Foundational

- Searching is performed more than insert, update, or delete
- Select is the most versatile / complex statement
    - Only one where you can have recursive select statements
    - Baseline for searching is Linear Search
- Terms
    - **Record**
        - A collection of values for attributes of a single entity instance
            - A row in a table
    - **Collection**
        - A set of records of the same entity type
            - A Table
    - **Search Key**
        - A value for an attribute from an entity type
            - Can b more than one
- Each record -> x bytes of memory
- Then for n records
    - Need n times x bytes of memory
- Two ways of storing in primary storage
    - Contiguously Allocated List
        - All n*x bytes are in one chunk of memory
        - Better for accessing but slower for adding
    - Linked List
        - Jumping around memory
        - Each block has additional space for 1 or 2 memory addresses.
        - Individual records are linked together in a type of chain using memory addresses.
        - Slower for accessing but faster for adding
- Linear versus Binary
    - Best: Both find the first try
    - Worst:
        - Linear: Target not in array n comparisons
        - Binary: Target is not in array log base 2 n comparisons.
    - **ONLY TRUE IN A CONTINUOUS ALLOCATED LIST**
- Database Searching
    - Can’t store data on disk sorted by oth id and specialVal at the same time.
    - B/c of this we need an external data structure to support faster searching on the value than a linear scan.
- Alternatives
    - Array of tuples sorted by specialVal
        - Can use Binary Search to locate particular special Val
        - Bt inserting into this would be very slow
    - Linked list of tuples sorted by specialVal

        - Searching is slow
- Binary Search Tree
- Creating / Inserting into a BST
    - Node in tree w/o parent is called the root
    - Tree Traversals
        - PreOrder
        - PostOrder
        - InOrder
        - LevelOrder
- Degue
    - Double ended Queue
    - Can insert and remove from both the front and the back

Lecture 1

- Binary Search Tree
    - Some cases will lead to an imbalance tree
- **AVL Trees**
    - Approximately balanced BST
    -
    - Maintains a balance factor
    - AVL Balance Factor
        - Abs value of the height of the left subtree and the right subtree is less than or equal to.
    - 4 Cases of imbalance
        - **Left-Left (LL) Insertion**: This imbalance occurs when a new node is inserted into the left subtree of the left child of a node. This situation typically requires a single right rotation. For instance, inserting a new node into the left subtree of an AVL tree's leftmost branch causes a "heavy" left side, requiring a rotation around the topmost of these left nodes.
        - **Left-Right (LR) Insertion**: This type of imbalance happens when a new node is inserted into the right subtree of the left child of a node. To fix this, a double rotation is required: first, a left rotation on the left child, followed by a right rotation on the node. This situation occurs when there's an insertion that creates deeper levels on the inner side (right side of the left child) of the subtree.
        - **Right-Left (RL) Insertion**: This occurs when a new node is inserted into the left subtree of the right child of a node. This imbalance is corrected by a double rotation: first, a right rotation on the right child, followed by a left rotation on the node. It's the mirror scenario of the LR insertion, where the insertion adds depth to the inner side (left side of the right child) of the subtree.
        - **Right-Right (RR) Insertion**: This type of imbalance happens when a new node is inserted into the right subtree of the right child of a node. Similar to the LL insertion, but on the opposite side, this situation requires a single left rotation.

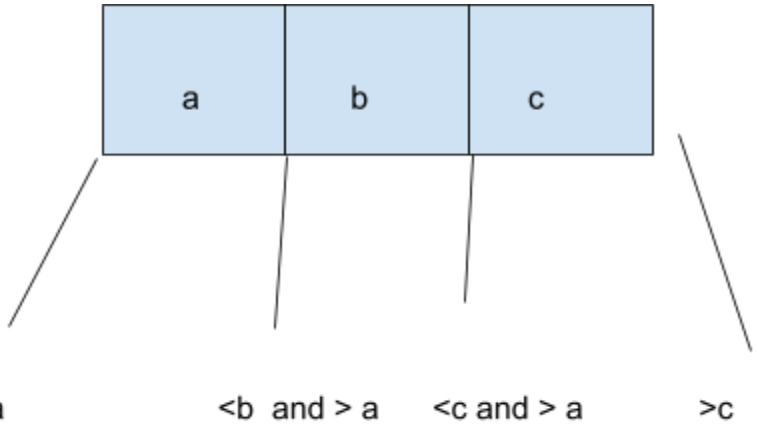
This case is typical when there's an insertion that makes the rightmost branches of the tree heavier.

Hash Tables

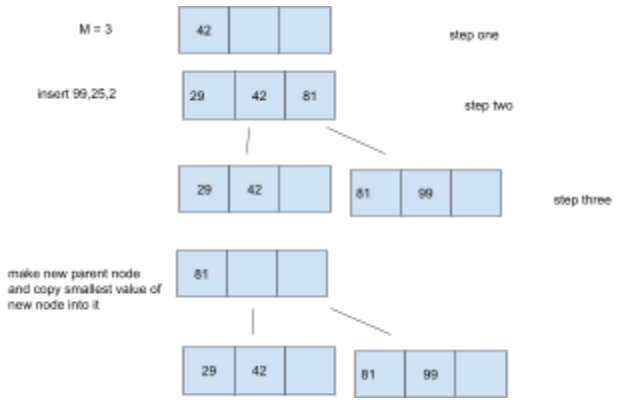
- Collection ofslots where each slot has an address or index.
- Apply hashing function to the key and produce an integer from it.
    - h(k) = k mod 6

B+ Trees

- optimised for task based indexing
- M way tree with order M
    - M is the maximum number of keys in each node
    - M+1 max children of each node
- node structure for m = 3



-
- all nodes other than the root must be ½ full minimally
- insertions done at leaves
- leaves are stored as double linked list
- Keys in nodes are kept sorted
- Internal Nodes
- leaf Nodes
- Insert 42,29,81

- 

- **B+ versus B Trees**
    - B tree node stores key adn values
        - For in memory indexing
    - B+ only leaves store key and values, internal nodes store key and pointers
        - Disk based indexing

Moving Beyond the Relational Model

- Relational Model
    - Beneffits
        - Standard Data Model and Query Language
        - ACID Compliance
            - Atomicity, Consistency, Isolation, Durability
    - Works well with structured data
    - Handle large amounts of data
    - Transation
        - Single unit of work
            - A a sequence of one or more of the CRUd operations performed as a single logical unit of work.
                - Either the entire sequence succeeds
                - OR the entire sequence fails
                - ALL or NONE
    - Efficiency
        - Indexing
        - Control Storage
        - Caching
        - Materialized views
        - Precompiled stored procedures
        - Data Replication and Partitioning
        - Column oriented storage versus row oriented storage
    - Isolation
        - Two transactions T1 and T2 are being executed at the same time but cannot affect each other.

        - If both are just reading data that is fine.
        - If T1 is reading the same data T2 is writing, can result in
            - Dirty Read
                - Transaction T is able to read a row that has been modified by another transaction T2 that has not been executed yet.
            - Non repeatable Read
                - two queries in a single translation execute a select statement but get different values because of another transaction.
            - Phantom read
                - Transaction T1 is running and T2 adds or deletes a row for set T1 is using
    - Durability
        - Once a transaction is completed and committed its changes are permanent
        - Even in the event of system failure the transaction are preserved

Distributed DBs

- ACID Transations
    - Focuses on data safety
        - Conflicts are prevented by locking the data
        - Kind of like borrowing a book from a library. If I borrow it no one else can
    - Optimistic Concurency
        - Transactions do not obtain locks on data when they read or write
        - Optimistic because assumes conflicts are nlikely to occurs
            - Even if conflicts happens its ok
        - Add last update time stamp and version number columns to every table. Then check at the end of the transaction to see if its been changed.
        - Low Conflict Systems
            - Read heavy systems
            - Conflicts that arise can be handled by rolling back and re running a transaction that notices a conflict
            - So optimistic onccurrency works well allows for higher concurrency
        - High Conflict Systems
            - rolling back a lot and rerunning transactions
    - NoSQL Databases
        - CAP Theorms
            - Can have two of three
                - Consitency
                - Availibilty
                - Partition Tolernace
        - Basically Avalible
            - Mostlikely available, but sometimes it does not.
        - Soft State

        - The state of teh system could change over tiem, even w/o input. Changes could result eventual consistency.
    - Eventual Consistency
        - The system will eventually become consistent.
    - Examples
        - Graph Databases
        - Document Databses
        - Vector Databses
- Key - Value Databases
    - Simplicity
        - Databadel is very simple
        - Retrieving a value given its key is typically a O)1_ op b/c hash tables or similar data strucs used under teh hood
        - No concept fo complex queries or joins. Slow things donw.\
    - Sleep
        - Alirght
    - Scalability
        - Just add more nodes
    - use Cases
        - Experimental Results Store
        - Feature Store
        - Model Monitoring
        - User Profiles
        - Storing Session information
        - Shopping Cart Data
        - Caching Layer
    - Redis DB
        - Primarily KV store, but cna be used with other models.
        - Considered in memory database system
            - Developed in C++
        - Very fast
            - >100000 set ops /s second
        - Does not handle complex data
        - Keys
            - Strings but any binary sequence
        - values
            - Strings
            - Lists
            - Sets
            - Sorted Sets
            - Hashes
            - Geospatial Data