DS 4300: Large Scale Information Storage and Retrieval
Foundations

Searching

- Searching is the most common operation performed by a database system.
- In SQL, `SELECT` is arguably the most versatile and complex statement.
- **Baseline for efficiency**: Linear Search (start at beginning and go element by element).
- **Record** = a collection of attribute values for a single entity instance (a row).
- **Collection** = a set of records of the same type (a table).
- **Search Key** = a value for an attribute used for searching.

Lists of Records

- If each record takes x bytes, n records need `n*x` bytes.
- **Contiguously Allocated List**: all `n*x` bytes allocated in one chunk.
- **Linked List**: each record needs x bytes + space for pointers.

Arrays vs Linked Lists:

- Arrays: Faster random access, slower insertion (except at the end).
- Linked Lists: Faster insertion anywhere, but slower random access.

Binary Search

- Input: sorted array + a target value.
- Output: index of target, or -1 if not found.
- Time complexity (worst case): O(log n).

Linear Search worst case: O(n).

Back to Database Searching

- Suppose data is stored on disk sorted by some primary key (like `id`), making searching on `id` fast.
- Searching by a different attribute (like `specialVal`) would require a linear scan unless we create an additional data structure (index).
- We can't store the table physically sorted by multiple attributes at once without duplicating data.

Potential Data Structures for Indexing

- An **array** of `(searchKey, rowNumber)` tuples sorted by `searchKey` → supports binary search, but inserting is slow (because you must keep it sorted).

- A **linked list** of `(searchKey, rowNumber)` → supports faster insertion but linear search.
- A **binary search tree** → can provide relatively fast insertion and lookup.

(A deeper discussion leads to balanced trees like B-trees, B+ trees, etc.)

03 - Moving Beyond the Relational Model

DS 4300: Moving Beyond the Relational Model

Benefits of the Relational Model

- Mostly standard data model and query language.
- ACID compliance (Atomicity, Consistency, Isolation, Durability).
- Handles large amounts of structured data, with lots of tooling and expertise available.

Relational Database Performance

- Mechanisms to increase efficiency include:
  - Indexing
  - Direct control over storage
  - Column-oriented storage vs. row-oriented storage
  - Query optimization
  - Caching/prefetching
  - Materialized views
  - Stored procedures
  - Data replication and partitioning

Transaction Processing

- **Transaction** = sequence of CRUD operations executed as a single unit (all succeed or all fail).
- Ensures data integrity, concurrency control, error recovery, and simpler error handling.

ACID Properties

1. **Atomicity**: A transaction is all-or-nothing.
2. **Consistency**: A transaction takes the database from one consistent state to another, respecting constraints.
3. **Isolation**: Concurrent transactions do not interfere with each other (avoid dirty reads, non-repeatable reads, phantom reads).
4. **Durability**: Once committed, the transaction's changes are permanent, even if there's a system failure.

Example: ACID Transaction (Money Transfer)

(SQL pseudo-code showing `START TRANSACTION`, updates to two accounts, checking for negative balance, ROLLBACK or COMMIT, etc.)

Limits of Relational Databases

- Some applications do not need the full strength of ACID.
- Joins can be expensive for certain access patterns.
- A lot of modern data is semi-structured or unstructured (JSON, XML).
- Horizontal scaling can be challenging.
- Some apps require extremely high performance or real-time access.

Scalability: Scale Up vs. Scale Out

- Traditional approach: scale vertically (bigger machines) until forced to scale out due to availability concerns.
- Horizontal scaling can be more complex but is often necessary for very large systems.

Distributed Data and the CAP Theorem

- **Distributed system**: A collection of independent computers that appear as a single system to the user.
- **CAP Theorem**: A distributed data store cannot provide more than two of:
  - Consistency
  - Availability
  - Partition Tolerance
- In practice, systems must make trade-offs.

04-B+Tree Walkthrough

B+ Tree Example

- Order `m=4`.
- Insert a series of keys, showing how the tree grows and splits.
- B+ trees keep data in leaf nodes, with interior nodes holding copies of keys for navigation.
- Balanced, ensuring O(log n) lookup and insertion.

(Slides walk through example insertions of 42, 21, 63, 89, 35, 10, 27, 96, 30, 37, showing how leaves split and eventually how the root splits.)

04 - Data Replication

Replicating Data

Distributing Data - Benefits

1. Scalability / High throughput (handle larger volumes).
2. Fault Tolerance / High Availability (system can continue if nodes fail).
3. Latency (place data copies closer to global users).

Distributed Data - Challenges

- Consistency: updates must propagate across the network.
- Increased application complexity: the application code may need to handle distribution.

Replication

- Replicas contain copies of the same data.

Common Replication Strategies

- **Single Leader**: One node (leader) handles writes; other nodes (followers) replicate.
- **Multiple Leaders**: Multiple nodes can accept writes, but conflict resolution is more complex.
- **Leaderless**: All nodes can accept writes, often employing quorum reads/writes and version vectors.

Leader-Based Replication

- All writes go to the leader, which sends replication updates to followers.
- Clients can read from leader or followers.
- Used by many relational and NoSQL systems (MySQL, PostgreSQL, MongoDB replica sets, Kafka, etc.).

How Replication Info Is Transmitted

- *Statement-based*: Forward actual SQL statements (risk with non-deterministic functions).
- *Write-ahead Log (WAL)* shipping: Byte-level log of storage engine changes.
- *Logical (row-based) log*: Higher-level description of row changes, more flexible.
- *Trigger-based*: Custom triggers log changes to special tables.

Synchronous vs Asynchronous Replication

- **Synchronous**: Leader waits for a follower's acknowledgment, ensuring immediate consistency but possibly higher latency.
- **Asynchronous**: Leader does not wait. Better performance but leads to *eventual consistency* across replicas.

Handling Leader Failure

- Need a failover mechanism: pick a new leader, ensure no conflicting leaders, handle partial replication (some writes might be lost).

Replication Lag

- In asynchronous replication, followers may be behind the leader. This is the *inconsistency window*.

Consistency Models with Replicas

- **Read-after-Write Consistency**: Ensure a user sees their own writes immediately (can force reads from leader).
- **Monotonic Read Consistency**: Prevent a user from reading older data after newer data in consecutive queries.
- **Consistent Prefix Reads**: Ensure that reads respect the write order (no out-of-order writes).

05 - NoSQL Intro + KV DBs

NoSQL

- "Not Only SQL": broad category of non-relational DBs.
- Often address large-scale data, flexible schemas, and distributed data with relaxed consistency.
- Rely on **BASE**: Basically Available, Soft State, Eventual Consistency.

ACID vs. BASE

- Traditional RDBMS focus on strong consistency (ACID).
- Many NoSQL systems adopt BASE (eventual consistency).

Categories of NoSQL

- Key-Value Stores
- Document Stores

- Wide-Column Stores
- Graph Databases
- Others (time-series, etc.)

Key-Value Databases

- Simple: `key = value`.
- Fast, typically O(1) lookups.
- Great for caching, session storage, shopping cart data, etc.

Redis

- In-memory data structure store.
- Supports various data types (strings, lists, sets, sorted sets, hashes, etc.).
- Very high performance (100k+ ops/sec).
- Optional persistence to disk (snapshots or append-only file).

Redis Data Types (Overview)

- **Strings**: Basic text/binary up to 512 MB.
- **Lists**: Linked lists of strings.
- **Hashes**: Field-value pairs (like an object).
- **Sets**: Unordered unique strings, with set operations (union, intersect, diff).
- **Sorted Sets**: Like sets but with a score for ordering.
- **Geo**: Geospatial indexes.
- **JSON**: Can store and manipulate JSON data (via Redis modules).

Common Use Cases for KV Stores

- Caching
- Session management
- Real-time analytics
- Feature store in ML
- Queues (producer/consumer patterns)

05b - Redis in Docker

Setting Up Redis in Docker

1. In Docker Desktop, search for the “Redis” image and click **Run**.
2. Optionally expose port 6379 to the host.
3. In a production environment, secure Redis with a password and do not expose it directly to the internet.

4. Use DataGrip (or another client) to connect to Redis on `localhost:6379`.

06 - Redis + Python

`redis-py`

- Standard Python client library for Redis.
- Install via `pip install redis`.
- Basic usage: `python`
  CopyEdit

```
import redis
```

- `r = redis.Redis(host='localhost', port=6379, db=0, decode_responses=True)`
-

- `decode_responses=True` ensures strings are returned (instead of bytes).

Redis Command Highlights (Strings)

- `set(key, value)`, `get(key)`, `incr(key)`, `decr(key)`, etc.
- `mset(...)` to set multiple keys at once, `mget(...)` to retrieve multiple.

Redis Command Highlights (Lists)

- `lpush(key, *values)`, `rpush(key, *values)`, `lpop(key)`, `rpop(key)`, etc.
- `lrange(key, start, stop)` to retrieve sublist.

Redis Command Highlights (Hashes)

- `hset(key, field, value)` or `hset(key, mapping={...})`
- `hget(key, field)`, `hgetall(key)`, etc.

Pipelines

- Group multiple commands in a pipeline to reduce network round trips: `python`
  CopyEdit

```
pipe = r.pipeline()
```

- `pipe.set("seat:1", "#1")`
- `pipe.set("seat:2", "#2")`
- `results = pipe.execute()`
-

Redis in ML/DS

- Used for storing and quickly retrieving features, session data, or model outputs in real time.

07 - Document DBs and Mongo

Document Databases

- Store data as JSON or similar (BSON).
- Flexible schema: each document can have different fields.
- Great for applications passing JSON/XML data.

MongoDB

- Leading document store built on BSON (binary JSON).
- Basic hierarchy:
  - **Database**
    - **Collection**
      - **Documents**
- Flexible, easy horizontal scaling, powerful query language (`find`, `insert`, `update`, etc.), and indexing support.

MongoDB Features

- Rich query support (CRUD).
- Indexing on document fields.
- Replication via replica sets.
- Horizontal sharding / partitioning.

MongoDB Editions

- **MongoDB Atlas**: fully managed in the cloud.
- **MongoDB Enterprise**: subscription-based, self-managed.
- **MongoDB Community**: free to use, self-managed.

Interacting with MongoDB

- `mongosh`: CLI shell.
- MongoDB Compass: official GUI.
- 3rd party tools: DataGrip, etc.
- Programmatically: PyMongo (Python), Mongoose (Node.js), etc.

Example: Docker + MongoDB

- Pull official image, run container, map port `27017`.
- Set environment variables for root username/password if needed.
- Use MongoDB Compass or another tool to connect.

Example Queries in MongoDB Shell

- `db.movies.find(...)`
- `db.movies.find({year: 2010})`
- `db.movies.find({year: 2010, $or: [{"awards.wins": {$gte: 5}}, {"genres": "Drama"}]})`
- `db.movies.countDocuments({...})`
- Projections: `db.movies.find({<filters>}, {name: 1, _id: 0})`

08 - PyMongo

Connecting with PyMongo

python
CopyEdit
```
from pymongo import MongoClient

client = MongoClient('mongodb://
user_name:pw@localhost:27017')
db = client['ds4300']
collection = db['myCollection']
```

Basic Operations

- **Insert a document**: `python`
  CopyEdit

```
post = {
    "author": "Mark",
    "text": "MongoDB is Cool!",
    "tags": ["mongodb", "python"]
}
post_id = collection.insert_one(post).inserted_id
print(post_id)
```

- **Find documents**: `python`
  CopyEdit

```
db.movies.find({"year": 2000})
```

- **Counting documents**: `python`
  CopyEdit

```
db.collection.count_documents({})
```

- *Etc*. for updates, deletes, indexing, etc.

09 - Introduction to Graph Data Model

Title: DS 4300 – Introduction to the Graph Data Model
Key Points:

1. **Graph Databases**
   - Data model based on nodes (vertices) and edges (relationships).
   - Nodes and edges each can contain properties (like key-value pairs).
   - Edges connect nodes; edges are often directed and can have labels.

- Graph queries often involve traversals (finding paths, shortest paths, etc.).

2. **Where Graphs Show Up**
    - **Social networks** (modeling people and relationships).
    - **The Web** (pages as nodes, hyperlinks as edges).
    - **Chemical and biological data** (genes, proteins, chemical compounds, etc.).
3. **Basics of Graphs**
    - **Labeled Property Graph**: nodes have labels (grouping them), properties on nodes and edges.
    - Nodes can exist without relationships; edges must connect two nodes.
    - **Paths**: ordered sequence of connected nodes/edges with no repeated nodes or edges.
4. **Flavors of Graphs**
    - **Connected** vs. **Disconnected**.
    - **Weighted** vs. **Unweighted** (edges may have weights).
    - **Directed** vs. **Undirected** (edges may be directional).
    - **Acyclic** vs. **Cyclic**.
    - **Sparse** vs. **Dense**.
5. **Types of Graph Algorithms**
    - **Pathfinding** (shortest path, BFS, DFS, spanning tree, max flow, etc.).
    - **Centrality**: determining which nodes are most "important" (e.g., social media influencers).
    - **Community Detection**: clustering or grouping nodes (detecting subgraphs or communities).
6. **Famous Graph Algorithms**
    - **Dijkstra's**: single-source shortest path with positive edge weights.
    - **A***: uses a heuristic to guide pathfinding.
    - **PageRank**: ranks nodes by importance based on incoming links from other "important" nodes.
7. **Neo4j**
    - A popular graph database that supports transactional and analytical workloads.
    - Similar alternatives include Microsoft Cosmos DB (graph features) and Amazon Neptune.

10 - Neo4j

Title: DS 4300 – Neo4j
Key Points:

1. **Neo4j**

- Graph DB system for transactional and analytical graph-based data.
- Schema-optional (can impose a schema if desired).
- ACID-compliant. Supports indexing, distribution, and sharding.

2. **Cypher Query Language**

   - Introduced by Neo4j in 2011.
   - Pattern-based syntax: `(nodes)-[:REL]->(otherNodes)`.
   - A visual, declarative query style for matching relationships in a graph.

3. **APOC Plugin**

   - "Awesome Procedures on Cypher": library with hundreds of extra procedures/ functions.
   - Extends functionality for data export/import, graph algorithms, integration tasks.

4. **Graph Data Science Plugin**

   - Implements common graph algorithms (PageRank, community detection, etc.) efficiently within Neo4j.

5. **Docker Compose for Neo4j**

   - Example `docker-compose.yaml` snippet:
     - Uses `image: neo4j:latest` and sets ports `7474` (HTTP browser UI) and `7687` (Bolt protocol).
     - Exposes environment variables for password, enabling APOC, GDS plugins, etc.
     - Maps volumes for data, logs, import, and plugins.

6. **Neo4j Browser**

   - Web UI on `localhost:7474` once the container is up.
   - Can execute Cypher commands directly.

7. **Basic Cypher Examples**

   - `CREATE (:User {name: "Alice", birthPlace: "Paris"})`
   - Matching a node: `c`
     CopyEdit

```
MATCH (u:User {birthPlace: "London"})
RETURN u.name, u.birthPlace
```

   -
   - Relationship creation: `cypher`
     CopyEdit

```
MATCH (alice:User {name:"Alice"}), (bob:User
{name:"Bob"})
```

- ```
  CREATE (alice)-[:KNOWS {since: "2022-12-01"}]-
  >(bob)
  ```
-

8. **Importing CSV**

    - Place CSV file in the `import` directory used by the Docker container.
    - Example: `cypher`
      CopyEdit

```
LOAD CSV WITH HEADERS
```

- `FROM 'file:///netflix_titles.csv' AS line`
- ```
  CREATE (:Movie { id: line.show_id, title:
  line.title, releaseYear: line.release_year })
  ```
-

- Using `split(...)` + `UNWIND` to handle multiple directors in one record, possibly `MERGE` to avoid duplicates.

11 - AWS Intro

Title: DS 4300 – AWS Introduction
Key Points:

1. **Amazon Web Services (AWS)**

    - A leading public-cloud platform with 200+ services.
    - Global infrastructure of "Regions" and "Availability Zones."
    - Pay-as-you-use model.

2. **Historical Context**

    - Launched in 2006 with S3 (object storage) and EC2 (virtual servers).
    - Grew to include RDS, DynamoDB, CloudFront, and more.

3. **AWS Service Categories**

    - **Compute** (e.g., EC2, Lambda, ECS, EKS).

- **Storage** (e.g., S3, EFS, EBS).
   - **Databases** (Relational: RDS/Aurora; NoSQL: DynamoDB, DocumentDB, etc.).
   - **Analytics** (EMR, Athena, Glue, Redshift, Kinesis).
   - **Machine Learning** (SageMaker, Comprehend, Rekognition, etc.).
4. **Cloud Models (IaaS, PaaS, SaaS)**
   - **IaaS**: Infrastructure as a Service (e.g., EC2).
   - **PaaS**: Platform as a Service (removes infrastructure management).
   - **SaaS**: Ready-to-use software hosted by the vendor (e.g., Salesforce).
5. **Shared Responsibility Model**
   - **AWS**: Security "OF" the cloud (data centers, hypervisors, etc.).
   - **Customer**: Security "IN" the cloud (own data, IAM, encryption, OS patches if self-managed).
6. **AWS Global Infrastructure**
   - **Regions**: distinct geographic areas (e.g., `us-east-1`).
   - **Availability Zones**: isolated data centers in each region.
   - **Edge Locations**: content delivery endpoints (CDN, caching).
7. **Key Services**
   - **Compute**: EC2 (VMs), ECS/EKS (containers), Lambda (serverless).
   - **Storage**: S3 (object store), EBS (block store), EFS (shared file system).
   - **Databases**: RDS, DynamoDB, DocumentDB, Neptune, etc.
   - **Analytics**: Athena, EMR, Glue, Redshift.
   - **ML**: SageMaker, pre-trained AI services.
8. **AWS Free Tier**
   - Offers limited free usage for new accounts (12 months for select services).
   - Must watch for usage exceeding free tier to avoid unexpected charges.

12 - EC2 & Lambda

Title: DS 4300 – Amazon EC2 & Lambda
Key Points:

Amazon EC2

1. **EC2 Overview**
   - "Elastic Cloud Compute": scalable virtual machines in the cloud.
   - Many instance types with different CPU, memory, GPU, etc.
2. **EC2 Features**
   - **Elasticity**: programmatically scale up/down.
   - **AMI**: Amazon Machine Image for pre-configured OS or custom images.

  - Integrates easily with other AWS services (S3, RDS, etc.).
3. **EC2 Lifecycle**
  - *Launch* (create a new instance).
  - *Start/Stop* (pause usage if instance is EBS-backed).
  - *Terminate* (delete the instance).
  - *Reboot* (restart, preserving data on root volume).
4. **EC2 Storage Options**
  - *Instance Store*: ephemeral storage tied to the instance's lifecycle.
  - *EBS* (Elastic Block Store): persistent block-level storage volumes.
  - *EFS* (Elastic File System): managed NFS-like file system for multiple instances.
  - *S3* for object storage or backups.
5. **EC2 Use Cases**
  - Web hosting, data processing, ML training, disaster recovery, etc.
6. **Example Setup**
  - Launching an EC2 instance with Ubuntu, installing software (conda, Streamlit, etc.), opening security group ports.

AWS Lambda

1. **Lambda Overview**
  - Serverless compute: run code in response to events (HTTP requests, S3 file upload triggers, etc.).
  - Pay only for execution time, not for idle time.
2. **Lambda Features**
  - Event-driven (integrates with S3, SNS, API Gateway, etc.).
  - Supports multiple runtimes (Python, Node, Java, etc.).
  - Automatically scales.
3. **Workflow**
  - Upload code via AWS Console or CLI, configure triggers/events.
  - Lambda runs code on-demand.
4. **Example**
  - Creating a Lambda function in the AWS Console, writing Python code (handler function), configuring test events or real event triggers.

12.6. B-Trees — CS3 Data Structures & Algorithms

Source: Summarized from a tutorial on B-trees and B+ trees.
Key Points:

1. **B-Trees**

    - Proposed by Bayer & McCreight (1972) for disk-based search trees.
    - Shallow (height-balanced) with potentially high branching factor, minimizing disk I/O.
    - Each node can have multiple children; typically matches disk block size.
2. **Properties of a B-tree of order m**:

    - The root has at least two children if it's not a leaf.
    - Each internal (non-root) node has between ⌈m/2⌉ and m children.
    - All leaves are at the same level (height-balanced).
    - Each node typically holds key-value pairs up to a certain maximum, ensuring each node is at least half-full.
3. **Operations**:

    - *Search*: similar to searching in a 2-3 tree, do a binary search in the current node's keys, descend the appropriate child.
    - *Insertion*: find the correct leaf; if the leaf is full, split it and promote the middle key to the parent, possibly recursively splitting upward.
    - *Deletion*: if removing from a node leaves it underfull (< half capacity), attempt to rebalance by borrowing from siblings or merging nodes.
4. **B+ Trees**

    - A variant of B-trees (often used in databases).
    - **Difference**: all actual records (data) are stored in leaf nodes; internal nodes store only keys for navigation.
    - Leaves are often linked for range scans (doubly linked list).
    - Also must remain half-full.
    - Searching continues down to leaf level, even if a matching key is found in an internal node (since that internal node is just an index).
5. **B+ Tree Insertion/Deletion**

    - Similar to B-tree logic but with the rule that data resides only in leaves, internal nodes hold guide keys.
    - Leaves also have a linked-list chain for sequential access.
    - Balanced structure ensures O(log n) search, insertion, and deletion times.

AVL-Tree-Rotations.pdf

This file is a **PNG image** illustrating **AVL tree rotations**. The essential knowledge (textual) about AVL rotations includes:

- **AVL Trees**: self-balancing binary search trees where the height of the left and right subtrees of any node differ by at most 1.
- **Rotations** fix imbalances. Key rotation patterns:
    - **LL rotation** (Right rotation).

    - **RR rotation** (Left rotation).
    - **LR rotation** (Left-Right).
    - **RL rotation** (Right-Left).
- Each rotation re-links a small set of pointers in constant time, preserving the BST property while reducing subtree height.

(The PNG visually shows how subtrees are reattached during rotations.)

B-trees.pdf

Title: "B-trees"
Key Points:

1. **Context**: B-trees store multiple elements in each node, exploiting locality (useful for caching or disk storage).
2. **Order m**: Non-leaf nodes can have up to m children, leaves hold the actual data. Invariants:
    - All leaves are at the same level (height-balance).
    - Each node is at least half-full (except possibly root).
    - The number of keys in an internal node is one less than the number of children.
3. **Operations**:
    - *Lookup*: find correct child pointer by comparing the search key to node keys, continuing until leaf is reached.
    - *Insertion*: insert into a leaf. If leaf is full, split, push up the middle key to the parent. Possibly recurse upward if the parent is full, etc.
    - *Deletion*: if a leaf goes below half-full, borrow from siblings or merge with a sibling; can cascade upward if the parent likewise becomes underfull.
4. **Benefits**:
    - Minimizes disk I/O by keeping branching factor large.
    - Balanced: O(log n) worst-case height and search times.

C12-bst.pdf

Title: Chapter 12 – Binary Search Trees
Key Points:

1. **BST-Property**

    - For any node x, every key in its left subtree < key[x], and every key in its right subtree > key[x].
2. **BST Operations**

    - **Traversal**: inorder, preorder, postorder.
        - Inorder of a BST yields sorted keys.

- **Search**: start from root, compare target key to current node's key, go left or right accordingly until found or nil.
- **Minimum/Maximum**: leftmost node is min, rightmost node is max.
- **Successor/Predecessor**: next larger (or smaller) in sorted order, found by a combination of descending right-then-left or parent references.
- **Insertion**: find the correct leaf position; place the new node there.
- **Deletion**:
  - No children: remove the node outright.
  - One child: splice out the node, promote its single child.
  - Two children: find the node's successor (or predecessor), copy that key/data into the node to be deleted, then remove the successor from its location (which is a simpler case, as successor has ≤ 1 child).

3. **Complexity**
   - If the tree height is **h**, basic operations are O(h). Balanced BSTs have h = O(log n), but worst-case (unbalanced) can be O(n).

ICS 46 Spring 2022, Notes and Examples: AVL Trees

Title: ICS 46 – AVL Trees
Key Points:

1. **Binary Search Tree Balancing**
   - BST performance depends on shape (height). A "degenerate" BST can degrade operations to O(n).
2. **AVL Trees**
   - **Definition**: A BST where for every node, the heights of the left and right subtrees differ by at most 1.
   - Ensures worst-case height is O(log n).
3. **Rotation Operations**
   - **LL** (single right rotation), **RR** (single left rotation).
   - **LR** (left then right rotation), **RL** (right then left).
   - Rotations rebalance the tree in constant time by adjusting only a few pointers.
4. **Insertion in an AVL Tree**
   - Insert as in a normal BST.
   - Then "walk back up" to the root, checking for violations of the AVL balance property.
   - If a node is imbalanced, apply one of LL, RR, LR, or RL rotations depending on the direction of insertion.
5. **Deletion in an AVL Tree**
   - Delete as in a normal BST.

- Then walk back up, re-check balance factors, applying rotations where needed.
- Maintaining the node's heights in each node helps check balance quickly.

6. **Why a "near-balance" criterion?**

   - Perfect or complete balance is expensive to maintain for every insertion.
   - AVL's local rebalancing ensures O(log n) height, with O(log n) insertion/deletion overhead.