LFU Cache

Question here is to remove the LFU element if cache is full. If get or put is called and element exist then freq is updated every time. For this question we maintain two data structures.

unordered_map<int,Node*> mp which stores the address of every node.

unordered_map<int, DLL*> fq which stores the freq and the nodes that have frequency equal to the freq will be stores as doubly linked list in this structure with the LRU at the tail and MRU at the head of the linked list.

- get ( )- If element is present store the value, updatefreq and return value else return -1;
- put ( )- If element is present update freq else check whether or not space is available if currSize<capacity then simply set minfreq=1 because this element was not found previously and attach it to fq[1] and update the mp[key]=newNode else if capacity is full then remove the LRU node and decrease the size also delete the entry in mp because the node is deleted.

While adding a node a list is created and it is checked whether it exists in fq if yes then this newList is assigned to existing list and then the newNode gets added. We also update the mp[key] and then at last set fq[minfreq]=newList;

- updateFreq ( ) - This function updates the freq of the node. Firstly we delete the node wherever it is present in fq because now the freq will be updated. fq[node→cnt].deleteNode(node).

In this function we need to take care of minfreq, if the node being dleeted is the only single node that has minfreq then we need to increment minfreq by 1 because the current freq value of this node will increase by 1 and the minfreq will also increase by 1.

Now we create a new list making the same check whether or not this exist in fq and repeating same procedure as in put ( ). We increment node→cnt by 1 before adding this node.

Code :

```
#include <bits/stdc++.h>
class Node
{
    public:
    int key,val,cnt;
    Node* next,*prev;
    Node(int key,int val)
    {
        this->key=key;
        this->val=val;
        cnt=1;
        next=NULL;
        prev=NULL;
    }
};
class DLL
{
    public:
    Node* head=new Node(-1,-1);
    Node* tail=new Node(-1,-1);
    int size;
    DLL()
    {
        head->next=tail;
        tail->prev=head;
        size=0;
    }
    void addNode(Node* newNode)
    {
        Node* temp=head->next;
        newNode->prev=head;
        newNode->next=temp;
        temp->prev=newNode;
        head->next=newNode;
        size++;
    }
    void deleteNode(Node* delNode)
    {
        Node* delprev=delNode->prev;
        Node* delnext=delNode->next;
        delprev->next=delnext;
        delnext->prev=delprev;
        size--;
    }
};
class LFUCache
{
public:
    int maxSize,currSize,minfreq;
    map<int,Node*> mp;
    map<int,DLL*> fq;
    LFUCache(int capacity)
```

```
    {
        // Write your code here.
        maxSize=capacity;
        currSize=0;
        minfreq=0;
    }

    void updateFreq(Node* node)
    {
        fq[node->cnt]->deleteNode(node);
        if(minfreq==node->cnt && fq[node->cnt]->size==0)
        {
            minfreq++;
        }
        DLL *newList=new DLL();
        if(fq.find(node->cnt+1)!=fq.end())
        {
            newList=fq[node->cnt+1];
        }
        node->cnt+=1;
        newList->addNode(node);
        fq[node->cnt]=newList;

    }
    int get(int key)
    {
        // Write your code here.
        if(mp.find(key)!=mp.end())
        {
            Node* found=mp[key];
            int res=found->val;
            updateFreq(found);
            return res;
        }
        return -1;
    }

    void put(int key, int value)
    {
        // Write your code here.
        if(mp.find(key)!=mp.end())
        {
            Node* found=mp[key];
            found->val=value;
            updateFreq(found);
        }
        else
        {
            if(currSize==maxSize)
            {
                mp.erase(fq[minfreq]->tail->prev->key);
                fq[minfreq]->deleteNode(fq[minfreq]->tail->prev);
                currSize--;
            }
```

```
            currSize++;
            minfreq=1;
            DLL *newList=new DLL();
            if(fq.find(minfreq)!=fq.end())
            {
                newList=fq[minfreq];
            }
            Node *newNode = new Node(key,value);
            newList->addNode(newNode);
            mp[key]=newNode;
            fq[minfreq]=newList;
        }
    }
};
```

- Time Complexity : O(1)
- Space Complexity : O(2*capacity)