CSCI 241

A Very Brief Intro to Generics

AVL Trees I: Rotations, Balance Factor

Announcements

- Schedule adjustments:
  - A2 is due Friday 2/15 instead of 2/11
  - **Midterm exam is 2/22 instead of 2/15**
- A1 grading is underway
  - A lot of people submitted A1 late
  - Not a lot of people visited me in office hours
  - You may resubmit **once** for half of **unit test** credit back

Goals

- Know why Java has **generics**, and how to use and implement them.
- Be prepared to implement **rotations** in BSTs
- (probably next time) Be prepared to implement AVL **rebalancing**.

A Very Brief Intro to Generics

(because your lab depends on it)

Before Generics

```
/** A collection that contains no duplicate elements. *
interface Set {
  /** Return true iff the collection contains ob */
  boolean contains(Object ob);
  /** Add ob to the collection; return true iff
   * the collection is changed. */
  boolean add(Object ob);
  /** Remove ob from the collection; return true iff
   * the collection is changed. */
  boolean remove(Object ob);
  ...
}
```

Can contain anything that extends Object (any class at all)

- But **not primitive types**: int, double, float, boolean, …

The Problem

```
Set c = ...
c.add("Hello")
c.add("World");
...
for (Object ob : c) {
  String s = (String) ob;
  // do things with s
}
```

Notice: Arrays don't have this problem!

```
String[] a = ...
a[0]= ("Hello")
a[1]= ("World");
...
for (String s : a) {
  System.out.println(s);
}
```

The Solution: Generics

```
Object[] oa= ...             // array of Objects
String[] sa=  ...            // array of Strings
ArrayList<Object> oA= ... // ArrayList of Objects
ArrayList<String> oA= ... // ArrayList of Strings
```

Now the Set interface written like this:

```
interface Set<T> {
  /** Return true iff the collection contains x */
  boolean contains(T x);

  /** Add x to the collection; return true iff
   * the collection is changed. */
  boolean add(T x);

  /** Remove x from the collection; return true iff
   * the collection is changed. */
  boolean remove(T x);
  ...
}
```

The Solution: Generics

The Set interface is now written like this:

```
interface Set<T> {
  /** Return true iff the collection contains x */
  boolean contains(T x);

  /** Add x to the collection; return true iff
   * the collection is changed. */
  boolean add(T x);

  /** Remove x from the collection; return true iff
   * the collection is changed. */
  boolean remove(T x);
  ...
}
```

Key idea: I don’t need to know what T is to implement these!

The Solution: Generics

Key idea: I don't need to know what T is to implement these!

```
Set<String> c= ...
c.add("Hello")  /* Okay */
c.add(1979);    /* Illegal: compile error! */
```

Generally speaking,

`Collection<String>`

behaves like the parameterized type

`Collection<T>`

where all occurrences of `T` have been replaced by `String`.

The Solution: Generics

The bummer: T must extend Object - no primitive types.
Can't do:

```
Collection<int> c = ...
```

Have to use:

```
Collection<Integer>
```

Java often seamlessly converts `int` to `Integer` and back.

```
Integer x = 5; // works
int x = new Integer(5); // works
```

"Autoboxing/unboxing"

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;
  }
```

Questions to Ponder

- What's the runtime of each **ArraySet** operation?
- Sketch out the operations for a **LinkedListSet** and analyze their runtime.
- Sketch out the operations for a **BSTSet** and analyze their runtime.

Back to BSTs

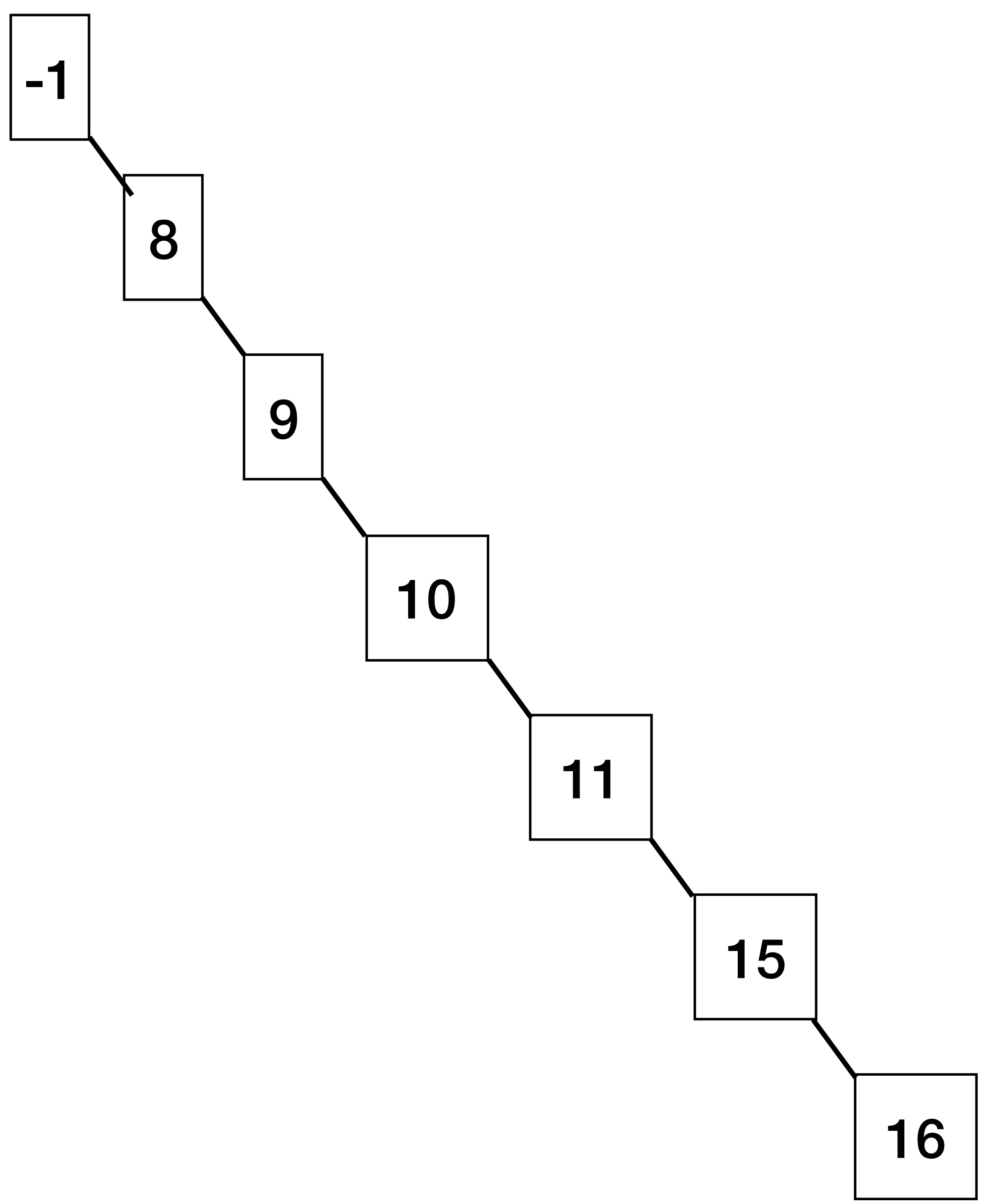
Long ago, we built some trees:

```
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)
```

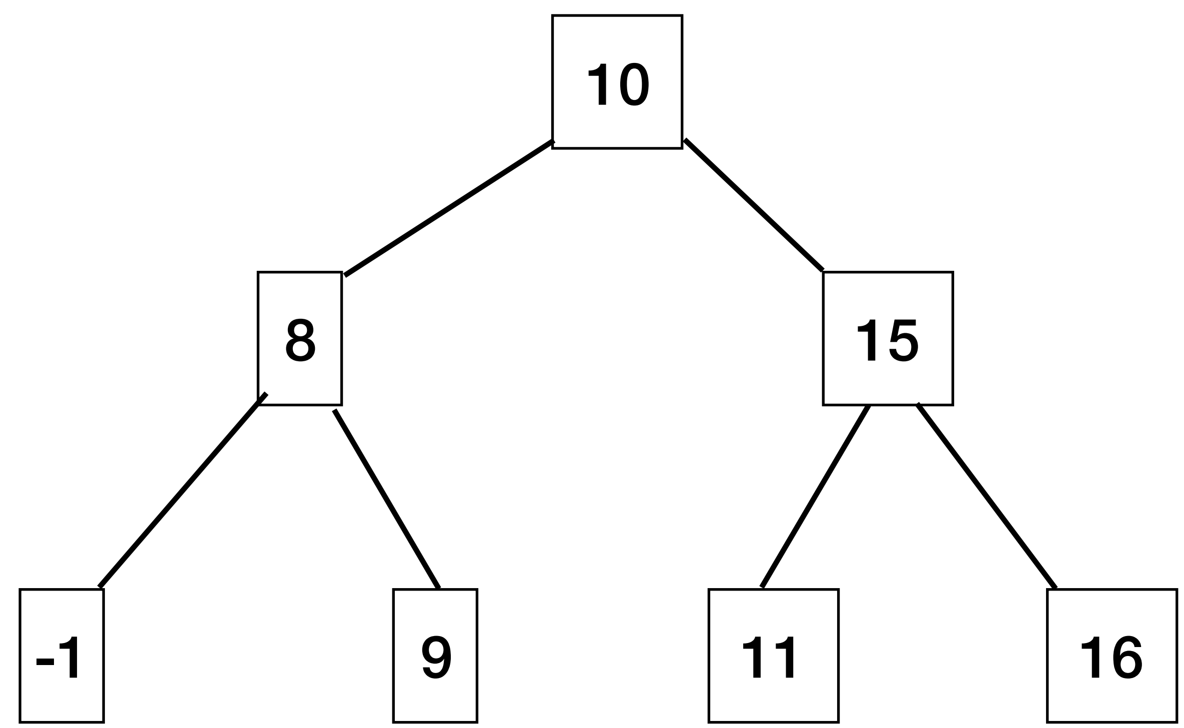
```
t = new BST();
t.insert(-1)
t.insert(8)
t.insert(9)
t.insert(10)
t.insert(11)
t.insert(15)
t.insert(16)
t.insert(16)
```

Same values, different trees

Bad tree =(

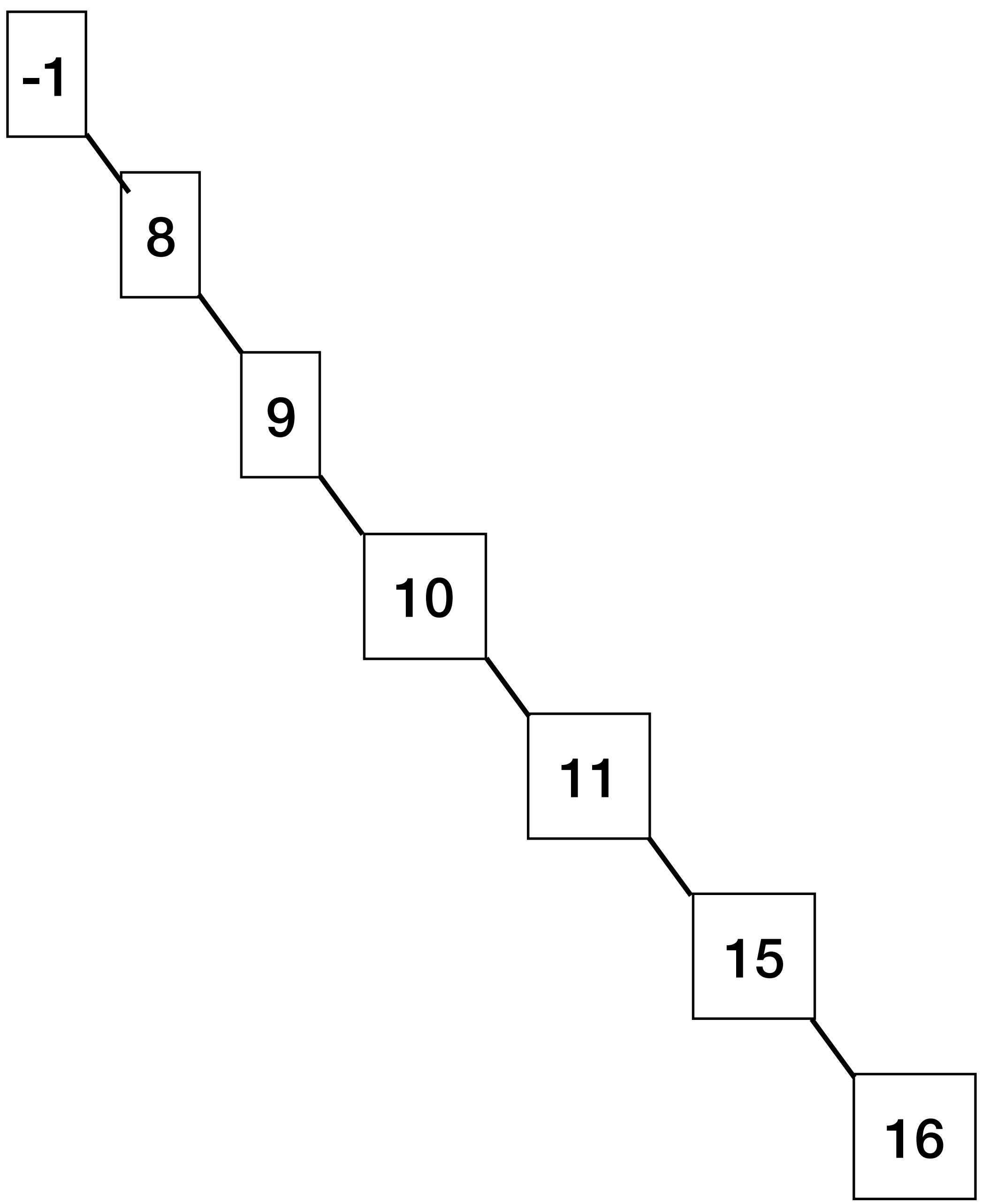


Good tree =)

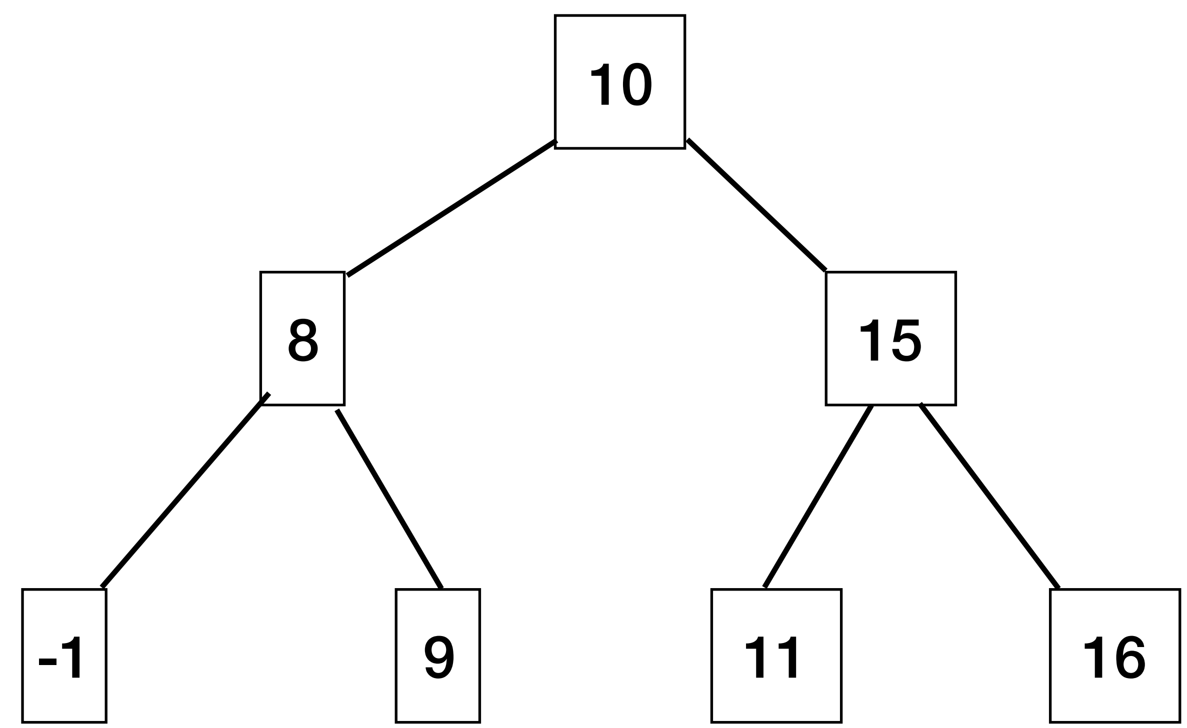


Can we make a tree less bad?

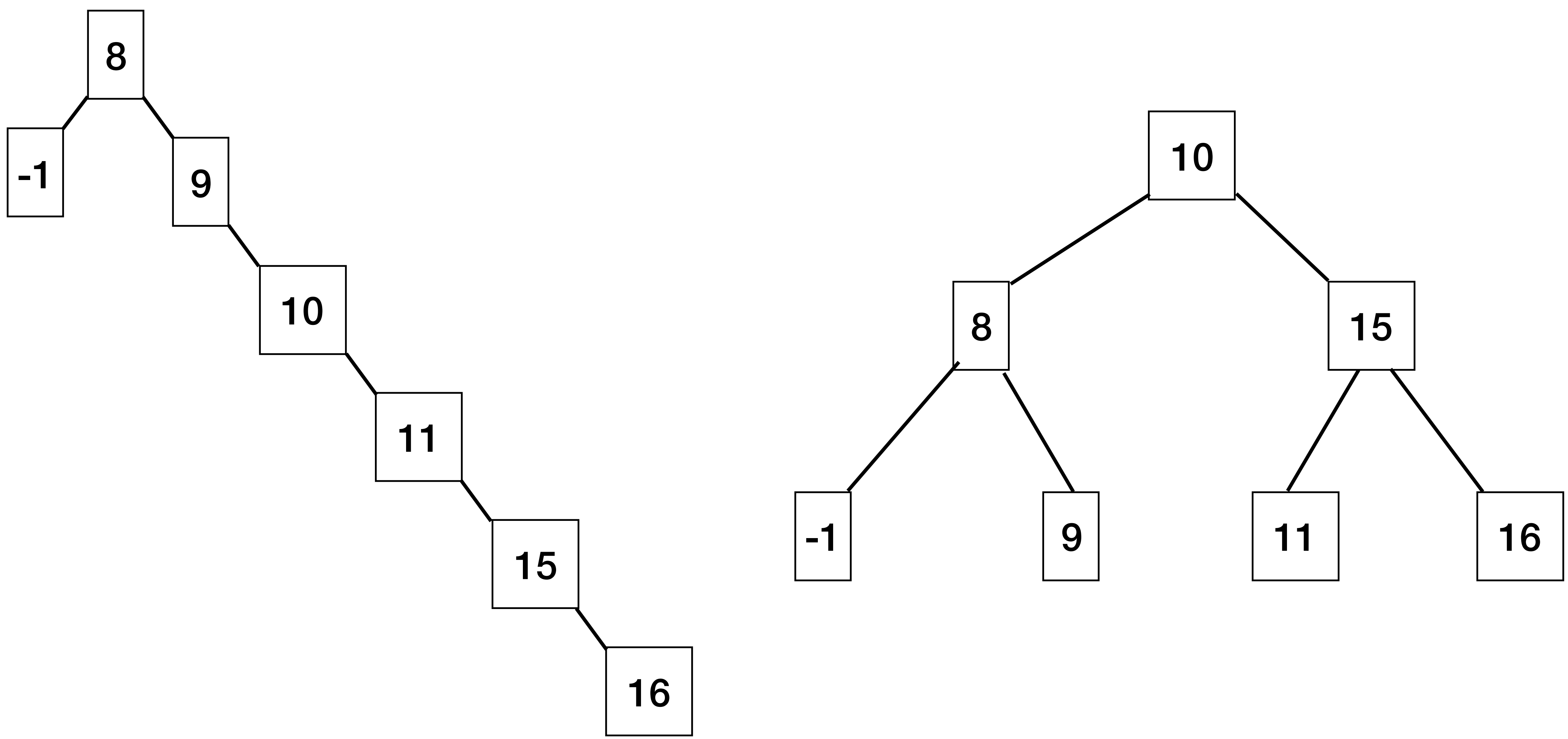
Bad tree =(



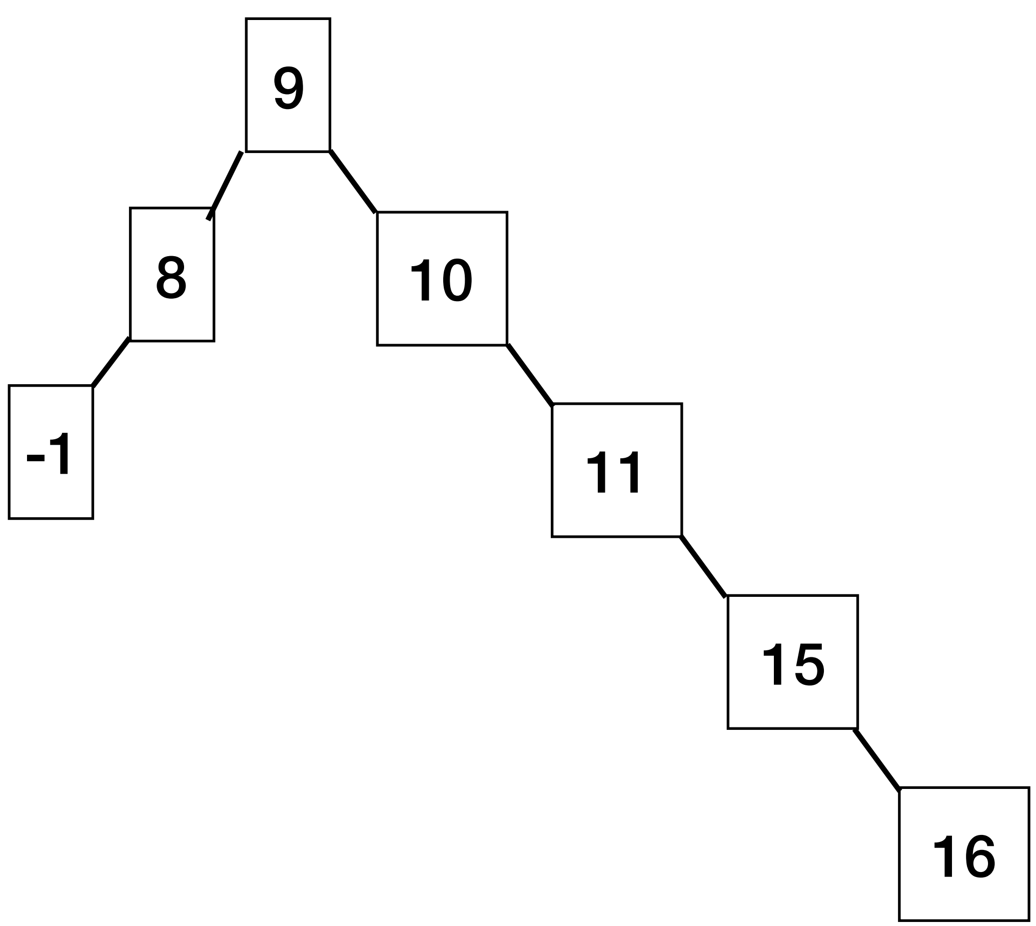
Good tree =)

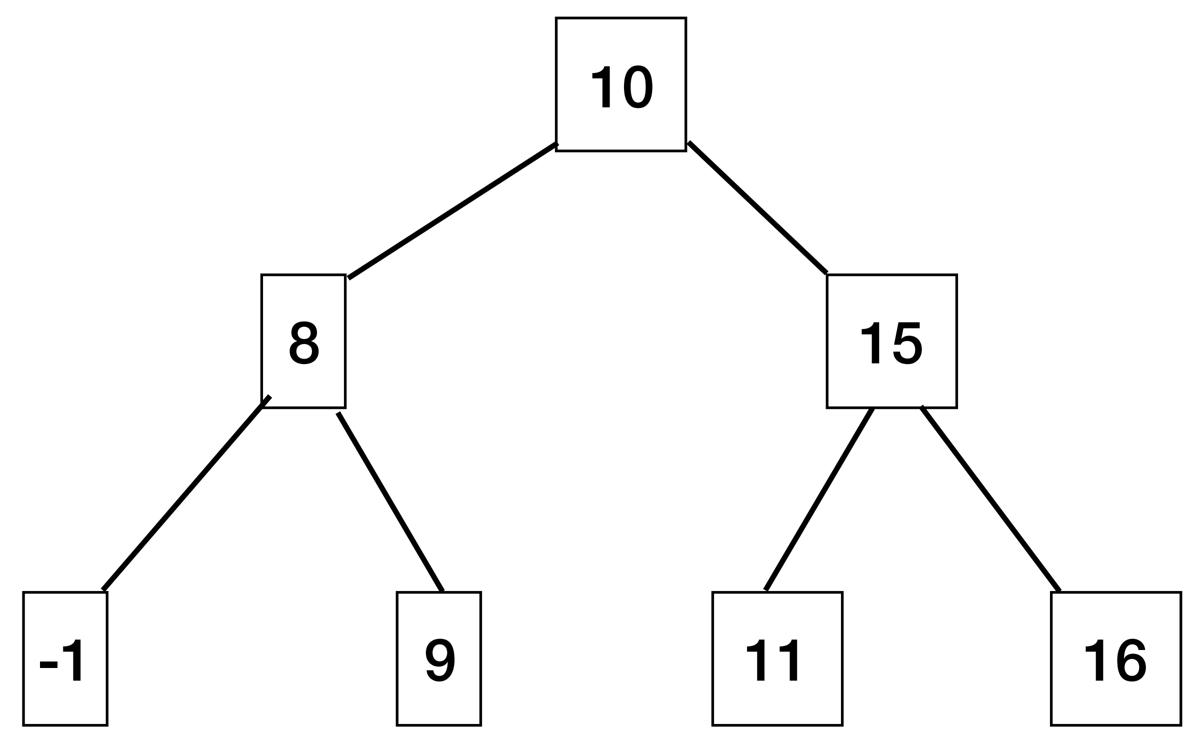


Can we make a tree less bad?

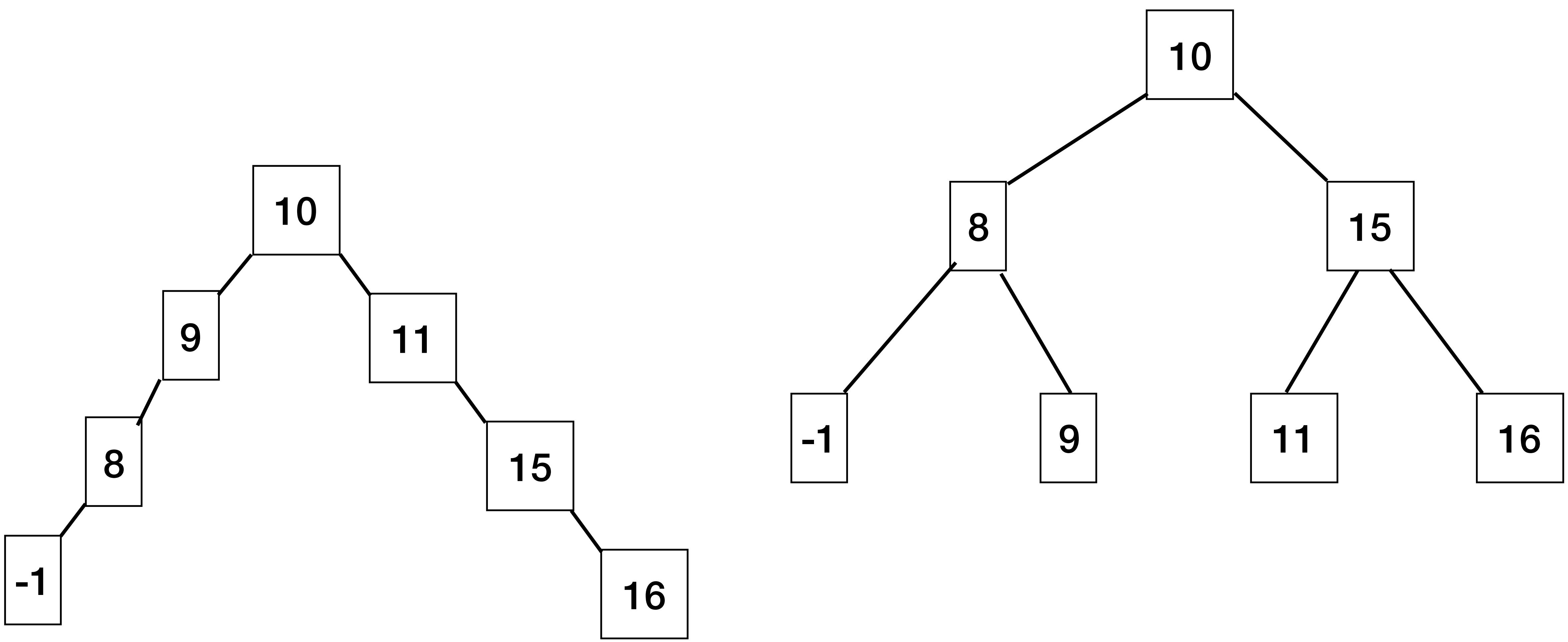


Can we make a tree less bad?

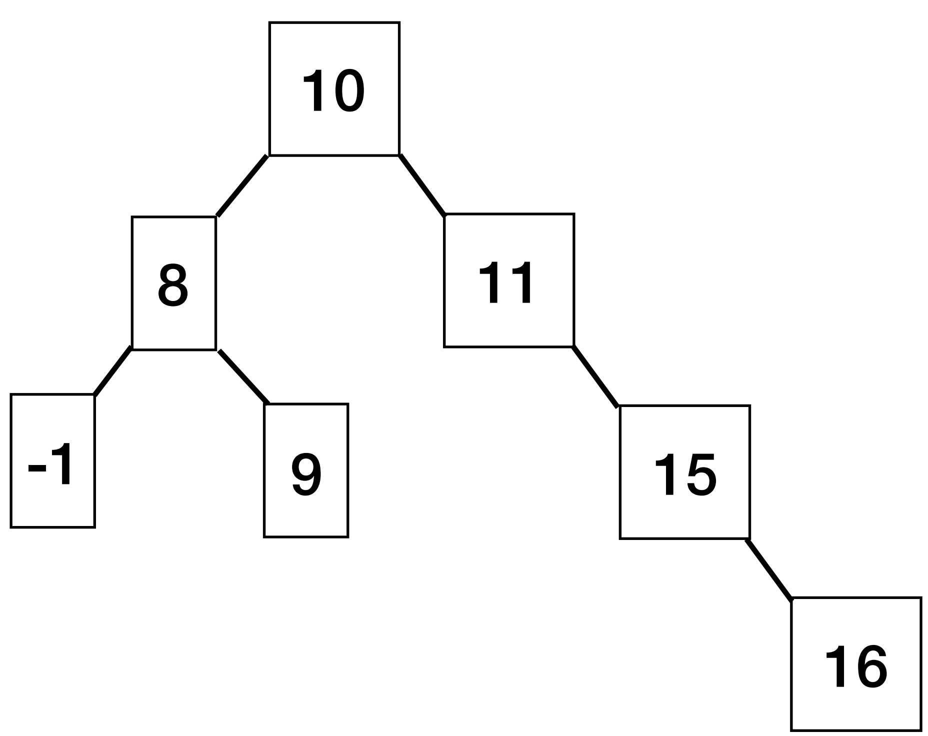


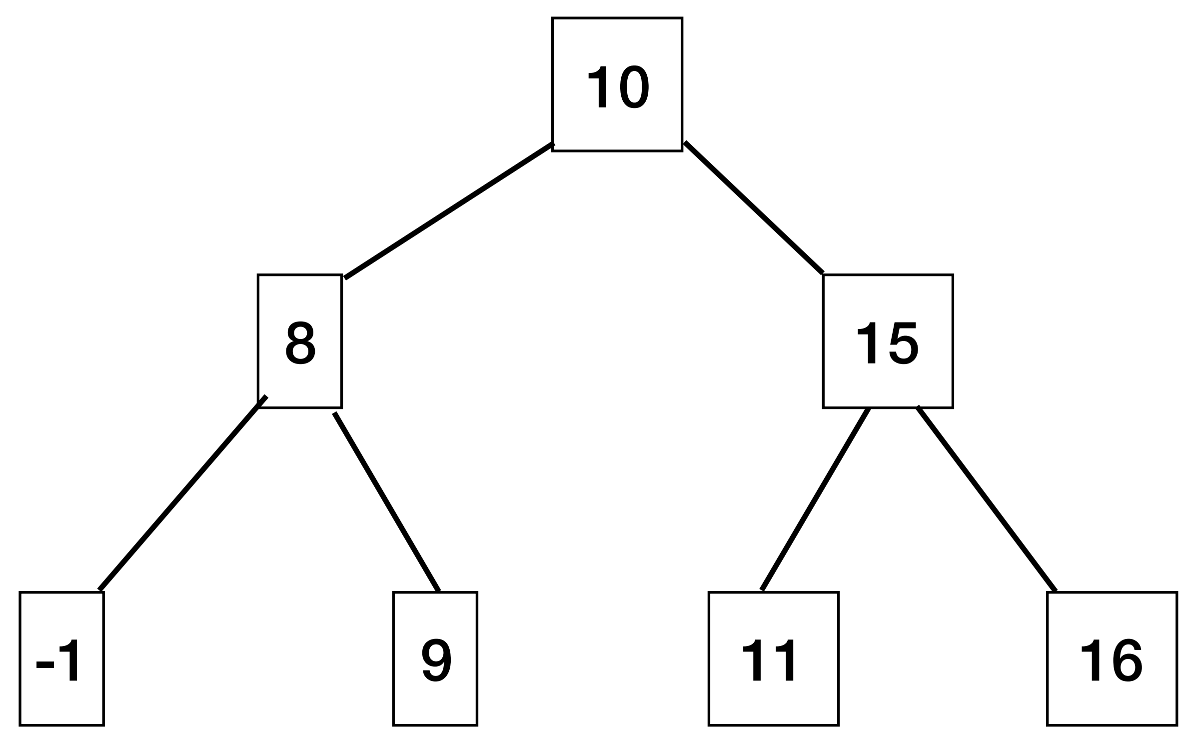


Can we make a tree less bad?

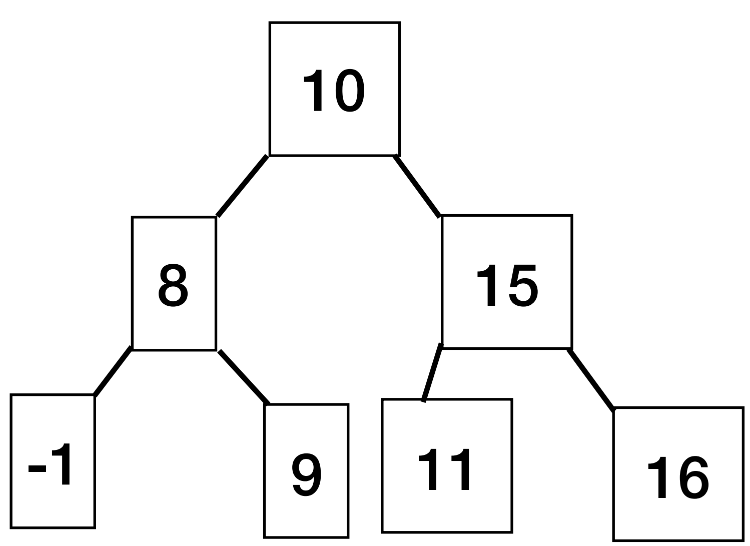


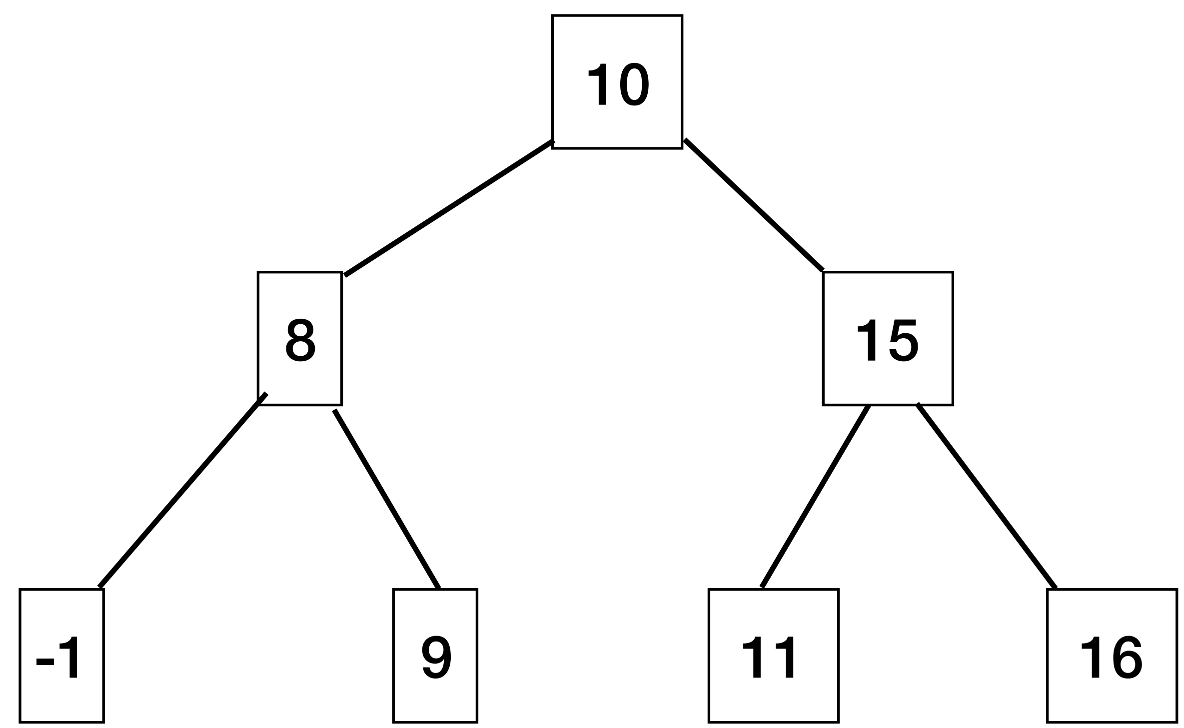
Can we make a tree less bad?





Can we make a tree less bad?



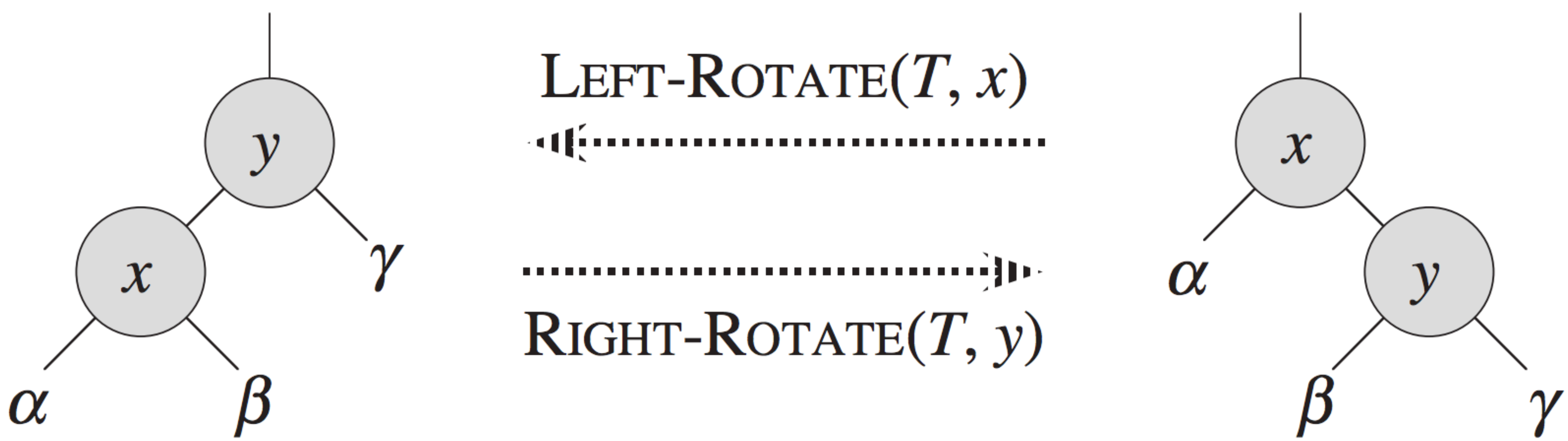


Tree Rotations

modify the structure without violating the BST property.

Steps in left rotation (move y up to its parent's position):

1. Transfer β: x's right subtree becomes y's old left subtree (β)
2. Transfer the parent: y's parent becomes x's old parent
3. Transfer x itself: x becomes y's left subtree

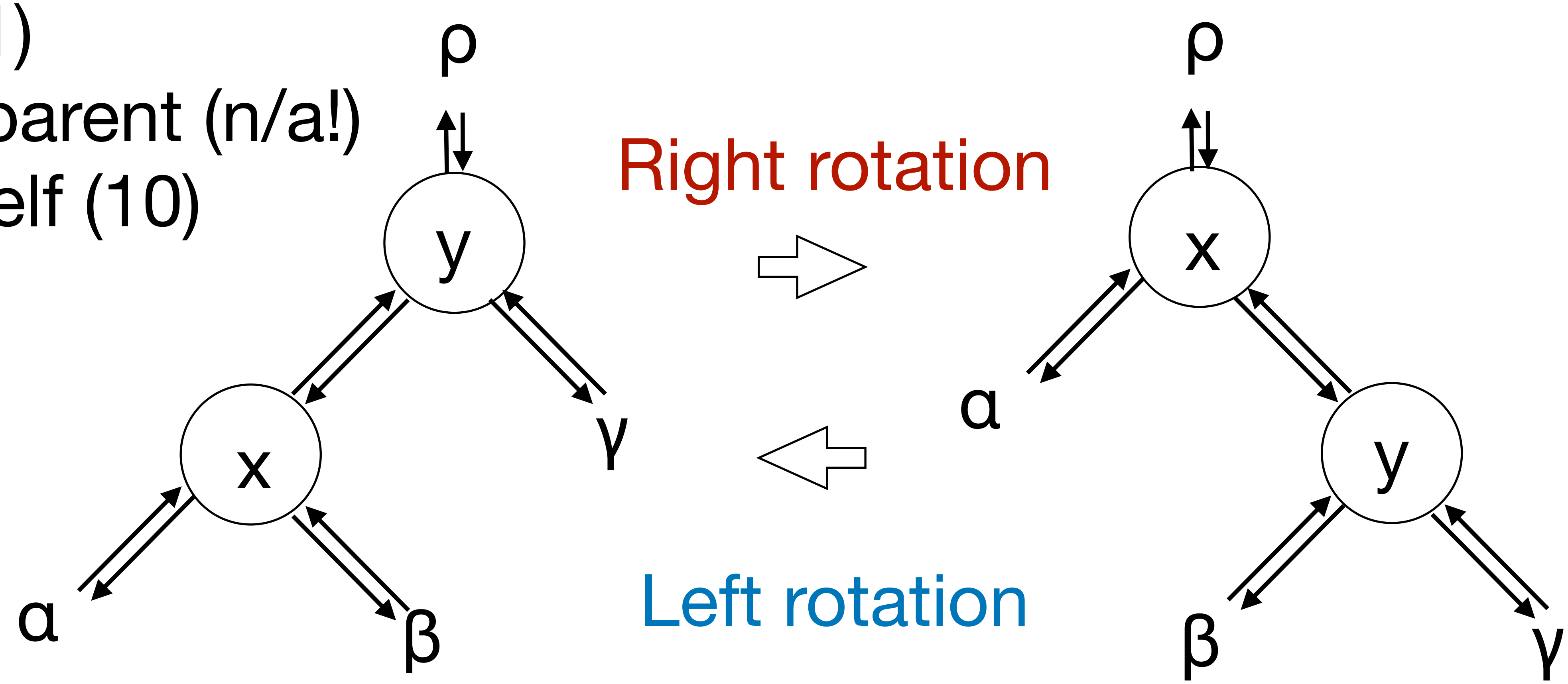


CLRS Fig 13.2, pg 313

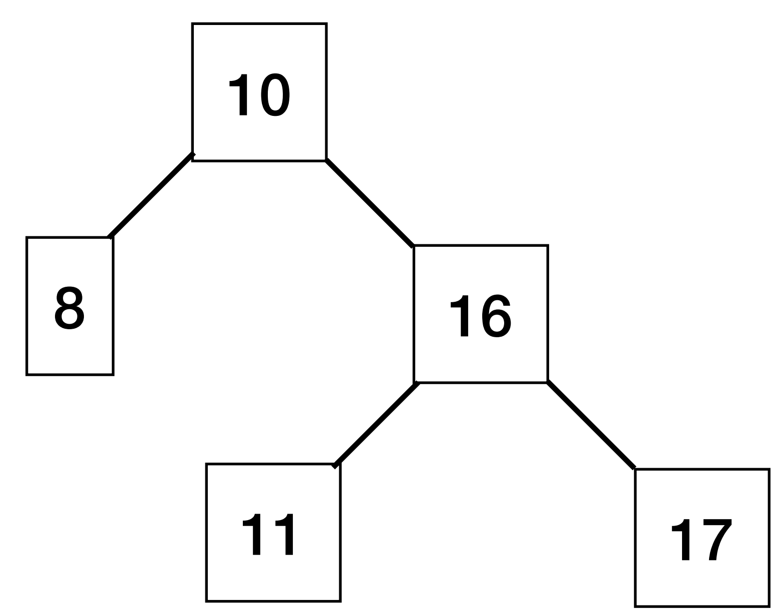
Tree rotations: DIY

Steps in left rotation (move y up to x's position):

1. Transfer β (11)
2. Transfer the parent (n/a!)
3. Transfer x itself (10)



Perform a **left** rotation
on the root of this tree:



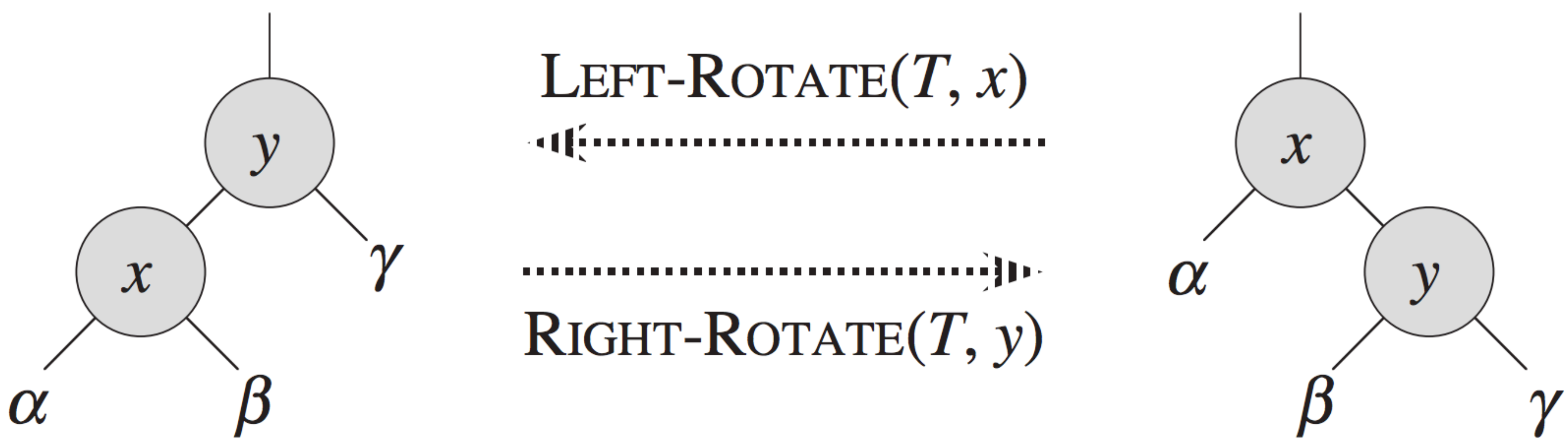
Tree Rotations

modify the structure without violating the BST property.

Steps in left rotation (move y up to its parent's position):

1. Transfer β: x's right subtree becomes y's old left subtree (β)
2. Transfer the parent: y's parent becomes x's old parent
3. Transfer x itself: x becomes y's left subtree

Details: need to update child, parent, and (possibly) root pointers.

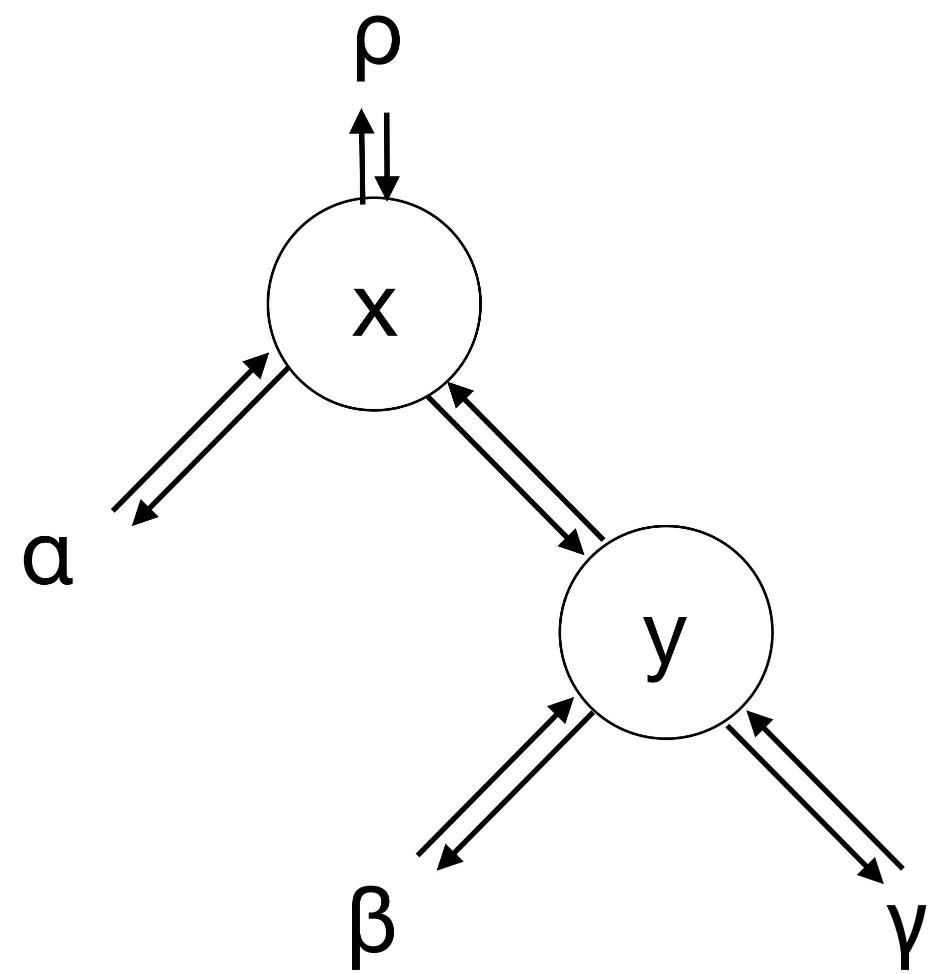


CLRS Fig 13.2, pg 313

Tree Rotations

Steps in left rotation (move y up to its parent's position):

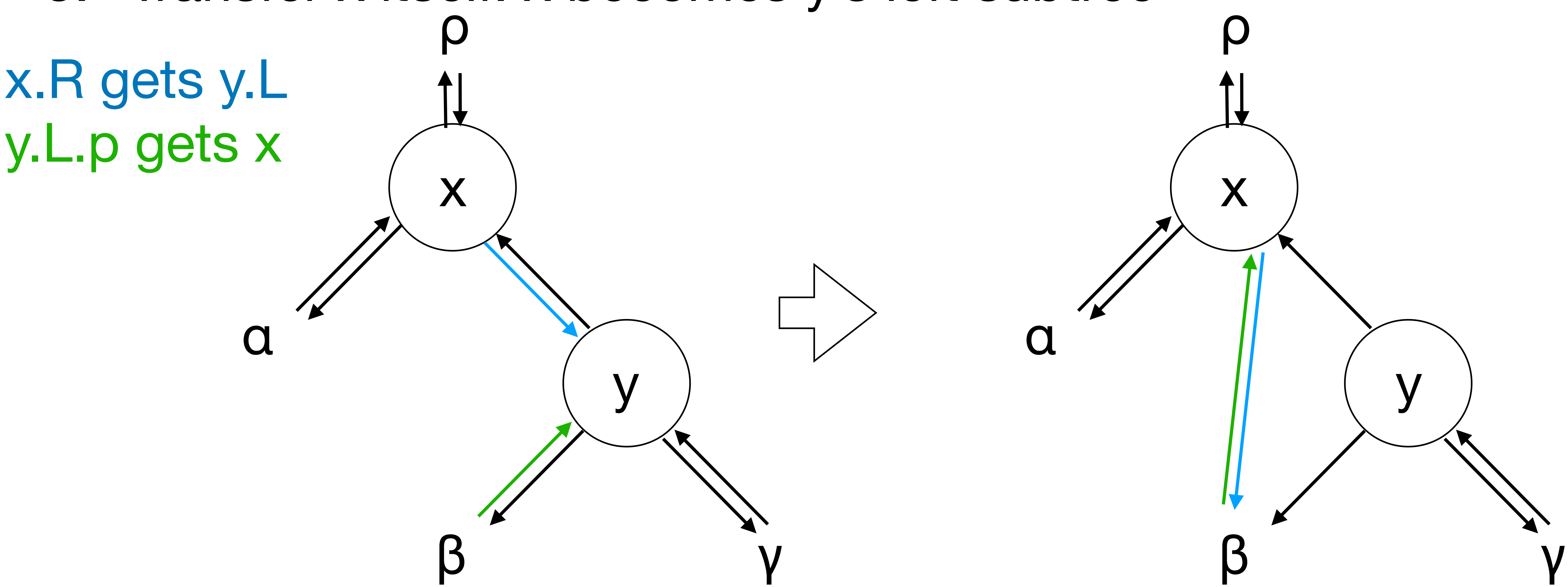
1. Transfer β: x's right subtree becomes y's old left subtree (β)
2. Transfer the parent: y's parent becomes x's old parent
3. Transfer x itself: x becomes y's left subtree



Tree Rotations

Steps in left rotation (move y up to its parent's position):

1. **Transfer β:** x's right subtree becomes y's old left subtree (β)
2. Transfer the parent: y's parent becomes x's old parent
3. Transfer x itself: x becomes y's left subtree

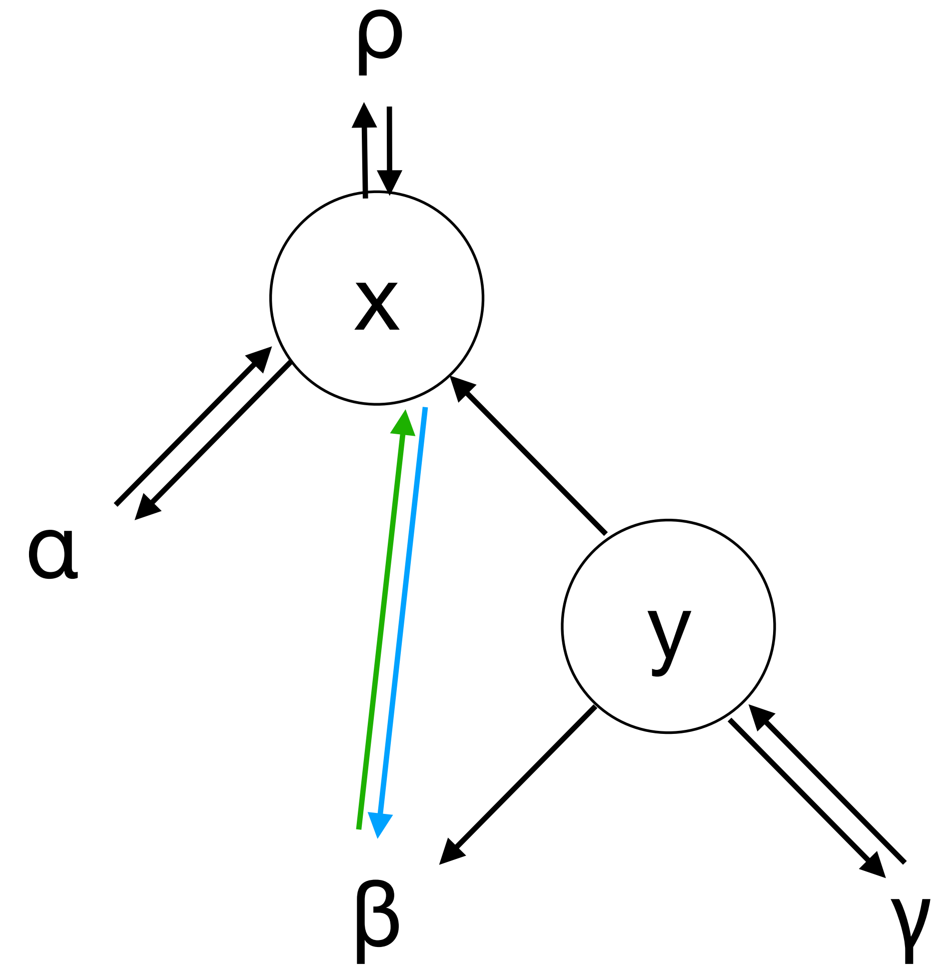


Tree Rotations

Steps in left rotation (move y up to its parent's position):

1. **Transfer β:** x's right subtree becomes y's old left subtree (β)
2. Transfer the parent: y's parent becomes x's old parent
3. Transfer x itself: x becomes y's left subtree

x.R gets y.L
y.L.p gets x

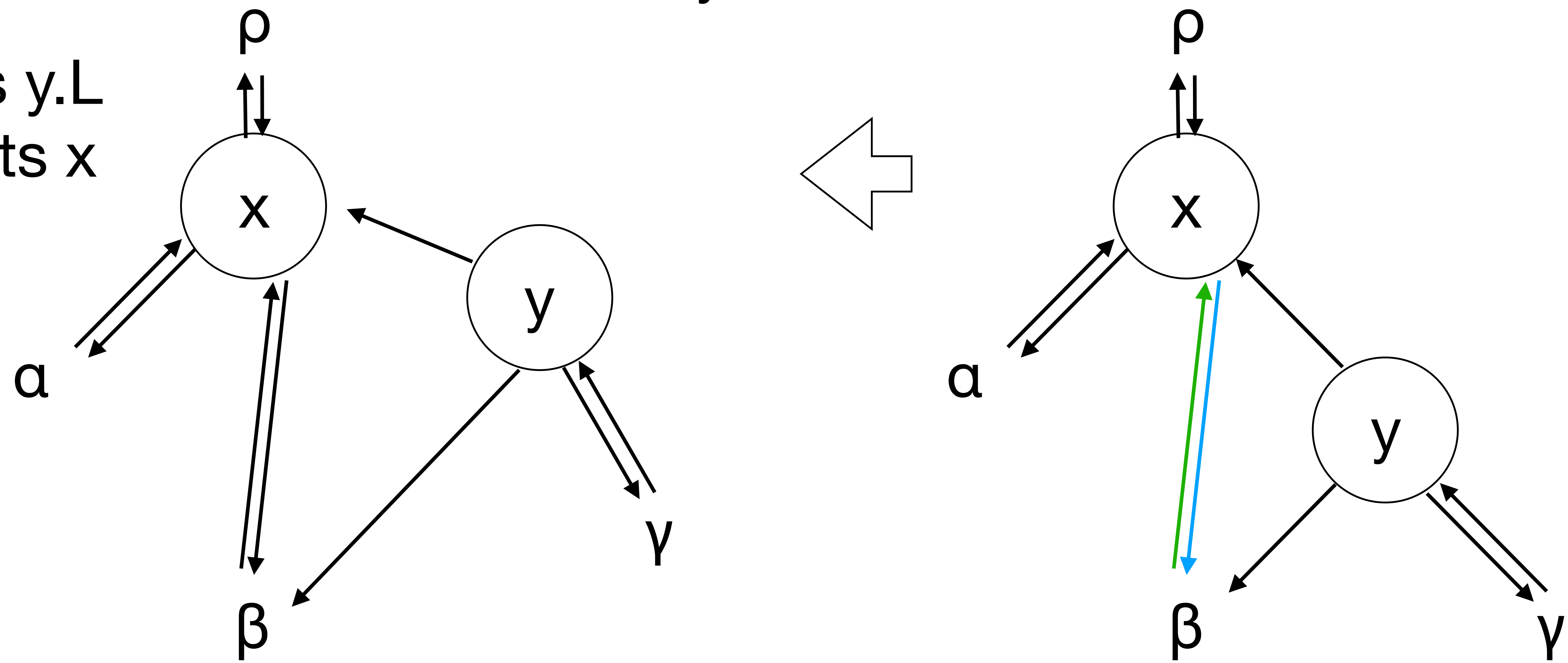


Tree Rotations

Steps in left rotation (move y up to its parent's position):

1. Transfer β: x's right subtree becomes y's old left subtree (β)
2. Transfer the parent: y's parent becomes x's old parent
3. Transfer x itself: x becomes y's left subtree

x.R gets y.L
y.L.p gets x



(**only** rearranged the picture)

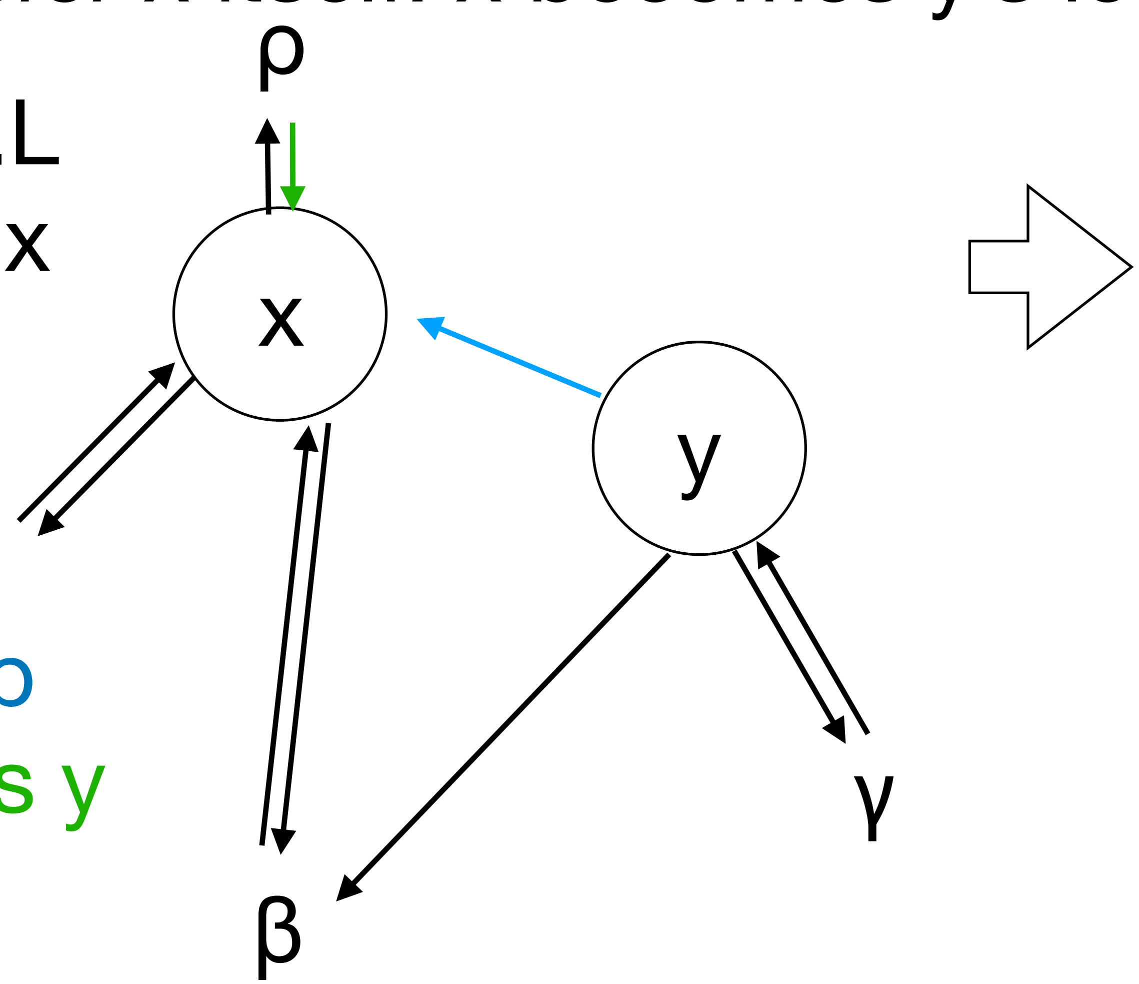
Tree Rotations

Steps in left rotation (move y up to its parent's position):

1. Transfer β: x's right subtree becomes y's old left subtree (β)
2. **Transfer the parent:** y's parent becomes x's old parent
3. Transfer x itself: x becomes y's left subtree

x.R gets y.L

y.L.p gets x



y.p gets x.p

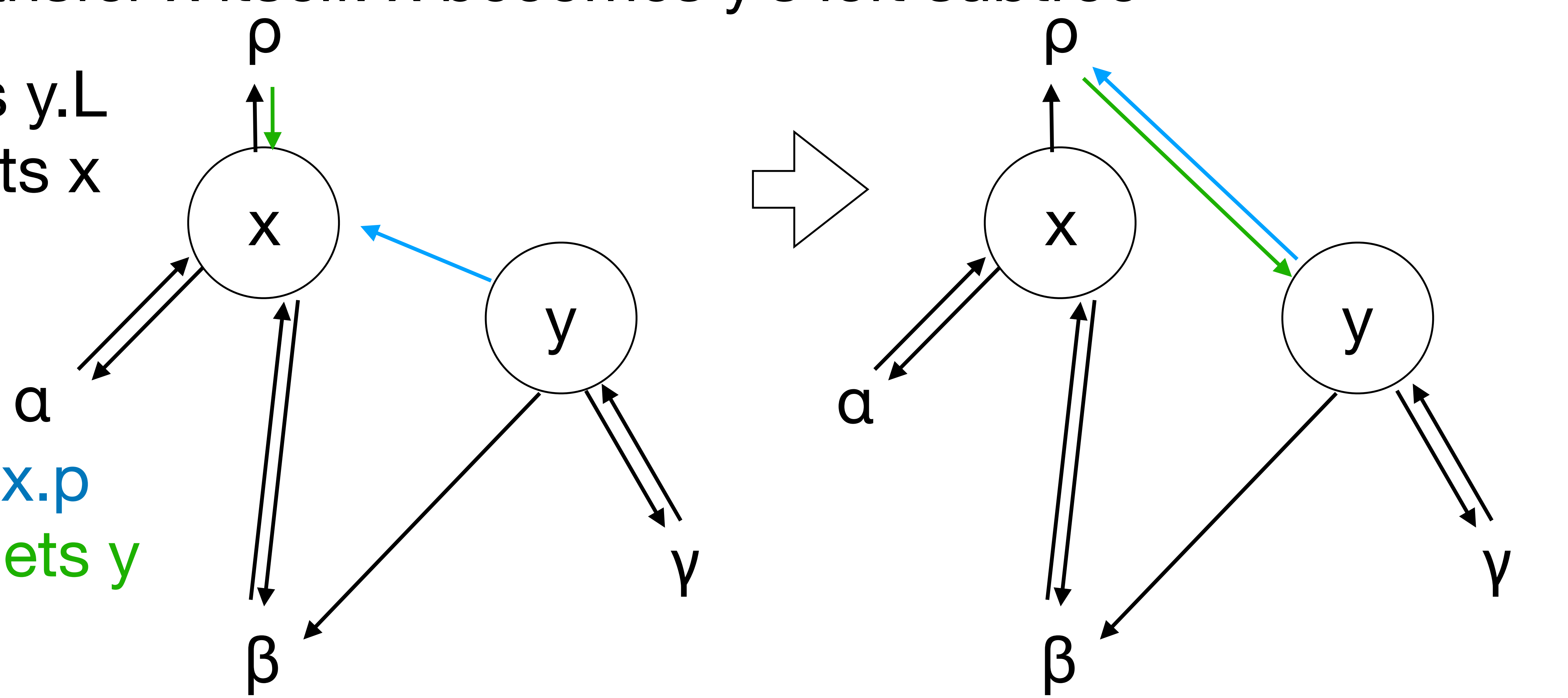
p.[L/R] gets y

Tree Rotations

Steps in left rotation (move y up to its parent's position):

1. Transfer β: x's right subtree becomes y's old left subtree (β)
2. **Transfer the parent:** y's parent becomes x's old parent
3. Transfer x itself: x becomes y's left subtree

x.R gets y.L
y.L.p gets x



y.p gets x.p
p.[L/R] gets y

(what if ρ is null / x was root?)

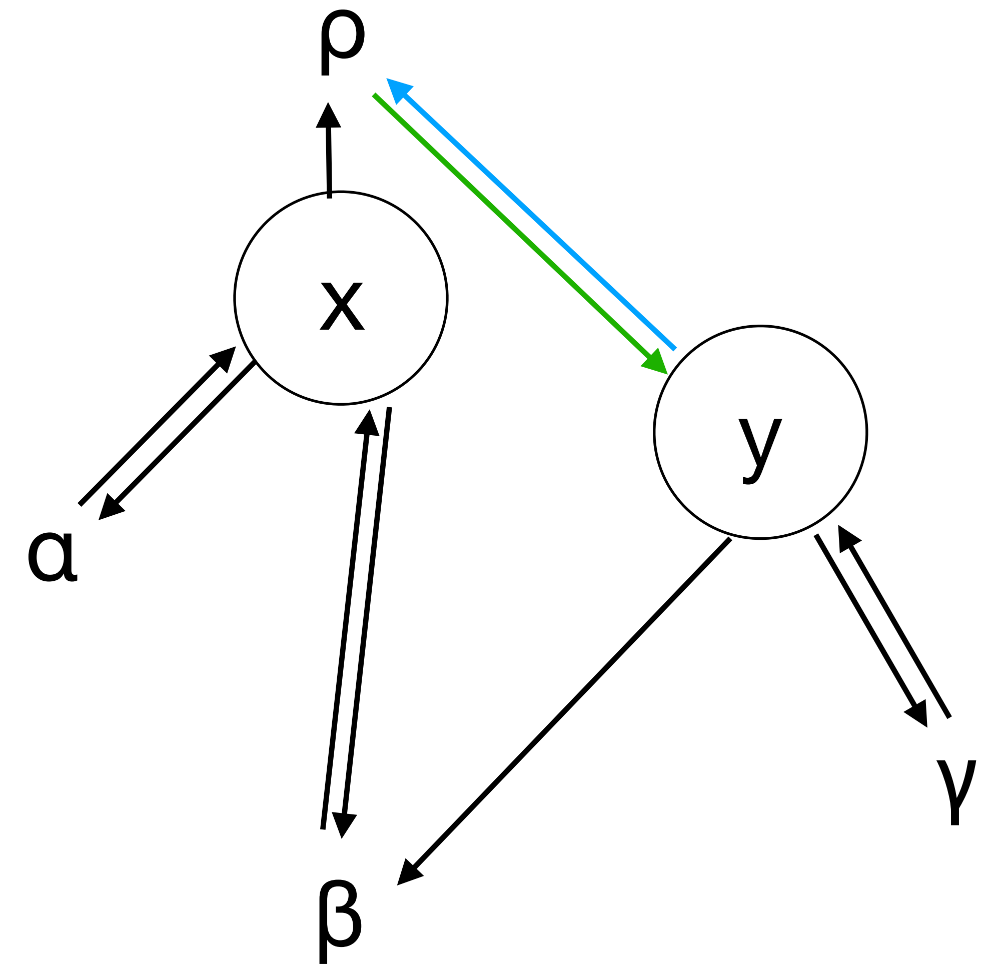
Tree Rotations

Steps in left rotation (move y up to its parent's position):

1. Transfer β: x's right subtree becomes y's old left subtree (β)
2. **Transfer the parent:** y's parent becomes x's old parent
3. Transfer x itself: x becomes y's left subtree

x.R gets y.L
y.L.p gets x

y.p gets x.p
p.[L/R] gets y



(what if ρ is null / x was root?)

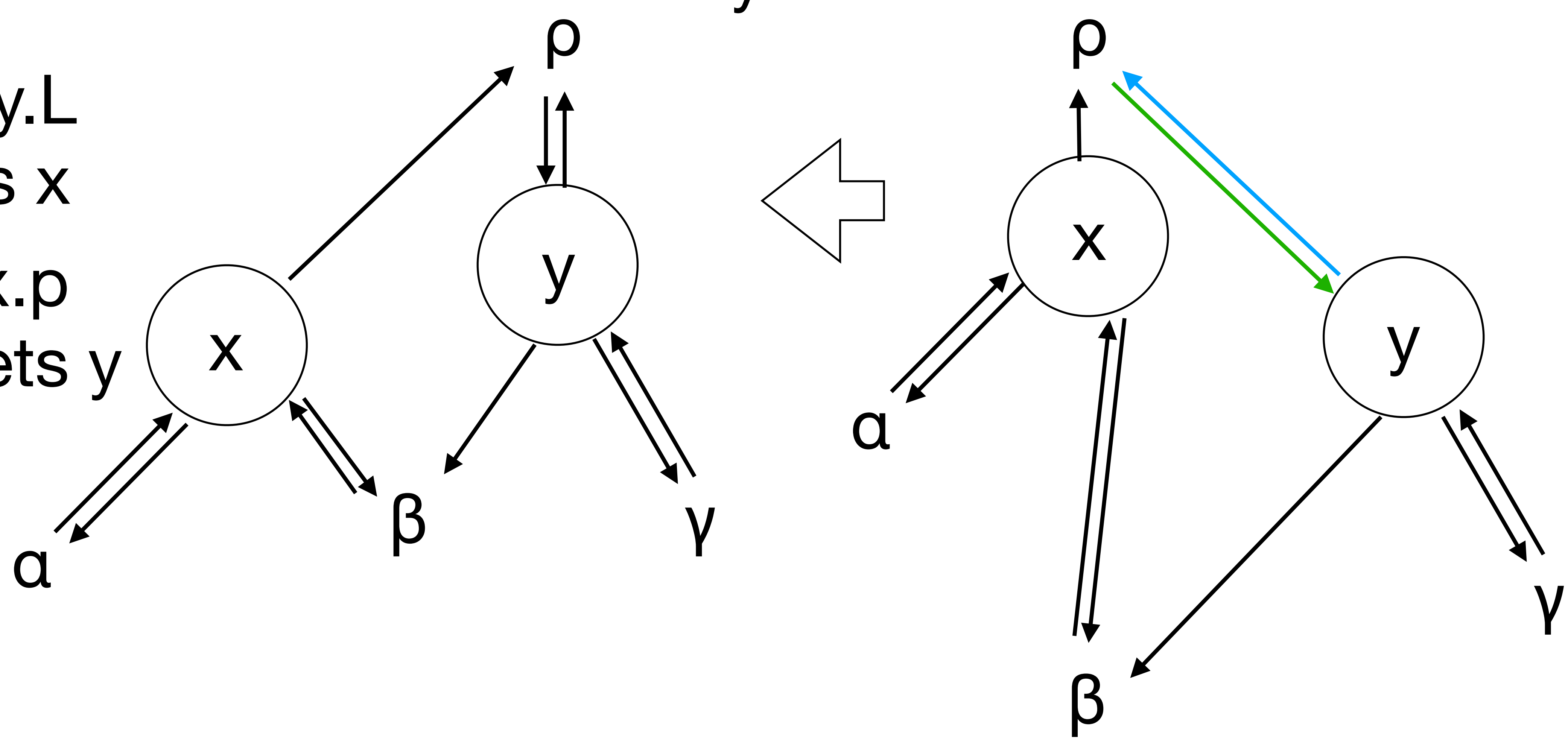
Tree Rotations

Steps in left rotation (move y up to its parent's position):

1. Transfer β: x's right subtree becomes y's old left subtree (β)
2. Transfer the parent: y's parent becomes x's old parent
3. Transfer x itself: x becomes y's left subtree

x.R gets y.L
y.L.p gets x

y.p gets x.p
p.[L/R] gets y



(**only** rearranged the picture)

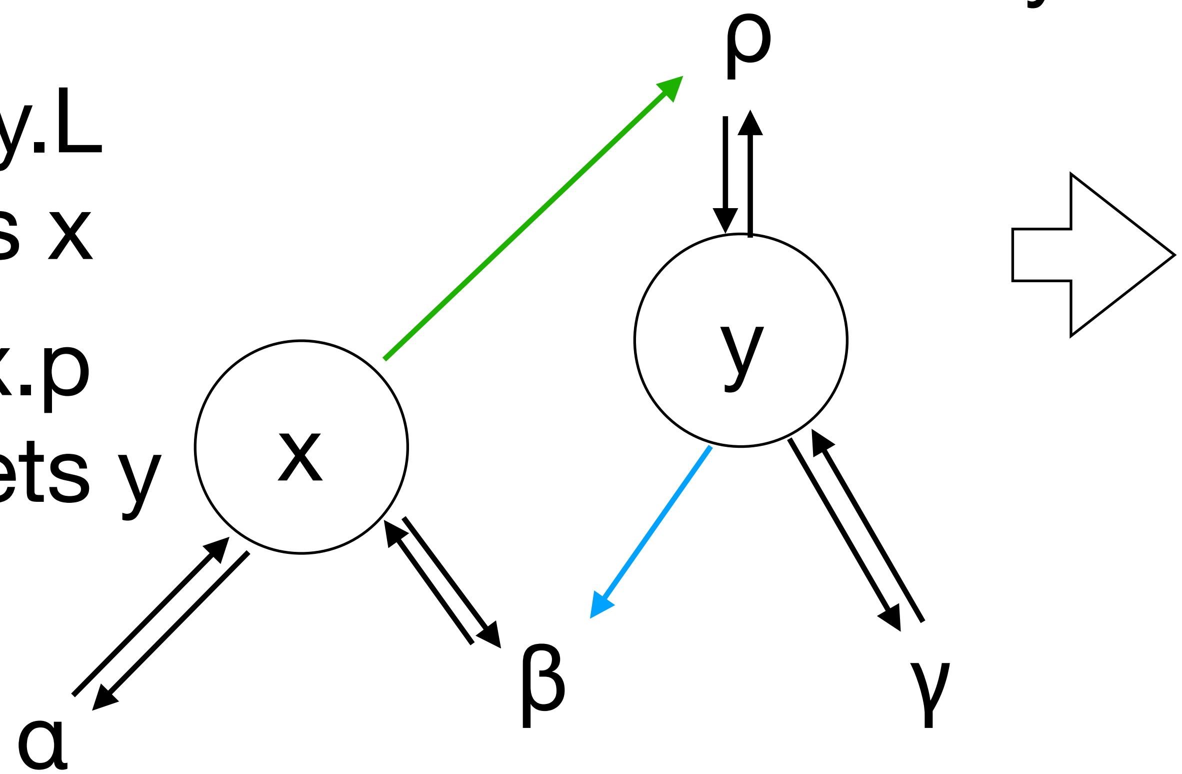
Tree Rotations

Steps in left rotation (move y up to its parent's position):

1. Transfer β: x's right subtree becomes y's old left subtree (β)
2. Transfer the parent: y's parent becomes x's old parent
3. **Transfer x itself:** x becomes y's left subtree

x.R gets y.L
y.L.p gets x

y.p gets x.p
p.[L/R] gets y



y.L gets x
x.p gets y

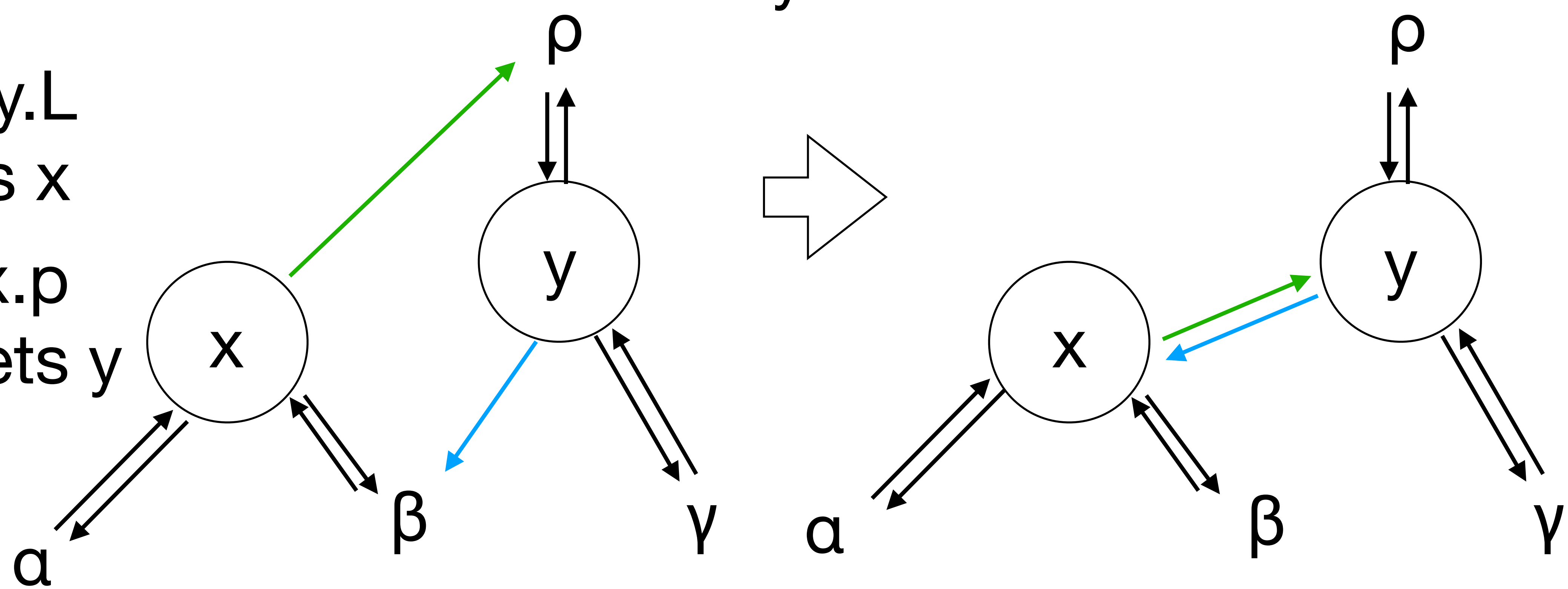
Tree Rotations

Steps in left rotation (move y up to its parent's position):

1. Transfer β: x's right subtree becomes y's old left subtree (β)
2. Transfer the parent: y's parent becomes x's old parent
3. **Transfer x itself:** x becomes y's left subtree

x.R gets y.L
y.L.p gets x

y.p gets x.p
p.[L/R] gets y



y.L gets x
x.p gets y

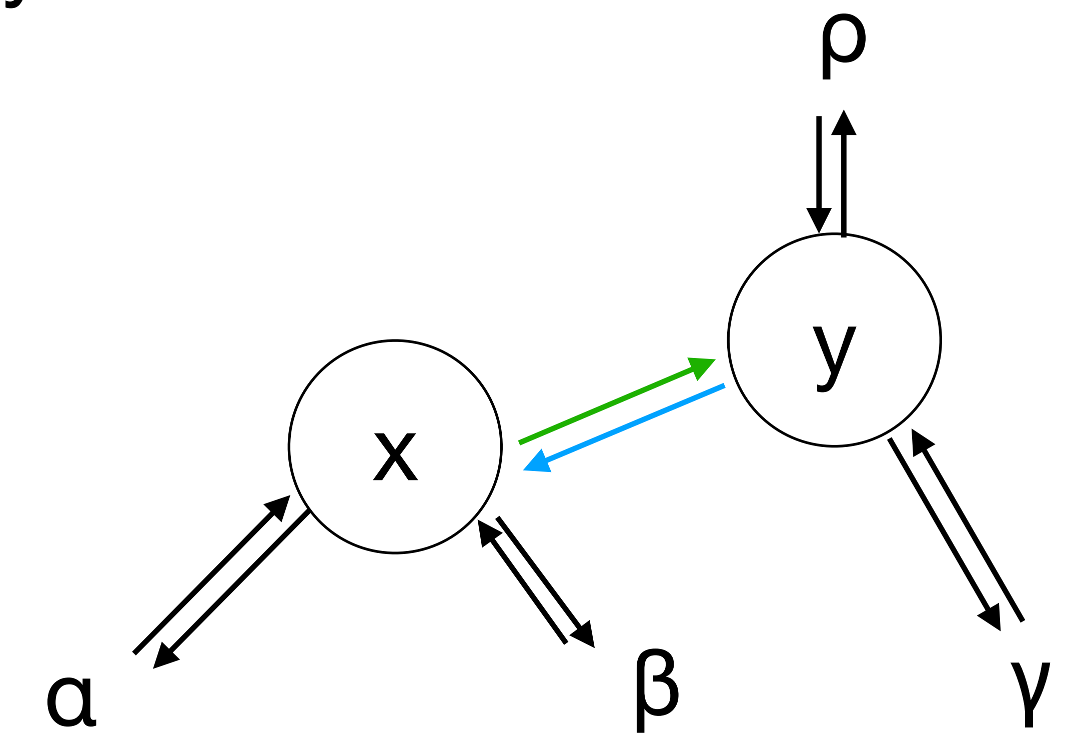
Tree Rotations

Steps in left rotation (move y up to its parent’s position):

1. Transfer β: x’s right subtree becomes y’s old left subtree (β)
2. Transfer the parent: y’s parent becomes x’s old parent
3. **Transfer x itself:** x becomes y’s left subtree

x.R gets y.L
y.L.p gets x

y.p gets x.p
p.[L/R] gets y



y.L gets x
x.p gets y

Tree Rotations

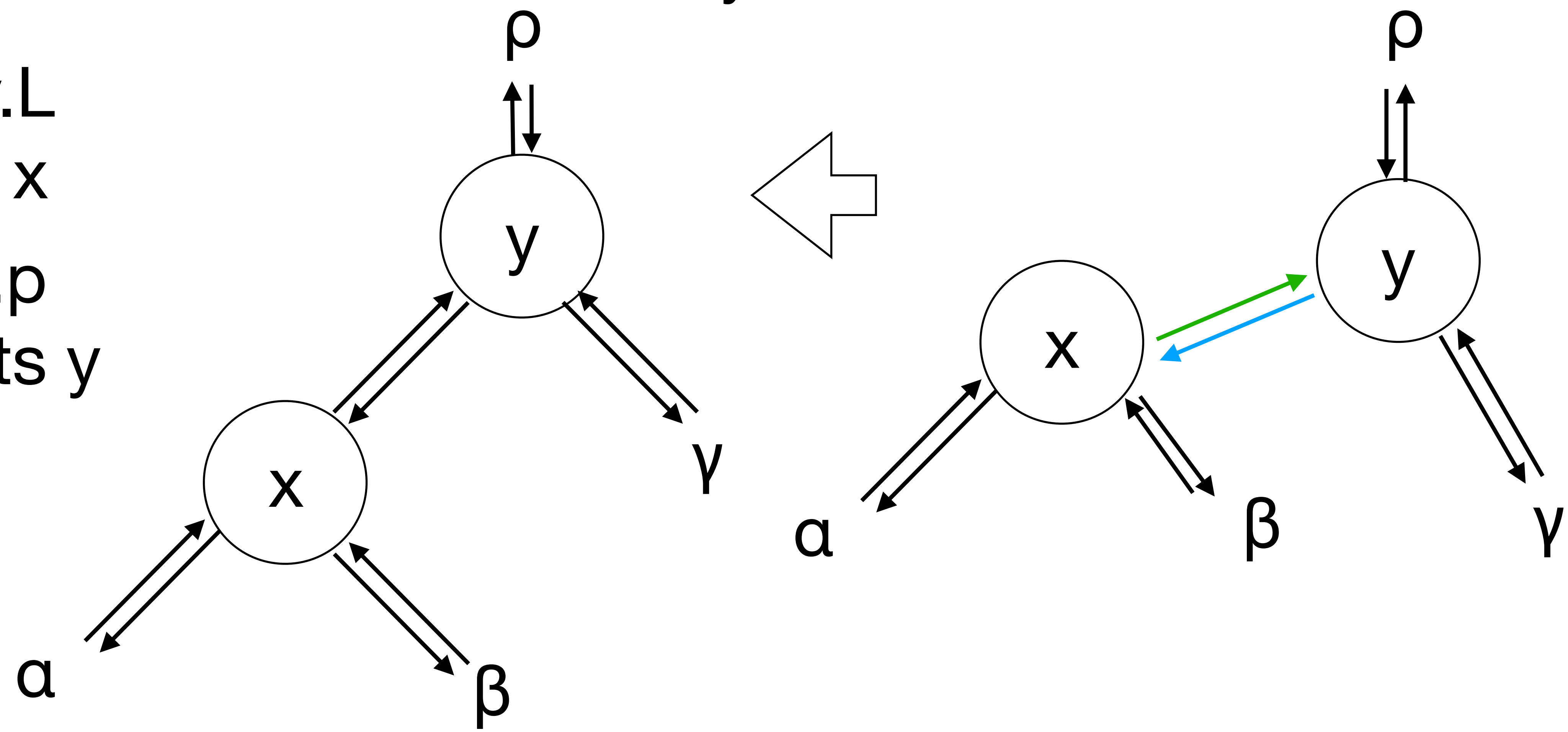
Steps in left rotation (move y up to its parent's position):

1. Transfer β: x's right subtree becomes y's old left subtree (β)
2. Transfer the parent: y's parent becomes x's old parent
3. Transfer x itself: x becomes y's left subtree

x.R gets y.L
y.L.p gets x

y.p gets x.p
p.[L/R] gets y

y.L gets x
x.p gets y



(**only** rearranged the picture)

Tree Rotations

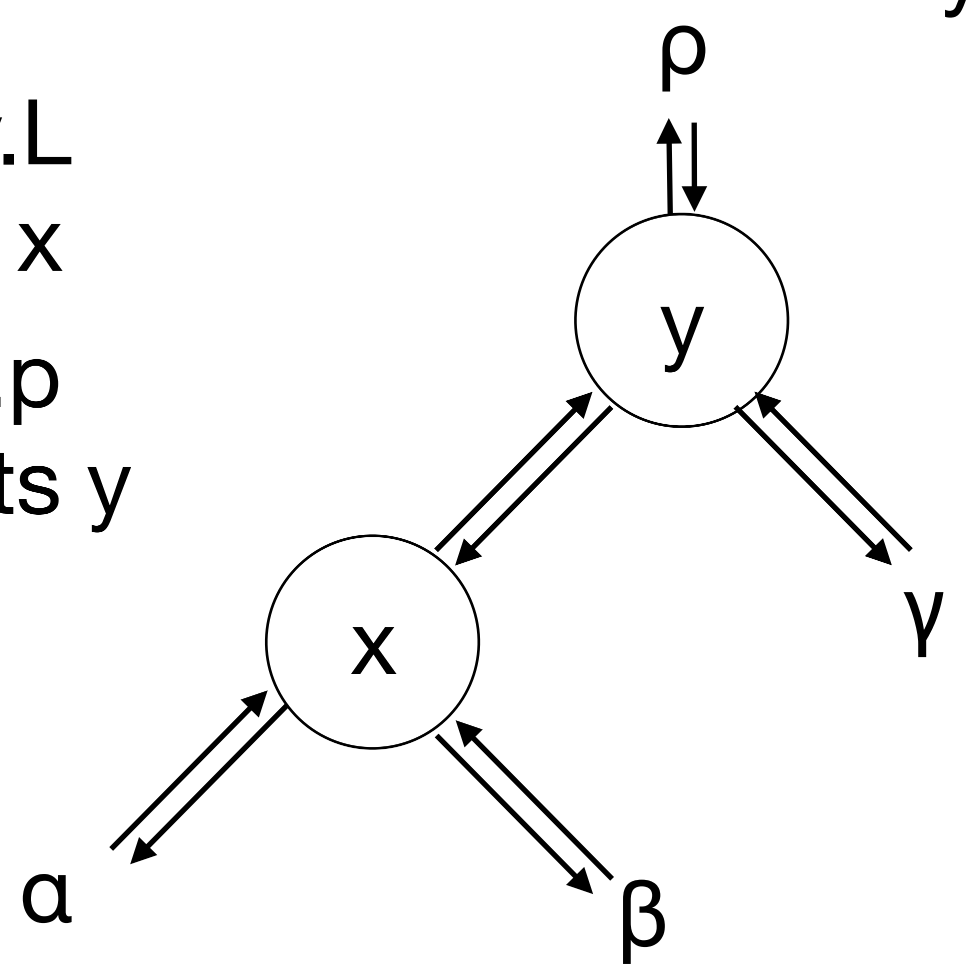
Steps in left rotation (move y up to its parent's position):

1. Transfer β: x's right subtree becomes y's old left subtree (β)
2. Transfer the parent: y's parent becomes x's old parent
3. Transfer x itself: x becomes y's left subtree

x.R gets y.L
y.L.p gets x

y.p gets x.p
p.[L/R] gets y

y.L gets x
x.p gets y



Tree Rotations

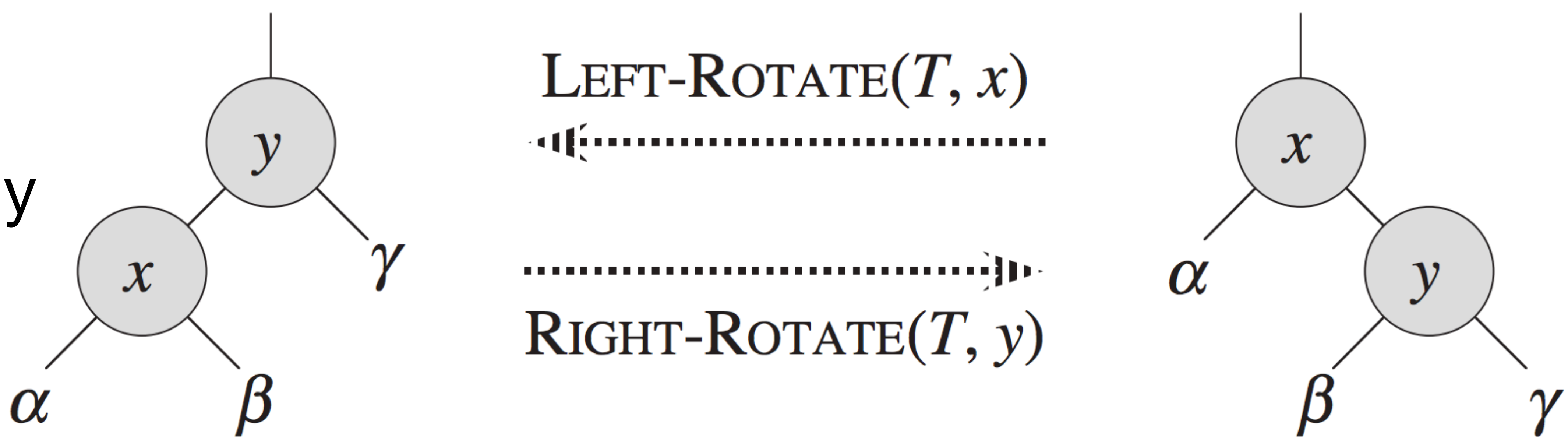
Steps in left rotation (move y up to its parent's position):

1. Transfer β: x's right subtree becomes y's old left subtree (β)
2. Transfer the parent: y's parent becomes x's old parent
3. Transfer x itself: x becomes y's left subtree

x.R gets y.L
y.L.p gets x

y.p gets x.p
p.[L/R] gets y

y.L gets x
x.p gets y



Overall Transformation

Pseudocode from CLRS

LEFT-ROTATE(T, x)

1. xfer β

```
y = x.right                    // set y
x.right = y.left               // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                      // link x's parent to y
```

2. xfer parent

```
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
```

3. xfer x

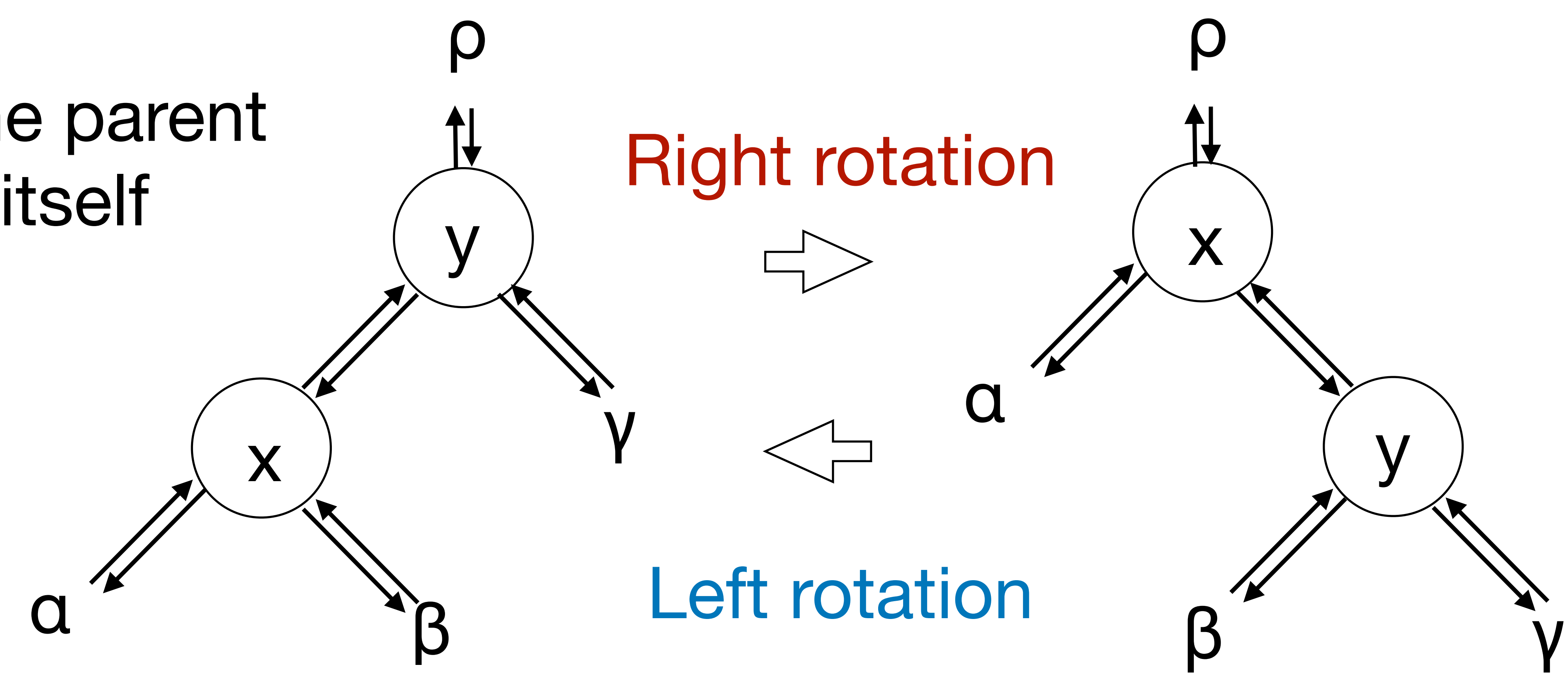
```
y.left = x                     // put x on y's left
x.p = y
```

Notational quirk: assume T.nil means “null”

Tree Rotations

Steps in **left** rotation (move y up to x's position):

1. Transfer β
2. Transfer the parent
3. Transfer x itself



x.R gets y.L
y.L.p gets x

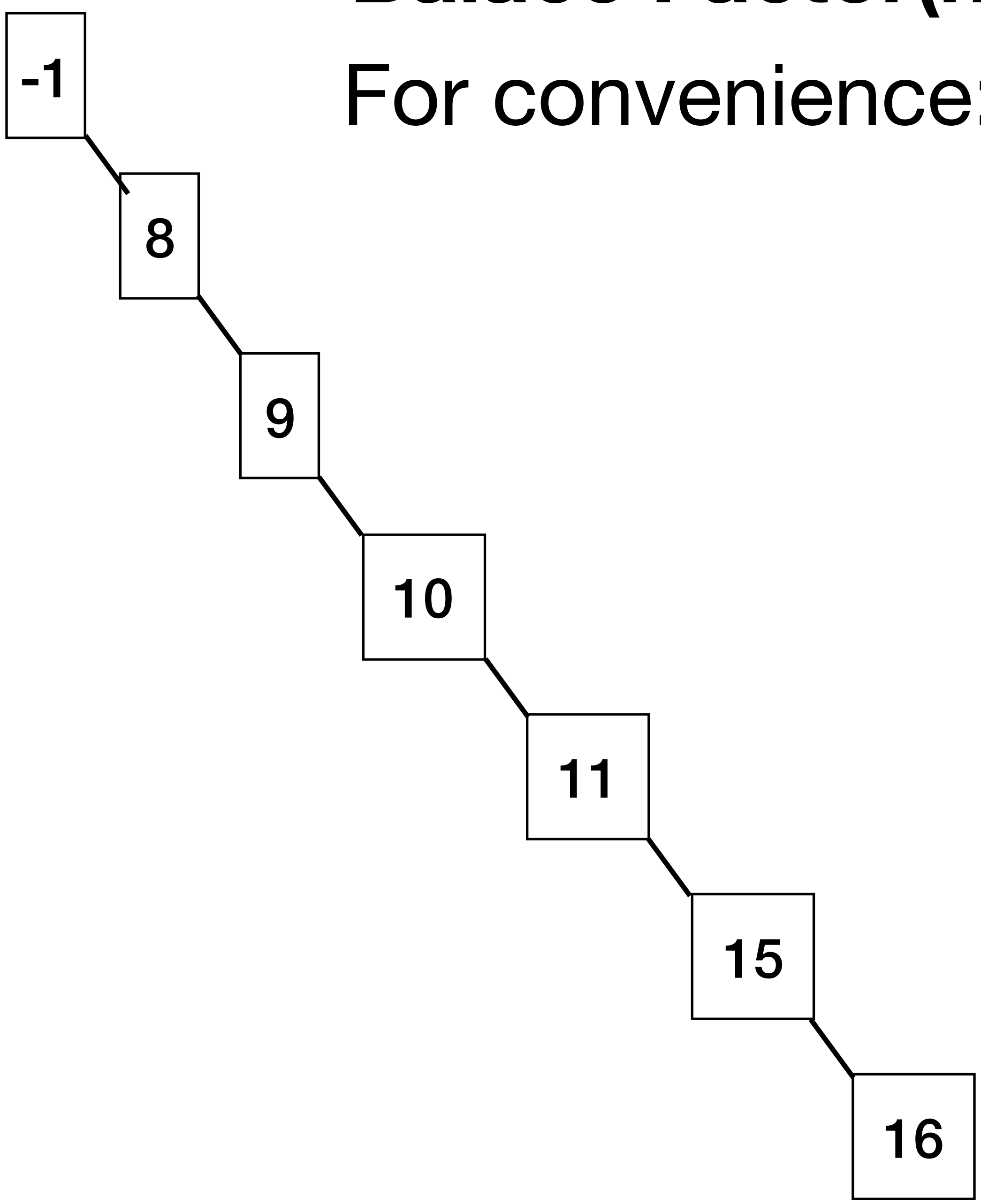
y.p gets x.p
p.[L/R] gets y

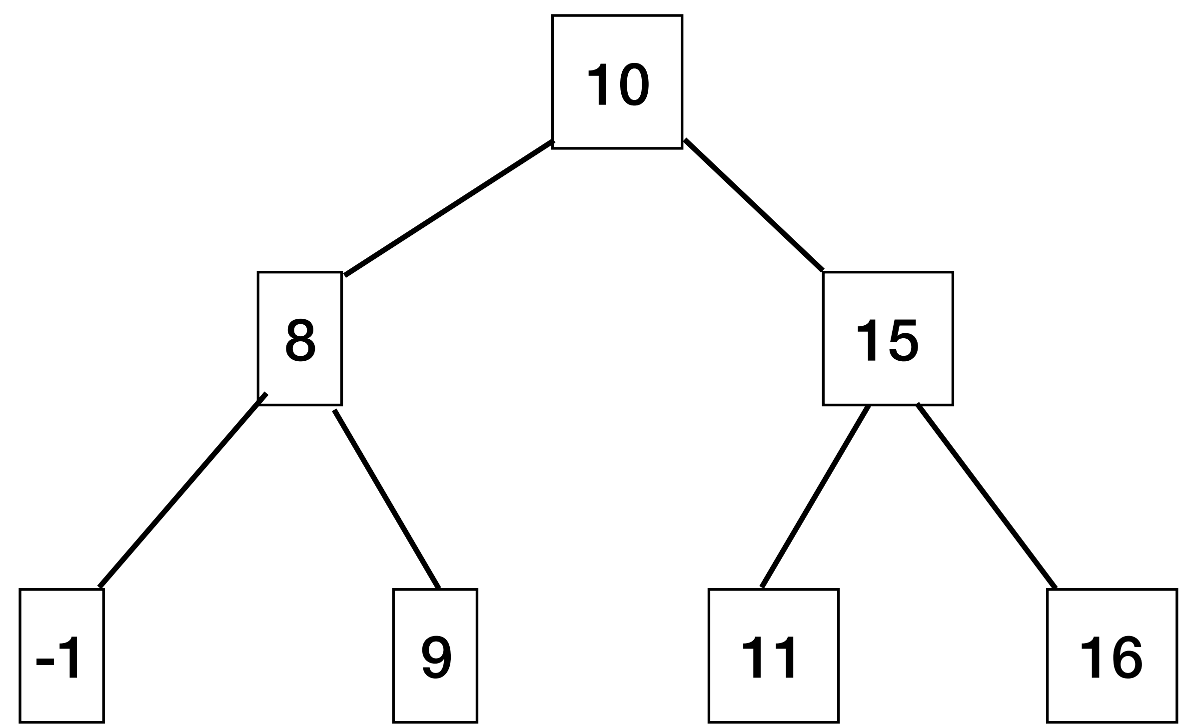
y.L gets x
x.p gets y

We can make a tree less bad. Let’s quantify badness:

Balace Factor(n) = height(n.right) - height(n.left)

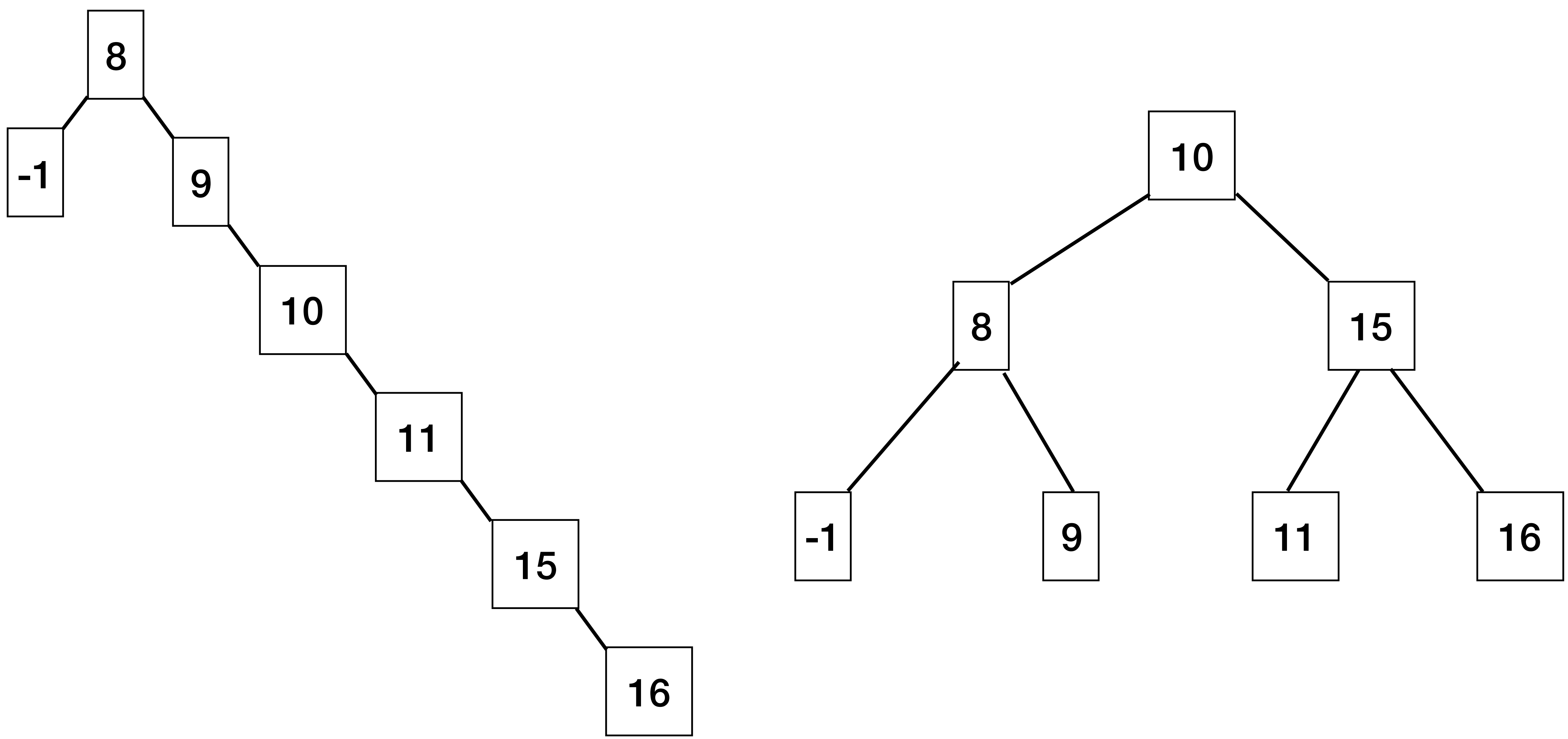
For convenience: define **height(null) = -1**





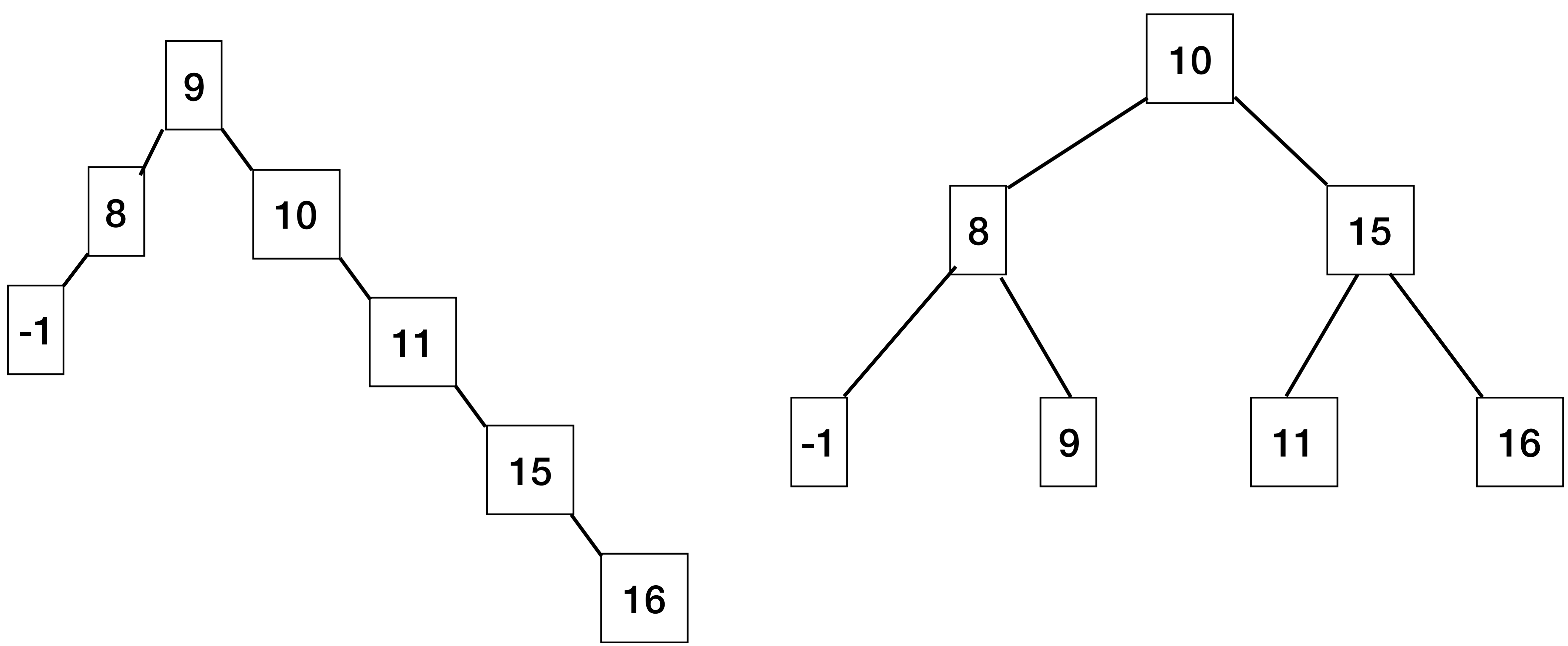
Can we improve balance?

Balace Factor(n) = height(n.right) - height(n.left)



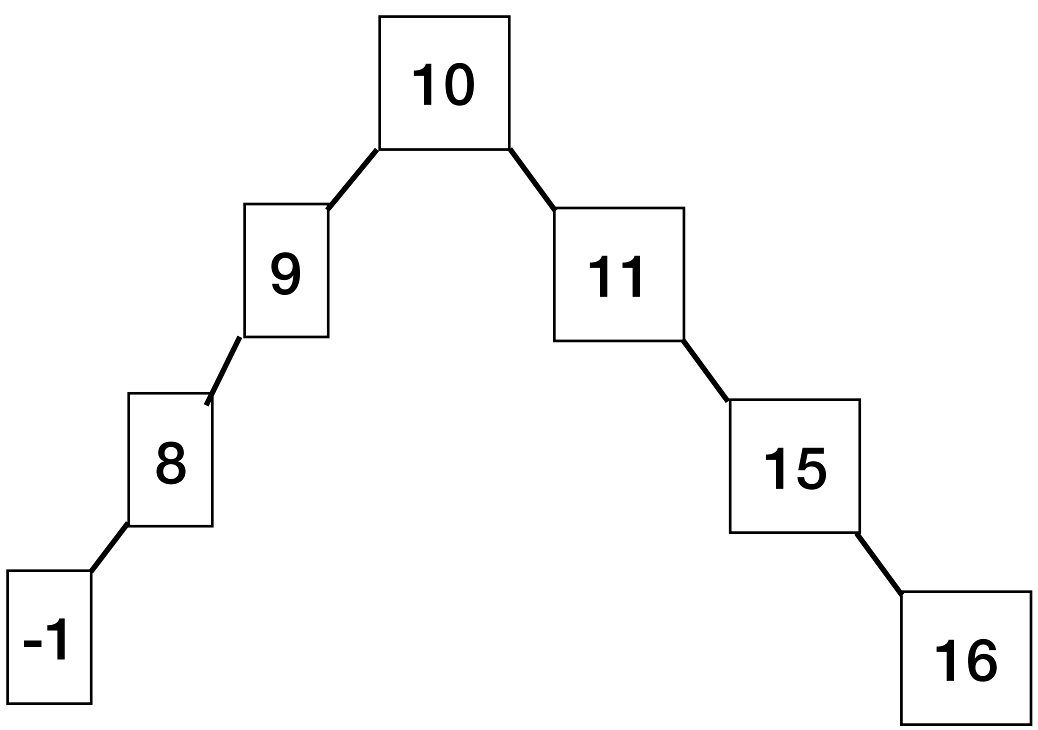
Can we improve balance?

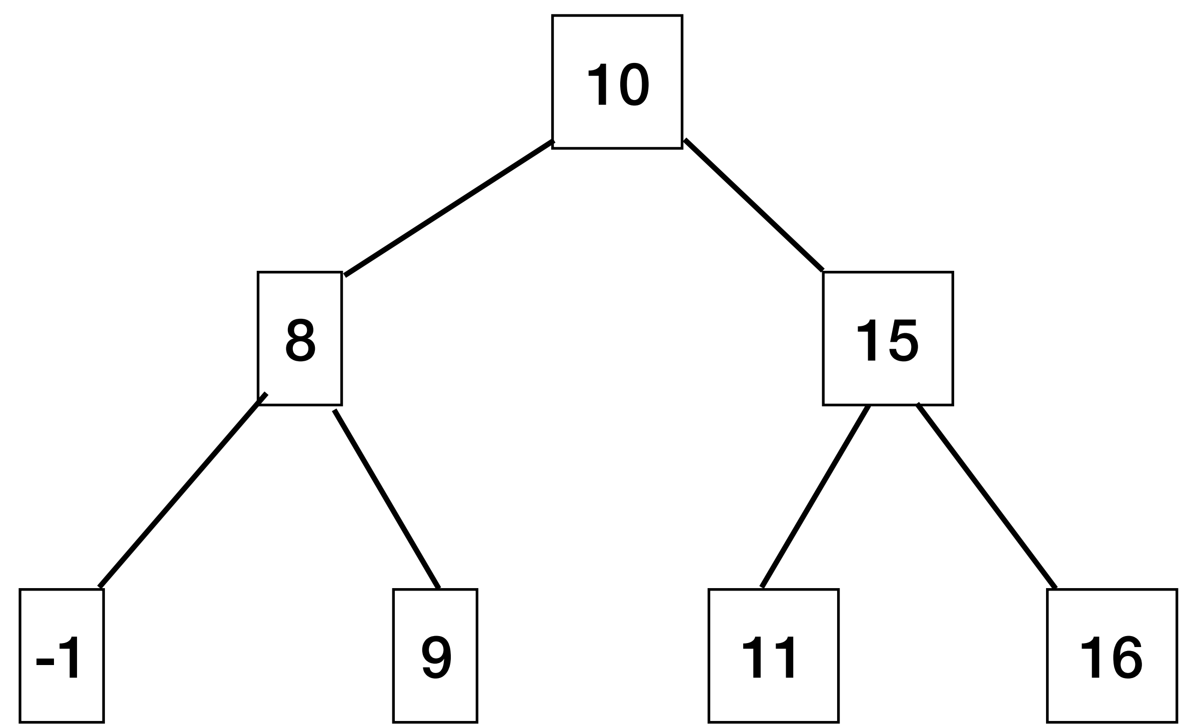
Balace Factor(n) = height(n.right) - height(n.left)



Can we improve balance?

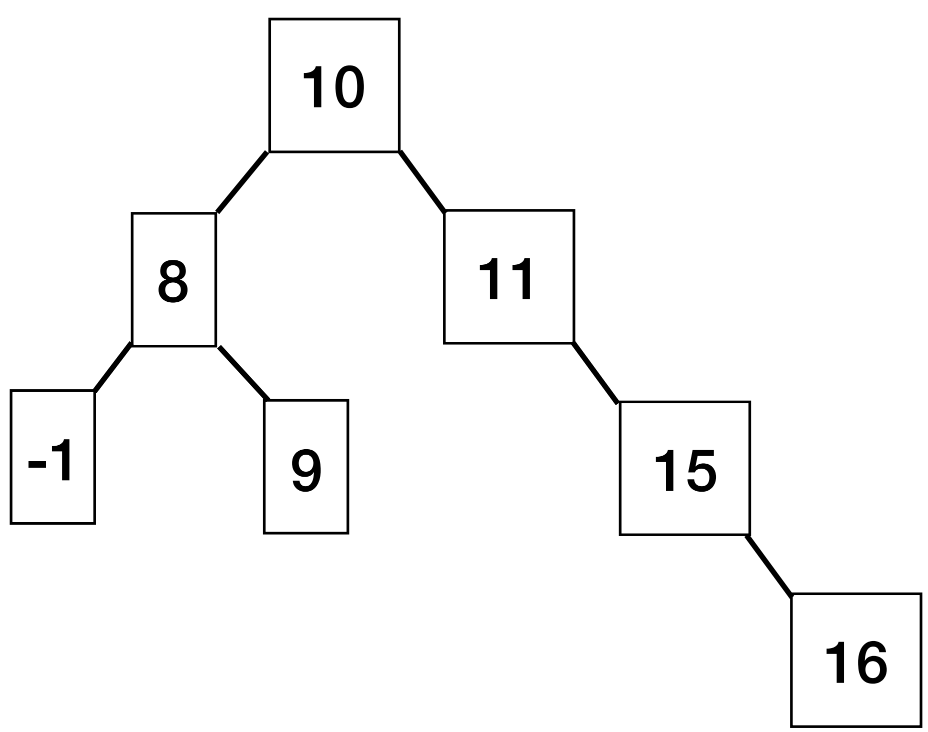
Balace Factor(n) = height(n.right) - height(n.left)

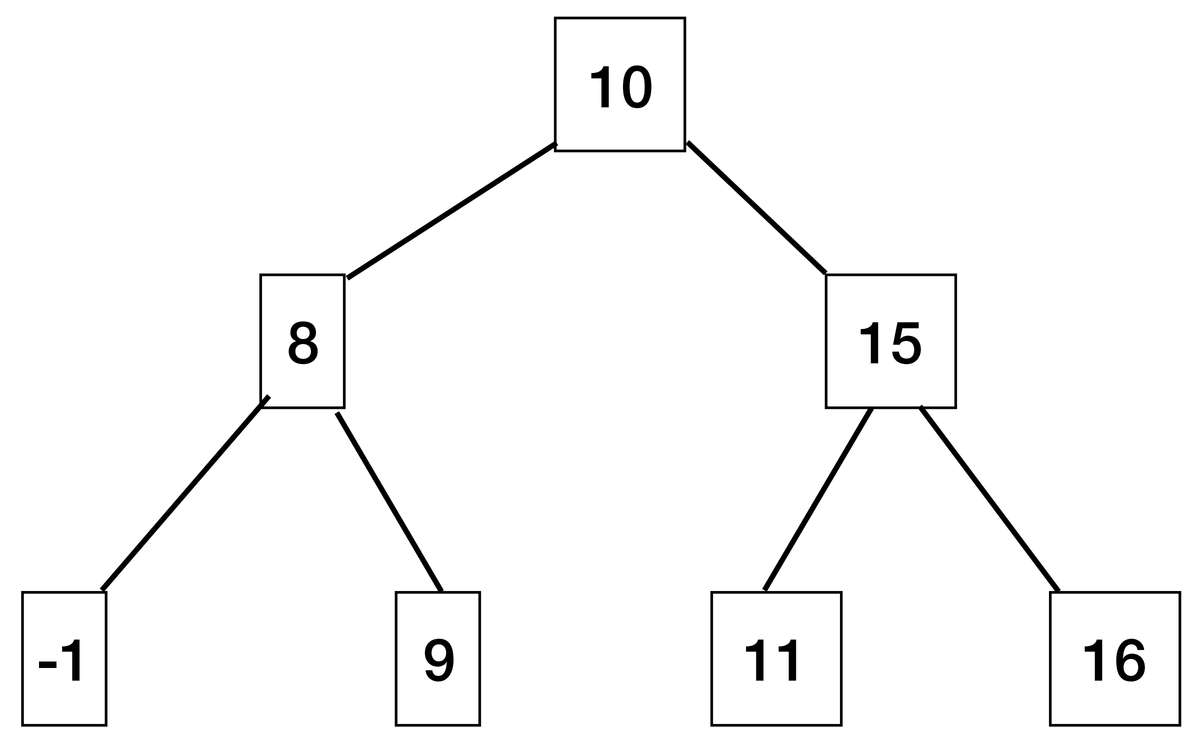




Can we improve balance?

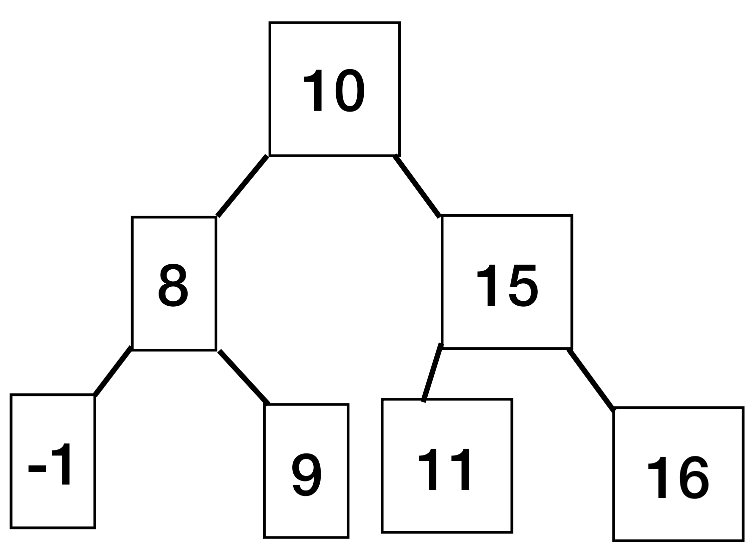
Balace Factor(n) = height(n.right) - height(n.left)

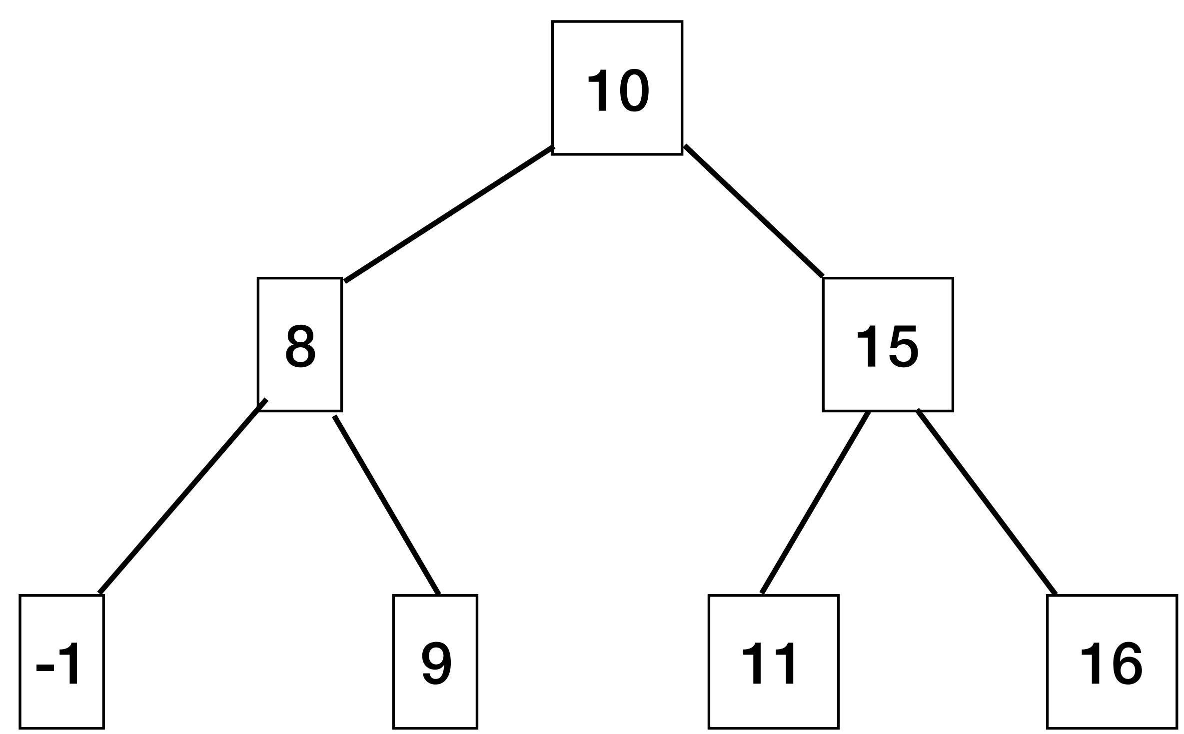




Can we improve balance?

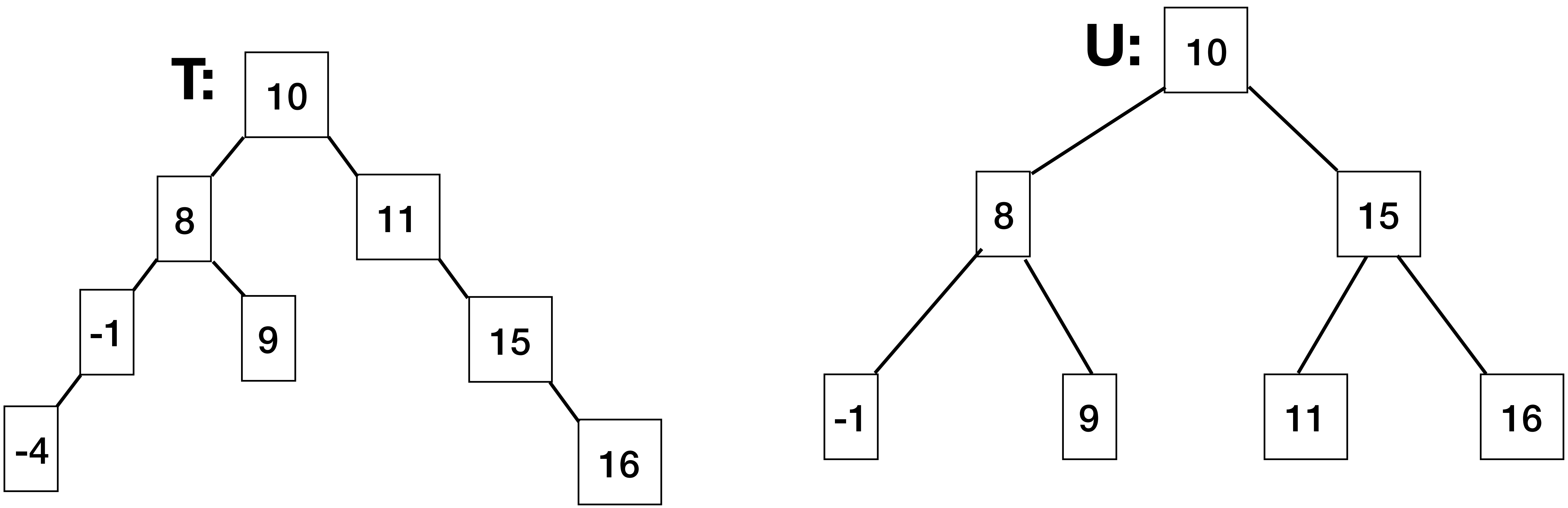
Balace Factor(n) = height(n.right) - height(n.left)





Balance Factor

Balace Factor b(n) = height(n.right) - height(left)



ABCD: What's the largest *absolute* balance factor of any node in each tree?

	T	U
A	0	0
B	2	1
C	2	0
D	1	1

AVL Trees

Balace Factor b(n) = height(n.right) - height(left)

- Devised by **A**delson-**V**elsky and **L**andis
- An AVL tree is a Binary Search Tree in which the following property holds:

AVL property: -1 <= b(n) <= 1 for all nodes n.