Foundations
- Searching Basics
    - Searching is most common DB operation
    - SELECT - most versatile
- Data Storage Structures
    - Arrays: Fast random access, slow insertions
    - Linked Lists: Fast insertions, slow random access
- Database Search Challenges
    - Data stored by ID is fast to search.
    - Searching by other attributes requires linear scan (inefficient)
- Optimizing Search & Inserts
    - Sorted arrays allow binary search but slow inserts
    - Linked lists allow fast inserts but slow searches
    - Binary Search Trees (BSTs) balance fast inserts and searches

BST URochester
- A binary tree where each node has a key, and for any node x, all keys in its left subtree are less than x's key, and all keys in its right subtree are greater
- Traversal Methods:
    - Inorder Traversal (Visits the left subtree, then the current node, and finally the right subtree. In a BST, this traversal retrieves keys in ascending order), Preorder Traversal (before sub), Postorder Traversal (after sub)
- Search → Compare with root, move left if smaller, right if larger
- Find Min/Max →
    - Min: Go left until no more left child
    - Max: Go right until no more right child
- Insertion → Find correct position, insert while maintaining order
- Successor →
    - If right child exists → Leftmost node in right subtree
    - Otherwise → First ancestor where node is in the left subtree
- Predecessor →
    - If left child exists → Rightmost node in left subtree
    - Otherwise → First ancestor where node is in the right subtree

AVL UCI NOtes
- AVL: self balancing
- Insertion: insert like BST, then rebalance if unbalanced
- Deletion: remove node, rebalance all the way up
- Search: same as BST but always O(log n)
- Single Rotation: Fixes left-left or right-right imbalance
- Double Rotation: Fixes left-right or right-left imbalance

AVL Rotations:
- Right Rotation: Fixes left-heavy imbalance
- Left Rotation: Fixes right-heavy imbalance
- Left-Right Rotation: First left rotate child, then right rotate parent
- Right-Left Rotation: First right rotate child, then left rotate parent

B Trees:

- Self-balancing tree: Keeps data sorted for fast operation
- Search, insert, delete in O(log n) time
- All leaf nodes are at the same level
- Each node holds m/2 to m children (except root)
- Start from root → insert in appropriate leaf (If full, split the node and push the middle key up)→ deletion (remove the key, if a node underflows, borrow from siblings or merge nodes)→ balancing (Nodes are split or merged to keep the tree balanced)

B+ Trees:

- Insertion: Start at the root and follow keys to the correct leaf node.
    - If leaf is full, split it, move half the values to a new node, and copy the smallest key to the parent
- Splits:
    - Leaf split → Copy smallest key to parent
    - Internal split → Move middle key up
    - Root split → Adds a new level to the tree

Mod 3 Moving Beyond Relational

- Relational Databases → Strong ACID compliance, great for structured data, but scaling is hard
- Challenge: Schema changes, expensive joins, data limitations

Mod 5 NoSQL

- NoSQL: Scales horizontally, handles unstructured data, uses BASE instead of ACID
- CAP Theorem: Can only pick two: Consistency, Availability, or Partition Tolerance
- Key-Value Stores: Fast (O(1)) lookups, simple key = value structure, no joins
- Use Cases:
    - Session storage
    - Caching
    - Feature stores
    - Shopping carts
- Redis: In-memory KV store, supports multiple data types, ultra-fast (>100K ops/sec)

Mod 6 Redis Python

- redis.Redis(host, port, db, decode_responses=True)
- Strings → set(), get(), incr(), mset().
- Lists → rpush(), lpush(), lrange(), lpop().
- Hashes → hset(), hgetall(), hdel()

Mod 7 Doc DBs

- MongoDB
    - Stores data as JSON/BSON
    - indexing, replication, load balancing
    - Similar to SQL

Mod 8 PyMongo

- Insert → insert_one(), insert_many().
- Find → find(), find_one().
- Update → update_one(), update_many().

- Delete → delete_one(), delete_many()