Day 2 - Data Foundations

- Most common (important?) feature of a database is searching (select in SQL)
- the baseline (worst case) for efficiency is linear (loop through all data once)
- **record -** row of a table (collection of values)
- n*x bytes of memory for records that are x bytes and n records
- **Contiguously allocated list:** all stored in a single memory chunk
- **Linked list:** reach record needs x bytes and spaces for 1-12 memory addresses (pointers)
    - memory addresses link records at the end of each record
- Python doesn't have built-in arrays
- arrays - faster for random access, slow for inserting except for at the end
- linked lists faster for inserting anywhere, slower fro random access
- **Binary search -** array of inputs in sorted order, start in middle and split, and do recursively
    - worst case: log(n) +1 (base 2)
    - best case: 1 (element is at mid)
- **linear:**
    - worst case is n
    - best case is 1 (first element)
- Generalization: In databases searching for id value is fast, but other columns are slow (because not in sorted order)

Binary Search Trees

- Use another data structure of a index for columns that need to be queried efficiently
    - binary search tree is how this is done
- Creating/inserting into binary search tree
- Tree traversals
    - pre-order
    - post-order
    - in order
    - level order - go level by level (top to bottom) and left to right within each level
        - temporarily store children of element when processing it (queue)
            - first in first out
            - can use a deque in python (can remove from front or back)
            - this is used to keep a running list of next elements to process
- For homework, question 2:
    - root = BinaryTreeNode(23)
    - root.left = BinraryTreeNode(17)
- Time complexity Insert is same as linked list, search is as good as sorted binary search
- Duplicate values

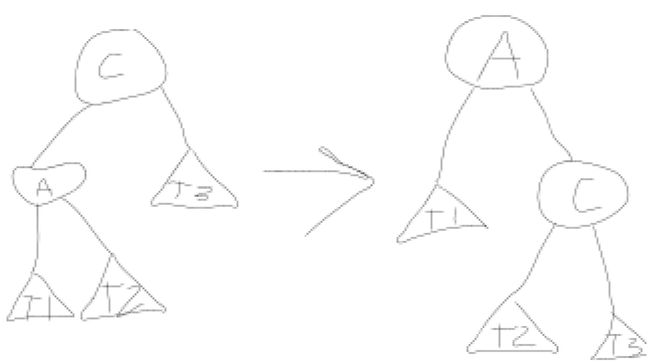
- choose that either always go left, always go right, or are ignored
- Want to minimize height of Tree (make more balanced)
    - need AVL tree for this

AVL Tree

- approx balanced binary search tree
    - each node has a balance factor
    - |h(LST) -H(RST)|<=1
        - height of left sub tree and right_sub_tree should have one or 0 level difference
- Check for imbalance in every node in path to root
- If one node is imbalance, the new node takes spot of imbalance node, and it is rebuilt from there
    - simplification
- 4 cases of imbalance

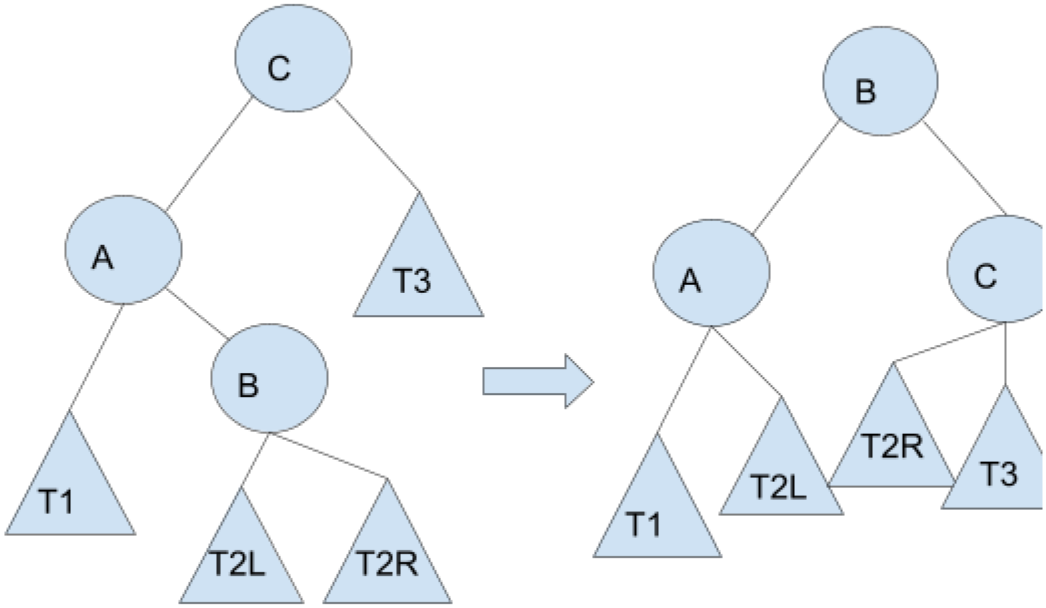
1. (LL) left left case
    a. use a single rotation
    b. Pseudocode:

        c.left = a.right
        a.right = c



2. (LR) left right case
    a. use a double rotation
    b. Pseudo code

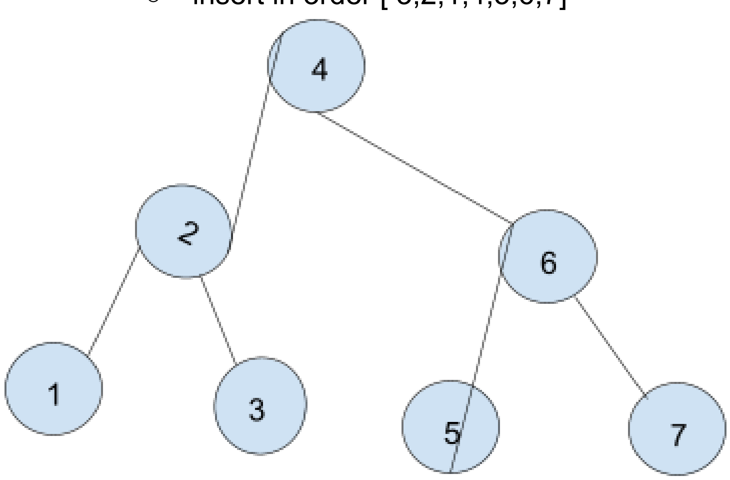
        c.left = b.a
        a.right = b.left
        b.left = a
        b.right = c



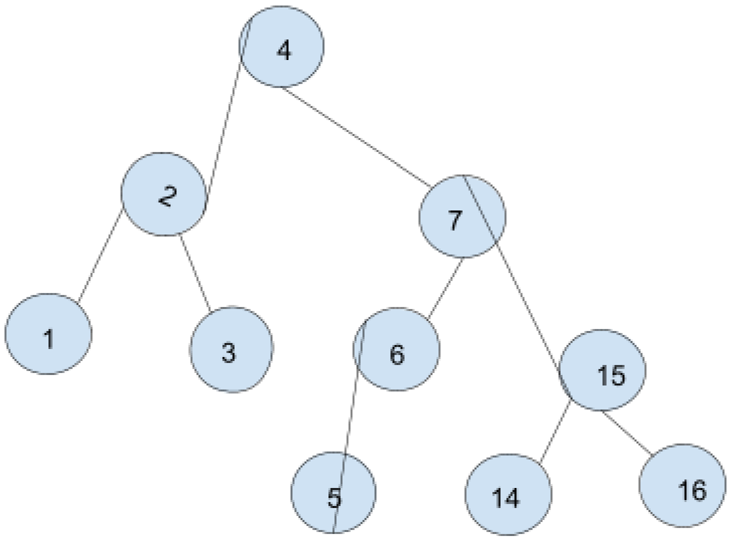
3. (RL) right left case
    a. mirror of LL (case 1)
    b. Pseudocode

        a.right = c.left
        c.left = a
4. (RR) right right case
    a. mirror of LR (case 2)
    b. Pseudocode

- When building out a BST, you start inserting at the root and follow down the tree to the correct position, then check if each position going back up is balanced
    - check the height of left and right subtrees and check the difference ≤1
    - store height as part of tree node?
        - update height when going back up subtree
    - insert in order [3,2,1,4,5,6,7]



Now insert 16, 15, 14



B+ Trees

Day 1

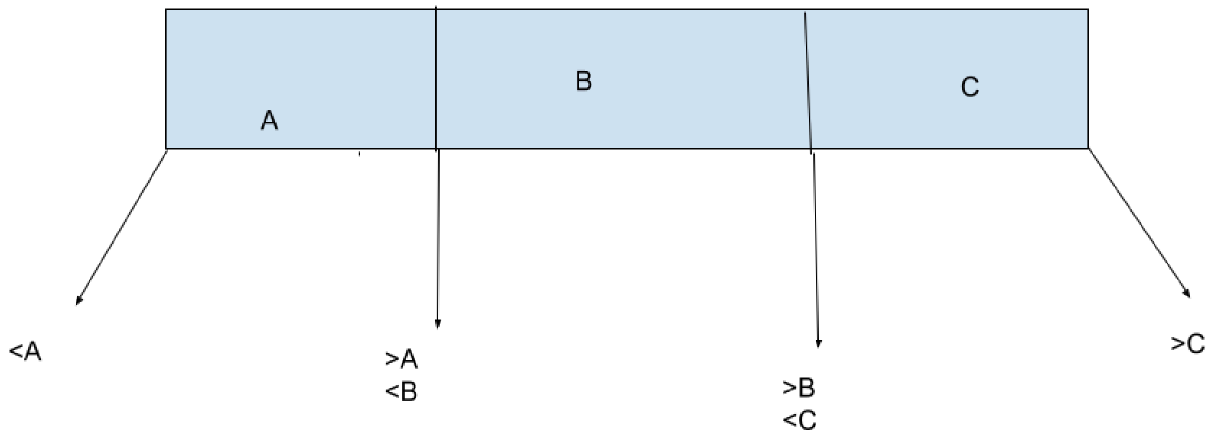
- general computer info
  - Minimize HDD/SDD accesses (slow) Hard drive access
  - CPU reads in chunks from hard drive could be 4kb block
  - things related aren't necessarily next to each other in hard drive
  - AVL trees don't optimize number of blocks read from memory

- each node will have 2 keys and up to 3 children
  - given a node with keys k1 and k2
    - one children < k1 and in between k1 and k2, and one >k2
- data must be sorted
  - this allows for groups of data equal to one block of memory stored together

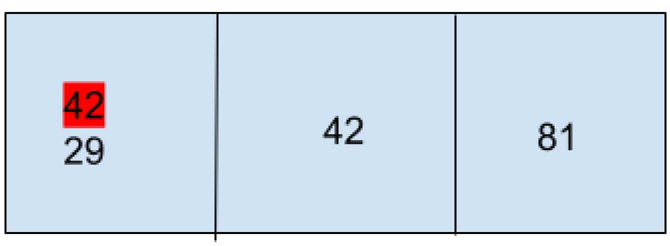
Day 2

- What is normally used in modern DBs
- goal is to minimize disc accesses
  - optimized for disk-based indexes
- B+ Tree is an m-way tree with order M
  - M = max # of keys in each node

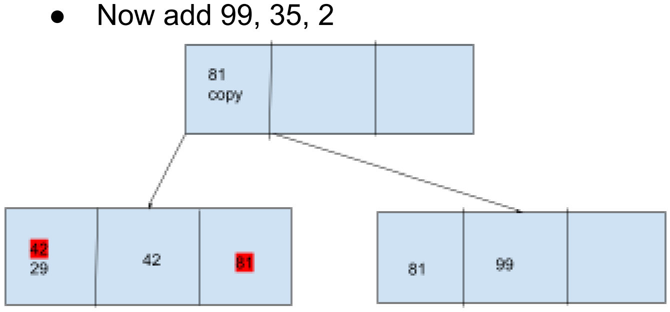
- M+1 = max children of each node
- For M=3



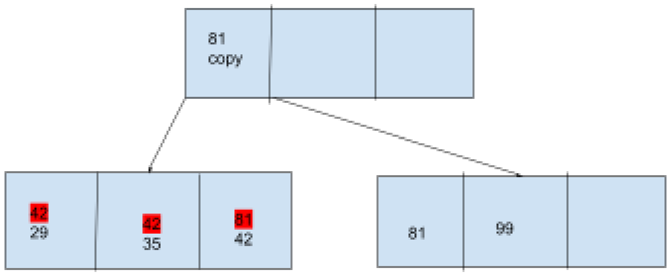
- All nodes (except the root) must be ½ full min
- root node doesn't have to be ½ full
  - this only because it starts not half full when you start inserting
- Insertions are always done at leaves
- Leaves are stored as a DLL (doubly linked list)
- Insert 42, 29, 81 in B+ Tree



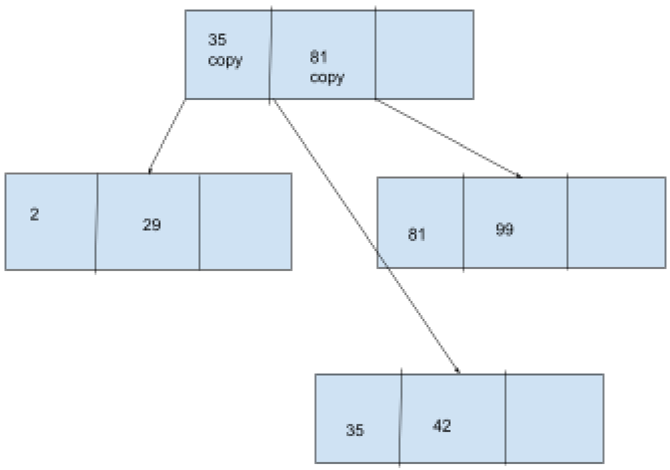
- Start at left then adjust when needed?
- **Internal nodes:** only store keys and pointers to children
- **Leaf nodes:** stores keys and data
- B and B+ trees are different
  - B trees for in-memory indexing
  - B+ trees for disk-based indexing
- Now add 99, 35, 2



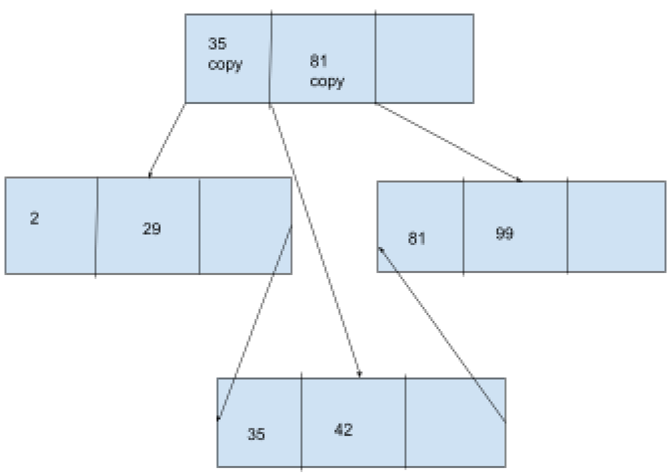
Then:



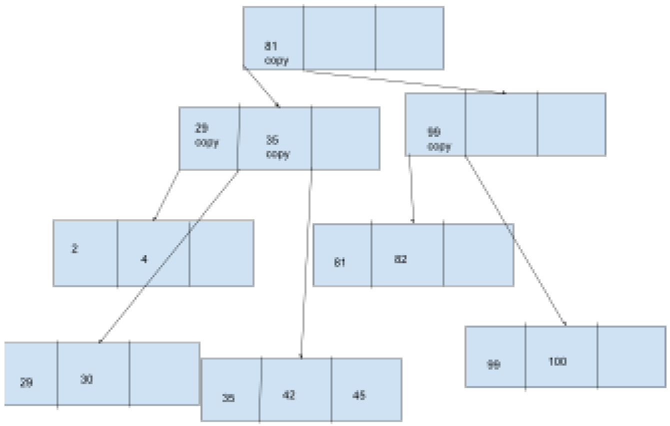
Then



- Always add new node to the right
- leaf nodes are also connected, so final from above is actualy



Now add 100, 30, 45, 82, 4



Hash Tables

- dicts are hash tables, but we can’t use them for prac 1
- key values pairs
- every time you give same input then you will get the same output
- # of inserted values = n
- table size = m
- load factor: $\lambda = \frac{n}{m}$
- constant work for every value k (key) that is being stored
- used a mod operation like $h(k) = k \, mod \, i$ where i is an prechosen number (table size)
    - this makes inserting constant time for any k
    - if already an item or multiple at stop, it becomes a list/bucket of in that location (**separate chaining)**
- You want to start with a big table size
    - want to keep load factor < .9 then increase size and rehash
- longest chain 5kv pairs, (rehash if greater?)
- can use a premade hashing function for practical (some hashing library?)
- good dispersion (don’t want hashes very clustered
- prebuilt hash function then mod with table size
-

Beyond Relational Models

- Benefits of relational models
    - (mostly) standard data model and sql is easy
    - ACID compliance (atomicity, consistency, isolation, durability)

- works well when highly structured
- easy to work with generally

- RDBMS (relational databases (MS?)) management?
- ways to increase efficiency
    - inedexing
    - column vs. row oriented
    - directly controlling storage
    - query optimization
    - chaching
    - materialized views
    - precompiled stored procedures
- Transaction processing - a sequence of multiple actions in one unit of work
- Atomicity - transaction is considered on unit (either all fails or all executes)
- Consistancy
- Isolation - two transactional happen at same time and will not affect each other
    - Dirty read - a transaction T^1 is able to read a row that has been modified transaction T2 that hasn't been committed yet
        - violation of isolation
    - Non-repeatable read
        - two queries execute a single transaction but gets different values ,beacuse one changed data
    - Oftem implemented in different ways (can be difficult to ensure without hurting performance)
    - phantom read- transaction adds or deltes rows from the set that another transaction is using
- Durability
    - once a transaction happens, then it is permanent
- why non-relational is sometimes better
    - Not all data needs ACID compliance
    - many data is semi-structured like JSON
    - joins are expensive
    - evolving schemas cause problems with relational databases
    - horizontal scaling, having multiple compute nodes is challenging
- Distributed system
    - a collection of computers working together (if one fails then the others are still ok)
    - no shared clock
- **replications**
    - multiple nodes that have everything
- **sharding**
    - split between different nodes in a cluster
- Major data systems have both sharding and replication
- distributed database can be relational or non-relational
- Network partitioning is inevitable
    - there will be the network failures and system failures

- **CAP THEOREM**
    - states that it is impossible for a data store to simultaneously provide more than two out the following three guarantees
    - **Consistency**, every read received the most recent write or error
    - **availability** - every request doesn't receive an error, no guarantee of most recent write
    - **partition tolerance**- the system can continue to operate despite arbitrary network issues
    - CA - is good for relational databases
    - CP - i good on nosql databases like mongo, redis
    - AP i things like CouchDB, Cassandra, CynamoDB

NOSQL and KV DBs

- ACID transactions focus of *data safety*
    - considered overly porte tive in some sense, assumes if something can go wrong it will go wrong
    - use locking data tables to ensure stability/security
    - pessimistic concurrency model
- Optimistic Concurrency - transaction do not obtain locks
    - assumes conflicts are unlikely and assumes things will work out
    - uses timestamp and version number on every table (reads them when changing. check and start and end of each transaction
    - good for low conflict systems
        - read heavy systems or only updated at night
        - conflicts are solved by rolling back
- Pessimistic is better if a lot of conflicts are expected a s roll back and rerunning is not efficient
- Alternative to ACID is BASE
    - Basically Available- data is usually available but sometimes the response will be "failure / unreliable"
        - systems appear to work most of the time
    - Soft State - the state of the system can change over even without updates
        - not all replicas are always the same
    - Eventual Consistency
        - the system will eventually become consistent as long as there is a break in updates
- ## KEY VALUE databases
    - operate similarly to a hash table
    - very simple data model
    - usually in-memory DB
    - very easily scalable (adding more nodes)

- operate under eventual consistency rather than absolute
- EDA/expriement data
    - can use to store intermediate results from modeling/EDA
- Feature store
    - store frequently accessed data for low latency retrieval
- MOdel monitoring
    - keep track of metrics on model performance
- shopping cart data
    - tied to the user
- user profiles/preferences
- Storing session info
- Caching layer - most frequent use case for SWE
    - short term storage on top of disk based database

- Redis
    - remote directory server
    - open source, and in memory
    - sometimes called a data structure store
    - most popular key value database (next is a dynamo DB)
    - mostly a Key Value DB but can do other stuff
    - developed in 2009 in C++
    - very very fast
    - only supports lookups by key
    - by default provides 16 database numbered 0-15
    - commands
        - SET to set something
        - GET to get something
        - SELECT to go to a new DB 0-15
        - INCR increase values by one
        - INCRBY someValue x - increase by x
        - DECR and DECRBY
    - HSET - hash set
        - other H-other command for most redis commands
        - HGET, HGETALL, HMGET, HINCRBY
    - values in redis can be linked list of string values
        - bidirectional stack/queue?
    - Queue like Ops
        - LPUSH to push on to left side of queue
            - Can do RPUSH
        - LPOP and RPOP to pop out
    - can have a set as a value
    - Set operations
        - SADD - set add
        - SINTER - intersection
        - SDIFF - difference

        - SREM
        - SRANDMEMBER - random item

Redis + Python

- redis-pu
- import redis
- to connect:
    - client = redis.Redis(host='lcoalhost', port =6379, db=2, decode_responses=True)
- similar to redis commands
    - say r is the redis option
    - r.set(key, val) to set
    - x = r.get(key) to get
    - can create pipelines r.pipeline()
        - add things to to it pipe.get().set()...
        - them rin pipe.execute() tor run
        - to decrease call over the network

Document databases

- type of noSQL database
- usually in a JSON format
- JSON stands for (JavaScript Object Notation)
    - not very space-efficient
    - human-readable
- BSON - Binary JSON
    - what MongoDB uses to store JSON
    - more space efficient, but is still structured like JSON
- XML (eXtensible Markup Language)
    - precursor of JSON
    - sort of like JSON but with HTML-like syntax
        - but can make any tags
    - specific XML query and formatting tools
        - probably don't need to know these but things
- Why doc databases
    - object persistence, in the way that objects are created in OO programming
    - don't need a schema like in a relational DB
    - you can just add new items/keys without doing tons of schema changes
        - eg online shop with different types of items with different attributes

MongoDB

- all cloud providers have a version of document DBs
  - couchDB, amazon document DB, postgres can sorta do it
- MongoDB - short for humongous database
- owned by google?
  - MongoDB Atlas is their managed DB
- No pre-defined schema (different keys in every object)
- terms
  - collection = table/view
  - document = row
  - field = column
  - embedded document = join
  - reference = foreign key
- has its own query language, that can do similar things
- mongosh → CLI tool
- compass - oen source GUI to work with it
- most languages and database apps like datagrip can interact with it
  - PyMongo
- do collection. for commands
  - .find() is like select *
  - .find({“filter key” : “filter value}) for where =
-

Graph DBs

- can be sussed to store relations and interactions
  - things like social media
- webs is a graph of pages as nodes and hyperlinks as edges
- set of nodes and vertices
- labels and nodes can both have labels (attributes)
- <u>path:</u> way to get from one node to another node
  - how to get from one to another
- <u>Connected:</u> all nodes are connected
- <u>Weighted:</u> nodes have weights/roperties
- <u>**Cyclic:**</u>
- <u>Directed:</u> edges have direction
- Hold **sparse** graph in adjacency list
- hold **dense** graph in adjacency matrix
- **Complete** graphs are when every node is connected
- **Rooted Tree** has root node and no cycles
- **Spanning Tree** get rid of nodes while still being fully connected

- **pathfinding :** shortest path between the nodes
    - algos like djkistras

Neo4j

- ACID compliant
- language is cypher
- similar to SQL but not the same
- APOC plugin
    - useful graph algos
    - awesome procedures on cypher
- Graph Data Science Plugin
    - similar to APOC
- Docker Compose
    - for managing multiple containers
    -