Week 0 - Jan 8

- Searching is the most common operation
    - In relational DB, SELECT is the most versatile/complex
- Baseline for efficiency is linear search
- Vocab:
    - **Record** - collection of values (row of a table)
    - **Collection** - set of records of the same entity types (a table)
    - **Search Key** - a value for an attribute from the entity type
- Memory types - each record takes up x bytes, we need n * x bytes for n records
    - **Contiguously Allocated List**: all bytes are allocated as a chunk of memory
    - **Linked List** - each record uses x bytes + 1 or 2 memory addresses
        - Records are linked together with memory addresses
    - Arrays (contiguous list) faster for random access, slower for inserting at anywhere but the end (since anything after has to be moved to make space)
    - Linked lists faster for inserting anywhere, slower for random access
- Tree Traversal Types:
    - Pre order, Post order, In order, Level order (used on HW0)
- Python queue is referred to as a “degue” (double ended queue) - can insert/remove from front and back

Data Structures

- Binary search tree: each value to the left of a node is smaller, right is larger
- Balanced BST (both left & right have equal-ish lengths) has a better searching time
    - The most "imbalanced" tree would have all values on 1 side
    - This is pretty much just a sorted linked list
- Self-balancing BST is referred to as an **AVL**
- Want to minimize "secondary storage accesses" - searching
    - Inserting occurs less, so sorted data structures are preferred

B+ Trees

- M-way tree
    - M = maximum # of keys in each node
    - M + 1 = max children of each node
- Example - for m=3:
    - Four "pointers", one for each child, and 3 keys
    - First is < first key, second is between 1st and 2nd key, etc
- Properties/Mechanics:
    - Insertions done at the leaf level, not the root level
    - Leaves stored as a doubly linked list
    - Root node doesn't HAVE to be half full, unlike other types of trees
        - This is because it can have odd numbers of leaves per root
    - Keys in nodes are kept sorted

        - If a node is not full and a new data point is inserted, values in that node are shuffled to accommodate the new data point and keep it sorted
    - 2 data structures - internal nodes & leaf nodes
        - Leaf nodes only at the bottom level, internal nodes everywhere else
        - Internal nodes have no data, only leaves do
    - When you insert into a node that is full, it “splits” into 2 nodes
        - One of the old values is used as a “determining” value to decide which new node inserted values go to
        - Tree only gets a level deeper when you have to split a root node
    - When you split leaf nodes, you copy the smallest value up to the parent node
    - When you split internal nodes, you move the smallest value up
- B+ tree and B tree are different
    - In a B+ tree, the leaf nodes store only data, and internal nodes store keys & pointers
        - Disk-based memory indexing
    - In a B tree, all nodes including internal nodes store key & data values

Database Properties

- ACID properties - atomicity, consistency, isolation, durability; all good
    - Atomicity - transaction is treated as an atomic unit (process whole transaction or nothing)
    - Consistency - transaction takes a DB from one consistent state to another
        - All data meets integrity constraints (no partial transactions)
    - Isolation - transactions don’t affect each other (no race condition)
    - Durability - changes are permanent, transactions are preserved even if there is a system failure