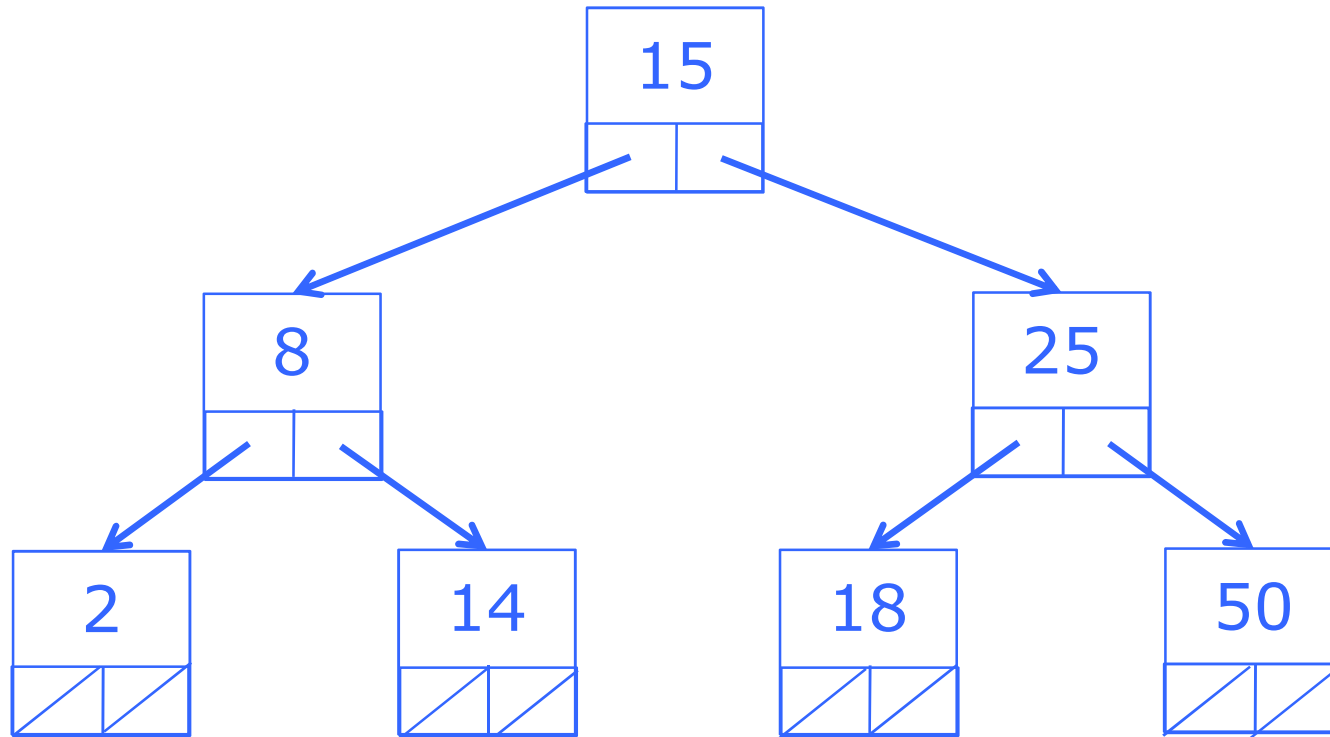


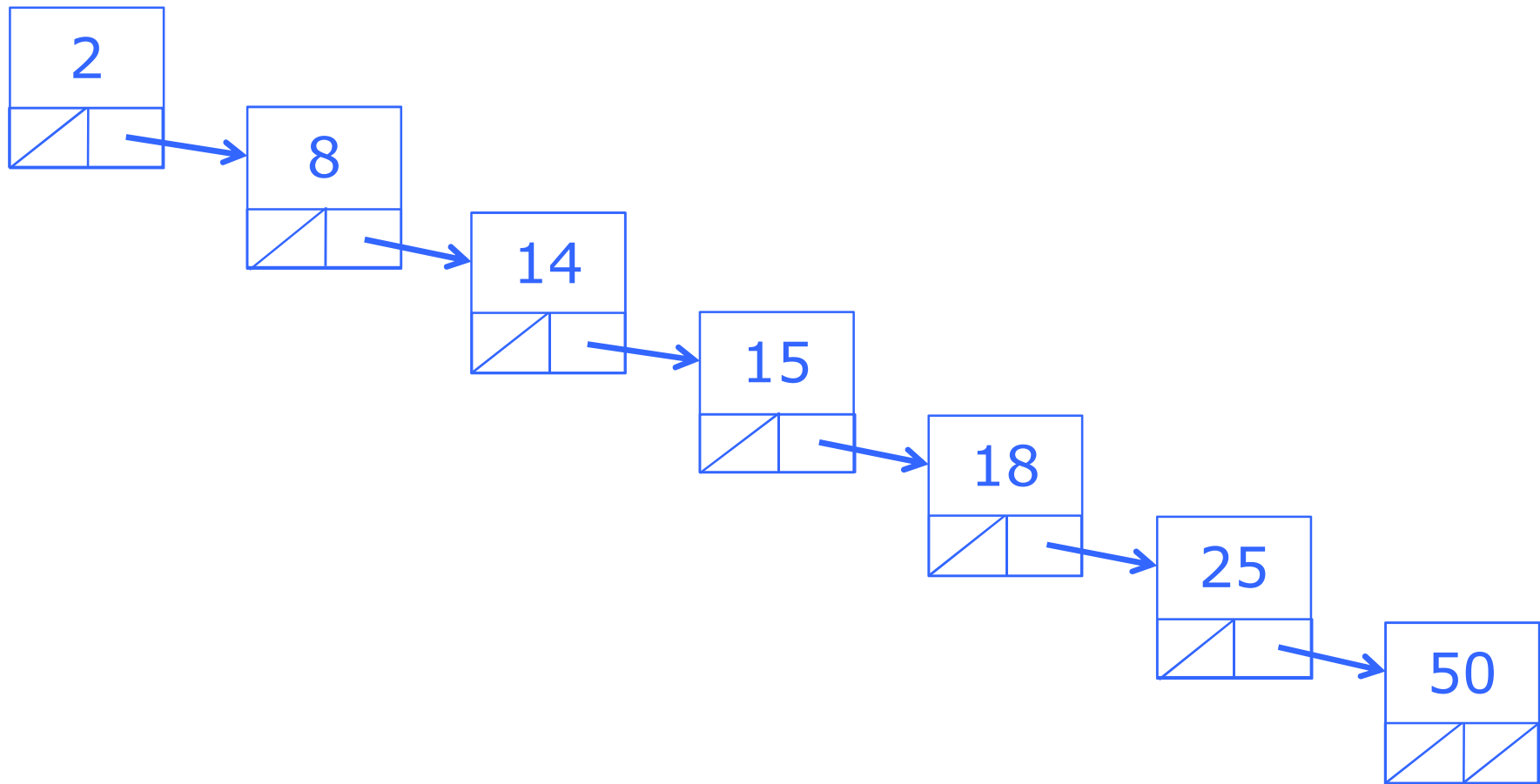
A (balanced) Tree



Values: 2 8 14 15 18 25 50
Inserted: 15 8 14 25 2 50 18

Look-up time: $O(\lg n)$

Another (imbalanced) Tree



Values: 2 8 14 15 18 25 50
Inserted: 2 8 14 15 18 25 50

Look-up time: $O(n)$, just like a linked list (b/c it is!)

In the event of imbalance...

Do nothing

- Only happens with particular order of insertion
- Ignoring the problem doesn't make it go away
- We still have time in class :)

Periodically Rebalance

- If the tree does get imbalanced, remove and reinsert nodes in appropriate order
- Periodic cost of $O(n)$, and might not be necessary to do
- How are we to define "imbalance"?

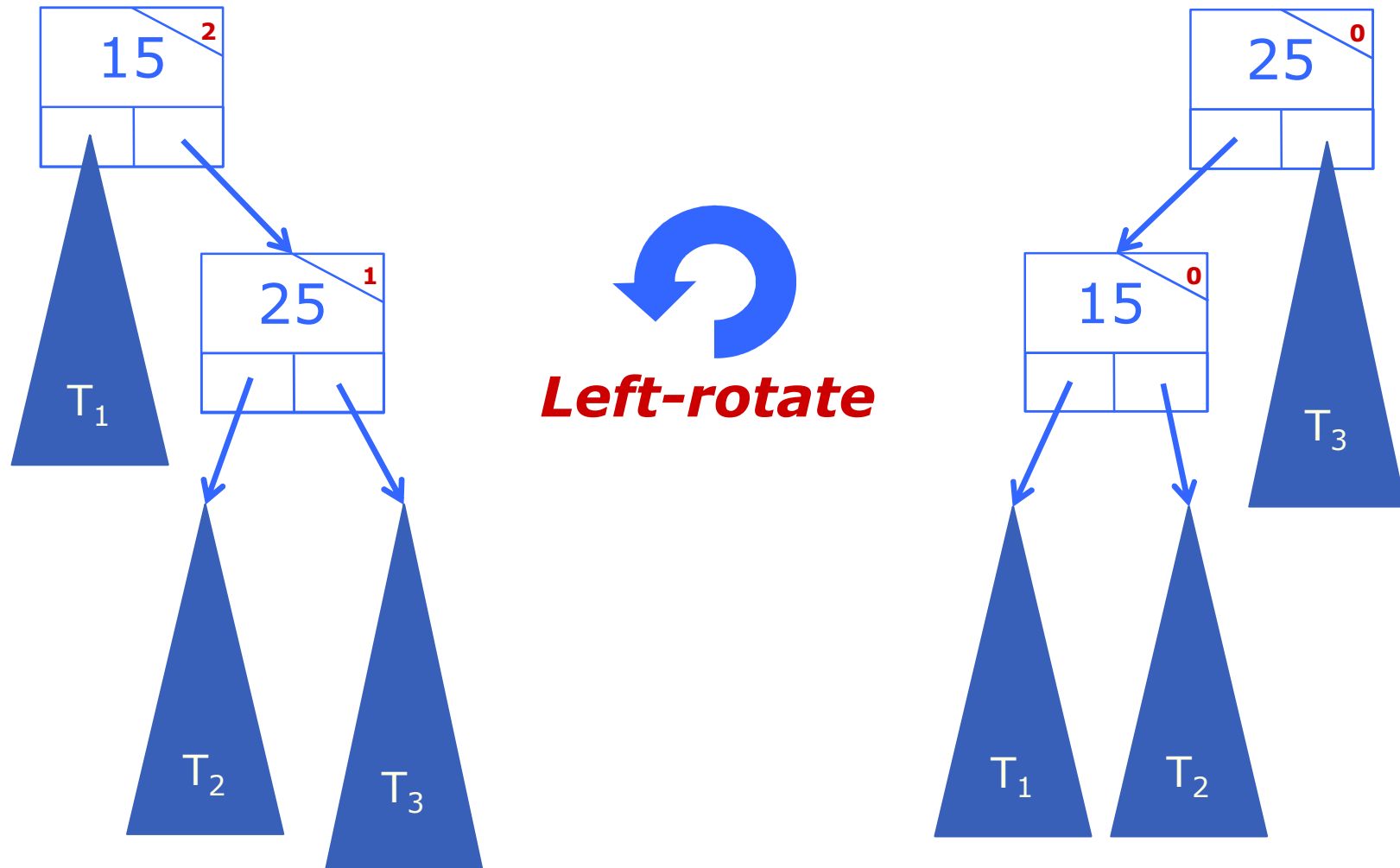
Always Maintain Balance

- With *every* insertion and deletion, ensure tree is balanced
- Trickier (and more challenging)
- Guarantees use $O(\lg n)$ insertion and fetch

AVL Trees

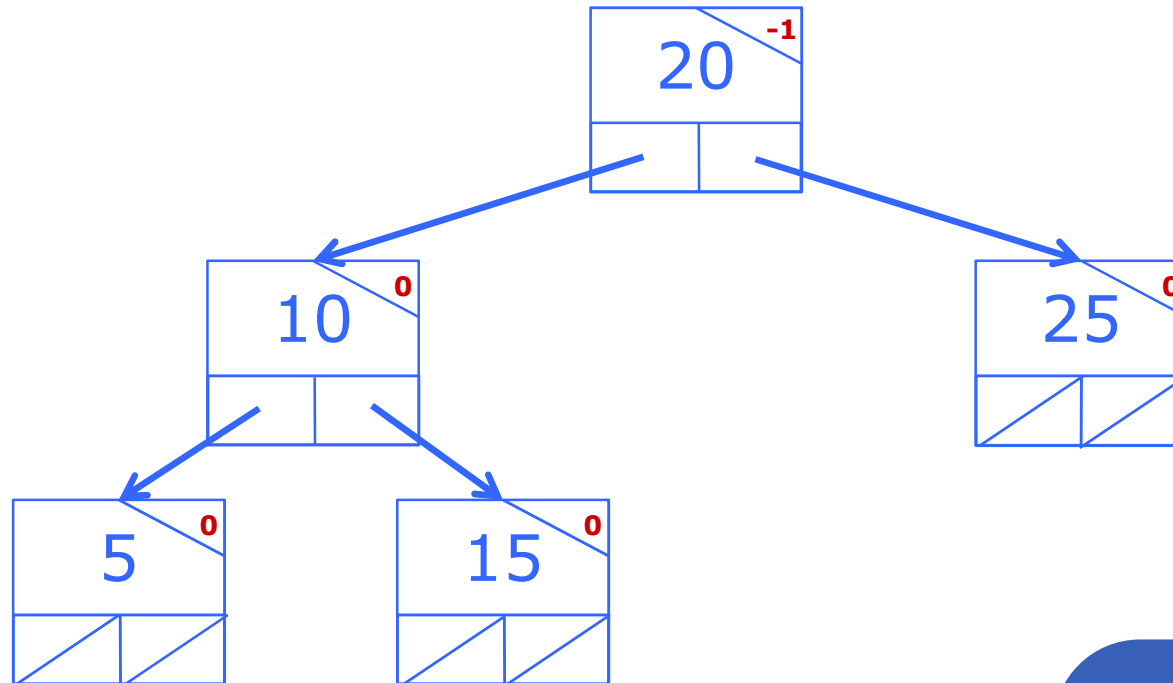
- Invariant: the heights of any two subtrees of a node differs by at most one.
- With each node, track a "balance factor" (BF)
`node->bf = treeHeight(node->right) - treeHeight(node->left)`
- BF values of -1, 0, 1 are fine (carry on as usual)
- Other BF values requires tree to be fixed by rotating.

Rotation Example



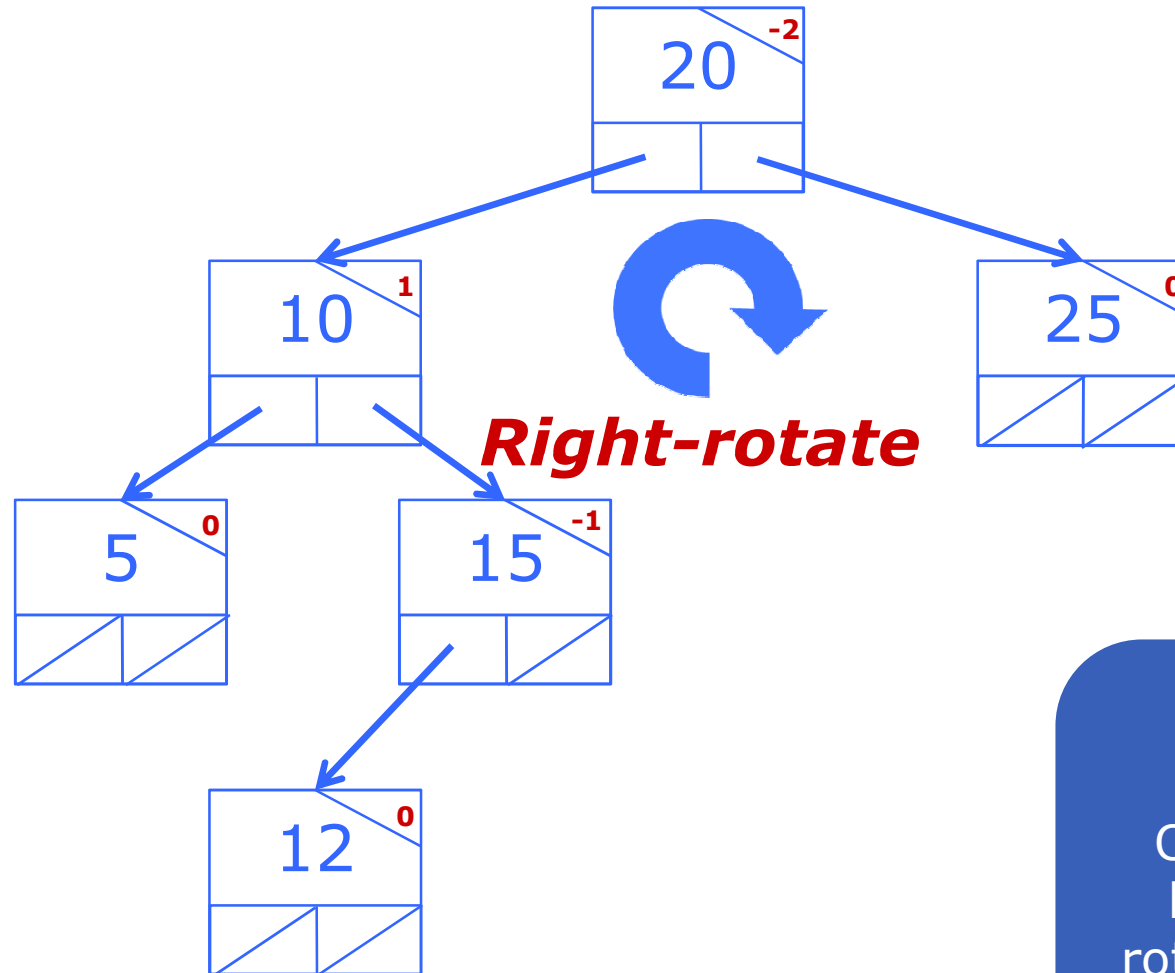
$$\text{treeHeight}(T_1) == \text{treeHeight}(T_2) == \text{treeHeight}(T_3) + 1$$

Another Rotation Example



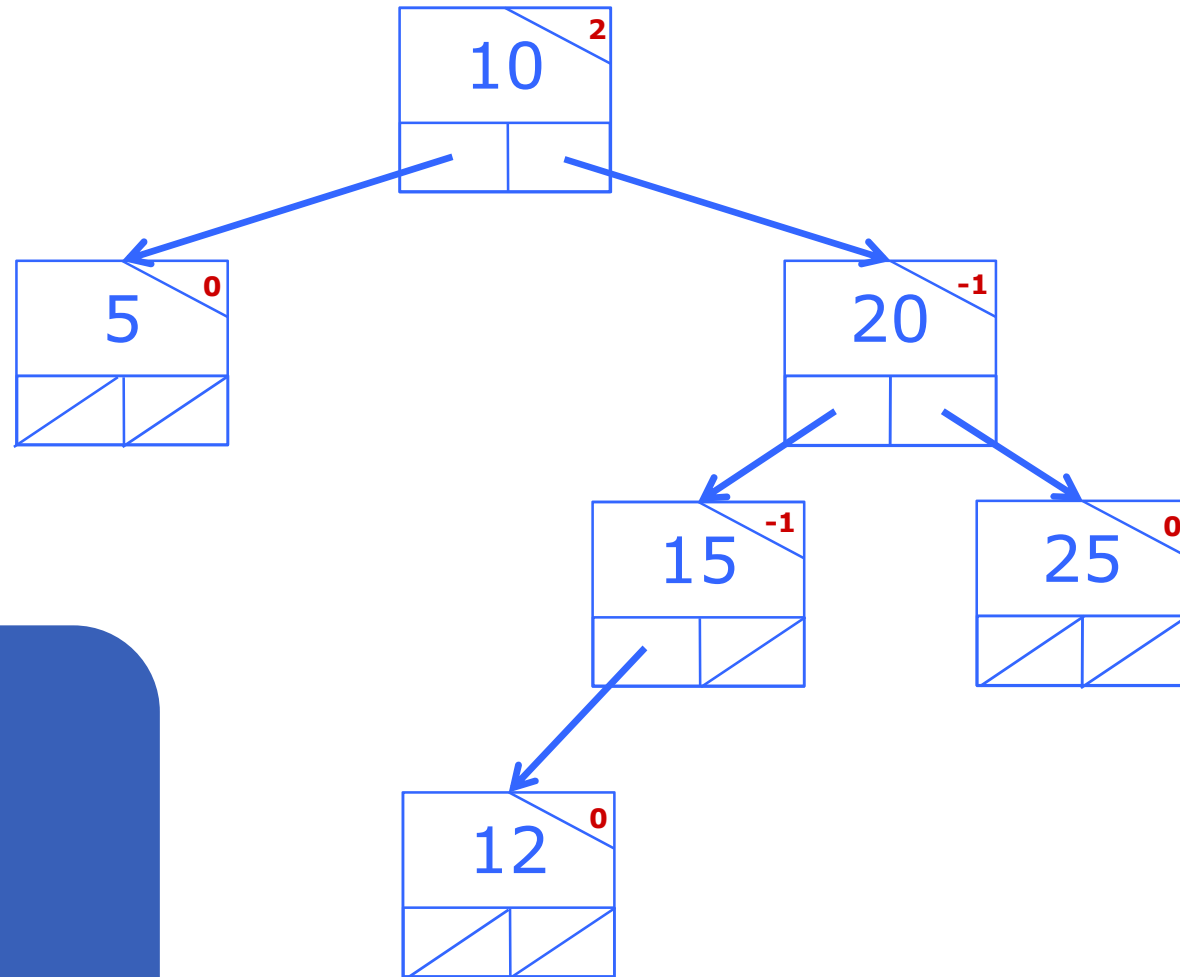
Let's go ahead and
add 12 to the mix

Another Rotation Example



Oops. We're not balanced. Let's rotate to the right!

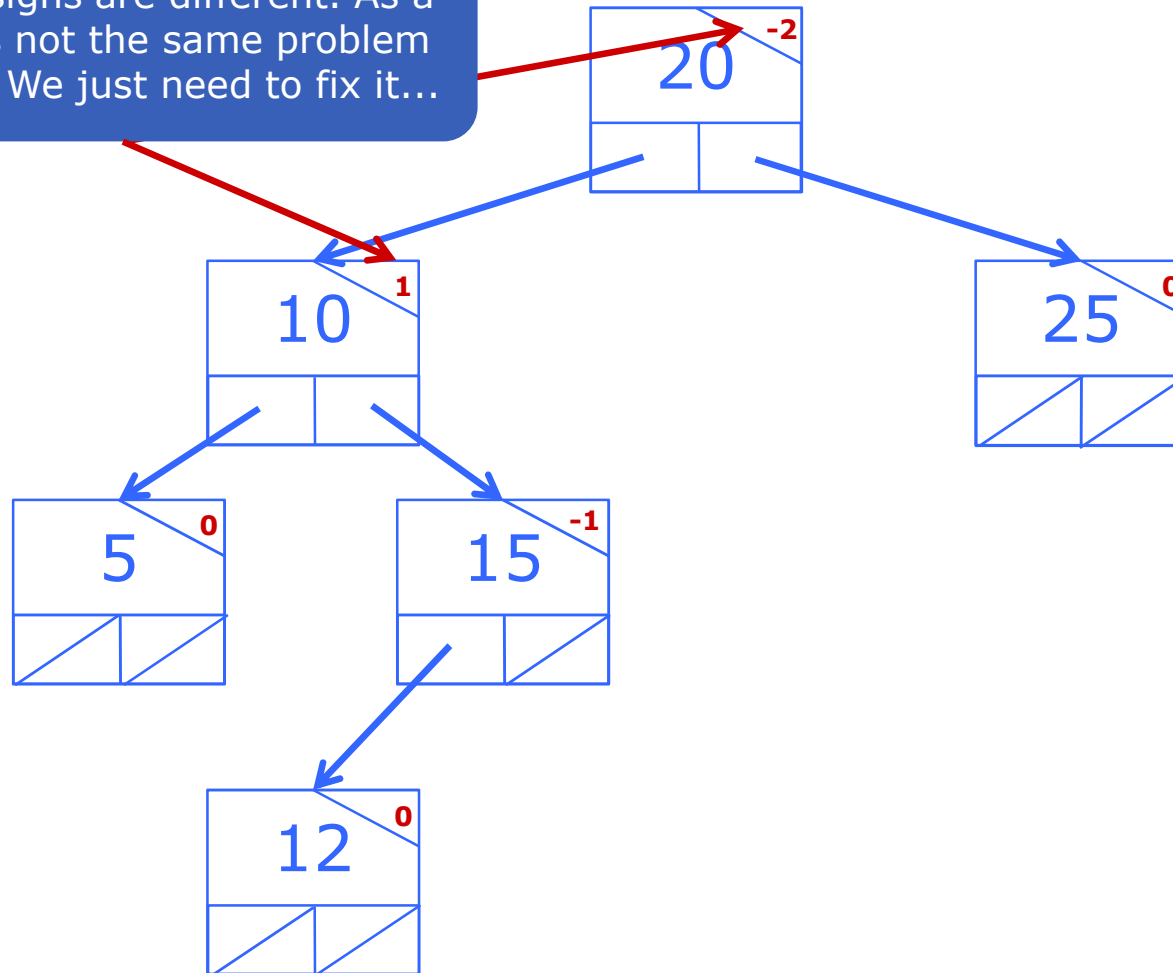
Another Rotation Example



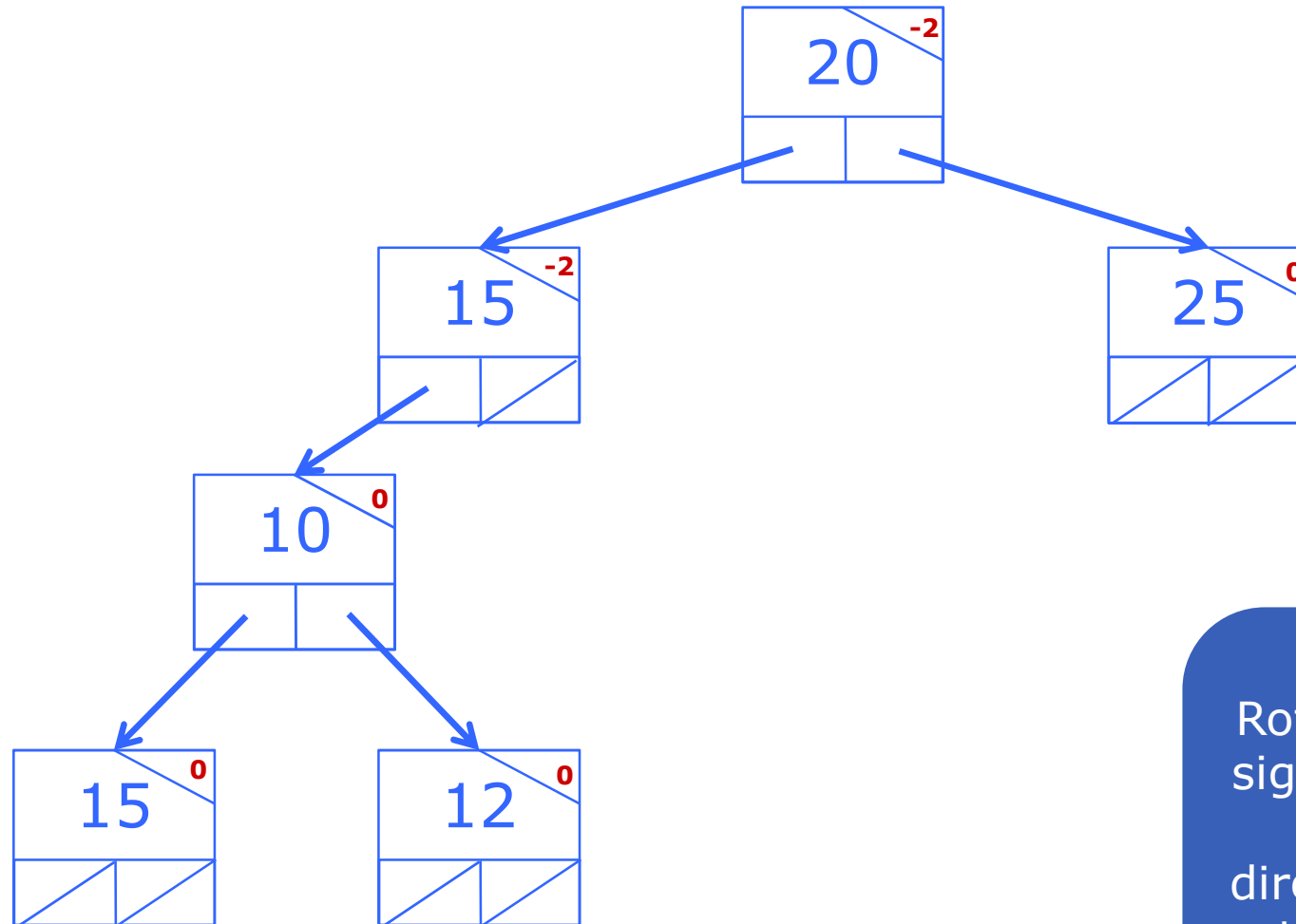
Yeah, um
EPIC FAIL!

Take a Step Back

Note the signs are different. As a result, it's not the same problem as before. We just need to fix it...

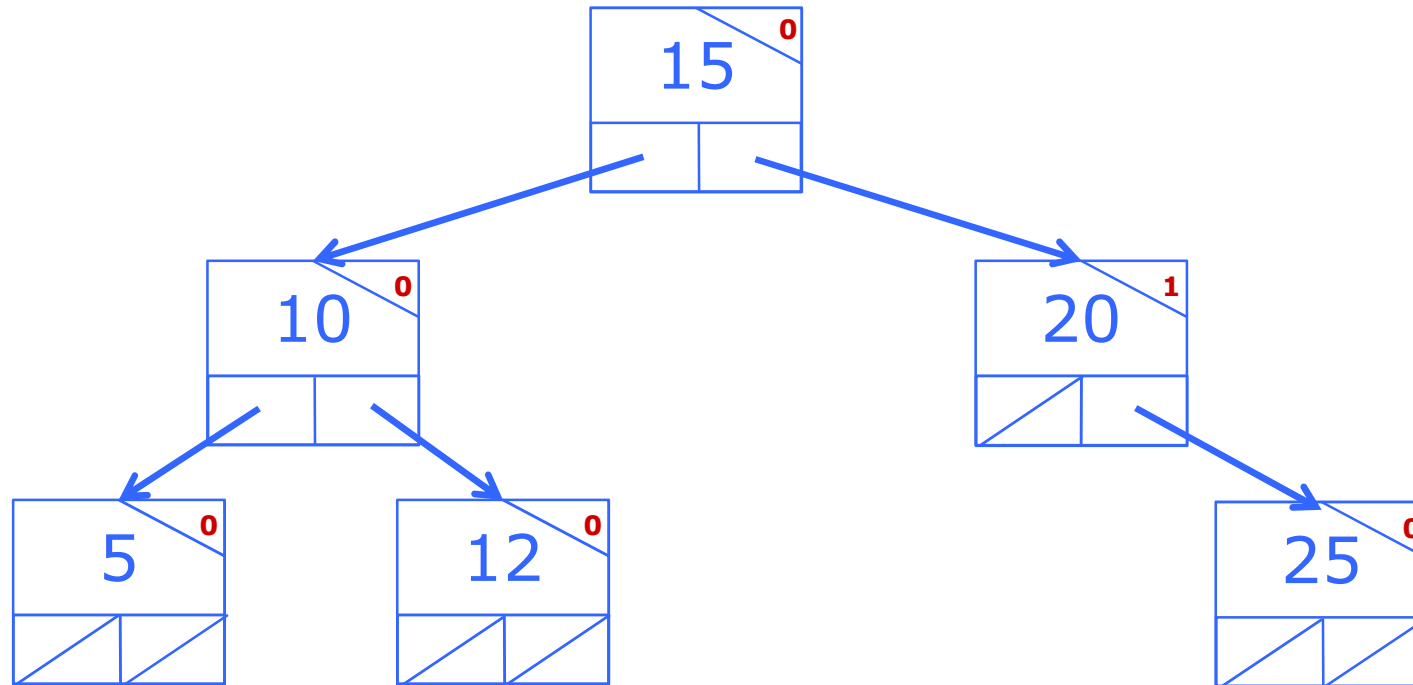


Double Rotations



Rotate the opposite signed child first (in the opposite direction) to get the tree in the right form, then...

Double Rotations



```
#ifndef _bst_h
#define _bst_h

#include "cmpfn.h"
#include "vector.h"

template <typename ElemType>
class BST {
public:
    BST(int (cmpFn)(ElemType one, ElemType two) = OperatorCmp);
    ~BST();
    int size();
    bool isEmpty();
    ElemType *find(ElemType key);
    bool add(ElemType elem);
    bool remove(ElemType key);
    void clear();
};
```

```
template <typename ElemType>
int OperatorCmp(Type one, Type two)
{
    if (one == two) return 0;
    else if (one < two) return -1;
    else return 1;
}
```

```

private:
    struct nodeT {
        ElemType data;
        nodeT *left, *right;
        int bf; // AVL balance factor
    };

    nodeT *root;
    int numNodes;
    int (*cmpFn)(ElemType, ElemType);

    static const int RightHeavy = +1;
    static const int Even = 0;
    static const int LeftHeavy = -1;

    nodeT *recFindNode(nodeT * t, ElemType & key);
    bool recAddNode(nodeT * & t, ElemType & key, bool &createdNewNode);
    bool recRemoveNode(nodeT * &, ElemType & key, bool & didRemove);
    bool removeTargetNode(nodeT * & t);
    void updateBF(nodeT * &t, int bfDelta);
    void recDeleteTree(nodeT * t);
    void fixRightImbalance(nodeT * &t);
    void fixLeftImbalance(nodeT * &t);
    void rotateRight(nodeT * &t);
    void rotateLeft(nodeT * &t);
};

#endif

```

```
template <typename ElemType>
BST<ElemType>::BST(int (cmp)(ElemType, ElemType))
{
    root = NULL;
    cmpFn = cmp;
    numNodes = 0;
}
```

```
template <typename ElemType>
BST<ElemType>::~~BST()
{
    recDeleteTree(root);
}
```

```
template <typename ElemType>
void BST<ElemType>::recDeleteTree(nodeT * t)
{
    if (t != NULL) {
        recDeleteTree(t->left);
        recDeleteTree(t->right);
        delete t;
    }
}
```

```

template <typename ElemType>
bool BST<ElemType>::recAddNode(nodeT * & t, ElemType & data, bool &createdNewNode)
{
    if (t == NULL) {
        t = new nodeT;
        t->data = data; t->bf = Even; t->left = t->right = NULL;
        createdNewNode = true;
        numNodes++;
        return (true);
    }

    int sign = cmpFn(data, t->data);
    if (sign == 0) {
        t->data = data;
        createdNewNode = false;
        return (false);
    }

    int bfDelta = 0;
    if (sign < 0) {
        if (recAddNode(t->left, data, createdNewNode))
            bfDelta = -1; // inserted node has caused increase in left subtree height
    } else {
        if (recAddNode(t->right, data, createdNewNode))
            bfDelta = +1; // inserted node has caused increase in right subtree height
    }
    updateBF(t, bfDelta); // adds height change and rebalances if necessary
    // height of t increased iff subtree increased and this node ended heavy
    return (bfDelta != 0 && t->bf != Even);
}

```

```
template <typename ElemType>
void BST<ElemType>::updateBF(nodeT * & t, int bfDelta)
{
    //add height change to existing balance factor
    t->bf += bfDelta;
    if (t->bf < LeftHeavy)
        fixLeftImbalance(t);
    else if (t->bf > RightHeavy)
        fixRightImbalance(t);
}
```

```
template <typename ElemType>
void BST<ElemType>::updateBF(nodeT * & t, int bfDelta)
{
    //add height change to existing balance factor
    t->bf += bfDelta;
    if (t->bf < LeftHeavy)
        fixLeftImbalance(t);
    else if (t->bf > RightHeavy)
        fixRightImbalance(t);
}
```

```

template <typename ElemType>
template <typename ElemType>
void BST<ElemType>::fixLeftImbalance(nodeT * & t)
{
    nodeT * child = t->left;

    if (child->bf == RightHeavy) {
        int oldBF = child->right->bf;
        rotateLeft(t->left);
        rotateRight(t);
        t->bf = Even;
        switch (oldBF) {
            case LeftHeavy: t->left->bf = Even; t->right->bf =
RightHeavy; break;
            case Even: t->left->bf = t->right->bf = Even; break;
            case RightHeavy: t->left->bf = LeftHeavy; t->right->bf =
Even; break;
        }
    } else if (child->bf == Even) {
        rotateRight(t);
        t->bf = RightHeavy;
        t->right->bf = LeftHeavy;
    } else {
        rotateRight(t);
        t->right->bf = t->bf = Even;
    }
}

```

```
template <typename ElemType>
template <typename ElemType>
void BST<ElemType>::rotateLeft(nodeT * & t)
{
    nodeT * child = t->right;
    t->right = child->left;
    child->left = t;
    t = child;
}
```

BST – AVL Trees

- We're guaranteed to have $O(\lg n)$ insertion and look-ups
- Other implementations focus on making particular operations happen faster
 - AVL – Look-ups
 - Red-Black Trees – Insertions
- Which implementation to use? Depends on what you need it for.