Hashing Cheatsheet

Topic Overview

Hashing in Java uses hash functions to map keys to values, enabling efficient operations via HashMap and HashSet. This cheatsheet details hashing techniques and their applications.

Prerequisites

Arrays, Strings

List of Subtopics

- Hash Tables
- Maps
- Sets
- Frequency Counting
- Collision Resolution
- Custom Hash Function
- Hash Map with Chaining
- Open Addressing (Linear Probing)

Key Concepts Explained

- **Hash Tables**: Structures that use a hash function to index key-value pairs in an array.
- **Collision Resolution**: Techniques to handle multiple keys hashing to the same index.
- **Frequency Counting**: Method to track element occurrences using a hash-based approach.

Approaches to Solve Problems with Step-by-Step Algorithms

- **Hash Tables**:
  - **Algorithm**:
    1. Define an array of linked lists or buckets based on a chosen size.
    2. Apply a hash function to the key to compute an index.
    3. Store the key-value pair in the bucket at that index.
    4. For retrieval, rehash the key and search the corresponding bucket.
    5. Handle edge cases like resizing when the load factor exceeds a threshold.
  - **Context**: Provides O(1) average-time access, with O(n) worst-case for collisions.
- **Maps**:
  - **Algorithm**:
    1. Use a HashMap to associate keys with values.
    2. Hash the key to determine the bucket index.
    3. Insert or update the value in the bucket, overwriting if the key exists.
    4. For search, rehash the key and traverse the bucket to find the value.
    5. Remove by locating and deleting the key-value pair.
  - **Context**: Ideal for key-value storage, with built-in Java support.
- **Sets**:
  - **Algorithm**:
    1. Initialize a HashSet to store unique elements.
    2. Hash each element to assign it to a bucket.
    3. Add only if the element is not present, using a contains check.
    4. For membership test, hash and search the bucket.
    5. Remove by hashing and deleting the element if found.
  - **Context**: Ensures uniqueness, useful for duplicate detection.
- **Frequency Counting**:
  - **Algorithm**:
    1. Create a HashMap with elements as keys and counts as values.
    2. Iterate through the data, incrementing the count for each key.
    3. For each new element, add it with an initial count of 1 if absent.
    4. Access the count for any element by looking up its key.
    5. Process the map for analysis (e.g., find max frequency).
  - **Context**: Optimizes counting tasks, with O(n) time for a single pass.
- **Collision Resolution**:

  - **Algorithm**:
    1. For chaining, link colliding keys in a linked list at the same index.
    2. Traverse the list to handle insertions, updates, or searches.
    3. For open addressing, start at the hashed index.
    4. Probe the next slot linearly or with a step function until empty or matching key.
    5. Update or insert at the found slot.
  - **Context**: Manages hash conflicts, with chaining preferred for high load factors.

- **Custom Hash Function**:
  - **Algorithm**:
    1. Analyze key properties (e.g., character sum for strings, field combination for objects).
    2. Design a function to distribute keys evenly across the table size.
    3. Use modulo with a prime number to map to indices.
    4. Test for collisions and adjust the function if clustering occurs.
    5. Handle special cases like null or duplicate keys.
  - **Context**: Customizes hashing for specific data types, requiring even distribution.

- **Hash Map with Chaining**:
  - **Algorithm**:
    1. Initialize an array of linked lists with a specified size.
    2. Hash the key to determine the bucket index.
    3. Append the key-value pair to the list at that index.
    4. For search or update, hash the key, traverse the list, and perform the operation.
    5. Resize the array and rehash all entries if the load factor is high.
  - **Context**: Handles collisions effectively, with O(1) average time.

- **Open Addressing**:
  - **Algorithm**:
    1. Hash the key to get the initial index.
    2. If occupied, probe the next slot using a linear or quadratic step.
    3. Continue probing until an empty slot or the original key is found.
    4. Insert or update at the empty slot, marking deletions appropriately.
    5. Search by repeating the probe sequence.
  - **Context**: Saves space but requires careful load factor management.

Common LeetCode Problems with Approaches

- **Two Sum (1)**: Iterate through the array, for each element, compute the complement for the target, and check a HashMap for its presence, adding elements as you go.
- **Contains Duplicate (217)**: Use a HashSet, iterating through the array and attempting to add each element; if add fails, a duplicate exists.
- **Group Anagrams (49)**: Sort characters of each string to form a key, use a HashMap to group strings with identical sorted keys.
- **Valid Anagram (242)**: Use two HashMaps to count character frequencies in both strings, then compare the maps for equality.
- **Longest Consecutive Sequence (128)**: Add all numbers to a HashSet, then for each number, check consecutive values to find the longest sequence.
- **Subarray Sum Equals K (560)**: Use a HashMap to store cumulative sum frequencies, iterating and checking if (sum - k) exists for each position.

Time & Space Complexities

- Access: O(1) average, O(n) worst (collisions)
- Insert: O(1) average, O(n) worst
- Delete: O(1) average, O(n) worst
- Space: O(n) for table, O(k) for chaining

Important Tips & Tricks

- Leverage Java's HashMap and HashSet for built-in efficiency.
- Select a prime table size to reduce clustering in custom implementations.
- Handle null keys with a default value in custom hash functions.
- Optimize frequency counting with a single pass for multiple queries.
- Use chaining for datasets with frequent collisions.