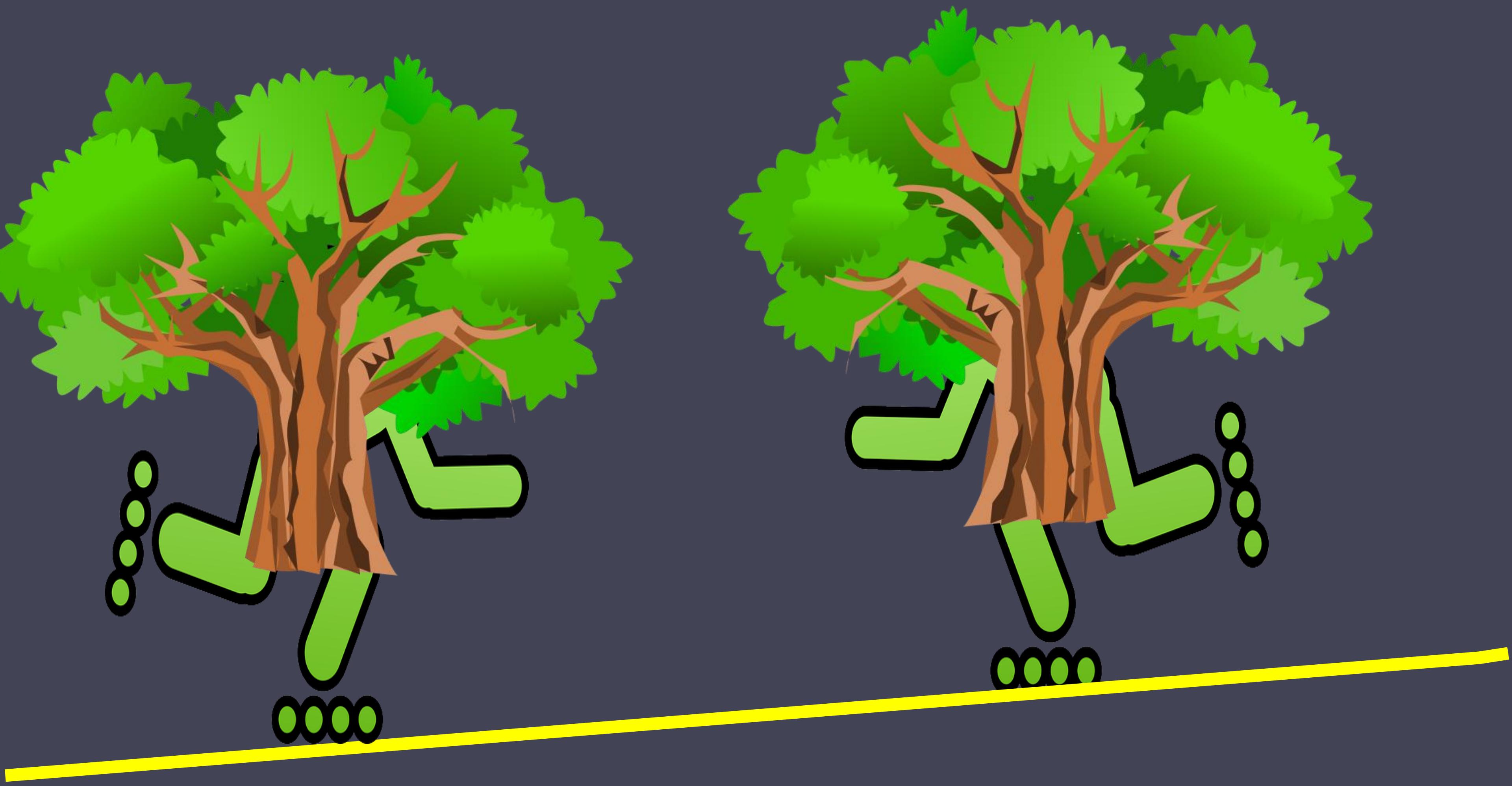


AVL TREES: CONCLUSION

Lecture 38

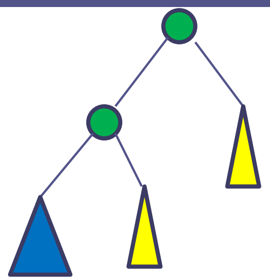
Assignment

- Examine the code (recitation will help)
- Ensure that you can build AVL trees from a given sequence of numbers (i.e., {34, 15, 7, 21, 90, 3, 50, 16, 4, 9}) reasonably fast (5-10 min.)
- (Extra credit) Implement the code discussed in class—with your own variations, if you wish and be ready to present it

The taxonomy of unbalanced trees and the respective balancing action

- **Left-Left Tree**
  - Root's balance factor is *-2*
  - Left child's balance factor is *-1*

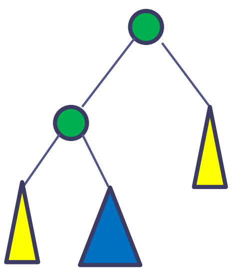
  *Action**:* Rotate right around the parent



- **Left-Right Tree**
  - Root's balance factor is *-2*
  - Left child's balance factor is *+1*

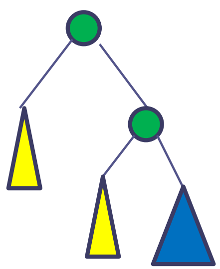
  *Action**: 1)* Rotate left around the child

  2)Rotate right around the parent



- **Right-Right Tree**
  - Root's balance factor is *+2*
  - Right child's balance factor is *+1*

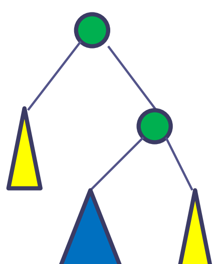
  *Action**:* Rotate left around the parent



- **Right-Left Tree**
  - Root's balance factor is *+2*
  - Right child's balance factor is *-1*

  *Action**: 1)* Rotate right around the child

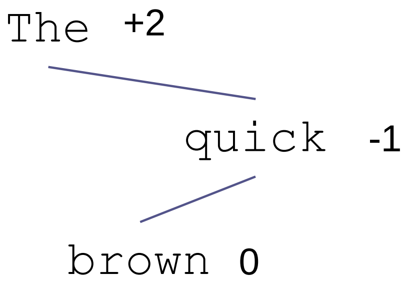
  2)Rotate left around the parent



An AVL Tree Example (from the book)

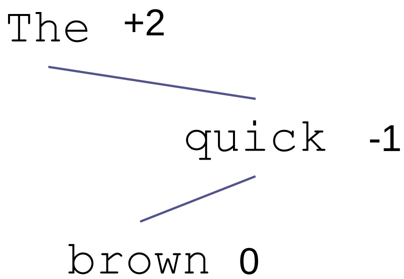
Build an AVL tree from the words in
"The quick brown fox jumps over the lazy dog"

AVL Tree Example



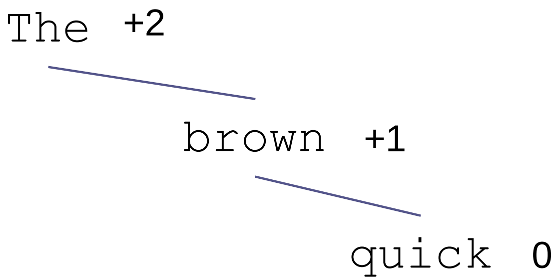
The overall tree is right-heavy
(Right-Left)
parent balance = +2
right child balance = -1

AVL Tree Example (cont.)



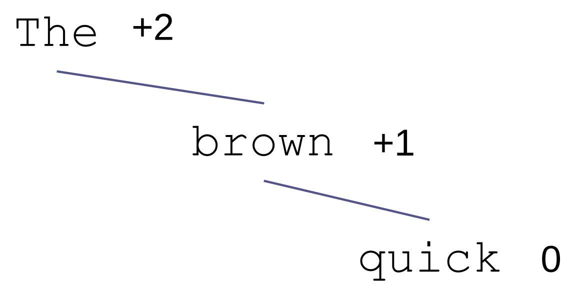
1. Rotate right around the child

AVL Tree Example (cont.)



1. Rotate right around the child

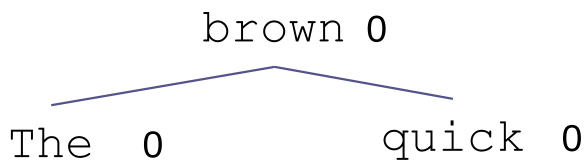
AVL Tree Example (cont.)



1. Rotate right around the child

2. Rotate left around the parent

AVL Tree Example (cont.)



1. Rotate right around the child

2. Rotate left around the parent

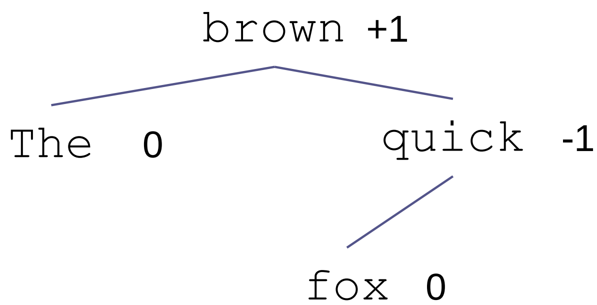
AVL Tree Example (cont.)

brown 0

The 0 quick 0

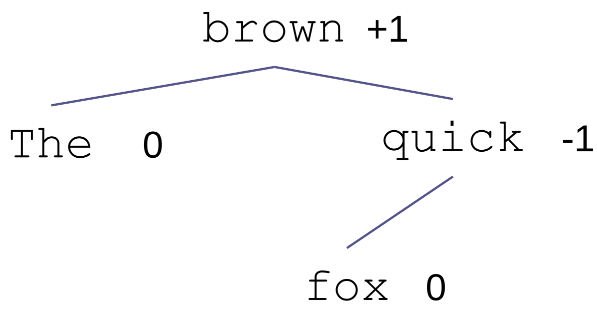
Insert *fox*

AVL Tree Example (cont.)

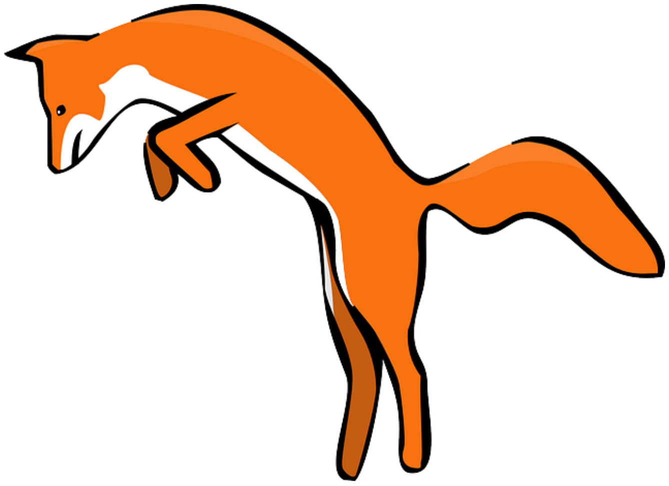


Insert *fox*

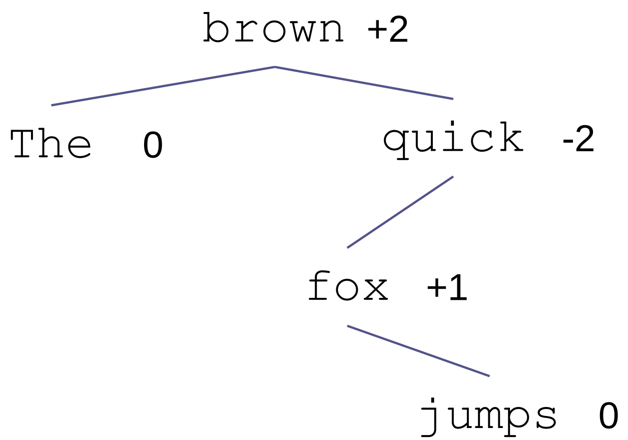
AVL Tree Example (cont.)



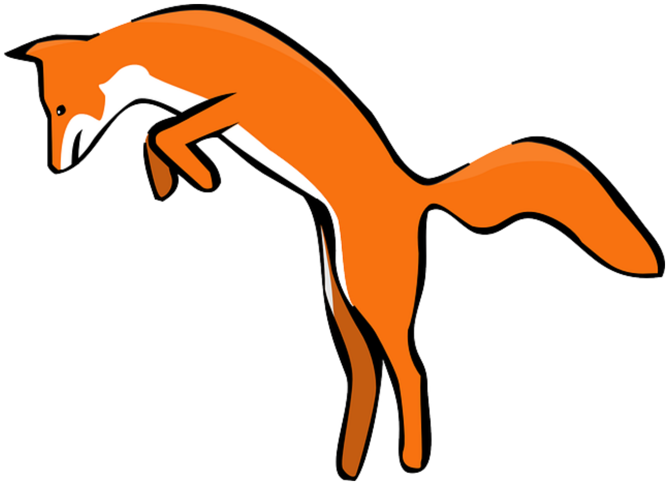
Insert *jumps*



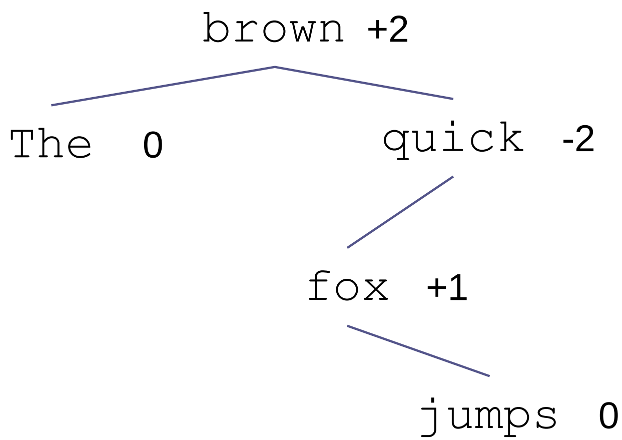
AVL Tree Example (cont.)



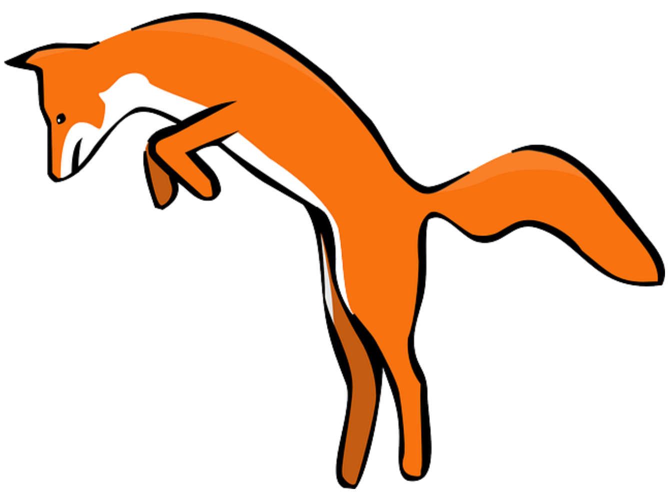
Insert *jumps*



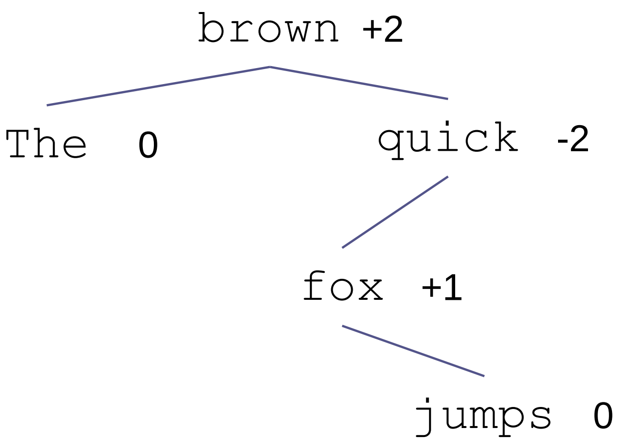
AVL Tree Example (cont.)



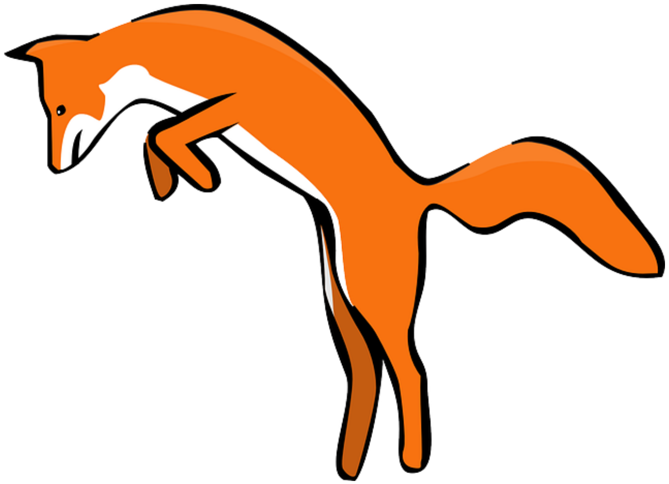
The tree is now left-heavy about *quick* (Left-Right case)



AVL Tree Example (cont.)



1. Rotate left around the child



AVL Tree Example (cont.)

brown +2

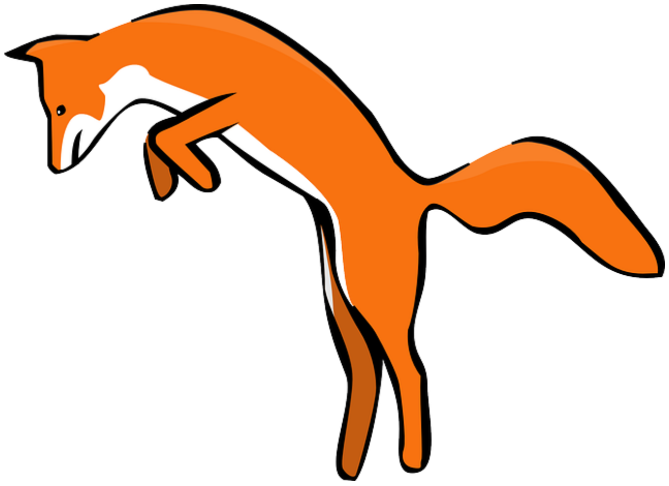
The 0

quick -2

jumps -1

fox 0

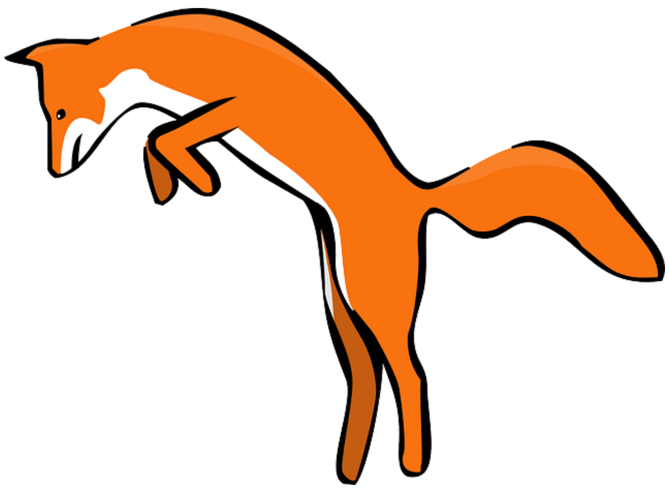
1. Rotate left around the child



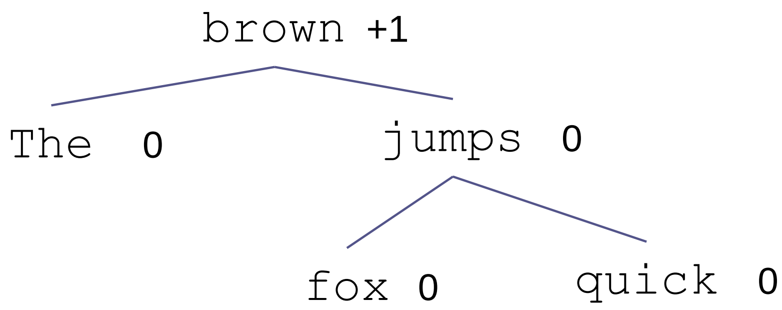
AVL Tree Example (cont.)

```
                 brown +2
              /            \
    The  0                  quick  -2
                           /
                    jumps  -1
                   /
             fox  0
```

1. Rotate left around the child
2. Rotate right around the parent

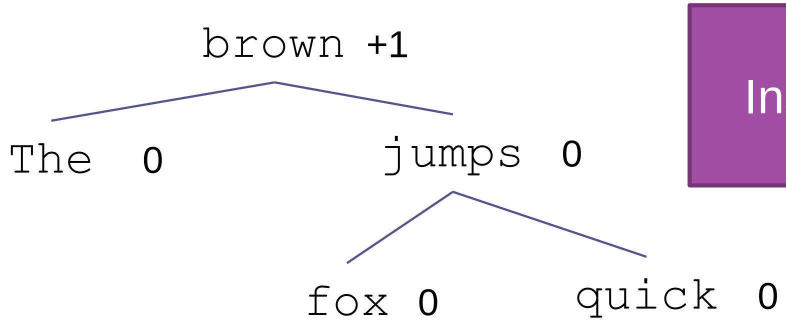


AVL Tree Example (cont.)

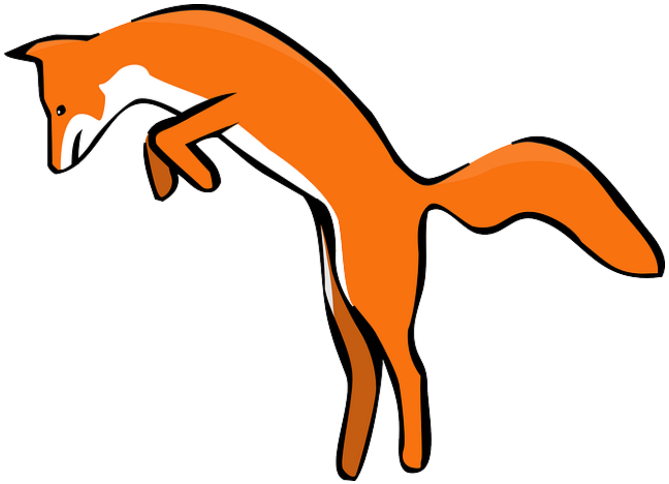


1. Rotate left around the child
2. Rotate right around the parent

AVL Tree Example (cont.)

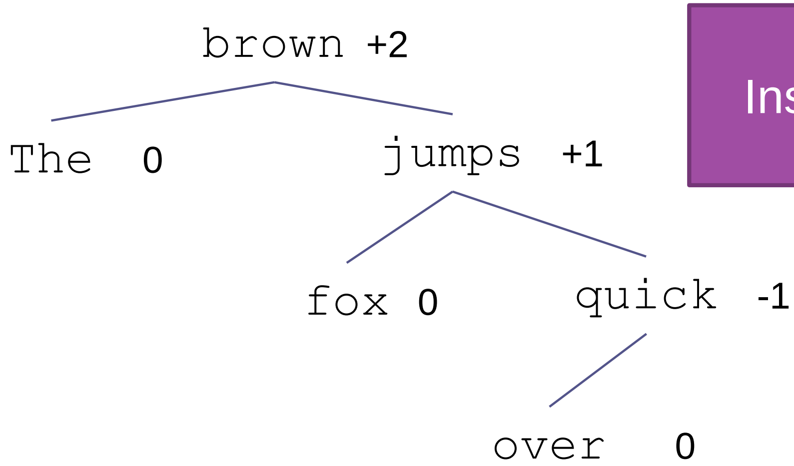


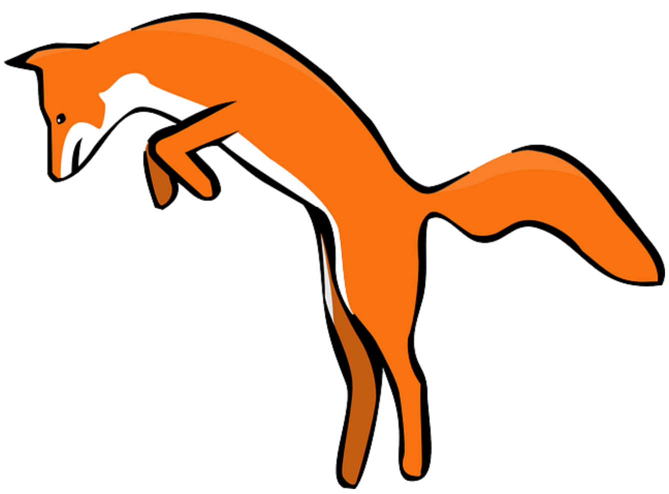
Insert *over*



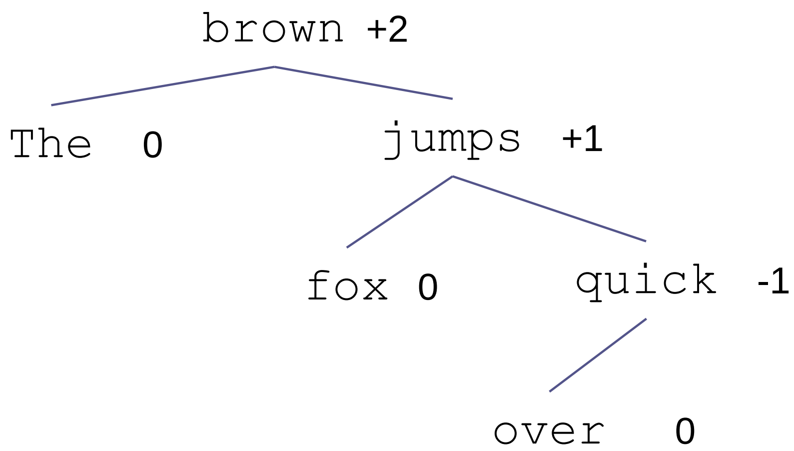
AVL Tree Example (cont.)

Insert *over*

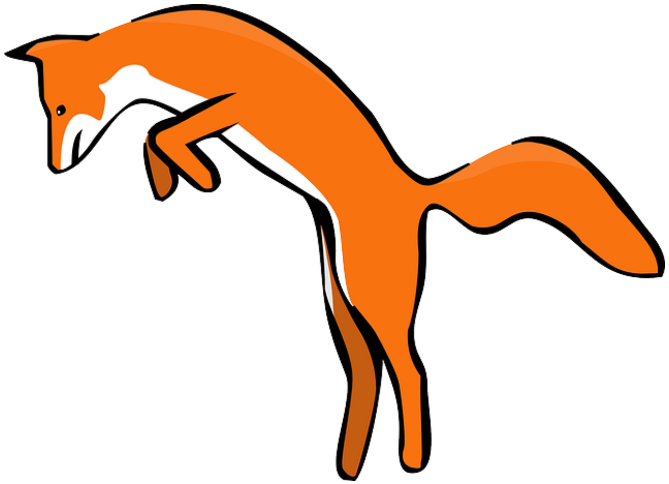




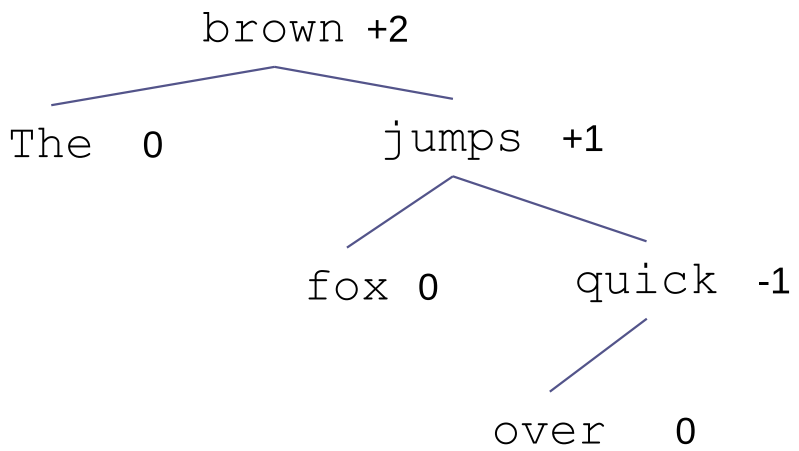
AVL Tree Example (cont.)



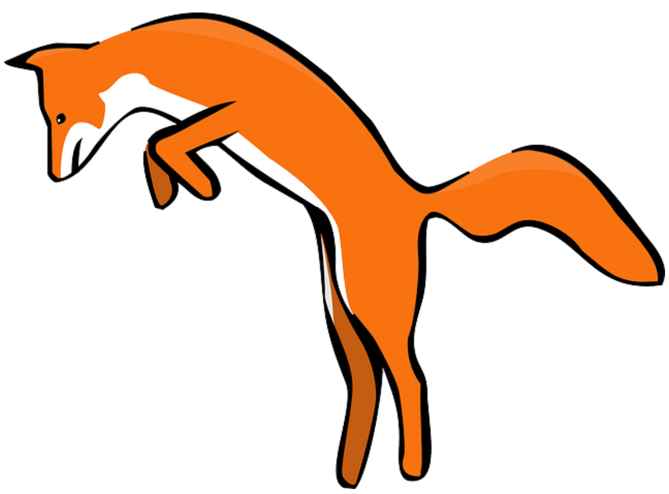
We now have a Right-Right imbalance



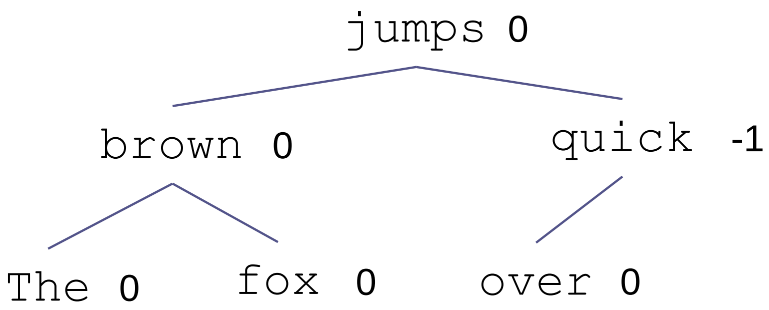
AVL Tree Example (cont.)



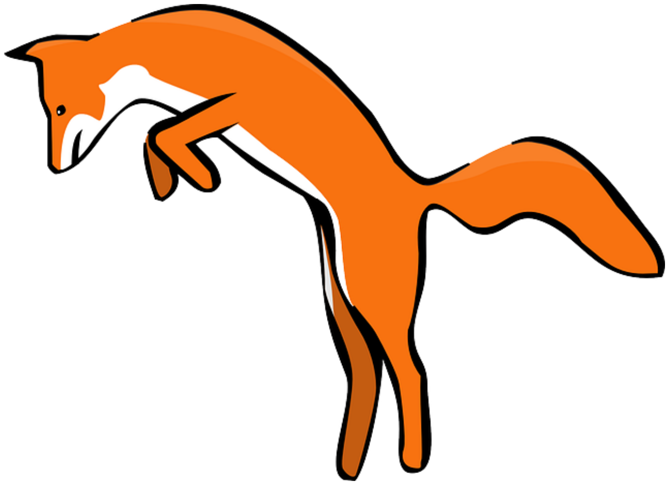
1. Rotate left around the parent



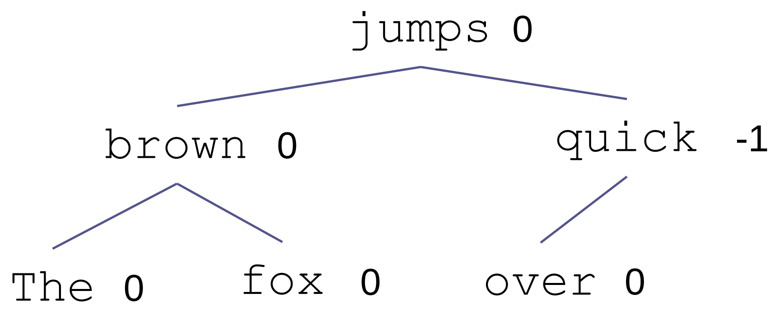
AVL Tree Example (cont.)



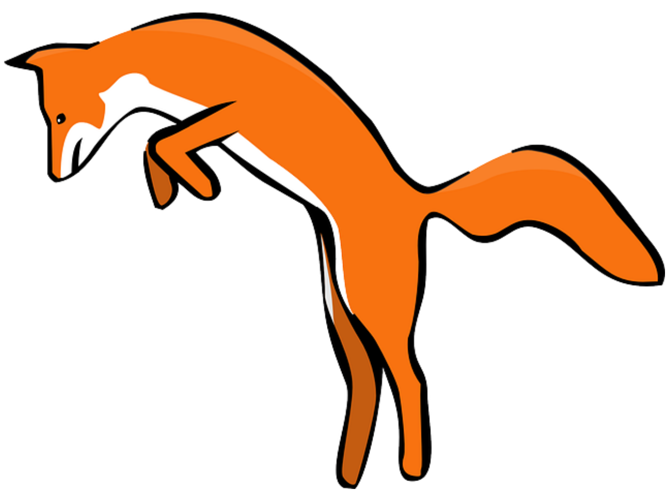
1. Rotate left around the parent



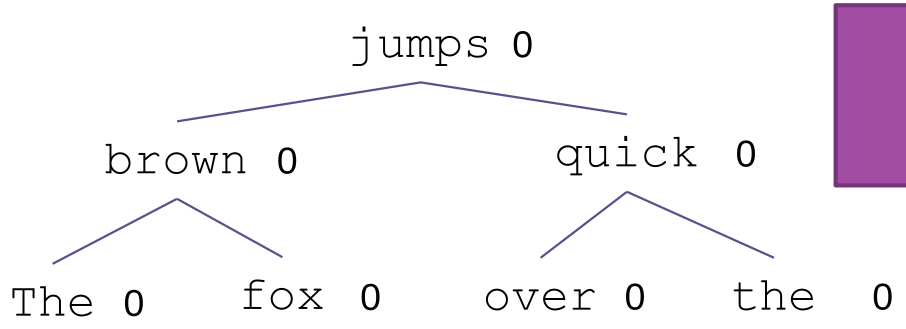
AVL Tree Example (cont.)



Insert *the*

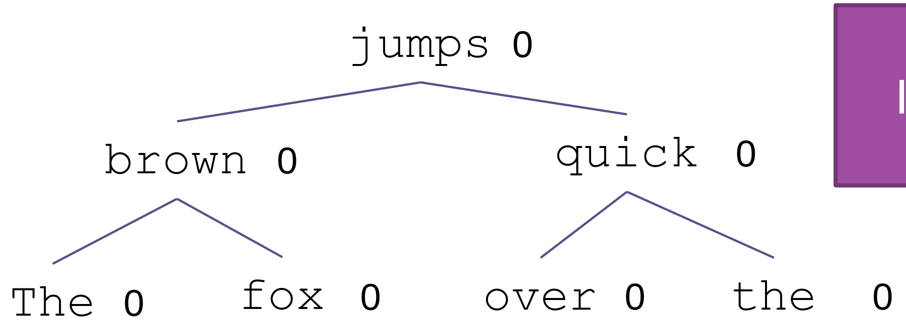


AVL Tree Example (cont.)



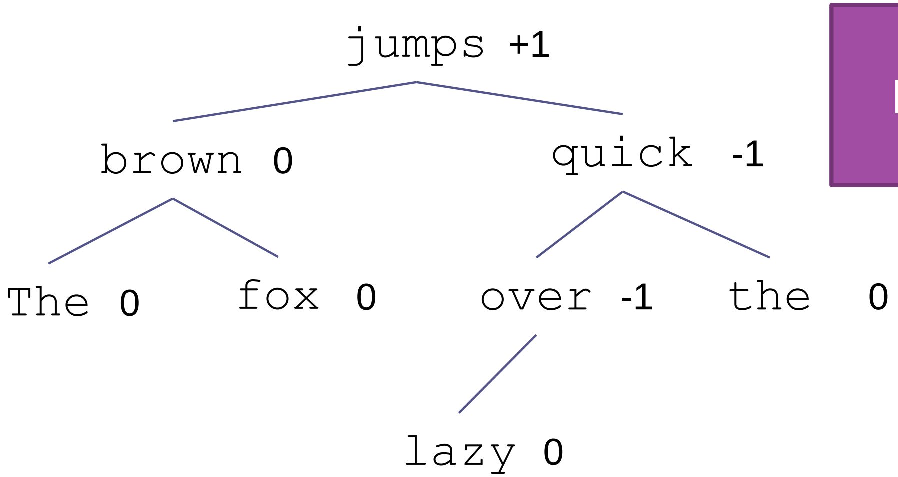
Insert *the*

AVL Tree Example (cont.)



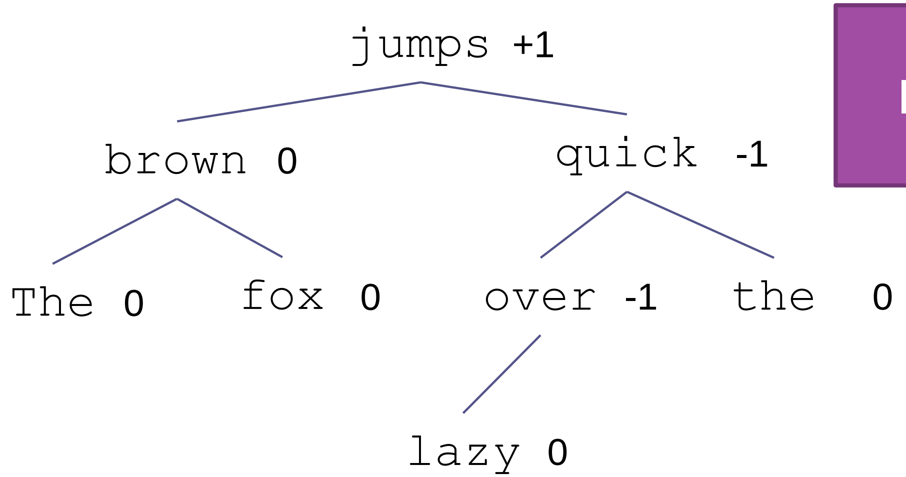
Insert *lazy*

AVL Tree Example (cont.)

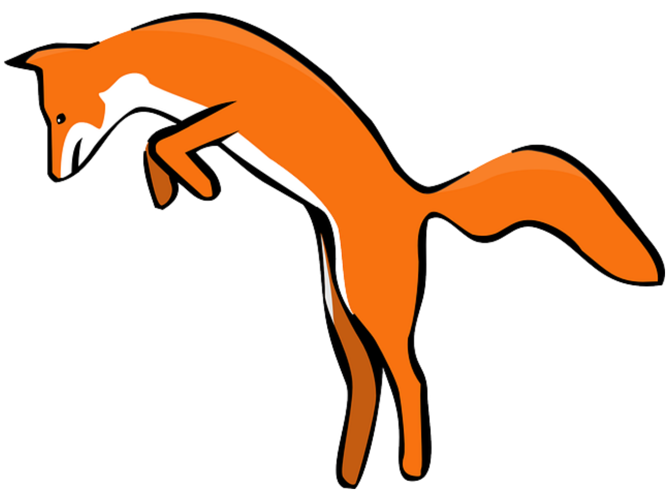


Insert *lazy*

AVL Tree Example (cont.)

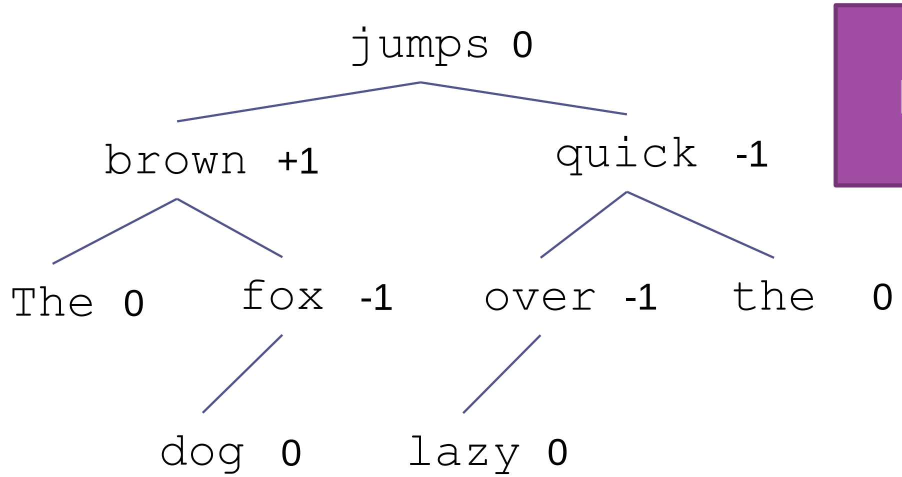


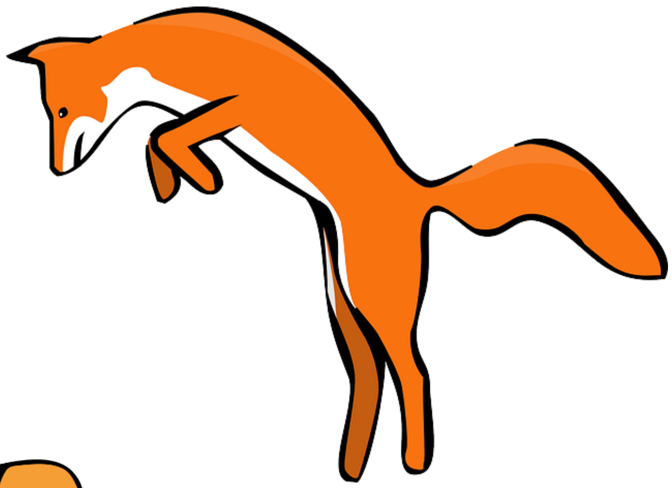
Insert *dog*

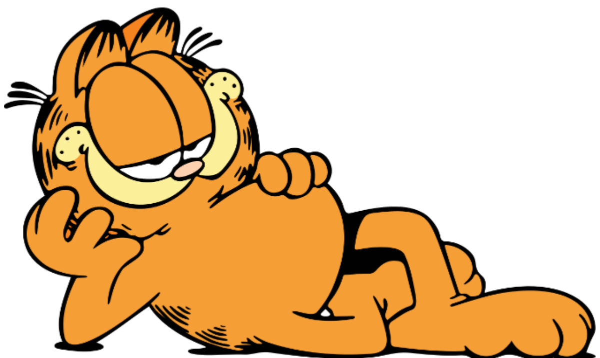


AVL Tree Example (cont.)

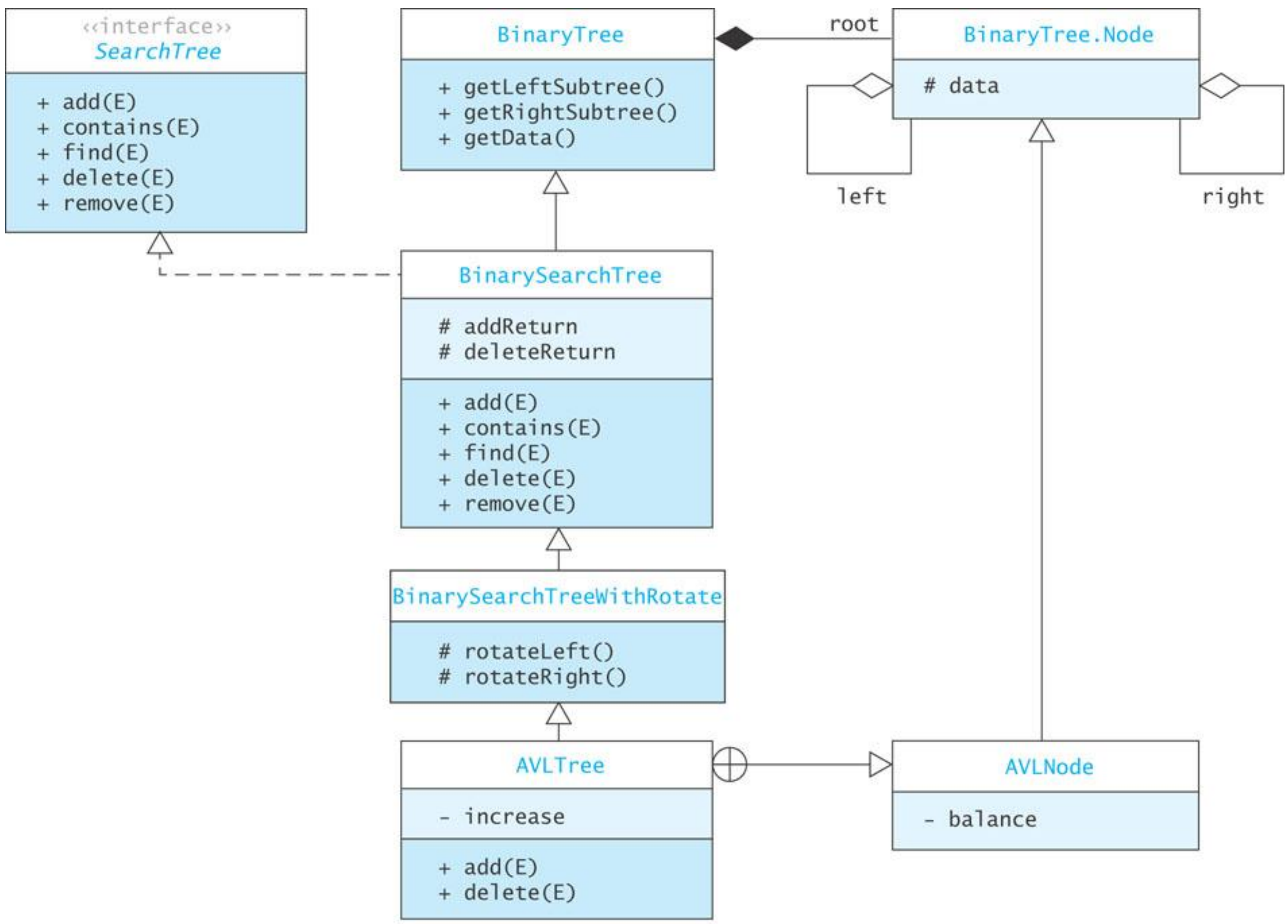
Insert *dog*







Implementing an AVL Tree



The `AVLNode` Class

- Listing 9.2 (`The AVLNode Class`, pages 482-483)

Inserting into an AVL Tree

- The easiest way to keep a tree balanced is never to let it become unbalanced
- If any node becomes critical, rebalance immediately
- Identify critical nodes by checking the balance at the root node as you return along the insertion path

Inserting into an AVL Tree (cont.)

Algorithm for Insertion into an AVL Tree

1. **if** the **root** is **null**
2. Create a new tree with the item at the `root` and return **true.**

   else if the item is equal to **root.data**
3. The item is already in the tree; return **false**.

   else if the item is less than **root.data**
4. (Recursively) insert the item in the left subtree.
5. **if** the height of the left subtree has increased (**increase** is **true**)
6. Decrement balance.
7. **if balance** is zero, reset **increase to false.**
8. **if balance** is less than –1
9. Reset **increase** to **false**.
10. Perform a rebalanceLeft.

    else if the item is greater than **root.data**
11. The processing is symmetric to Steps 4 through 10. Note that balance is incremented if **increase** is true.

add Starter Method

```
/** add starter method.
    pre: the item to insert implements the Comparable interface.
    @param item The item being inserted.
    @return true if the object is inserted; false
         if the object already exists in the tree
    @throws ClassCastException if item is not Comparable
*/
@Override
public boolean add(E item) {
    increase = false;
    root = add((AVLNode<E>) root, item);
    return addReturn;
}
```

Recursive `add` method

```
/** Recursive add method. Inserts the given object into the tree.
     post: addReturn is set true if the item is inserted,
        false if the item is already in the tree.
     @param localRoot The local root of the subtree
     @param item The object to be inserted
     @return The new local root of the subtree with the item
        inserted
*/
private AVLNode<E> add(AVLNode<E> localRoot, E item)
if (localRoot == null) {
     addReturn = true;
     increase = true;
     return new AVLNode<E>(item);
}
if (item.compareTo(localRoot.data) == 0) {
     // Item is already in the tree.
     increase = false;
     addReturn = false;
     return localRoot;
}
```

Recursive `add` method (cont.)

```
else if (item.compareTo(localRoot.data) < 0) {
      // item < data
      localRoot.left = add((AVLNode<E>) localRoot.left, item);
. . .

if (increase) {
      decrementBalance(localRoot);
      if (localRoot.balance < AVLNode.LEFT_HEAVY) {
          increase = false;
          return rebalanceLeft(localRoot);
      }
}
return localRoot; // Rebalance not needed.
```

Initial Algorithm for `rebalanceLeft`

Initial Algorithm for `rebalanceLeft`

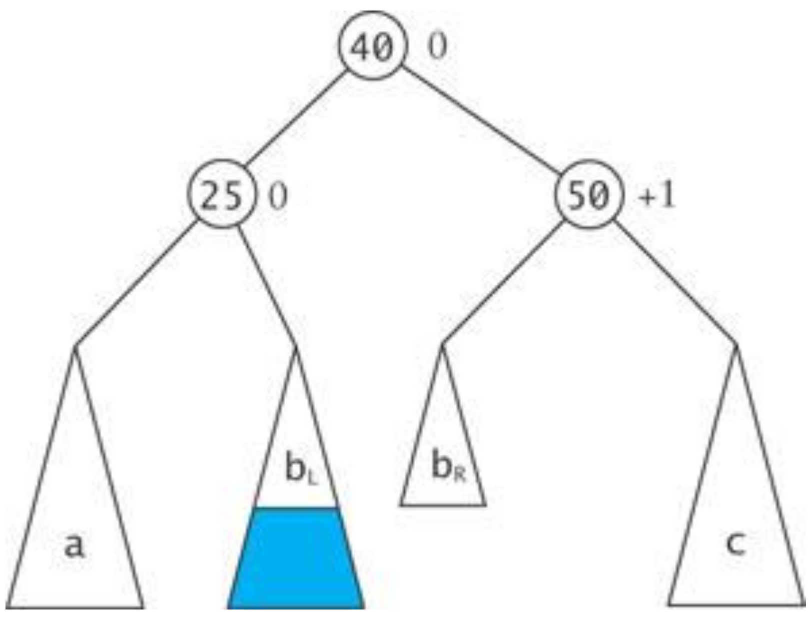
1. **`if`** the left subtree has positive balance (Left-Right case)
2. Rotate left around left subtree root.
3. Rotate right.

Effect of Rotations on Balance

- The rebalance algorithm on the previous slide was incomplete as the balance of the nodes was not adjusted
- For a Left-Left tree the balances of the new root node and of its right child are 0 after a right rotation
- The Left-Right tree is more complex: the balance of the root is 0

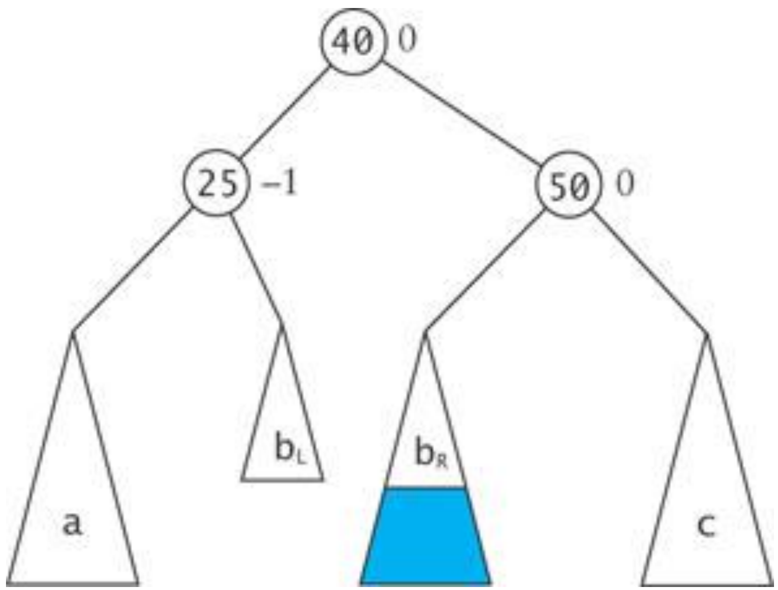
Effect of Rotations on Balance (cont.)

- if the critically unbalanced situation was caused by an insertion into
  - subtree b_L (Left-Right-Left case), the balance of the root's left child is 0 and the balance of the root's right child is +1



Effect of Rotations on Balance (cont.)

- if the critically unbalanced situation was caused by an insertion into
  - subtree b_R (Left-Right-Right case), the balance of the root's left child is -1 and the balance of the root's right child is 0



Revised Algorithm for `rebalanceLeft`

Revised Algorithm for `rebalanceLeft`

```
1.  if the left subtree has a positive balance (Left-Right case)
2.      if the left-left subtree has a negative balance (Left-Right-Left case)
3.          Set the left subtree (new left subtree) balance to 0.
4.          Set the left-left subtree (new root) balance to 0.
5.          Set the local root (new right subtree) balance to +1.
        else (Left-Right-Right case)
6.          Set the left subtree (new left subtree) balance to –1.
7.          Set the left-left subtree (new root) balance to 0.
8.          Set the local root (new right subtree) balance to 0.
9.      Rotate the left subtree left.
    else (Left-Left case)
10.     Set the left subtree balance to 0.
11.     Set the local root balance to 0.
12. Rotate the local root right.
```

Method `rebalanceLeft`

- Listing 9.3 (`The rebalanceLeft Method`, page 487)

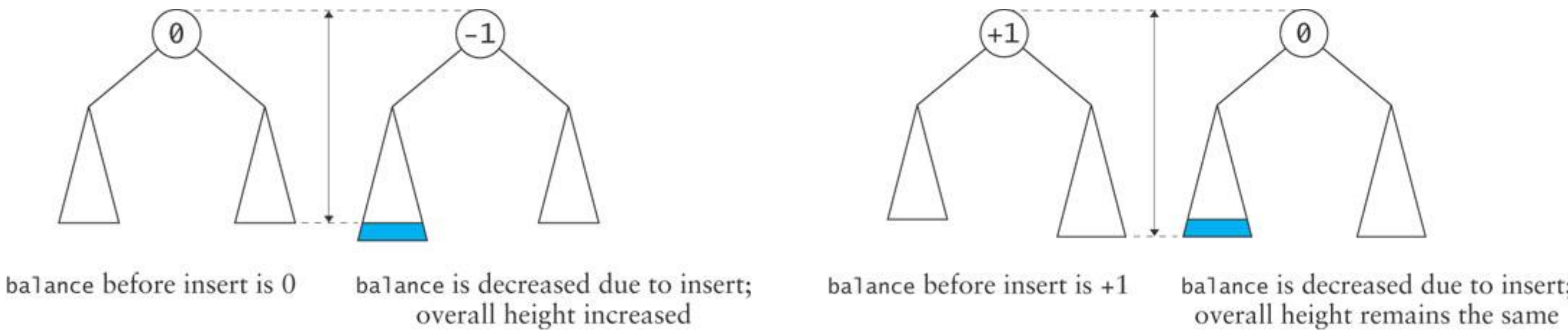
Method `rebalanceRight`

- The `rebalanceRight` method is *dual* with respect to the `rebalanceLeft` method

Method `decrementBalance`

- As we return from an insertion into a node's left subtree, we need to decrement the balance of the node
- We also need to indicate if the subtree height at that node has not increased (setting `increase` to `false`)

Method `decrementBalance` (cont.)



- Two cases to consider:
  - a balanced node – insertion into its left subtree will make it left-heavy and its height will increase by 1
  - a right-heavy node – insertion into its left subtree will cause it to become balanced and its height will not increase

Method `decrementBalance` (cont.)

```
private void decrementBalance(AVLNode<E> node) {
    // Decrement the balance.
    node.balance--;
    if (node.balance == AVLNode.BALANCED) {
       /** If now balanced, overall height has not increased. */
       increase = false;
    }

}
```

Removal from an AVL Tree

- Removal
  - from a left subtree, increases the balance of the local root
  - from a right subtree, decreases the balance of the local root
- The binary search tree removal method can be adapted for removal from an AVL tree
- A data field `decrease` tells the previous level in the recursion that there was a decrease in the height of the subtree from which the return occurred
- The local root balance is incremented or decremented based on this field
- If the balance is outside the threshold, a rebalance method is called to restore balance

Removal from an AVL Tree (cont.)

- Methods `decrementBalance`, `incrementBalance`, `rebalanceLeft`, and `rebalanceRight` need to be modified to set the value of `decrease` and `increase` after a node's balance is decremented
- Each recursive return can result in a further need to rebalance

Performance of the AVL Tree

- On each modification, we perform O(log n) operations
- In the worst case an AVL tree may be be 1.44 times the height of a full binary tree that contains the same number of items
- Empirical tests show that on average log n + 0.25 comparisons are required to insert the nth item into an AVL tree – close to a corresponding complete binary search tree