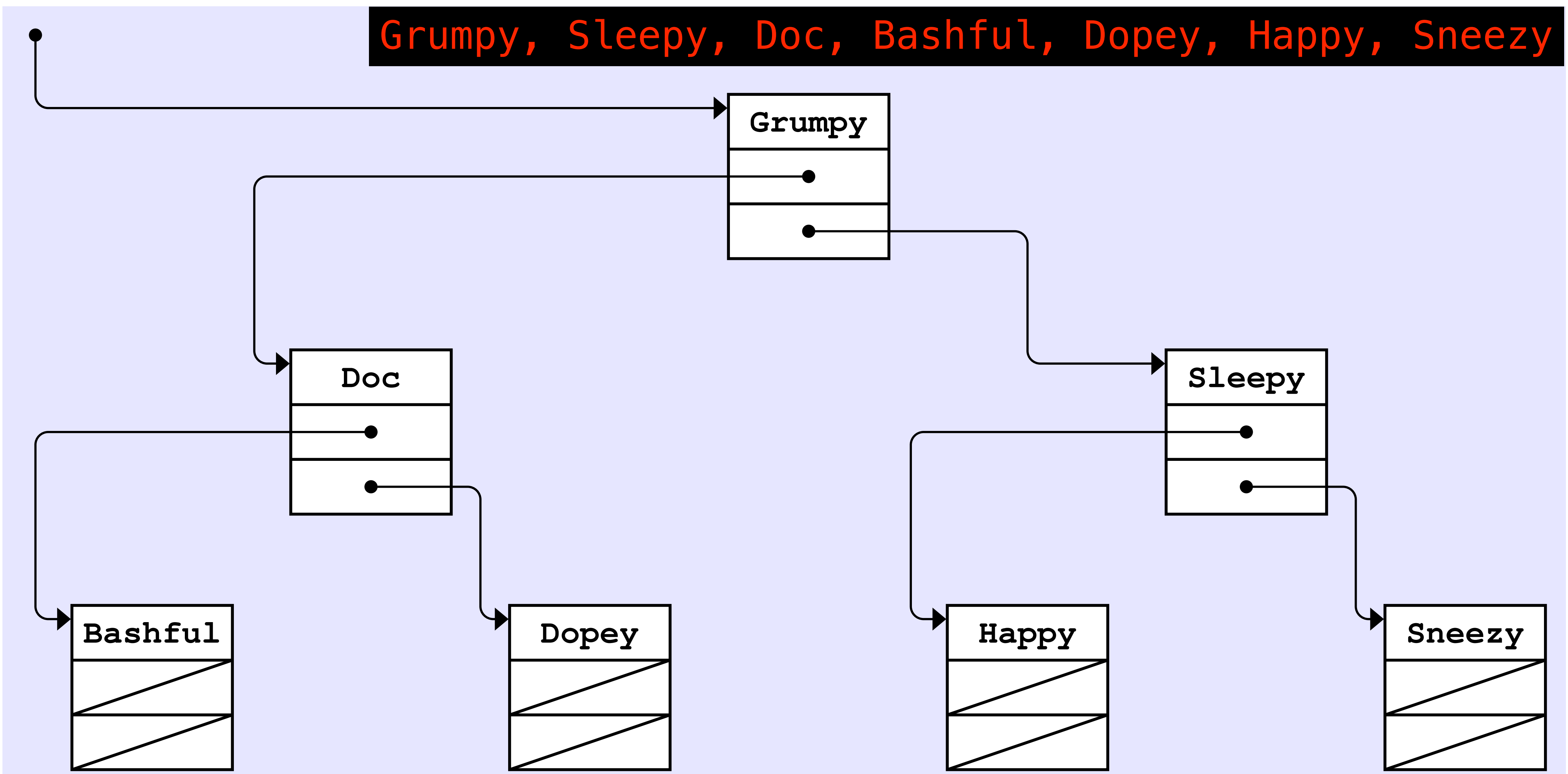
ALGORITHMS AND DATA STRUCTURES

BINARY TREES

BALANCED TREES

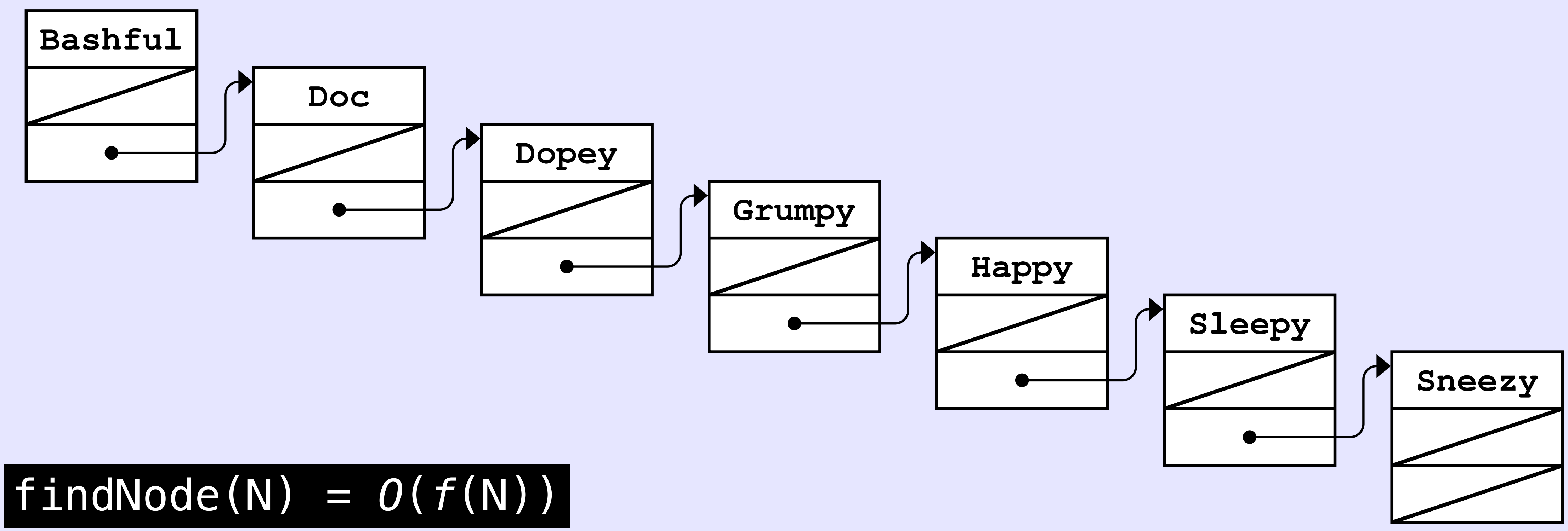
- The order of insertion matters



BALANCED TREES

- The order of insertion matters

`Bashful, Doc, Dopey, Grumpy, Happy, Sleepy, Sneezy`

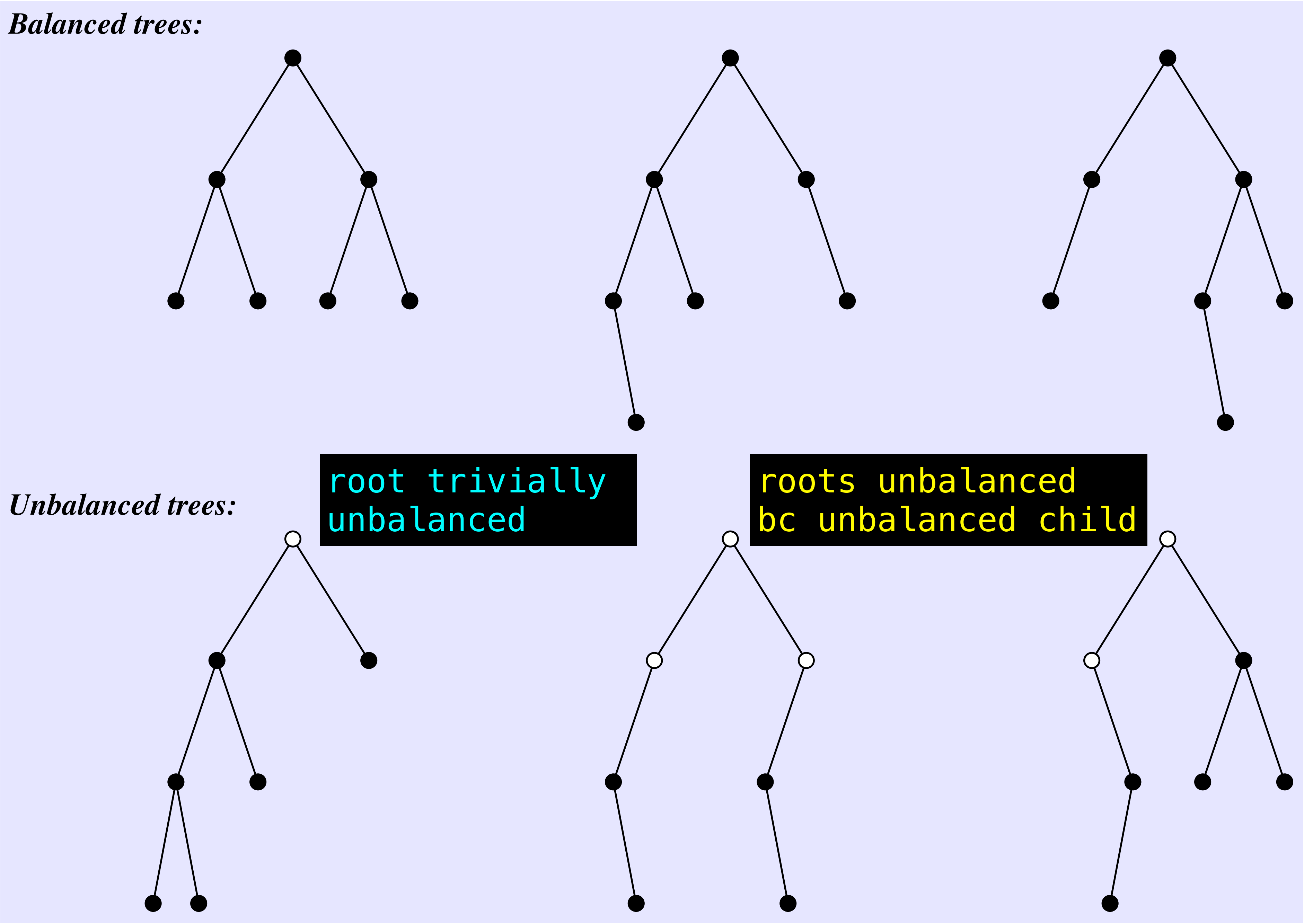


`findNode(N) = O(f(N))`

BALANCED TREES: DEFINITION

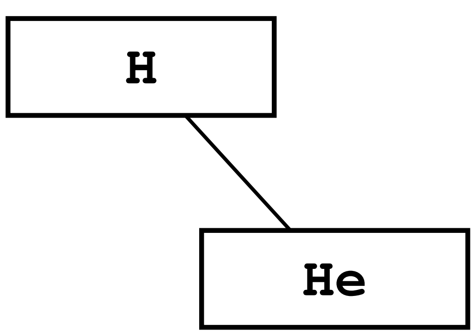
- Structure of tree impacts algorithmic performance
- Ideal performance: same size for left and right subtrees
    - these are ***balanced trees***
- A binary tree is balanced if, at each node, the heights of the left and right subtrees differ by at most one
    - recursive definition
- A binary tree is perfectly/optimally balanced if the heights of the two subtrees at each node are equal

BALANCED TREES: EXAMPLES

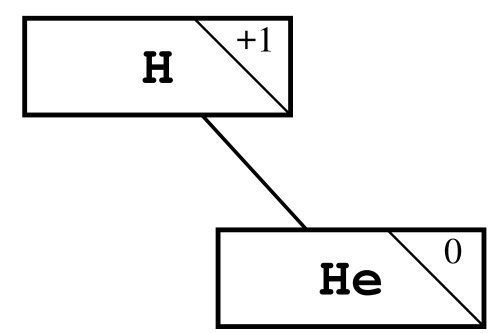


TREE-BALANCING STRATEGIES: AVL TREES

- *Example*: Suppose you want to arrange the periodic table as a BST
    - Consider the fist six elements: H, He, Li, Be, B, C
    - Inserting in that order we get

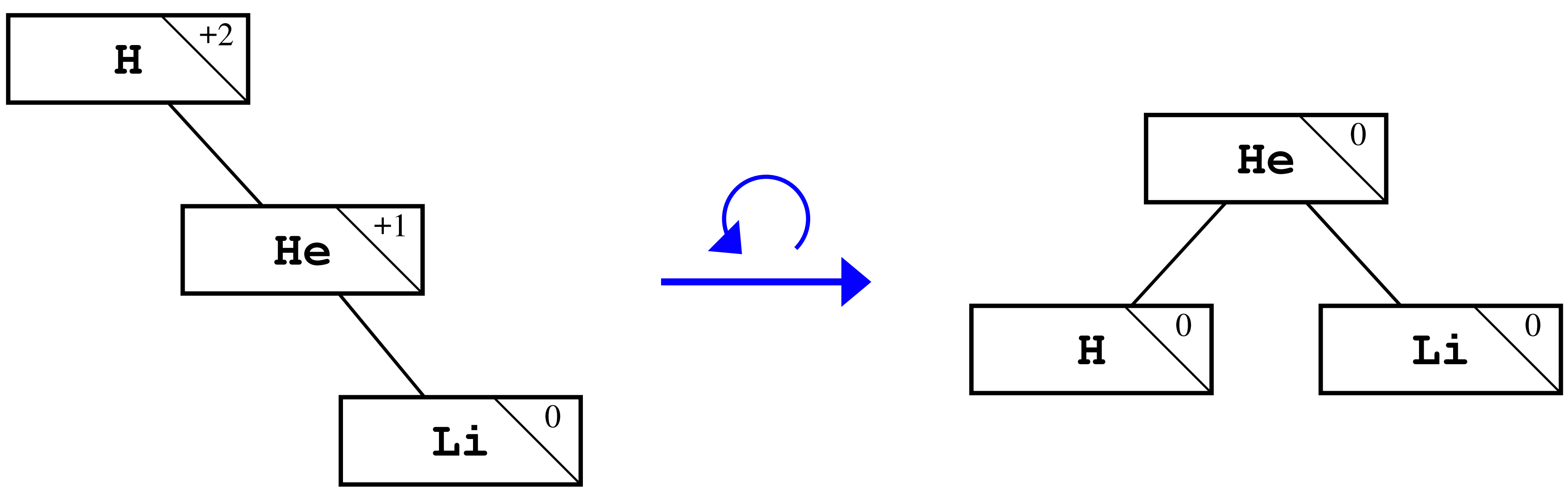


balance factor
(AVL trees)
Adelson-Velskii and Landis



TREE-BALANCING STRATEGIES: AVL TREES

- *Balance factor = height left subtree – height right subtree*



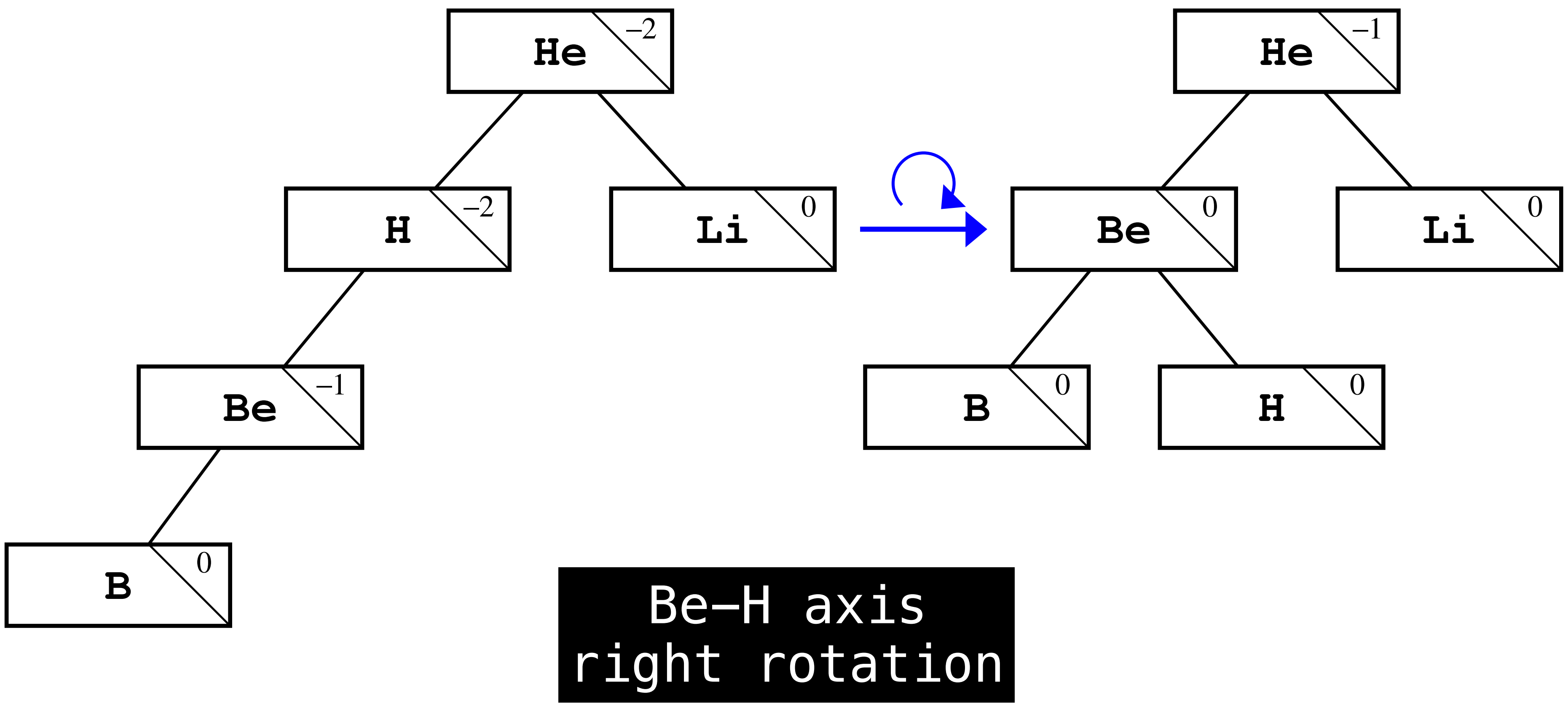
root of the node out of balance

only one balanced configuration

- What operations to perform to balance a tree?

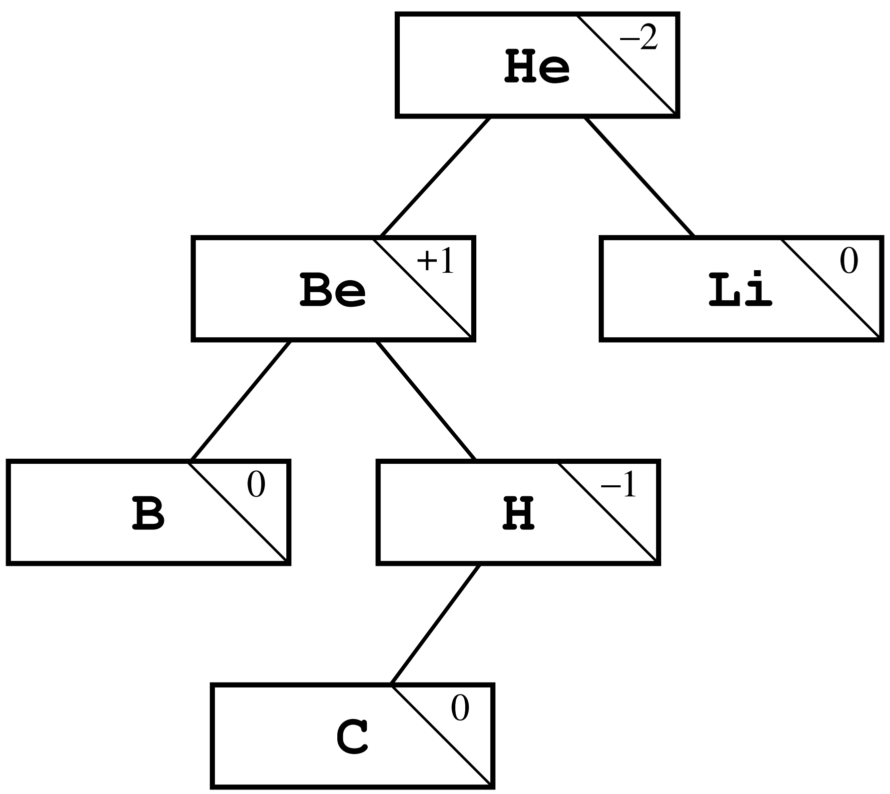
TREE-BALANCING STRATEGIES: AVL TREES

- Rotations around an axis (the H-He axis)
  - left and right rotations



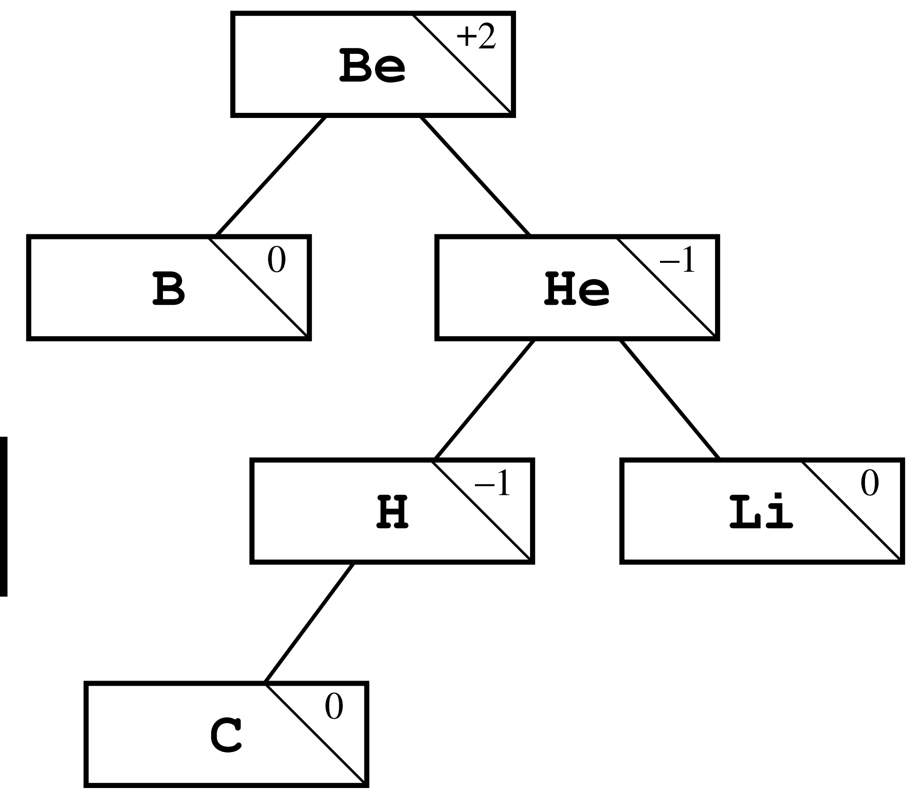
TREE-BALANCING STRATEGIES: AVL TREES

- Not all rotations are simple



unbalanced tree

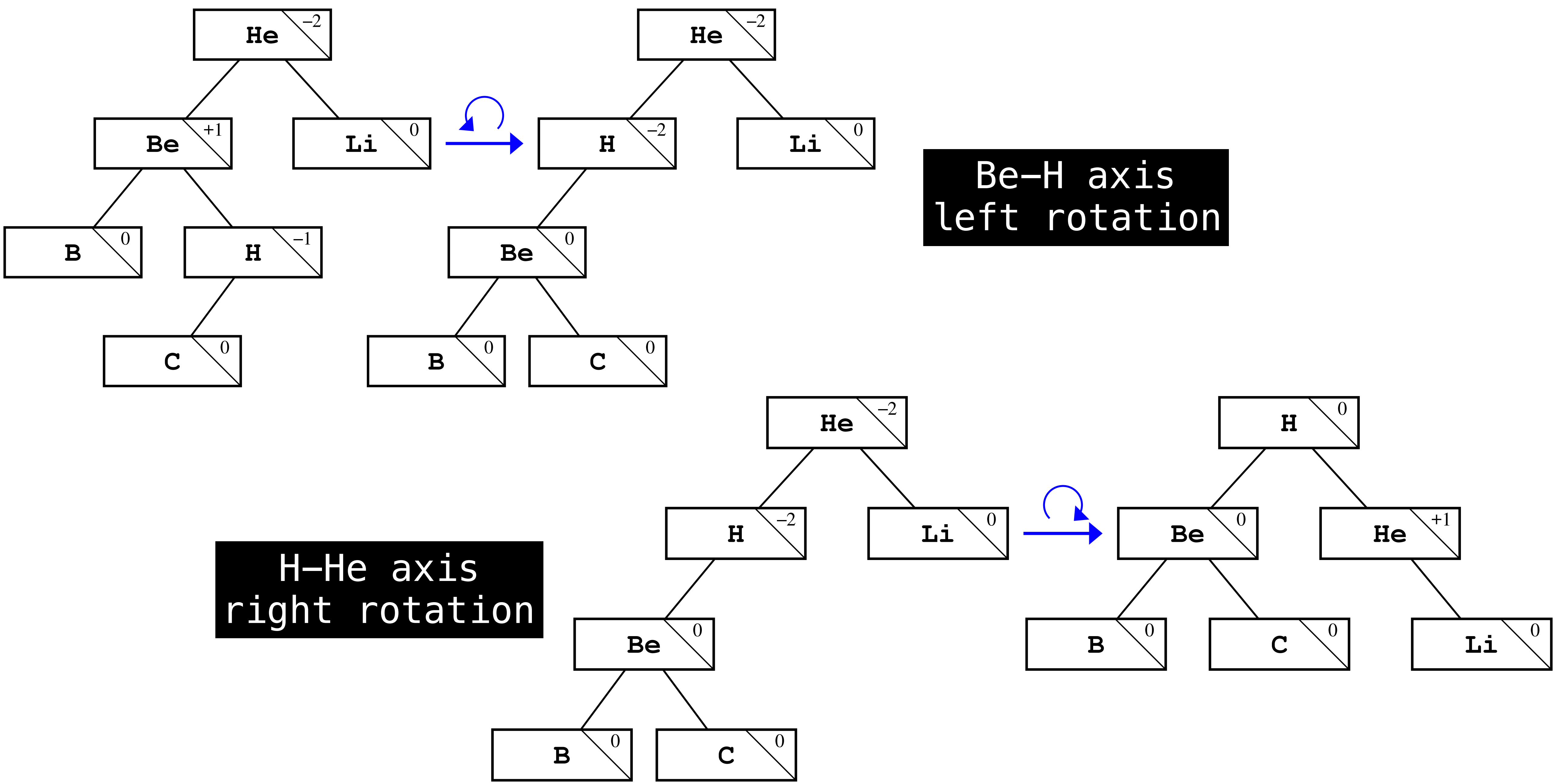
Be–He axis
right rotation



unbalanced tree

PROBLEM:
OPPOSITE SIGN

TREE-BALANCING STRATEGIES: AVL TREES



TREE-BALANCING STRATEGIES: AVL TREES

- ***Double rotation***:
  1. rotate child in the opposite direction
  2. rotate parent (now with same sign for balance factor)
- Properties of AVL trees
  - After inserting a new node, balance can be restored by performing at most one operation: single or double rotation
  - After a rotation, the height of the subtree at the axis of rotation is always the same as it was before inserting the new node

IMPLEMENTING THE AVL ALGORITHM

A. Augment the structure BSTNode to account for the balance factor

B. Keep track of the height of the tree after each insertion. Consider three situations when inserting in a subtree

   1. Subtree was previously shorter than the other subtree in this node

   2. The two subtrees in the current node were previously the same size

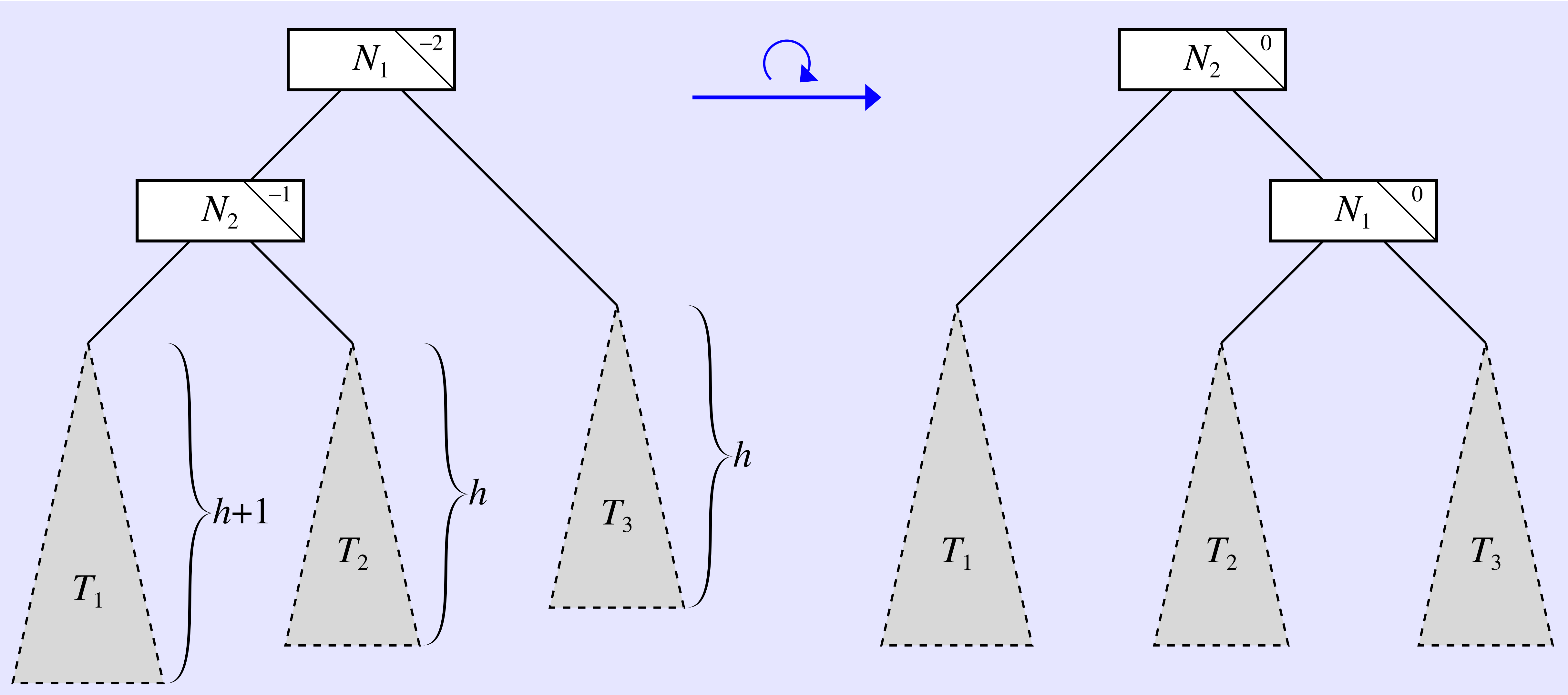
   3. The subtree that grew taller was already taller than the other subtree

IMPLEMENTING THE AVL ALGORITHM

1. Inserting the node makes tree more balanced
    - `bf = 0`; height remains the same
2. Increases the size of one of the subtrees. Slightly out of balance, no rotations are required
    - `bf = ±1`; height increases by 1
3. The tree is now out of balance, because one subtree is two nodes taller than the other
   Execute appropriate rotations and correct balance factors

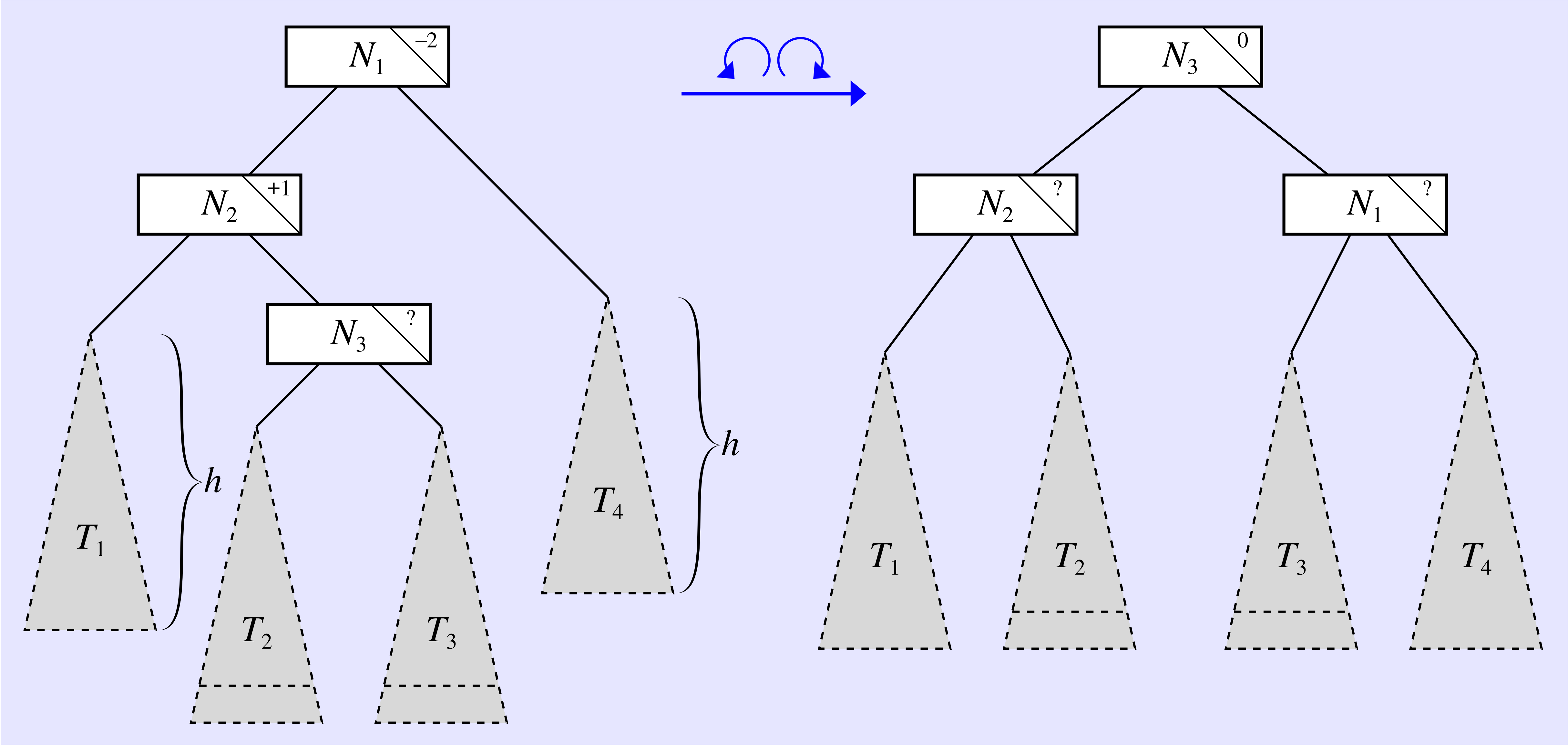
IMPLEMENTING THE AVL ALGORITHM

- Single rotation



IMPLEMENTING THE AVL ALGORITHM

- Double rotation



```
struct BSTNode {
    std::string str;
    BSTNode *left;
    BSTNode *right;
    int bf;
};

void insertNode(BSTNode * & t, string key) {
    insertAVL(t, key);
}
```

```
int insertAVL(BSTNode * & t, string key) {
    if (t == nullptr) {
        t = new BSTNode;
        t->key = key;
        t->bf = 0;
        t->left = t->right = nullptr;
        return +1;
    }
    if (key == t->key) return 0;
    if (key < t->key) {
        int delta = insertAVL(t->left, key);
        if (delta == 0) return 0;
        switch (t->bf) {
            case +1: t->bf =  0; return 0;
            case  0: t->bf = -1; return +1;
            case -1: fixLeftImbalance(t); return 0;
        }
    } else { /* same for right insertion */ }
}
```

```
void fixRightImbalance(BSTNode * & t) {
    BSTNode *child = t->right;
    if (child->bf != t->bf) {
        int oldBF = child->left->bf;
        rotateRight(t->right);
        rotateLeft(t);
        t->bf = 0;
        switch (oldBF) {
            case -1: t->left->bf = 0; t->right->bf = +1;
                     break;
            case  0: t->left->bf = t->right->bf = 0;
                     break;
            case +1: t->left->bf = -1; t->right->bf = 0;
                     break;
        }
    } else {
        rotateLeft(t);
        t->left->bf = t->bf = 0;
    }
}
```

```
void rotateRight(BSTNode * & t) {
    BSTNode *child = t->left;
    if (DEBUG) {
        cout << "rotateRight(" << t->key << "-"
             << child->key << ")" << endl;
    }
    t->left = child->right;
    child->right = t;
    t = child;
}
```