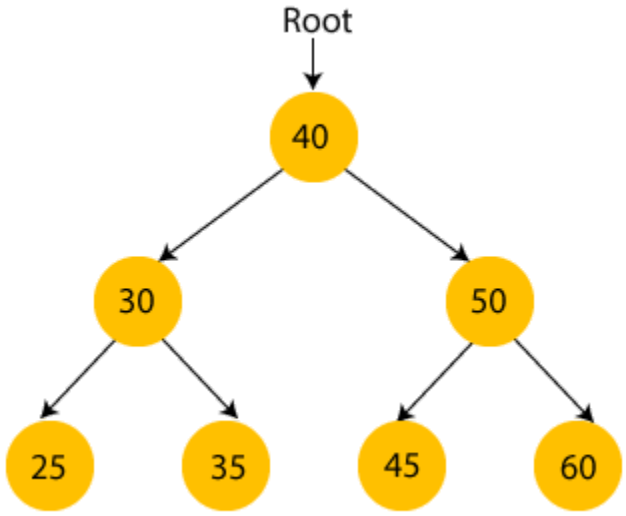
- Binary Search
    - Only for sorted arrays
    - Best case is 1
    - Worst case is O(log2n) + 1
    - Better for searching
    - Worse for inserting
- Linear Search
    - Can be used on unsorted
    - Best case is 1
    - Worst case is O(n)
    - Better for inserting
    - Worse for searching
- Binary Search Tree
    - Start with number (40), if next number is smaller branch to the left (30), if bigger to the right (50). Next number if smaller than original branch left (35), then compare to next and continue branching (35 is smaller than 40 so go left, then bigger than 30 so go right).



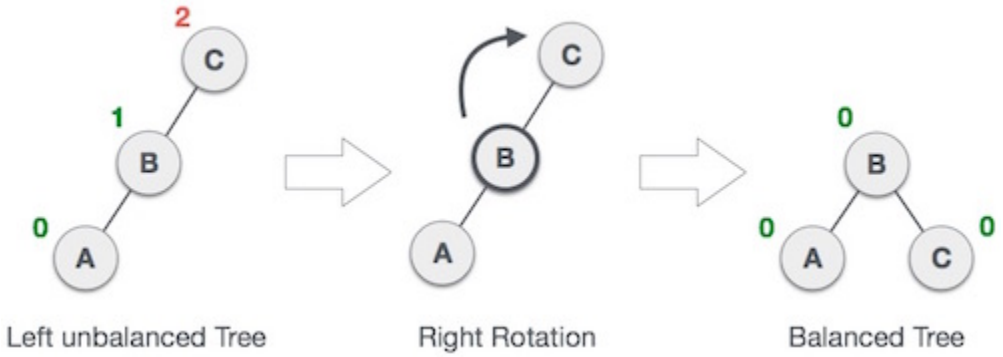
    -
    - Level order traversal for this : 40 30 50 25 35 45 60
    - Always has access to the root (40)
    - Level order traversal uses queue: 40 has access to 30 and 50. 40 Added to traversal, 30 and 50 added to queue. 30 has access to 25 and 35. 30 added to traversal, removed from queue, 25 and 35 added to queue. THEN 50 is next in queue so 50 gone to next, 25 next in queue but has nothing so nothing added to queue but 25 removed and added to traversal. So on and so forth.
        - Queue always added on right, removed from left.
    -

AVL Tree - (inventors Adelson-Velsky and Landis)
Self balancing binary search tree

When adding new depth to the tree you often encounter these imbalances (not every addition leads to an imbalance though):

Case 1 LL - Mirror of Case 4 RR
Insertion into left subtree of left child of node of imbalance

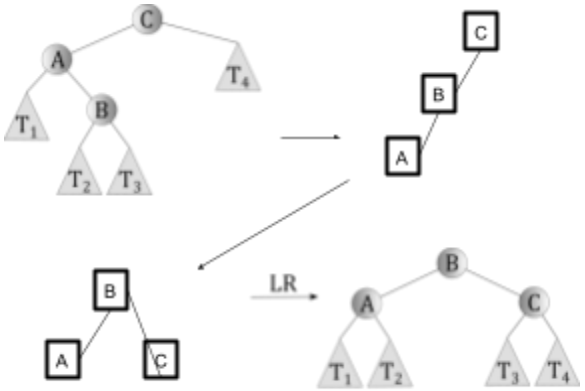
(Image is LL)



Note: Remember to adjust children of rotated nodes accordingly. (Diagram does NOT show)

Handle = programming term for references (or also pointers in languages with pointers)

Case 2 LR - Mirror of Case 3 RL
Insertion into right subtree of left child of node of imbalance



In this diagram T2 or T3 are inserted, then due to imbalance the LR correction is necessary

Step 1 - rotate B with A (This is a case 4 RR rotation)
Step 2 - rotate rightward so B is highest depth

Hypothetical:
Sorted array of 128 integers - each integer 64 bit integer is stored in 8 bytes so this takes 1024 bytes - WILL fit in a 2048 byte block

Hard drives are significantly slower than RAM - SO accessing all 128 integers is significantly faster than a single additionally access.

Def _rotate_left(self, x: AVLNode) -> AVLNode:
    Use rotations
    Y = x.right
    Update heights

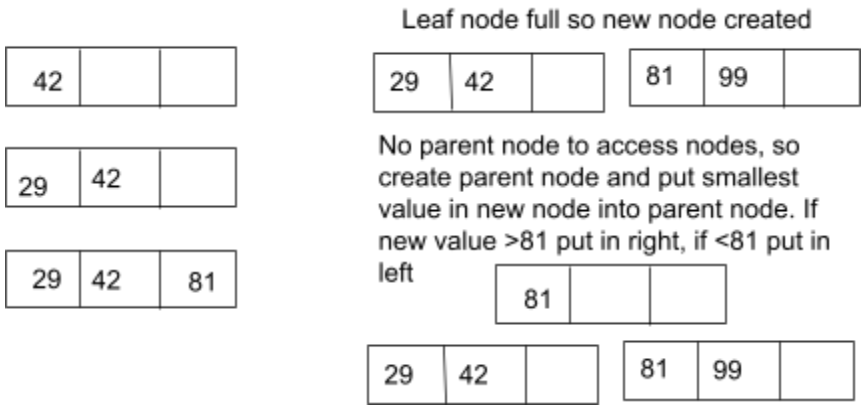
Hash Tables

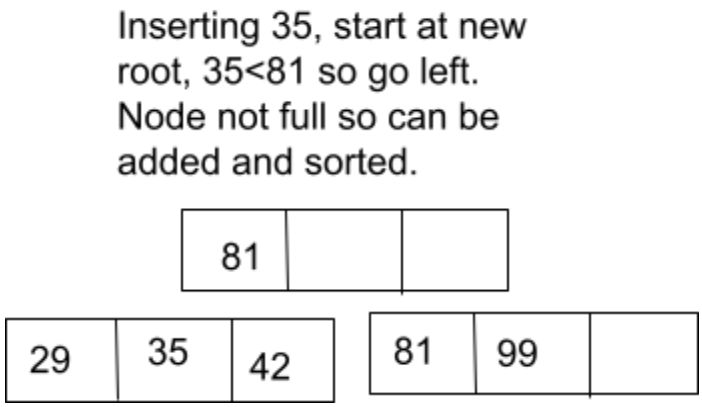
- Like a python dictionary
- Everytime given the same input will give the same output
- Table size is an important factor.
- Constant work for any k.
- If there are multiple things mapping to the same location, make it a list.
    - For our example, make hash a numpy array, individual locations can be python lists. Key = keyword , values = documents with keyword
    - Mod with table size so that table
- To find something from a hash, use the hash equation on the key you are looking for the value of, then search through the list at the output location. This is a linear search

B+ TREE

- Most common structure for database indexing
- Fast search within minimum space usage
- n-1 keys gives n children each with n values, each child between two keys or lower than lowest and higher than highest. Two levels of B+ tree holds about n^2 nodes, AVL tree holds 3 nodes.
- B+ tree with m=100 nodes. Level 0 = 1 node, Level 1 = (100 + 1 node), Level 2 = (101 * 101 nodes)
- **Goal is to minimize disk accesses by sorting with a __,** can run during background for easy access, but heavier computation
- B trees - in memory, B+ trees - disk based
- **PROPERTIES**
    - All nodes (except the root) must be ½ min

- Root node does not have to be ½ full
- Insertions must **always** start at leaf level
- Leaves are stored as a DLL
- Keys in nodes are kept sorted
- Internal nodes, leaf nodes
- Internal nodes only store keys and pointers to children, leaf nodes store keys and data. Internal nodes act as indexes.
- For project: internal nodes only store words, leaf nodes store list of articles its in/reference to list.
- When you split an internal node the smallest value gets moved up to a parent node **NOT COPIED**
    - More things in node = less room for keys = deeper tree

- Example: Insert 42, 29, 81, 99, 35, 2 in an m=3 tree
- 

- 

Transaction Isolation Levels

- Dirty Reads A dirty read occurs when a transaction reads data that has not yet been committed.
    - For example, suppose transaction 1 updates a row. Transaction 2 reads the updated row before transaction 1 commits the update. If transaction 1 rolls back

the change, transaction 2 will have read data that is considered never to have existed.

- Nonrepeatable Reads A non repeatable read occurs when a transaction reads the same row twice but gets different data each time.
    - For example, suppose transaction 1 reads a row. Transaction 2 updates or deletes that row and commits the update or delete. If transaction 1 rereads the row, it retrieves different row values or discovers that the row has been deleted.
- Phantoms A phantom is a row that matches the search criteria but is not initially seen.
    - For example, suppose transaction 1 reads a set of rows that satisfy some search criteria. Transaction 2 generates a new row (through either an update or an insert) that matches the search criteria for transaction 1. If transaction 1 re executes the statement that reads the rows, it gets a different set of rows.

ACID Properties

- Atomicity - Entire transaction takes place at ones, or doesn't happen at all
- Consistency - Database must be consistent before and after transaction
- Isolation - Multiple transactions occur without interference
- Durability - Changes of successful transaction occurs even if system failure occurs.

To avoid this, order of transaction operations must exist.
NEW SQL BUILT IN FUNCTIONS -

- CREATE PROCEDURE
    - Procedure is like a function
- BEGIN
    - Begin a procedure
- IF, ELSE, END IF, THEN (no ':' needed)
    -
- ROLLBACK
    - Undo actions if called
- COMMIT

Ways to increase power

- Vertical - use stronger computer
- Horizontal - use multiple computers - distributed dbs system
    - Distributed System, connects many computers to divide and conquer
    - Replication, replicates data
        - Good as data can be spread geographically across many computers, faster sending to people far away
    - Sharding, splits data
- CAP Principle, consistency, availability, partition tolerance
    - If you have a distributed data store, you can always have 2, but NOT 3.

    - Consistency - every read gets most recent write, else error thrown
    - Availability - every request receives a non error response, but no guarantee response contains most recent write
    - Partition Tolerance - the system can continue to operate despite arbitrary network issues

- ACID transactions
    - Locking is based on rows, tables, or sometimes single cells (weird)
    - Pessimistic model that assumes if something bad can happen, it will happen
        - Assumes one transaction has to protect itself from other transactions
        - Always locks
    - Optimistic concurrency models assume that conflicts are unlikely to occur
        - Last updated timestamp is kept on every transaction
        - If timestamps
    - Low conflict systems
        - Typically read heavy (often read only), update overnight, analytic dbs, conflicts can be fixed by roll backs
        - These can use optimistic strategy
    - High conflict systems
        - Often roll backs and rerunning transactions lower efficiency
        - These should use pessimistic scheme

- NoSQL - Not Only SQL
    - Sometimes thought of as non-relational DBs
    - Document, Graph, Key Value, Columnar, Vector
- BASE -
    - Is an ACID alternative for distributed systems
    - Basically Available
        - Guarantees CAPs availability, but may return “failure”/other error if DB is updating
    - Soft State
        - The state of the system could change over time
    - Eventual Consistency
        - The system will eventually become consistent
- Key-Value Databases
    - Simplicity
    - Good for simple CRUD operations
    - Accessing value from key = constant time
    - No complex queries - no joins, etc.
    - Very scalable - just add more nodes
        - Eventual consistency
    - Usages
        - Easy storage, raw data, use to store in between steps in model params

GRAPHS

- Made up of nodes and edges
- Label property graph
-

DOCKER COMMANDS

- In docker .yaml file:
- Docker compose up # starts from images, all services in yaml (creates containers)
- Docker compose up -d # detach, don't show logs in terminal
- Docker compose down
- Docker compose start # containers exist, start containers
- Docker compose stop
- Docker compose build
- Docker compose build –no cache # redownload all from scratch

AWS

- Compute service
  - EC2 - Elastic Cloud Compute
- Container based services
  - ECS - Elastic Container Service
  - ECR - Elastic Container Registry (Similar to docker hub)
  - EKS - Elastic Kubernetes Service
    - When load on x gets too high, creates new x to help
  - *Fargate* - Serverless container service
- Serverless
  - AWS Lambda
- Storage Services
  - Amazon S3 - Simple Storage Service
    - Store items in buckets, very scalable, different storage classes, spans regions
  - Amazon EFS - Elastic File System
    - Simple, serverless,
- Database Services
  - Relational databases
    - RDS, Aurora
  - Key - value
    - Amazon DynamoDB
  - In memory

        - MemoryDB, Amazon ElastiCache
    - Document
- Analytics Services

- Elasticity means you can easily create new instances if load is too high, delete if lower load
- EC2 Lifecycle
    - Launch - starting instance on the hardware for the first time
    - Start / stop - temporarily pause without deleting instance
    - Terminate - permanently delete instance
    - Reboot - force reboot
-