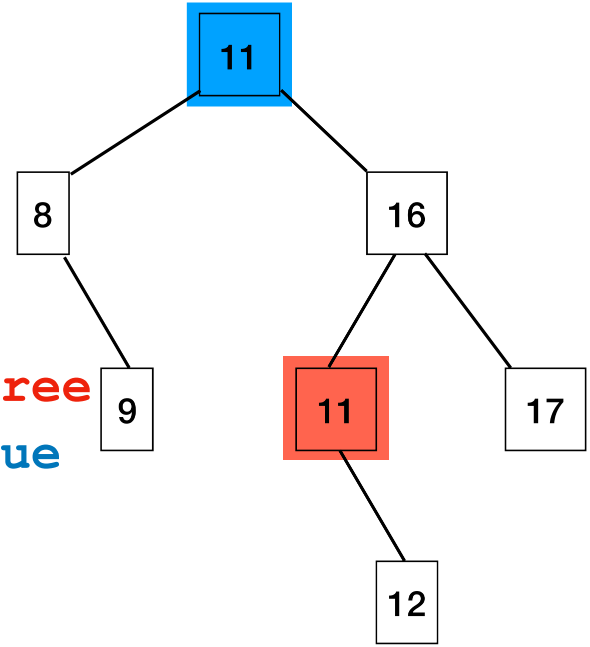
```
if (n has two children)
   let k = min node in right subtree
   replace n's value with k's value
```

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
```



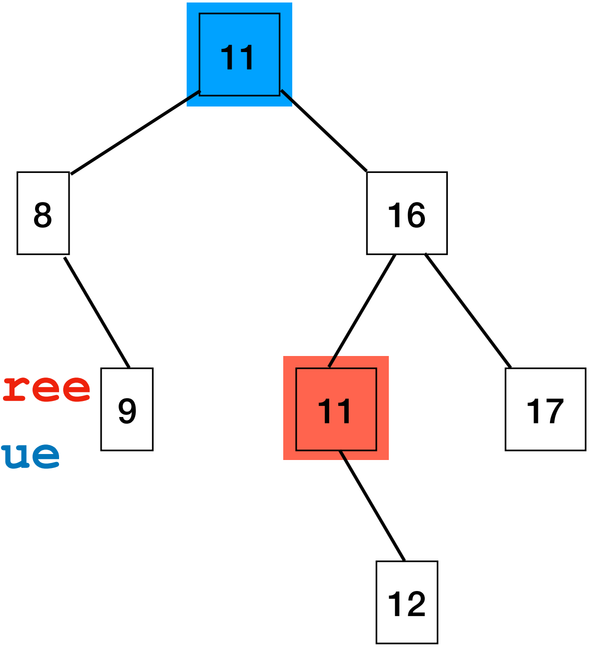
Deleting a node from a BST: Case 3

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?



Deleting a node from a BST: Case 3

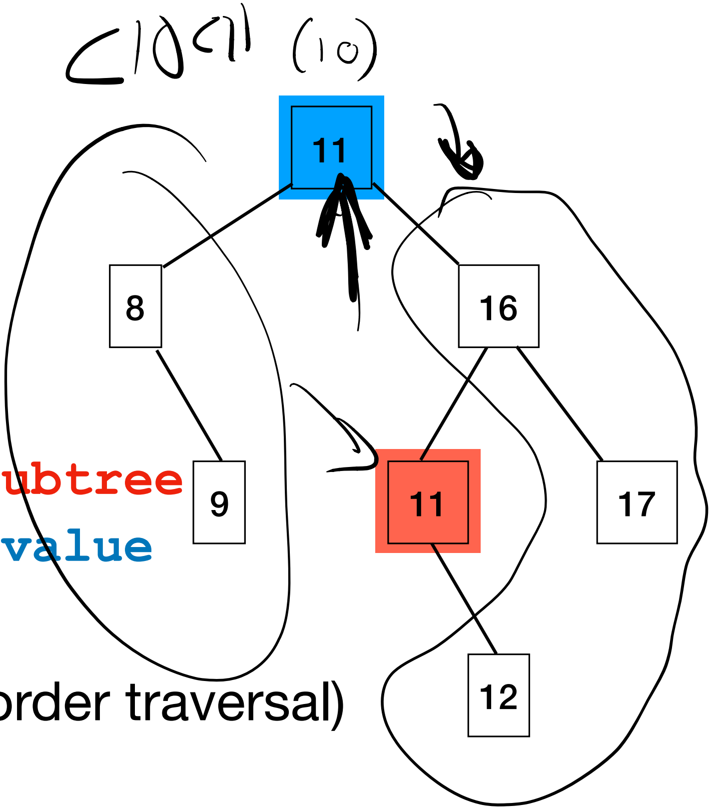
Three possible cases:

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?

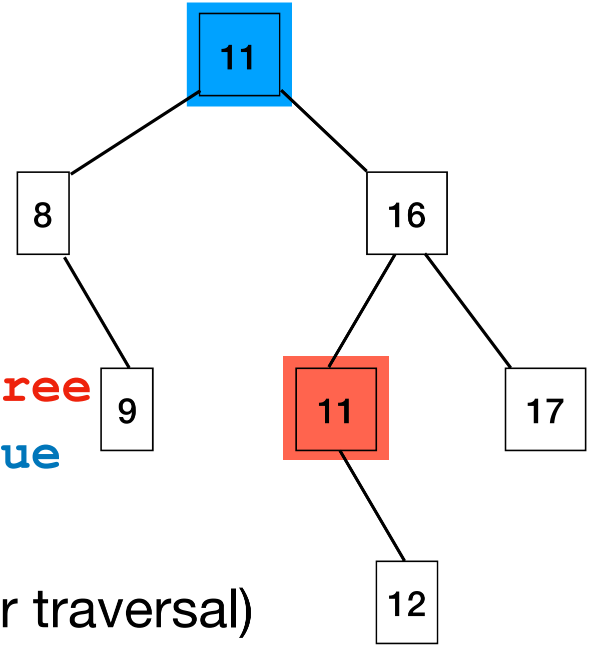
- **k** is **n**'s **successor** (next in an in-order traversal)



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
```

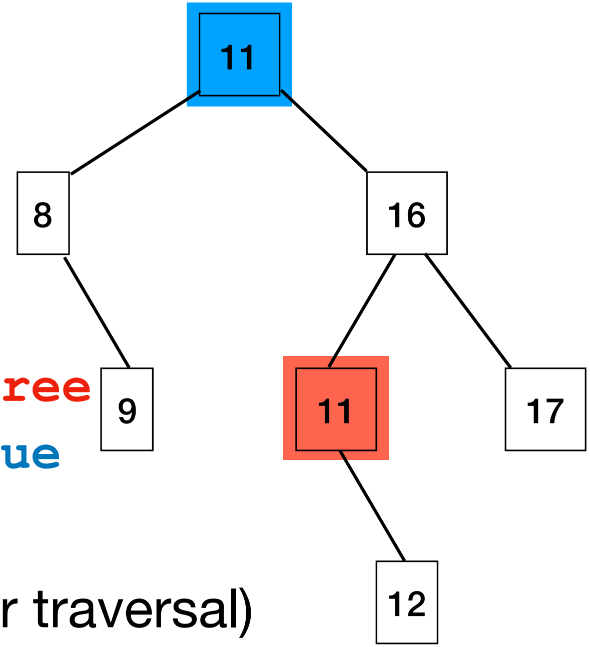
Can we do that?

- **k** is **n**'s **successor** (next in an in-order traversal)
- Everything *else* in **n**'s right subtree is bigger than it

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
```

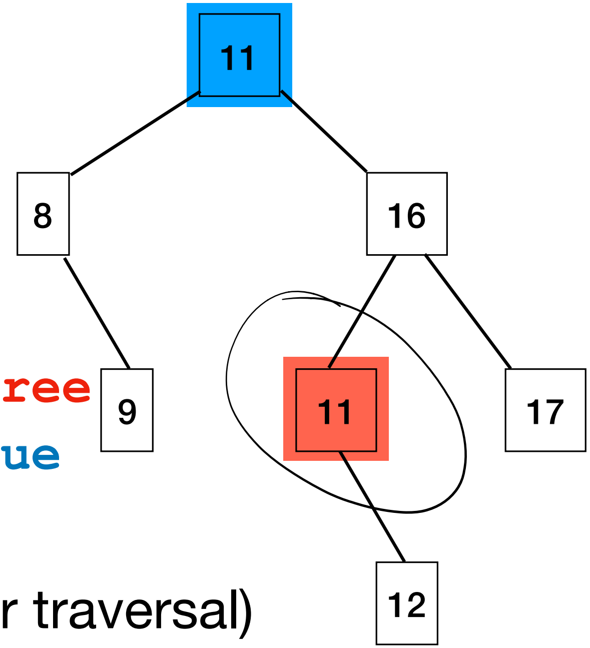
Can we do that?

- 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

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?

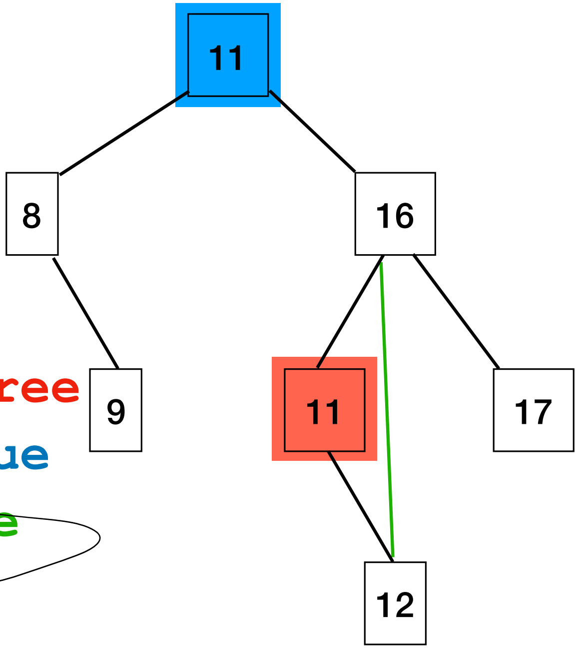
- k is n's **successor** (next in an in-order traversal)
- Everything *else* in n's right subtree is bigger than it
- Everything in n's left subtree is smaller than it
- k's value can safely replace n's…but now we have a duplicate.

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
```

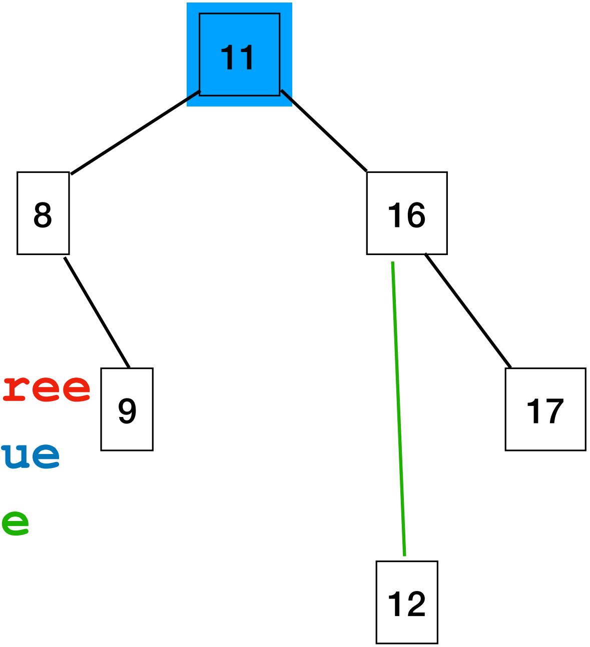


Deleting a node from a BST: Case 3

Three possible cases:

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

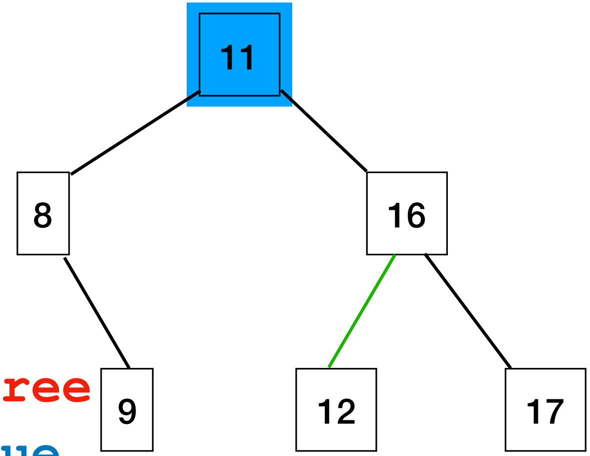
```
if (n has two children)
   let k = min node in right subtree
   replace n's value with k's value
   remove k from n's right subtree
```



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**

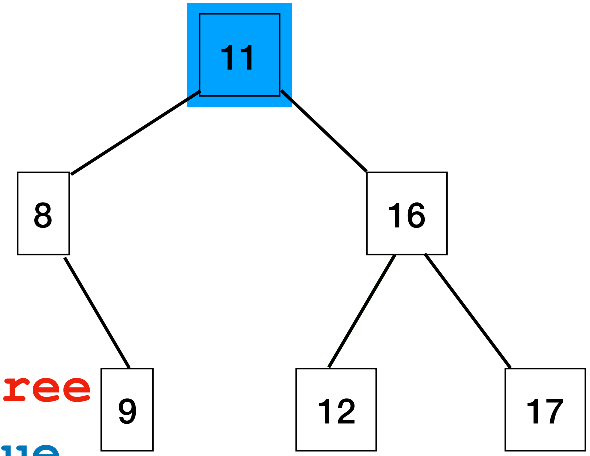


```
if (n has two children)
   let k = min node in right subtree
   replace n's value with k's value
   remove k from n's right subtree (recursively!)
```

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**



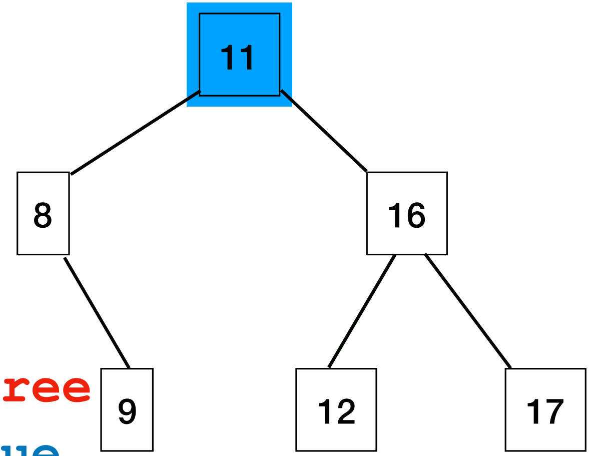
```
if (n has two children)
   let k = min node in right subtree
   replace n's value with k's value
   remove k from n's right subtree
```

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
```

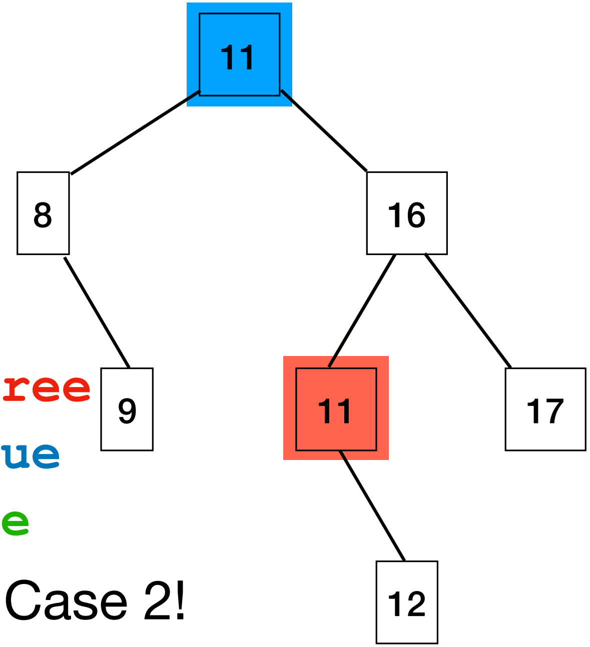
this **has to be** either Case 1 or Case 2!

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
```

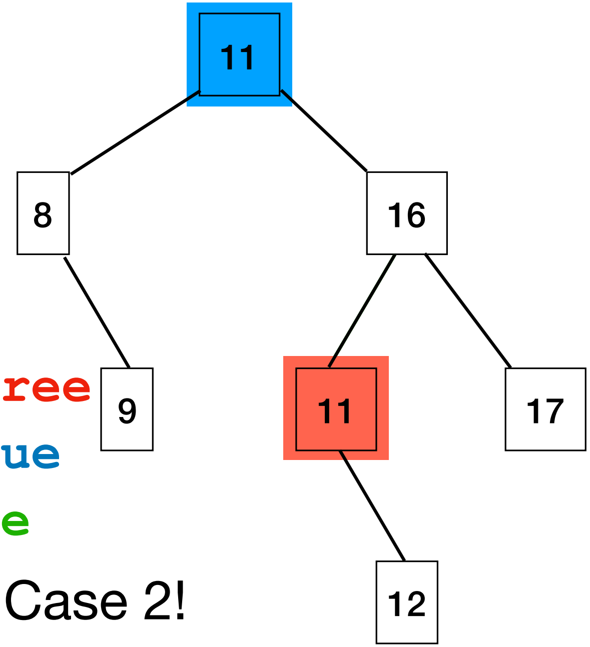
this **has to be** either Case 1 or Case 2!

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
```

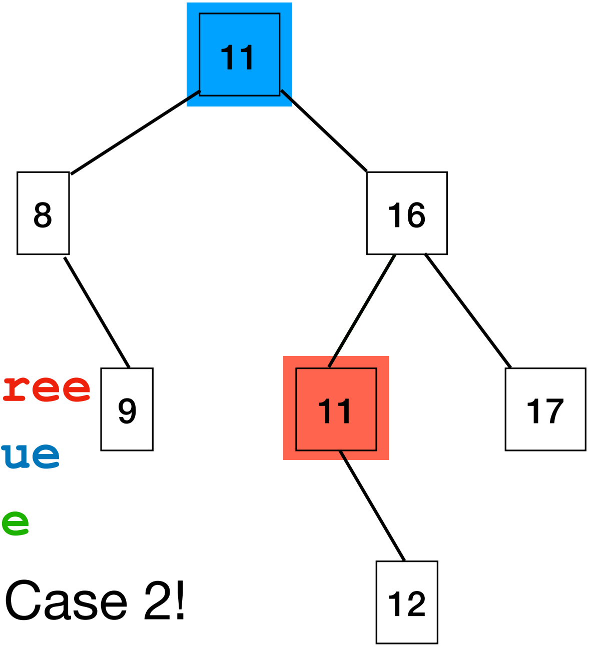
this **has to be** either Case 1 or Case 2!

Why? Rewind to before we removed it:

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!

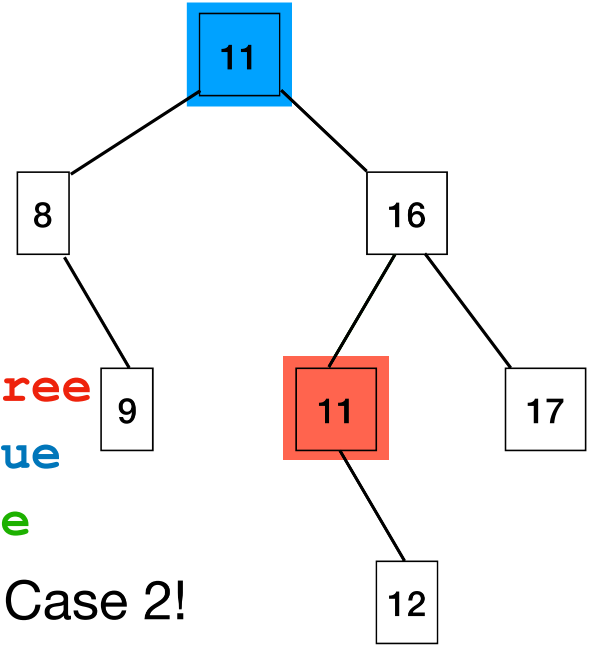
Why? Rewind to before we removed it:

- k is the smallest node in n's right subtree.

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!

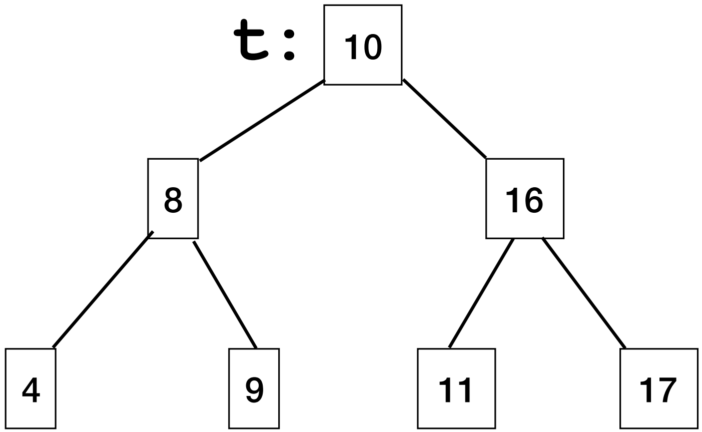
Why? Rewind to before we removed it:

- 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.
- 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.

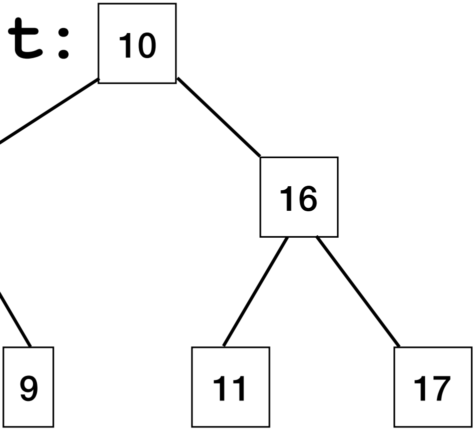
Practice



Do the following operations in sequence:

```
remove(9)
remove(4)
remove(10)
```

Practice



Do the following operations in sequence:

```
remove(9)
remove(4)
remove(10)
```

30 second kitten break

abstract data type

The Set ADT

```
/** A collection that contains no duplicates. */
```

Supports these operations:

```
boolean contains(Object ob);
boolean add(Object ob);
boolean remove(Object ob);
```

Set ADT

```
/** A collection that contains no duplicate
 * elements. */
interface Set {
  /** Return true if the set contains ob */
  boolean contains(Object ob);

  /** Add ob to the set; return true iff
   * the collection is changed. */
  boolean add(Object ob);

  /** Remove ob from the set; return true iff
   * the collection is changed. */
  boolean remove(Object ob);
  ...
}
```

The Set ADT

```
/** A collection that contains no duplicates. */
```

Supports these operations:

```
boolean contains(Object ob);
boolean add(Object ob);
boolean remove(Object ob);
```

Possible concrete implementations?

array < sorted
unsorted > linked list

BST

The Set ADT

```
/** A collection that contains no duplicates. */
```

Supports these operations:

```
  boolean contains(Object ob);
  boolean add(Object ob);
  boolean remove(Object ob);
```

Runtimes of possible concrete implementations?

	contains	add	remove
array (unsorted)			
array (sorted)			
linked list (unsorted)			
linked list (sorted)			
binary search tree			

Example: (unsorted) ArraySet<T>

```
class ArraySet<T> implements Set<T> {
 T[] a;
 int size;
 /** Return true iff the collection contains x */
 boolean contains(T x) {
  for (int i = 0; i < size; i++) {
   if a[i].equals(x)
    return true;
  }
  return false;
 }
 /** Add x to the collection; return true iff
  * the collection is changed. */
 boolean add(T x) {
  if (!contains(x)) {
   a[size] = x; // let's hope a is big enough...
   size++;
   return true;
  }
  return false;
 }
```