CSCI 241

Scott Wehrwein

Binary Search Trees:
Motivation (Set ADT)
Definition, Search, and Insertion

Goals

Know the purpose and operations of the Set Abstract Data Type.

Know the motivation for and the definition of a binary search tree.

Be able to execute on paper, and be prepared to implement the `search` and `add` operations on a BST.

The 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 changed. */
  boolean add(Object ob);

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

Set ADT: Possible Implementations

	contains	add	remove
LinkedList	O(n)	O(n)	O(n)
Array (sorted)	O(log n)	O(n)	O(n)
Array (unsorted)	O(n)	O(n)	O(n)
Tree?	O(n)	??	??

Searching a Binary Tree

A binary tree is

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

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?)

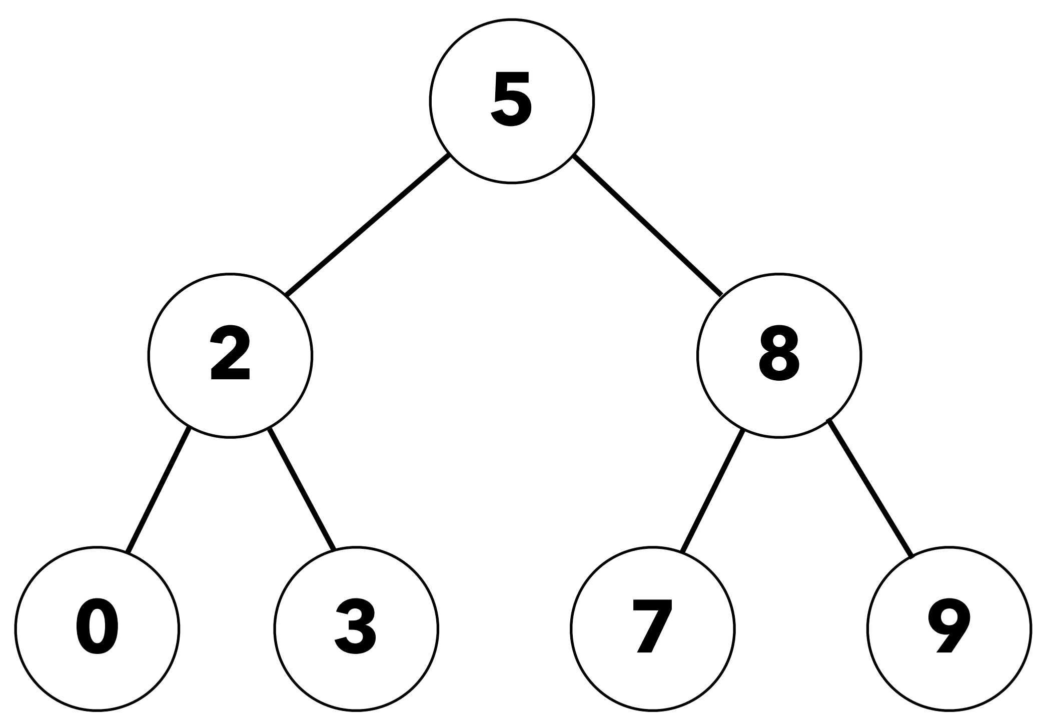
```
return findVal(t.left)
    || findVal(t.right)
```

(recursive call - is v in right?)

An opportunity

- `contains` is O(n) because we have to search every node.

- Can we somehow avoid that?



Binary Tree

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

aside: sometimes it's helpful to keep a pointer to your parent

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;
}
```

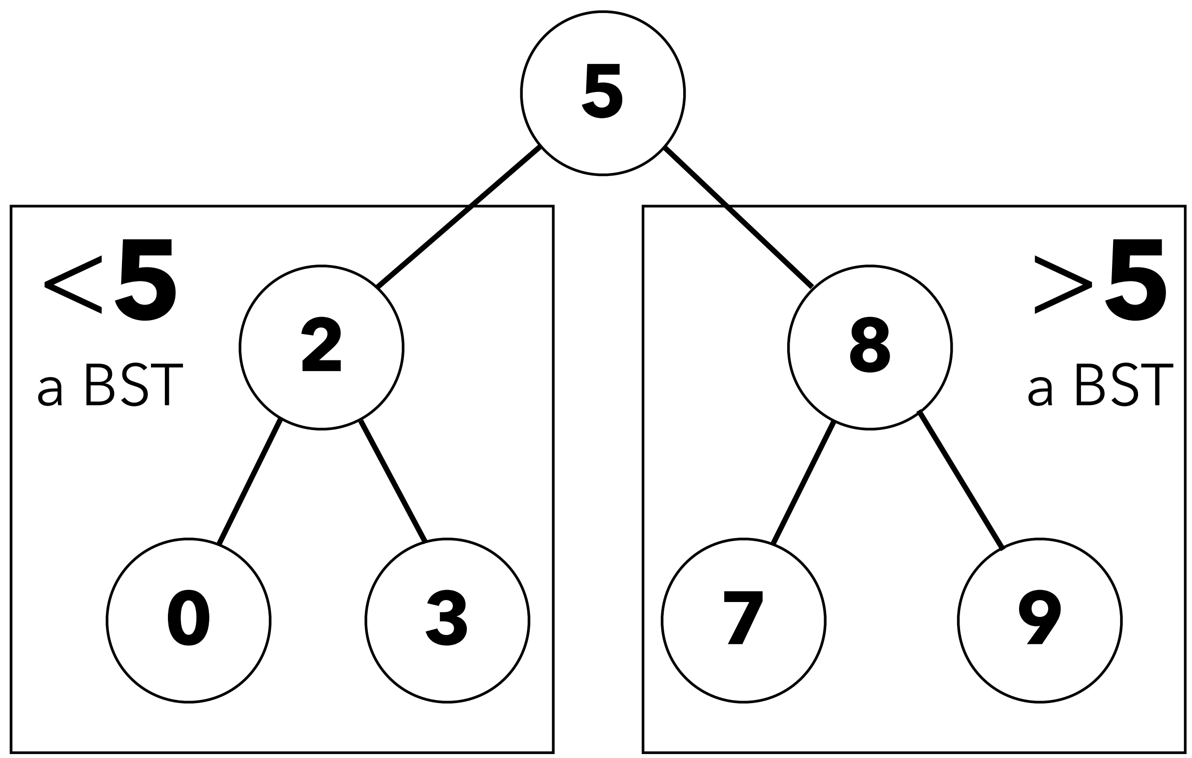
Binary Search Tree

```
/** 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!

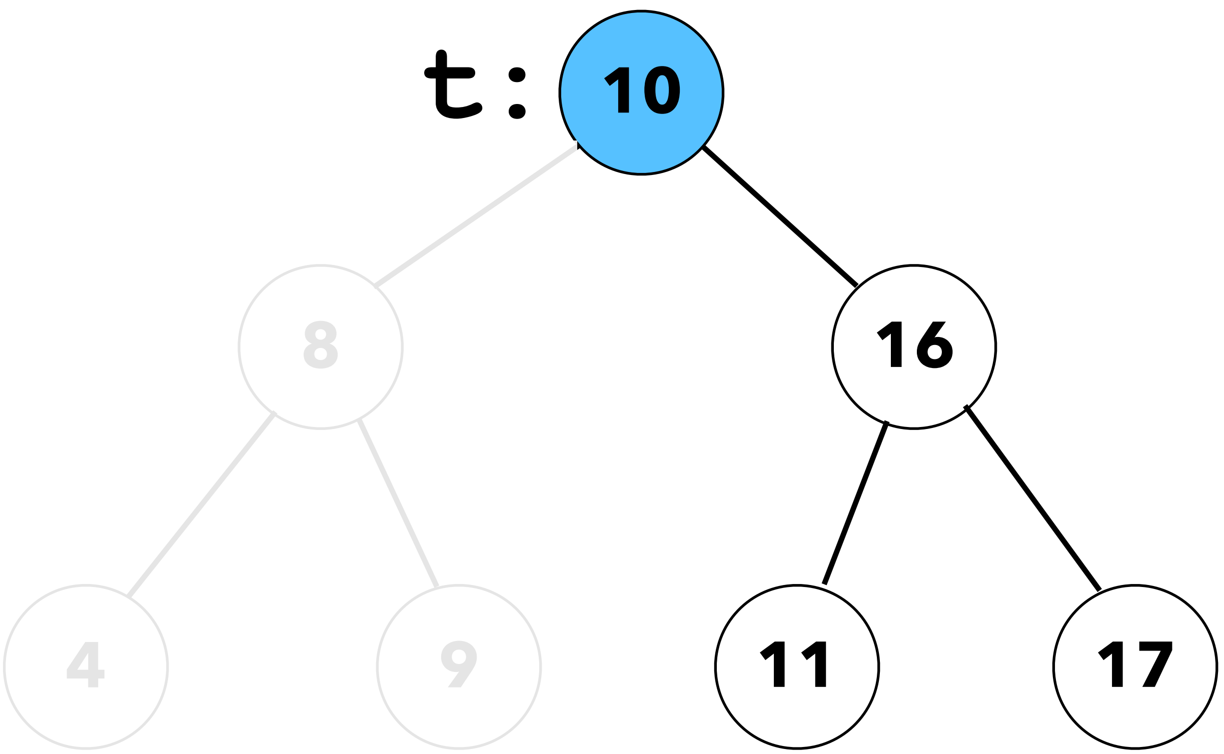
(but not coincidence)

Binary Search Tree: Example



Searching a BST

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

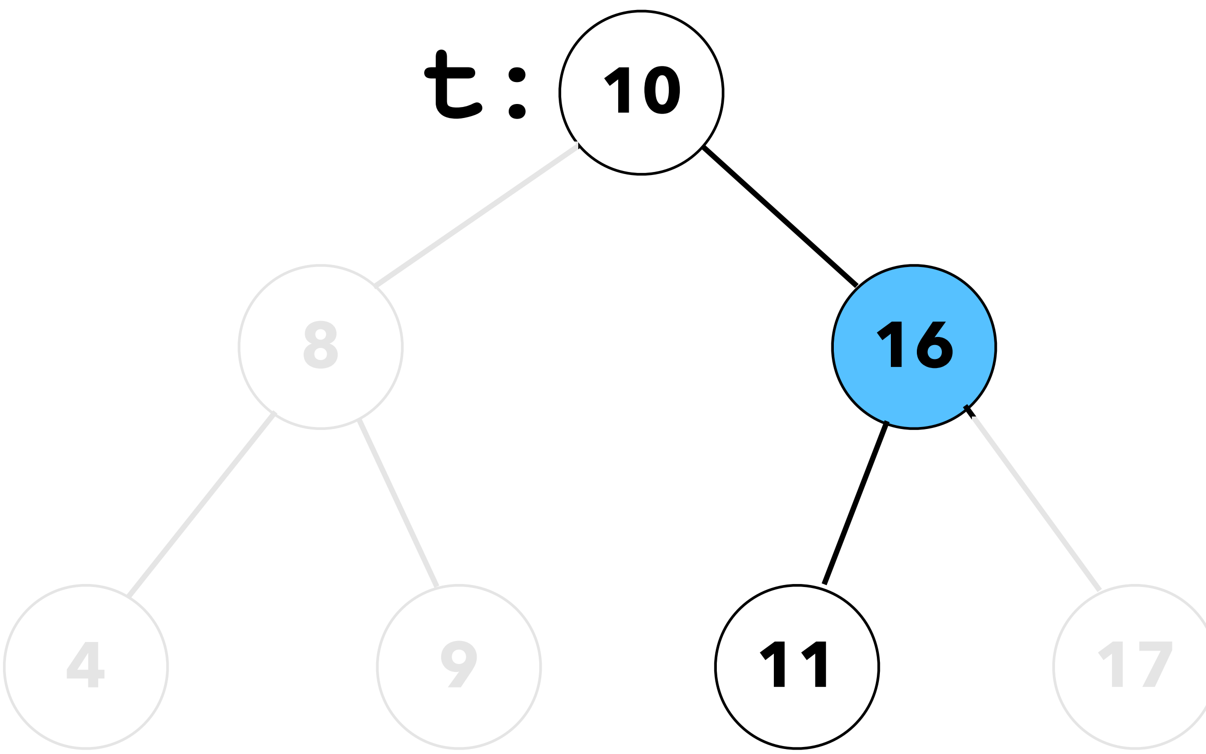


11 > 10

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

Searching a BST

`search(t, 11)`



$11 > 10$

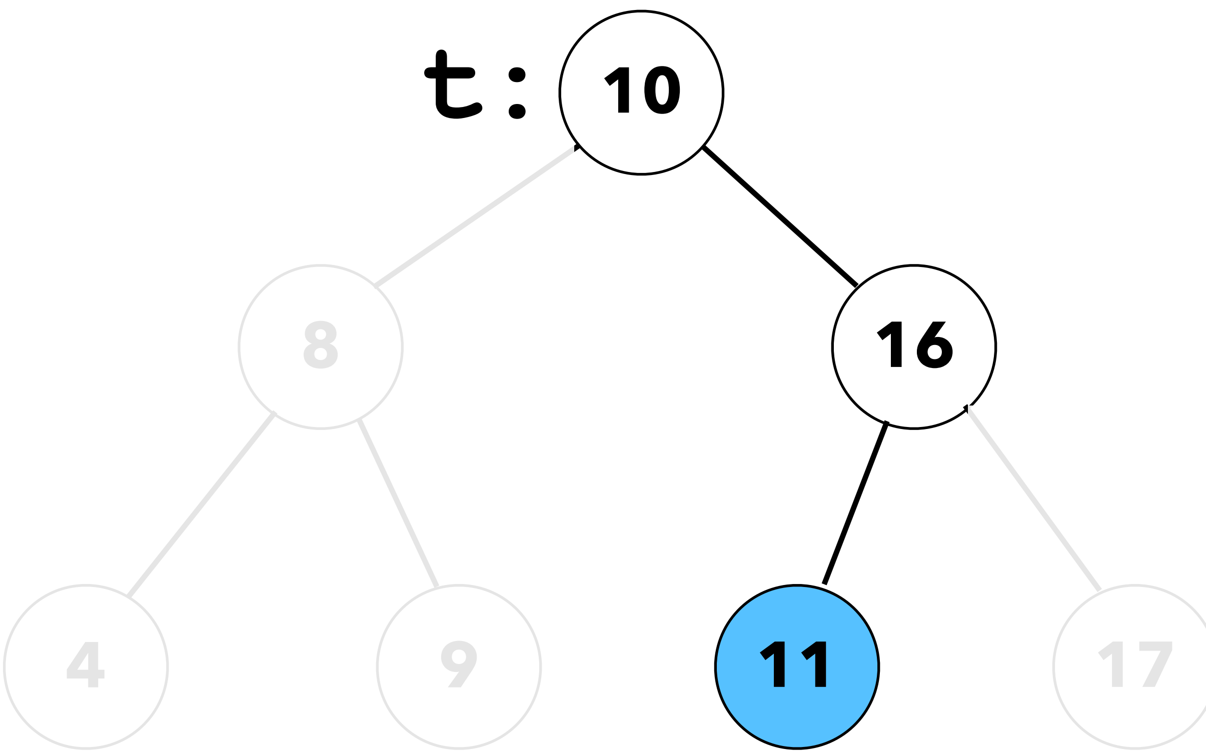
`search(right, 11)`

$11 < 16$

`search(left, 11)`

Searching a BST

search(t, 11)



11 > 10

search(right, 11)

11 < 16

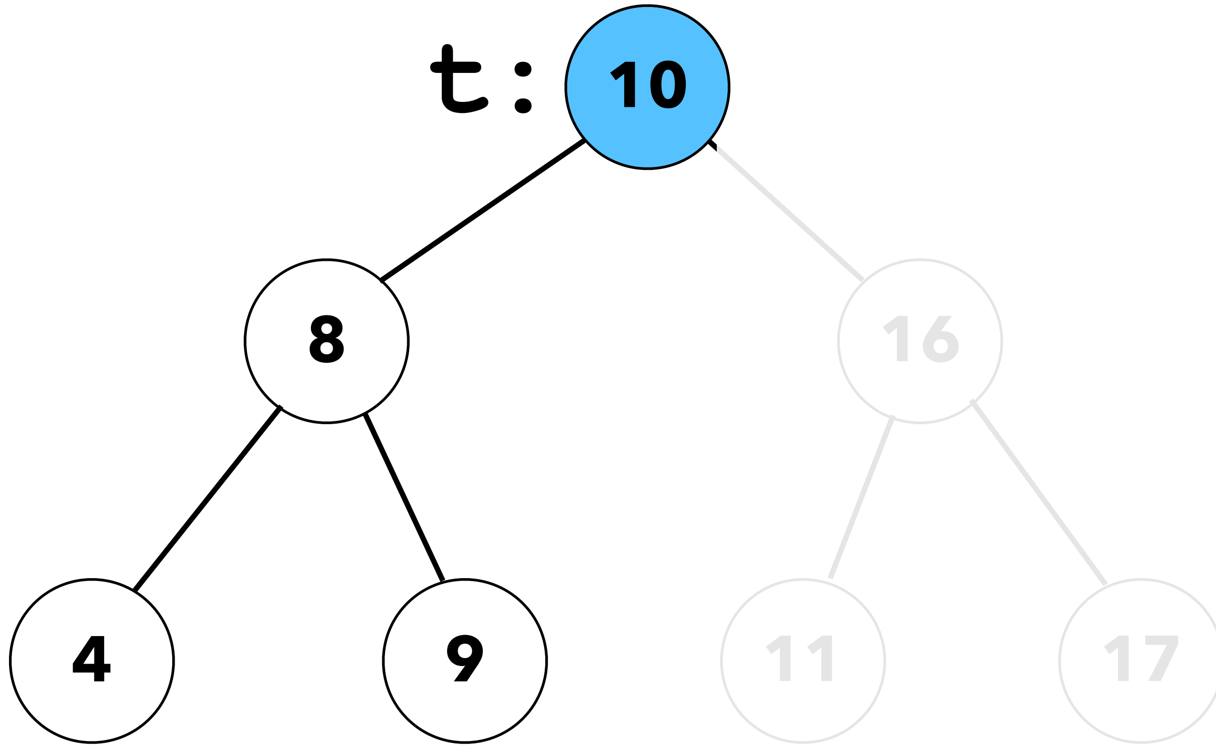
search(left, 11)

11 == 11

found it! return.

Searching a BST - the nonexistent case

`search(t, 5)`

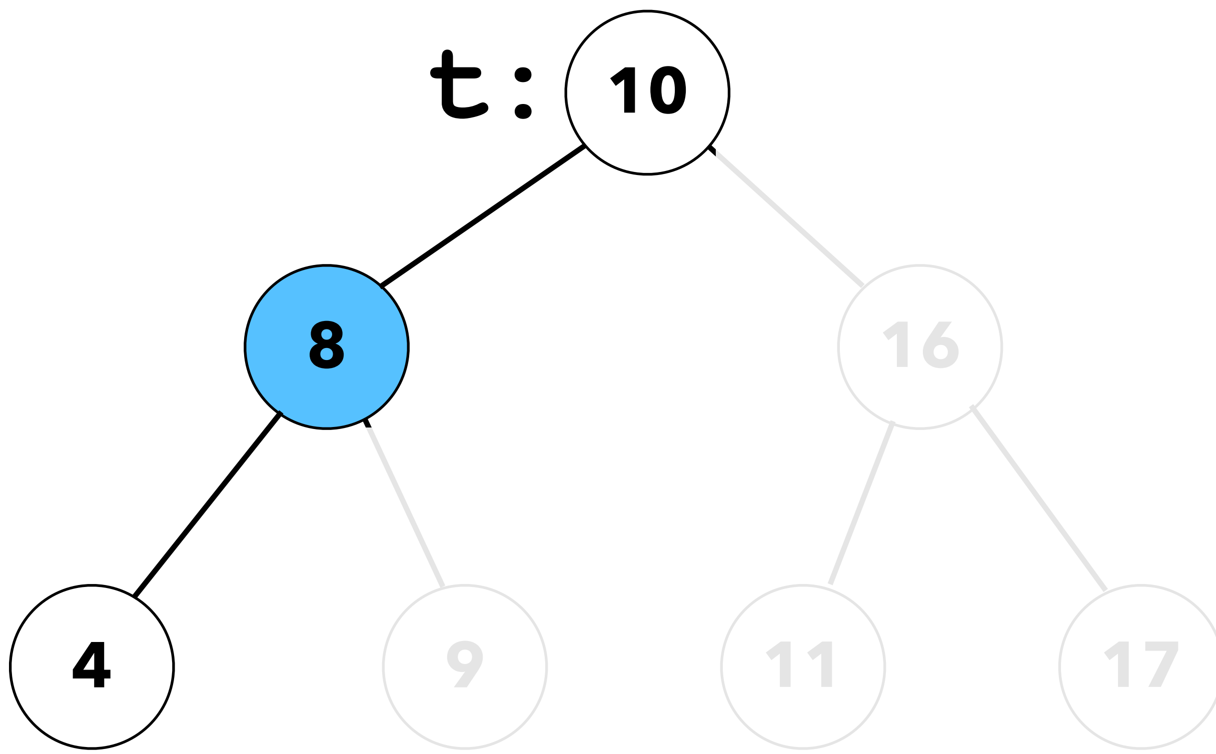


5 < 10

`search(left, 5)`

Searching a BST - the nonexistent case

`search(t, 5)`



5 < 10

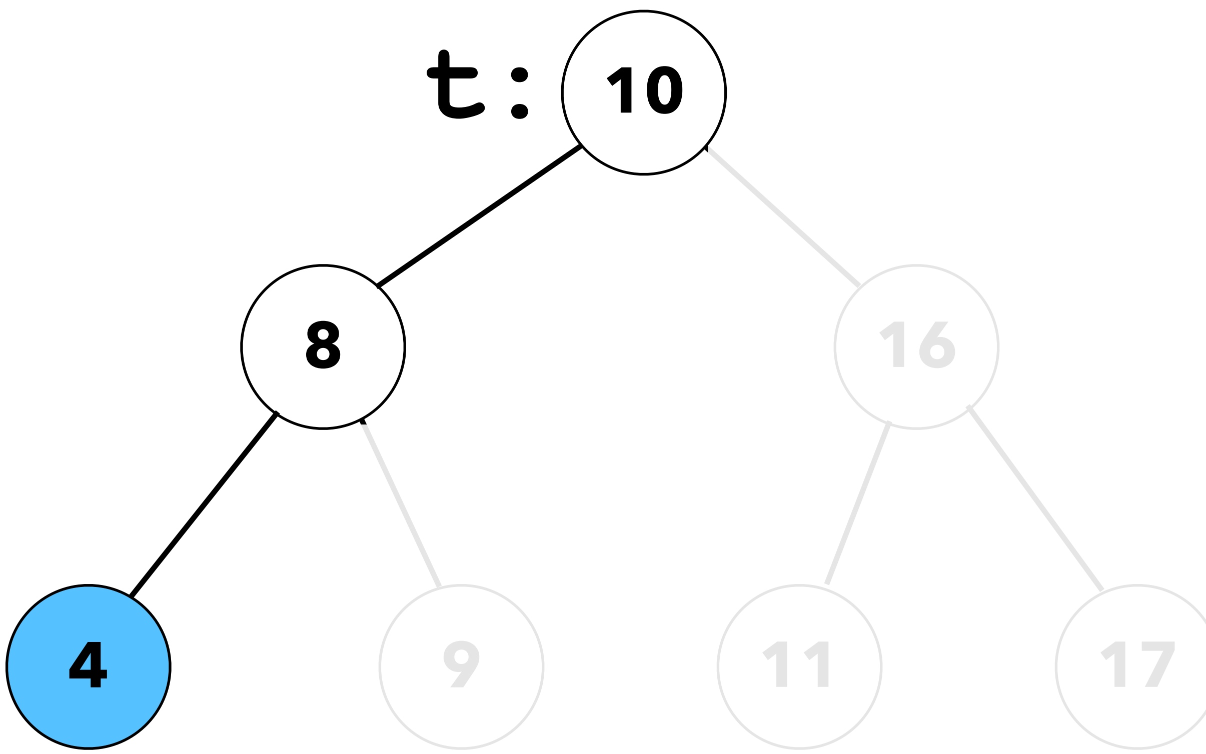
`search(left, 5)`

5 < 8

`search(left, 5)`

Searching a BST - the nonexistent case

`search(t, 5)`



$5 < 10$

`search(left, 5)`

$5 < 8$

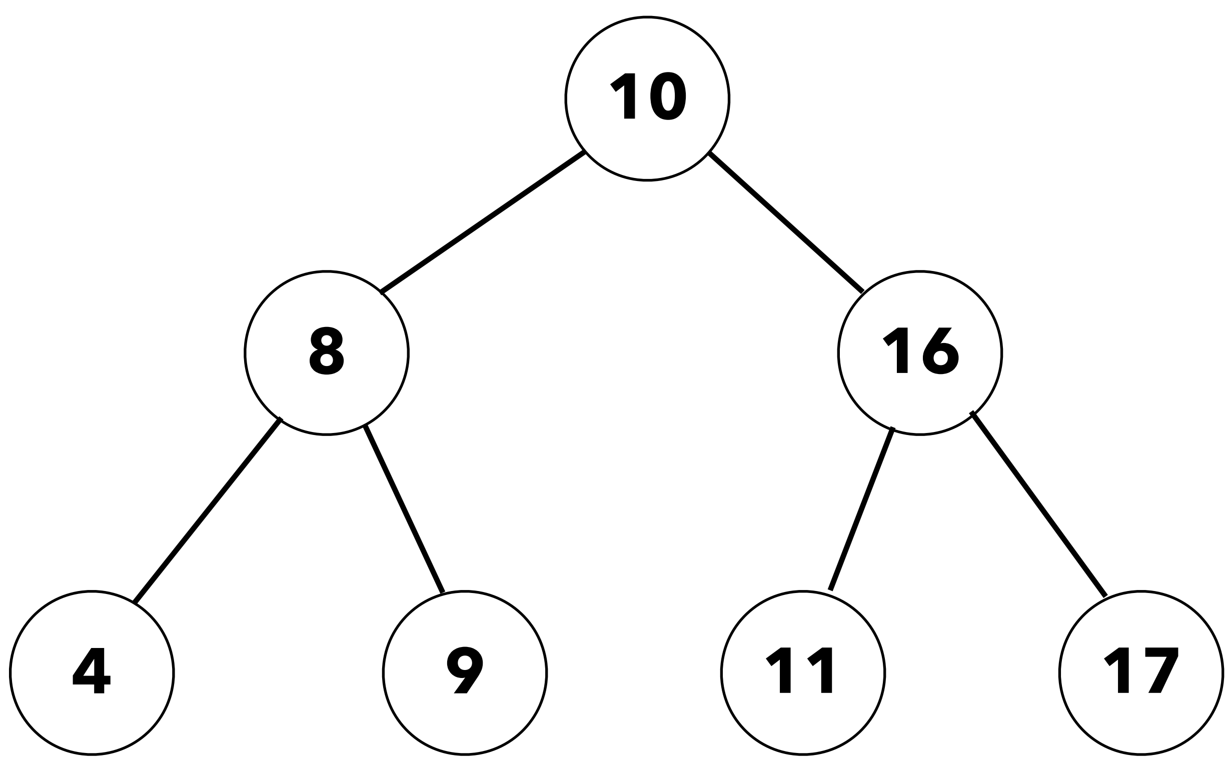
`search(left, 5)`

$5 > 4$

`search(right, 5)`

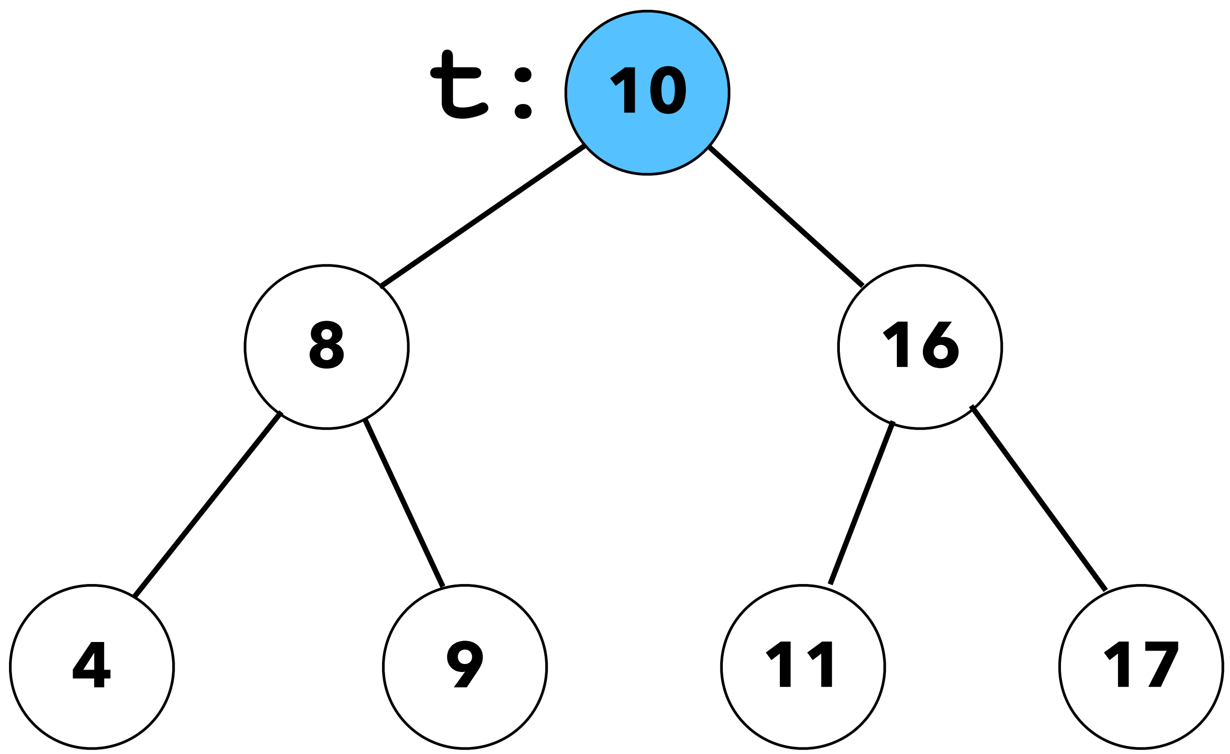
`null - not found!`

Inserting into a BST



Inserting into a BST

`insert(t, 11)`

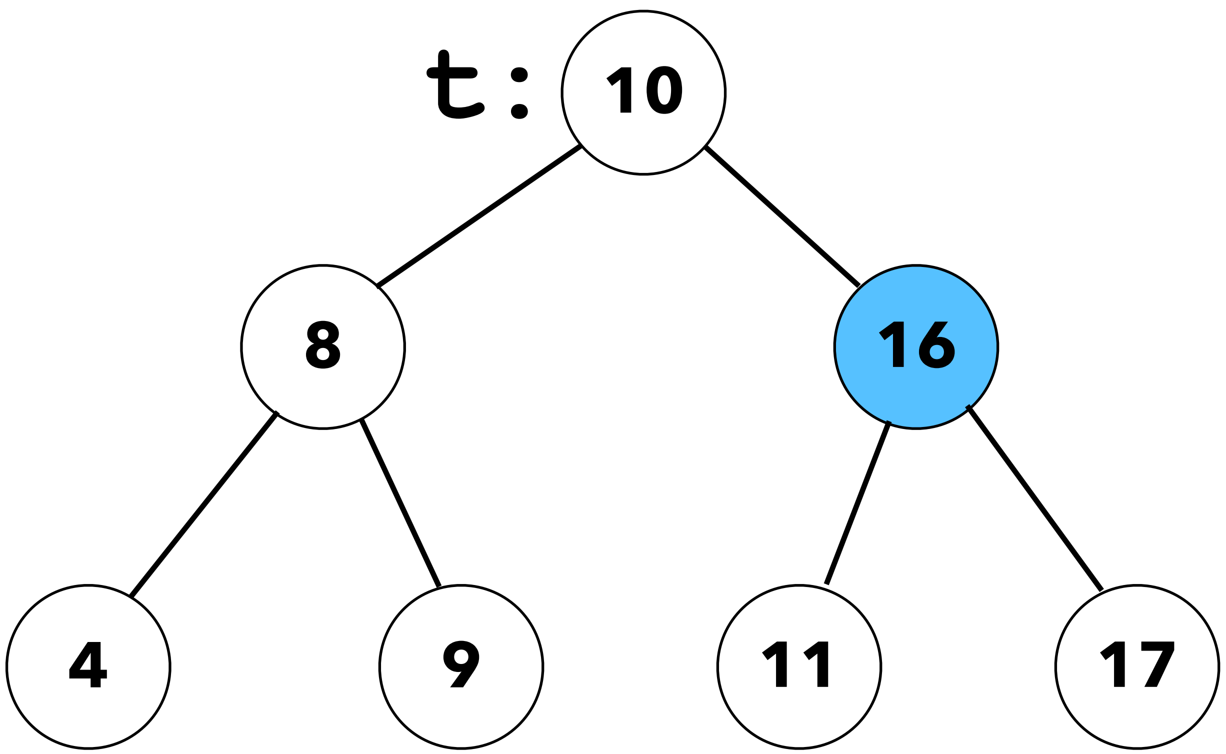


11 > 10

`insert(right, 11)`

Inserting into a BST

`insert(t, 11)`



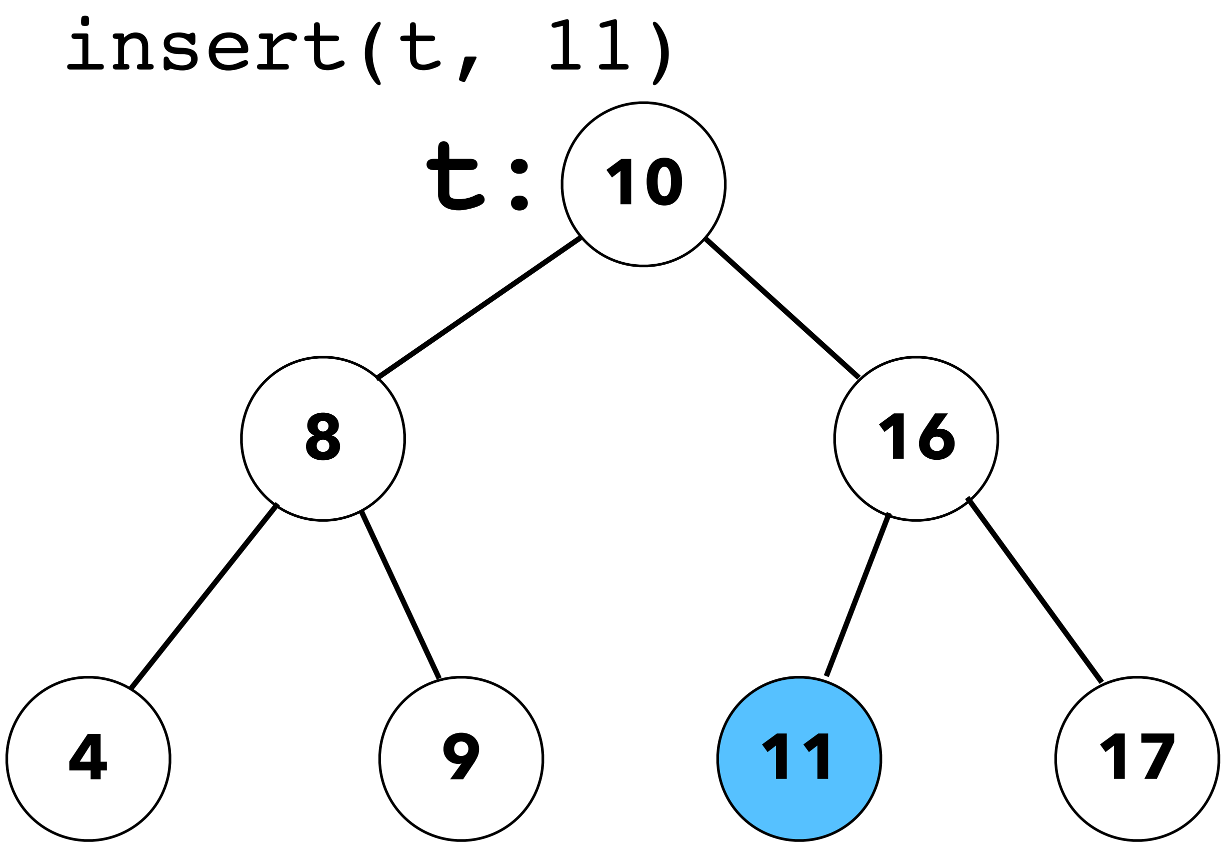
11 > 10

`insert(right, 11)`

11 < 16

`insert(left, 11)`

Inserting into a BST



11 > 10

insert(right, 11)

11 < 16

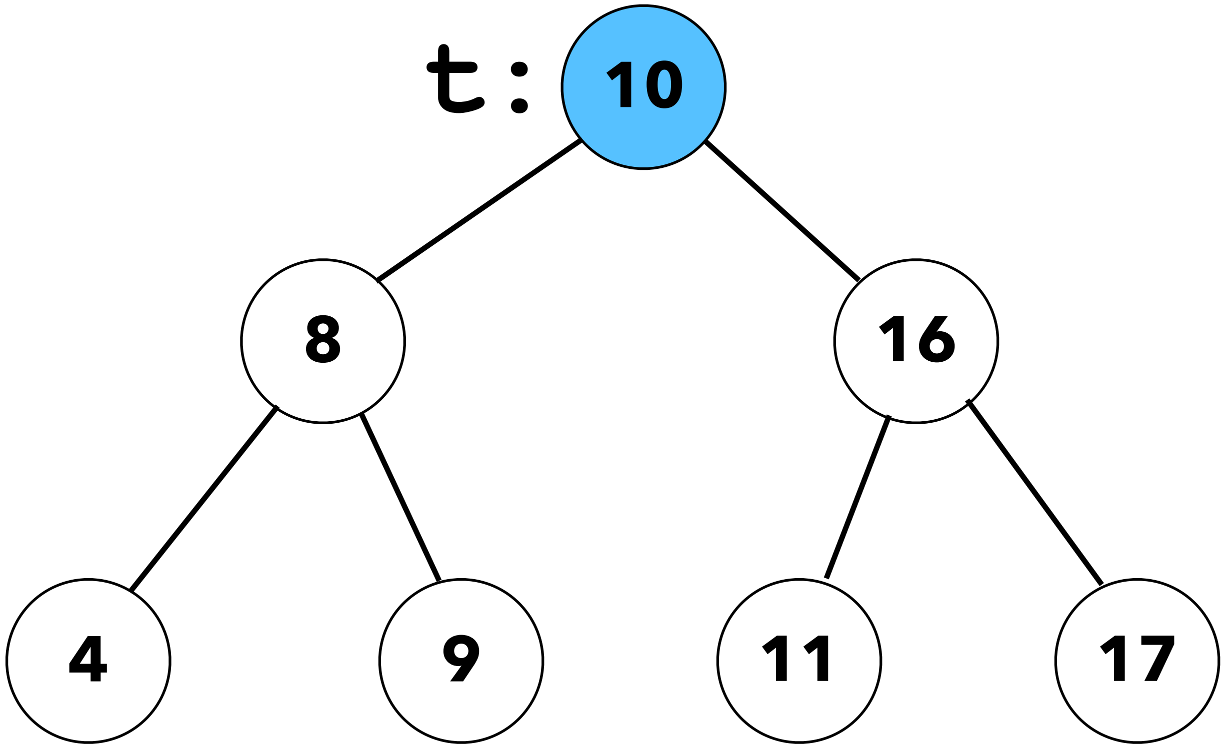
insert(left, 11)

11 == 11

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

Inserting into a BST - the nonexistent case

`insert(t, 5)`

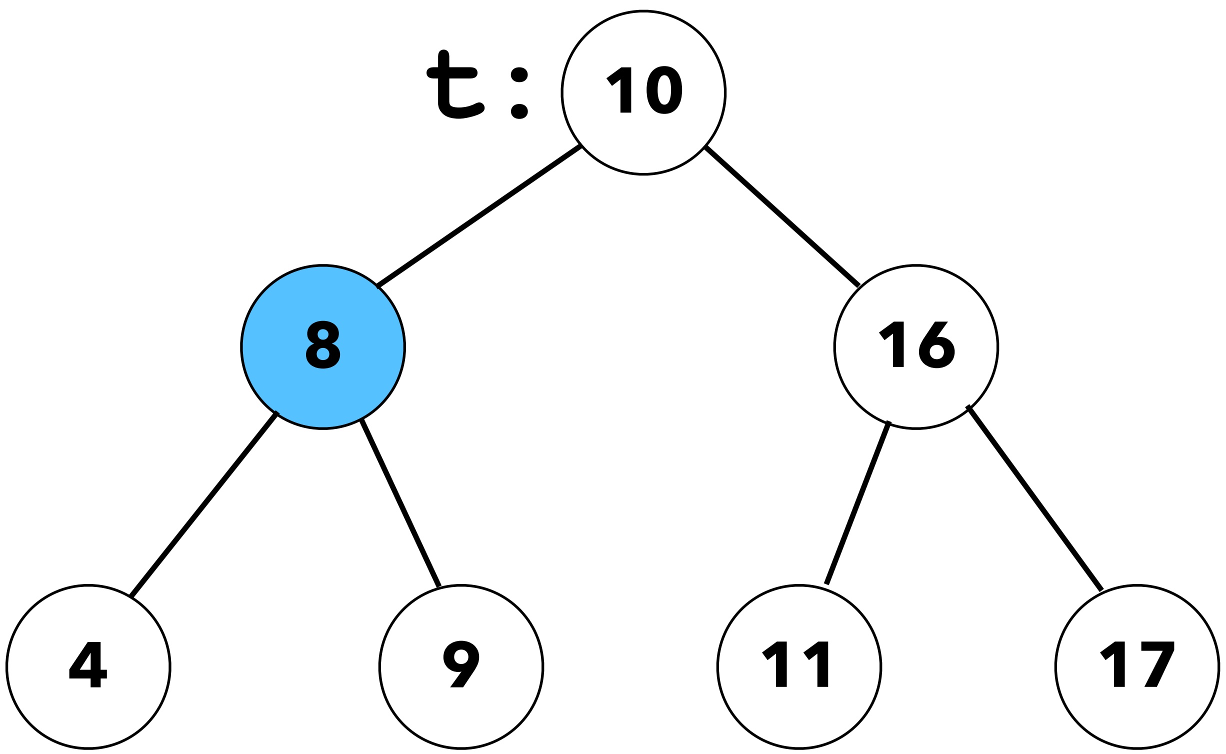


5 < 10

`insert(left, 5)`

Inserting into a BST - the nonexistent case

`insert(t, 5)`



5 < 10

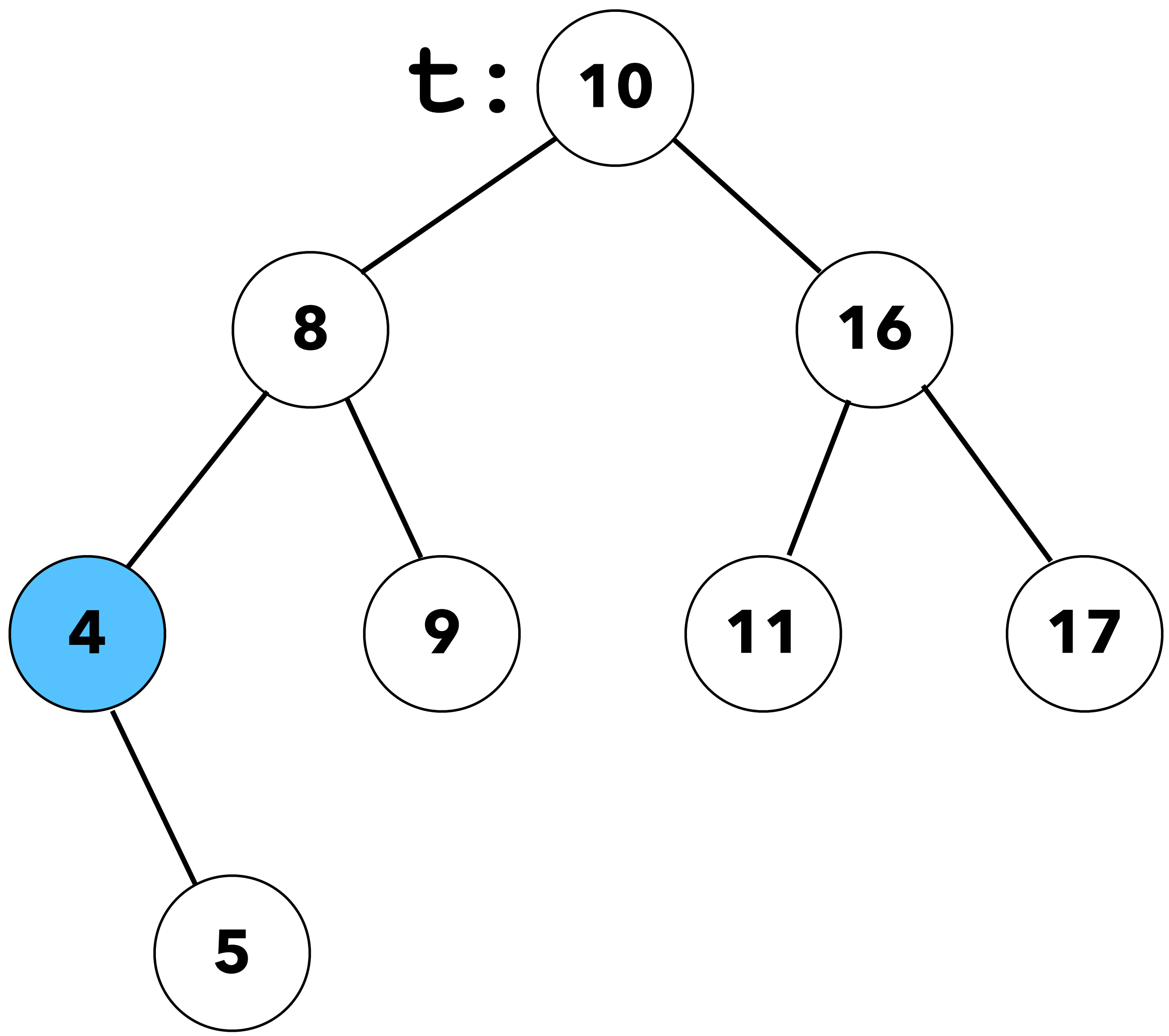
`insert(left, 5)`

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!`