Hashmaps

- Like a python dictionary
- Collection of each slots; each slot has an "address"
- Given a key, apply some hashing function to it that maps it to an integer from 0 to n - 1, where n is the size of the table
- To ensure that the hash function maps to an integer between 0 and n - 1, you can mod it by the table size
- Many values may map to the same slot, therefore we have to store the key and the value to get the information we're looking for
- The load factor λ can be computed by dividing the the number of inserted values (n) by the table size (m)
    - $\lambda = n / m$
    - We usually like to keep the load factor under a particular threshold
- Why would we use hash tables over AVL trees?
    - Hash functions take O(1) time and passing the same input results in the same output each time, making it consistent across insertion and searching
    - Insertion can be done in O(1) time, making it faster than AVL insertion
- Collisions - occurs when two keys are inserted into the same slot
    - Resolutions
        - Not insert
        - Look for next open space (open addressing)
        - Make the value a list of values (separate chaining)
- How to choose the best table size?
    - Start with some set size
    - With each insertion, check the load factor, and resize if needed
    - When you do need to resize, multiply the original table size by some factor k
    - You have to re-hash every time you resize the table
- Hash Table efficiency
    - Searching in a large table with short chains is practically constant time
        - Finding one slot out of 100,000 = O(1)
        - Once you find a slot, finding one key out of maximum five key-value pairs is also constant time, similar to searching in an AVL tree with height log(100,000)
    - We want hash tables with good dispersion (broad distribution)

B+ Trees

- B+ Trees are the most common indexing structure used in RBDs

- Optimized for disk-based accessing
- A B+ tree is an m-way tree with order M
    - M is the maximum number of keys in each node
    - M+1 is the maximum number of children that each node can have
    - Node structure for M = 3
        - A, B, C are keys, where A < B < C
        - The left and right children of the keys are pointers.
            - The left pointer of A contains keys that are < A. The right pointer of A contains values that are ≥ A, but < B. The right pointer of B contains values that are ≥B but < C. The right pointer of C contains values that are ≥ C.
- Properties of a B+ Tree
    - All nodes (except the root) must be at least half full (of children)
    - Root node does not have to be half full
    - Insertions are always done at the leaf level
    - Leaves are stored as a doubly-linked list
    - For the same set of values, a B+ tree will be shallowers, but wider, than a BST
    - Within a node, keys ar kept sortd
- Insertion in a B+ Tree
    - The insertion process involves adding new keys to the appropriate leaf nodes while maintaining the tree's balanced properties.
    - Steps for Insertion in B+ Tree:
        - Locate the Appropriate Leaf Node:
            - Begin by traversing the tree from the root to find the correct leaf node where the new key should be inserted.
        - Insert the Key:
            - If the leaf node has fewer than the maximum allowed keys, insert the new key in the correct position to maintain sorted order.
            - If the leaf node is full, split it into two nodes:
            - Distribute the existing keys and the new key between the two nodes evenly.
            - Copy the middle key (median) to the parent node to act as a separator between the two new nodes.
        - Handle Parent Node:
            - If the parent node also becomes full due to the insertion of the median key, repeat the splitting process recursively up the tree.

- If the splitting reaches the root and it becomes full, create a new root node, increasing the height of the tree.