432. All O`one Data Structure

Design a data structure to store the strings' count with the ability to return the strings with minimum and maximum counts.

Implement the `AllOne` class:

- `AllOne()` Initializes the object of the data structure.
- `inc(String key)` Increments the count of the string `key` by `1`. If `key` does not exist in the data structure, insert it with count `1`.
- `dec(String key)` Decrements the count of the string `key` by `1`. If the count of `key` is `0` after the decrement, remove it from the data structure. It is guaranteed that `key` exists in the data structure before the decrement.
- `getMaxKey()` Returns one of the keys with the maximal count. If no element exists, return an empty string `""`.
- `getMinKey()` Returns one of the keys with the minimum count. If no element exists, return an empty string `""`.

Note that each function must run in `O(1)` average time complexity.

Example 1:

```
Input
["AllOne", "inc", "inc", "getMaxKey", "getMinKey", "inc", "getMaxKey", "getMinKey"]
[[], ["hello"], ["hello"], [], [], ["leet"], [], []]
Output
[null, null, null, "hello", "hello", null, "hello", "leet"]
Explanation
AllOne allOne = new AllOne();
allOne.inc("hello");
allOne.inc("hello");
allOne.getMaxKey(); // return "hello"
allOne.getMinKey(); // return "hello"
allOne.inc("leet");
allOne.getMaxKey(); // return "hello"
allOne.getMinKey(); // return "leet"
```

Constraints:

- `1 <= key.length <= 10`
- `key` consists of lowercase English letters.
- It is guaranteed that for each call to `dec`, `key` is existing in the data structure.
- At most `5 * 104` calls will be made to `inc`, `dec`, `getMaxKey`, and `getMinKey`.

432. All O`one Data Structure

```
/*
   Doubly linked list+Hash set+Hash map
   Time compexlity:  Θ(1),O(n)
   soace compexlity:  Θ(n),O(n)
*/
class AllOne {
   private:
      class Node{
         public:
            int freq;
            Node* prev;
            Node* next;
            unordered_set<string> keys; // List of keys in each node

            Node(int freq) : freq(freq),prev(nullptr),next(nullptr) {}
      };

      Node* head; // Dummy head
      Node* tail; // Dummy tail
      unordered_map<string, Node*> which_node;  // Mapping each key to its node

   public:
      AllOne(){
         head = new Node(INT_MIN);  // Create dummy head
         tail = new Node(INT_MAX);  // Create dummy tail
         head->next=tail;   // Link dummy head to dummy tail
         tail->prev=head;   // Link dummy tail to dummy head
      }

      // Remove a node from the list
      void remove_node(Node* node){
            Node* prev_node = node->prev;
            Node* next_node = node->next;

            prev_node->next=next_node;
            next_node->prev=prev_node;

            delete node;
      }
```

// Function *inc(string)* : Inserts a new key `Key` with value 1. Or increments an existing key by 1.

```
void inc(string key){
        // If the new key `key` exists
        if (which_node.find(key)!=which_node.end()){
            Node* node = which_node[key]; // Get its node
            int freq = node->freq; // Get its frequency
            node->keys.erase(key);  // Remove it from current node's list

            Node* next_node = node->next; // Get its next node
            if (next_node==tail || next_node->freq!=freq+1) {
                // Create a new node if next node does not exist...
                // ... or freq is not freq+1
                Node* new_node=new Node(freq+1);
                new_node->keys.insert(key);
                new_node->prev=node;
                new_node->next=next_node;
                node->next=new_node;
                next_node->prev=new_node;
                which_node[key]=new_node;
            }
            else{
                // How to Increment the freq of the new key `key`?
                // Just put the new key in the next node's list of keys
                next_node->keys.insert(key);
                which_node[key]=next_node;
            }
            // Remove the current node if it has no keys left
            if (node->keys.empty()){
                remove_node(node);
            }
        }
        else{  // Key does not exist
            Node* first_node = head->next; // Get the first node
            if (first_node==tail || first_node->freq>1) {
                // Create a new node if the first node does not exist...
                // ... or freq not 1
                Node* new_node = new Node(1);
                new_node->keys.insert(key);
                new_node->prev=head;
                new_node->next=first_node;
                head->next=new_node;
                first_node->prev=new_node;
                which_node[key]=new_node;
            }
            else{ // If all keys in the first node have a frequency of 1
                first_node->keys.insert(key);
                which_node[key]=first_node;
            }
        }
    }
```

// Function *dec*(*string*) : Decrements an existing key by 1. If Key's value is 1, remove it from the data structure.

```
void dec(string key){
    // Key does not exist
    if (which_node.find(key)==which_node.end()) return;

    // Key exists
    Node* node=which_node[key]; // Get its node
    int freq=node->freq; // Get its frequency
    node->keys.erase(key); // Remove it from current node's list



    if (freq==1){
       // Remove the key from the map if freq is 1
       which_node.erase(key);
    }
    else{ // Otherwise, put it in the previous node's list of keys
       Node* prev_node = node->prev;
       if (prev_node==head || prev_node->freq!=freq-1){
          // Create a new node if the previous node does not exist or freq
          // is not freq-1
          Node* new_node=new Node(freq-1);
          new_node->keys.insert(key);
          new_node->prev=prev_node;
          new_node->next=node;
          prev_node->next=new_node;
          node->prev=new_node;
          which_node[key]=new_node;
       }
       else{
          // How to decrement the freq of the `key`?
          // Just put the new key in the previous node's list of keys
          prev_node->keys.insert(key);
          which_node[key]=prev_node;
       }
    }

    // Remove the node if it has no keys left
    if (node->keys.empty()) {
       remove_node(node);
    }
}
```

```
// Returns one of the keys with maximal value.
    string getMaxKey(){
        if (tail->prev == head) return "";
        return *(tail->prev->keys.begin());
    }

    // Returns one of the keys with minimal value.
    string getMinKey(){
        if (head->next == tail) return "";
        return *(head->next->keys.begin());
    }

};
```