7

Problema 10

WEN HU, JIANUO (MARP48)

ID envío	Usuario	Hora envío	Veredicto
77164	MARP48	2023-10-05 17:47	AC
77158	MARP48	2023-10-05 17:25	AC
77154	MARP48	2023-10-05 17:17	WA
77045	MARP48	2023-10-05 11:34	WA
77030	MARP48	2023-10-05 11:08	WA
76995	MARP48	2023-10-05 10:33	TLE
76991	MARP48	2023-10-05 10:30	TLE

Fichero `IndexPQ.h`

```
//
//  IndexPQ.h
//
//  Implementación de colas con prioridad mediante montículos.
//  Los elementos son enteros de 0 a N-1 que tienen asociada una prioridad
//  dentro de la cola. La prioridad de cualquier elemento puede ser
//  modificada con coste en O(log N).
//
//  Facultad de Informática
//  Universidad Complutense de Madrid
//
//  Copyright (c) 2020 Alberto Verdejo
//

#ifndef INDEXPQ_H_
#define INDEXPQ_H_

#include <iostream>
#include <stdexcept>
#include <vector>
#include <map>

using namespace std;

// T es el tipo de las prioridades
// Comparator dice cuándo un valor de tipo T es más prioritario que otro
template <typename T2, typename T = int, typename Comparator = std::less<T>>
class IndexPQ {
public:
    // registro para las parejas < elem, prioridad >
    struct Par {
        T2 elem;
        T prioridad;
    };

private:
    // vector que contiene los datos (pares < elem, prio >)
```

podría ser unordered_map (mejor)

```
    std::vector<Par> array;     // primer elemento en la posición 1

    // vector que contiene las posiciones en array de los elementos
    map<T2, int> posiciones;   // un 0 indica que el elemento no está

    /* Objeto función que sabe comparar prioridades.
     antes(a,b) es cierto si a es más prioritario que b */
    Comparator antes;

public:
    IndexPQ(Comparator c = Comparator()) :
        array(1), antes(c) {}

    IndexPQ(IndexPQ<T2, T, Comparator> const&) = default;

    IndexPQ<T2, T, Comparator>& operator=(IndexPQ<T2, T, Comparator> const&) = default;

    ~IndexPQ() = default;

    // e debe ser uno de los posibles elementos
    void push(T2 e, T const& p) {
        auto i = posiciones.find(e);
        if (i != posiciones.end() && i->second != 0)
            throw std::invalid_argument("No se pueden insertar elementos repetidos.");
        else {
            array.push_back({ e, p });
            posiciones[e] = size();
            flotar(size());
        }
    }

    void update(T2 e, T const& p) {
        auto i = posiciones.find(e);
        if (i == posiciones.end() || i != posiciones.end() && i->second == 0 ) // el elemento
     e se inserta por primera vez
            push(e, p);
        else {
            int aux = i->second;
            array[aux].prioridad = p;
            if (aux != 1 && antes(array[aux].prioridad, array[aux / 2].prioridad))
                flotar(aux);
            else // puede hacer falta hundir a e
                hundir(aux);
        }
    }

    int size() const {
        return int(array.size()) - 1;
    }

    bool empty() const {
```

```
        return size() == 0;
    }

    Par const& top() const {
        if (size() == 0)
            throw std::domain_error("No se puede consultar el primero de una cola vacia");
        else return array[1];
    }

    void pop() {
        if (size() == 0) throw std::domain_error("No se puede eliminar el primero de una cola
    vacía.");
        else {
            posiciones[array[1].elem] = 0; // para indicar que no está
            if (size() > 1) {
                array[1] = std::move(array.back());
                posiciones[array[1].elem] = 1;
                array.pop_back();
                hundir(1);
            }
            else
                array.pop_back();
        }
    }

private:

    void flotar(int i) {
        Par parmov = std::move(array[i]);
        int hueco = i;
        while (hueco != 1 && antes(parmov.prioridad, array[hueco / 2].prioridad)) {
            array[hueco] = std::move(array[hueco / 2]); posiciones[array[hueco].elem] = hueco
    ;
            hueco /= 2;
        }
        array[hueco] = std::move(parmov); posiciones[array[hueco].elem] = hueco;
    }

    void hundir(int i) {
        Par parmov = std::move(array[i]);
        int hueco = i;
        int hijo = 2 * hueco; // hijo izquierdo, si existe
        while (hijo <= size()) {
            // cambiar al hijo derecho de i si existe y va antes que el izquierdo
            if (hijo < size() && antes(array[hijo + 1].prioridad, array[hijo].prioridad))
                ++hijo;
            // flotar el hijo si va antes que el elemento hundiéndose
            if (antes(array[hijo].prioridad, parmov.prioridad)) {
                array[hueco] = std::move(array[hijo]); posiciones[array[hueco].elem] = hueco;
                hueco = hijo; hijo = 2 * hueco;
            }
```

```
            else break;
        }
        array[hueco] = std::move(parmov); posiciones[array[hueco].elem] = hueco;
    }

};

#endif /* INDEXPQ_H_ */
```

Fichero Twitter.cpp

```
 *
 * MARP16 Pedro Leon Miranda
 * MARP48 Jianuo Wen Hu
 *

Escribe aquí un comentario general sobre la solución, explicando cómo
se resuelve el problema y cuál es el coste de la solución, en función
del tamaño del problema.

Se utiliza una cola de prioridades variables para **ordenar y almacenar el tema**, el numero
    de citas y el tiempo (que va indicado por el indice del bucle)
en el que se han actualizado por ultima vez.
Siendo el mas prioritario el que mas numero de citas tenga. Se utiliza ademas un
   unordered_map auxiliar que guarda el tema y ayuda a actualizar el numero de citas
para poder acceder al valor de cada tema facilmente.

O(N log N) donde N es el numero de eventos


struct prio {
    int prioridad;
    int reciente;
public:
    bool operator <(const prio& other) const {
        return prioridad < other.prioridad || (prioridad == other.prioridad && reciente <
    other.reciente);
    }

    bool operator >(const prio& other) const {
        return other < *this;
    }
};

bool resuelveCaso() {

    int numEventos;
    cin >> numEventos;

    if (!cin)
        return false;

    IndexPQ<string, prio, greater<prio>> masCitados;
```

```
    unordered_map<string, int> temas;
    // leer el resto del caso y resolverlo
    for (int i = 0; i < numEventos; ++i) {
        string operacion;
        cin >> operacion;
        string tema;
        int citas;
        if (operacion == "C") {
            cin >> tema >> citas;
            temas[tema] += citas;
            masCitados.update(tema, { temas[tema], i });
        }
        else if (operacion == "E") {
            cin >> tema >> citas;
            temas[tema] -= citas;
            masCitados.update(tema, { temas[tema], i });
        }
        else if (operacion == "TC") {
            int j = 0;
            int max = 3;
            vector<pair<string, prio>> aux;
            while (!masCitados.empty() && j < max) {
                auto a = masCitados.top(); masCitados.pop();
                cout << (j + 1) << "␣" << a.elem << "\n";
                aux.push_back({ a.elem, {a.prioridad.prioridad, a.prioridad.reciente} });
                ++j;
            }
            for (auto e : aux)
                masCitados.push(e.first, e.second);
        }
    }

    cout << "---\n";
    return true;
}
```