CSE 12:
Basic data structures and object-oriented design

Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Twelve
25 July 2012

More on generics.

Collections to hold data of type `T`

- Up to now we have discussed generics in its simplest usage -- store data of an arbitrary type `T` in a container.
    - This worked fine for lists/arrays/stacks/queues, in which we ignore any *order relations* among the elements.
- Sometimes, however, the type `T` cannot be "just any old `Object`" -- type `T` must sometimes *satisfy some conditions*.

Constraints on `T`

- An example of this is the `HeapImpl12` class you are building for P4.
    - The elements must all be `Comparable` -- the heap implementation needs to be able to call `compareTo(o)` on every element stored in the tree.
    - If we place no restrictions on `T`, then the Java compiler cannot guarantee that an arbitrary element of the `_nodeArray` will actually be `Comparable`.

Constraints on `T`

- Suppose we add three objects to a heap:

```
heap = new Heap12<Object>();
heap.add(“Michael”);  // OK: String is Comparable
heap.add(“Bolton”);  // OK: String is Comparable
heap.add(new Object()); // Not OK: Object not Comparable
```

- Internally, the **`HeapImpl12`** class will need to call **`compareTo`** on all objects to implement **`bubbleUp`** and **`trickleDown`**, e.g.:

```
if (_nodeArray[idx1].compareTo(_nodeArray[idx2]) < 0) {
  ...
}
```

But if `idx1` refers to the `Object` we added, this method will fail because `Object` does not implement the `Comparable` interface.

Bounds on type parameters

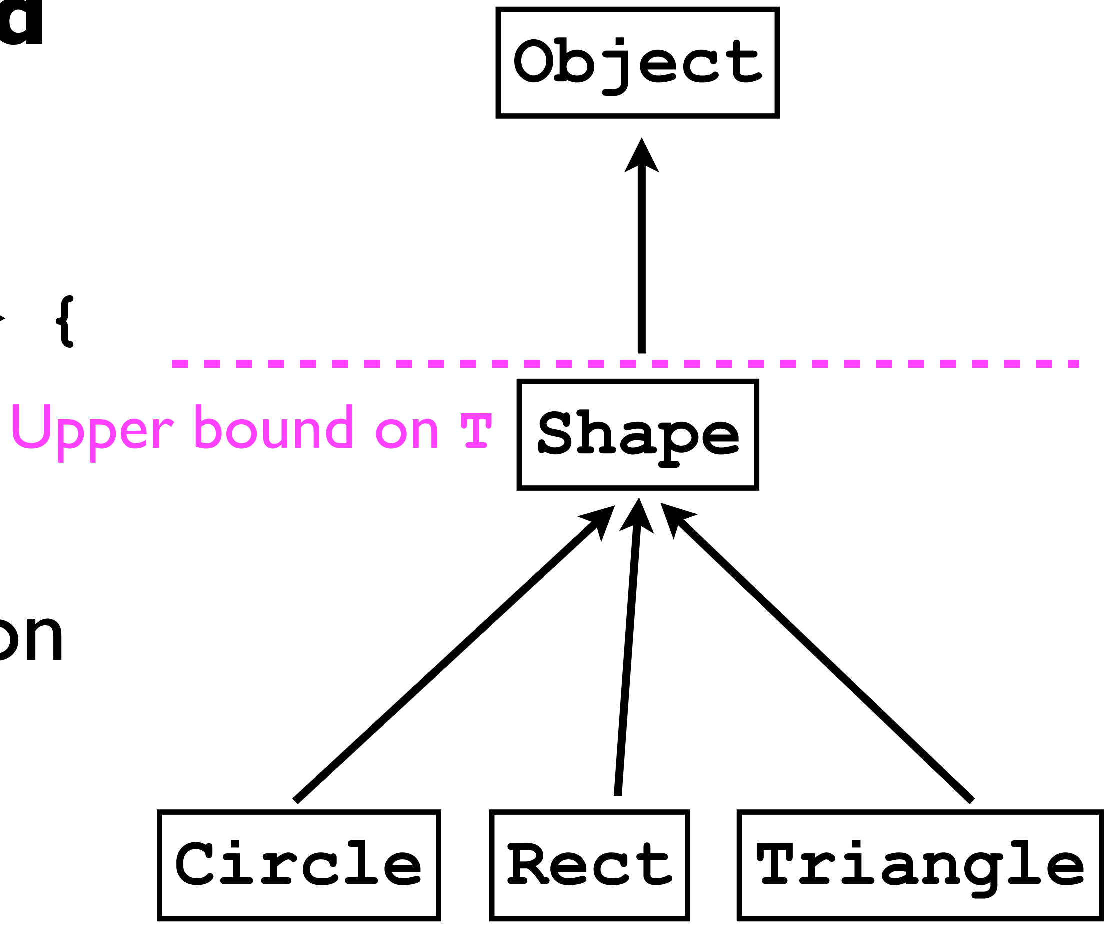
- What we want is a way of *enforcing* that the type parameter **`T`** allowed by the **`HeapImpl12`** class -- as well as the **`Heap12`** interface itself -- be of type **`Comparable`**.
- Java generics facilitates these constraints on **`T`** by supporting **bounds** on type parameters.
- Suppose, when implementing a generic class with type parameter **`T`**, we want to *ensure* that **`T`** must be *some subclass* of a class **`A`**.
    - *Example*: we want to implement a container for **`Shape`** objects -- we don't care what *particular* kind of **`Shapes`** they are, so long as they all *inherit from* the **`Shape`** class.

Bounds on type parameters

- To implement a generic class with the guarantee that type parameter `T` is a `Shape`, we can use an **upper bound** on `T`:

```
class MyContainer<T extends Shape> {
   ...
}
```

- Here, `Shape` is the **upper bound** on type parameter `T`.
  - `MyContainer` can only be instantiated when `T` is `Shape`, or *any sub-class of* `Shape`.



`Object`

Upper bound on `T`

`Shape`

`Circle`

`Rect`

`Triangle`

Bounds on type parameters

- Given this upper bound on `T`, the Java compiler will enforce that `T` be of type `Shape`:

```
MyContainer<Shape> container1 =
  new MyContainer<Shape>();  // OK

MyContainer<Circle> container2 =
  new MyContainer<Circle>();  // OK

MyContainer<Object> container4 =
  new MyContainer<Object>();  // Not OK

          Compiler error message:
            type parameter java.lang.Object is not within its bound
            MyContainer<Object> container4 = new MyContainer<Object>();

MyContainer<Student> container3 =
  new MyContainer<Student>();  // Not OK
```

Bounds on type parameters

- We can also require that type **`T`** *implement* some interface.
    - For example, a **`HeapImpl12`** should only store elements that are all **`Comparable`**.
- Java generics gives us this power:

```
class HeapImpl12<T extends Comparable> implements Heap12<T> {
   ...
}
```

- The "**`extends Comparable`**" enforces that any **`T`** we pass in as the type parameter *must* be of type **`Comparable`**.
    - Since **`Comparable`** is an *interface*, this means that type **`T`** must *implement* the interface **`Comparable`** (even though we use the word "**`extends`**").

Bounds on type parameters

- With this restriction on **`T`** in place, we can no longer instantiate a **`HeapImpl12`** with a type parameter **`T`** that does not implement **`Comparable`**:

```
// String and Integer are both Comparable
HeapImpl12<String> heap1 = new HeapImpl12<String>();  // OK
HeapImpl12<Integer> heap2 = new HeapImpl12<Integer>(); // OK

// Next line won't compile because Object is not Comparable
HeapImpl12<Object> heap3 = new HeapImpl12<Object>();
```

- The Java *compiler* will prevent us from instantiating a heap with a non-**`Comparable`** type.

- We may also wish to define the *interface* **`Heap12`** to accept only those types **`T`** that implement **`Comparable`**:

```
interface Heap12<T extends Comparable> {
  ...
}
```

Bounds on type parameters

- In the previous example, `Comparable` was the upper bound of `T`.
- The `Comparable` interface takes a type parameter of its own.

```
interface Comparable<U> {
  int compareTo (U o);
}
```

(In the previous example, we used the `Comparable` interface in “compatibility mode”, where we did not specify `U`).

- The type parameter `U` specifies what kinds of objects `o` we should be able to compare to.

Bounds on type parameters

- By offering **bounds** on type parameters, Java also gives us the power to define what kinds of objects `U` we can `compareTo`, *in terms of the type* `T` *we've already defined.*
- Example:

```
class HeapImpl12<T extends Comparable<T>> ... {
  ...
}
```

- Here, we require that whatever type `T` the `HeapImpl12` is instantiated with, it *must* be `Comparable` to *other objects of type* `T`.

Bounds on type parameters

- Consider the following example:

```
class B { }
class A implements Comparable<B> {
  int compareTo (B o) {
    return 0;
  }
}
```

- Given the definitions above, an object of type A can *only* be compared to objects of type **`B`**.

```
final A a = new A();
final B b = new B();
final int result = a.compareTo(b);  // OK
```

  - We *cannot* compare **`a`** to another object of type **`A`**!

Bounds on type parameters

- Given our definition of HeapImpl12,

  ```
  class HeapImpl12<T extends Comparable<T>> ... {
     ..
  }
  ```

  if we try to instantiate a **`HeapImpl12`** with **`A`** as the type parameter...

  ```
  HeapImpl12<A> heap = new HeapImpl12<A>();
  ```

  ... the compiler will complain:

  ```
    type parameter A is not within its bound
     HeapImpl12<A> h = new HeapImpl12<A>();
  ```

- This error occurs because, even though **`A`** is **`Comparable`** to *something* (**`B`**), it is not **`Comparable<A>`**.

Bounds on type parameters

- On the other hand,
  - `String` implements `Comparable<String>`
  - `Integer` implements `Comparable<Integer>`
- Both `String` and `Integer` would be accepted as type parameters for `HeapImpl12`:

```
HeapImpl12<String> h1 = new HeapImpl12<String>();
HeapImpl12<Integer> h2 = new HeapImpl12<Integer>();
```

Both are OK

Bounds on type parameters

- While useful, our current definition of **`HeapImpl12`** is a bit *overly restrictive*.
- Consider a hierarchy of **`Shape`** classes:

```
class Shape implements Comparable<Shape> {
   int compareTo (Shape o) { ... }
}
class Rectangle extends Shape {
  ...
}
```

- The **`Rectangle`** class inherits the **`compareTo (Shape o)`** method from its parent **`Shape`** class.

Bounds on type parameters

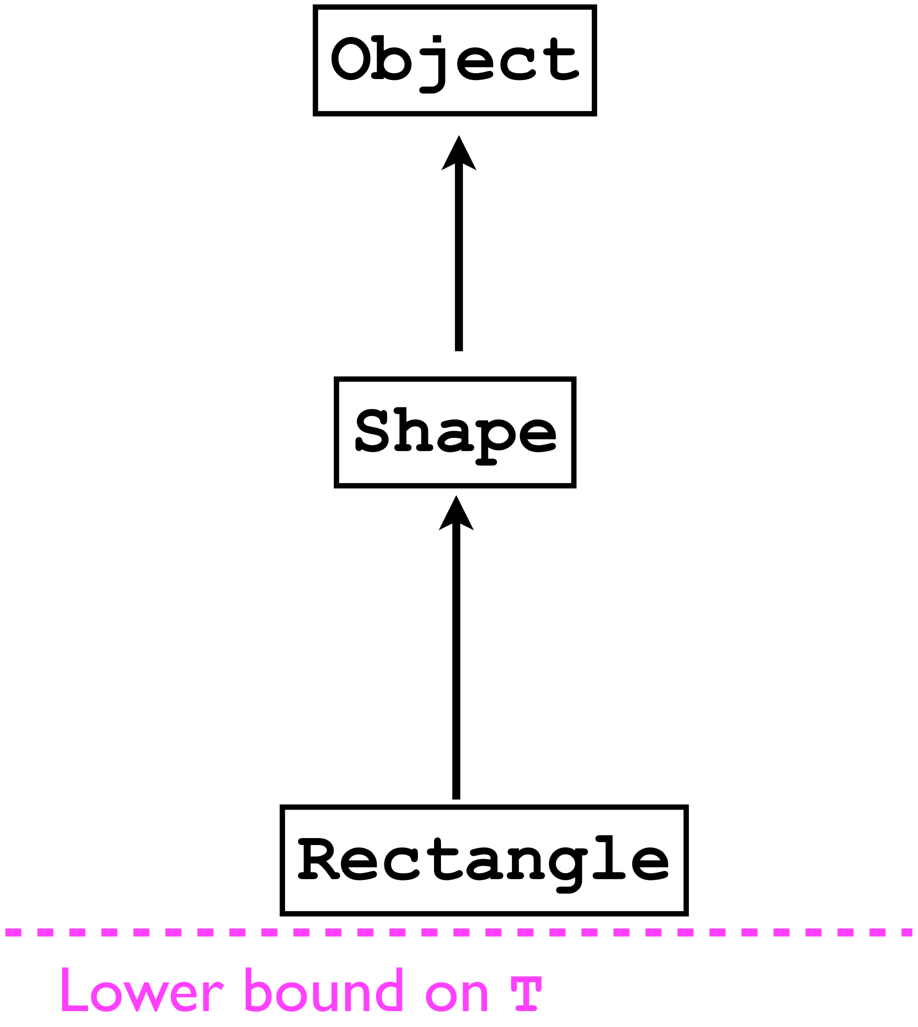
- However, **`Rectangle`** does not offer a method **`compareTo (Rectangle o)`** designed specifically for other **`Rectangle`** objects.
- Hence, the **`Rectangle`** class *could not be used* as the type parameter **`T`** when instantiating a **`HeapImpl12`**:

```
class HeapImpl12<T extends Comparable<T>> ...
```

  - *Reason*: Even though **`Rectangle`** is **`Comparable`** to other **`Shape`** objects, it is not **`Comparable<Rectangle>`**.
    - I.e., **`Rectangle`** offers no **`int compareTo (Rectangle o)`** method.

Lower bounds on types

- What we need is a way of expressing that type parameter `T` may be `Comparable` with class `T`, *or any super-class of* `T`.
  - E.g., we want to allow `HeapImpl12` to store `Rectangle` objects:
    - `Rectangle`s are all `Comparable` with `Shape`, where `Shape` is a *super-class* of `Rectangle`.
- To solve this problem, Java offers **lower bounds** on type parameters.



Lower bounds on types

- For example, we can allow the **`HeapImpl12`** class to accept any type **`T`** so long as **`T`** is **`Comparable`** to class **`T`**, or any super-class of **`T`**.

```
class HeapImpl12<T extends Comparable<? super T>> ... {
   ...
}
```

- The wildcard type **`?`** indicates:
  - "We don't care which type **`T`** is **`Comparable`** to, so long as it's **`Comparable`** to some super-class of **`T`** (or **`T`** itself)."
    - The keyword **`super`** indicates the **lower bound** of the type parameter.

Lower bounds on types

- Given this revised definition of `HeapImpl12`, we can now instantiate a heap of `Rectangle` objects:

```
HeapImpl12<Rectangle> heap =
  new HeapImpl12<Rectangle>();  // OK
```

Binary search trees

Still something to be desired

- Heaps offer fast access to the largest element in a collection.
  - This is most useful in a priority queue.
- However, finding an *arbitrary* element is still slow -- $O(n)$ time.
- We may want to sacrifice efficiency of access to the *largest* access in exchange for increased efficiency to access any *arbitrary* element.

Binary search trees

- A **binary search tree** (BST) is a binary-tree based data structure that offers *O*(log *n*) *average*-case time costs for:

  ```
  add(o)
  find(o)
  remove(o)
  findLargest/removeLargest(o)
  ```

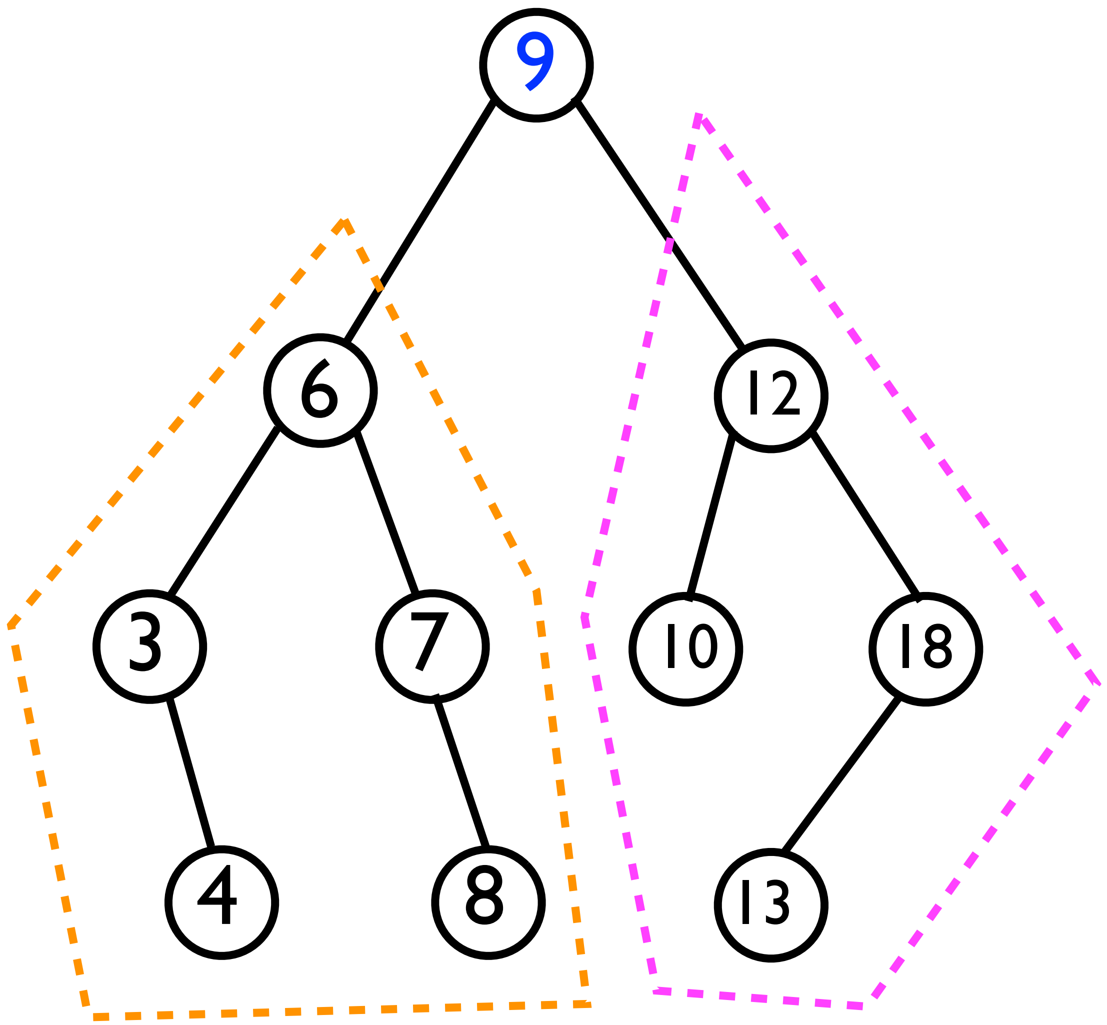
- As with heaps, BSTs exploit the order relations among elements.
  - Heaps required the *root* node r of each sub-tree to be no smaller than any *descendant* node of *r*.
  - BSTs impose constraints on the magnitude of nodes in the *left sub-tree* compared to the magnitude of nodes in the *right sub-tree*.

Binary search trees

- More specifically, a binary search tree (BST) is a binary tree (not necessarily complete) that has the following (recursive) *ordering property*:
    - For each node *n*:
        - All nodes in the *left sub-tree* of *n* are "less than" node *n* itself.
        - All nodes in the *right sub-tree* of *n* are "greater than or equal to" node *n* itself.
        - Both the left and right sub-trees are themselves BSTs.

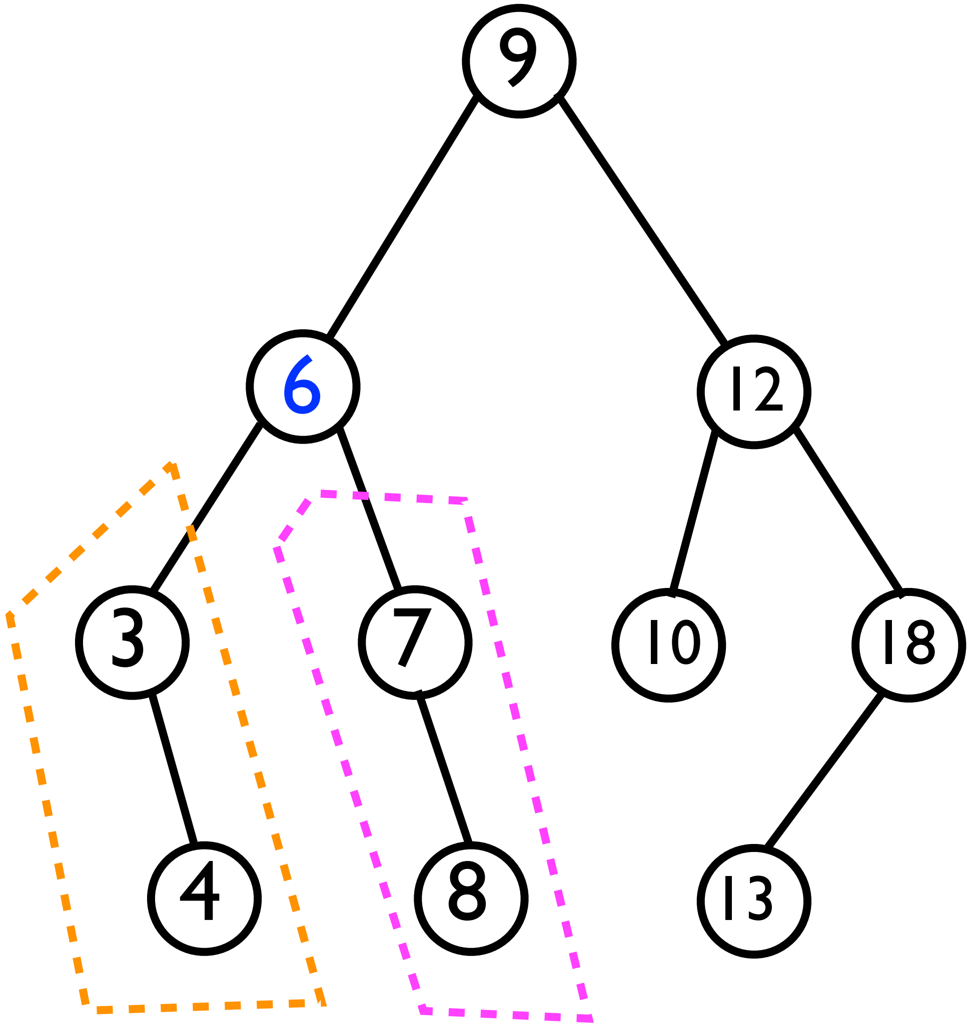
Binary search trees

Left sub-tree < Node (9) ≤ Right sub-tree

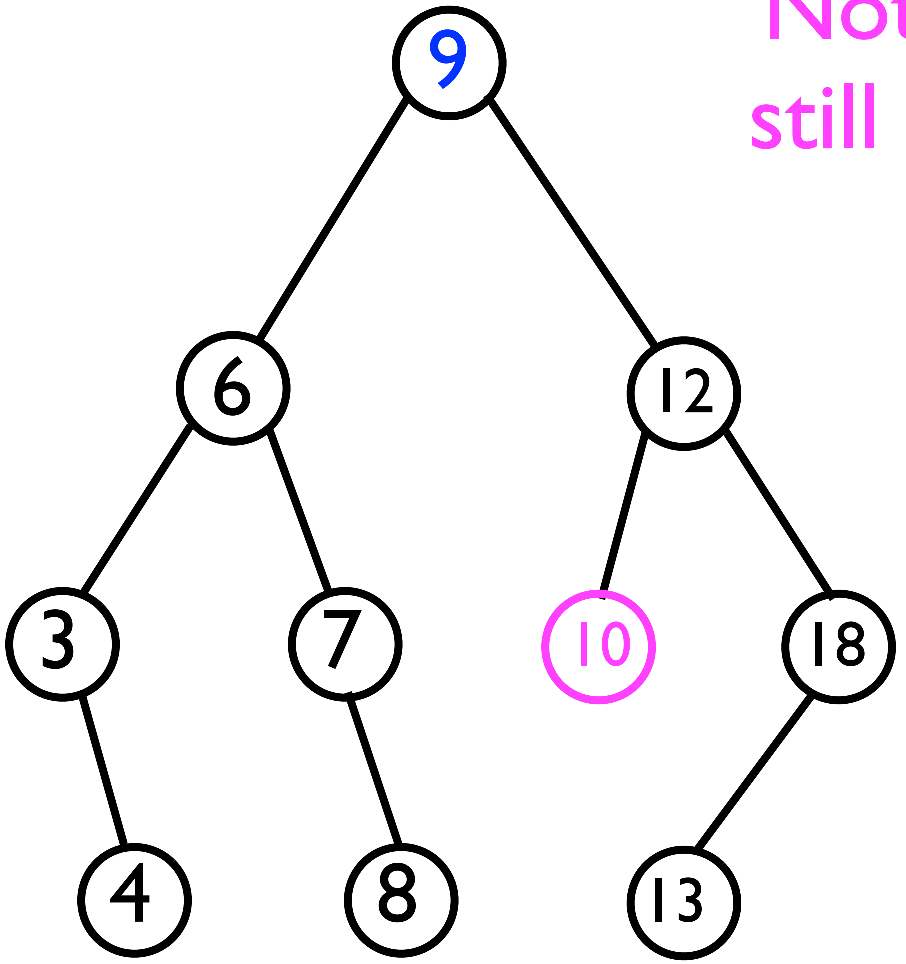


Binary search trees

Left sub-tree < Node (6) ≤ Right sub-tree



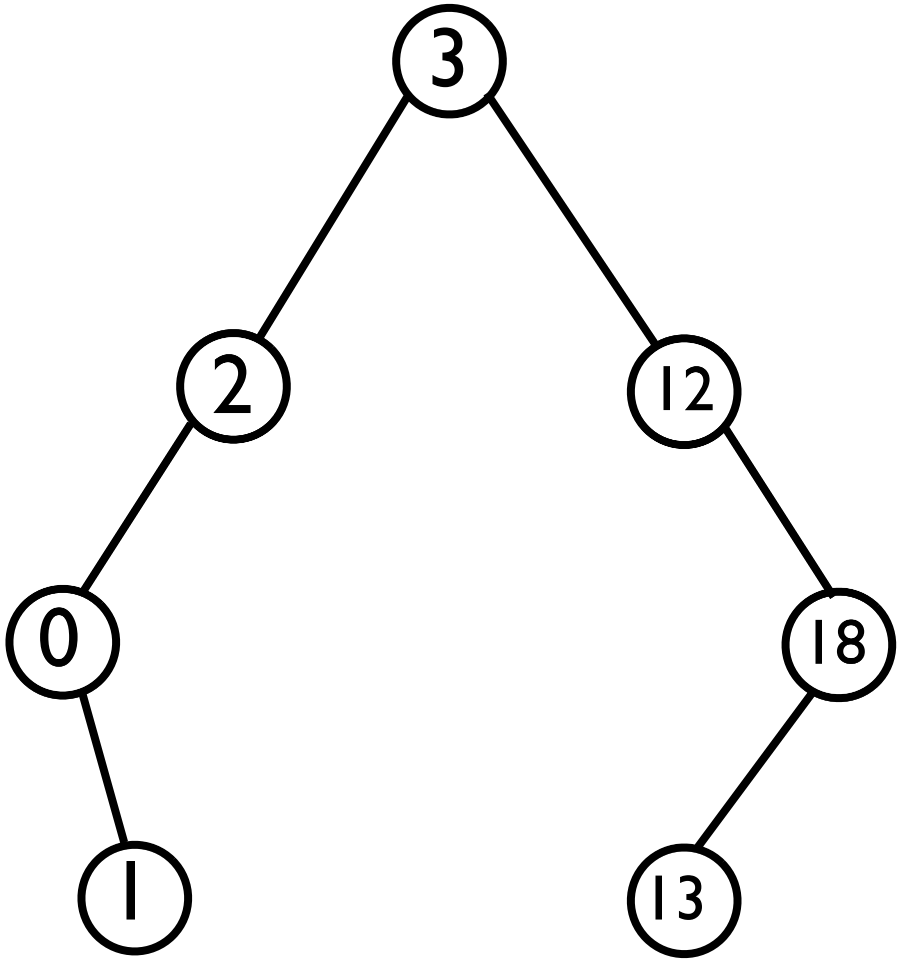
Binary search trees



Note that this node must still be greater or equal to 9!

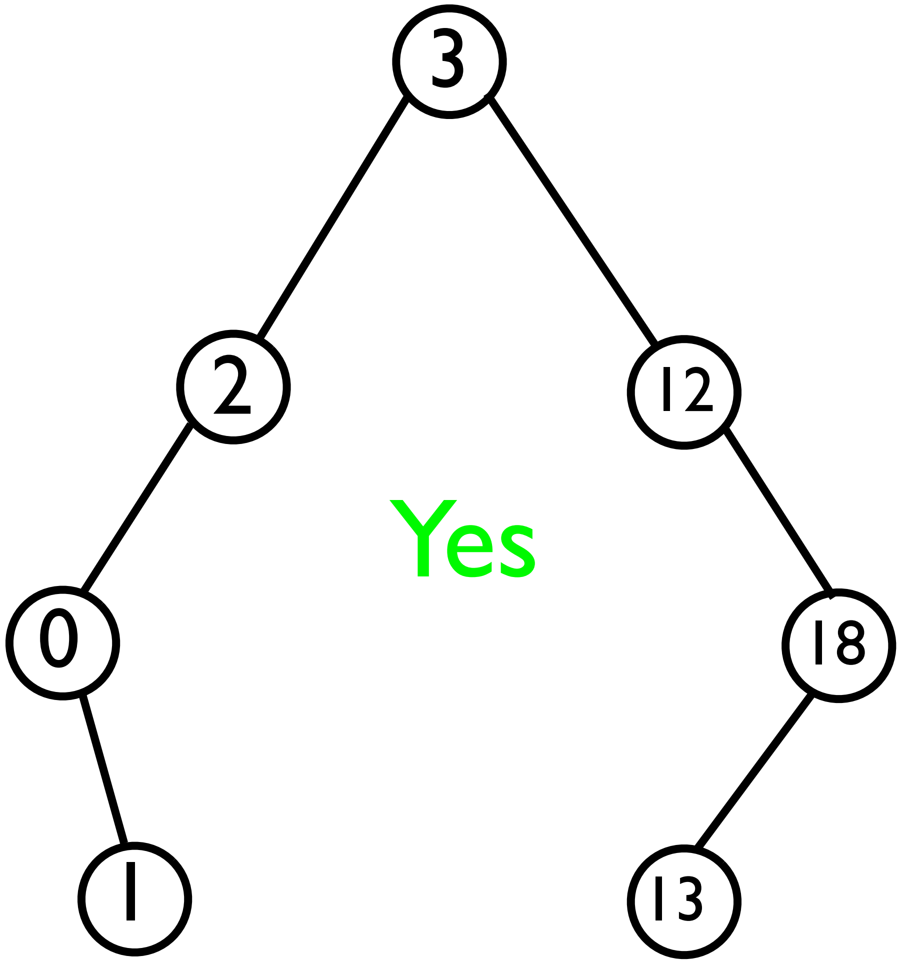
Binary search trees

Which of these trees are valid BSTs?



Binary search trees

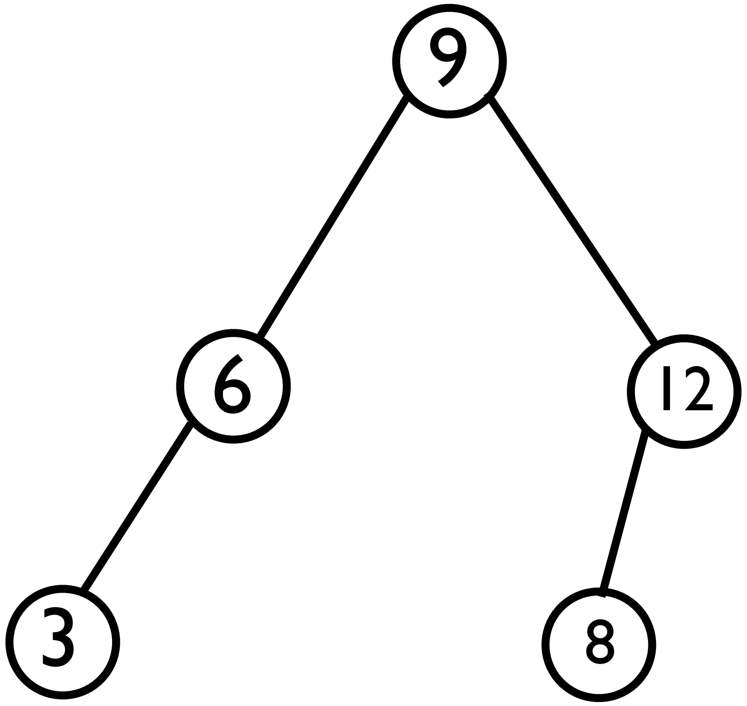
Which of these trees are valid BSTs?



Yes

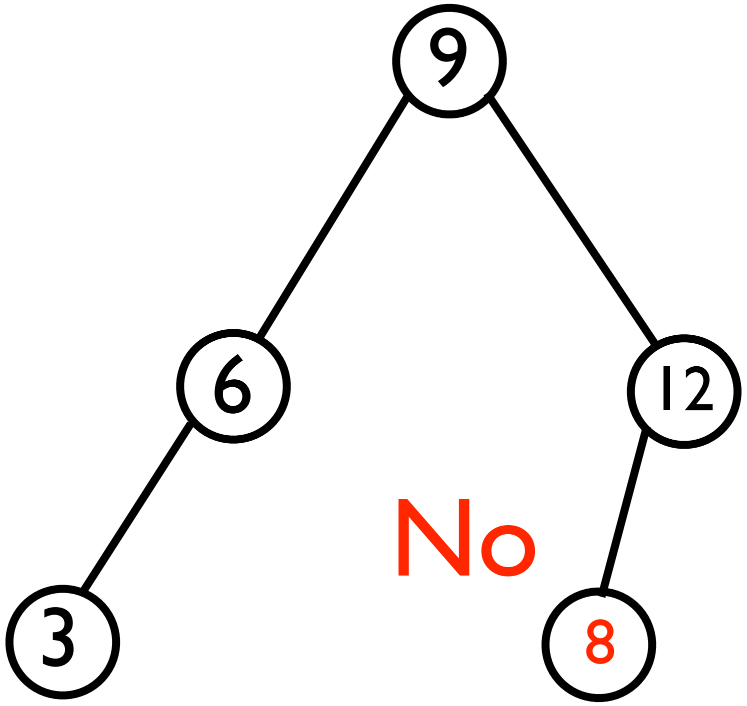
Binary search trees

Which of these trees are valid BSTs?



Binary search trees

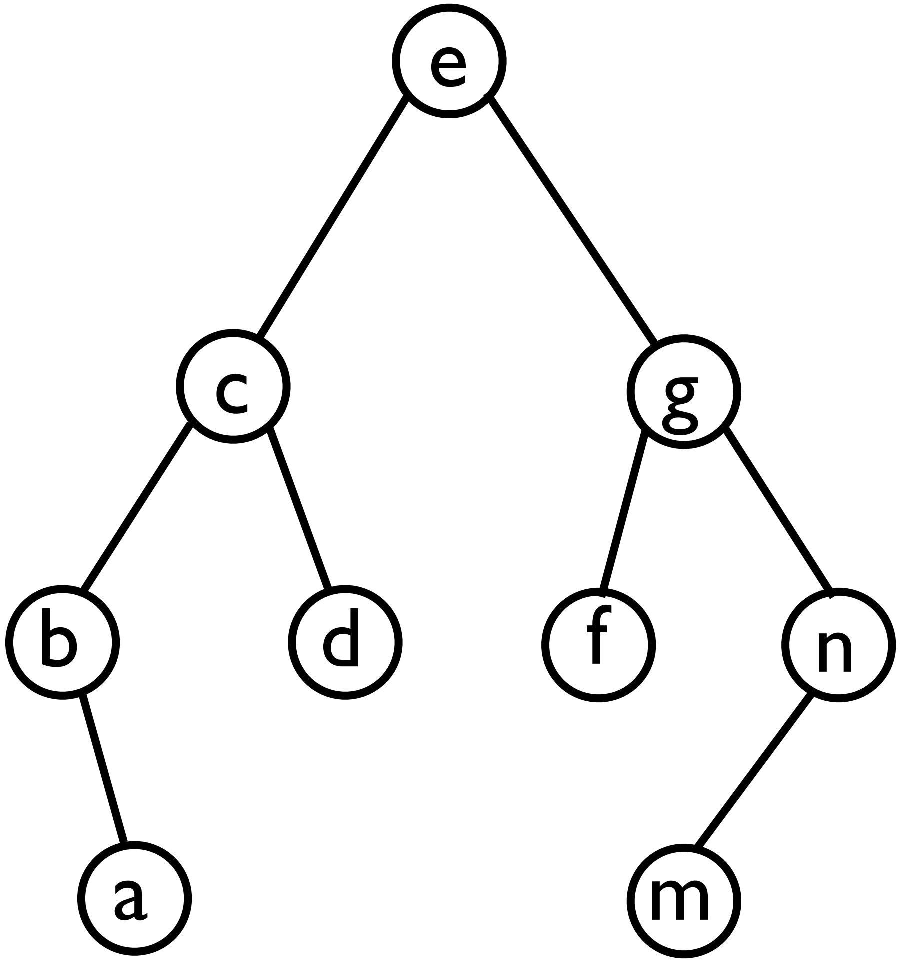
Which of these trees are valid BSTs?



No

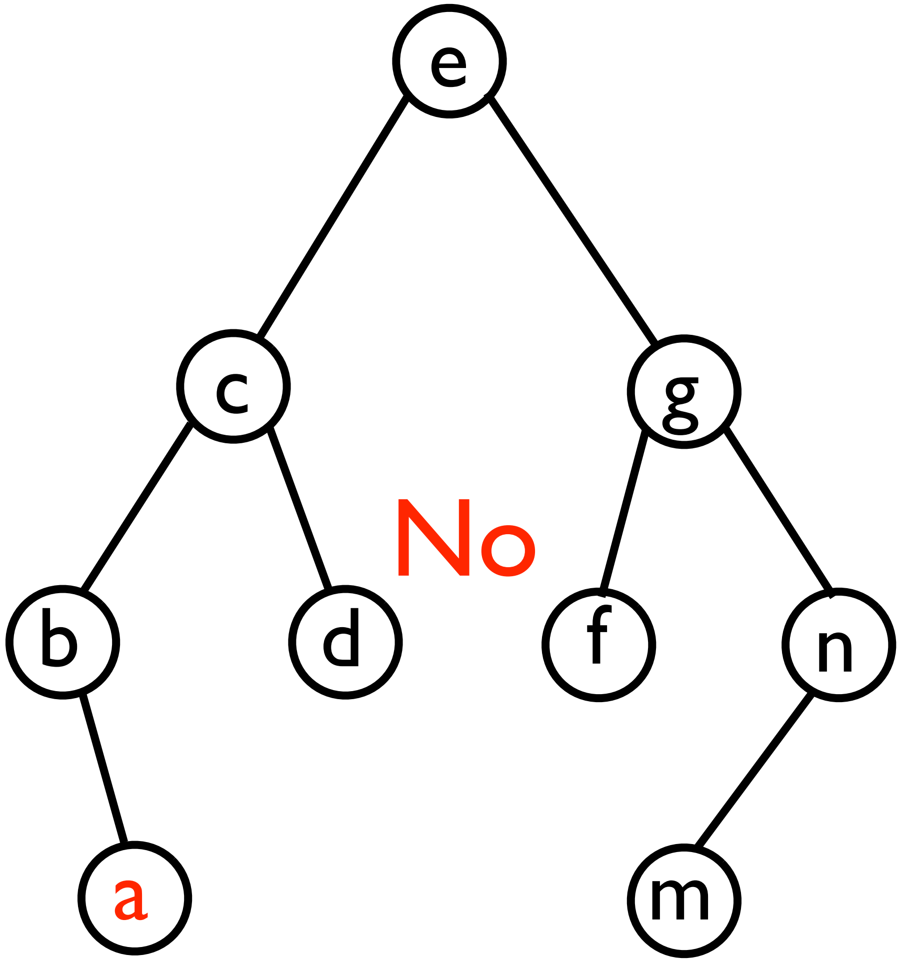
Binary search trees

Which of these trees are valid BSTs?



Binary search trees

Which of these trees are valid BSTs?



No

Binary search trees

```
class BinarySearchTree<T extends Comparable...> {
  static class Node<T> {
    T _data;
    Node<T> _leftChild, _rightChild;
  }
  Node<T> _root = null;  // BST is initially empty

  ...
}
```

Binary search trees

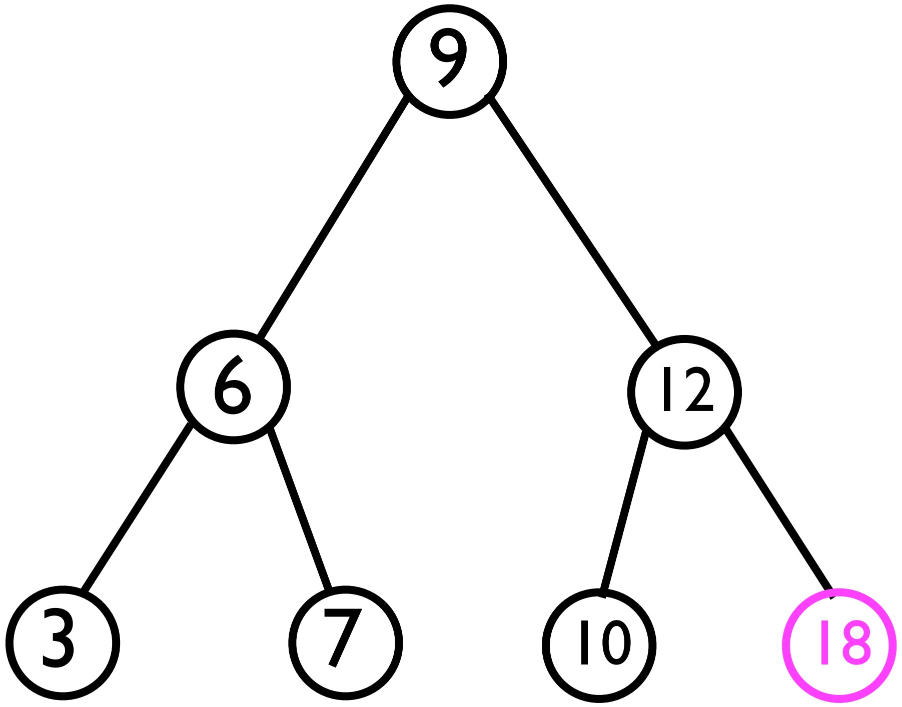
- BSTs do not permit `null` elements:
  - Unclear what "value" they should have compared to other elements.

Binary search trees

- Let us implement the following operations on BSTs:
  - `T find (T o);`
  - `T findSmallest ();`
  - `T findLargest ();`
  - `add (T o);`
  - `remove (T o);`
- To accomplish this, we will also need a few helper methods (not exposed to user):
  - `Node<T> findNode (Node<T> root, T o);`
  - `Node<T> findSuccessor (Node<T> node);`
  - `Node<T> findParent (Node<T> root, T o);`

Finding the largest element

- Due to the ordering property, finding the largest element of a BST is easy -- we just return the *right-most node* in the whole tree.



Finding the largest element

- Due to the ordering property, finding the largest element of a BST is easy -- we just return the *right-most node* in the whole tree.

```
T findLargest (Node<T> root) {

        // Iterative solution?

        // Recursive solution?

}
```

Finding the largest element

- Due to the ordering property, finding the largest element of a BST is easy -- we just return the *right-most node* in the whole tree.

Iterative solution

```
T findLargest (Node<T> root) {
  Node<T> node = root;
  while (node._rightChild != null) {
    node = node._rightChild;
  }
  return node._data;
}
```

Finding the largest element

- Due to the ordering property, finding the largest element of a BST is easy -- we just return the *right-most node* in the whole tree.

Recursive solution

```
T findLargest (Node<T> root) {
  if (root._rightChild == null) {
    return root._data;
  } else {
    return findLargest(root._rightChild);
  }
}
```

Base case

Recursive part

Finding the smallest element

- Due to the ordering property, finding the smallest element of a BST is easy -- we just return the *left-most node* in the whole tree.

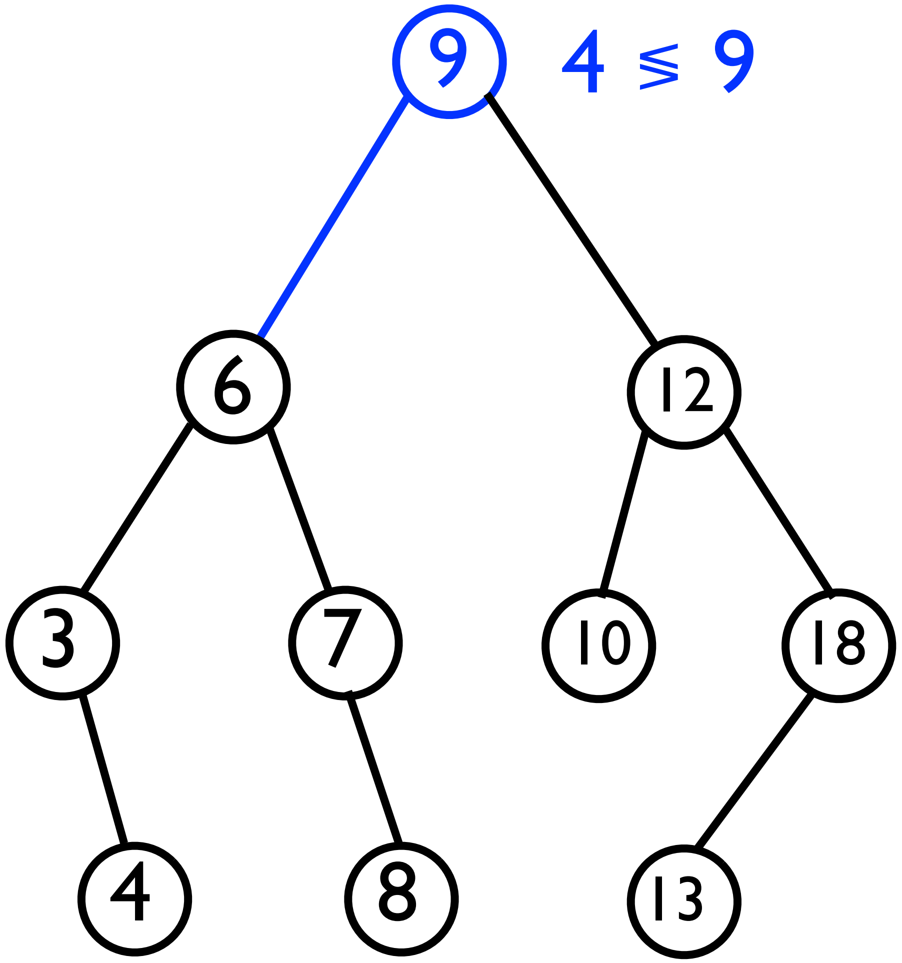
```
T findSmallest (Node<T> root) {
  Node<T> node = root;
  while (node._leftChild != null) {
    node = node._leftChild;
  }
  return node._data;
}
```

Finding a node

- The ordering property of binary search trees also enables efficient search for any *particular* node.
- Due to the ordering property, there is only one place in a given BST where value o would be stored.
    - If it's not there, then o is not contained in the BST -- hence, we return null.

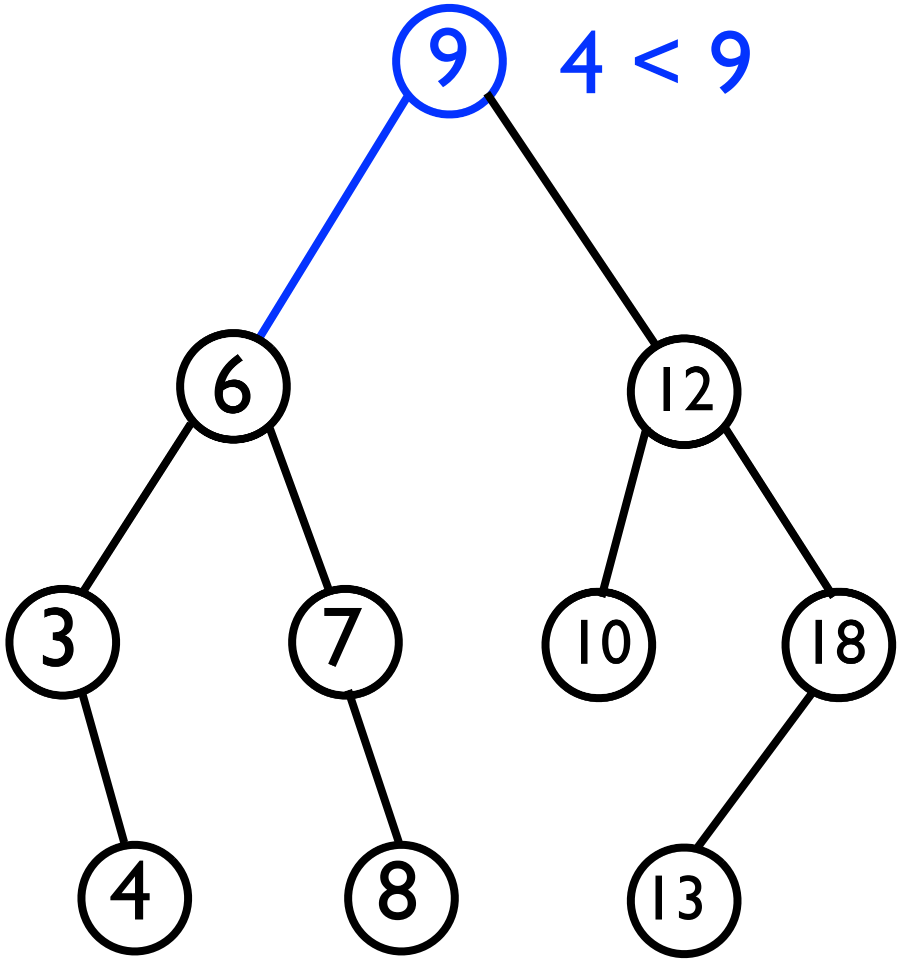
Finding a node

- Given the BST below, suppose we wish to find node 4.
  - We always start at the root and recurse.



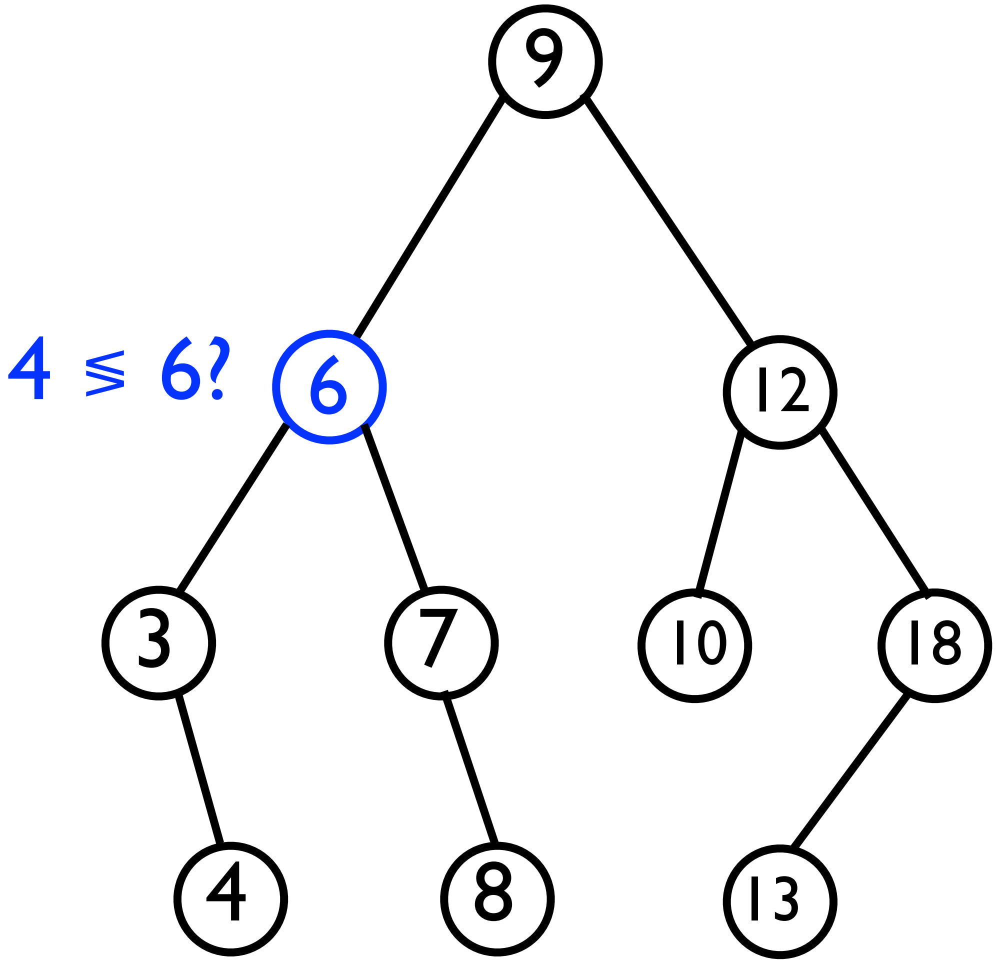
Finding a node

- Given the BST below, suppose we wish to find node 4.
  - We always start at the root and recurse.



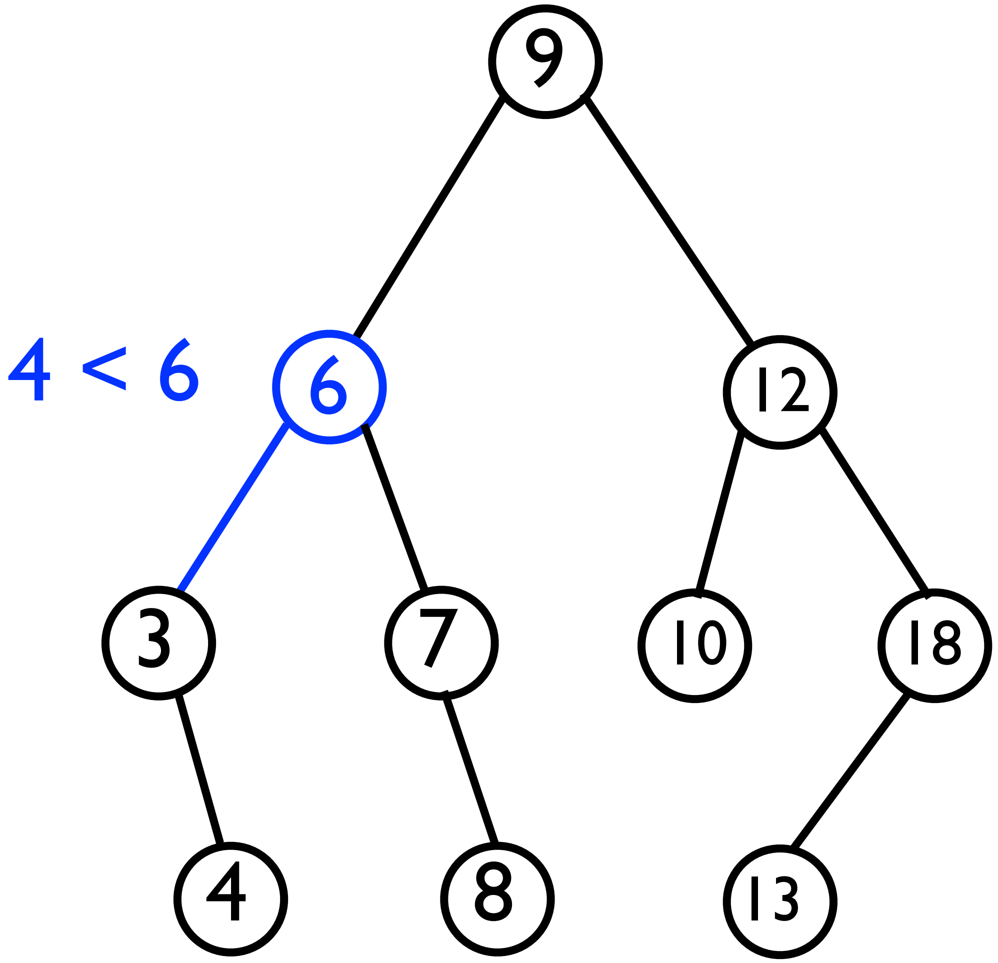
Finding a node

- Given the BST below, suppose we wish to find node 4.
  - We always start at the root and recurse.



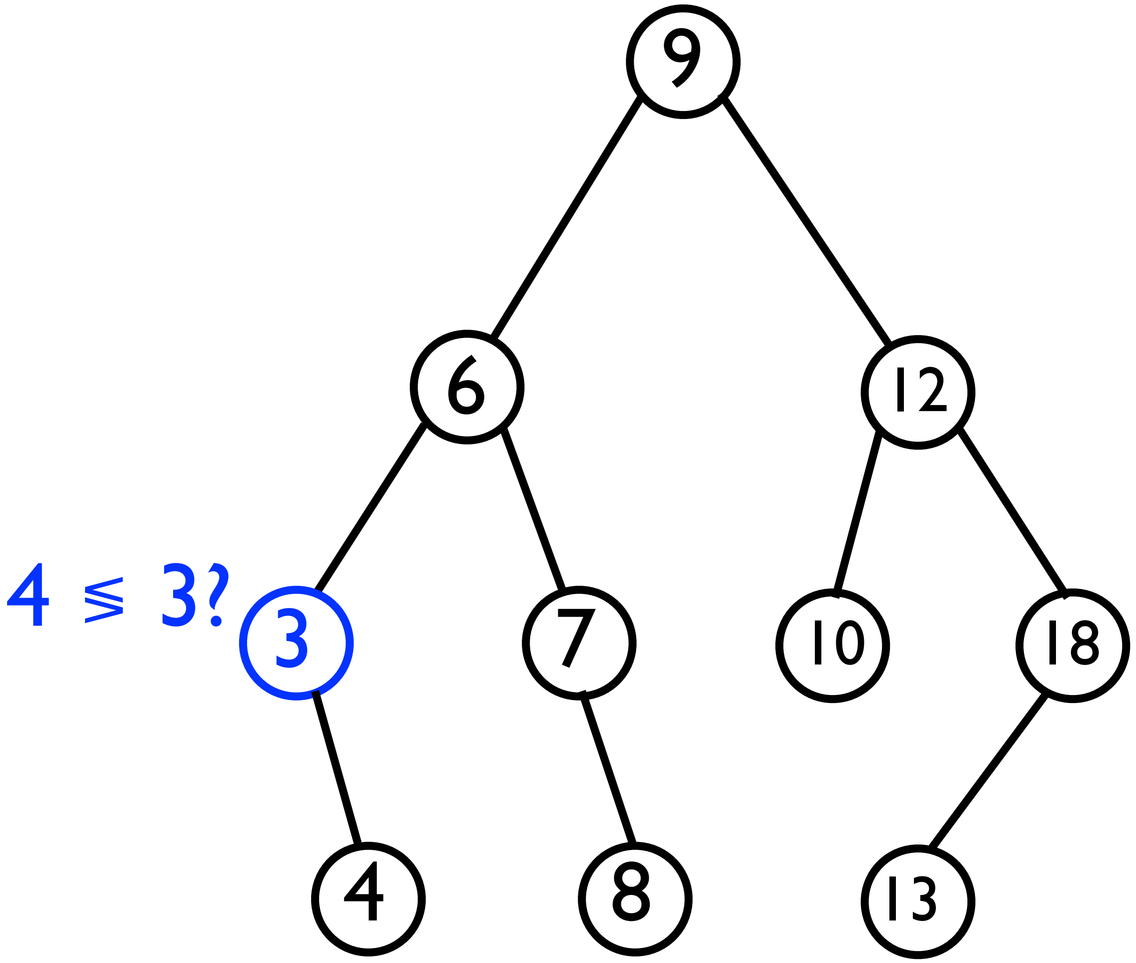
Finding a node

- Given the BST below, suppose we wish to find node 4.
  - We always start at the root and recurse.



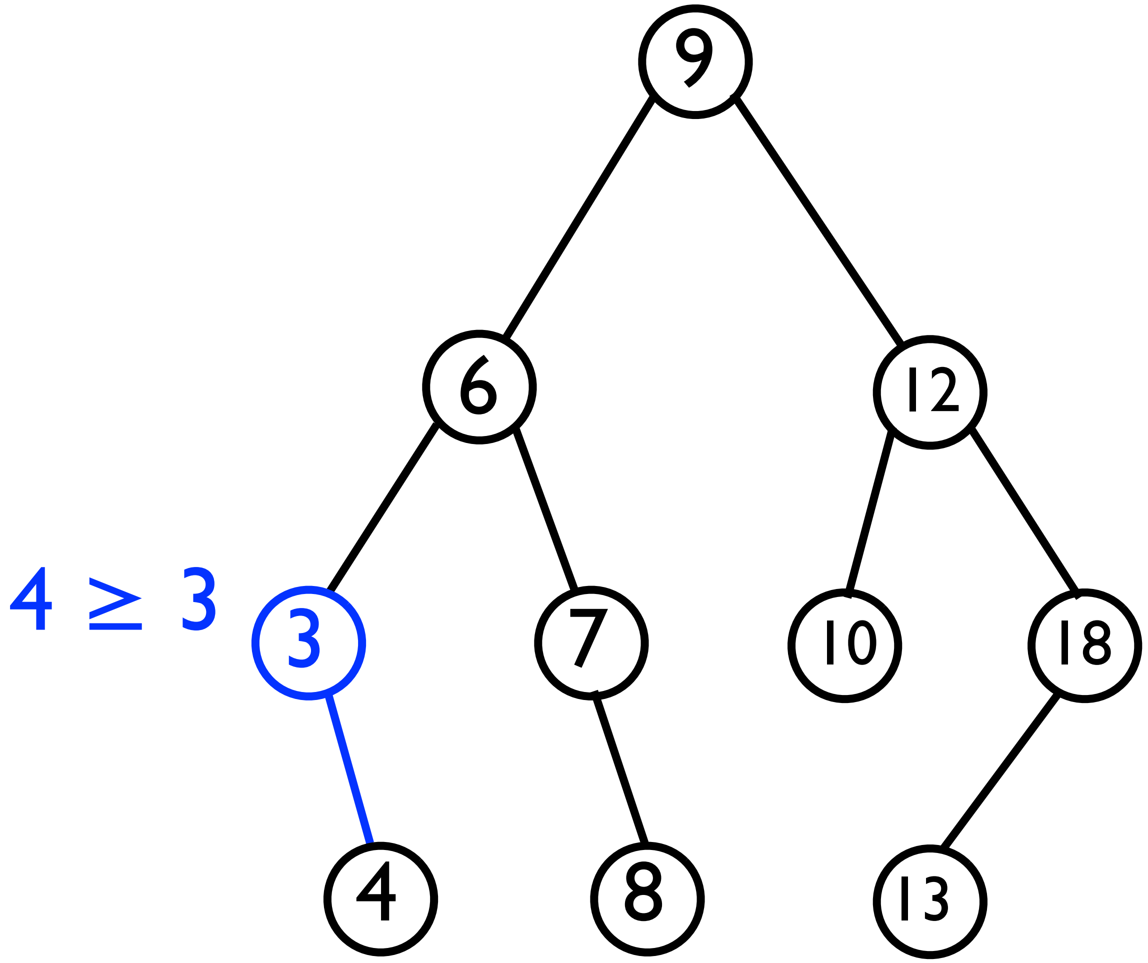
Finding a node

- Given the BST below, suppose we wish to find node 4.
  - We always start at the root and recurse.



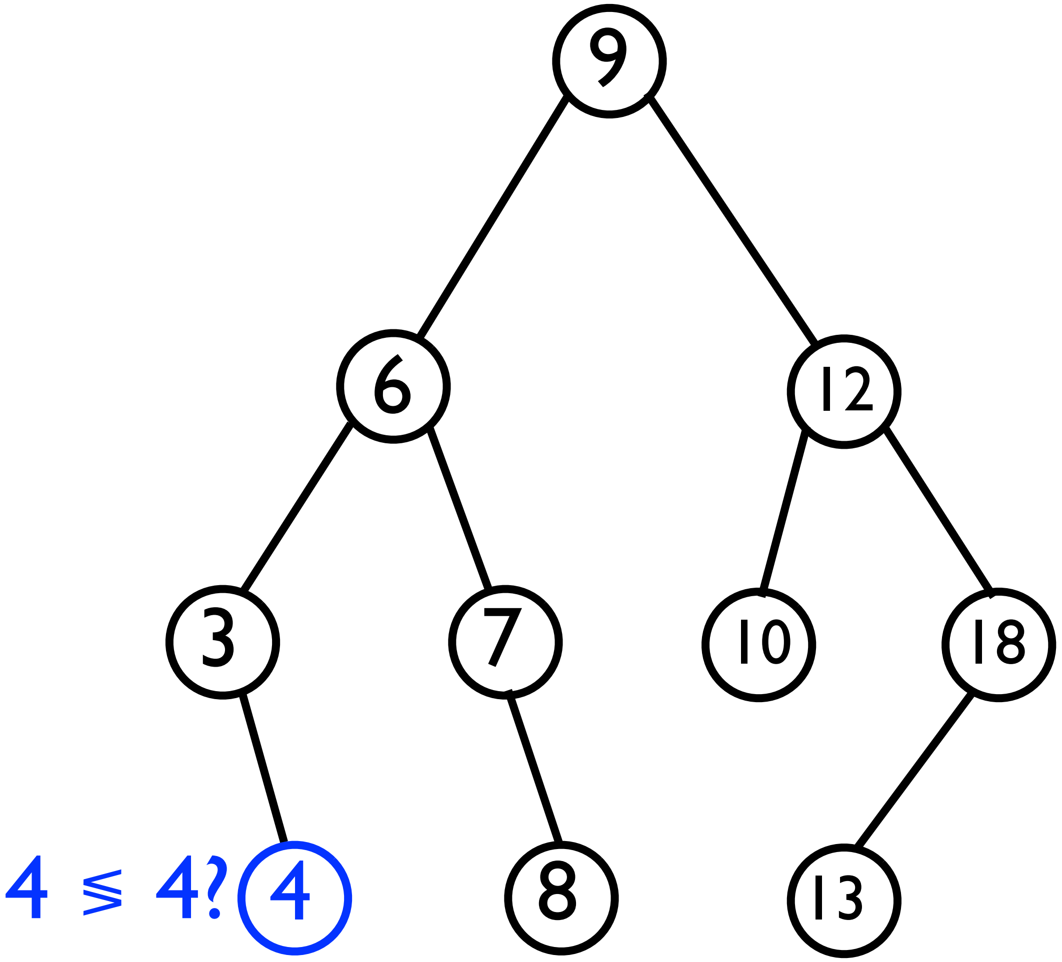
Finding a node

- Given the BST below, suppose we wish to find node 4.
  - We always start at the root and recurse.



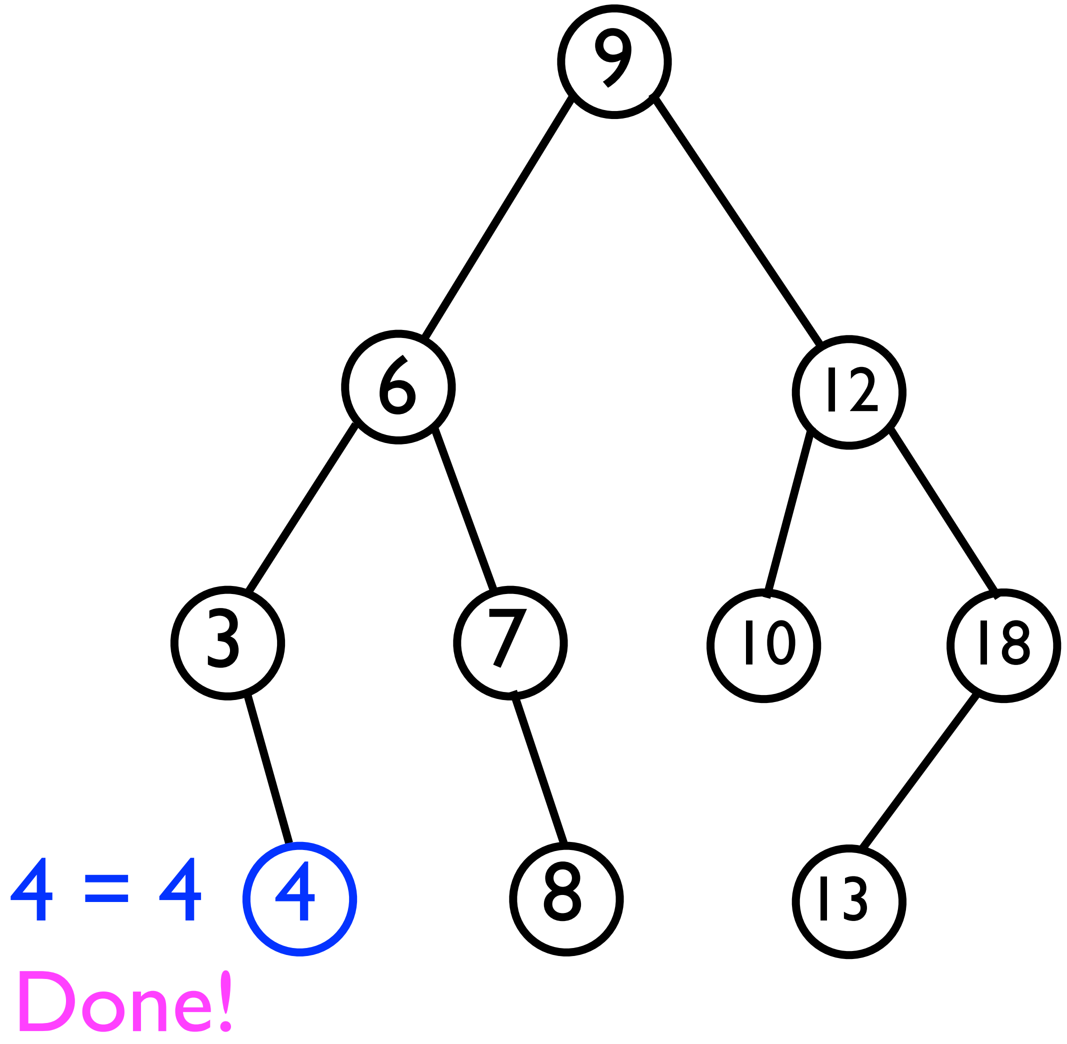
Finding a node

- Given the BST below, suppose we wish to find node 4.
  - We always start at the root and recurse.



Finding a node

- Given the BST below, suppose we wish to find node 4.
  - We always start at the root and recurse.



Finding a node

- Code:

```
// Returns the Node containing o, or else
// null if o is not contained in the BST.
Node<T> findNode (Node<T> root, T o) {
  if (root._data.equals(o) {
    return root;
  } else if (root._data.compareTo(o) < 0 && // Right subtree
             root._rightChild != null) {
      return findNode(root._rightChild, o);
  } else if (root._data.compareTo(o) >= 0 && // Left subtree
             root._leftChild != null) {
     return findNode(root._leftChild, o);
  } else {
    return null;
  }
}
```

Due to the ordering property, there is only *one* place in a given BST where value `o` would be stored. If it's not there, then `o` is *not contained in the BST* -- hence, we return `null`.

Finding a node

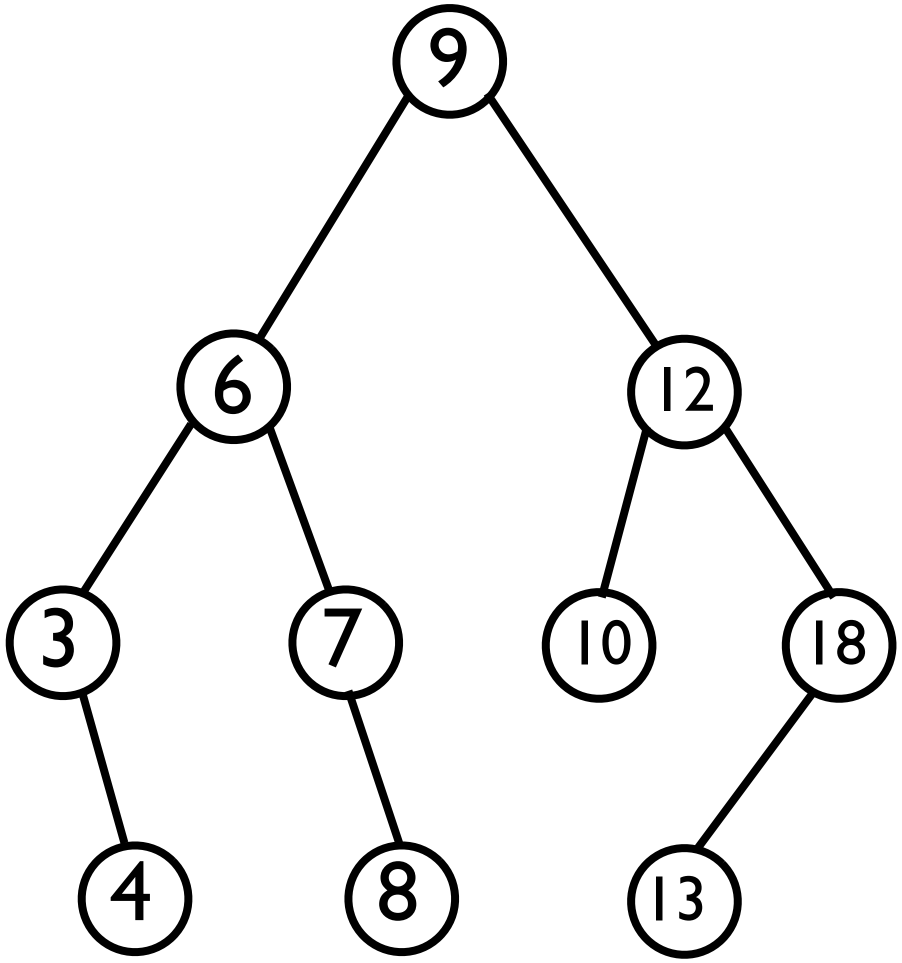
- The **`findNode(root, o)`** method would not be exposed to the user in the BinarySearchTree ADT interface.

- However, we can "wrap" this method with **`T find (T o)`** so that the underlying node infrastructure is hidden:

```
T findNode (T o) {
  if (_root == null) {
    throw NoSuchElementException();
  } else {
    final Node<T> node = findNode(_root, o);
    if (node == null) {
      throw NoSuchElementException();
    } else {
      return node._data;
  }
}
```

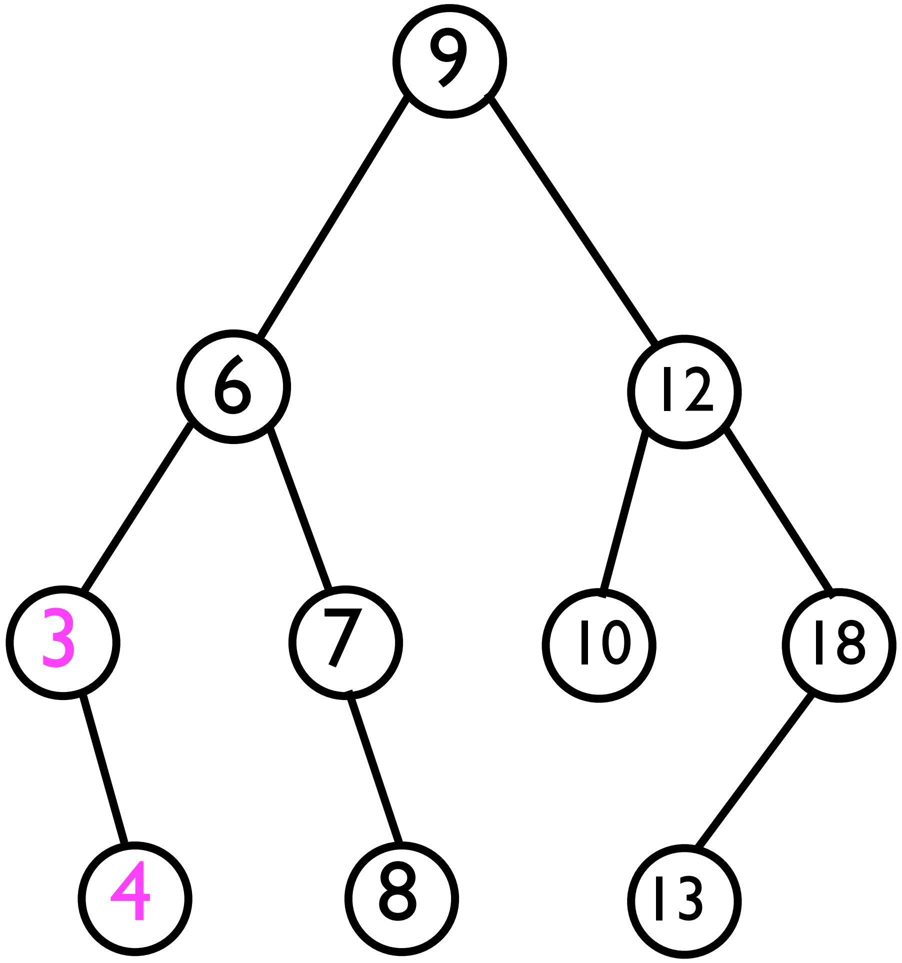
Finding a node's successor

- It will turn out to be useful to be able to find a node's *successor* in the BST.
  - The *successor* of node *n* is the node with the *next higher value*.



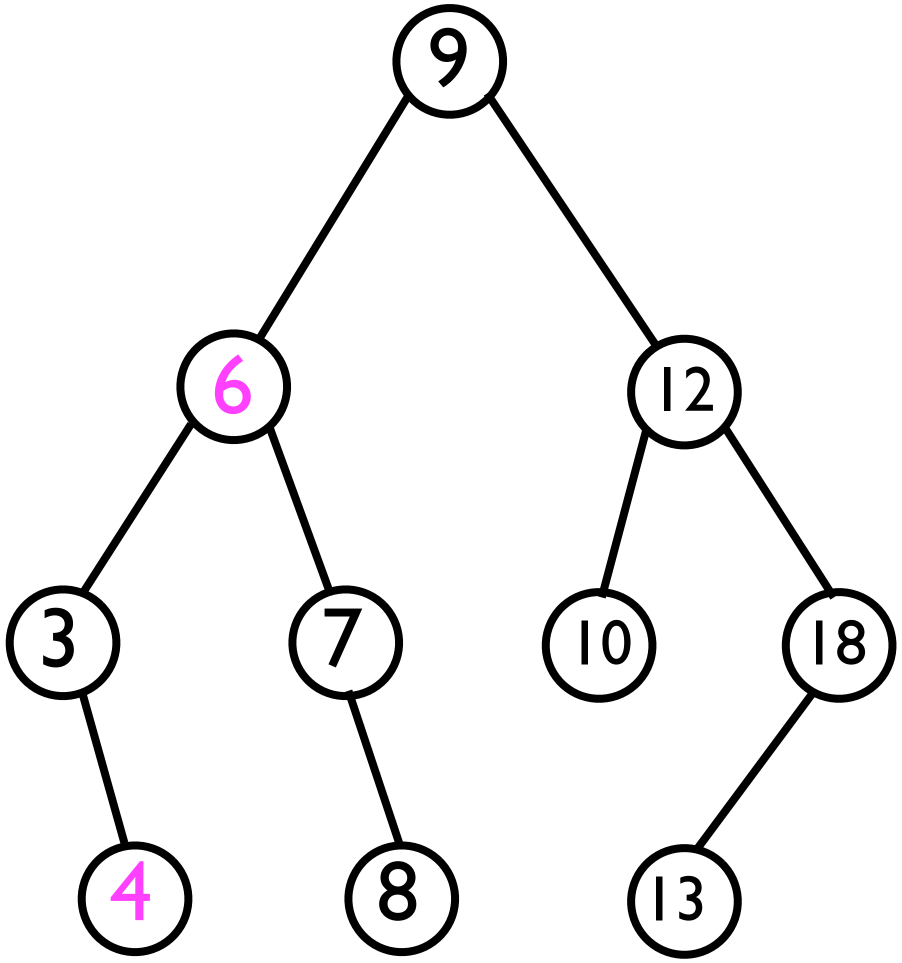
Finding a node’s successor

- It will turn out to be useful to be able to find a node’s *successor* in the BST.
  - The *successor* of node *n* is the node with the *next higher value*.
  - Examples:
    Successor of 3 is 4.
    Successor or 4 is 6.
    Successor of 12 is 13.
    Successor of 8 is 9.



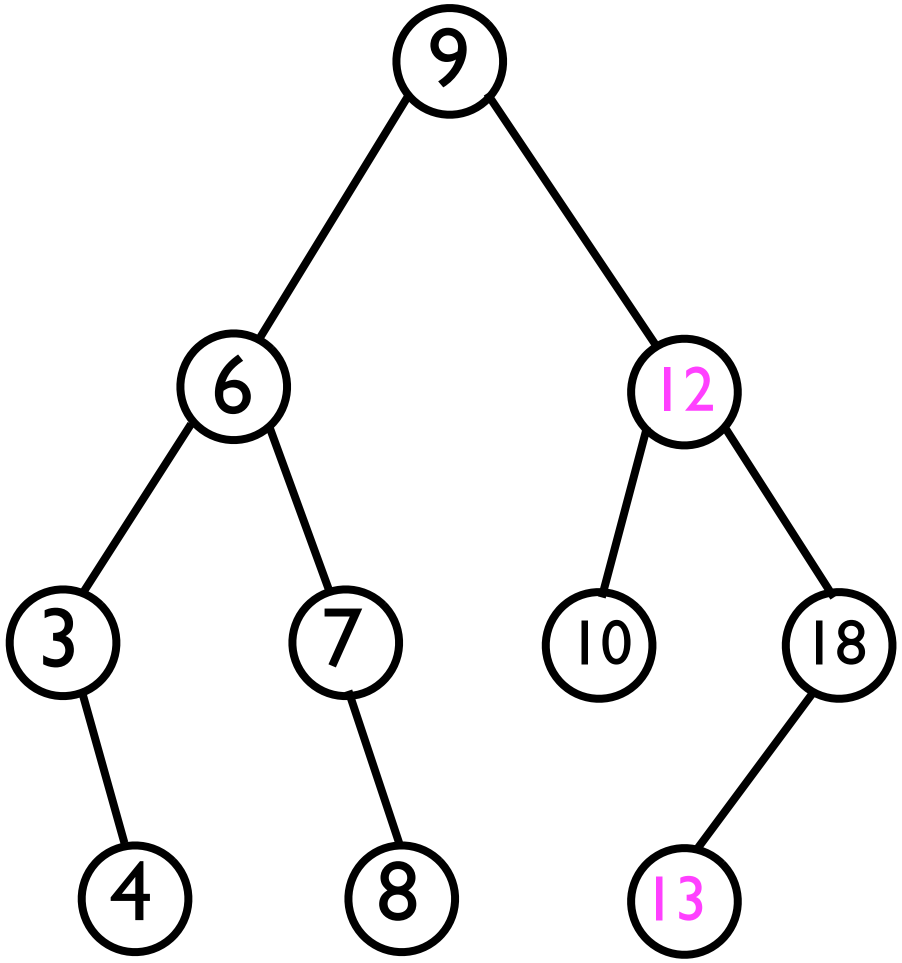
Finding a node's successor

- It will turn out to be useful to be able to find a node's *successor* in the BST.
  - The *successor* of node *n* is the node with the *next higher value.*
  - Examples:
    Successor of 3 is 4.
    Successor or 4 is 6.
    Successor of 12 is 13.
    Successor of 8 is 9.



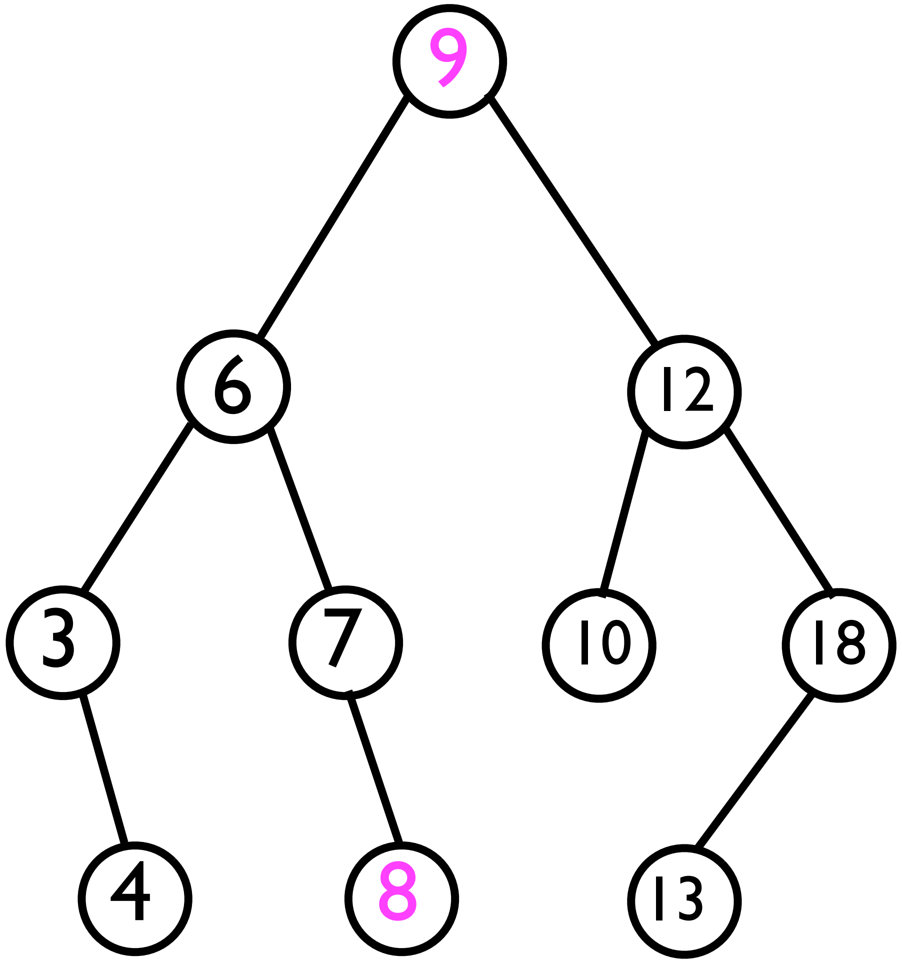
Finding a node's successor

- It will turn out to be useful to be able to find a node's *successor* in the BST.
  - The *successor* of node *n* is the node with the *next higher value.*
  - Examples:
    Successor of 3 is 4.
    Successor or 4 is 6.
    Successor of 12 is 13.
    Successor of 8 is 9.



Finding a node's successor

- It will turn out to be useful to be able to find a node's *successor* in the BST.
  - The *successor* of node *n* is the node with the *next higher value.*
  - Examples:
    Successor of 3 is 4.
    Successor or 4 is 6.
    Successor of 12 is 13.
    Successor of 8 is 9.



Finding a node's successor

- A *successor* node of *n* -- if it exists -- is found by *either*:

  1. Descending into *n*'s right sub-tree, and then recursively selecting left-child until no left child exists.

     - *Intuition*: The right sub-tree has values bigger than n; we want the smallest such value (left-most node).

  2. Finding the *lowest* ancestor of *n* whose left child is also an ancestor of *n*.

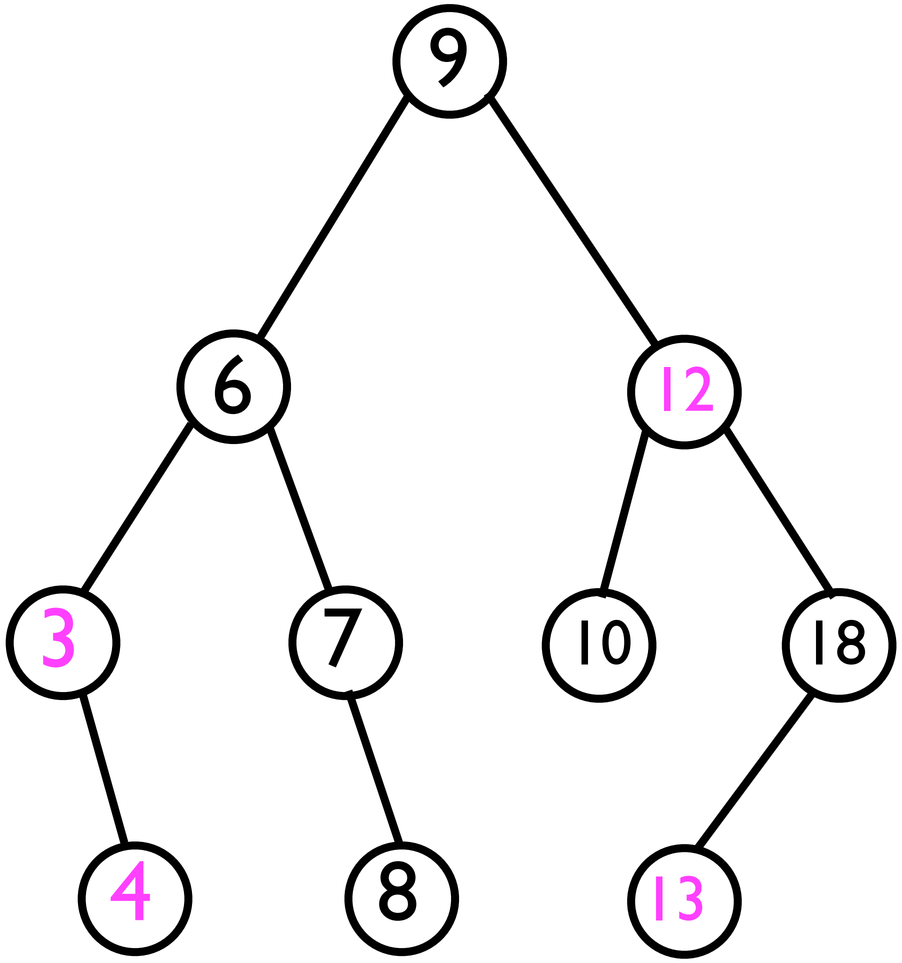
     - *Intuition*: Move "up-and-left" in the BST until we can finally "move right" again, i.e., *towards a higher valued node*.

Finding a node's successor

- A *successor* node of *n* -- if it exists -- is found by *either*:

1. Descending into *n*'s right sub-tree, and then recursively selecting left-child until no left child exists.
2. Finding the *lowest* ancestor of *n* whose left child is also an ancestor of *n*.

- Examples:
  Successor of 3 is 4.
  Successor or 4 is 6.
  Successor of 12 is 13.
  Successor of 8 is 9.

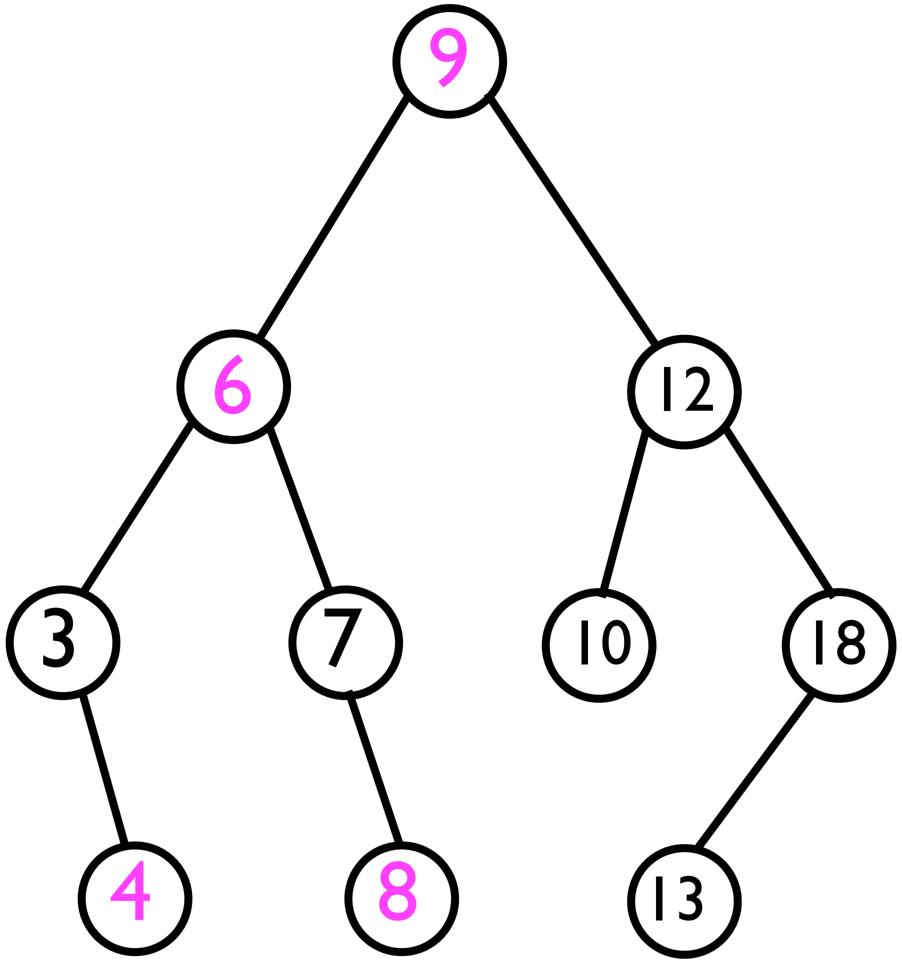


Finding a node's successor

- A *successor* node of *n* -- if it exists -- is found by *either*:

1. Descending into *n*'s right sub-tree, and then recursively selecting left-child until no left child exists.
2. Finding the *lowest* ancestor of *n* whose left child is also an ancestor of *n*.

- Examples:
  Successor of 3 is 4.
  Successor or 4 is 6.
  Successor of 12 is 13.
  Successor of 8 is 9.



Finding a node's successor

- The code for **`Node<T> findSuccessorNode (Node<T> node)`** will be left as an "exercise for the reader".

Adding a new node

- To add a new node, we must distinguish two cases:

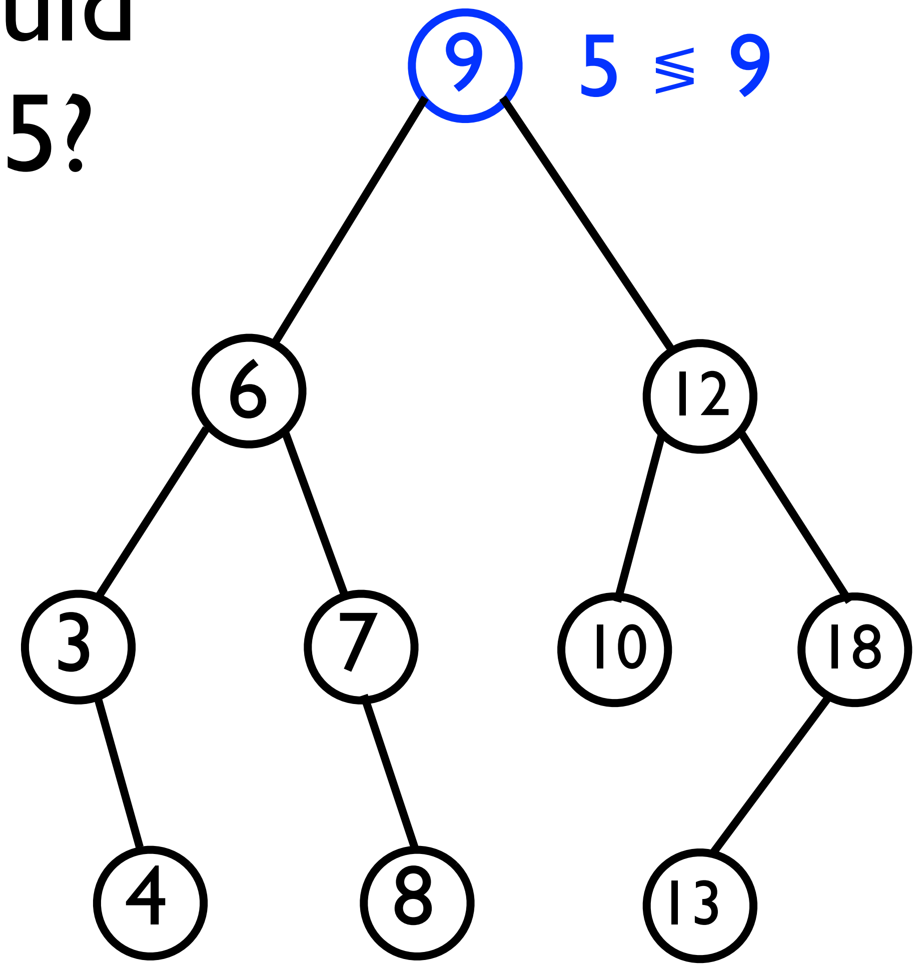
1. The new node is the *first* node in the BST.
    - In this case, we simply set this node to be the root.
2. The new node is *not* the first node in the BST.
    - Then we must find the *parent* node of the node we're about to add.
    - We then add the new node as a child of the parent.

Finding the parent of a new node

- To find the parent node of the new node *n* we want to add:
  - Recursively search from root down towards the leaf nodes, *as if node* n *were already inserted*.
  - Eventually, while recursing at node *p*, the search for the node would take us to a left/right child *that does not yet exist.*
    - At that point, we know *p* is the parent of *n*.
    - *p* is the "natural insertion point" for *n*.

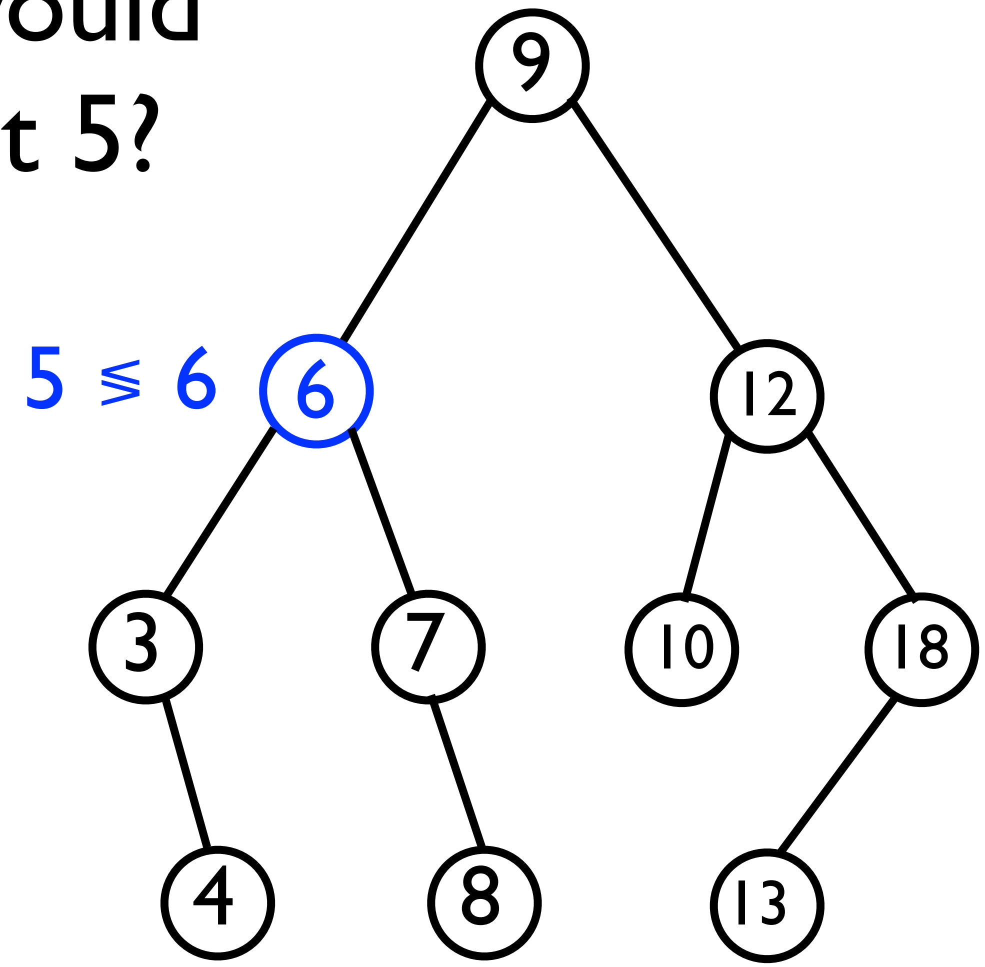
Finding the parent of a new node

Where would we insert 5?



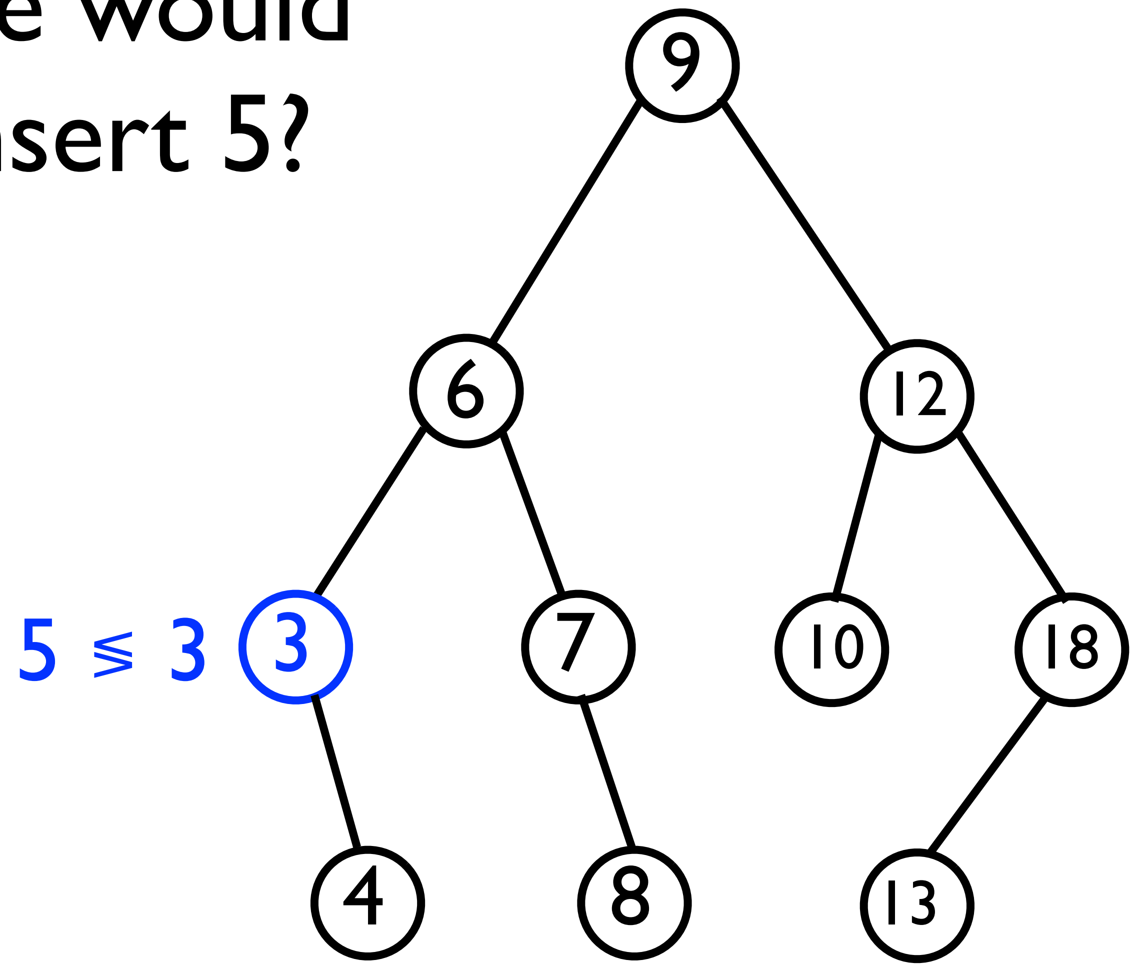
Finding the parent of a new node

Where would we insert 5?



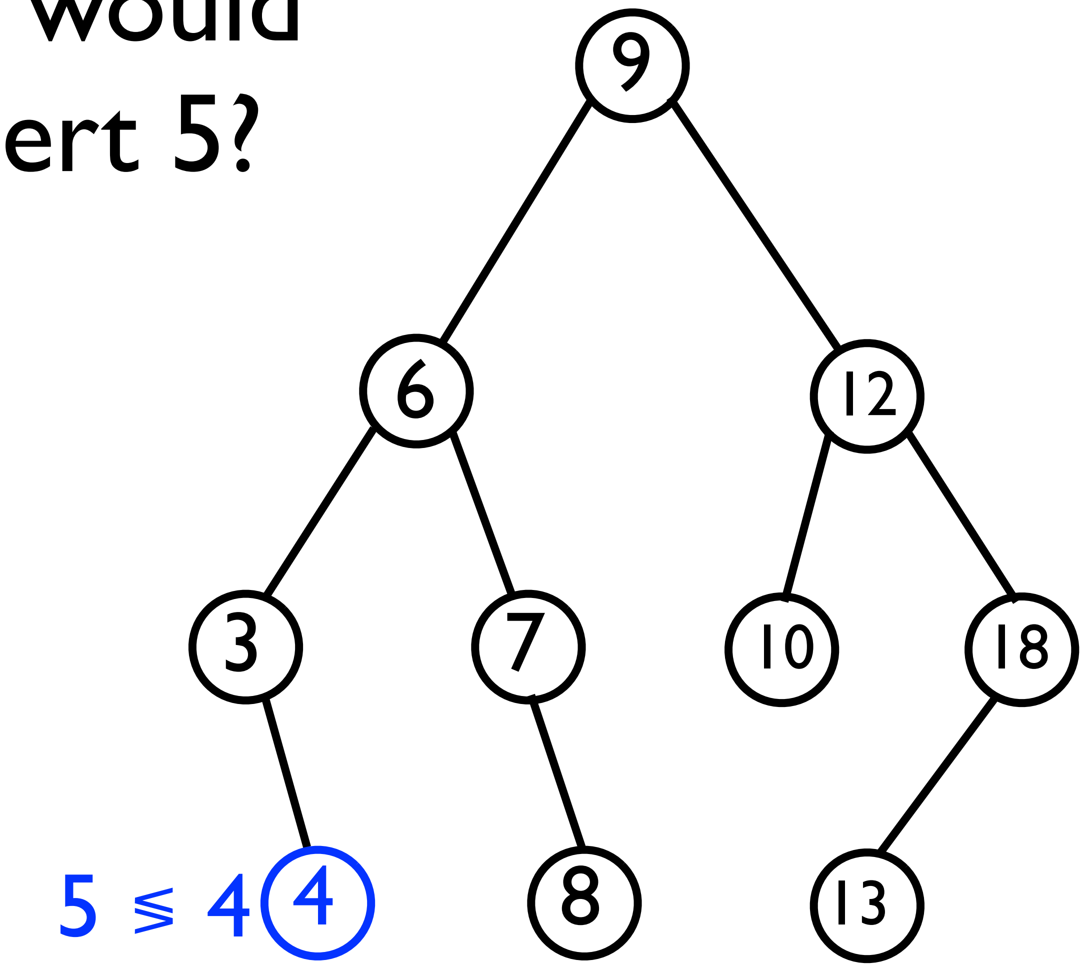
Finding the parent of a new node

Where would we insert 5?



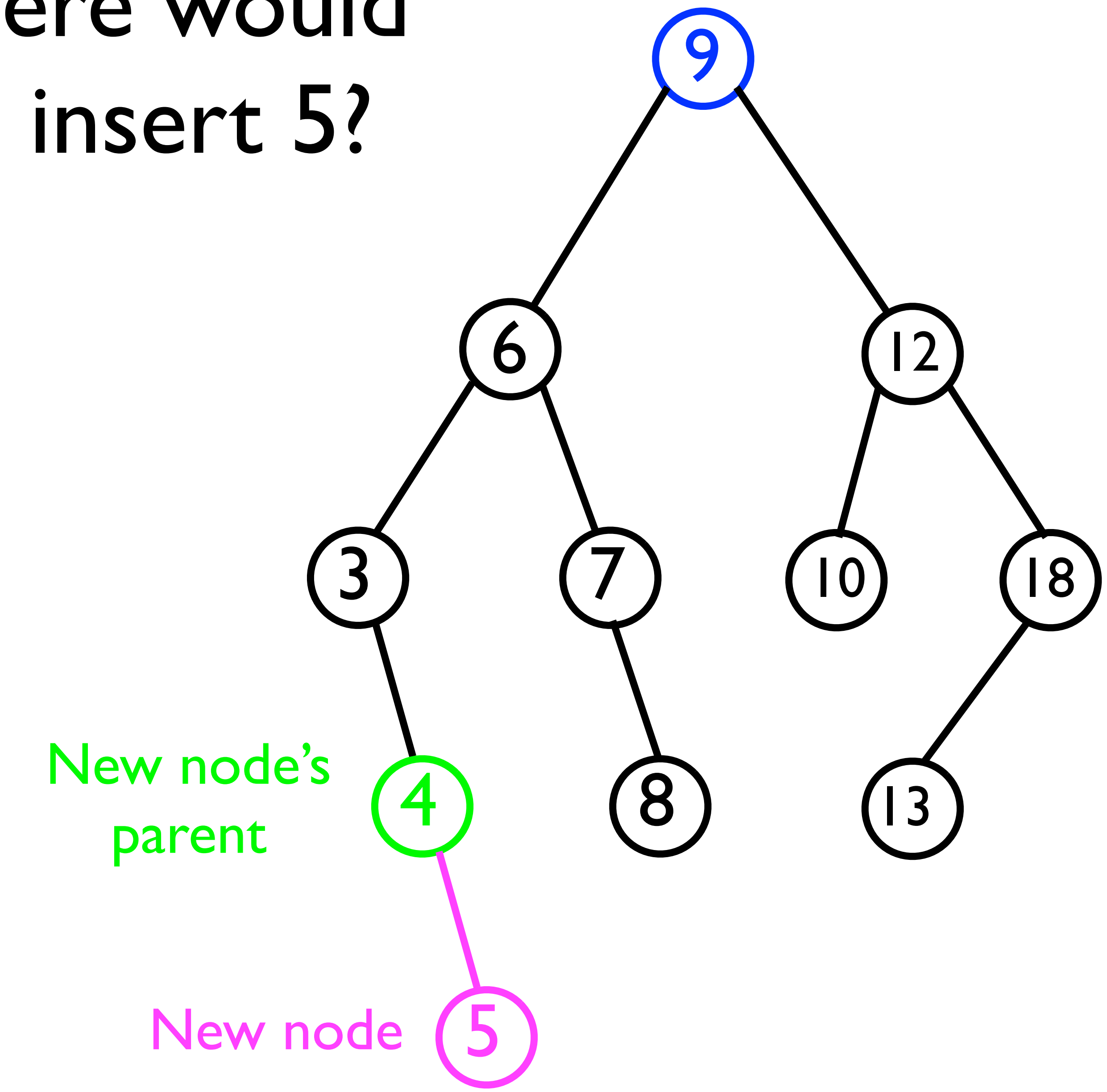
Finding the parent of a new node

Where would we insert 5?



Finding the parent of a new node

Where would we insert 5?



Finding the parent of a new node

```
// Searches from root for the parent node to which the
// specified new node should be added.
Node<T> findParentNode (Node<T> root, T o) {
  // Save comparison result
  final int comparison = root._data.compareTo(o);

  if (comparison < 0 && root._rightChild != null) {
    return findParentNode(root._rightChild, o);
  } else if (comparison >= 0 && root._leftChild != null) {
    return findParentNode(root._leftChild, o);
  } else {  // The appropriate left/child does not yet exist
    return root;  // Hence, we've found the parent
  }
}
```

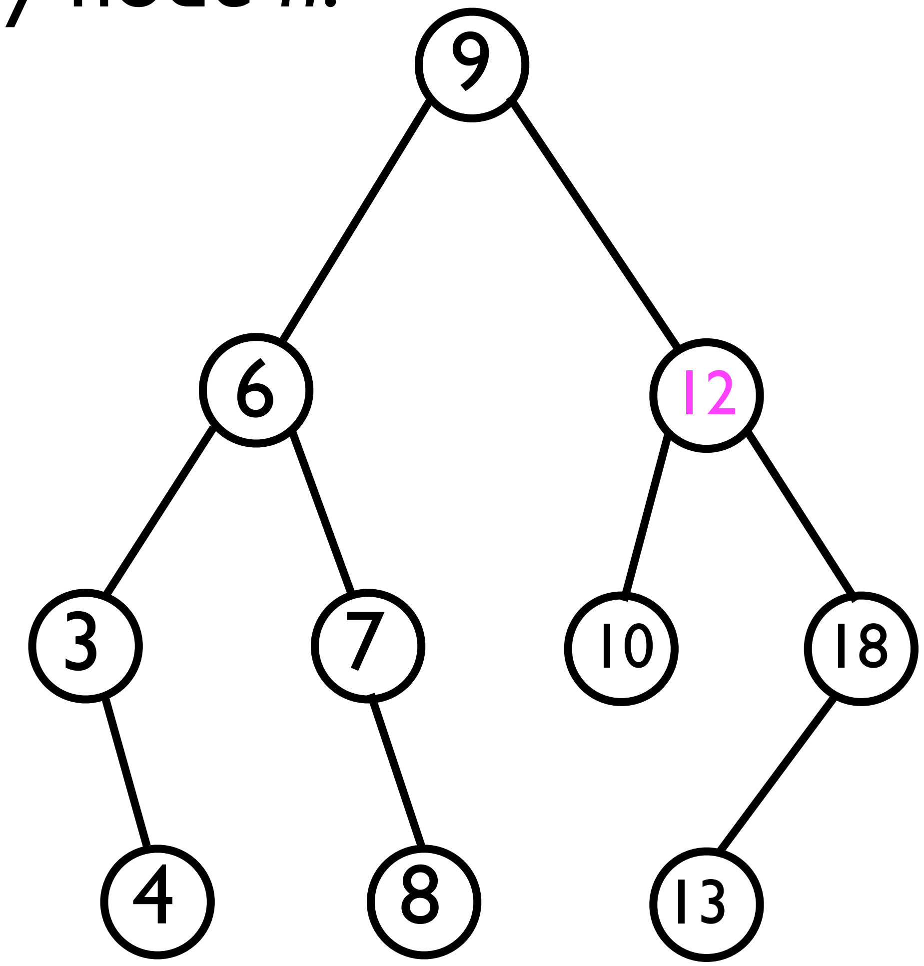
Adding a new node

- We can now implement the `add(o)` method:

```
void add (T o) {
  final Node<T> node = new Node<T>();
  node._data = o;
  if (_root == null) {  // Case 1
    _root = node;
  } else {              // Case 2
    final Node<T> parent = findParent(_root, o);
    if (parent._data.compareTo(o) < 0) {
      parent._rightChild = node;
    } else {
      parent._leftChild = node;
    }
  }
}
```

Removing a node

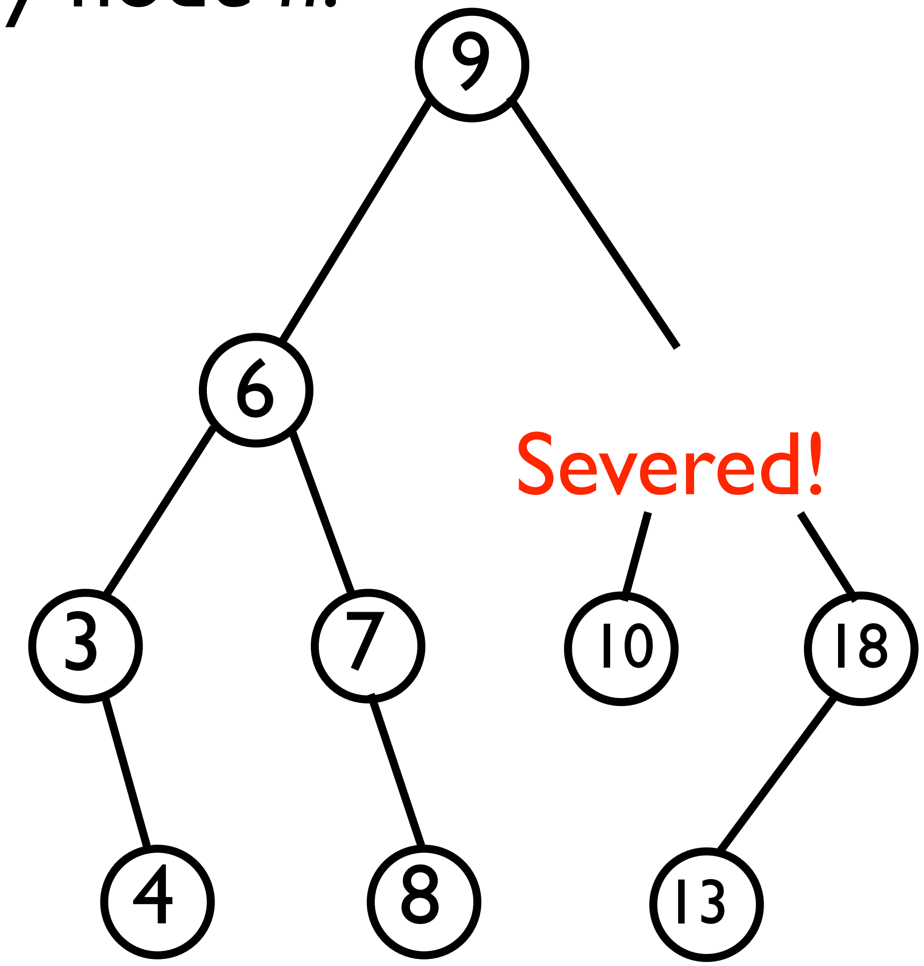
- When removing a node *n* from the BST, we must ensure that:
  - The resulting tree is still *connected*.
  - The resulting tree still has the *ordering property*.
- Consider what might "go wrong" when removing an arbitrary node *n*:



If we remove node 12, then we *sever its left and right sub-trees* from the rest of the BST.

Removing a node

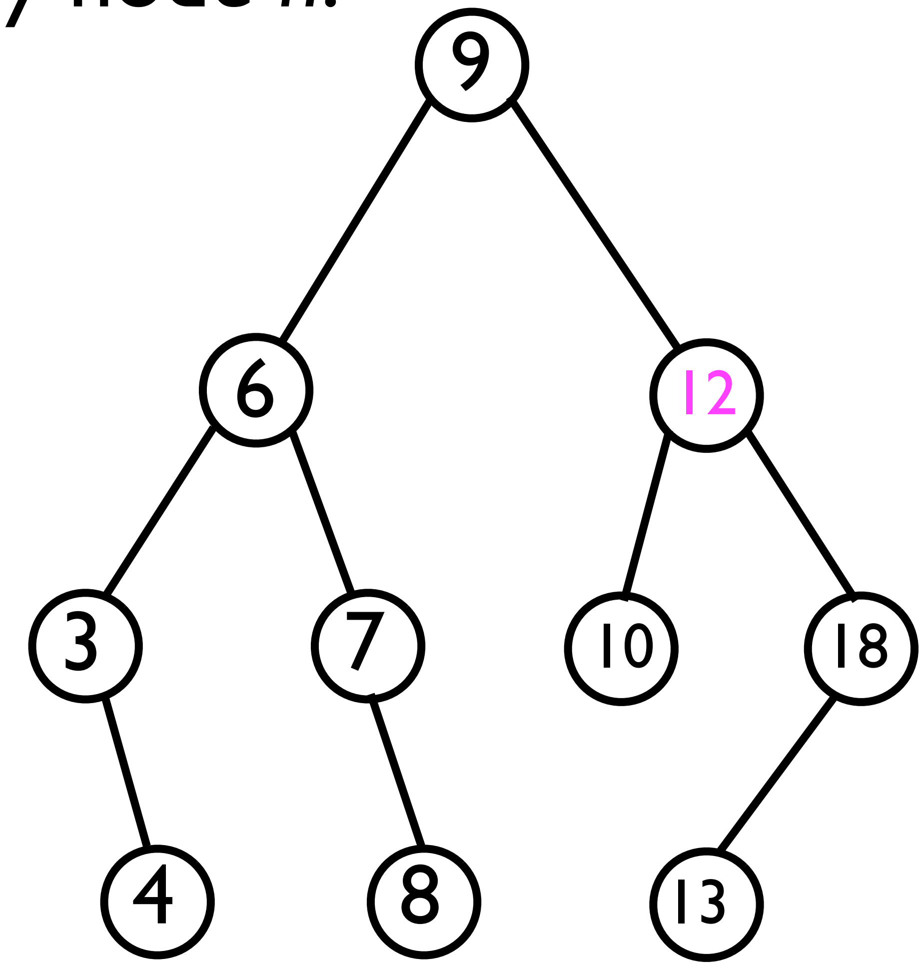
- When removing a node *n* from the BST, we must ensure that:
  - The resulting tree is still *connected*.
  - The resulting tree still has the *ordering property*.
- Consider what might "go wrong" when removing an arbitrary node *n*:



If we remove node 12, then we *sever its left and right sub-trees* from the rest of the BST.

Removing a node

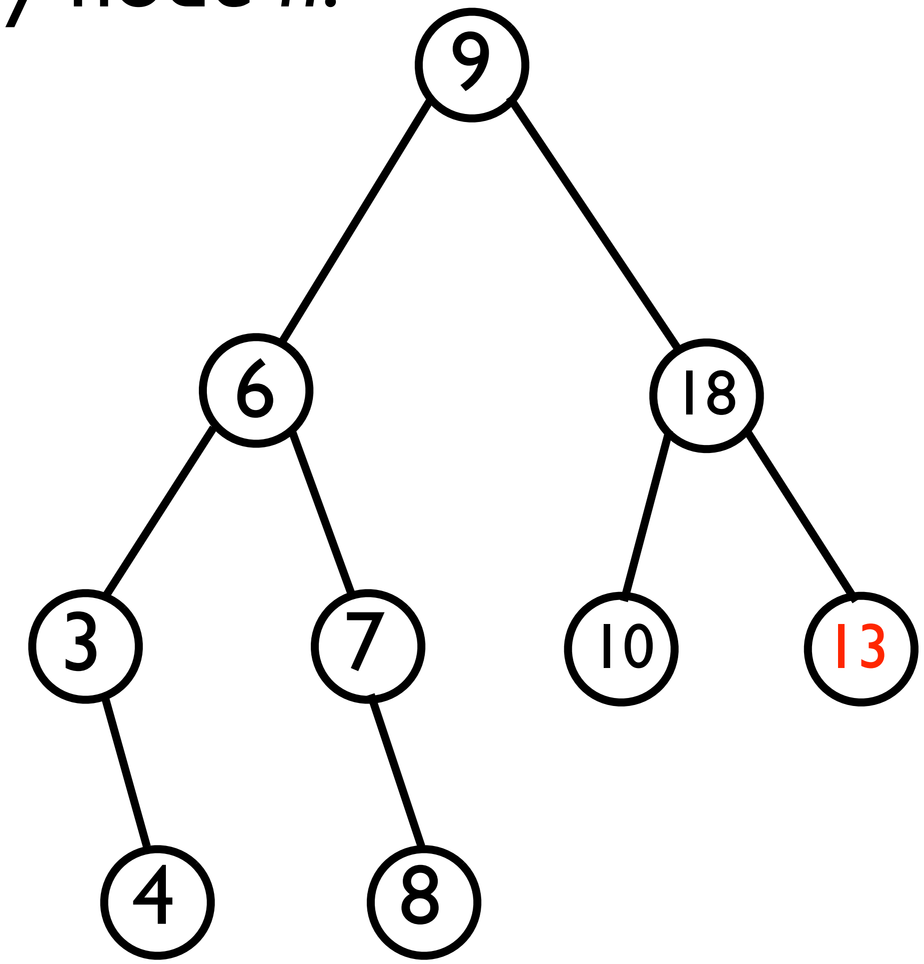
- When removing a node *n* from the BST, we must ensure that:
  - The resulting tree is still *connected*.
  - The resulting tree still has the *ordering property*.
- Consider what might "go wrong" when removing an arbitrary node *n*:



If instead we replace *n* with another node and "reconnect" another branch, we might *violate the ordering property*.

Removing a node

- When removing a node *n* from the BST, we must ensure that:
  - The resulting tree is still *connected*.
  - The resulting tree still has the *ordering property*.
- Consider what might "go wrong" when removing an arbitrary node *n*:



If instead we replace *n* with another node and "reconnect" another branch, we might *violate the ordering property.*

Ordering property is now violated!

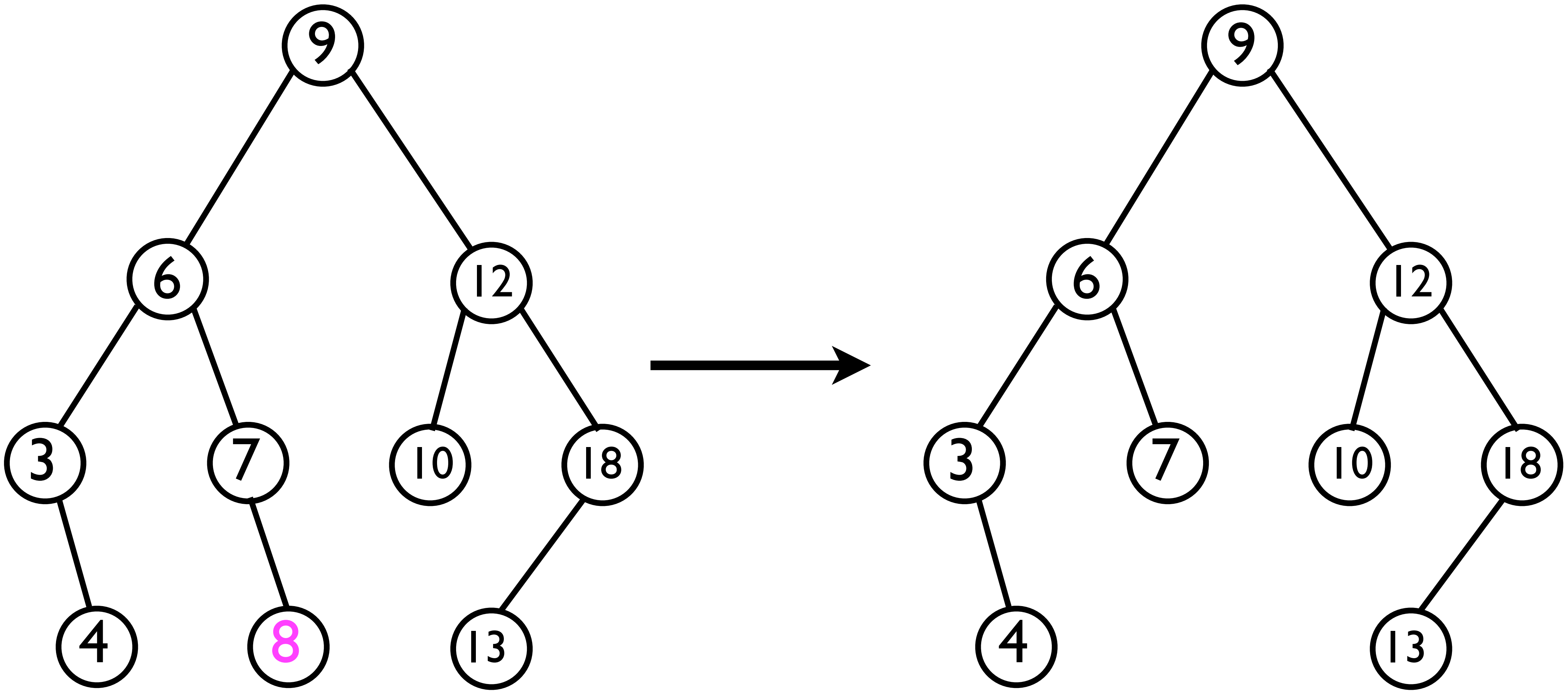
Removing a node

- To remove a node and still ensure the resulting tree is a proper BST, we must distinguish three cases:

1. *n* is a leaf node -- in this case, we just snip it off.
2. *n* is an internal node with only one child.
    - We remove *n* and "splice around" it.
3. *n* is an internal node with two child nodes.
    - We replace *n* with the value of its successor *s*, and then *recursively* remove *s*.

Removing a leaf node

Example: `bst.remove(8);`

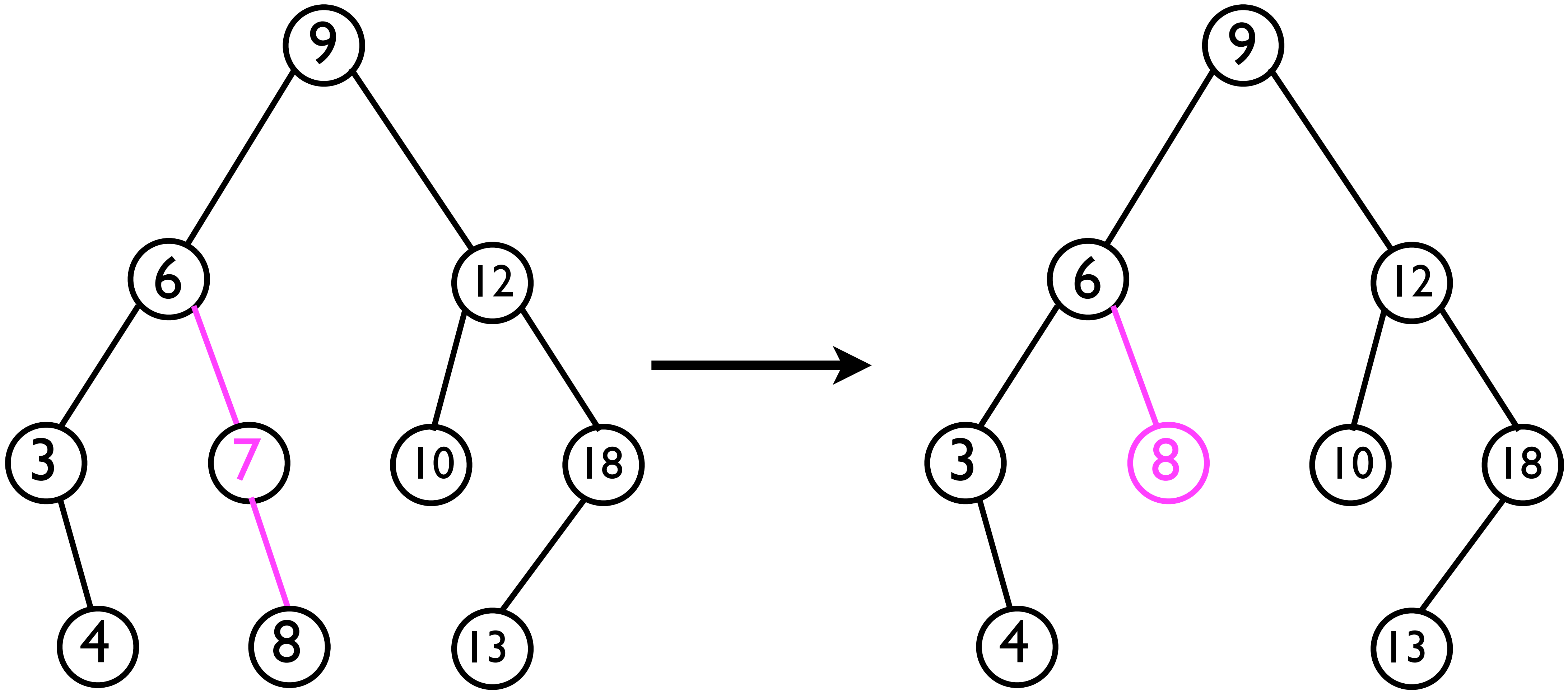


Just snip it off.

Result: We still have a BST with the ordering property preserved.

Removing a node with one child node

Example: `bst.remove(7);`

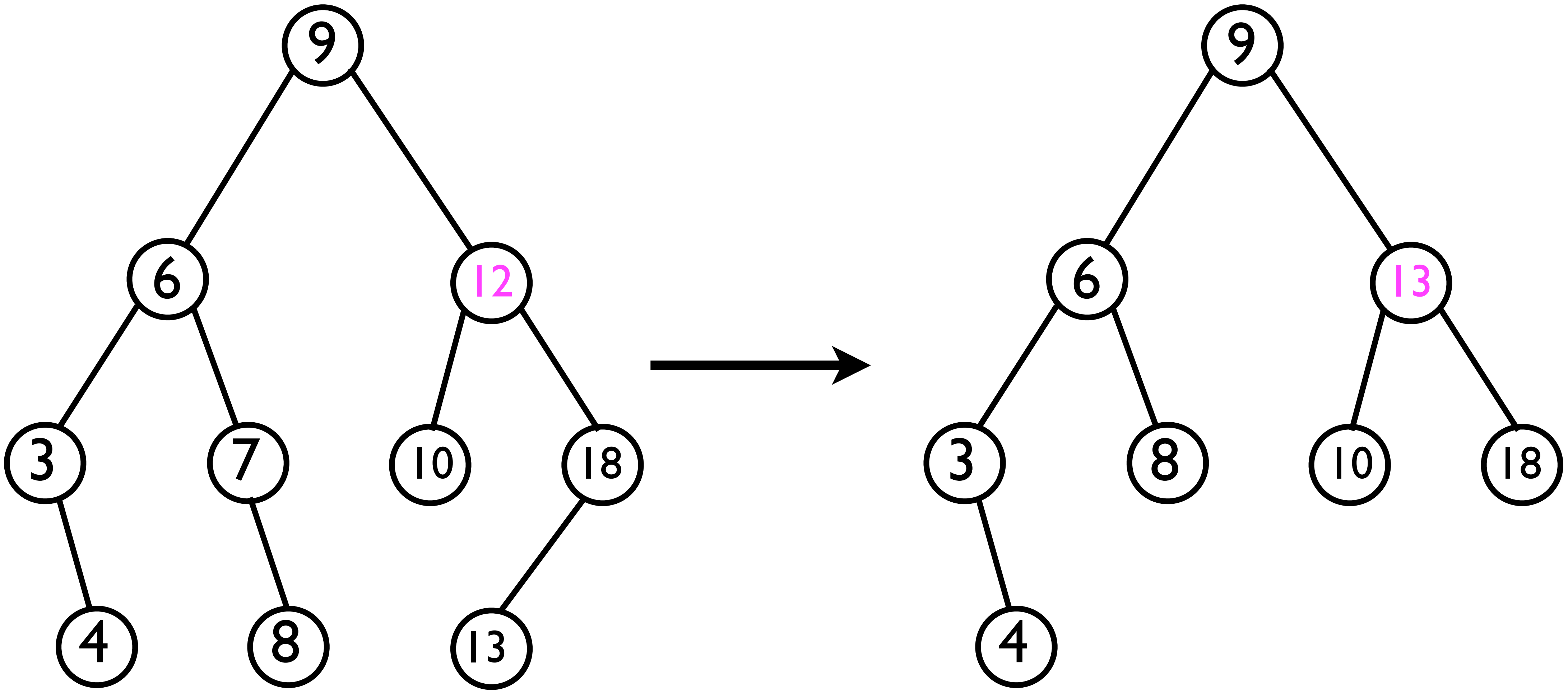


"Splice around" node 7.

Result: We still have a BST with the ordering property preserved.

Removing a node with two child nodes

Example: `bst.remove(12);`



Replace 12 with the value of its *successor*; then remove the successor node.

Result: We still have a BST with the ordering property preserved.

Removing the successor

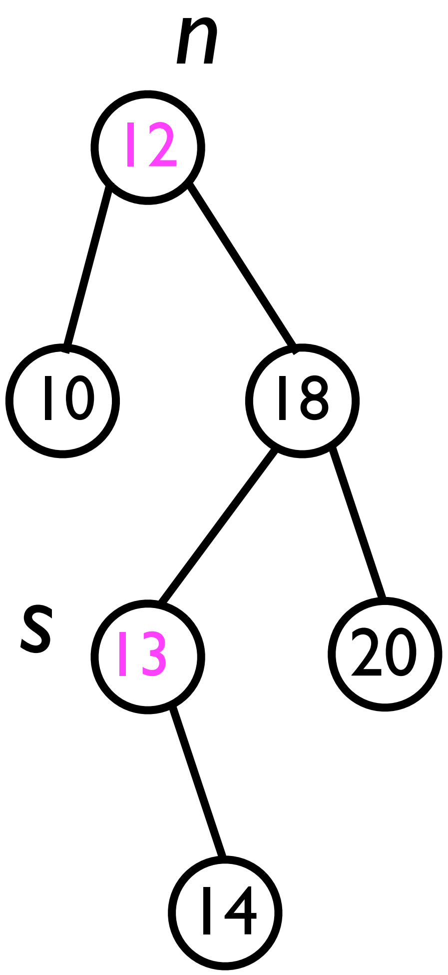
- When removing a node *n* with two children, we replace n with the value of its successor *s*, and then remove s itself.
  - But what if *s also* has two children; then we need to remove *its* successor, and so on.
  - Will the "removal" process ever terminate?
    - Yes -- if *n* has two children, then its successor *s cannot* have a left-child. Why?

Removing the successor

- When removing a node *n* with two children, we replace n with the value of its successor *s*, and then remove s itself.
  - But what if s *also* has two children; then we need to remove *its* successor, and so on.
  - Will the "removal" process ever terminate?
    - Yes -- if *n* has two children, then its successor s *cannot* have a left-child. Why?
      - If it did, s's *that left child* would be n's successor, and not s itself.

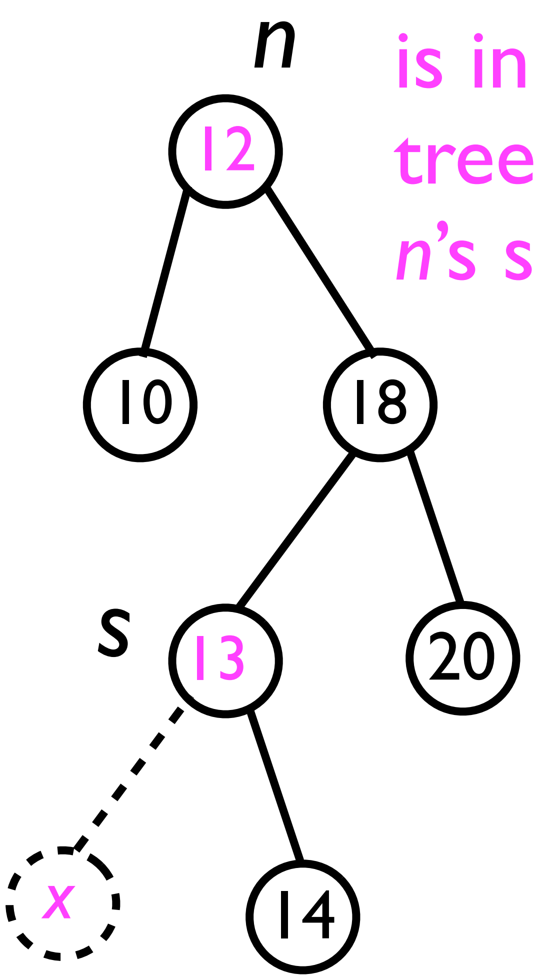
Successor of node with two children

- Example:
    - Let *n* be node 12.
    - Then *n*'s successor *s* is 13.
        - *s* only has one child.



Successor of node with two children

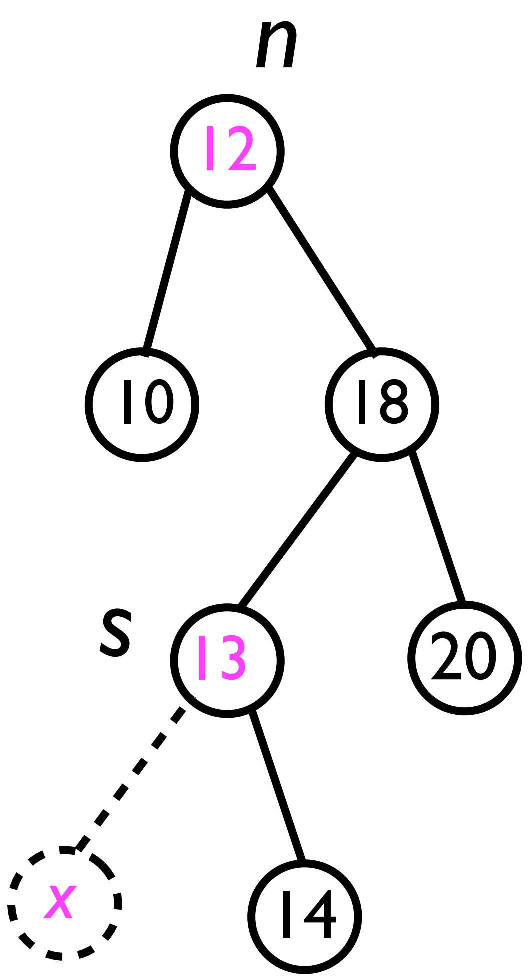
- Example:
  - Let *n* be node 12.
  - Then *n*'s successor *s* is 13.
    - *s* only has one child.
  - Suppose *s* had two children.
    - Then it would have a left child, *x*.
    - *Then x would have to be n's successor.*



Since *x* is still in *n*'s right sub-tree, *x*>12. And since *x* is in *s*'s left sub-tree, *x*<13. So, *x* is *n*'s successor.

Successor of node with two children

- We conclude that, if *n* has two children, then its successor *s* cannot have two children.
- Hence, removing *s* amounts to either just "snipping it off" (case 1), or "slicing around it" (case 2).
- Hence, the **`remove`** method will in fact terminate.



`remove(o)`

- We can finally define the **`remove(o)`** method:

```
void remove (T o) {
  final Node<T> node = findNode(_root, o);
  removeNode(node);
}

void removeNode (Node<T> node) {  // Helper method
  if (node._leftChild == null &&
      node._rightChild == null) {
    // “Snip” node from its parent
  } else if (node._leftChild == null ||
      node._rightChild == null) {
    // “Splice around” node
  } else {
    final Node<T> successor = findSuccessor(_root, o);
    node._data = successor._data;
    removeNode(successor);
  }
}
```

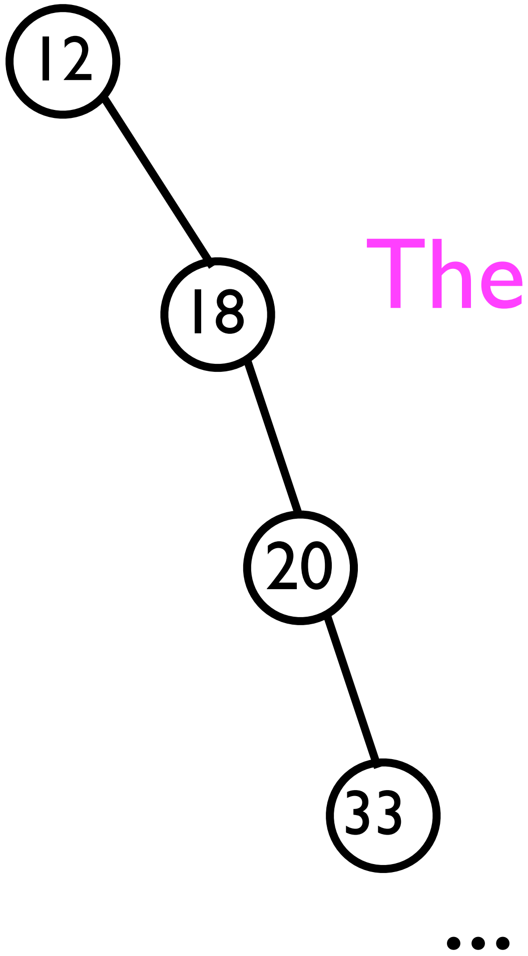
BSTs: Time costs of methods

- All of the fundamental operations -- **`add(o)`**, **`find(o)`**, **`remove(o)`**, and **`findLargest`**/**`findSmallest`** -- take time *O*(*h*), where *h* is the height of the BST.
- In the *average case*, the height *h* of the BST is log *n*.
- What about in the *worst case*?

BSTs:
Time costs of methods

- In the *worst case*, the user will call **`add`** and **`remove`** in an "unfortunate" order, resulting in a "degenerate" BST of the following variety:

- In this case, the height of the BST is *n* -- and hence the fundamental BST operations would also be *O*(*n*).



The "BST" is just a linked list!

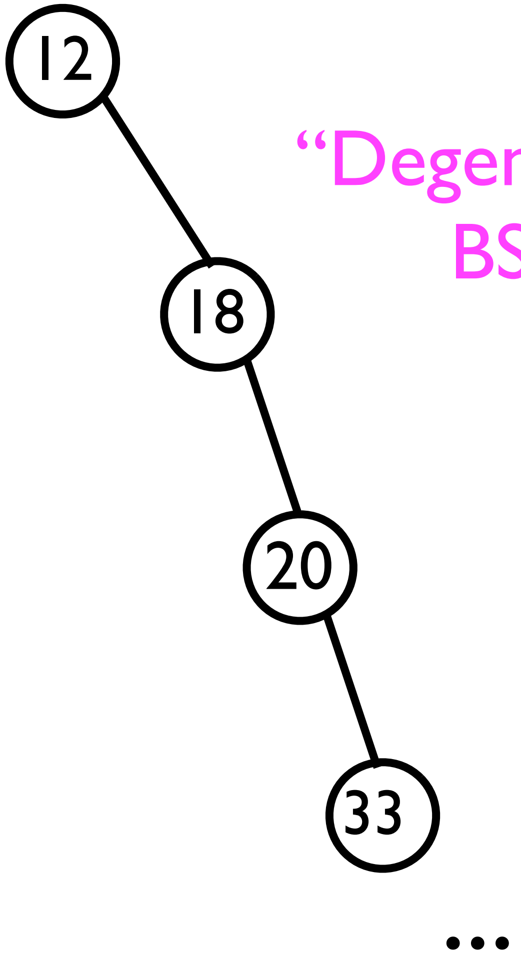
Balancing BSTs

- To prevent this “worst-case” condition from occurring, we need to employ some form of “tree balancing” to keep the tree from degenerating into a linked list.
- Two prominent data structures which ensure a *balanced tree* include:
  - AVL trees.
  - Red-black trees.

AVL trees.

Maintaining balance

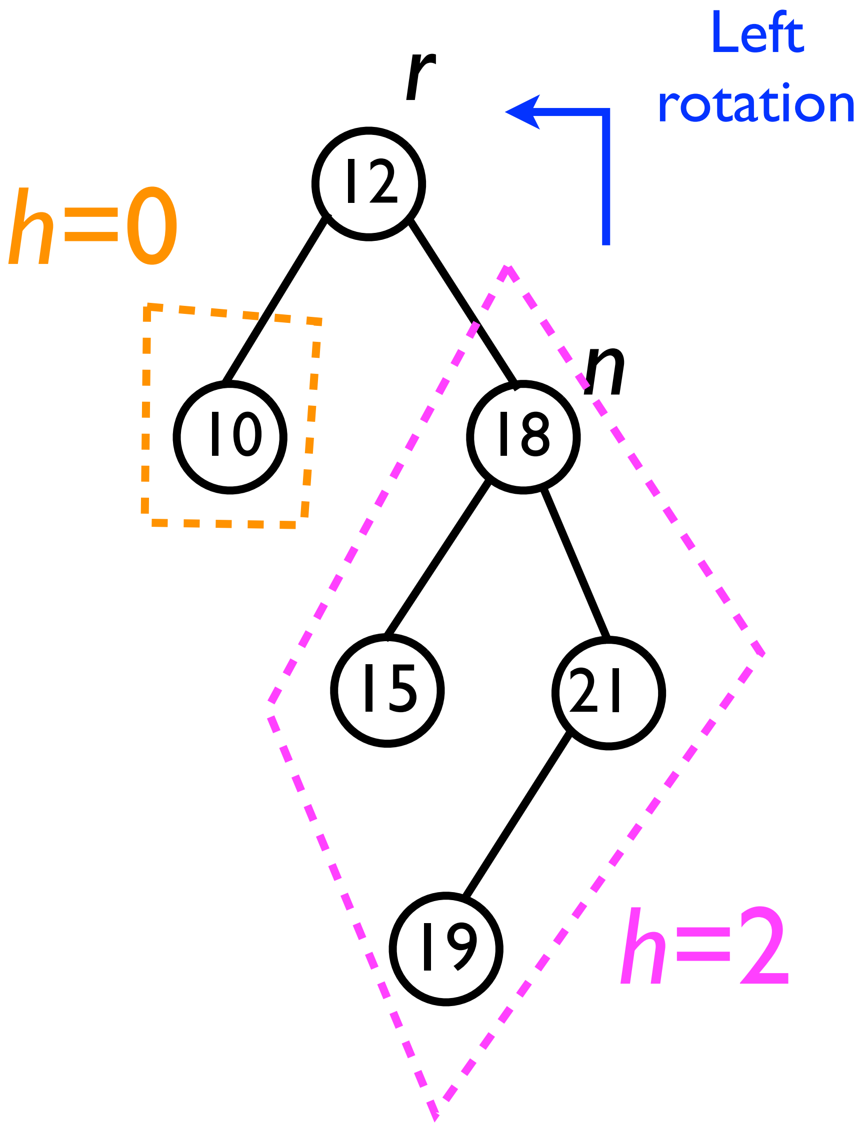
- The time cost of the fundamental add/find/remove operations in BSTs depends on the *height* of the BST.
- Given an “unfortunate” sequence of add/remove operations, the BST can “degenerate” into a long “chain” of nodes of height *n*.
  - Hence, in the worst case, the time cost of the fundamental BST operations is $O(n)$.
  - It would be beneficial to *prevent* this worst case from ever occurring.



“Degenerate” BST.

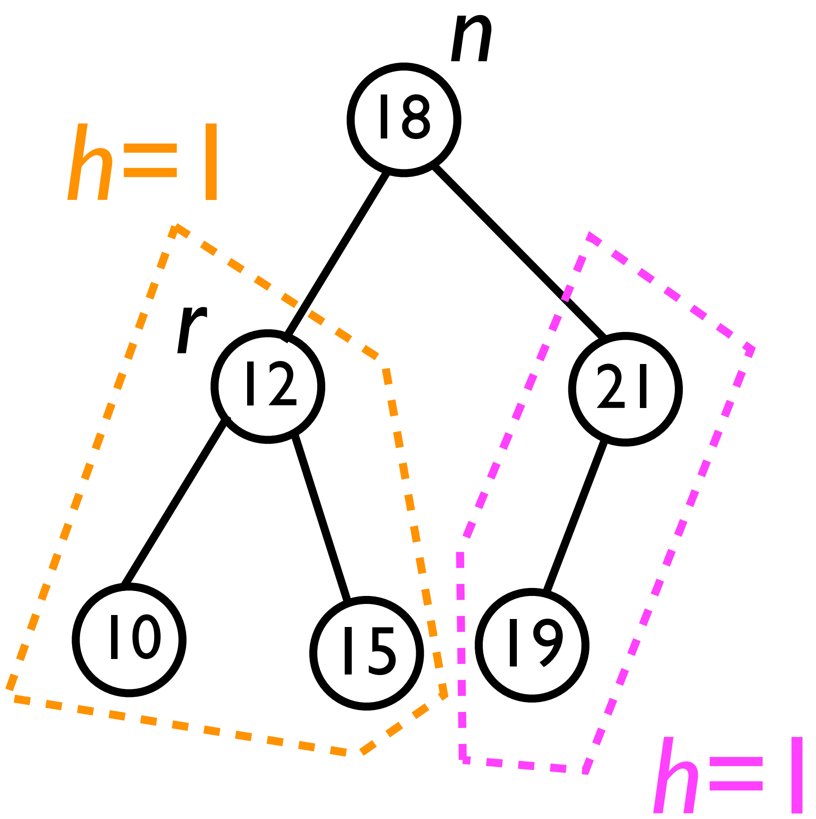
Maintaining balance

- Fortunately, it turns out that BSTs can be "fixed" to store the *same elements*, but to have a *smaller height*.
- Consider the BST on the right (with root *r*) with height 3.
  - It is *unbalanced* -- height of left sub-tree is 0, height of right sub-tree is 2.
- We can "fix" this BST to have *equal* height on both sub-trees by "rotating" node *n* towards *r*.



Maintaining balance

- Fortunately, it turns out that BSTs can be "fixed" to store the *same elements*, but to have a *smaller height*.
- Consider the BST on the right (with root *r*) with height 3.
    - It is *unbalanced* -- height of left sub-tree is 0, height of right sub-tree is 2.
- We can "fix" this BST to have *equal* height on both sub-trees by "rotating" node *n* towards *r*.



New root is *n*.
Height of BST is 2.
Left and right sub-trees both have height 1 (the BST is *balanced*).

Maintaining balance

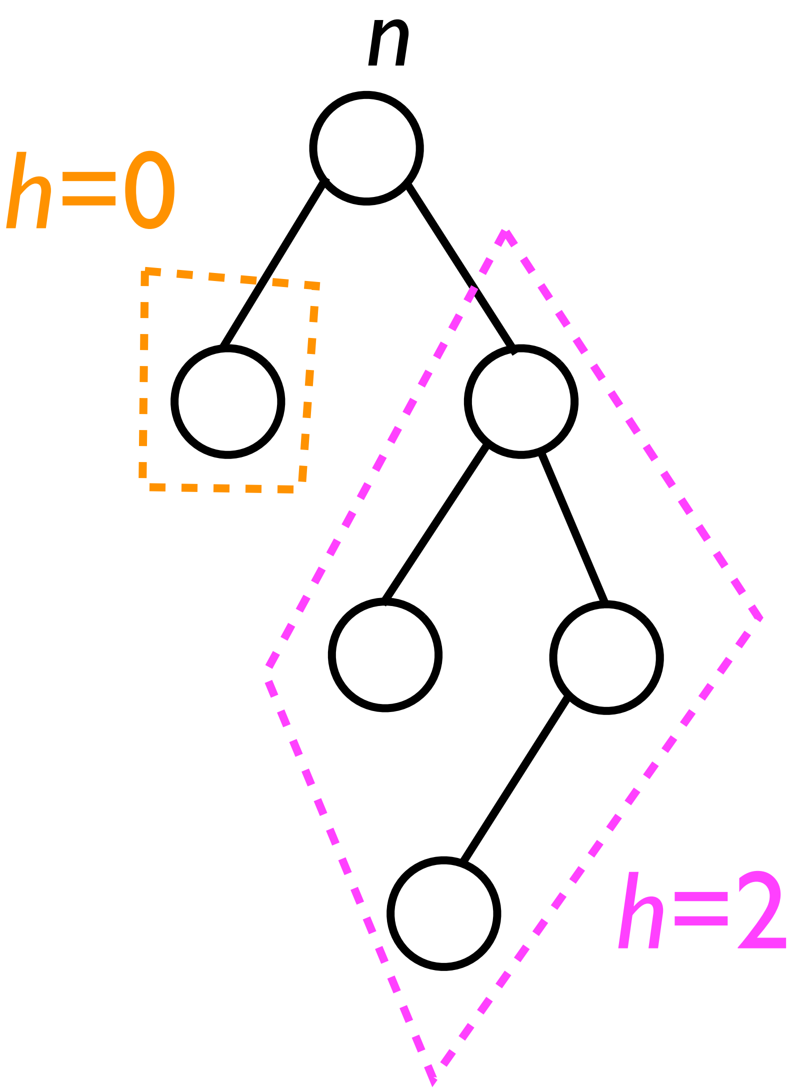
- By rotating nodes to either "up-to-the-left" or "up-to-the-right", we can restore *balance* to a BST and thereby *decrease its height*.
- The rotations will take place whenever the user `adds` or `removes` a node from the BST.
- By rotating properly, we can ensure that the BST remains balanced or "almost balanced" at all times.
- This system of node rotations was first developed in 1962 by G.M. Adelson-Velskii and E.M. Landis; hence, we call this technique an AVL-tree.

AVL trees

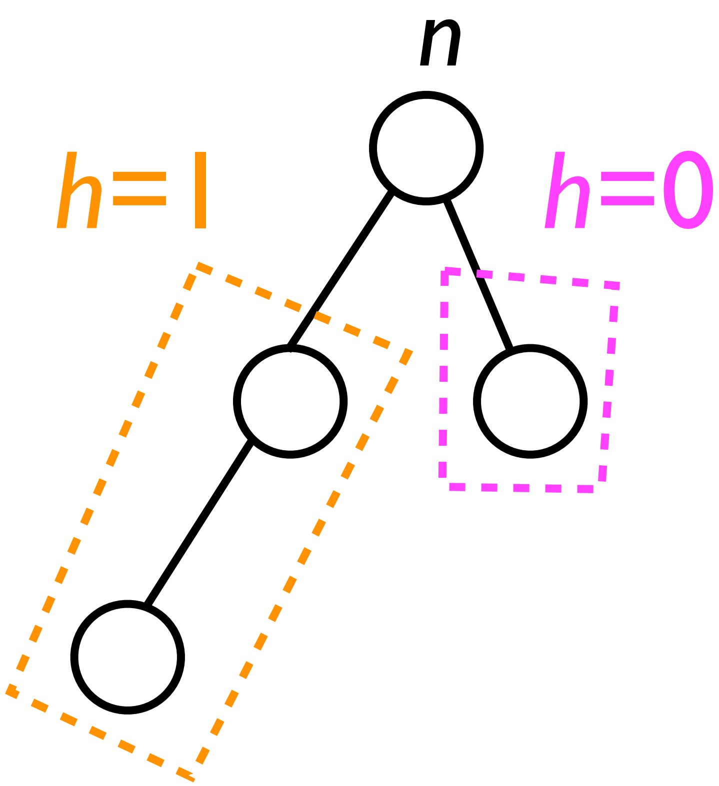
- An AVL tree is a BST in which two kinds of rotations -- *left-rotations* and *right-rotations* -- are applied to nodes as necessary, in order to keep the *balance* of each sub-tree within certain limits.
- The *balance* of a node *n* is the *difference in height* between *n*'s left sub-tree minus its right sub-tree.
  - A non-existent sub-tree is defined to have height 0.
- Rotations are applied to nodes during the **`add`** and **`remove`** methods to keep *every* node's balance within -1 and +1 (inclusive).

Height and balance

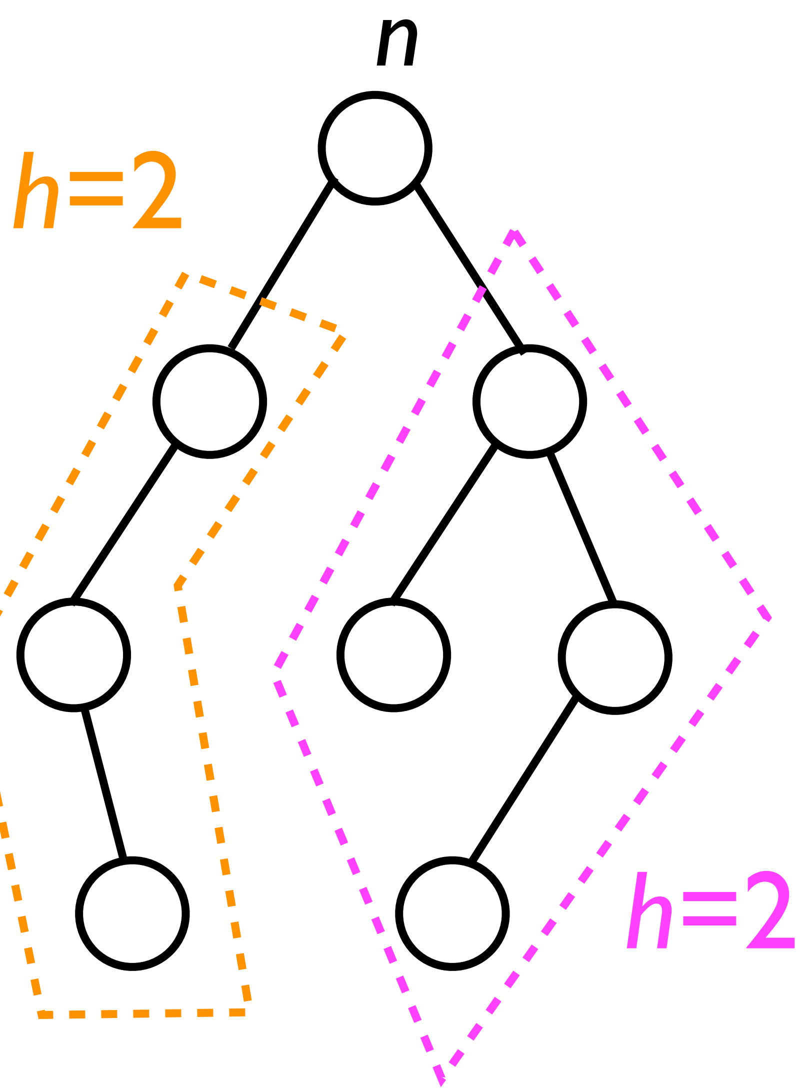
Balance = -2



Balance = +1



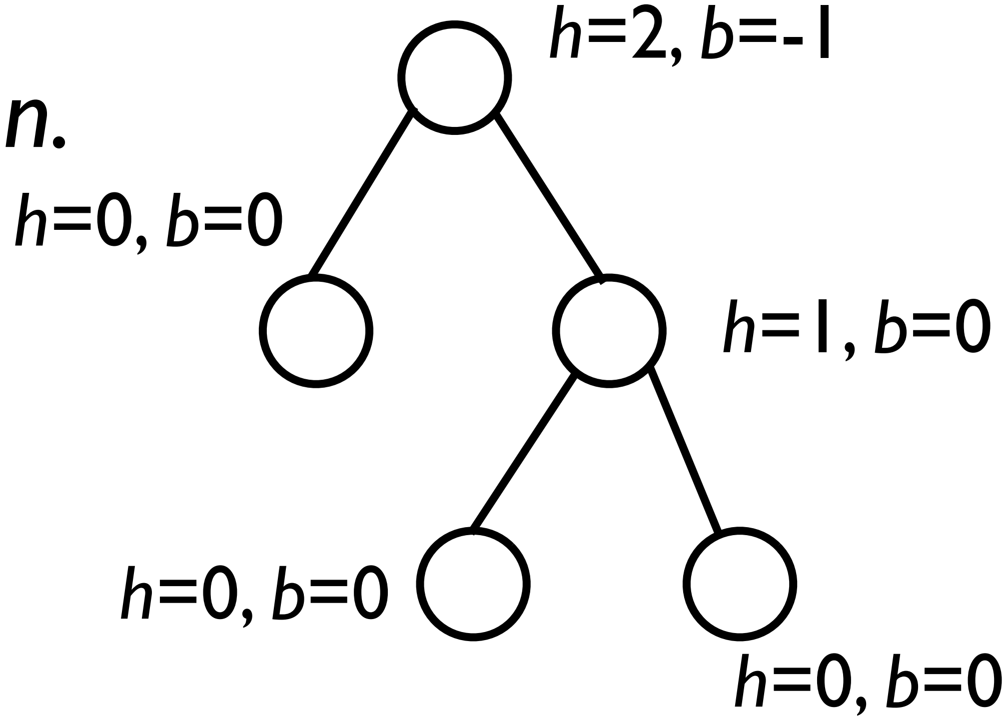
Balance = 0



Height and balance

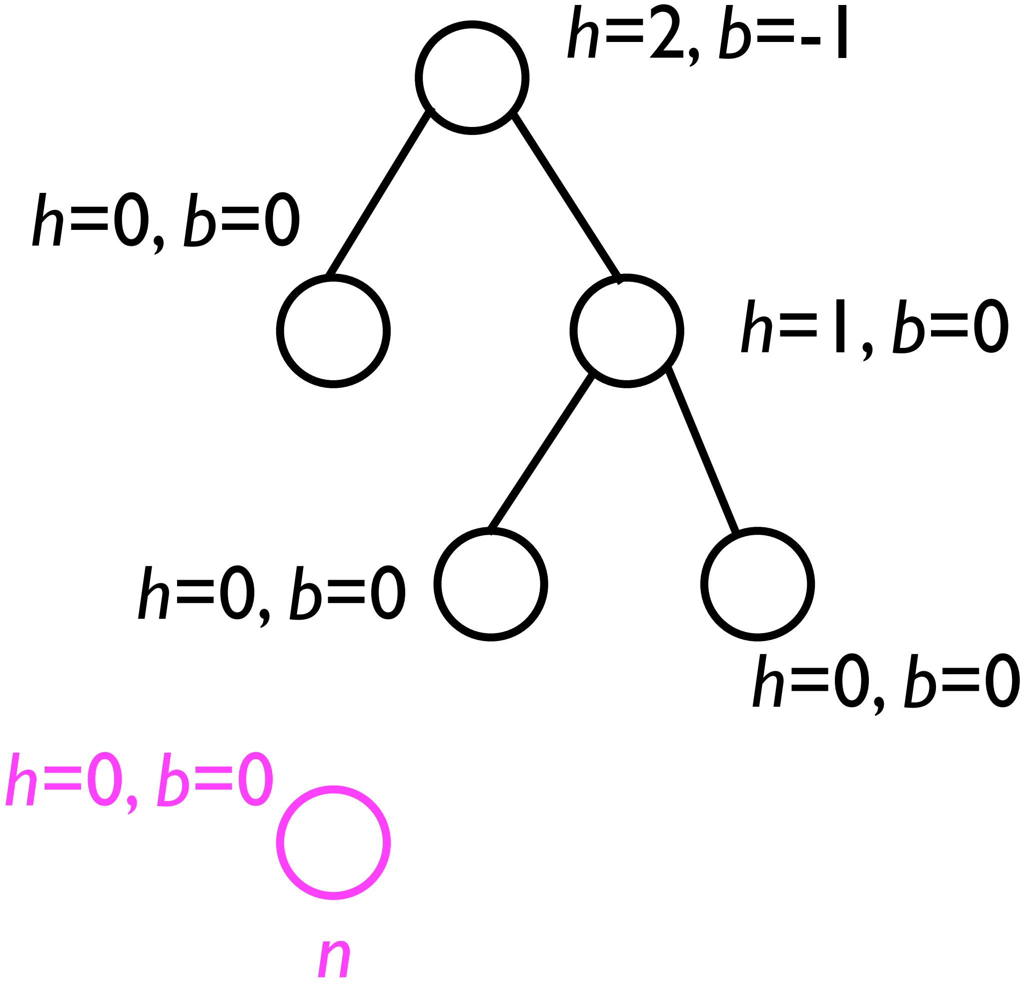
- AVL trees require that each node *n* record its *balance* as well as the *height* of the sub-tree rooted at *n*.
  - We can store these as extra instance variables in the **`Node`** class:

```
class Node<T> {
  Node<T> _parent;
  Node<T> _leftChild, _rightChild;
  int _balance, _height;
}
```



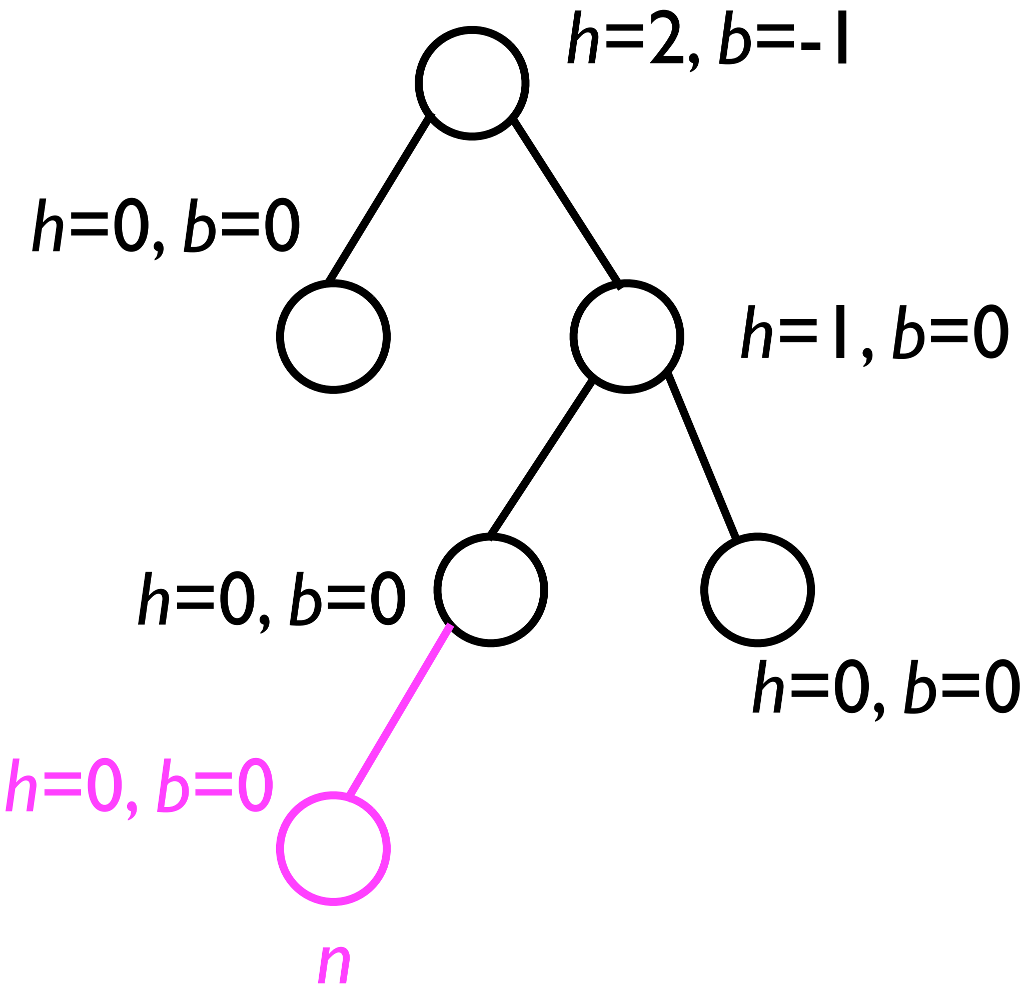
Adding a new node

- Whenever we add a new node *n*, we set its `_height` and `_balance` both to 0.
- We attach *n* as a left/right child of its parent.
- We must then recursively update the height and balance of all nodes from *n* up through the root of the whole BST.



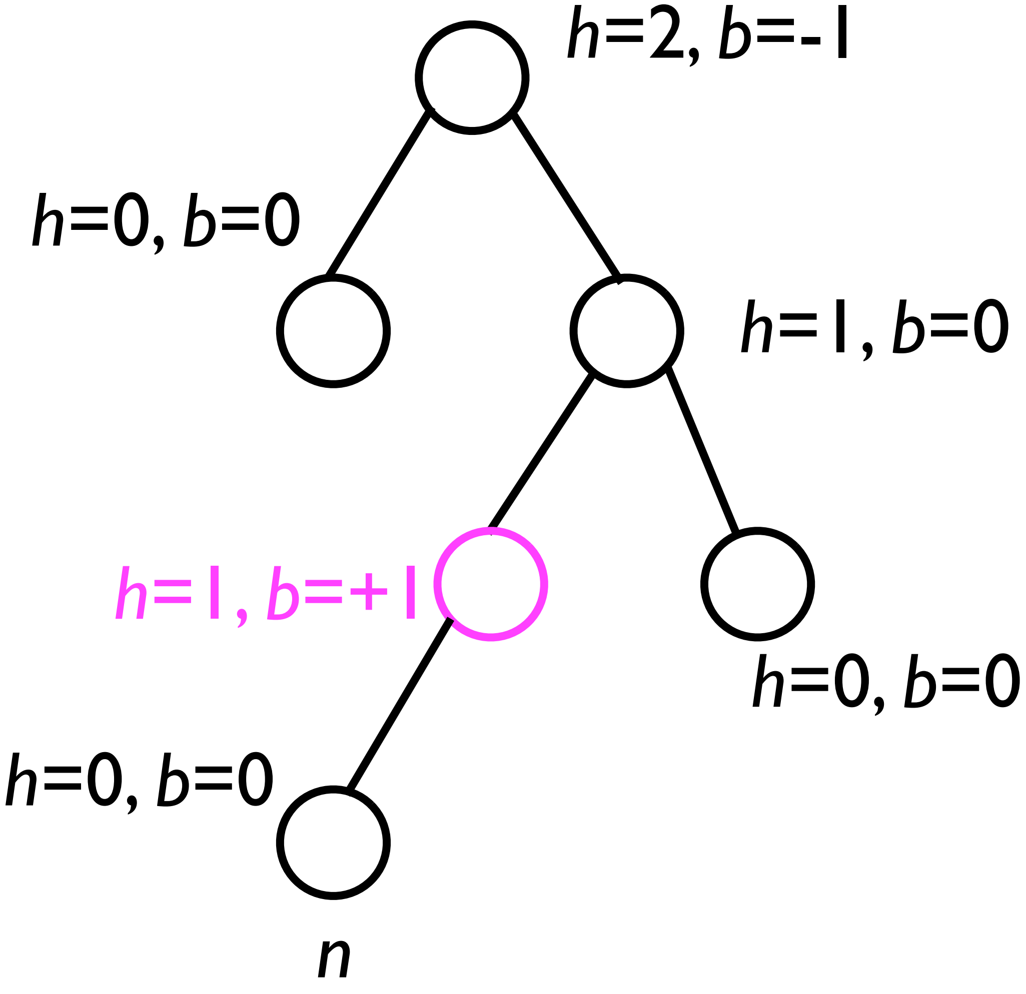
Adding a new node

- Whenever we add a new node *n*, we set its `_height` and `_balance` both to 0.
- We attach *n* as a left/right child of its parent.
- We must then recursively update the height and balance of all nodes from *n* up through the root of the whole BST.



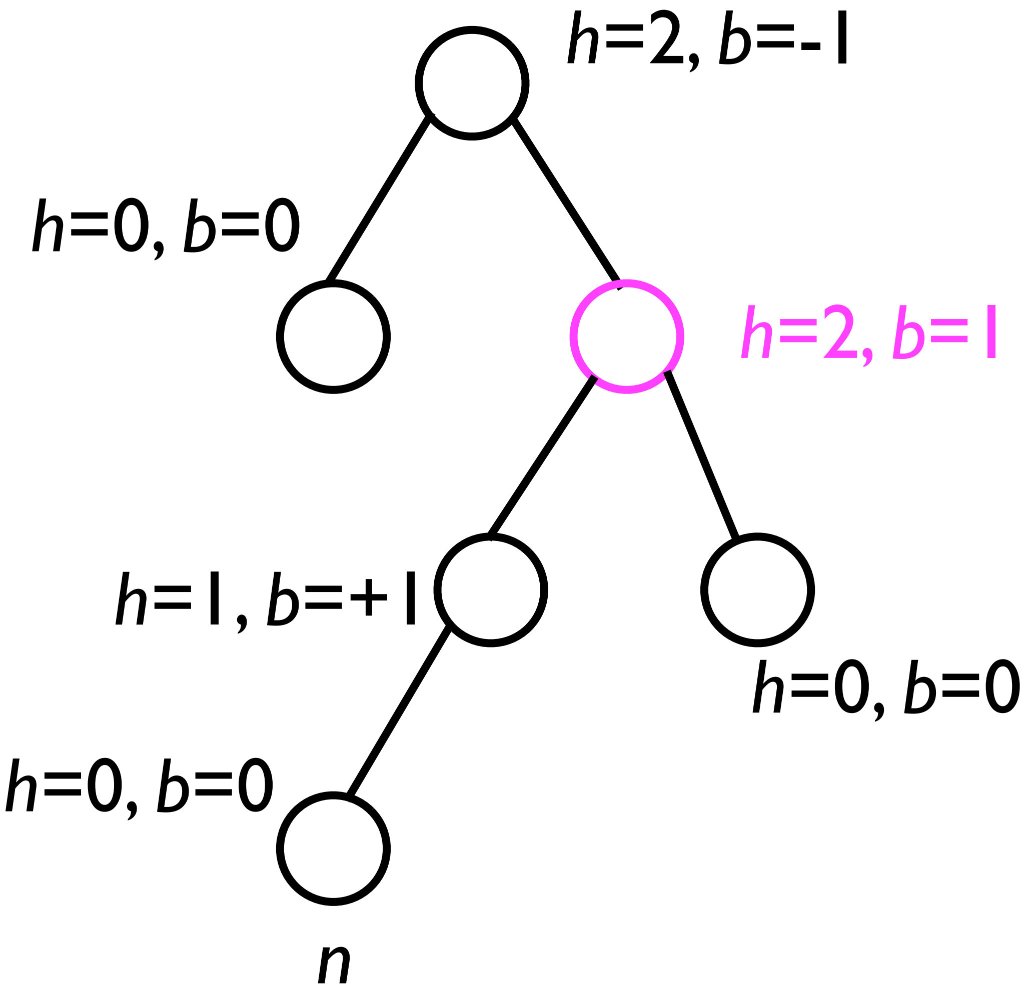
Adding a new node

- Whenever we add a new node *n*, we set its `_height` and `_balance` both to 0.
- We attach *n* as a left/right child of its parent.
- We must then recursively update the height and balance of all nodes from *n* up through the root of the whole BST.



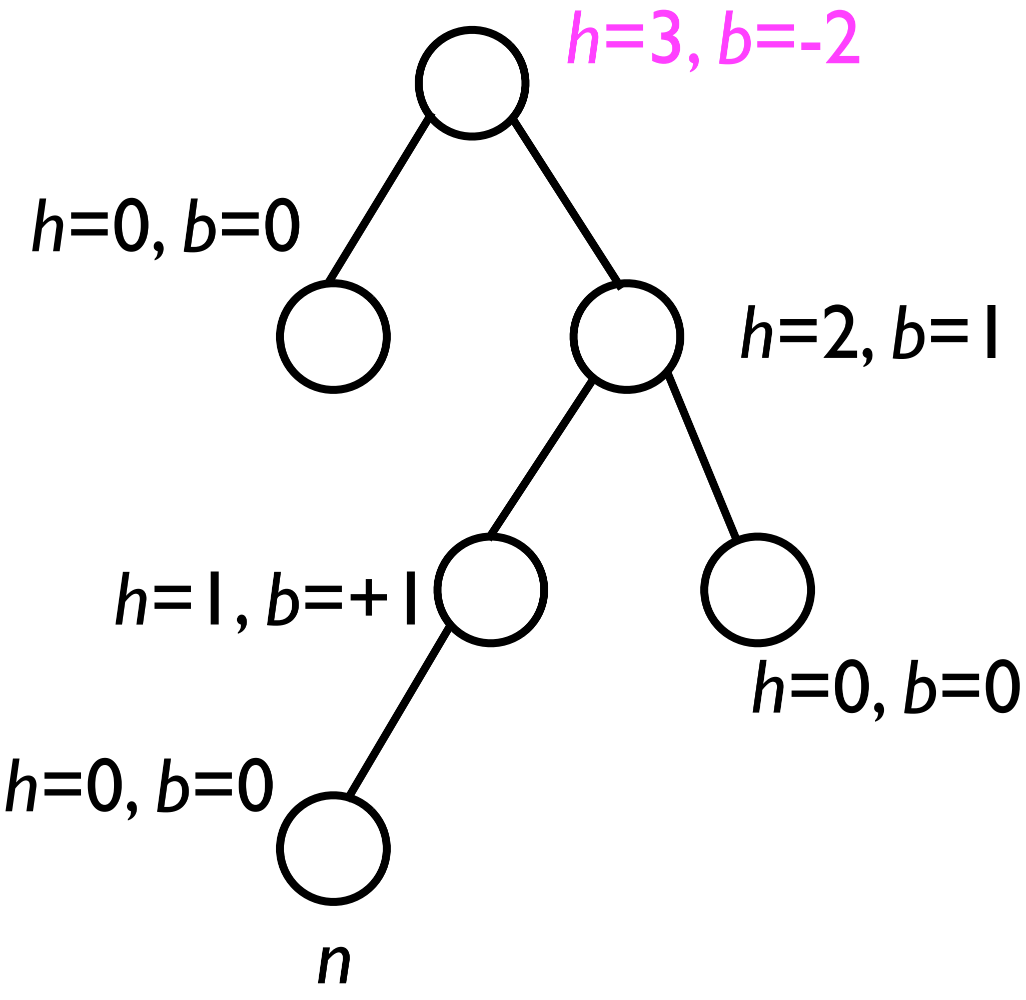
Adding a new node

- Whenever we add a new node *n*, we set its `_height` and `_balance` both to 0.
- We attach *n* as a left/right child of its parent.
- We must then recursively update the height and balance of all nodes from *n* up through the root of the whole BST.



Adding a new node

- Whenever we add a new node *n*, we set its `_height` and `_balance` both to 0.
- We attach *n* as a left/right child of its parent.
- We must then recursively update the height and balance of all nodes from *n* up through the root of the whole BST.



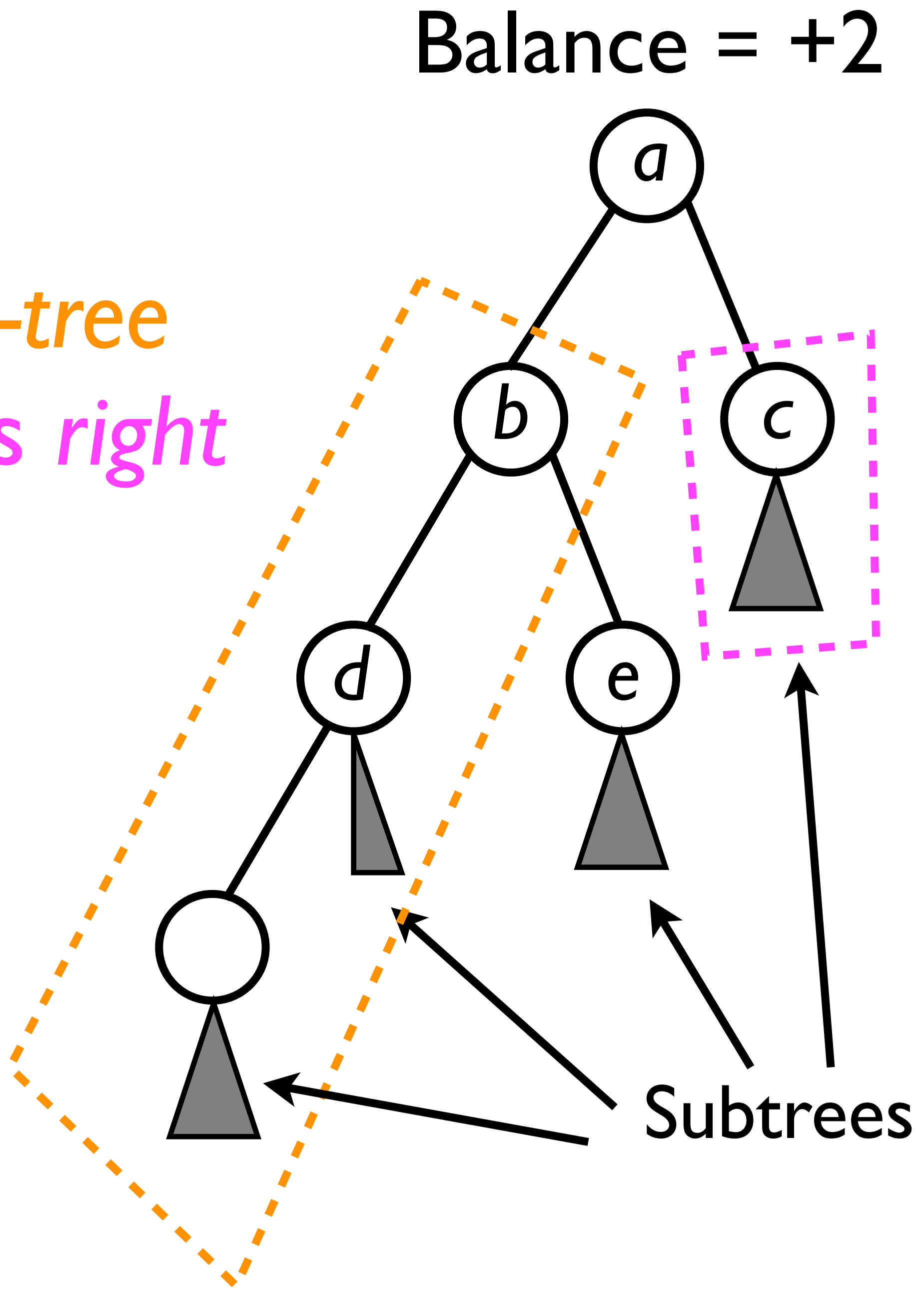
Correcting imbalances

- Suppose, when recursively updating the height and balance data, we determine that the balance of a node *n* is either -2 or +2.
    - *n* is considered *imbalanced*.
- Then we must apply an AVL rotation to *correct the imbalance*.
- Different rotations apply to different node configurations...

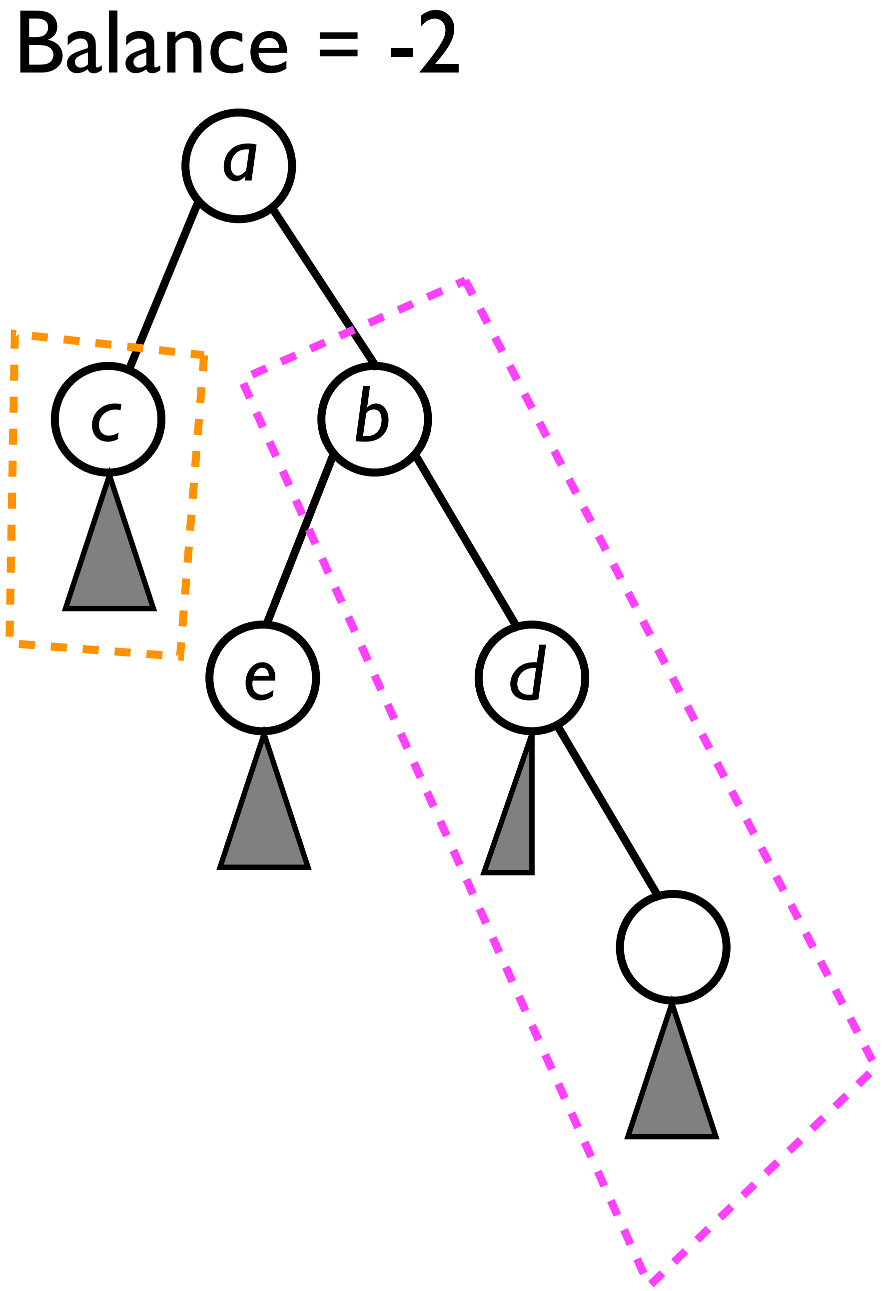
Imbalanced node configurations

The *Left child's Left sub-tree of a* is 2 higher than *a's right sub-tree*.

This case is called LL.



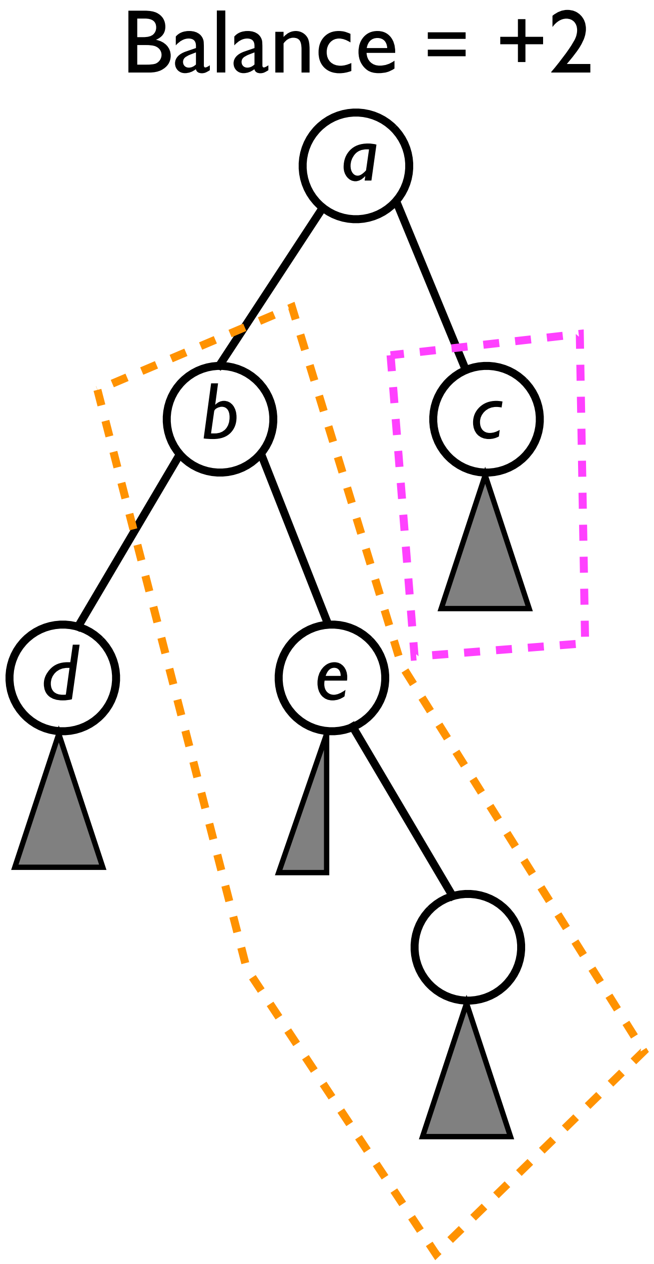
Imbalanced node configurations



The *Right child's Right sub-tree of a* is 2 higher than *a's left sub-tree.*

This case is called RR.

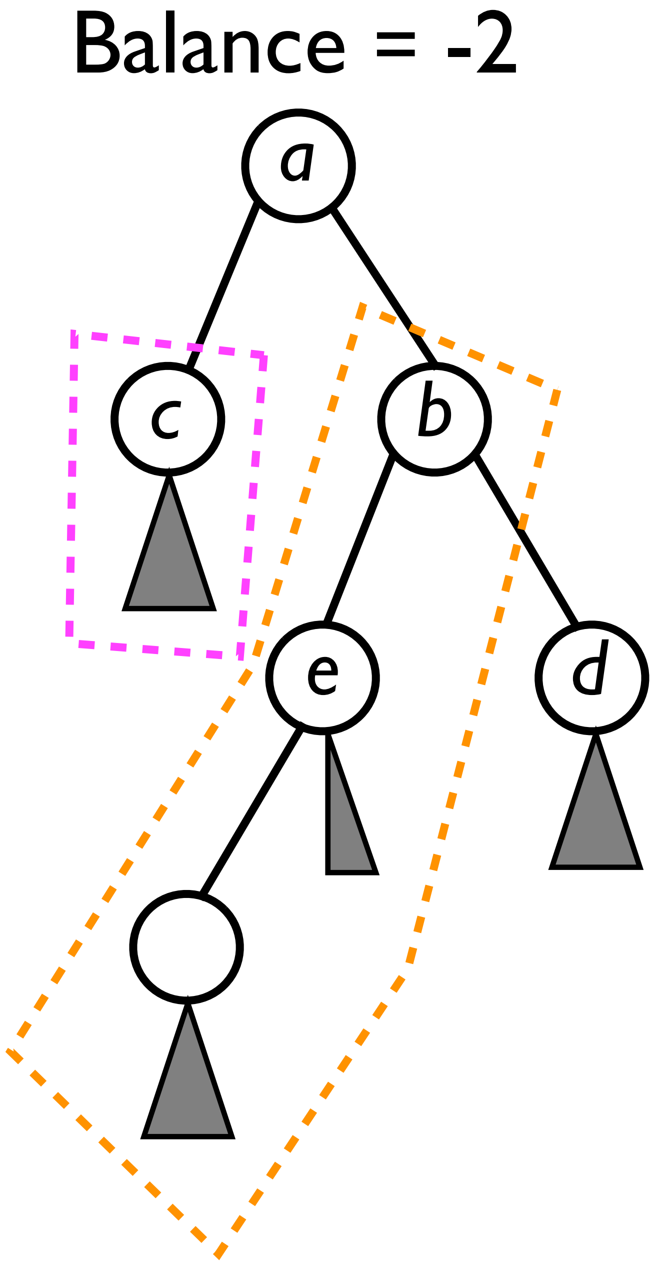
Imbalanced node configurations



The *Left child's Right sub-tree of a* is 2 higher than *a's right sub-tree*.

This case is called LR.

Imbalanced node configurations

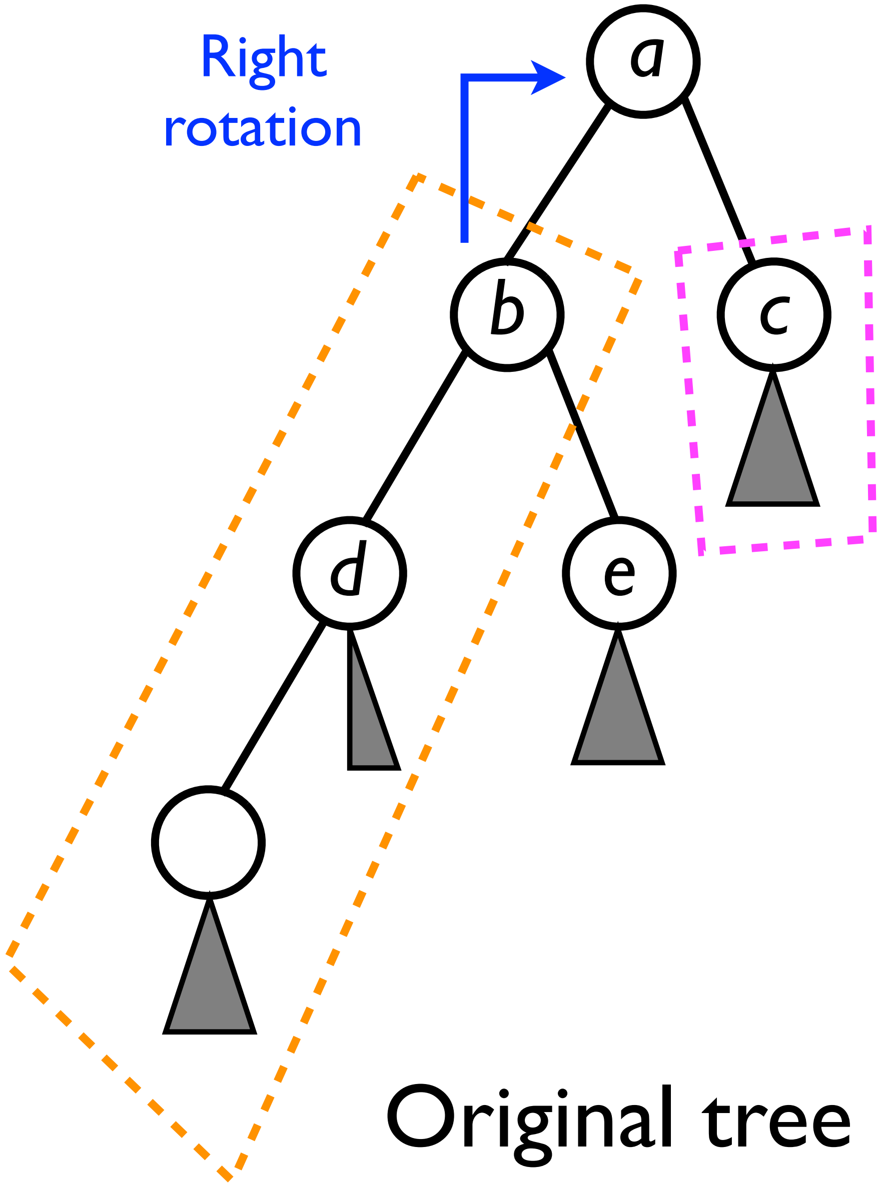


The *Right child's Left sub-tree of a* is 2 higher than *a's left sub-tree*.

This case is called RL.

Fixing configuration LL

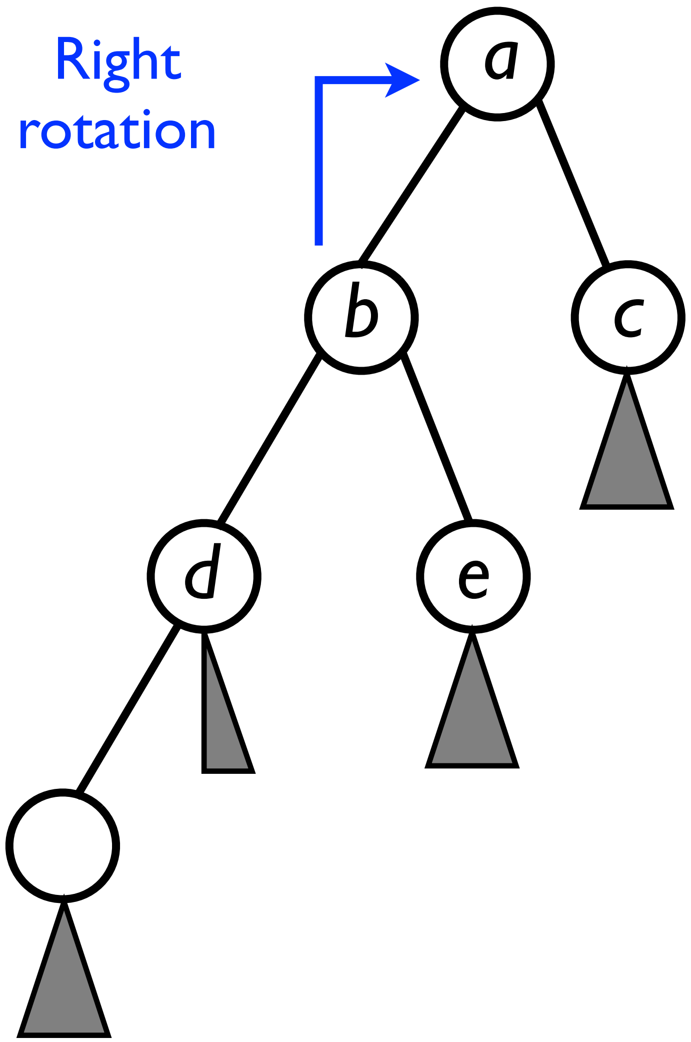
- To fix the imbalance in node *a*, we will perform a *right rotation* of node *b* towards *a*.



Original tree

Fixing configuration LL

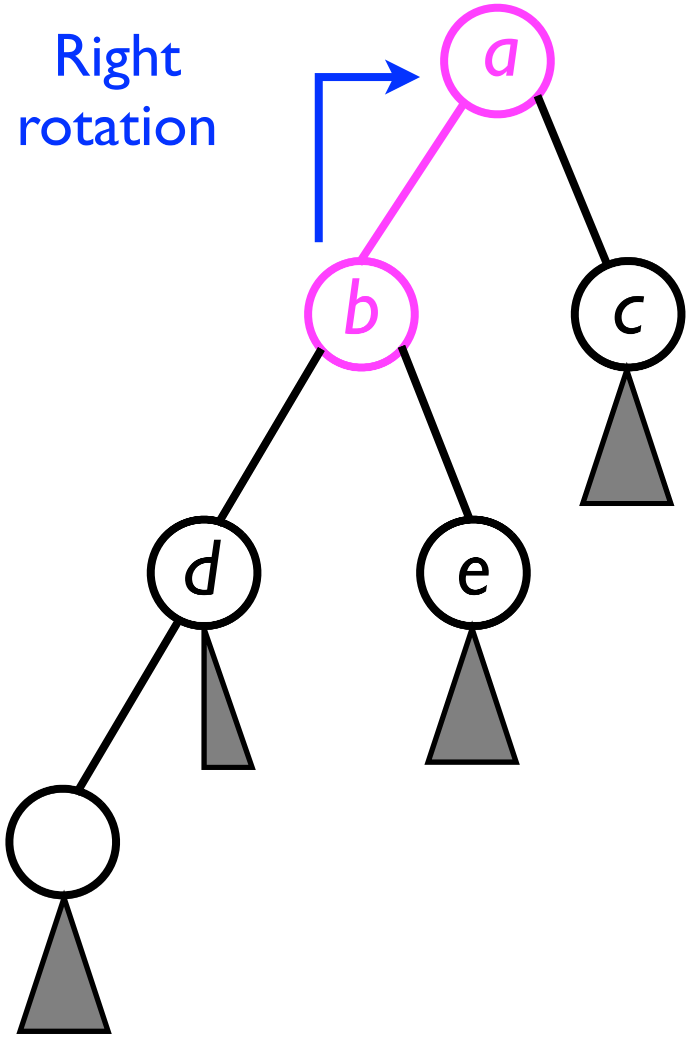
- To fix the imbalance in node *a*, we will perform a *right rotation* of node *b* towards *a*.



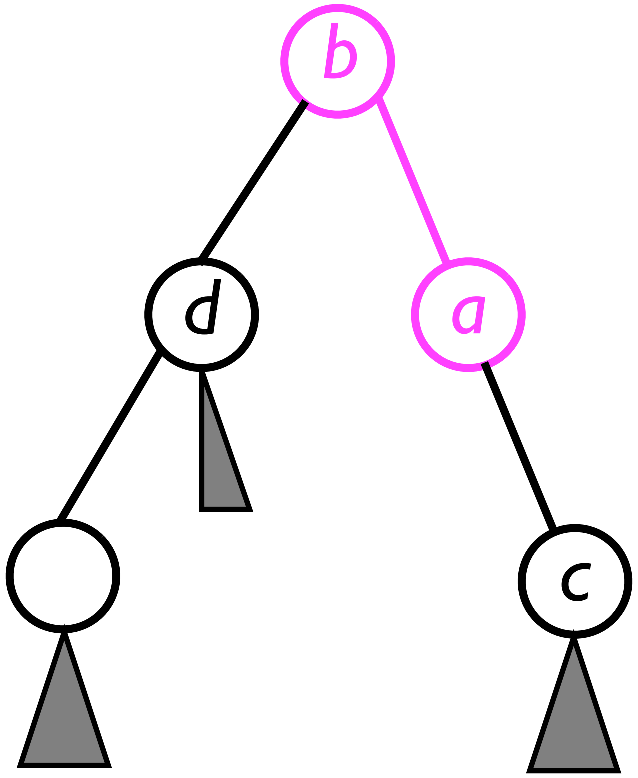
Original tree

Fixing configuration LL

- To fix the imbalance in node *a*, we will perform a *right rotation* of node *b* towards *a*.



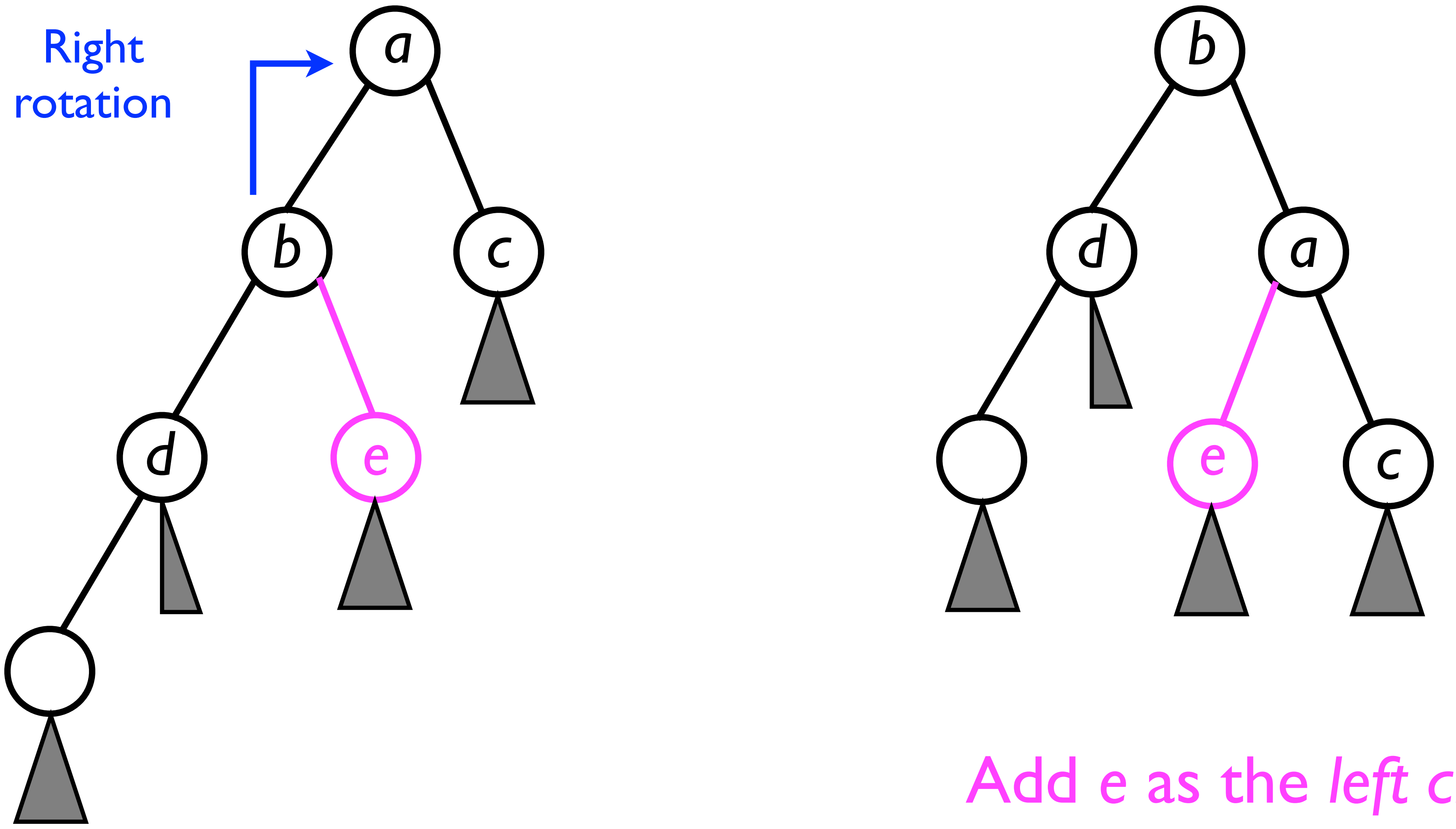
Original tree



Make *a* the *right child* of b, and make *b* the new root of the sub-tree.

Fixing configuration LL

- To fix the imbalance in node *a*, we will perform a *right rotation* of node *b* towards *a*.

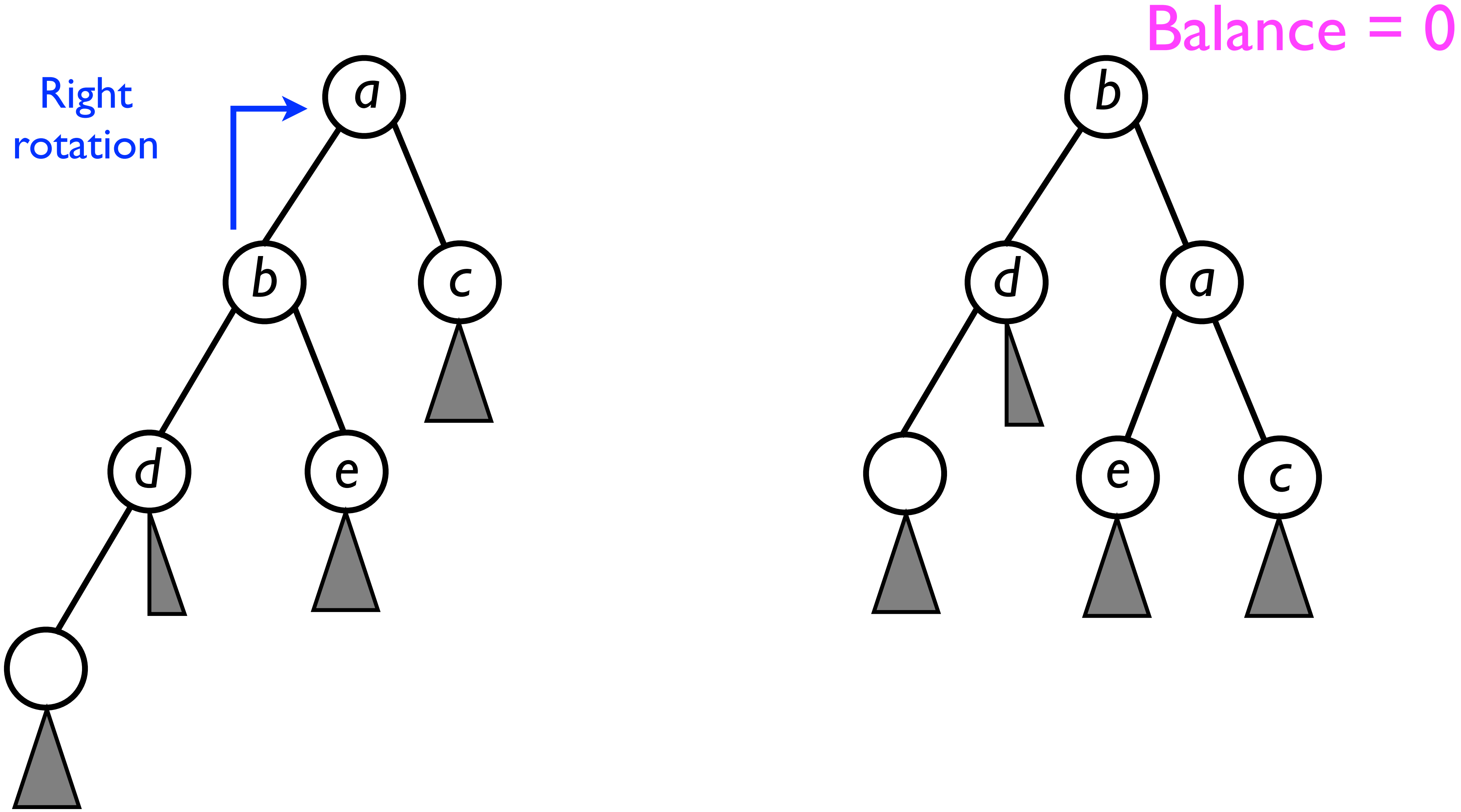


Original tree

Add e as the *left child* of *a*.

Fixing configuration LL

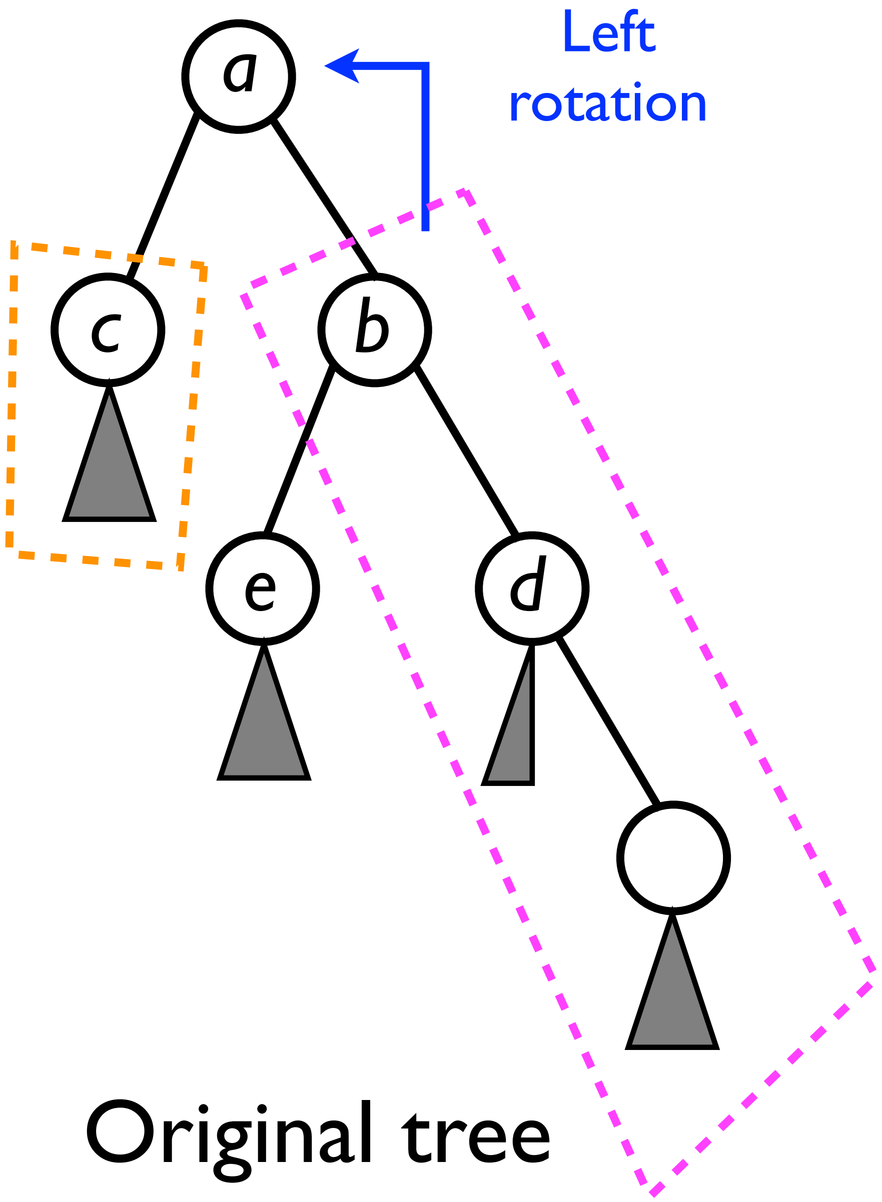
- To fix the imbalance in node *a*, we will perform a *right rotation* of node *b* towards *a*.



Original tree

Fixing configuration RR

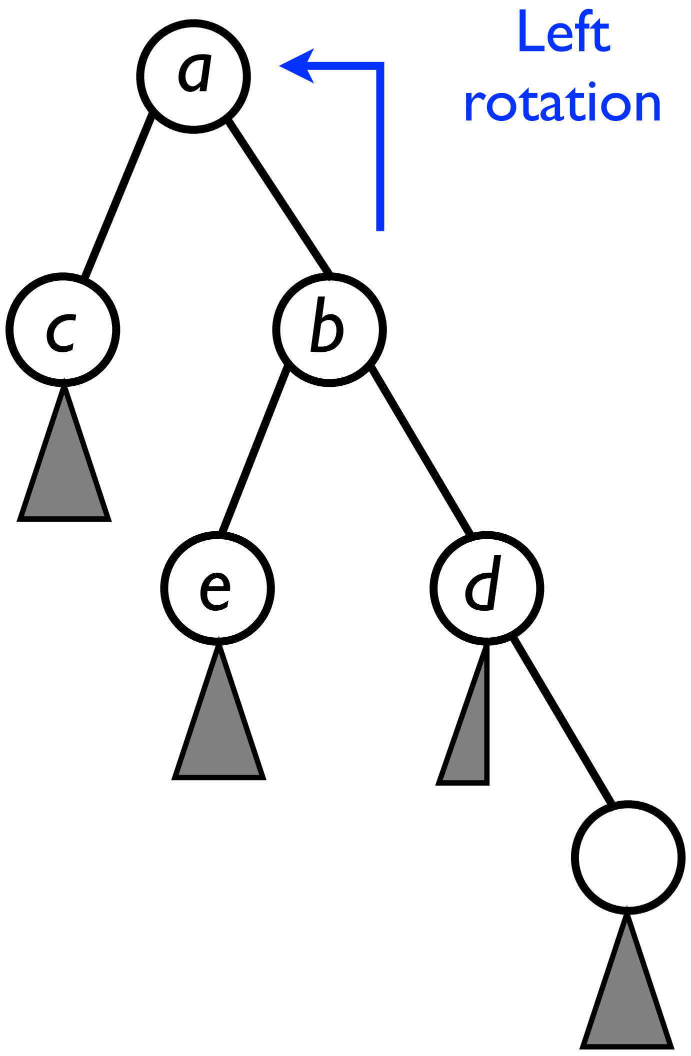
- To fix the imbalance in node *a*, we will perform a *left rotation* of node *b* towards *a*.



Original tree

Fixing configuration RR

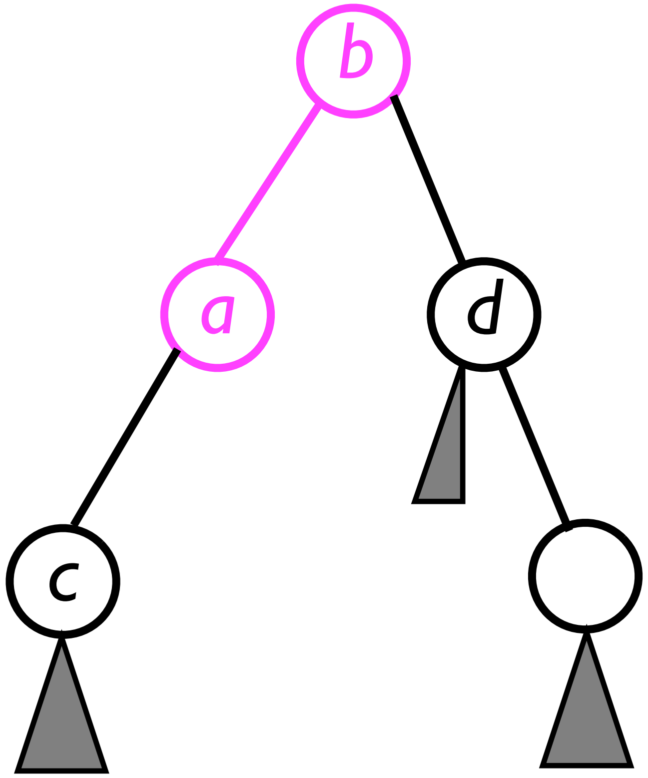
- To fix the imbalance in node *a*, we will perform a *left rotation* of node *b* towards *a*.



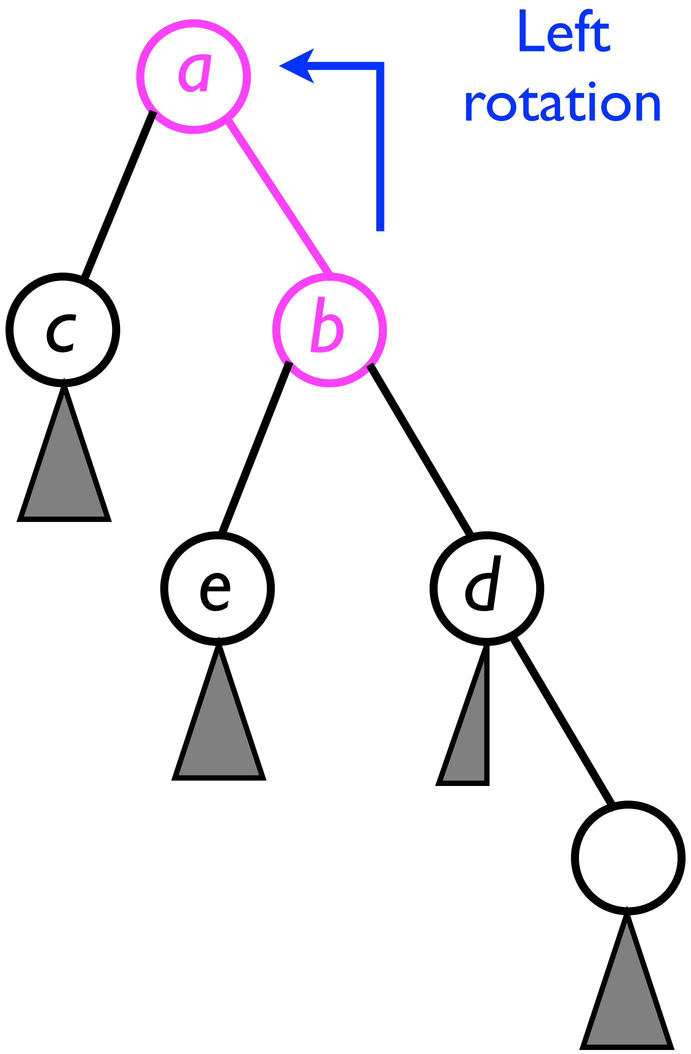
Original tree

Fixing configuration RR

- To fix the imbalance in node *a*, we will perform a *left rotation* of node *b* towards *a*.



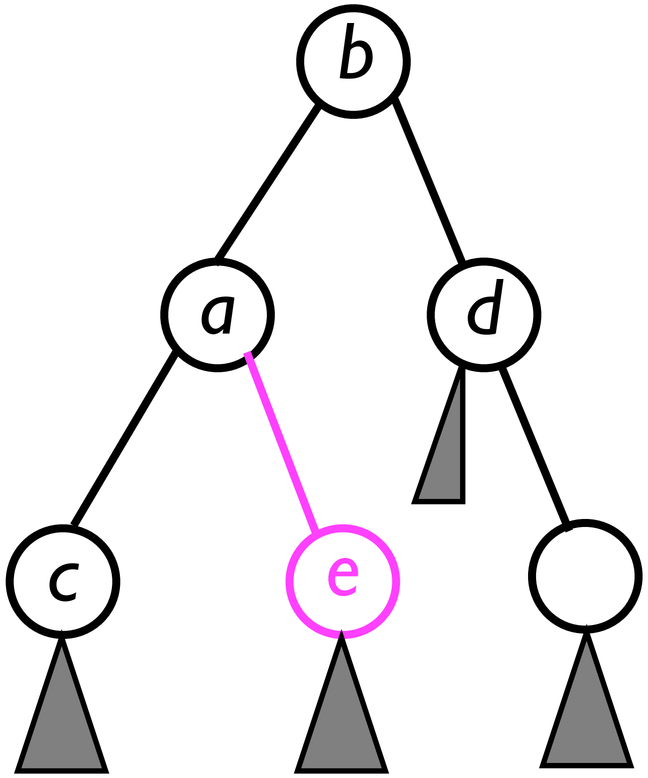
Make *a* the *right child* of b, and make *b* the new root of the sub-tree.



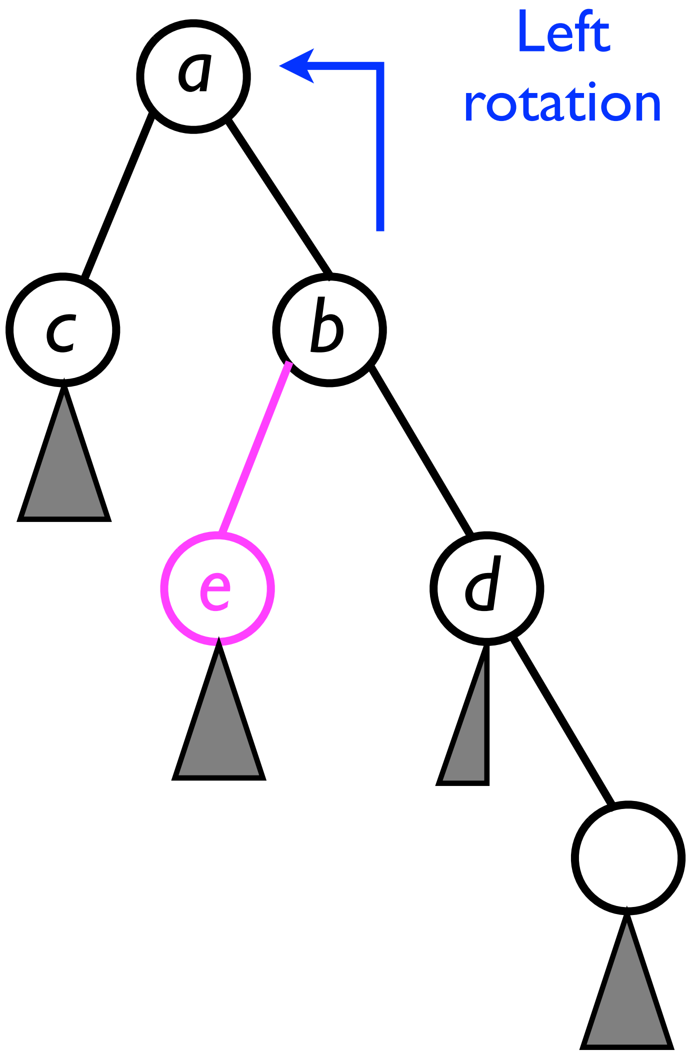
Original tree

Fixing configuration RR

- To fix the imbalance in node *a*, we will perform a *left rotation* of node *b* towards *a*.



Add e as the *left child of a*.

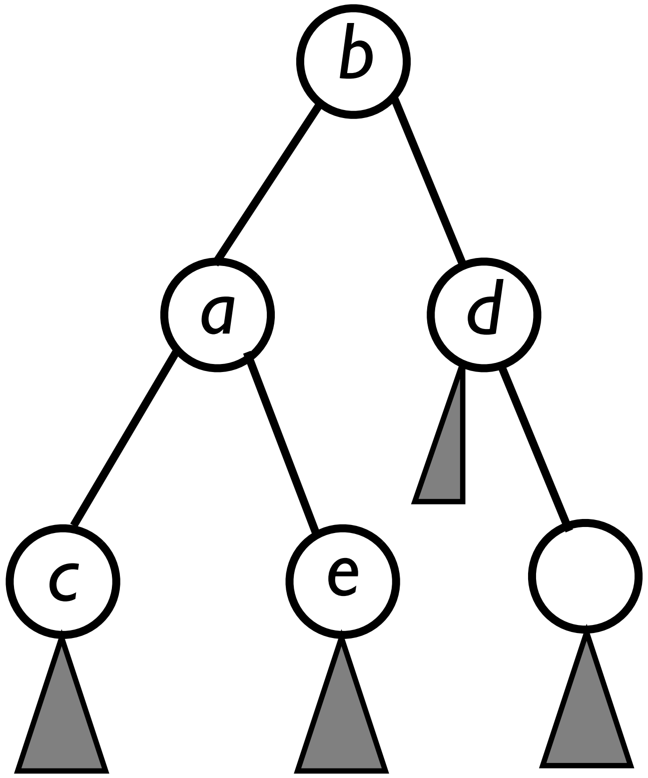


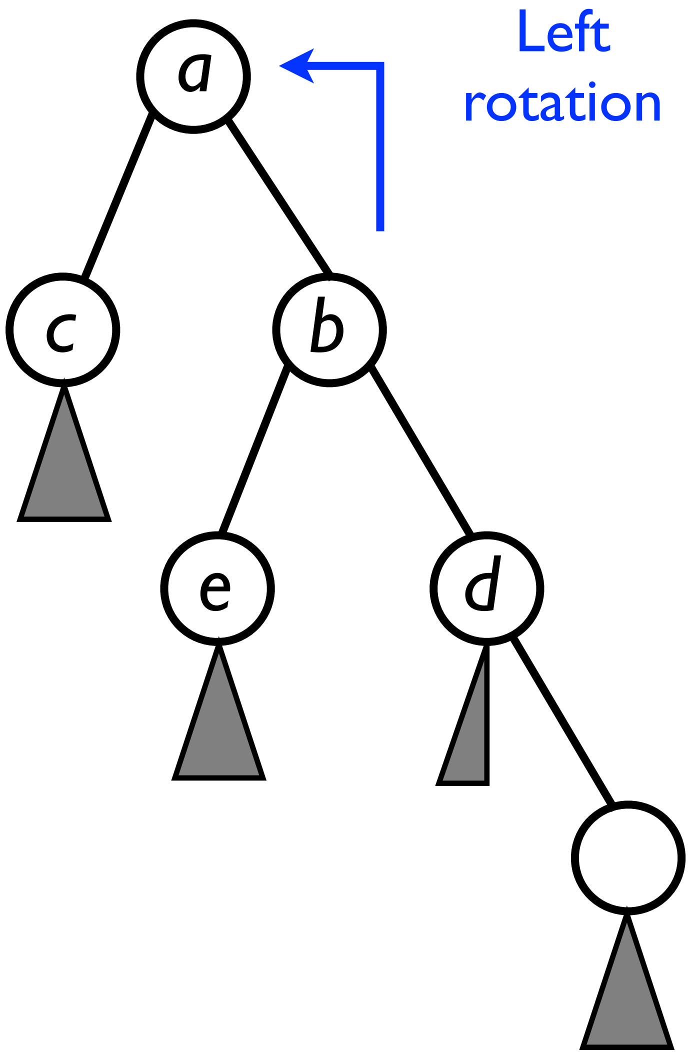
Original tree

Fixing configuration RR

- To fix the imbalance in node *a*, we will perform a *left rotation* of node *b* towards *a*.

Balance = 0





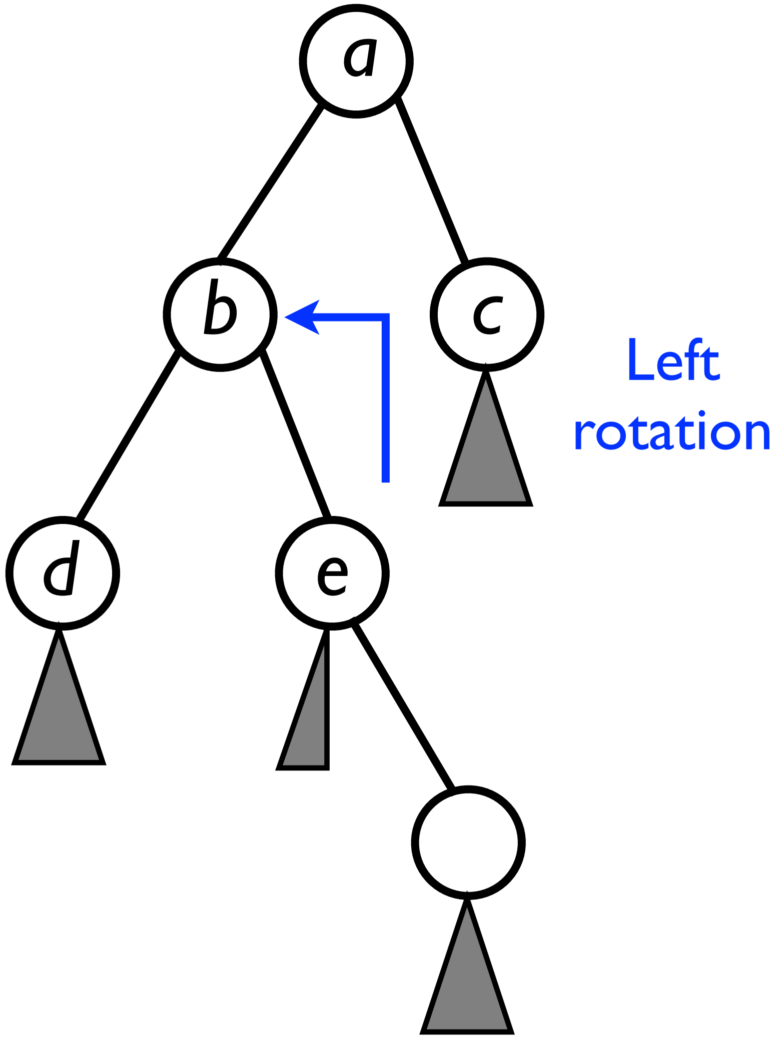
Original tree

Imbalanced node configurations

- Note how LL and RR, as well as LR and RL, are *symmetric* to each other.
  - LL is fixed by *right rotating a*.
  - RR is fixed by *left rotating a*.
- The other two cases -- LR and RL -- can be fixed by *two rotations in succession*.

Fixing configuration LR

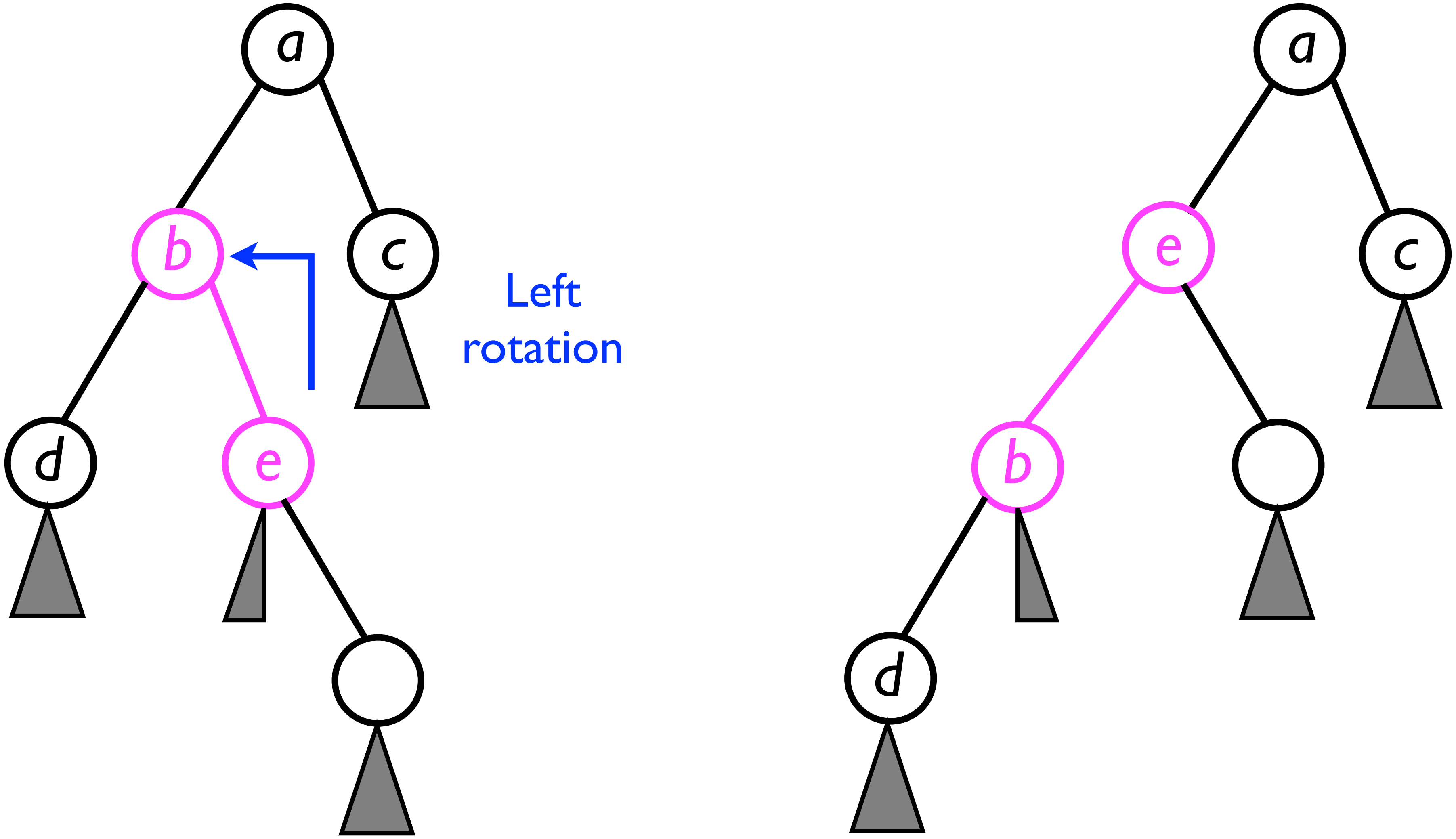
- To fix the imbalance in node *a*, we will *first* perform a *left rotation* of node e towards *b*.



Original tree

Fixing configuration LR

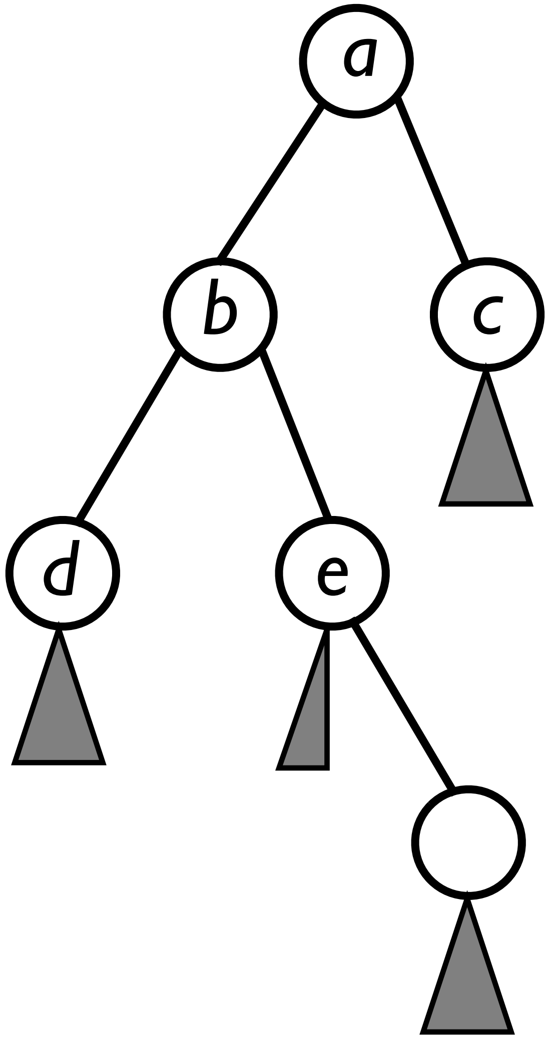
- To fix the imbalance in node *a*, we will *first* perform a *left rotation* of node e towards *b*.



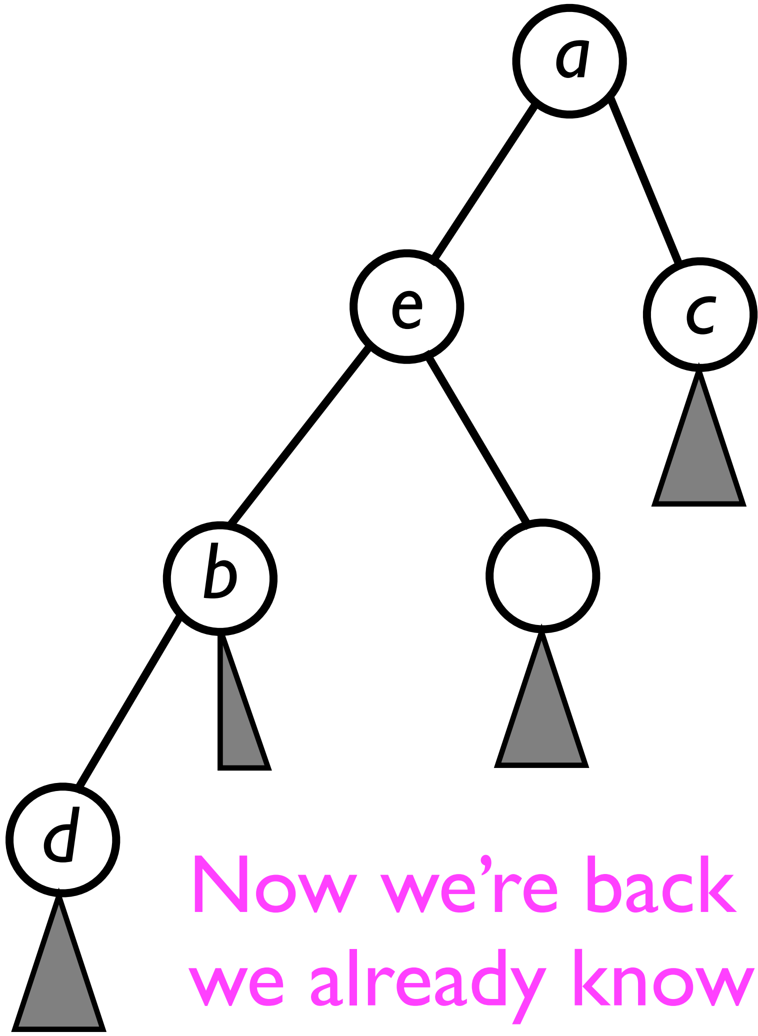
Original tree

Fixing configuration LR

- To fix the imbalance in node *a*, we will *first* perform a *left rotation* of node e towards *b*.



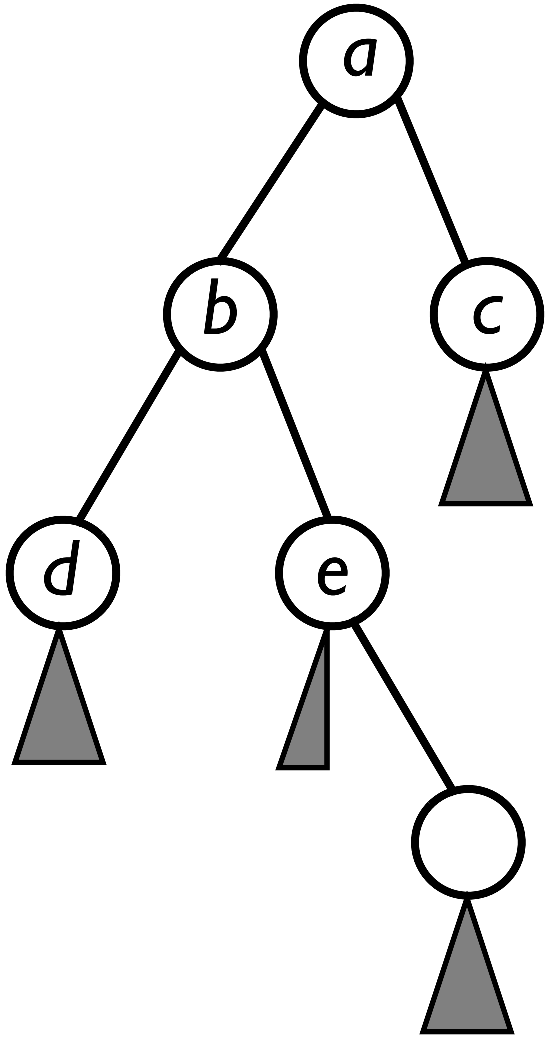
Original tree



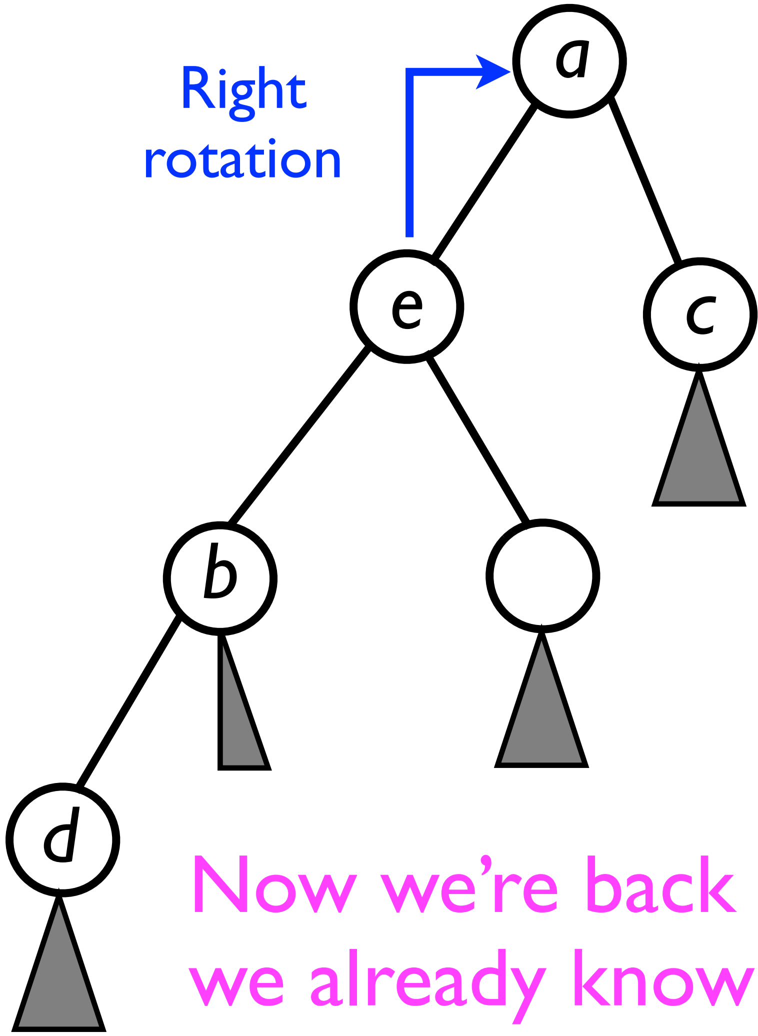
Now we're back to LL -- and we already know how to correct this (by applying a *right rotation of* e *towards* a).

Fixing configuration LR

- Now we perform a *right rotation* of e towards *a*.



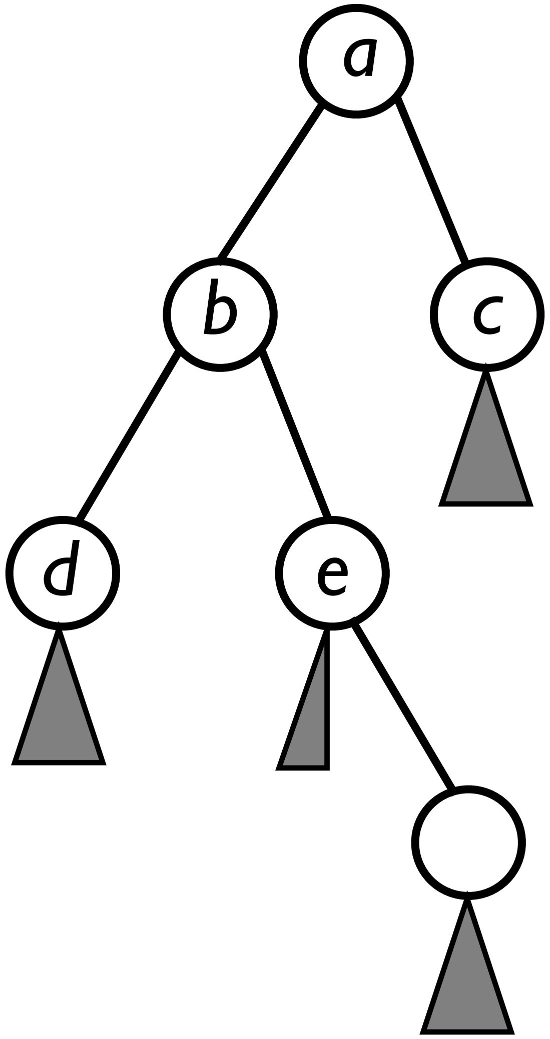
Original tree



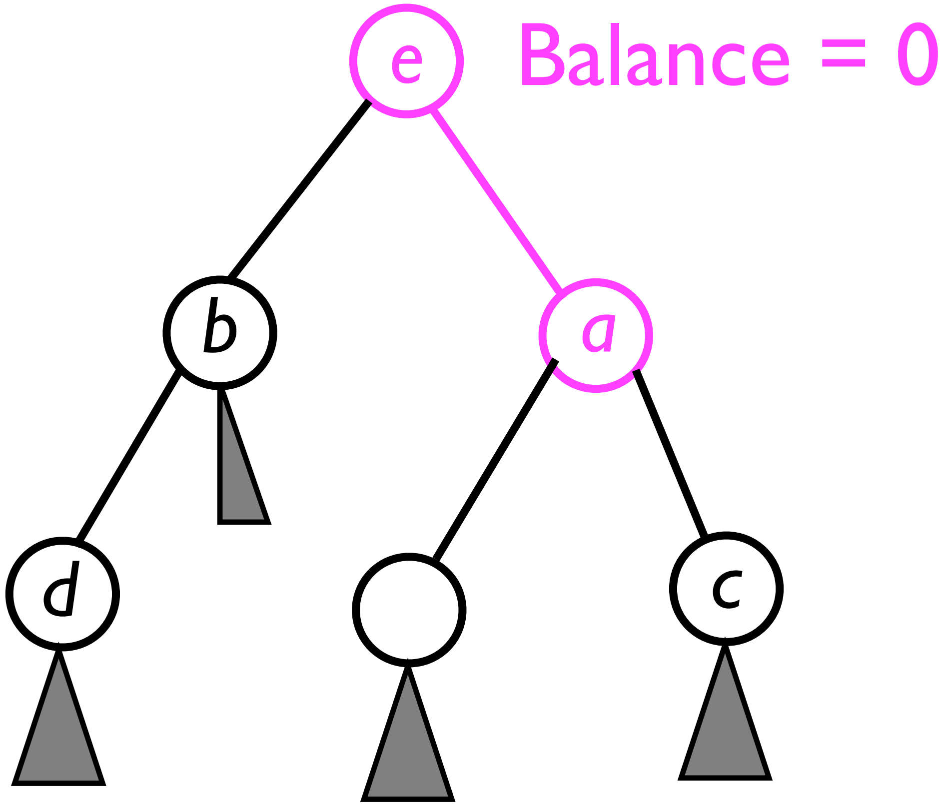
Now we're back to LL -- and we already know how to correct this (by applying a *right rotation of* e *towards* a).

Fixing configuration LR

- Now we perform a *right rotation* of e towards *a*.



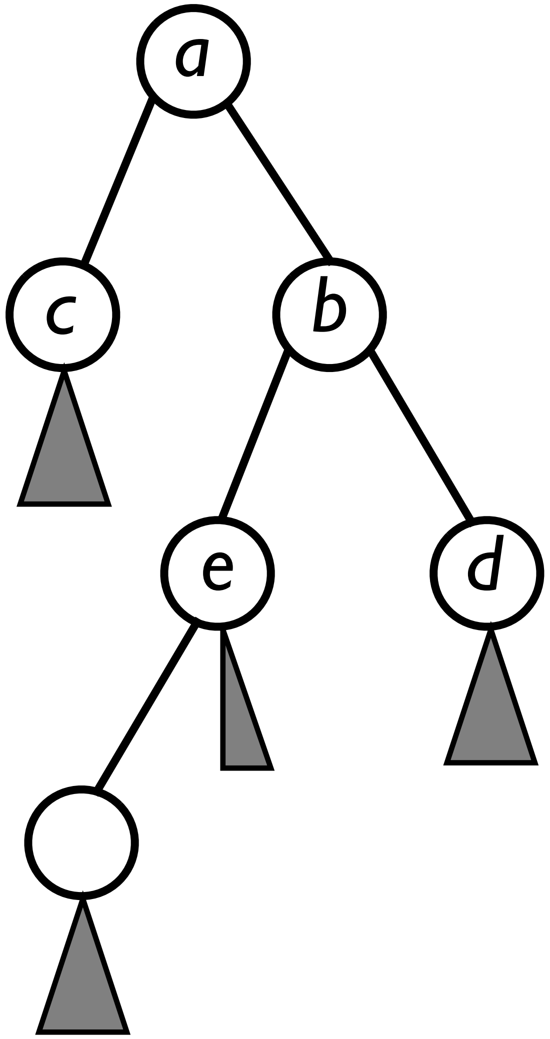
Original tree



Fixing configuration RL

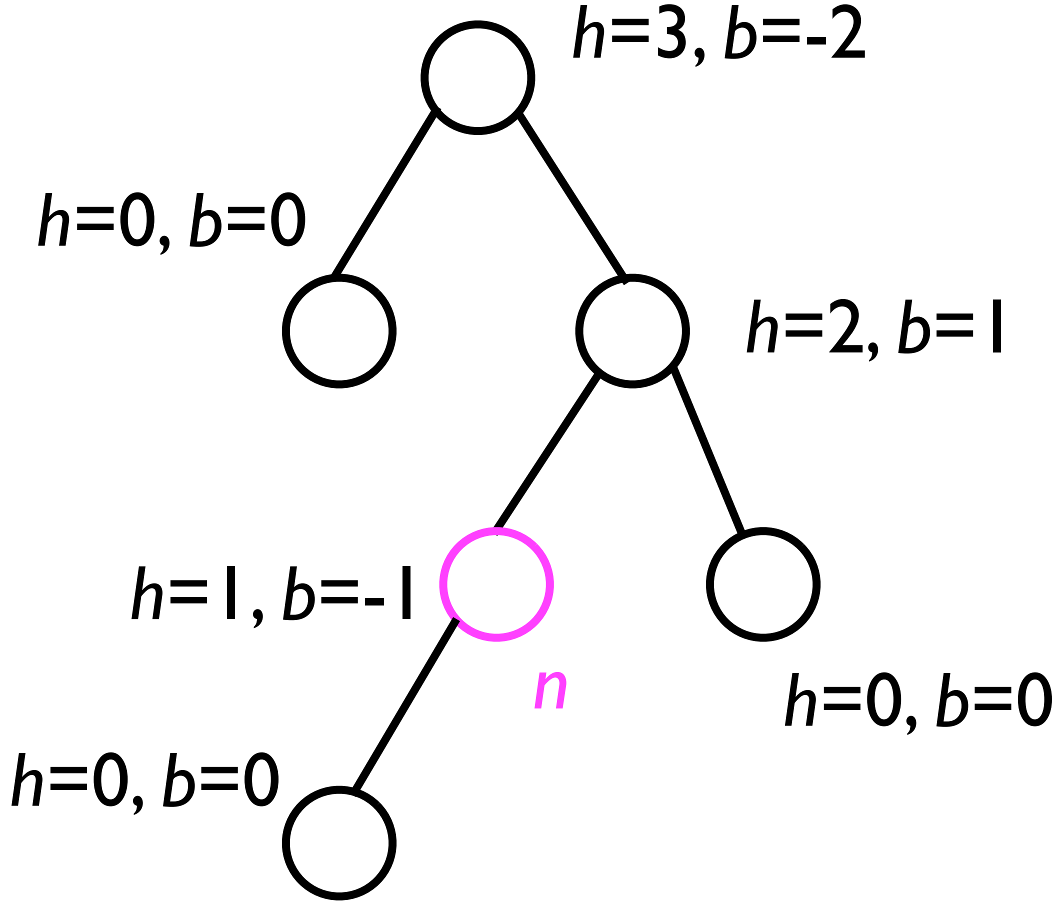
- Fixing configuration RL is exactly symmetric to fixing LR:
  - First apply a *right rotation of* e *towards* b.
    - This returns the configuration to RR.
  - Then apply a *left rotation of* e *towards* a.
  - Left as an "exercise for the reader".

Balance = -2



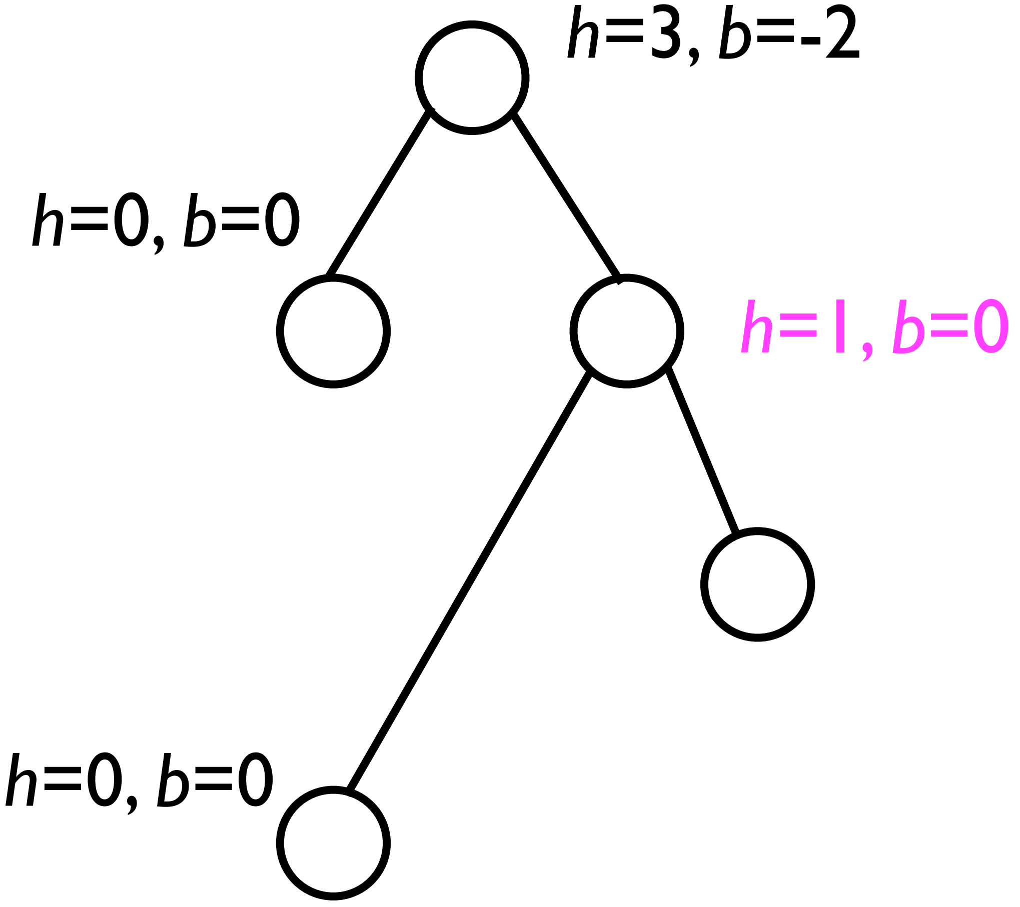
Removing a new node

- When we *remove* a node *n*, we must distinguish the three cases as outlined last lecture:
  - *n* is a leaf node.
  - *n* has only one child.
  - *n* has two children.
- After removing *n*, we must update the height and balance of all nodes between *n* and the root.



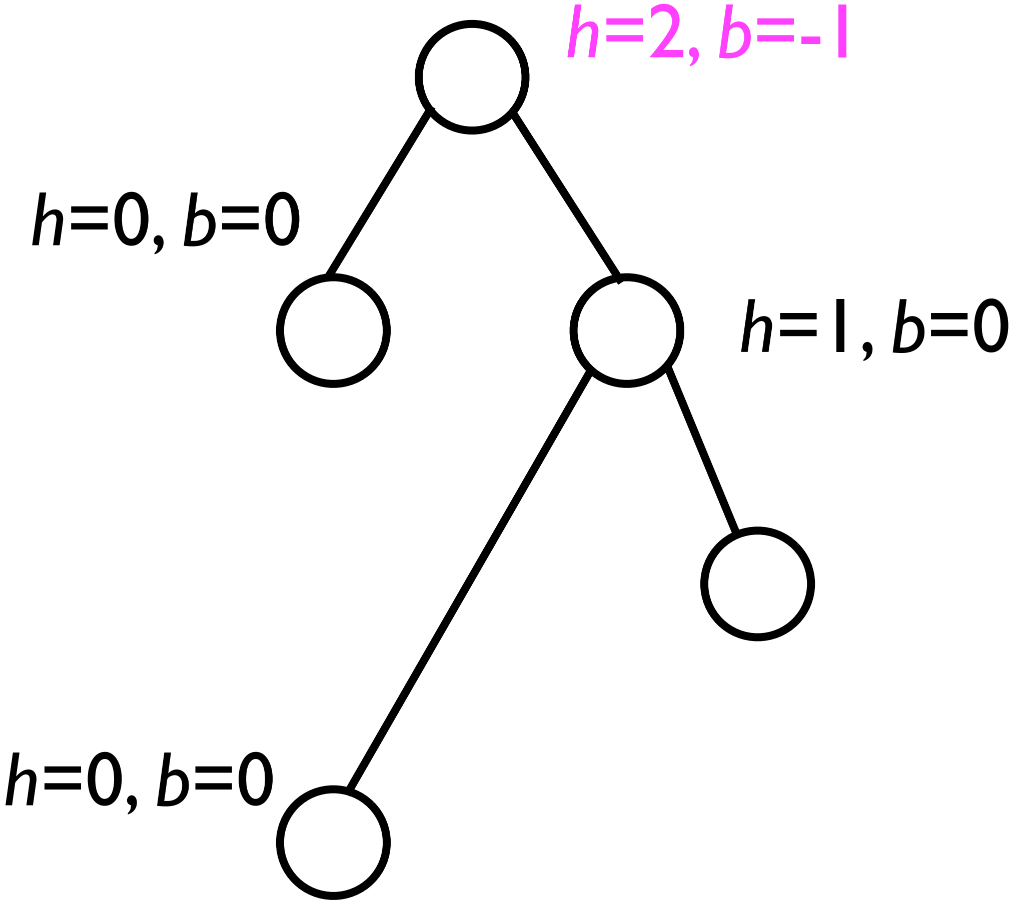
Removing a new node

- When we *remove* a node *n*, we must distinguish the three cases as outlined last lecture:
    - *n* is a leaf node.
    - *n* has only one child.
    - *n* has two children.
- After removing *n*, we must update the height and balance of all nodes between *n* and the root.



Removing a new node

- When we *remove* a node *n*, we must distinguish the three cases as outlined last lecture:
  - *n* is a leaf node.
  - *n* has only one child.
  - *n* has two children.
- After removing *n*, we must update the height and balance of all nodes between *n* and the root.
  - Might require an AVL rotation.



AVL trees

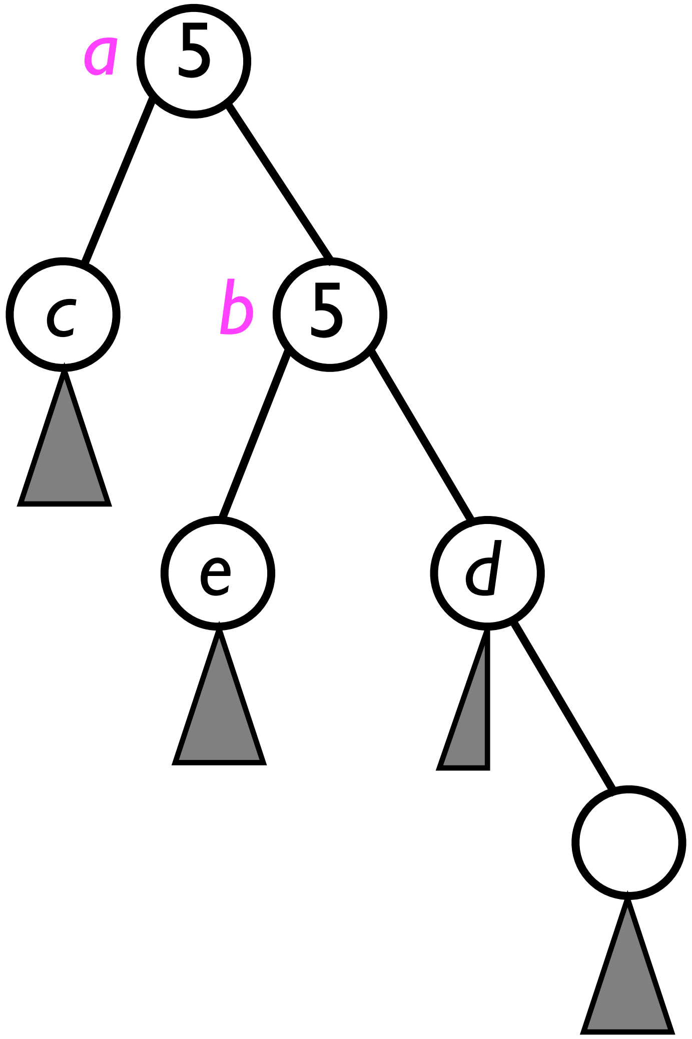
- Through storing the height and balance of each node and implementing AVL rotations as necessary, we can ensure that the BST is never “more imbalanced” than +1 or -1.
  - This yields a BST for which $h=O(\log n)$ in the *worst case*, not just the average case.
  - The AVL rotations themselves take O(1) time.
    - Each rotation takes a constant number of “node switches”.
  - Hence, with AVL trees, the fundamental tree operations **`add`**, **`find`**, and **`remove`** all operate in $O(\log n)$ time worst-case.

Duplicate keys

- In contrast to “regular” BSTs, duplicate keys are not permitted in AVL trees.
  - With duplicate keys, rotating sub-trees could cause the tree to violate the BST ordering property.

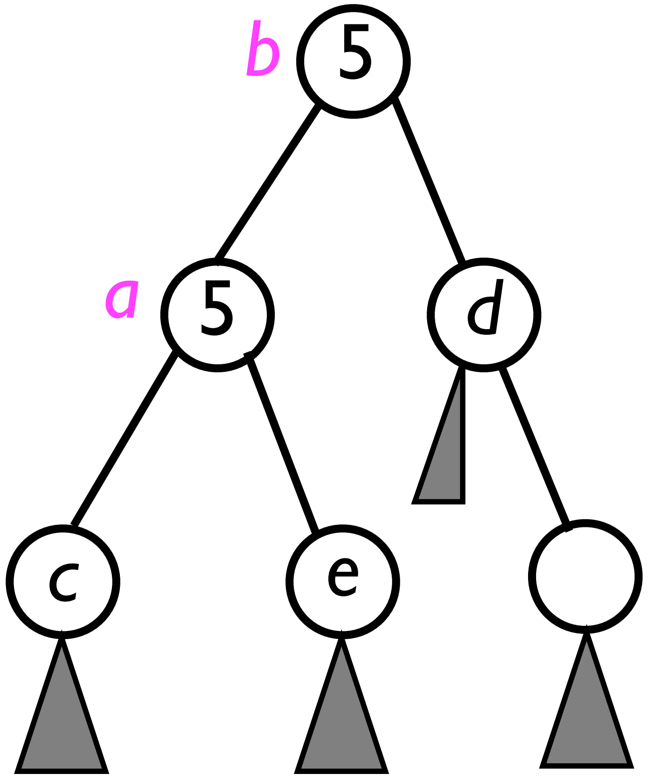
Duplicate keys

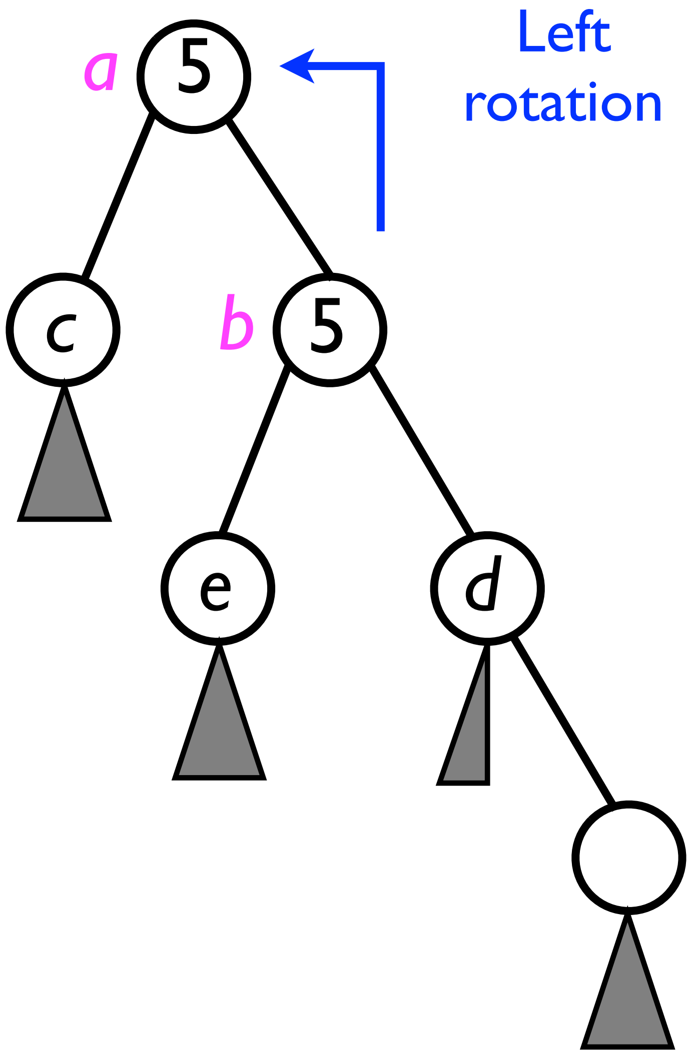
- However, a problem arises when we start rotating nodes in a sub-tree:
  - Suppose *a* and *b* have the same key (e.g., 5).



Duplicate keys

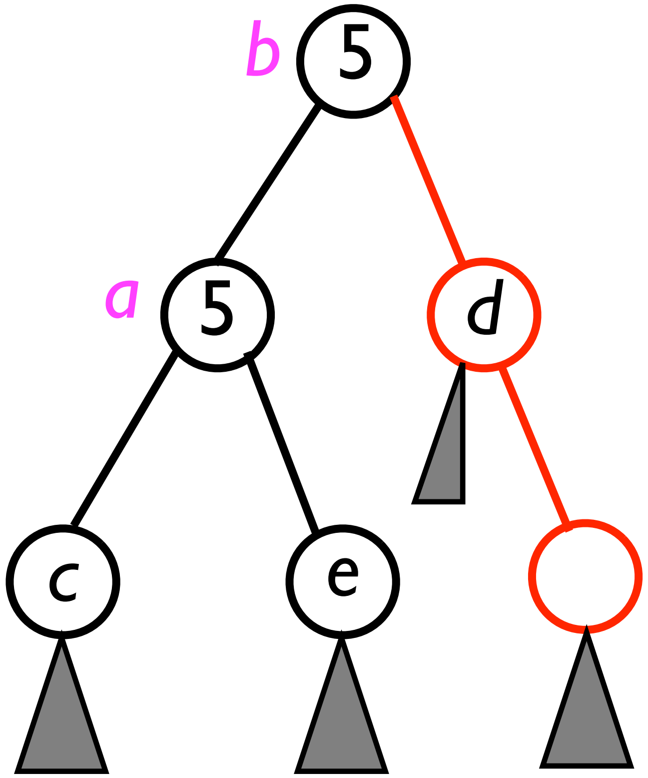
- However, a problem arises when we start rotating nodes in a sub-tree:
  - Suppose *a* and *b* have the same key (e.g., 5).
  - Suppose we then *left-rotate b* towards *a*.





Duplicate keys

- Now, suppose we want to find node *a* starting at the root (node *b*).
  - We will descend the *wrong sub-tree* of *b*.
  - We will never find *a*.



Duplicate keys

- One solution is to:
  - Disallow multiple *nodes* with the same key.
  - Whenever we add an element with the *same* key, we *append* that new element to that node's *list of objects*.