LFU CACHE

Implement a cache using suitable data structures with 2 functions. get function takes key as an input and returns the corresponding value as output if it exists.

put function takes a key and input as input, if the key already exists in the cache and updates its value and if it doesnt, we add it to the cache (provided size is not maxxed out). If size is already maxxed out, we remove the least frequently used node. If more than 1 nodes is at the least used frequency, we remove the LRU among them.

Since there is a combination of LFU and LRU, we will use Doubly Linked Lists for each frequency and will also maintain a key, value maps to maintain O(1) lookups.

Pseudocode :

```
struct Node {
    int key, value, count;
    Node * next;
    Node * back;
    Node (int k, int v) {
```

```
        key = k;
        value = v;
        count = 1;
    }
};
struct List {
    int size;
    Node * head;
    Node * tail;
    List() {
        head = new Node(0, 0);
        tail = new Node(0, 0);
        head->next = tail;
        tail->prev = head;
        size = 0;
    }
    void addFront(Node * N) {
        Node* t = head->next;
        N->next = t;
        N->prev = head;
        head->next = N;
        t->prev = N;
        size += 1;
    }
    void removeNode(Node * delNode) {
        delNode->prev->next = delNode->next;
        delNode->next->prev = delNode->prev;
        free(delNode);
        size -= 1;
    }
};
class LFUCache {
public:
```

```
int capacity ;
int freq , size ;
unordered_map < int , List*> fMap ;
unordered_map <int , Node * > lMap ;
LFUCache (int c) {
    capacity = c ;
    freq = 0 ;
    size = 0 ;
}
void updateFreqList (Node * n) {
    fMap.erase(n → key) ;
    lMap [n → count] → removeNode(n) ;
    if (n → count == freq && lMap [n → count
                                → size == 0){
        freq ++ ;
    }
    List * nextH = new List() ;
    if (lMap.find (n → count +1) != lMap.end()
        nextH = lMap [n → count +1] ;
    }
    n → count += 1 ;
    nextH → addFront (n) ;
    lMap [n → count] = List ;
    fMap [n → key ] = n ;
}
int get (int key) {
    if (fMap.find (key) != fMap.end()) ;
        Node * n = fMap [key] ;
        int v = n → value ;
        updateFreqList (n) ;
        return v ;
    } else {
        return -1 ;
    }
}
```

```
void put(int key, int value) {
    if (capacity == 0) {
        return;
    }
    if(fMap.find(key) != fMap.end());
        Node * n = fMap[key];
        n->value = value;
        updateFreqList(node);
    } else {
        if (size == capacity) {
            List * L = lMap[freq];
            fMap.erase(L->tail->prev->key);
            lMap[freq]->removeNode(L->tail->
                                   prev);
            size--;
        }
        size++;
        freq = 1;
        List* newL = new List();
        if (lMap.find(freq) != lMap.end()){
            newL = lMap[freq];
        }
        Node * n = new Node(key, value);
        newL->addFront(n);
        fMap[key] = n;
        lMap[freq] = newL;
    }
}
};
```