# Documentation AllOne Data Structure Documentation

## Table of Contents



## 1. Problem Statement

We need to design a data structure that supports the following operations efficiently:

- inc(key): Increments the count of the string key by 1. If the key does not exist, insert it with count 1.
- dec(key): Decrements the count of the key by 1. If key's count reaches 0, remove it from the data structure.
- getMaxKey(): Returns any one of the keys with the maximum count.
- getMinKey(): Returns any one of the keys with the minimum count.

Each function must run in O(1) average time complexity.

## 2. Intuition

To achieve O(1) operations:

- Use a hashmap (key_count) to track the count of each key.

- Maintain a doubly linked list (DLL) (freq_list) where each node represents a unique frequency and contains a set of keys with that frequency.
- Store mappings from key to its frequency node (key_to_node) to enable constant-time updates.

## 3. Key Observations

- The problem requires keeping track of the min and max frequency keys efficiently.
- A DLL allows O(1) insertion and deletion while maintaining an ordered sequence of frequency nodes.
- A hashmap provides quick lookup of a key's count and its corresponding node in the list.

## 4. Approach

Data Structures Used

1. key_count (dict): Maps a key to its count.
2. key_to_node (dict): Maps a key to its node in the freq_list.
3. freq_list (Doubly Linked List):
    - Each node in this list represents a frequency value and maintains a set of keys with that frequency.
    - The list is maintained in increasing order of frequency.

Operations

Increment (inc(key))

1. If key exists:
    - Remove key from its current frequency node.
    - Move key to the next frequency node (count + 1).
2. If key does not exist:
    - Insert key into the 1-frequency node.
3. Create a new node if the target frequency node does not exist.
4. Delete the old frequency node if it becomes empty.

Decrement (dec(key))

1. Remove key from its current frequency node.
2. If the count becomes 0, delete key from key_count and key_to_node.
3. Otherwise, move key to the count - 1 node.
4. Delete the old frequency node if it becomes empty.

Get Maximum Key (getMaxKey())

- The last node in freq_list contains the maximum frequency keys.
- Return any key from this node.

Get Minimum Key (getMinKey())

- The first node in freq_list contains the minimum frequency keys.
- Return any key from this node.

## 5. Edge Cases

- Calling getMaxKey() or getMinKey() on an empty data structure should return an empty string.
- inc(key) on a new key should correctly insert it into freq_list.
- dec(key) should correctly remove key if the count reaches 0.
- Handling duplicate keys in inc() and dec().
- Ensuring O(1) operations even with frequent insertions and deletions.

## 6. Complexity Analysis

Time Complexity

- inc(key): O(1) (Hashmap lookup + possible DLL move)
- dec(key): O(1) (Hashmap lookup + possible DLL move)
- getMaxKey(): O(1) (Retrieve from the last node of DLL)
- getMinKey(): O(1) (Retrieve from the first node of DLL)

Space Complexity

- O(N) where N is the number of unique keys stored.

7. **Alternative Approaches**

- Using Only HashMaps:
    - Store (key, count) in one hashmap and (count, keys) in another.
    - Would require iterating over keys to find min/max, breaking O(1) complexity.
- Using Heap (Priority Queue):
    - Maintaining a min/max heap would allow quick access to min/max elements.
    - However, heap operations are O(log N), making them slower than O(1) solutions.

The DLL + HashMap approach is optimal.

8. **Test Cases**

- allOne = AllOne()
- allOne.inc("hello")
- allOne.inc("hello")
- assert allOne.getMaxKey() == "hello"
- assert allOne.getMinKey() == "hello"
- allOne.inc("leet")
- assert allOne.getMaxKey() == "hello"
- assert allOne.getMinKey() == "leet"
- allOne.dec("hello")
- assert allOne.getMaxKey() == "hello"

9. **Final Thoughts**

This implementation efficiently supports O(1) operations for incrementing, decrementing, and retrieving the min/max key using **a doubly linked list and hashmaps**. It is optimal compared to heap-based or brute-force solutions. 🚀