AVL Trees and Intro to Graphs

Data Structures and Algorithms Tutorial

IIITH - S26

AVL Trees, Graphs (C++)

Agenda

1. AVL Trees
2. Graphs
3. Summary

AVL Trees

What is an AVL Tree?

Definition

- Self-balancing BST invented by Adelson-Velsky and Landis (1962)
- Balance factor: height(left) - height(right)
- For EVERY node: |balance factor| <= 1

Why AVL?

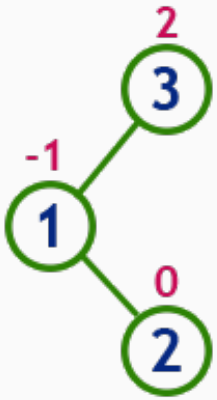
- Regular BST can degenerate to linked list: $O(n)$
- AVL guarantees $O(\log n)$ for all operations

Balanced Cases

- 0: Left and right subtrees same height
- 1: Left is 1 taller (left-heavy)
- -1: Right is 1 taller (right-heavy)
- |bf| > 1: Unbalanced! Needs rotation

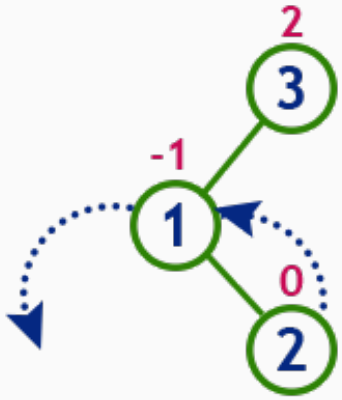
AVL Rotations

insert 3, 1 and 2



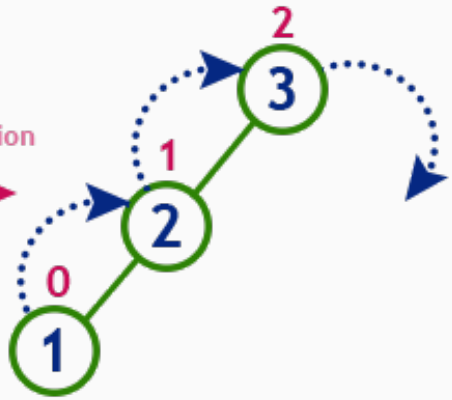
Tree is imbalanced

because node 3 has balance factor 2



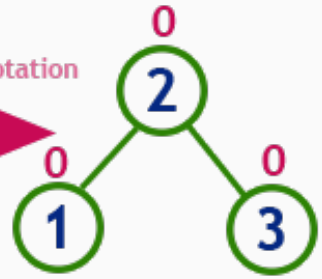
LL Rotation

After LL Rotation



RR Rotation

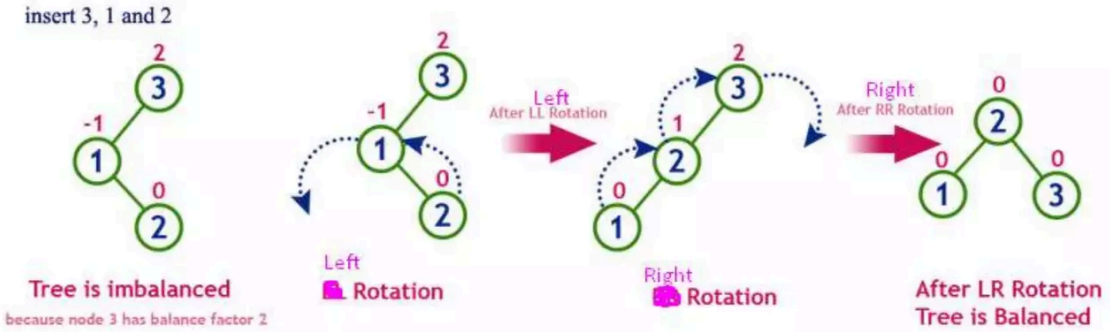
After RR Rotation



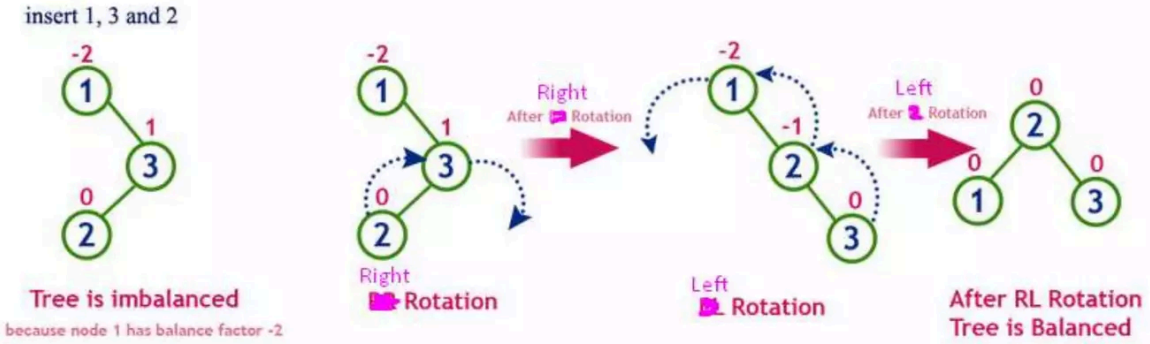
After LR Rotation
Tree is Balanced

AVL Tree Rotations:

Type 3: LR Relationship (First Left Rotation then Right Rotation)



Type 4: RL Relationship (First Right Rotation then Left Rotation)



AVL Insertion Algorithm

Steps

1. Insert normally (BST insert)
2. Update heights going up
3. Check balance factor at each node
4. If unbalanced, perform appropriate rotation

Rotation Selection

- Left-Left (bf > 1, left child bf >= 0): Right rotation
- Left-Right (bf > 1, left child bf < 0): Left-Right rotation
- Right-Right (bf < −1, right child bf <= 0): Left rotation
- Right-Left (bf < −1, right child bf > 0): Right-Left rotation

Time: $O(\log n)$ - height of tree

AVL Deletion

Steps

1. Delete normally (BST delete)
2. Update heights going up
3. Check balance factor at each node
4. Rebalance if needed

Deletion Cases

- Leaf: Just remove
- One child: Replace with child
- Two children: Replace with inorder successor (or predecessor), then delete successor

Key Difference from Insert

- Multiple rotations might be needed on path to root
- Deletion can cause imbalance at multiple levels

Time: $O(\log n)$

AVL Visualization

Graphs

Graph Representation

Graph: G = (V, E)

- V = set of vertices (nodes)
- E = set of edges (connections)

Two main representations

1. Adjacency Matrix: int adj[n][n]
2. Adjacency List: Array of linked lists

Which one to use?

- Matrix: Simple, but $O(V^2)$ space
- List: Efficient for sparse graphs, $O(V + E)$ space

For this course: We'll use adjacency lists in C++

Why C++ for Graphs?

Problem with C

- Adjacency list: array of linked lists
- Lots of pointer management
- Memory allocation overhead
- Verbose code

C++ Solution: vector

- Dynamic array that grows automatically
- Clean syntax for adjacency lists
- Still only what you need (no extra STL features)

Allowed for graphs ONLY

```
vector<int> adj[MAX_N];  // Array of vectors
adj[u].push_back(v);     // Add edge
```

C++ Vector Basics

Vector: Dynamic array (only STL feature you can use)

```cpp
#include <vector>
using namespace std;
int main() {
    int n;
    cin >> n;
    vector<int> v(n);
    v[0] = 5;
    int x = v[0];
    // Add element (dynamic growth)
    v.push_back(10);  // O(1) amortized
    // Size
    int sz = v.size();
    return 0;
}
```

Adjacency List in C++

Unweighted Graph

```
const int MAX_N = 100005;
vector<int> adj[MAX_N];
int main() {
    int n, m;
    cin >> n >> m;  // n vertices, m edges
    for (int i = 0; i < m; i++) {
        int u, v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);  // Remove for directed
    }
}
```

Space: $O(V + E)$ | Add edge: $O(1)$

Weighted Graphs in C++

Need to store: (neighbor, weight) pairs

```
const int MAX_N = 100005;
struct Edge {
    int to, weight;
};
vector<Edge> adj[MAX_N]; // or just use pair<int,int>
int main() {
    int n, m;
    cin >> n >> m;
    for (int i = 0; i < m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        adj[u].push_back({v, w});
        adj[v].push_back({u, w});
```

```
    }
}
```

Traversal:

```
for (Edge e : adj[u]) {
    cout << u << " -> " << e.to << " (weight: " << e.weight << ")\n";
}
```

Common Input Format for Graphs

Format 1: Edge List

```
n m          // n vertices, m edges
u1 v1        // edge 1
u2 v2        // edge 2
...
```

Format 2: Weighted Edge List

```
n m
u1 v1 w1     // edge with weight
u2 v2 w2
...
```

Reading in C++

```
int n, m;
cin >> n >> m;
vector<Edge> adj[n];
while (m--) {
    int u, v, w;
    cin >> u >> v >> w;
    adj[u].push_back({v, w});
}
```

CSES Graph Problems

Recommended starting problems

- Building Roads: Count connected components
- Building Teams: Bipartite check
- Building Roads: Find shortest path
- Shortest Routes I: Dijkstra's algorithm

Summary

Summary: AVL Trees

Key Points

- Self-balancing BST with $O(\log n)$ guarantees
- Balance factor must be in [-1, 0, 1] for all nodes
- Four rotation types: LL, RR, LR, RL
- Insert: One rotation max (at first unbalanced ancestor)
- Delete: Multiple rotations possible

When to use AVL

- Need guaranteed $O(\log n)$ operations
- Lots of lookups, fewer inserts/deletes
- Memory overhead is acceptable

Takeaway Problems!

Sorted Array to BST

Convert sorted array to height-balanced BST.

Recover BST

Two nodes swapped in BST. Recover it.

Shortest Routes II

Floyd-Warshall all-pairs shortest path.

More you know ...

Gathering Children

Sparse Table for RMQ.

Road Construction

Union-Find with rollback.

Questions?

Thank you!