Trees Notes

AVL Trees:

- Order of insertions matters for tree result!
- Min height = min number of comparisons to find value
- Trees are annoying to keep sorted -> need to reduce the complexity of balance requirements -> AVL tree! (self-balancing)
- AVL trees are approx balanced binary search trees
    - Maintains balance factor
    - AVL balance property: height of left sub tree and right subtree don't differ by more than one
        - |h(LST) - h(RST)| <= 1
- Use alpha to represent node of imbalance
- If one node causes imbalance, can always move one node to rebalance
- Four cases of imbalance:
    1. LL (e.g. left child of LST)
    2. LR (e.g. right child of LST)
    3. RL (e.g. left child of RST)
    4. RR (e.g. right child of RST)
- Node of imbalance is not necessarily the root of the whole tree
- Rebalancing case 1: single rotation
- Rebalancing case 2: two rotations
- Rebalancing case 4: mirror of case 1 (single rotation)
- Rebalancing case 3: two rotations
    - Either T2L or T2R will be at same level as T1/T3 but not both!
- Can keep track of height using separate height variable off to the side (squared)
- def insert(val, curr) (ex: insert(1, root.left.left)
- Only update height in diagrams once balanced!
- Think of node you're counting height from as 0, nodes with no children have height of -1 (?)

B Trees:

- SDD/HD -> RAM -> L2 cache -> L1 cache -> Registers -> CPU
    - Lots of storage, persistent, SLOW at SDD/HD level (bottom)
    - DB systems -> min HDD/SDD accesses
- 64 bit integer -> 8 bytes, 2048 byte block size is one block of hard drive
- (key, value) pairs are 32 bytes; exist in memory
- Balance is important because it reduces the number of nodes/amount of memory that needs to be searched -> lower height means faster search
- Sorted array of 128 integers (128 * 8 = 1024 bytes)

    - Worst case binary search on 128 integers is way faster than a single additional disk access
    - More keys allows you to do less disk accesses (less deep tree to search) -> faster!
- B+ tree allows you to search more keys in fewer levels compared to AVL trees; fewer reads from disk so faster
    - Want to minimize height of the tree for indexing data!

B+ Trees:

- Optimized for disk-based indexing; minimizing disk accesses for indexing
- B+ tree is an m-way tree with order M
    - M -> maximum number of keys in each node
    - M + 1 -> maximum number of children of each node
- All nodes (except the root) must be ½ full minimally
- Root node does not have to be half full
- Insertions are always done at leaves first
- Leaves are stored as doubly-linked lists for easy searching (don't need to keep going up and back from root for each search)
- Keys in nodes are kept sorted
- Internal nodes -> only store key values + pointers to children (act like an index)
- Leaf nodes -> store keys and data
- B trees are not the same as B+ trees
    - In B trees, nodes store keys and values and are for in-memory indexing
    - In B+ trees, only leaves store keys and values and they are for disk-based indexing
- Only leaf nodes contain actual data, internal nodes are just used for indexing (ex: everything to the left of 81 in root will be less, everything to the right will be more)
    - Go to internal node looking for pointer, get to leaf node in search mode for where to put the value numerically
- Always insert new node to the right of existing node
- Tree only gets a level deeper when you have to split from root node
- A node is considered half full based on the number of children it has
- If you split internal node (ex: root) to put in a number, smallest value of new node is not copied, it is moved to a new node by itself

Hash Tables:

- Like Python dictionary
- # inserted vals = n
- Table size = m
- Lambda load factor = n/m

- Uses (key, value) pairs
- Possible format of hash table: Numpy array of Python lists each containing (k, v) pairs
- h(k) = k mod [# of slots]
    - Constant work for any k (less than AVL tree), good dispersion!
    - h(k) = 7 -> worst version of dispersion for a table with 6 slots, all pairs in same slot
- Ex: 10 mod 6 = 4 -> (10, cat) in slot 4; 20 mod 6 = 2 -> (20, dog) in slot 2; 2 mod 6 = 2 -> add to slot 2 to create list
- Target lambda load factor is less than 0.9
- When table is large, there is a higher probability each bucket has fewer items in it
- For a table with 6 slots, the longest chain is 5 kv pairs -> “essentially” constant time (searching through 5 things instead of look/full table) -> if you know 20 mods to 2, can go straight to slot 2’s bucket
- When lambda exceeds 0.9, need to make new table and re-index each k, v pair individually using new hash function
    - Need to always mod the mod function by m itself (size of table)!!
        - Can use built-in library to hash, just be sure to still mod by table size at the end on the output
- Say we have the data (finance 7.json), (money 10.json), (bank 15.json), (money 7.json)
    - A made-up hash function generates sample hash values for each word: 381, 767, 951, 767, respectively
    - Mod by table size next! -> 381 mod 10 = 1 (goes to slot 1), 767 mod 10 = 7 (goes to slot 7), 951 mod 10 = 1 (goes to slot 1)
    - Slot 1: (finance, (7.json)), (bank, (15.json)) ; Slot 7: (money, (10.json, 7.json))
- Ideas to improve speed: take pre-processed text and put into set to remove duplicates, map filenames to short integers (ex: instead of 7.json, use 7)