Balanced Trees: AVL Trees

Maintaining O(log n) Performance

Prof. Antonio Khalil Moretti

SCIS 313: Data Structures and Algorithm Analysis

Spelman College

The Problem: Unbalanced BSTs

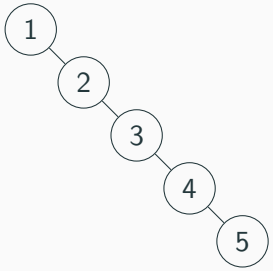
Recall: BST operations are O(h) where h = height

Best case (balanced): h = O(log n)

Worst case (degenerate): h = O(n)

Example: Insert 1, 2, 3, 4, 5 in order

Result: Linked list!



Height = 5 = n

The Question:

Can we *guarantee* O(log n) height?

The Solution:

Balanced Trees!

Keep tree "balanced" automatically through rotations.

Today: AVL Trees

- First balanced BST (1962)
- Named after inventors: Adelson-Velsky and Landis
- Guarantee h = O(log n)

What Does "Balanced" Mean?

Height of a node:
Length of longest path to a leaf

Balance Factor:

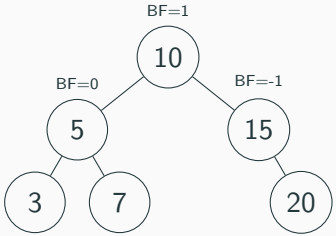
$$BF(node) = height(left) - height(right)$$

AVL Property:
For **every** node:

$$|BF(node)| \leq 1$$

That is: $BF \in \{-1, 0, 1\}$

Example AVL Tree:



All balance factors: $\{-1, 0, 1\}$ ✓

Why Does AVL Property Give O(log n) Height?

Key insight: With AVL property, tree cannot be too unbalanced

Minimum nodes for height h:
Let $N(h) =$ min nodes in AVL tree of height h

Recurrence:

$$N(0) = 1$$
$$N(1) = 2$$
$$N(h) = N(h-1) + N(h-2) + 1$$

This is similar to Fibonacci!

Solution: $N(h) \approx 1.618^h$

Therefore: If tree has n nodes and height h:

$$n \geq N(h) \approx 1.618^h$$

Taking logarithms:

$$\log_2 n \geq h \cdot \log_2(1.618)$$

Therefore:

$$h \leq \frac{\log_2 n}{\log_2(1.618)} = \frac{1}{0.694} \log_2 n \approx 1.44 \log_2 n$$

Height is O(log n)! More precisely: $h < 1.44 \log n$
This means AVL trees are never more than 44% taller than perfectly balanced.

AVL Tree Operations

Same as BST, but maintain AVL property after modifications

Operation	BST	AVL
Search	O(h)	O(log n)
Insert	O(h)	O(log n)
Delete	O(h)	O(log n)
Height	O(n) worst	O(log n) always

The challenge: How do we maintain balance?

The tool: **Rotations**

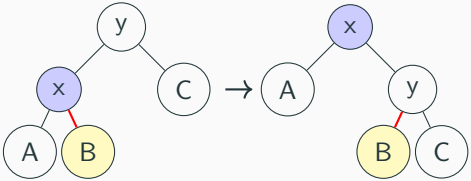
Tree Rotations: The Key Idea

Rotation: Promote a child to parent, demote parent to child, preserve BST order

Mirror operations: Right = fix left-heavy, Left = fix right-heavy

Right Rotation on y:

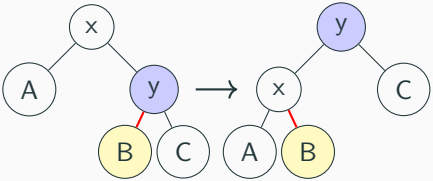
Before: y root, x left child



After: x promoted, B moves to y

Left Rotation on x:

Before: x root, y right child



After: y promoted, B moves to x

Key: Order preserved ($A < x < B < y < C$), middle subtree B changes parent, O(1) time

When Do We Need to Rotate?

After insert/delete: Balance factor might become ± 2

Four cases based on where imbalance occurs:

Case	Imbalance Pattern	Fix
LL	Left-Left (heavy on left of left)	Right rotation
RR	Right-Right (heavy on right of right)	Left rotation
LR	Left-Right (heavy on right of left)	Left then Right
RL	Right-Left (heavy on left of right)	Right then Left

Strategy:

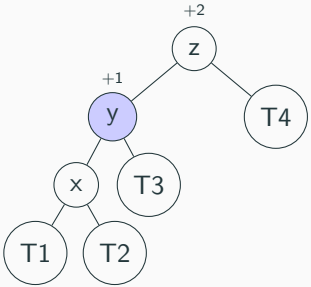
1. Insert/delete as in normal BST
2. Walk back up to root, checking balance factors
3. If $|BF| = 2$, identify case and rotate
4. One rotation fixes the imbalance!

Case 1: Left-Left (LL)

Heavy on left side of left child

Problem: z has BF $= +2$

Left child y has BF ≥ 0

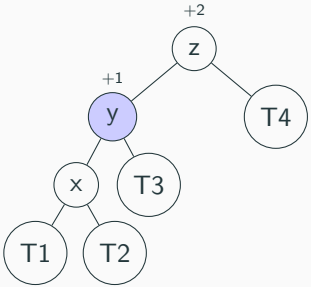


Too heavy left-left

Case 1: Left-Left (LL)

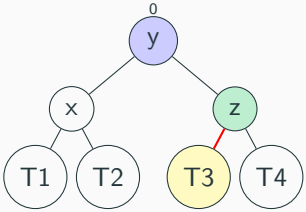
Heavy on left side of left child

Problem: z has BF $= +2$
Left child y has BF ≥ 0



Too heavy left-left

Solution: Right rotation on z
y promoted, z demoted

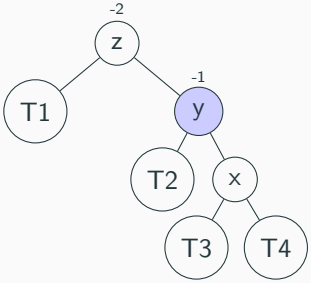


Balanced! T3 moved to z

Case 2: Right-Right (RR)

Heavy on right side of right child

Problem: z has BF = -2
Right child y has BF $\leq$ 0

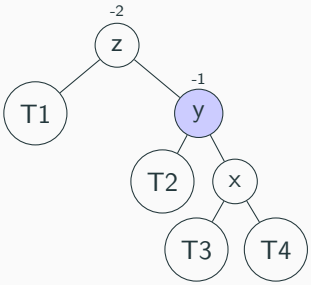


Too heavy right-right

Case 2: Right-Right (RR)

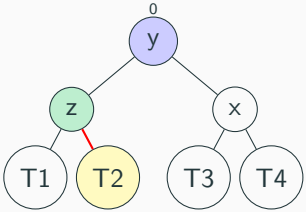
Heavy on right side of right child

Problem: z has BF = -2
Right child y has BF ≤ 0



Too heavy right-right

Solution: Left rotation on z
y promoted, z demoted



Balanced! T2 moved to z
(Mirror of LL)

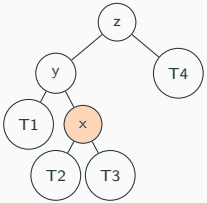
Case 3: Left-Right (LR)

Heavy on right side of left child - requires two rotations

Problem:

z: BF $= +2$

Left child: BF < 0



Zig-zag

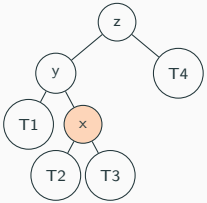
Case 3: Left-Right (LR)

Heavy on right side of left child - requires two rotations

Problem:

z: BF $= +2$

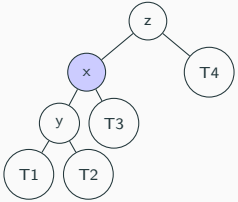
Left child: BF < 0



Zig-zag

Step 1:

Left rotate y



Now LL!

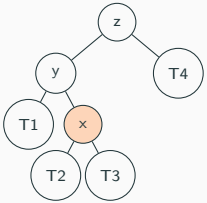
Case 3: Left-Right (LR)

Heavy on right side of left child - requires two rotations

Problem:

z: BF $= +2$

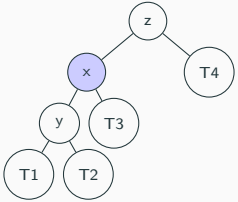
Left child: BF < 0



Zig-zag

Step 1:

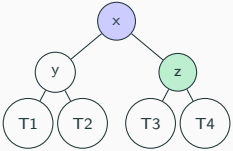
Left rotate y



Now LL!

Step 2:

Right rotate z



Balanced!

Two rotations: Left on y, then right on z

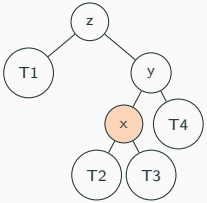
Case 4: Right-Left (RL)

Heavy on left side of right child - requires two rotations

Problem:

z: BF = -2

Right child: BF > 0



Zig-zag

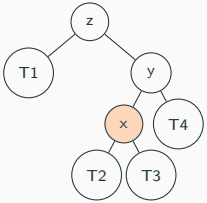
Case 4: Right-Left (RL)

Heavy on left side of right child - requires two rotations

Problem:

z: BF = -2

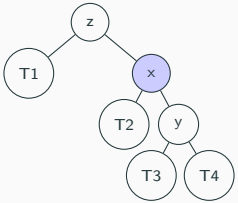
Right child: BF > 0



Zig-zag

Step 1:

Right rotate y



Now RR!

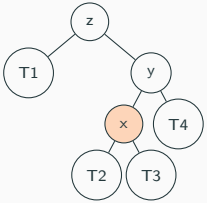
Case 4: Right-Left (RL)

Heavy on left side of right child - requires two rotations

Problem:

z: BF = -2

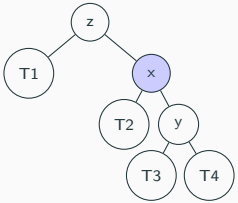
Right child: BF > 0



Zig-zag

Step 1:

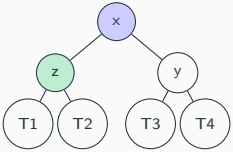
Right rotate y



Now RR!

Step 2:

Left rotate z



Balanced!

Two rotations: Right on y, then left on z (mirror of LR)

Summary: Four Rebalancing Cases

Case	Pattern	Rotations	When
LL	Left-Left	Right(z)	BF(z)=+2, BF(left)$\geq$0
RR	Right-Right	Left(z)	BF(z)=-2, BF(right)$\leq$0
LR	Left-Right	Left(left), Right(z)	BF(z)=+2, BF(left)$<$0
RL	Right-Left	Right(right), Left(z)	BF(z)=-2, BF(right)$>$0

Key insights:

- LL and RR: Single rotation (straight line)
- LR and RL: Double rotation (zig-zag)
- All cases: O(1) time for rotation
- Only need to check from inserted node up to root

AVL Insert Algorithm

Step-by-step:

1. Insert as normal BST
2. Walk back to root
3. Update heights
4. Check balance factors
5. If $|BF| = 2$, rotate
6. Continue to root

Time complexity:

- Insert: O(log n)
- Walk up: O(log n)
- Rotations: O(1) each

Total: O(log n)

```
Node* insert(Node* node, int key) {
    // 1. Normal BST insert
    if (node == NULL)
        return new Node(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);
    // 2. Update height
    node->height = 1 + max(height(node->left),
                           height(node->right));
    // 3. Get balance factor
    int balance = getBalance(node);
    // 4. Check four cases
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);       // LL
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);        // RR
    if (balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);       // LR
    }
    if (balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);        // RL
    }

    return node;
}
```

Implementing Rotations

Right Rotation:

```
Node* rightRotate(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left),
                    height(y->right)) + 1;
    x->height = max(height(x->left),
                    height(x->right)) + 1;

    return x; // New root
}
```

Just pointer updates - O(1)!

Left Rotation:

```
Node* leftRotate(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left),
                    height(x->right)) + 1;
    y->height = max(height(y->left),
                    height(y->right)) + 1;

    return y; // New root
}
```

Mirror of right rotation

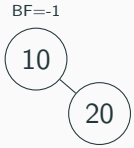
Key: Update heights after rotation!

Example: Insert 10, 20, 30

Insert 10:

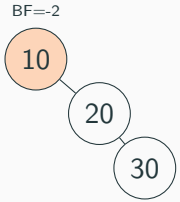
Balanced

Insert 20:



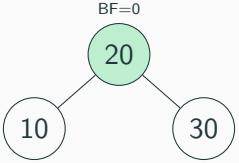
Still balanced

Insert 30:



Imbalanced! RR

After left rotation on 10:



Balanced!

AVL Delete

Similar to insert:

1. Delete as normal BST
2. Walk back to root
3. Update heights
4. Check balance factors and rotate if needed

Key difference: May need multiple rotations along path to root

Time complexity: Still O(log n)

Note: Delete is more complex to implement correctly, but same principles apply

AVL Trees vs Regular BST

	Regular BST	AVL Tree
Search	O(n) worst	O(log n) always
Insert	O(n) worst	O(log n) always
Delete	O(n) worst	O(log n) always
Extra space per node	0	O(1) (height)
Rebalancing needed	No	Yes
Implementation complexity	Simple	Moderate
Best for	Random data	Any data

When to use AVL:

- Need guaranteed O(log n)
- Frequent searches
- Can't control insertion order

Other Self-Balancing Trees

AVL is not the only balanced tree!

Tree Type	Balance Property	Height
AVL	$\lvert BF \rvert \leq 1$	$1.44 \log n$
Red-Black	Color rules	$2 \log n$
2-3 Tree	All leaves same depth	$\log n$
B-Tree	Nodes have many children	$\log n$

Red-Black Trees:

- More relaxed balance (faster insert/delete)
- Used in C++ `map`, Java `TreeMap`
- Height up to $2 \log n$ (vs AVL's $1.44 \log n$)

Trade-off: AVL more strictly balanced (better search), Red-Black faster updates

Summary

Problem: Regular BST can degenerate to O(n)

Solution: Self-balancing trees maintain O(log n) height

AVL Trees:

- Balance factor: $|height(left) - height(right)| \leq 1$
- Rotations restore balance after insert/delete
- Four cases: LL, RR, LR, RL
- All operations: O(log n) guaranteed

Key concepts:

- Height and balance factor
- Single vs double rotations
- Trade-off: guaranteed performance vs implementation complexity

What's Next

Today: AVL Trees - guaranteed O(log n) through rotations

Next: Graphs

- Representations (adjacency matrix/list)
- Traversals (BFS, DFS)
- Applications (connectivity, topological sort)

Later: More algorithm design paradigms

- Greedy algorithms
- Dynamic programming
- Complexity theory (P vs NP)