CSE 12:

Basic data structures and object-oriented design

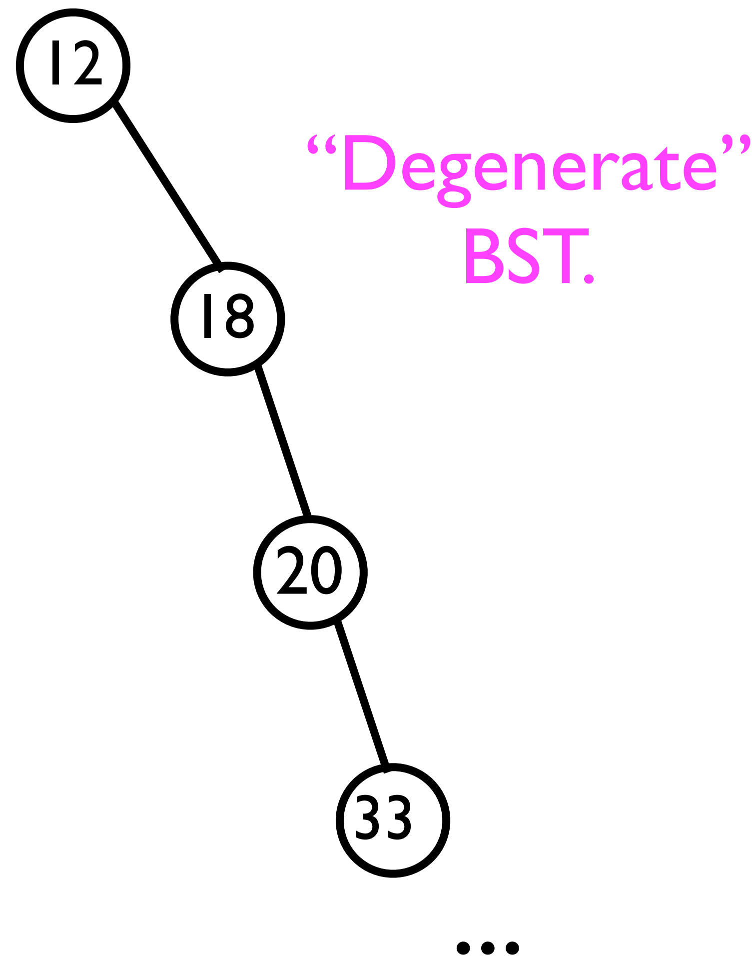
Jacob Whitehill
jake@mplab.ucsd.edu

Lecture Thirteen
23 Aug 2011

More on BSTs.

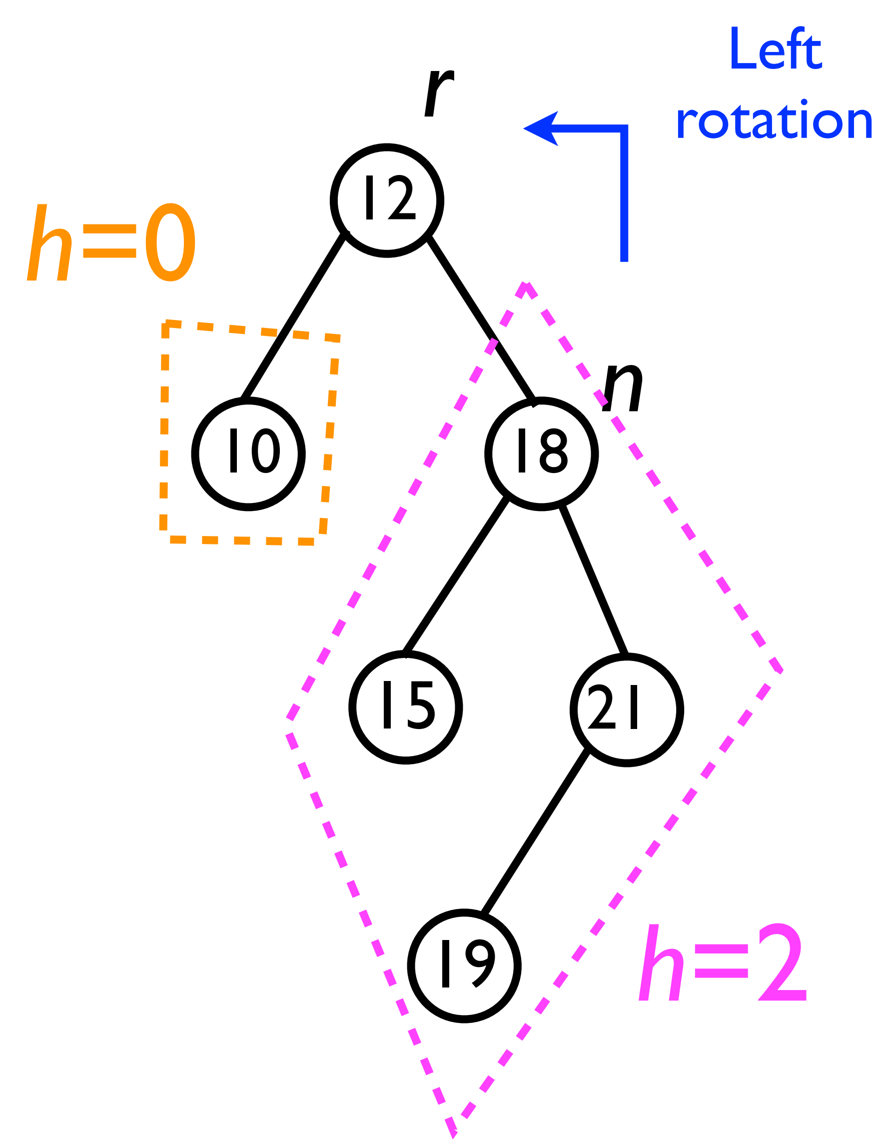
Maintaining balance

- 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.



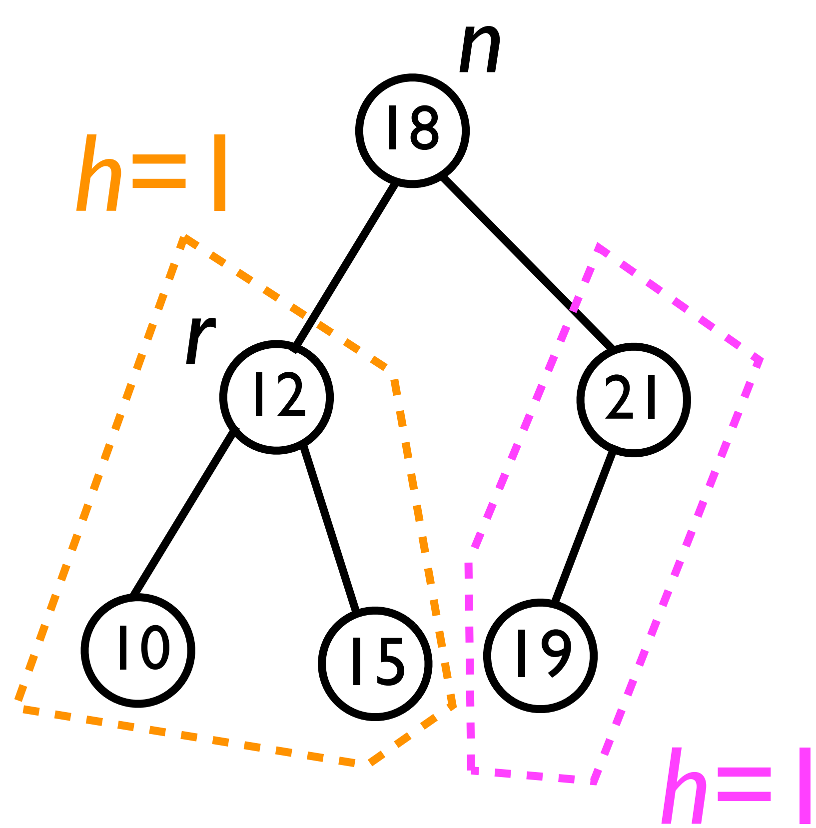
Maintaining balance

- 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*).

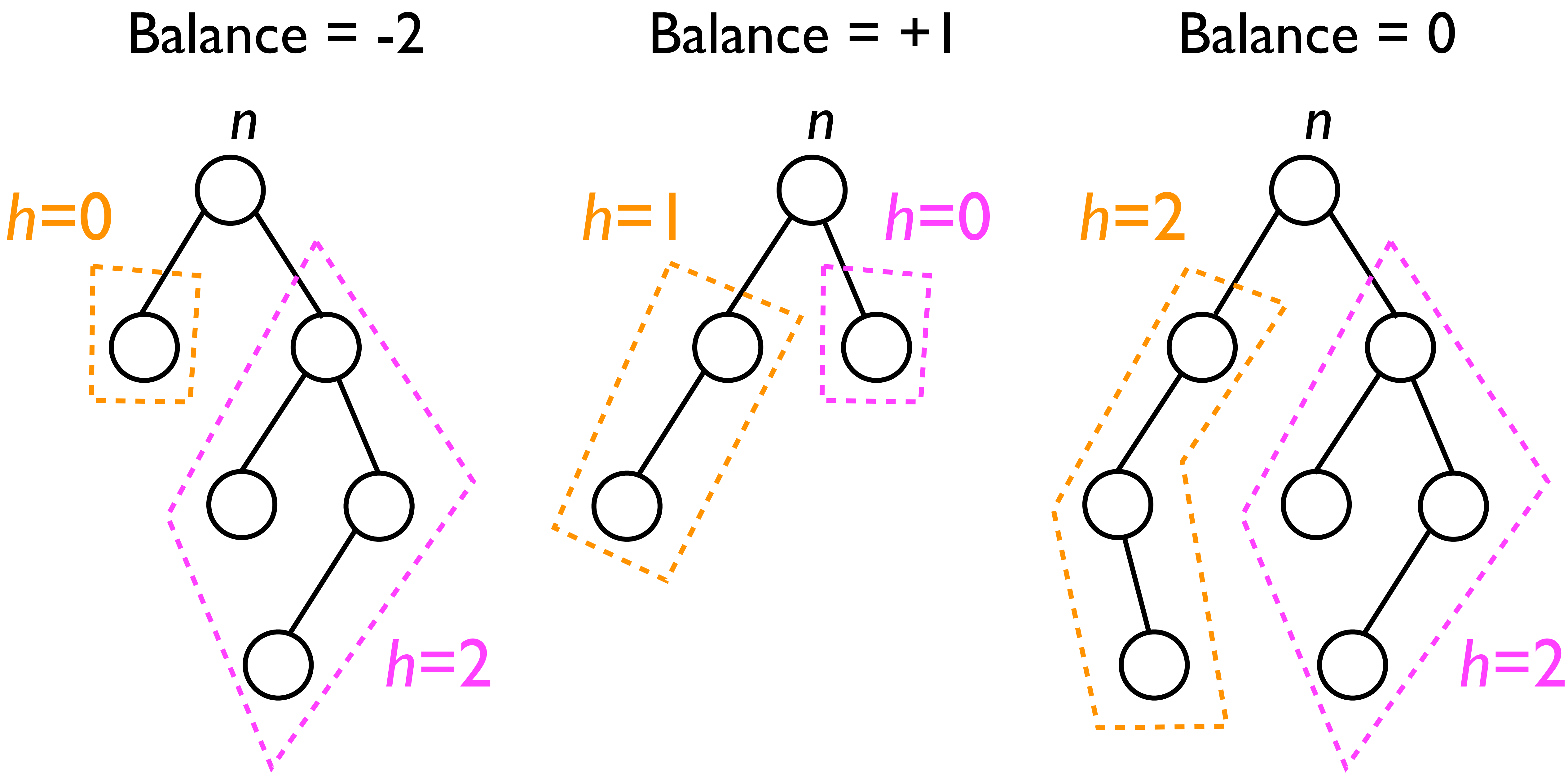
Maintaining balance

- 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).

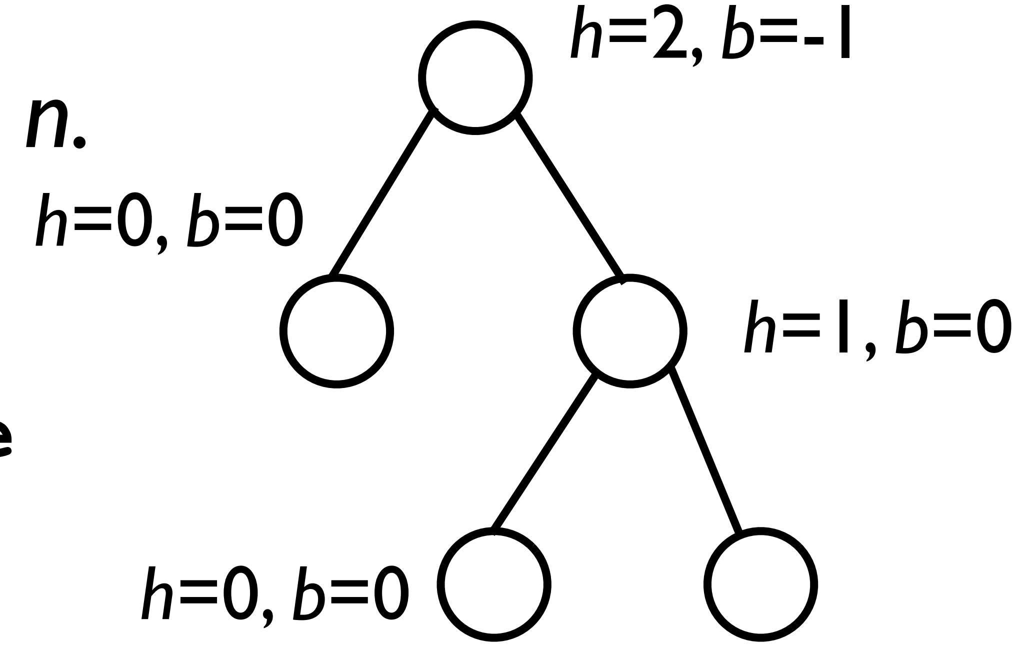
Height and balance



Height and balance

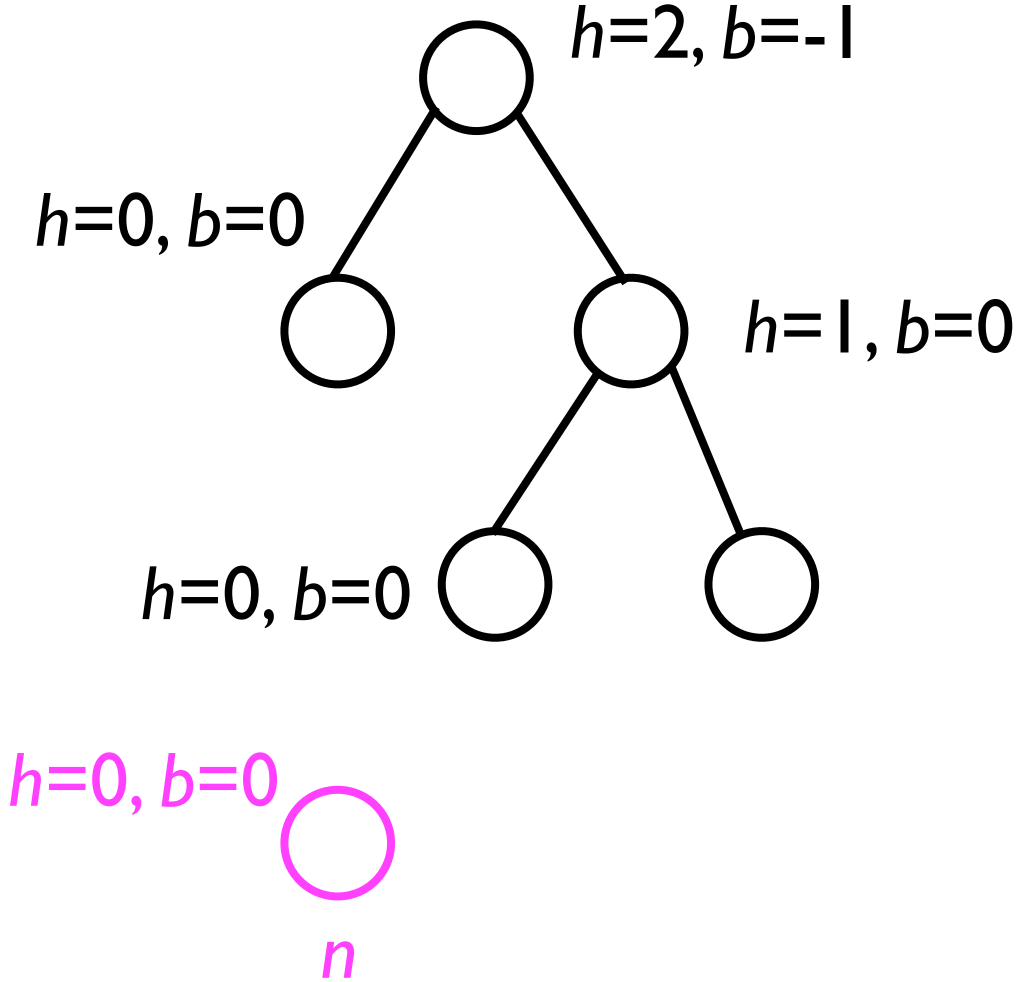
- AVL trees require that each node *n* record its *balance* as well as the *height* of the sub-tree rooted at *n*.
    - We can store these as extra instance variables in the `Node` class:

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



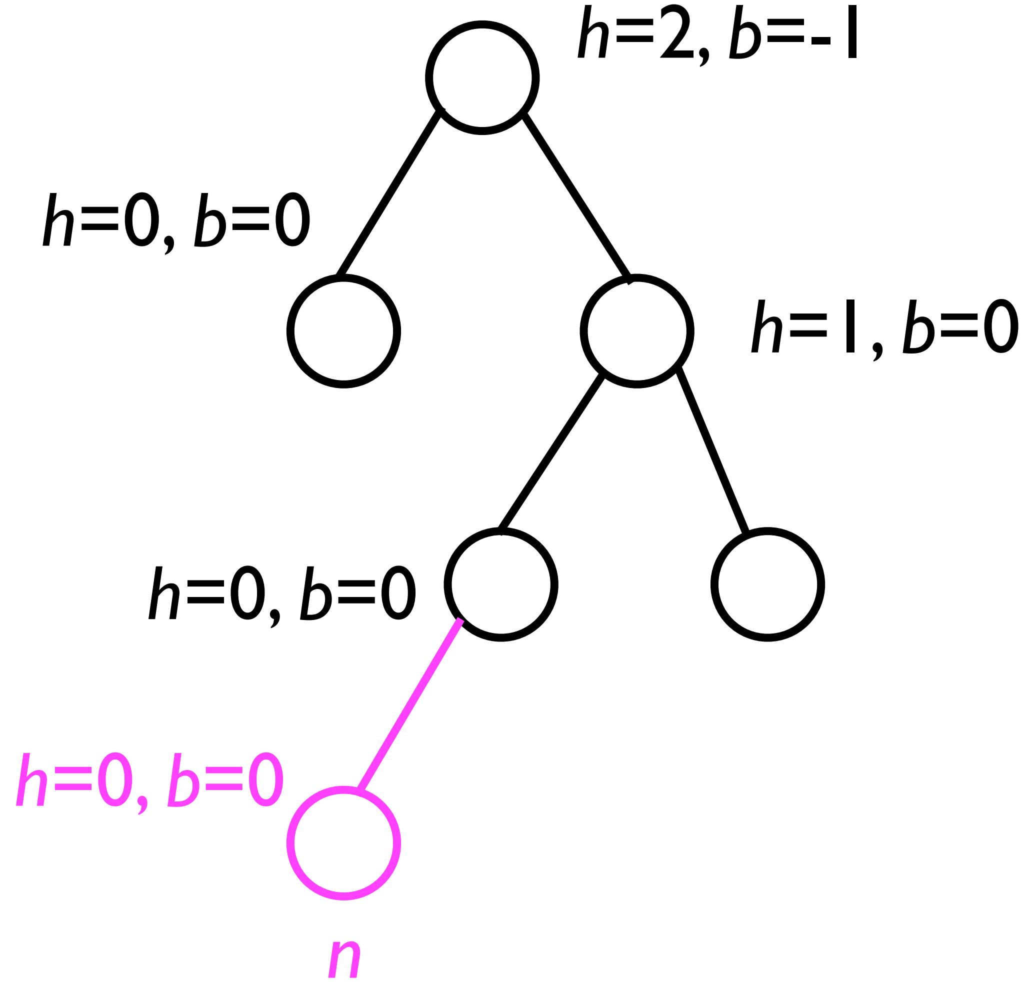
Adding a new node

- 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.



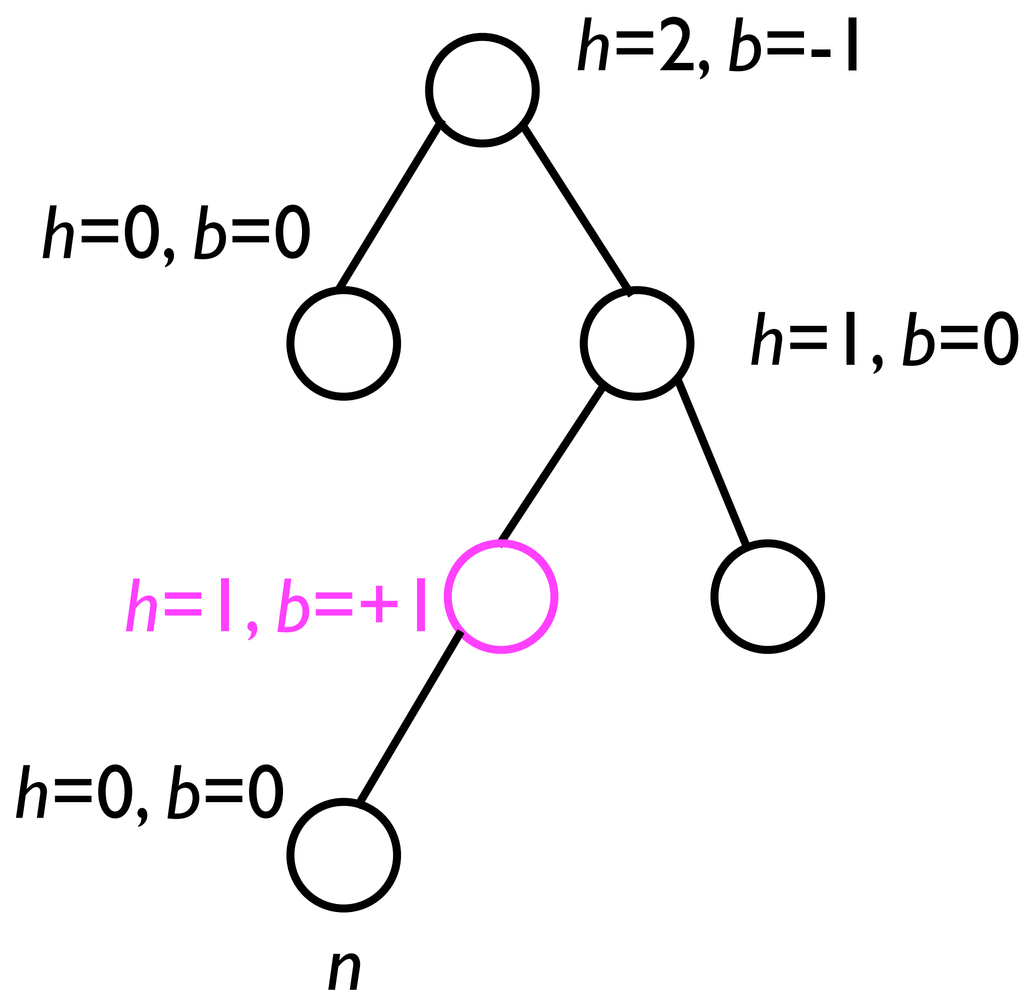
Adding a new node

- 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.



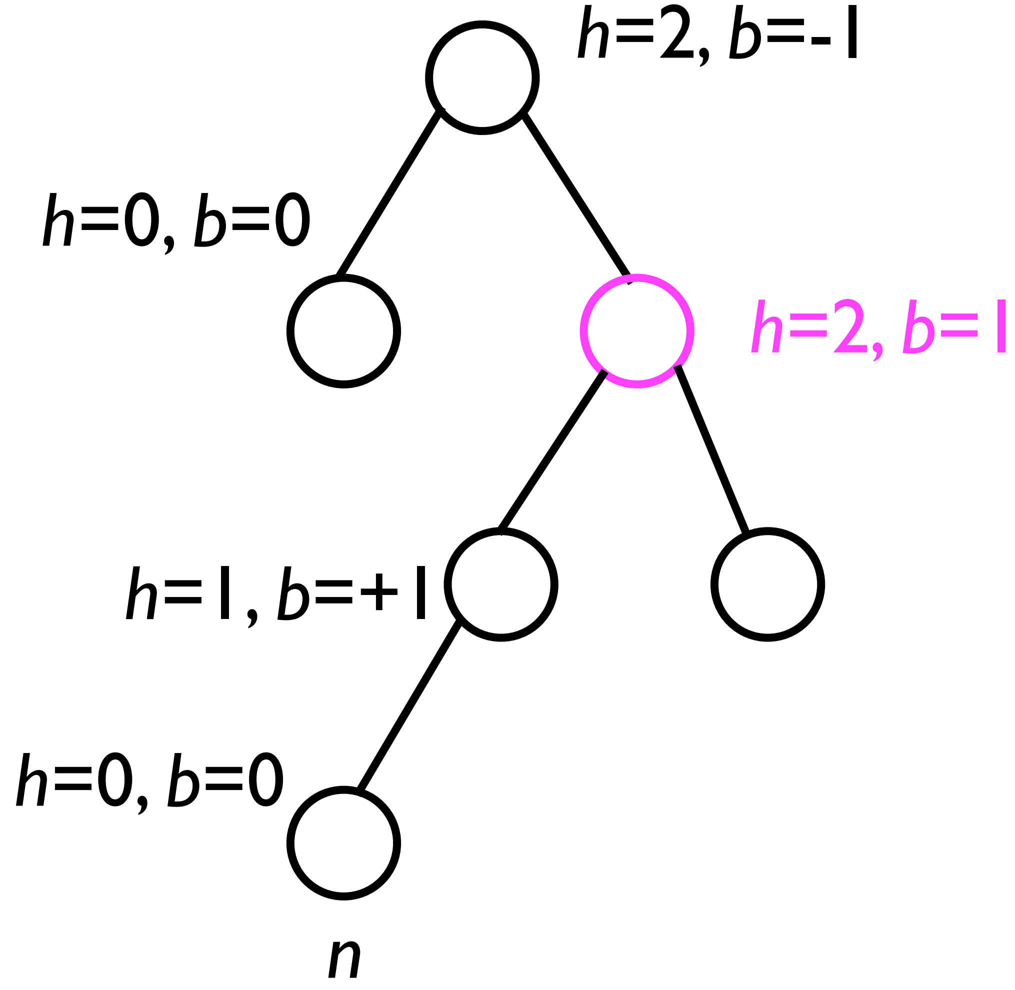
Adding a new node

- 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.



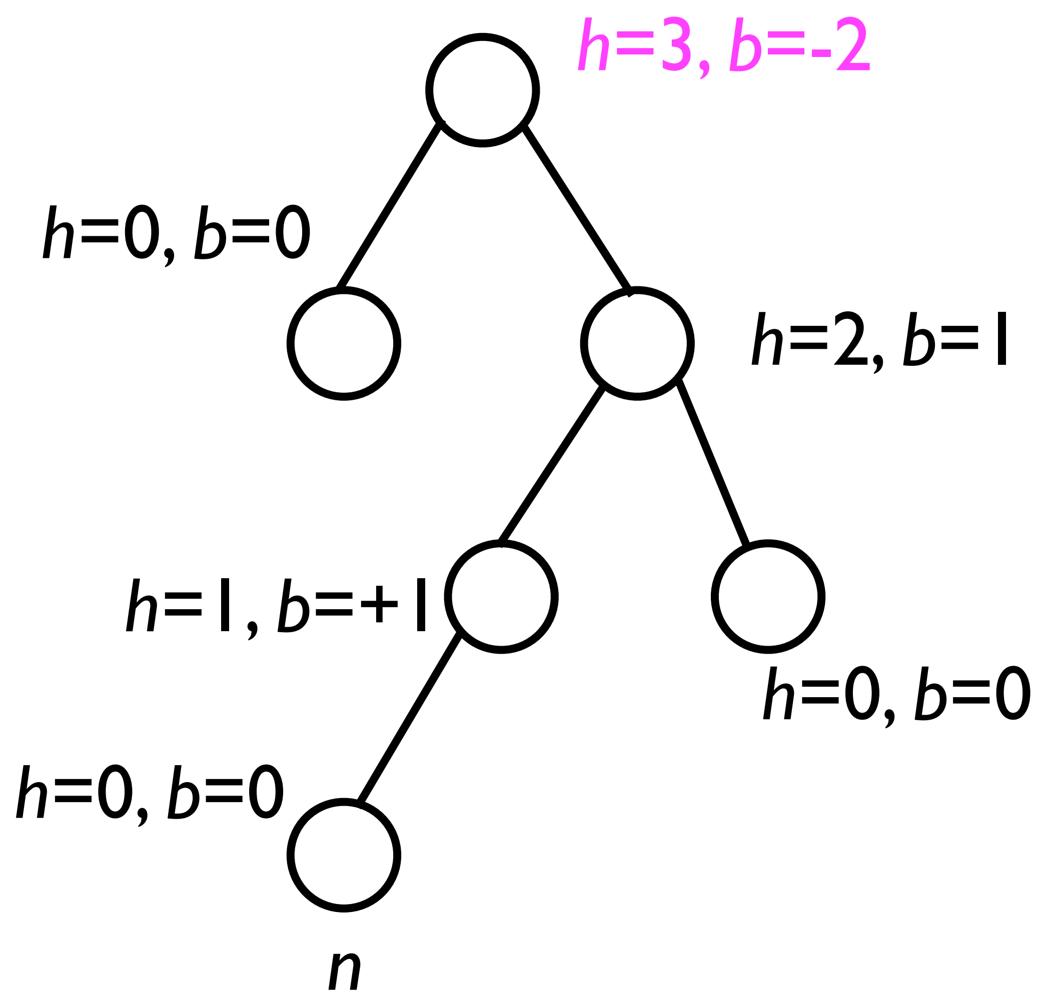
Adding a new node

- 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.



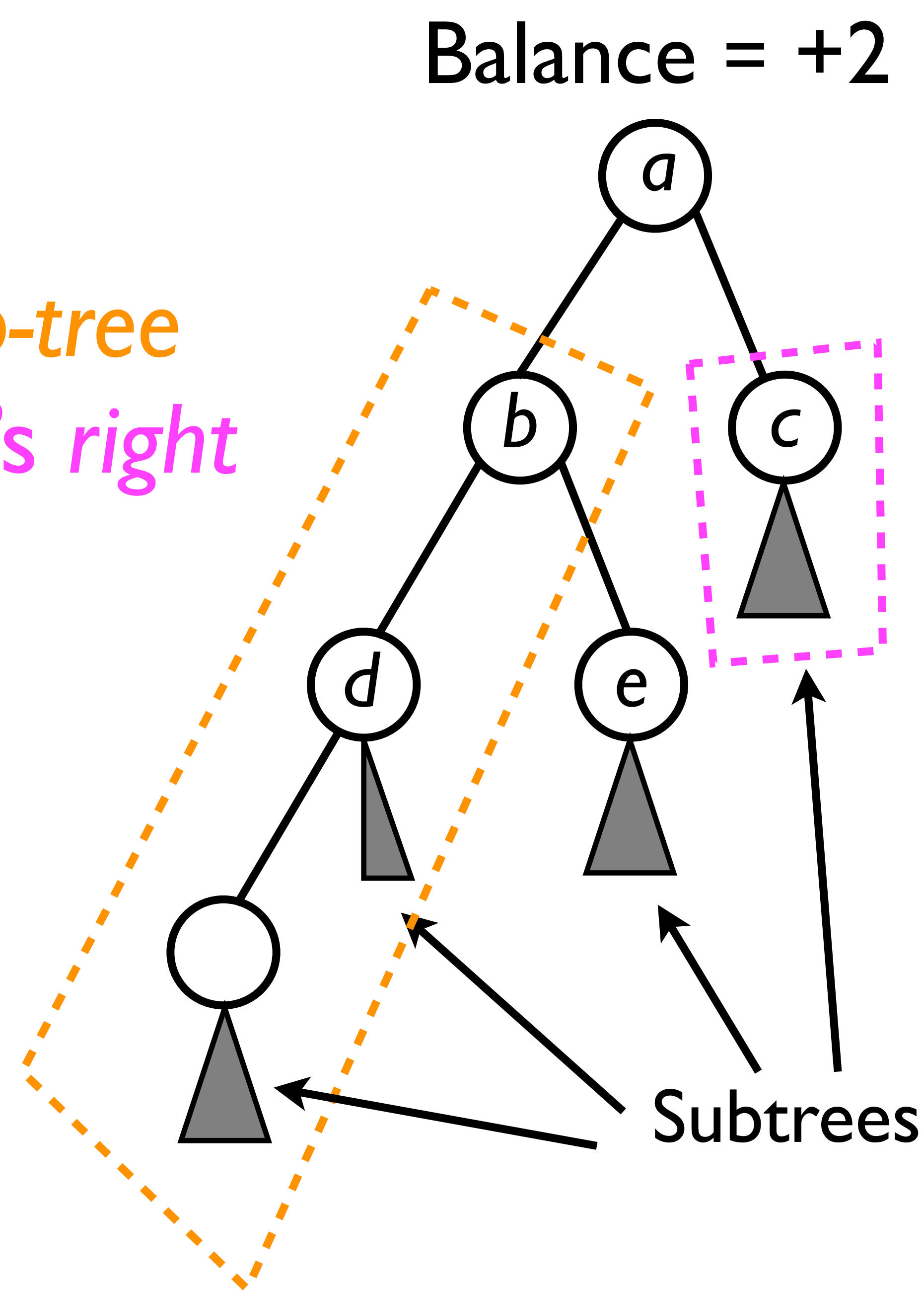
Correcting imbalances

- 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*.

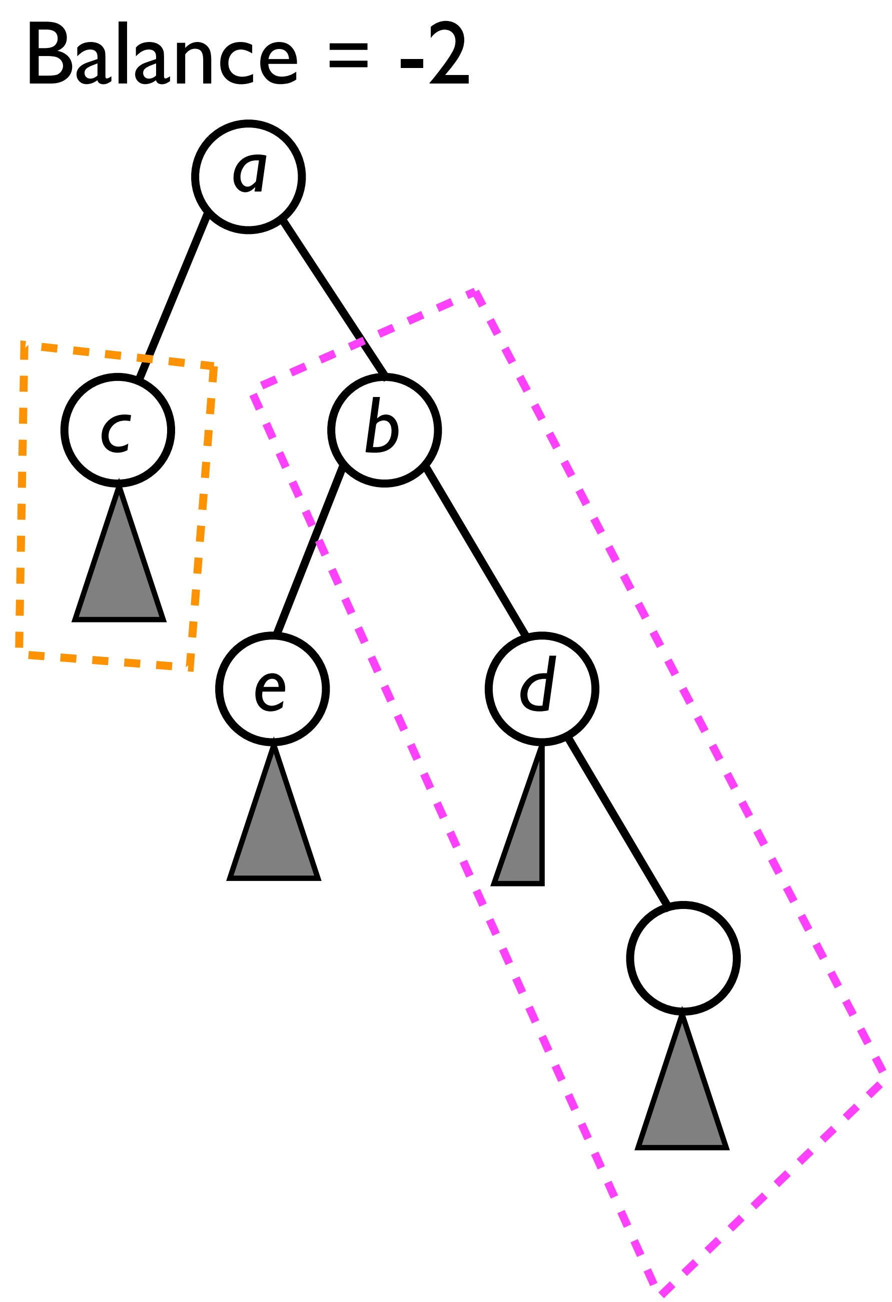
This case is called LL.



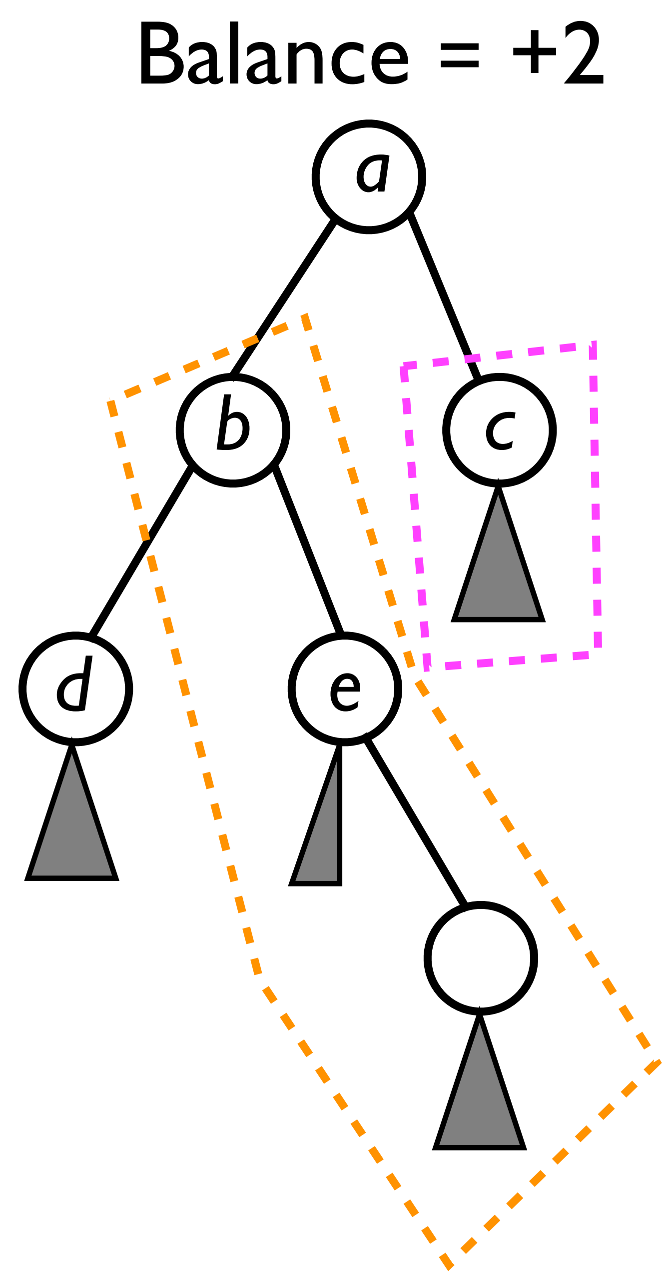
Imbalanced node configurations

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



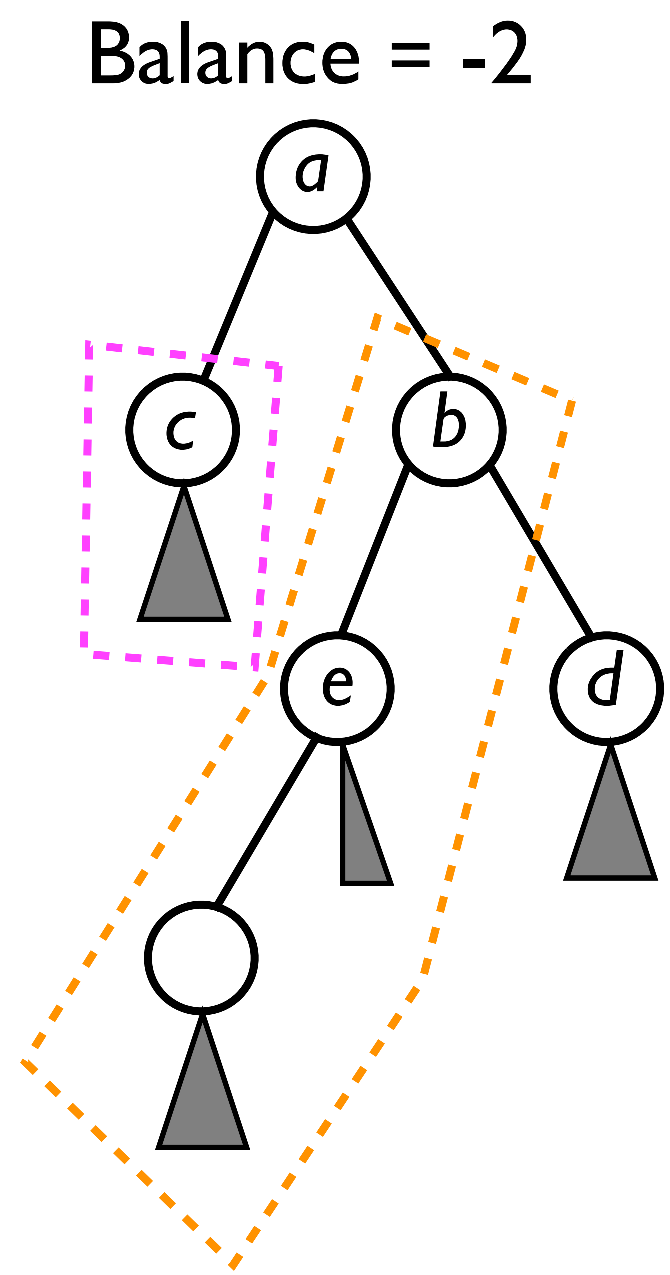
The *Left child's Right sub-tree of a* is 2 higher than *a's right sub-tree.*

This case is called LR.

Imbalanced node configurations

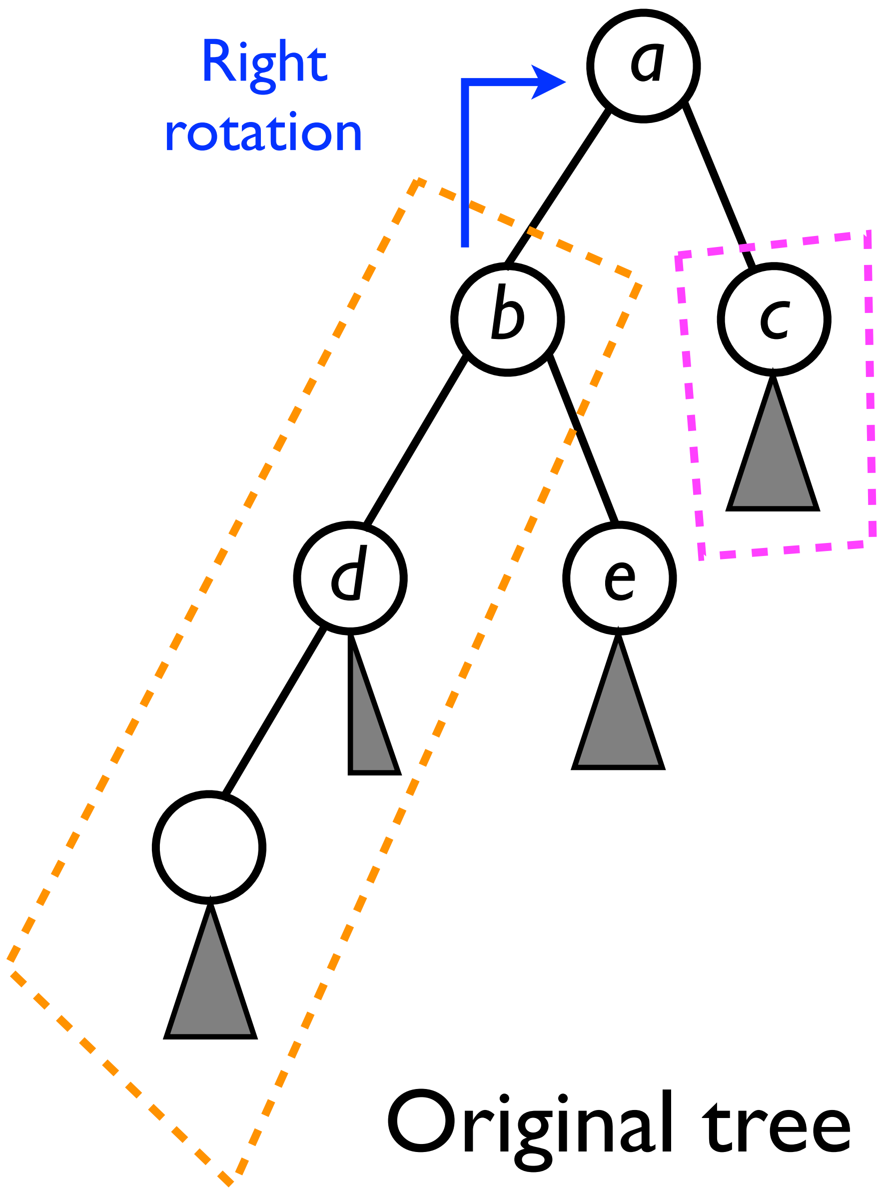
The *Right child's Left sub-tree of a* is 2 higher than *a's left sub-tree*.

This case is called RL.



Fixing configuration LL

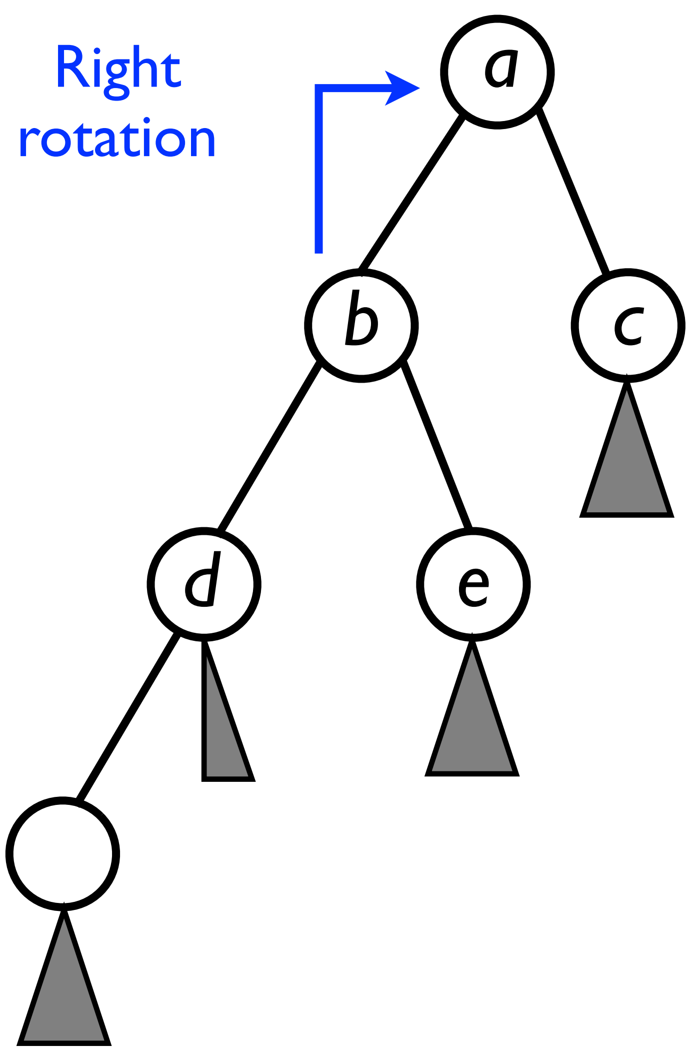
- To fix the imbalance in node *a*, we will perform a *right rotation* of node *b* towards *a*.



Original tree

Fixing configuration LL

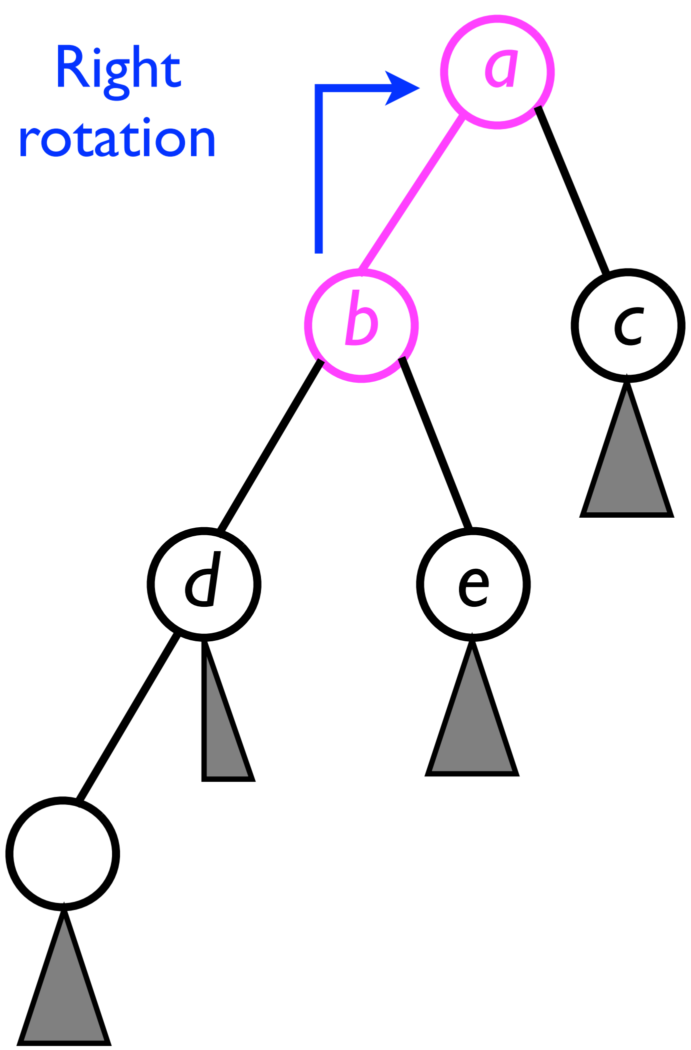
- To fix the imbalance in node *a*, we will perform a *right rotation* of node *b* towards *a*.



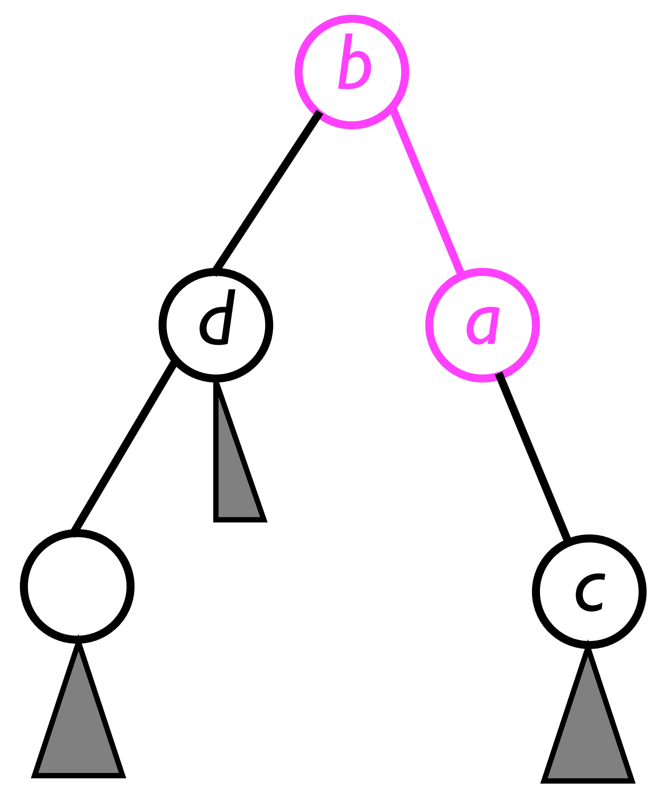
Original tree

Fixing configuration LL

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



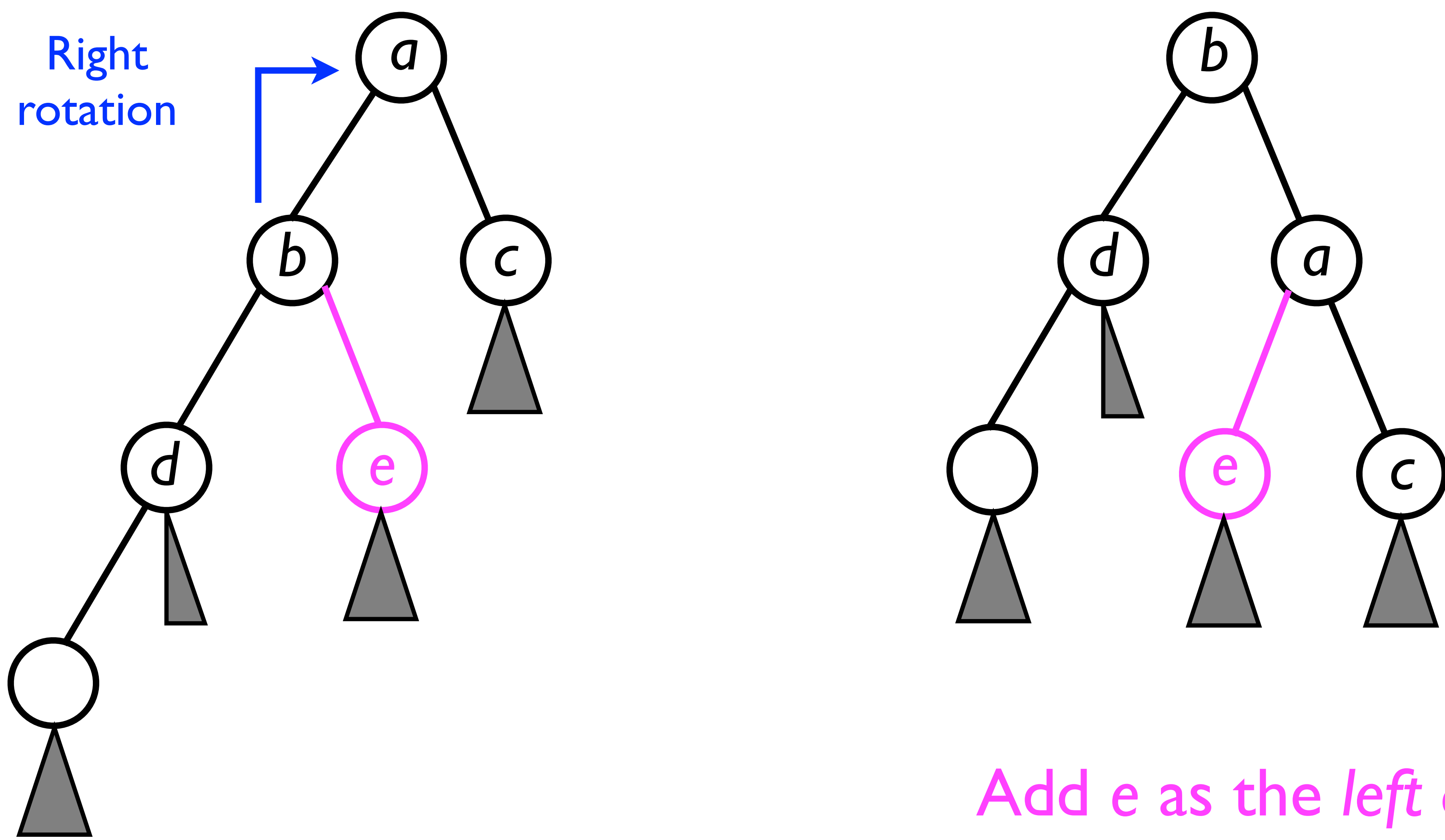
Original tree



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*.

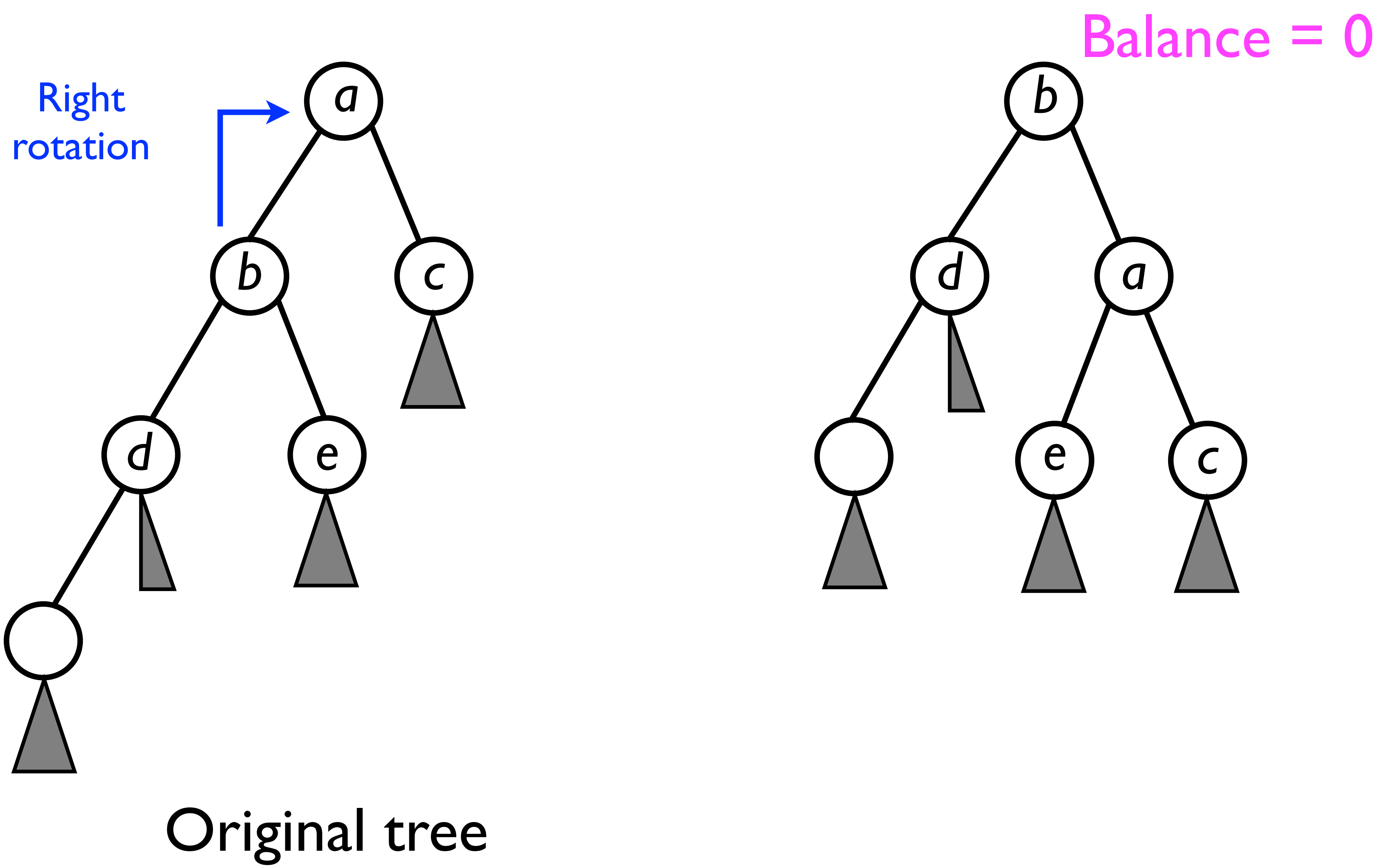


Original tree

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

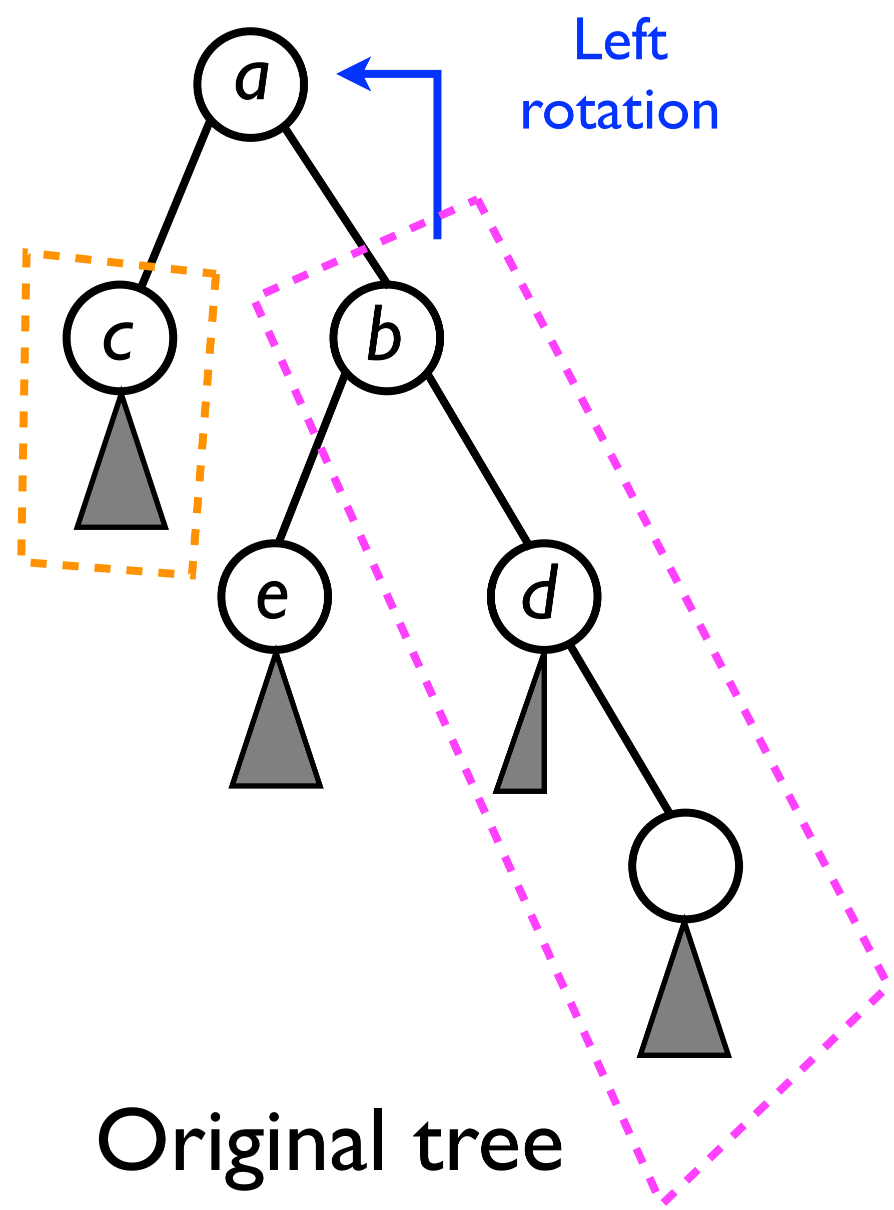
Fixing configuration LL

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



Fixing configuration RR

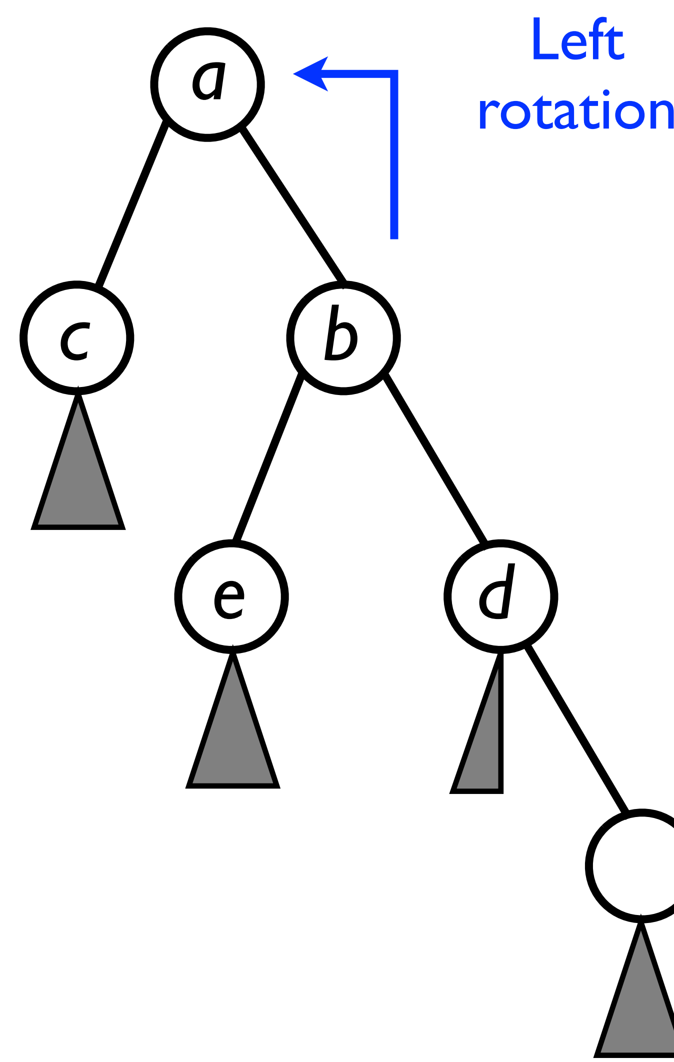
- To fix the imbalance in node *a*, we will perform a *left rotation* of node *b* towards *a*.



Original tree

Fixing configuration RR

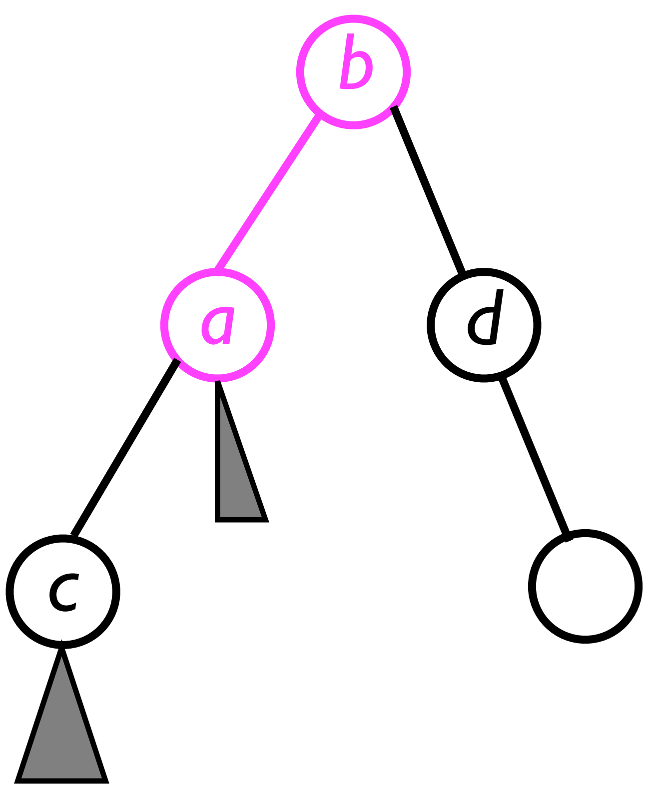
- To fix the imbalance in node *a*, we will perform a *left rotation* of node *b* towards *a*.



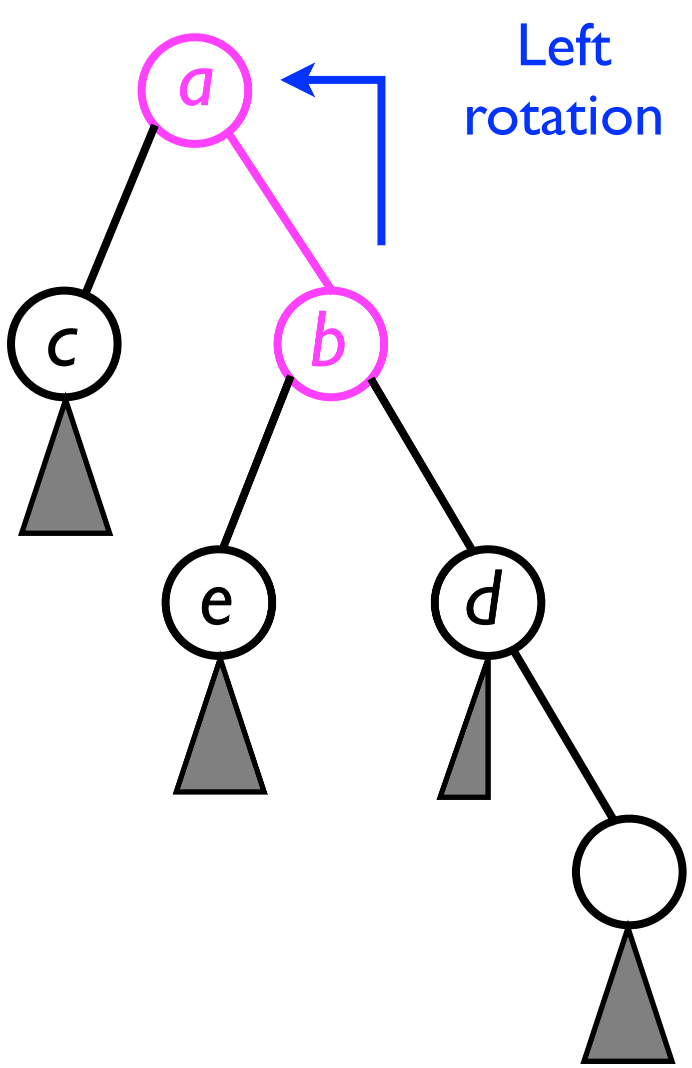
Original tree

Fixing configuration RR

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



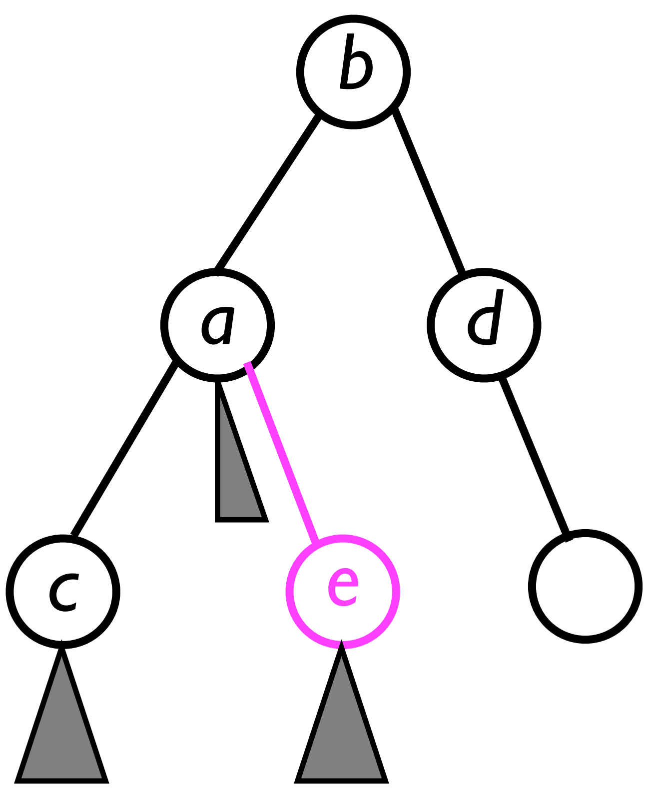
Make *a* the *right child* of b, and make *b* the new root of the sub-tree.



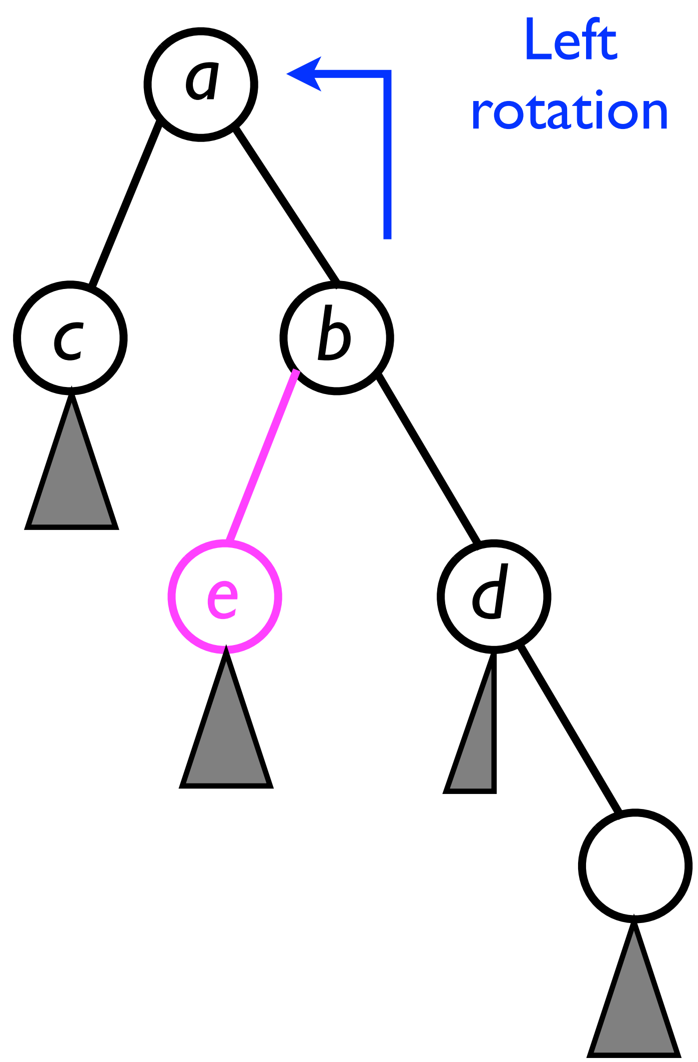
Original tree

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*.

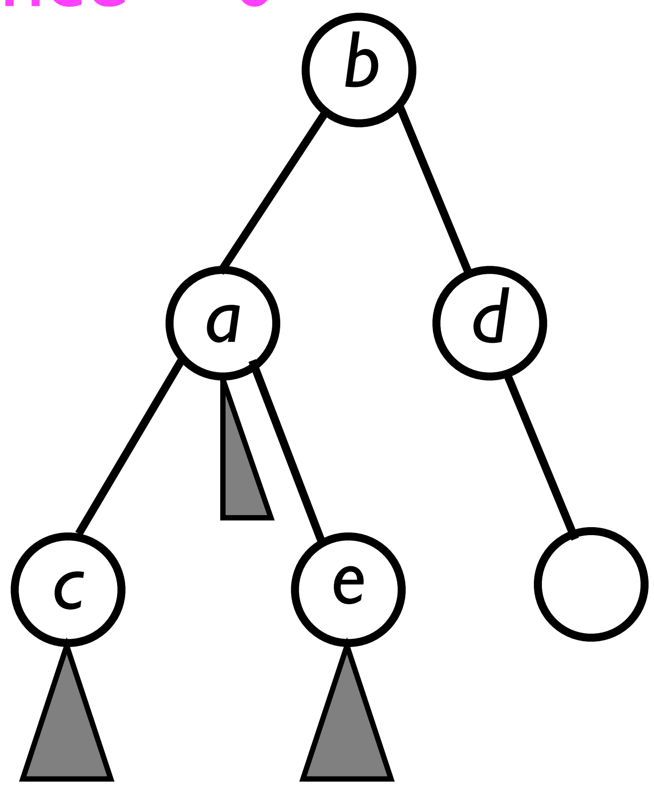


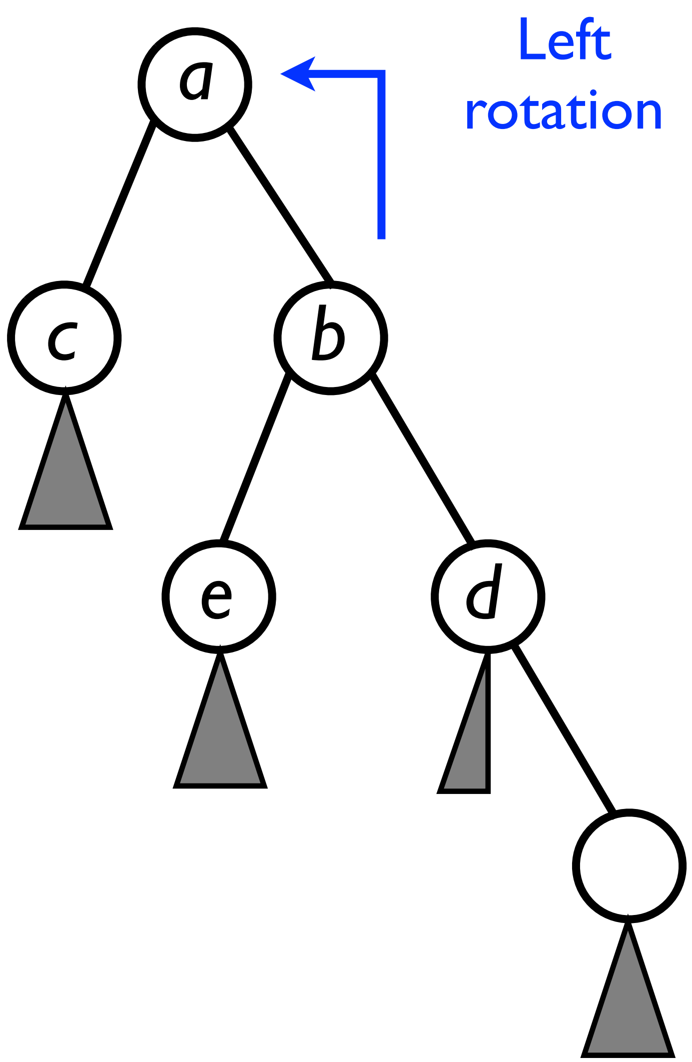
Original tree

Fixing configuration RR

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

Balance = 0





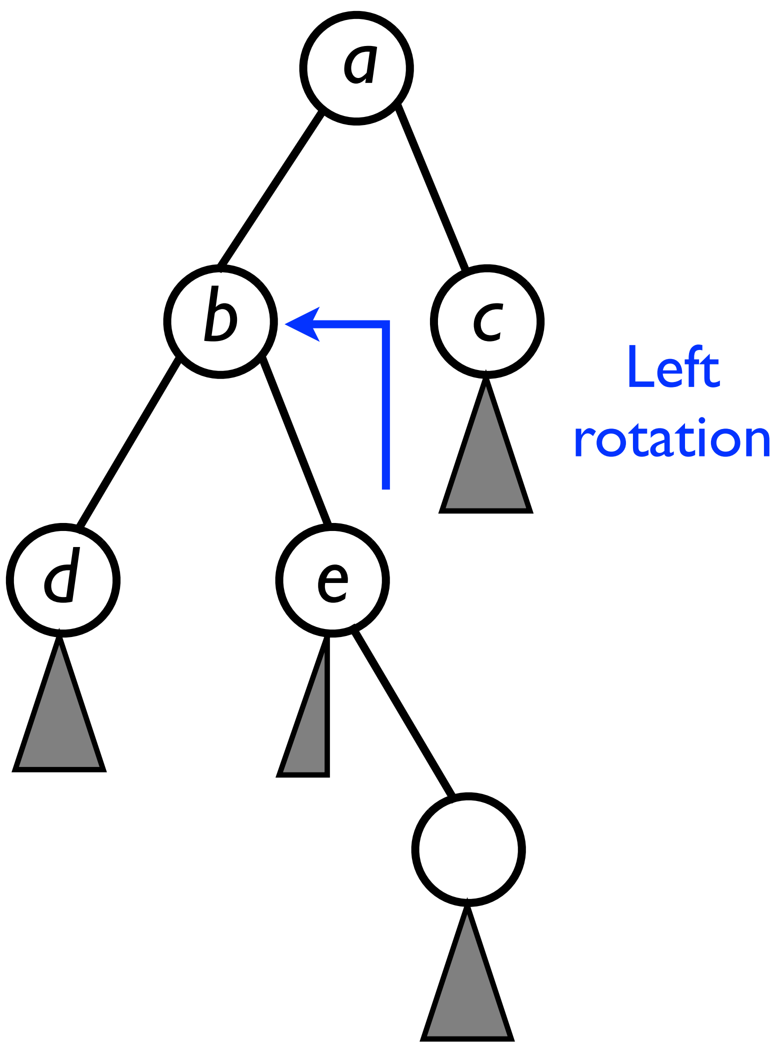
Original tree

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

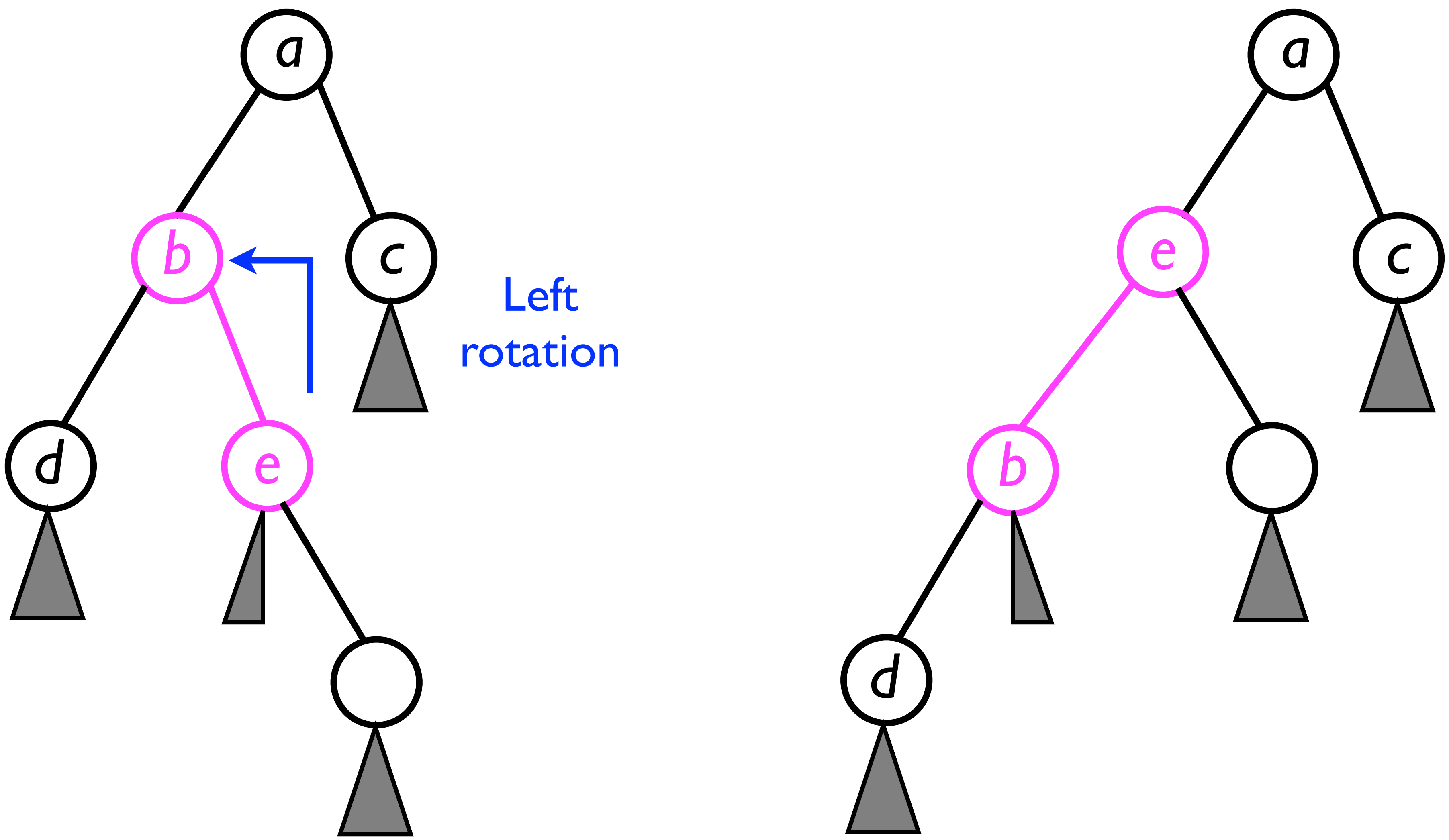
- To fix the imbalance in node *a*, we will *first* perform a *left rotation* of node e towards *b*.



Original tree

Fixing configuration LR

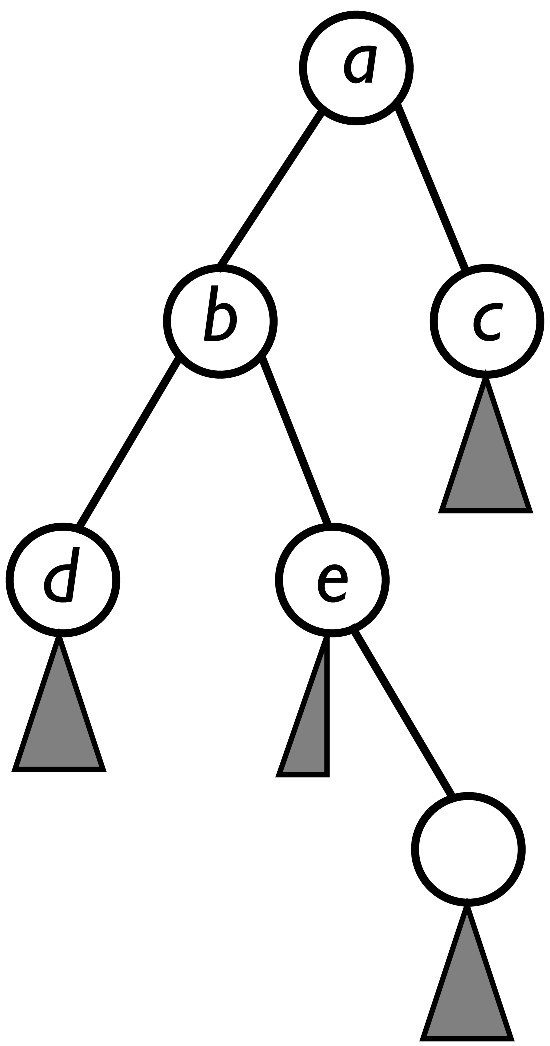
- To fix the imbalance in node *a*, we will *first* perform a *left rotation* of node e towards *b*.



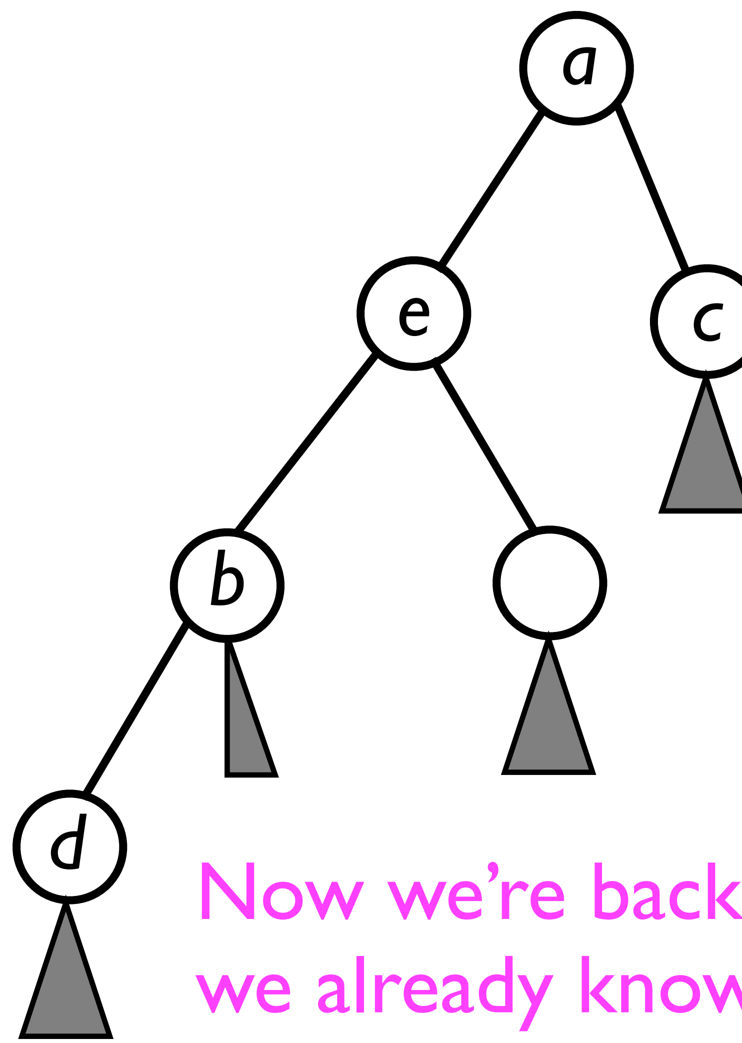
Original tree

Fixing configuration LR

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



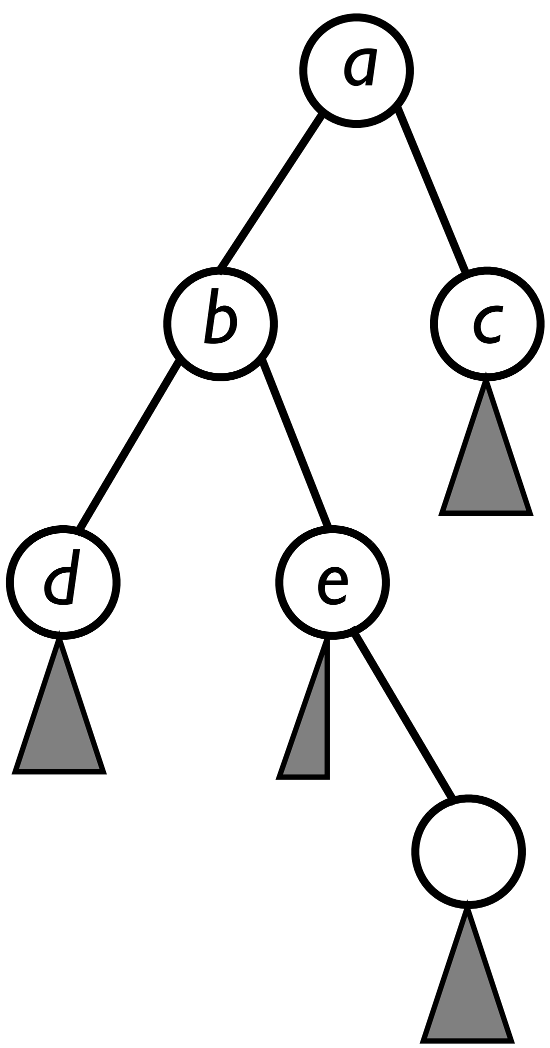
Original tree



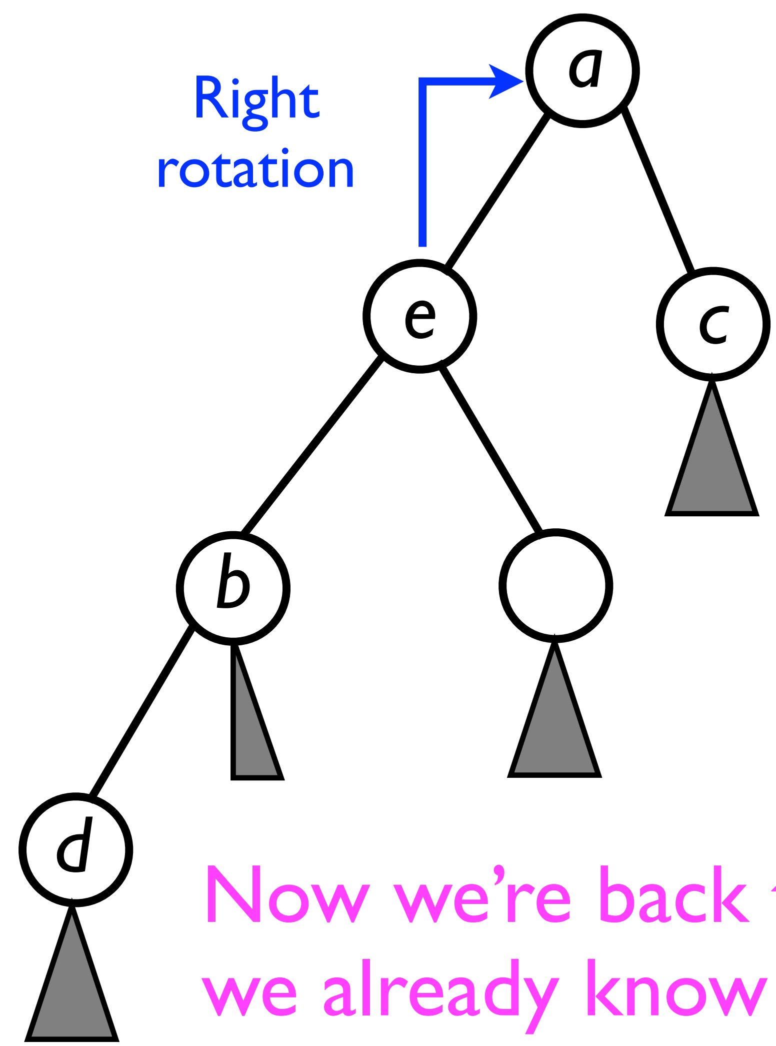
Now we're back to LL -- and we already know how to correct this (by applying a *right rotation of* e *towards* a).

Fixing configuration LR

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



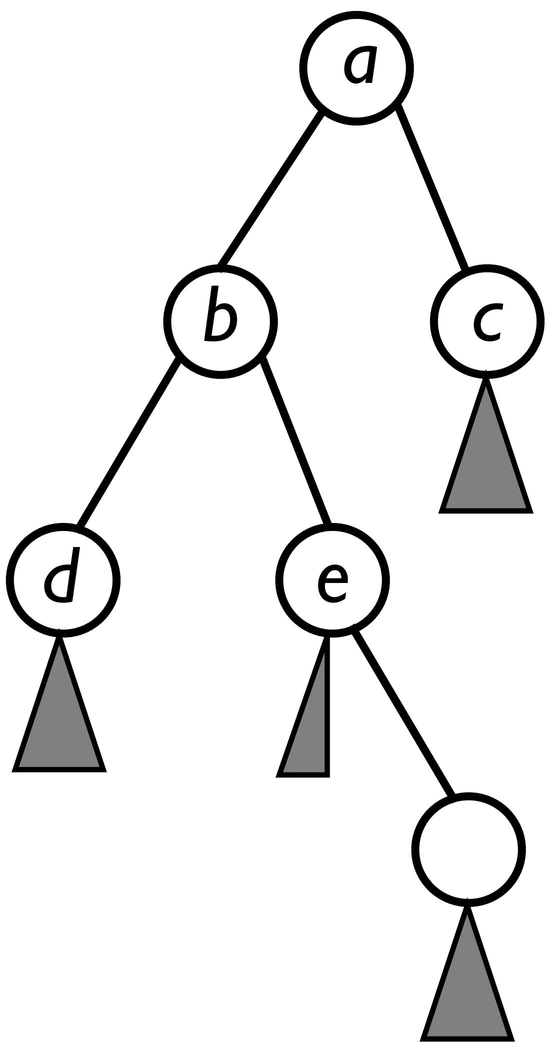
Original tree



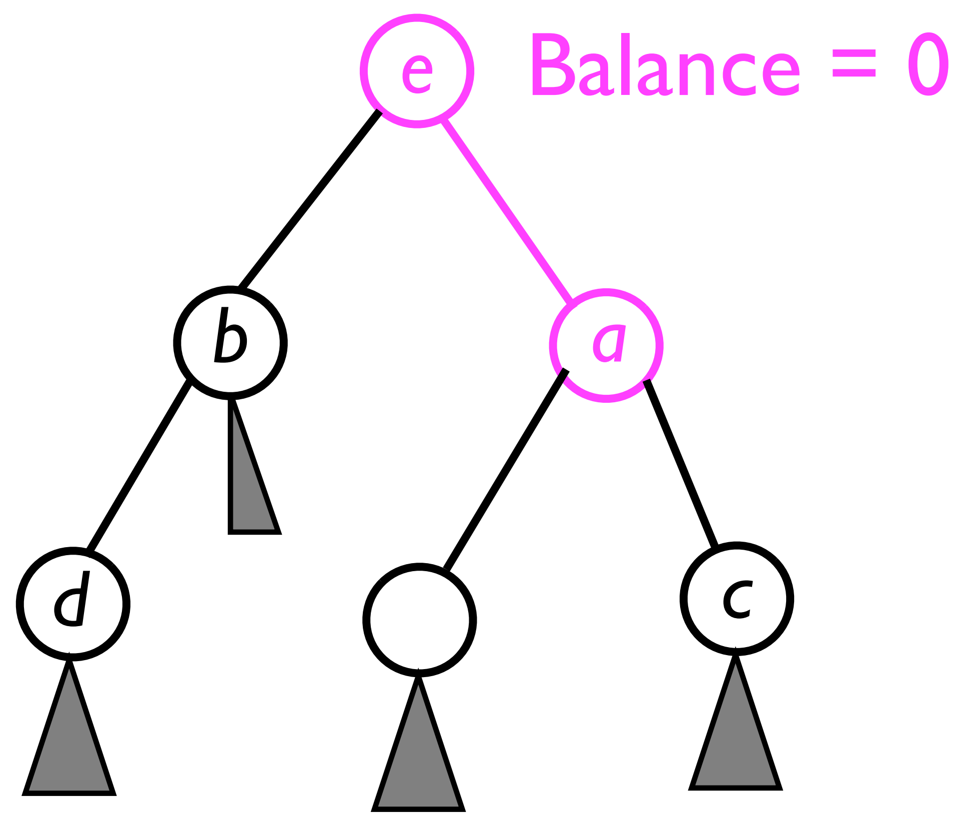
Now we're back to LL -- and we already know how to correct this (by applying a *right rotation of* e *towards* a*)*.

Fixing configuration LR

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

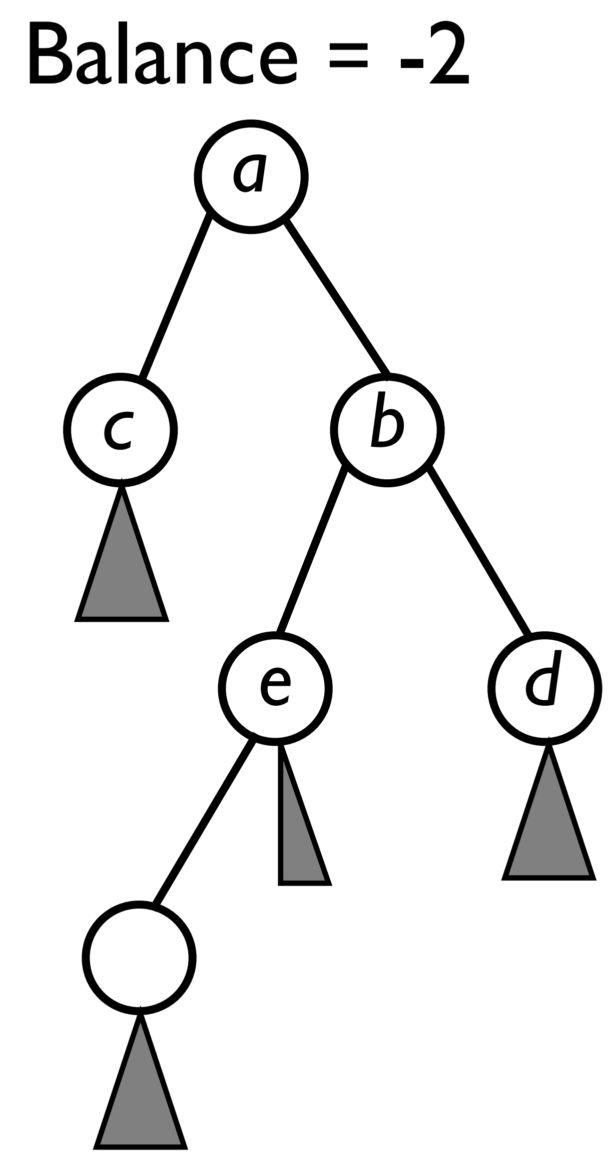


Original tree



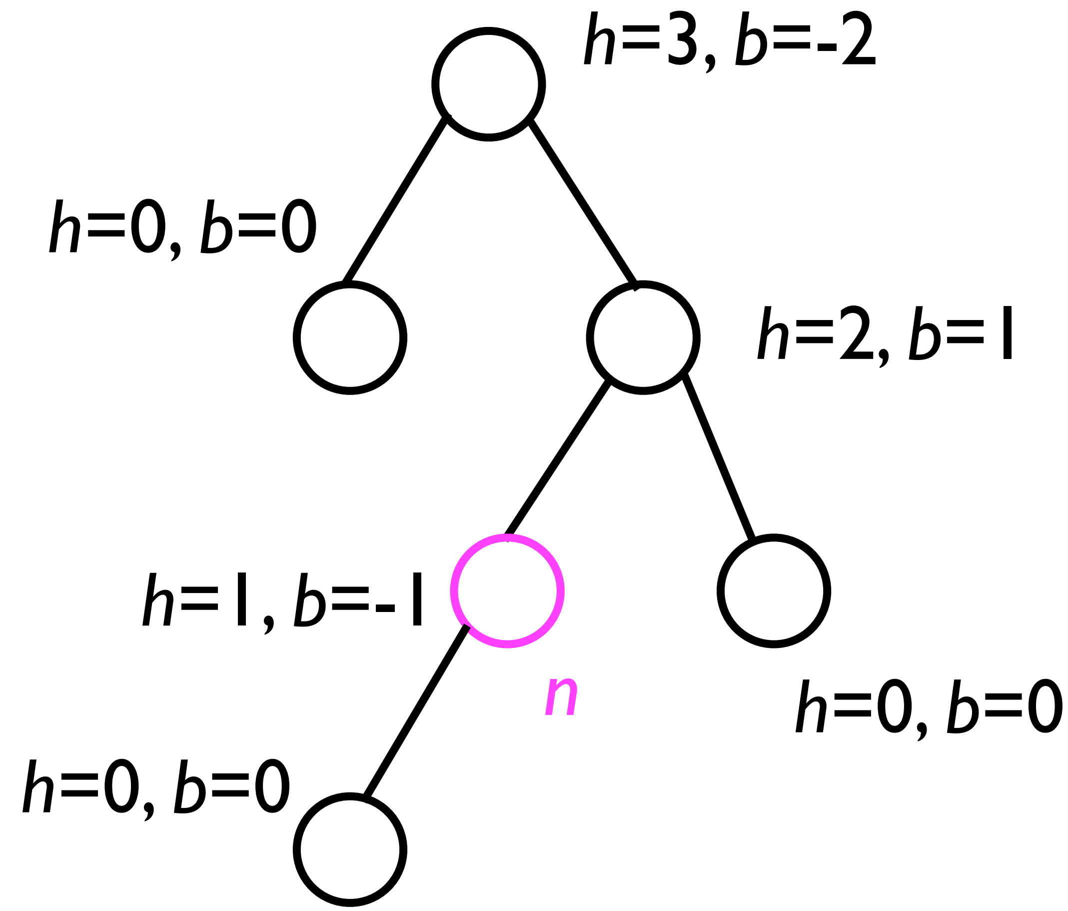
Fixing configuration RL

- 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”.



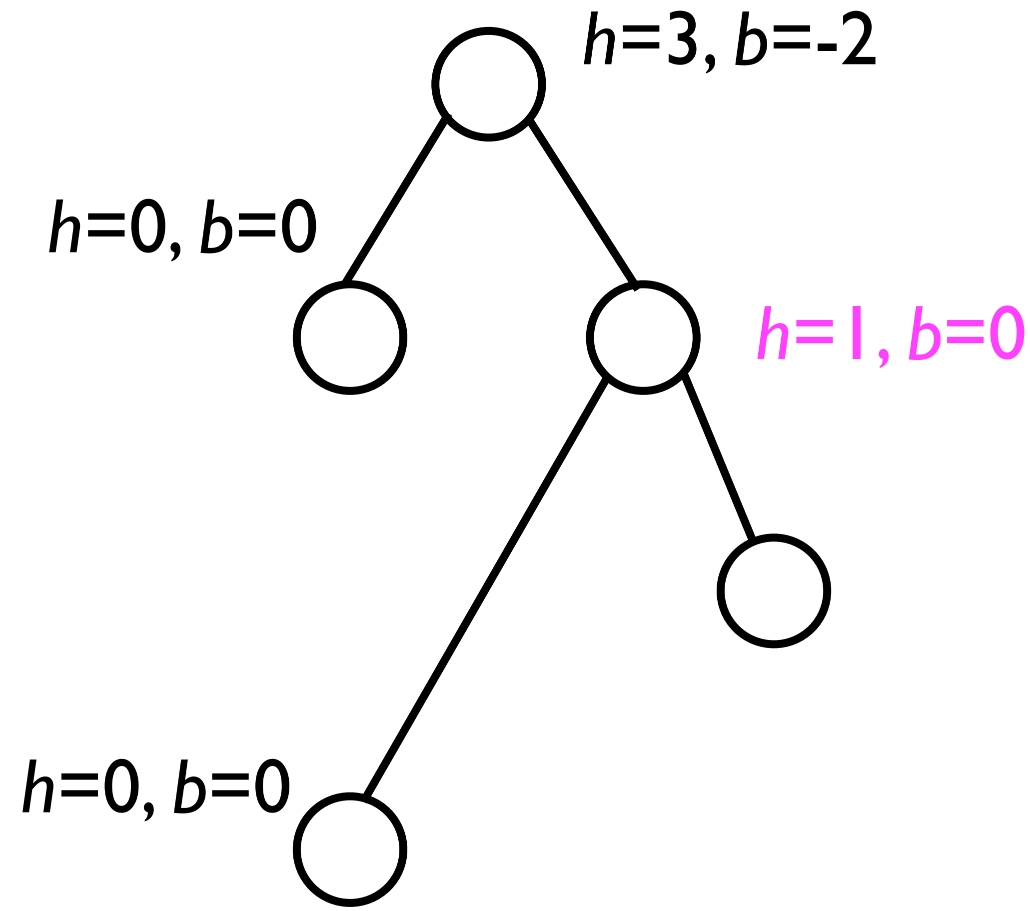
Removing a new node

- 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.



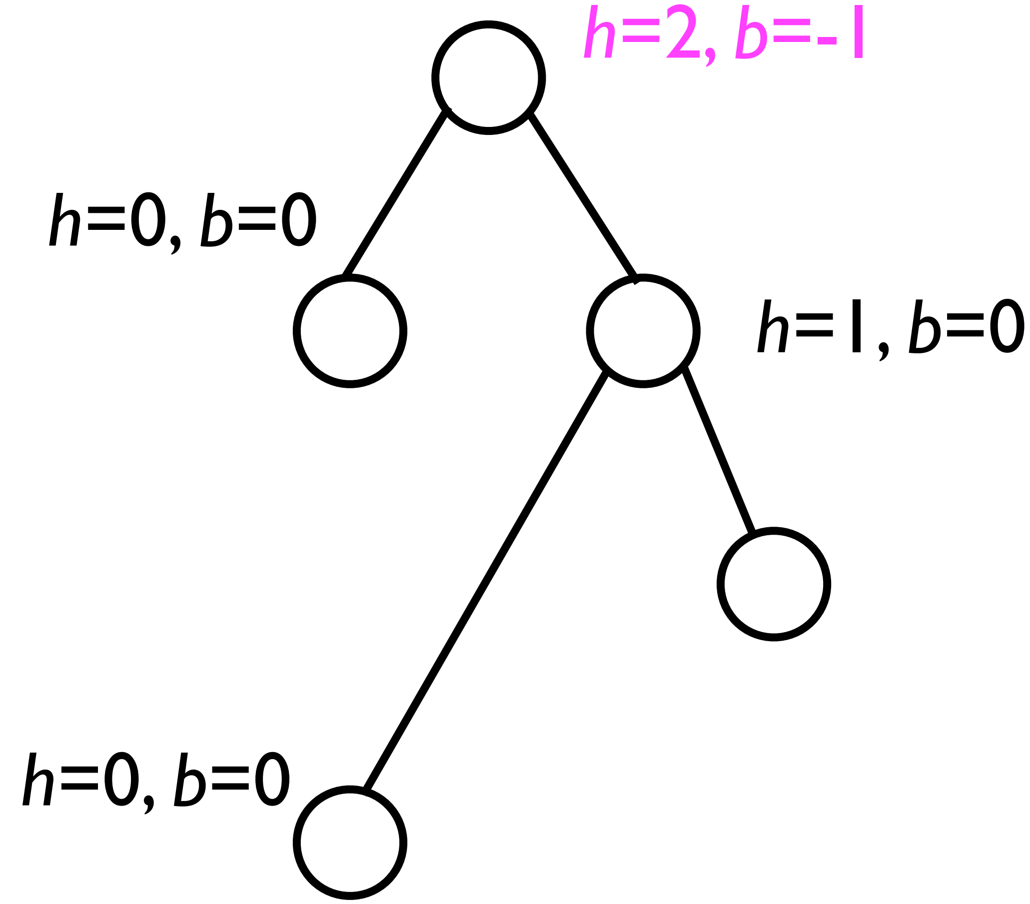
Removing a new node

- 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

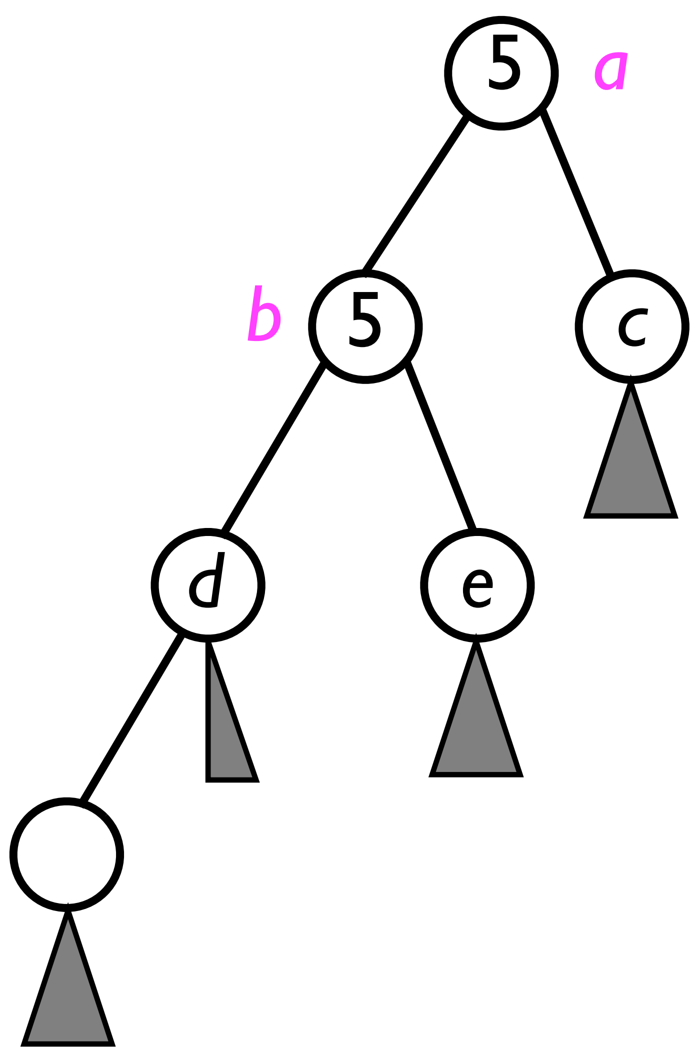
- With "regular" BSTs, the downside of storing multiple elements with the same key was primarily related to *performance*:
  - Adding multiple elements of the same keys requires unnecessary node storage and slows down the tree operations.
- However, with AVL trees, allowing duplicates would cause a problem in *correctness*:
  - Rotating nodes where duplicate keys are allowed can violate the BST ordering property.

Duplicate keys

- Consider:
  - To allow duplicates in a BST, we might "relax" the ordering condition slightly:
    - Given node *n*, every node in *n*'s *left* sub-tree should be *less-than-or-equal-to* *n*.
    - Every node in *n*'s *right* sub-tree should be greater than *n*.
  - The **`findNode`** method will rely on this ordering property to find a given node properly.

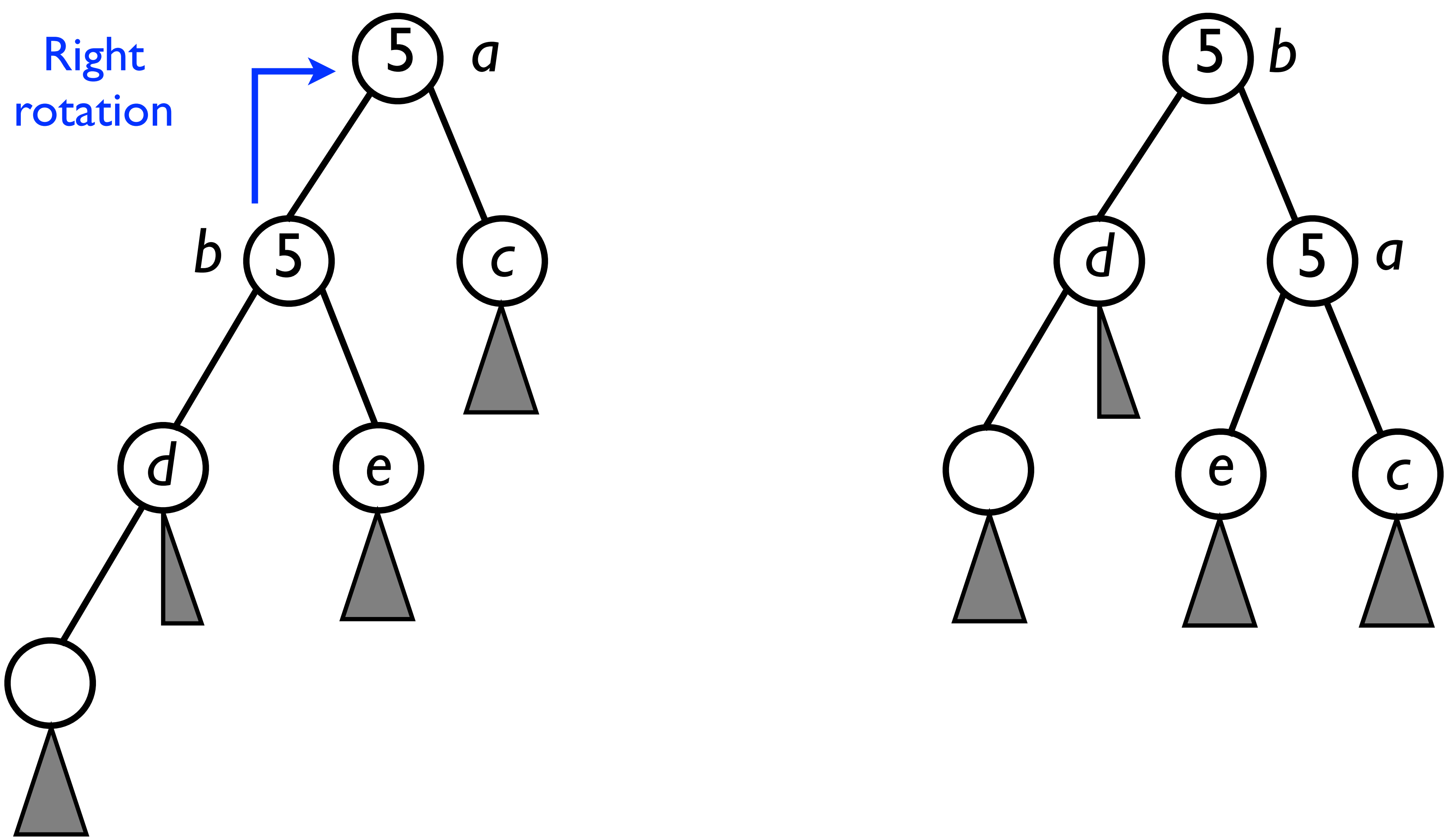
Duplicate keys

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



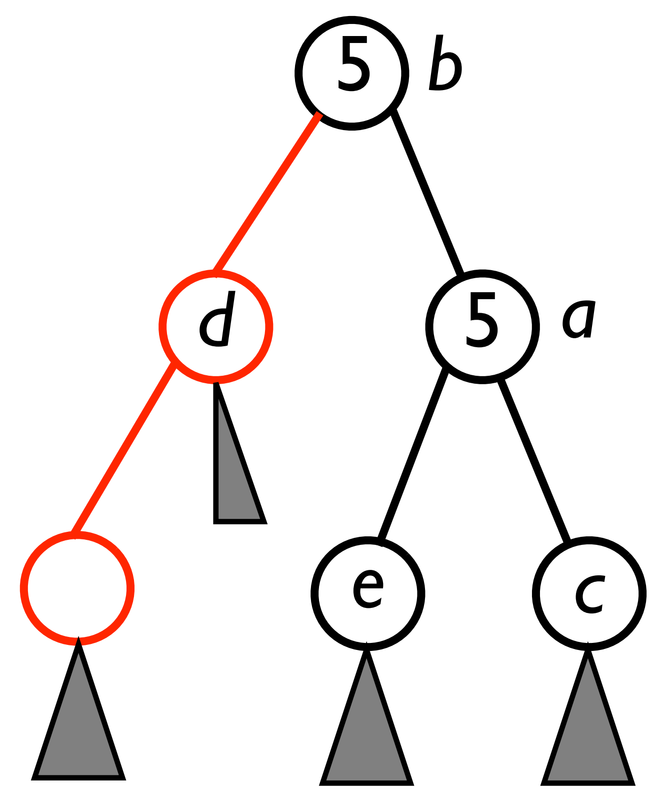
Duplicate keys

- 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 *right-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*.