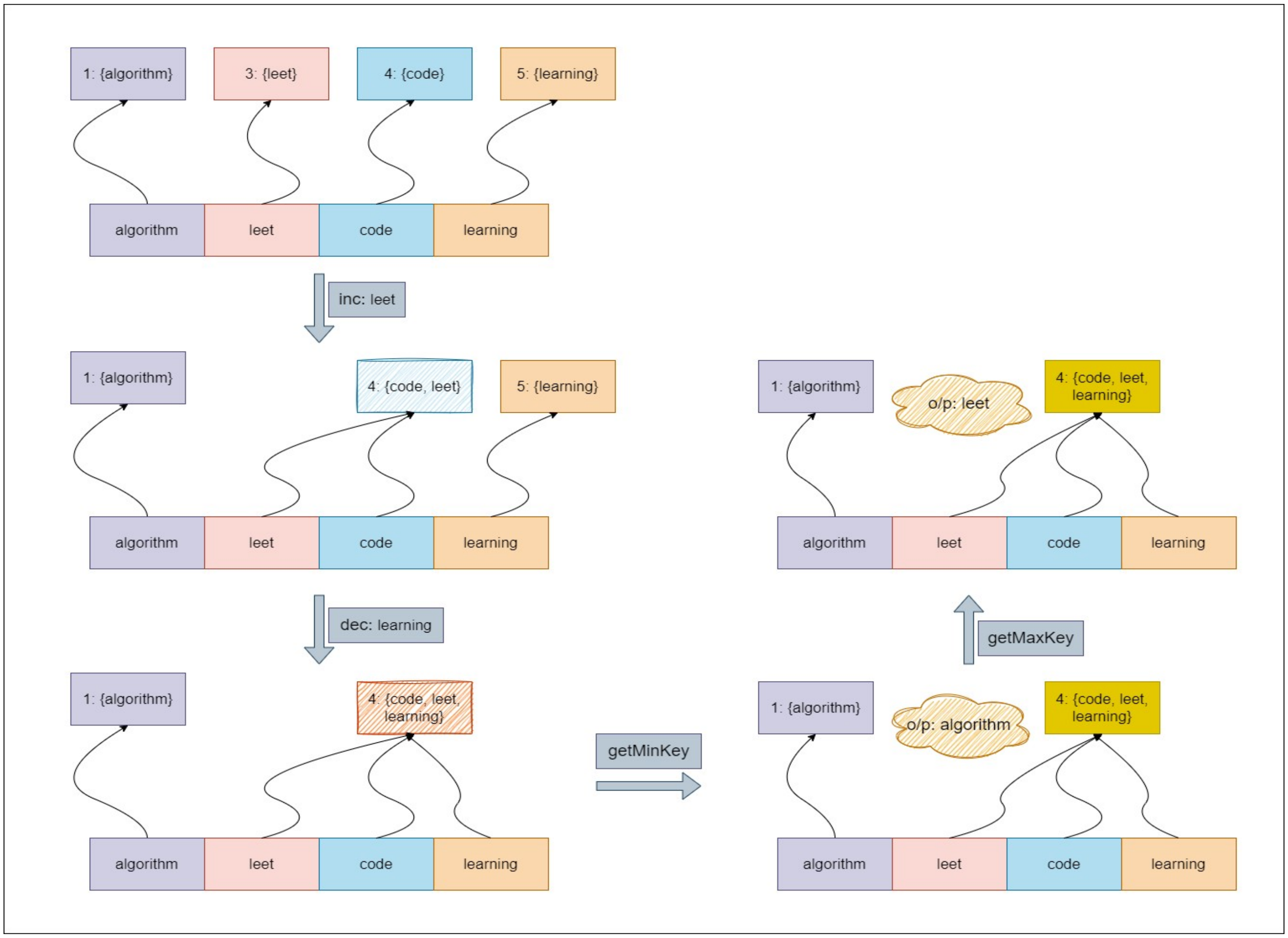
432. All O`one Data Structure

leetcode editorial

Overview:

We need to create a specialized data structure that efficiently handles the following operations on strings and their associated counts:

Increase the count of a specified string.
Decrease the count of a specified string.
Retrieve the string with the highest count.
Retrieve the string with the lowest count.
A key requirement is that each of these operations must be performed in constant time, $\Theta(1)$ on average.



Approach: Using Doubly Linked List

Intuition

To manage a collection of keys and their frequencies, we need a structure that updates easily and provides quick access to maximum and minimum frequencies. We start with a hashmap to look up each key’s frequency quickly.

However, a hashmap alone does not track frequencies well. We need a way to group keys by their frequencies and find keys with the same frequency. We use a doubly linked list for this. Each node represents a frequency and holds all keys linked to that frequency. This setup allows us to add and remove keys efficiently as their frequencies change.

To handle edge cases better, we include dummy head and tail nodes in the list. These nodes make it easier to manage operations when the list is empty or when we add or remove nodes at the ends.

When we increment a key, we first check if it exists in the hashmap. If the key is new, we look at the node after the dummy head. If that node does not have a frequency of 1, we create a new node for frequency 1. We add the key to this node and update the hashmap. If the key already exists, we find its current frequency node and check the next node, which shows the next higher frequency. If that next node is the tail or does not have the expected frequency, we create a new node with the increased frequency. We then move the key to the right node, remove it from the old node, and delete the old node if it becomes empty.

When we decrement a key, we first check if it is in the hashmap. If it is, we remove it from its current node. If the key’s frequency is greater than one, we check the previous node. If needed, we create a new node for the decreased frequency and add the key to the appropriate previous node, updating the hashmap. If the frequency is one, we remove the key from the hashmap completely.

To find the key with the maximum frequency, we return one of the keys from the last node in the list. For the minimum frequency key, we get a key from the first node after the dummy head. If there are no keys, we return an empty string.

Algorithm

- `Node` Class:
- Each `Node` contains:
    - `freq`: the frequency of the keys.
    - `prev`: a pointer to the previous node.
    - `next`: a pointer to the next node.
    - `keys`: a set of strings representing the keys with this frequency.
- The constructor initializes the `freq`, and sets `prev` and `next` to `nullptr`.
- `AllOne` Class:
    - Create a dummy head node and a dummy tail node.
    - Link the dummy head to the dummy tail and vice versa.
    - Incrementing a Key (`inc` function):
        - If the key already exists:
            - Retrieve the corresponding `node` from the `map`.
            - Erase the key from the current `node`.
            - Check the next node:
                - If it doesn't exist or its frequency is not `freq + 1`:
                    - Create a new node with frequency `freq + 1`.
                    - Insert the key into this new node.
                    - Link the new node with the current and next nodes.
                    - Update the `map` to point to the new node.
                - Otherwise, insert the key into the existing next node.
            - If the current node has no keys left, remove it.
        - If the key does not exist:
            - Check the first node after the head:
                - If it doesn't exist or its frequency is greater than `1`:
                    - Create a new node with frequency `1`.
                    - Insert the key into this new node.
                    - Link this new node with the head and the first node.
                - Otherwise, insert the key into the first node.
    - Decrementing a Key (`dec` function):
        - If the key does not exist in the `map`, return immediately.
        - Retrieve the node corresponding to the key.
        - Erase the key from the current node.
        - If the frequency is `1`:

            - Remove the key from the `map`.
        - Otherwise, check the previous node:
            - If it doesn’t exist or its frequency is not `freq - 1`:
                - Create a new node with frequency `freq - 1`.
                - Insert the key into this new node and link it with the current node and the previous node.
            - Otherwise, insert the key into the existing previous node.
        - If the node has no keys left, remove it.
    - Getting the Maximum Key (`getMaxKey` function):
        - If there are no keys (i.e., the tail's previous node points to the head), return an empty string.
        - Return one of the keys from the tail's previous node.
    - Getting the Minimum Key (`getMinKey` function):
        - If there are no keys (i.e., the head's next node points to the tail), return an empty string.
        - Return one of the keys from the head's next node.
- Removing a Node (`removeNode` function):
    - Link the previous node to the next node and vice versa to remove the specified node from the linked list.
    - Delete the removed node to free its memory