Universidad Nacional de San Agustín de Arequipa

Escuela Profesional de Ciencia de la Computación
Estructura de Datos Avanzados

Laboratorio 3: Eviction Policy

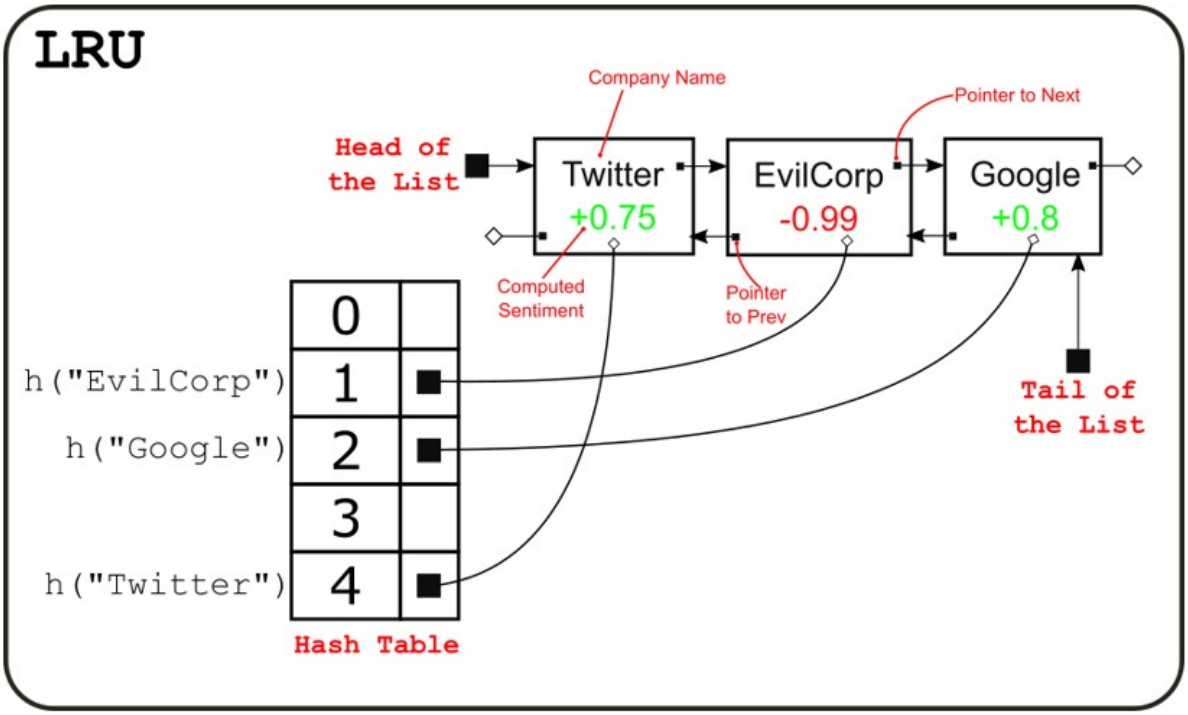
Presentado por:
Rushell Vanessa Zavalaga Orozco

Docente :
Rolando Jesús Cárdenas Talavera

1. Políticas de Memoria Cache

1.1. LRU

Este algoritmo reemplaza el bloque que ha sido usado menos recientemente. Se basa en la suposición de que los datos que no se han utilizado recientemente probablemente no se usarán en el futuro cercano.



1.1.1. Implementación:

```
#include <bits/stdc++.h>

using namespace std;

int main(){
    list<string> my_list;
    map<string, list<string>::iterator> my_hash;
    int max;cin>>max;
    string str;
    int count = 0;

    cout<<"INSERTAR >> ";
    while(cin>>str && str != "0"){
        if(my_hash.find(str) != my_hash.end()){
```

```
            cout<<">Encontrado!\n";
            my_list.remove(str);
            my_list.emplace_front(str);
            my_hash[str] = my_list.begin();
        } else{
            // INSERT
            cout<<">No encontrado!\n";
            if(my_hash.size() < max){
                // HAY ESPACIO
                cout<<">>Insertar sin LRU \n";
                my_list.emplace_front(str);
                my_hash[str] = my_list.begin();
            }
            else{
                // NO HAY ESPACIO
                // ELIMINAR LRU
                cout<<">>Insertar con LRU \n";
                my_hash.erase(my_list.back());
                my_list.pop_back();
                my_list.emplace_front(str);
                my_hash[str] = my_list.begin();
            }
        }

        cout<<"\n --- MY LIST ---\n\n";
        for(auto it = my_list.begin(); it != my_list.end(); it++){
            cout<<*it<<" ";
        }
        cout<<endl;

        cout<<"\n --- MY HASH ---\n\n";
        cout<<" KEY  |  VALUE apuntando a (Dir) -> (Value in List) \n";
        for(auto  it = my_hash.begin(); it != my_hash.end(); it++){
            cout<<it->first<<" apuntando a "<<&(it->second)<<"->"<<*(it->second)<<endl
        }
        cout<<endl;
        cout<<"-----------------------------------\n";

        cout<<"INSERTAR >> ";
    }
}
```

```
 rushh  ~/Documentos/CODE/S6/Estructuras-de-Datos-Ava
3
INSERTAR >> Rushell
>No encontrado!
>>Insertar sin LRU

 --- MY LIST ---

Rushell -

 --- MY HASH ---

 KEY   |  VALUE apuntando a (Dir) -> (Value in List)
Rushell apuntando a 0x6126b1fceb50->Rushell

-----------------------------------
INSERTAR >> Vanessa
>No encontrado!
>>Insertar sin LRU

 --- MY LIST ---

Vanessa - Rushell -

 --- MY HASH ---

 KEY   |  VALUE apuntando a (Dir) -> (Value in List)
Rushell apuntando a 0x6126b1fceb50->Rushell
Vanessa apuntando a 0x6126b1fcebe0->Vanessa

-----------------------------------
INSERTAR >> Zavalaga
>No encontrado!
>>Insertar sin LRU

 --- MY LIST ---

Zavalaga - Vanessa - Rushell -

 --- MY HASH ---

 KEY   |  VALUE apuntando a (Dir) -> (Value in List)
Rushell apuntando a 0x6126b1fceb50->Rushell
Vanessa apuntando a 0x6126b1fcebe0->Vanessa
Zavalaga apuntando a 0x6126b1fcec70->Zavalaga
```

```
----------------------------------
INSERTAR >> Orozco
>No encontrado!
>>Insertar con LRU

 --- MY LIST ---

Orozco - Zavalaga - Vanessa -

 --- MY HASH ---

 KEY  |  VALUE apuntando a (Dir) -> (Value in List)
Orozco apuntando a 0x6126b1fceb50->Orozco
Vanessa apuntando a 0x6126b1fcebe0->Vanessa
Zavalaga apuntando a 0x6126b1fcec70->Zavalaga

----------------------------------
INSERTAR >> Caligaris
>No encontrado!
>>Insertar con LRU

 --- MY LIST ---

Caligaris - Orozco - Zavalaga -

 --- MY HASH ---

 KEY  |  VALUE apuntando a (Dir) -> (Value in List)
Caligaris apuntando a 0x6126b1fcebe0->Caligaris
Orozco apuntando a 0x6126b1fceb50->Orozco
Zavalaga apuntando a 0x6126b1fcec70->Zavalaga
```

1.1.2. Complejidad Computacional

El algoritmo **Least Recently Used (LRU)** tiene una complejidad computacional eficiente gracias al uso de estructuras de datos como una lista doblemente enlazada y un hash map. La combinación de estas estructuras permite operaciones rápidas tanto para acceder como para actualizar los datos, logrando un rendimiento óptimo en escenarios donde el reemplazo de datos es crítico.

- **Acceso a datos:** O(1). La estructura de hash map permite acceder directamente a cualquier elemento en tiempo constante O(1). Cada entrada del hash map apunta a un nodo en la lista doblemente enlazada, lo que permite ubicar y mover los nodos sin tener que recorrer la lista completa.
- **Inserción/Actualización:** O(1). Cuando se accede a un dato, el algoritmo LRU lo actualiza moviéndolo al frente de la lista doblemente enlazada. Como las operaciones

de inserción y eliminación de nodos en una lista doblemente enlazada también tienen una complejidad de O(1), el proceso de actualización es muy eficiente.

- **Reemplazo de datos (cache eviction):** O(1). En caso de que la cache esté llena y sea necesario eliminar el elemento menos recientemente utilizado, se puede acceder al nodo al final de la lista doblemente enlazada en O(1), eliminarlo, y actualizar el hash map en el mismo tiempo.
- **Búsqueda de elementos:** O(1). Gracias al hash map, encontrar cualquier elemento en la cache se realiza en tiempo constante, sin necesidad de recorrer la lista.

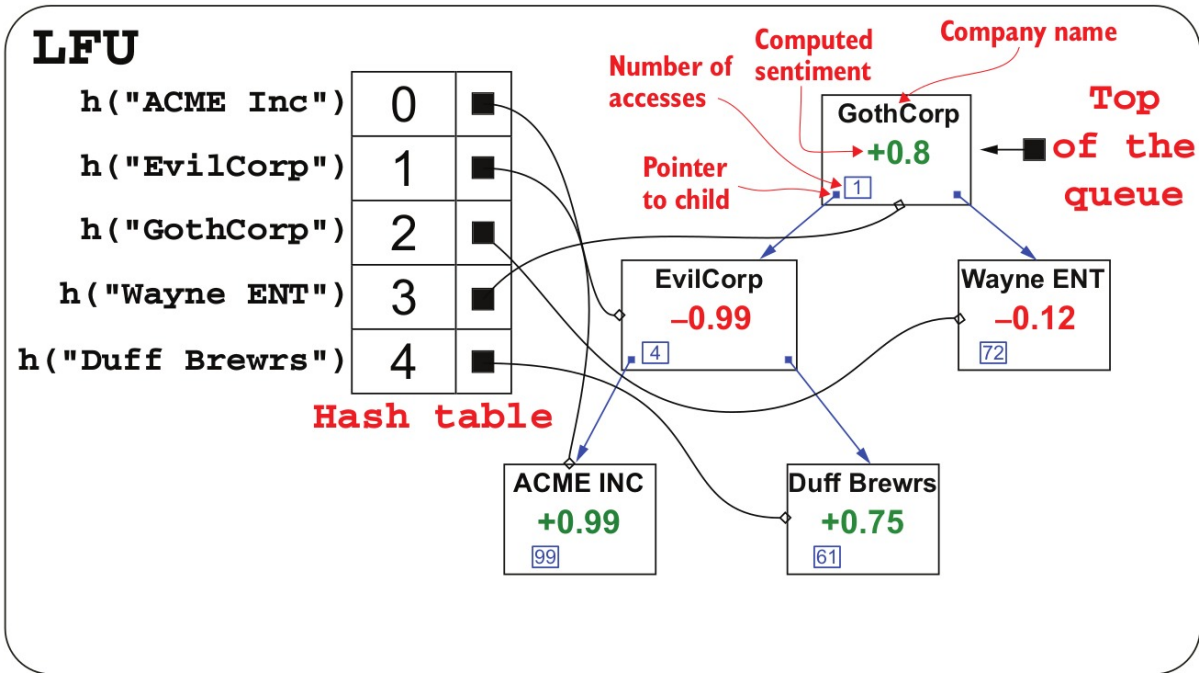
En resumen, todas las operaciones del algoritmo LRU —acceso, inserción, eliminación y actualización— se realizan en tiempo **constante** O(1) gracias al uso combinado de una lista doblemente enlazada y un hash map.

1.1.3. Complejidad del espacio

La complejidad del espacio es O(n), donde n es el número máximo de bloques que se pueden almacenar en la caché, ya que se requiere espacio para los elementos almacenados y para las estructuras de datos asociadas (lista y hash map).

1.2. LFU

Reemplaza el bloque que ha sido usado con menos frecuencia. Se basa en el principio de que los datos que no se han usado con frecuencia probablemente no se usarán en el futuro.



1.2.1. Implementación:

Main:

```
#include <bits/stdc++.h>
#include "FibonaciHeap.cpp"

using namespace std;

int main(){
    Fibonaci_Heap * my_heap = new Fibonaci_Heap();
    map<string, NodoF*> my_hash;
    int max;cin>>max;
    string str;

    cout<<"INSERTAR >> ";
    while(cin>>str && str != "0"){
        if(my_hash.find(str) != my_hash.end()){
            cout<<">Encontrado!\n";
            my_heap->Update(str);
        } else{
            // INSERT
            cout<<">No encontrado!\n";
            if(my_hash.size() < max){
                // HAY ESPACIO
                cout<<">>Insertar sin LFU \n";
                my_hash[str] = my_heap->Insert(make_pair(str,1));
            }
            else{
                // NO HAY ESPACIO
                // ELIMINAR LRU
                cout<<">> Eliminando "<<my_heap->GetMin()->m_Dato.first<<"\n";
                my_hash.erase(my_heap->GetMin()->m_Dato.first);
                my_heap->Extrac_Min();
                my_hash[str] = my_heap->Insert(make_pair(str,1));
            }
        }

        cout<<"\n --- MY HEAP ---\n\n";
        my_heap->showList();
        my_heap->ShowDot();


        cout<<"\n --- MY HASH ---\n\n";
        for(auto  it = my_hash.begin(); it != my_hash.end(); it++){
            cout<<it->first<<" "<<it->second<<" / "<<(*it->second).m_Dato.first<<endl
        }
```

```
        cout<<endl;
        cout<<"-----------------------------------\n";

        cout<<"INSERTAR >> ";
    }
}
```

Fibonacci Heap modificado para que soporte par de string y entero

```
#include <iostream>
#include <list>
#include <math.h>
#include <fstream>
#include <utility> // Para std::make_p
#include <vector>

using namespace std;

struct NodoF {
    int             m_Grado;
    pair<string,int>                m_Dato;
    NodoF*       m_Padre;
    list<NodoF*> m_Sons;
    bool            m_Color;

    NodoF(const pair<string,int>& d){
        m_Dato = d;
        m_Grado = 0;
        m_Padre = nullptr;
        m_Color = false;
    }
};

class Fibonaci_Heap {
    public:
    list<NodoF*>     m_Roots;
    NodoF*           m_pMin;
    int              m_size;

    Fibonaci_Heap(){
        m_pMin = nullptr;
        m_size = 0;
    }
    ~Fibonaci_Heap();

    NodoF* Insert(const pair<string,int>& d);
    void Extrac_Min();
```

```
    void Delete(NodoF* e);
    void Decrease_Key(NodoF* e, const pair<string,int>& val);
    NodoF * GetMin();
    void showList();
    void ShowDot(NodoF* actual, ostream& out);
    void ShowDot(const string& filename);
    void Compactar();
    NodoF* Unir(NodoF* p, NodoF* q);
    void Update(const string& str);
};

/// IMplementacion para pares
void Fibonaci_Heap::Update(const string& str) {
    int veces = 0;
    bool doit = false;
    if(m_pMin->m_Dato.first == str) {
        cout << m_pMin->m_Dato.first << " " << m_pMin->m_Dato.second << endl;
        veces = m_pMin->m_Dato.second;
        Delete(m_pMin);
        doit = true;
    }else{
        for (auto it = m_Roots.begin(); !doit && it != m_Roots.end(); ++it) {
            if((*it)->m_Dato.first == str) {
                cout << (*it)->m_Dato.first << " " << (*it)->m_Dato.second << endl;
                veces = (*it)->m_Dato.second;
                Delete(*it);
                doit = true;
                break;
            }
            auto it2 = (*it)->m_Sons.begin();
            while (it2 != (*it)->m_Sons.end() && !doit) {
                if ((*it2)->m_Dato.first == str) {
                    cout << (*it2)->m_Dato.first << " " << (*it2)->m_Dato.second << en
                    veces = (*it2)->m_Dato.second;

                    // Almacenar un iterador temporal antes de eliminar
                    auto temp = it2;
                    ++it2;  // Avanzar el iterador antes de eliminar

                    // Eliminar el nodo de los hijos
                    (*it)->m_Sons.remove(*temp);
                    Delete(*temp);
                    doit = true;
                    break;
                } else {
                    ++it2;  // Solo avanzar si no se ha eliminado
                }
```

```
            }
        }
    }

    cout << "Se actualizo el valor de " << str << " a " << veces + 1 << endl;
    if(doit) Insert(make_pair(str, veces + 1));
}

void Fibonaci_Heap::showList() {
    for (auto it = m_Roots.begin(); it != m_Roots.end(); ++it) {
        cout << (*it)->m_Dato.first << " / " << (*it)->m_Dato.second << " [" << (*it)
        for(auto it2 = (*it)->m_Sons.begin(); it2 != (*it)->m_Sons.end(); ++it2) {
            cout << "-> " << (*it2)->m_Dato.first << " / " << (*it2)->m_Dato.second <<
        }
        cout << endl;
    }
}

NodoF* Fibonaci_Heap::Insert(const pair<string, int>& d) {
    NodoF* pNew = new NodoF(d);
    if (!m_pMin || d.second < m_pMin->m_Dato.second) m_pMin = pNew;
    m_Roots.push_back(pNew);
    m_size++;
    return pNew;
}

void Fibonaci_Heap::Extrac_Min() {
    if (!m_pMin) return;
    for (auto it = m_pMin->m_Sons.begin(); it != m_pMin->m_Sons.end(); ++it) {
        (*it)->m_Color = false;
        (*it)->m_Padre = nullptr;
        m_Roots.push_back(*it);
    }

    m_Roots.remove(m_pMin);
    m_size--;
    Compactar();

    m_pMin = nullptr; // Resetear m_pMin
    for (auto it = m_Roots.begin(); it != m_Roots.end(); ++it) {
        if (!m_pMin || (*it)->m_Dato.second < m_pMin->m_Dato.second) {
            m_pMin = *it;
        }
    }
}

void Fibonaci_Heap::Delete(NodoF* e) {
```

```
    Decrease_Key(e, make_pair(e->m_Dato.first, m_pMin->m_Dato.second - 1));
    m_pMin = e; //!
    Extrac_Min();
}

void Fibonaci_Heap::Decrease_Key(NodoF* e, const pair<string, int>& val) {
    e->m_Dato = val;
    if (e->m_Dato.second < m_pMin->m_Dato.second && !e->m_Padre) m_pMin = e;
    if (e->m_Padre && e->m_Padre->m_Dato.second > val.second) {
        do {
            e->m_Color = false;
            e->m_Padre->m_Sons.remove(e);
            e->m_Padre->m_Grado--;
            m_Roots.push_front(e);
            e = e->m_Padre;
        } while (e->m_Padre && e->m_Padre->m_Color);

        if (e->m_Padre) e->m_Color = true;
        else e->m_Color = false;
    }
}

NodoF* Fibonaci_Heap::GetMin() {
    return m_pMin;
}

void Fibonaci_Heap::Compactar() {
    const int size = ceil(log2(m_size));
    vector<NodoF*> vec(size, nullptr); // Usar vector

    auto it = m_Roots.begin();
    while (it != m_Roots.end()) {
        int grado = (*it)->m_Grado;
        if (!vec[grado]) {
            vec[grado] = *it;
            it++;
        } else {
            NodoF* r = Unir(*it, vec[grado]);
            m_Roots.remove(vec[grado]);
            it = m_Roots.erase(it);
            m_Roots.push_back(r);
            vec[grado] = nullptr;
        }
    }
}

NodoF* Fibonaci_Heap::Unir(NodoF* p, NodoF* q) {
```

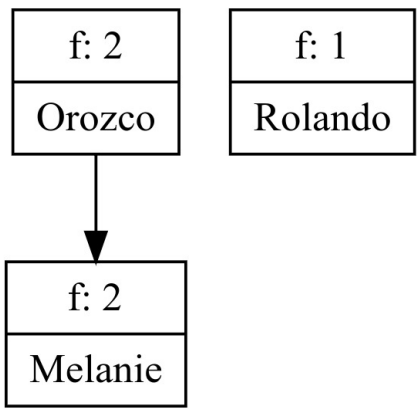
```
    if (p->m_Dato.second > q->m_Dato.second)
        swap(p, q);
    p->m_Sons.push_back(q);
    q->m_Padre = p;
    p->m_Grado++;
    return p;
}

void Fibonaci_Heap::ShowDot(NodoF* actual, ostream& out) {
    if (!actual) return;
    out << actual->m_Dato.first << "[label = \"{ {f: " << actual->m_Dato.second << "}
    for (auto it = actual->m_Sons.begin(); it != actual->m_Sons.end(); ++it) {
        out << actual->m_Dato.first << " -> " << (*it)->m_Dato.first << ";" << endl;
        ShowDot((*it), out);
    }
}

void Fibonaci_Heap::ShowDot(const string& filename) {
    ofstream out(filename);
    out << "digraph G {" << endl;
    out << "label= \"Fibonaci Heap\";" << endl;
    out << "node [shape = record];" << endl;
    for (auto it = m_Roots.begin(); it != m_Roots.end(); ++it) ShowDot((*it), out);
    out << "}" << endl;
    out.close();
}
```

1.2.2. Resultados

```
3
INSERTAR >> Rushell
>No encontrado!
>>Insertar sin LFU

 --- MY HEAP ---

Rushell / 1 [0]


 --- MY HASH ---

Rushell 0x6247af9bab00 / Rushell

---------------------------------
INSERTAR >> Vanessa
>No encontrado!
>>Insertar sin LFU

 --- MY HEAP ---

Rushell / 1 [0]

Vanessa / 1 [0]


 --- MY HASH ---

Rushell 0x6247af9bab00 / Rushell
Vanessa 0x6247af9badb0 / Vanessa

---------------------------------
INSERTAR >> Zavalaga
>No encontrado!
>>Insertar sin LFU

 --- MY HEAP ---

Rushell / 1 [0]

Vanessa / 1 [0]

Zavalaga / 1 [0]


 --- MY HASH ---

Rushell 0x6247af9bab00 / Rushell
Vanessa 0x6247af9badb0 / Vanessa
Zavalaga 0x6247af9bae80 / Zavalaga
```

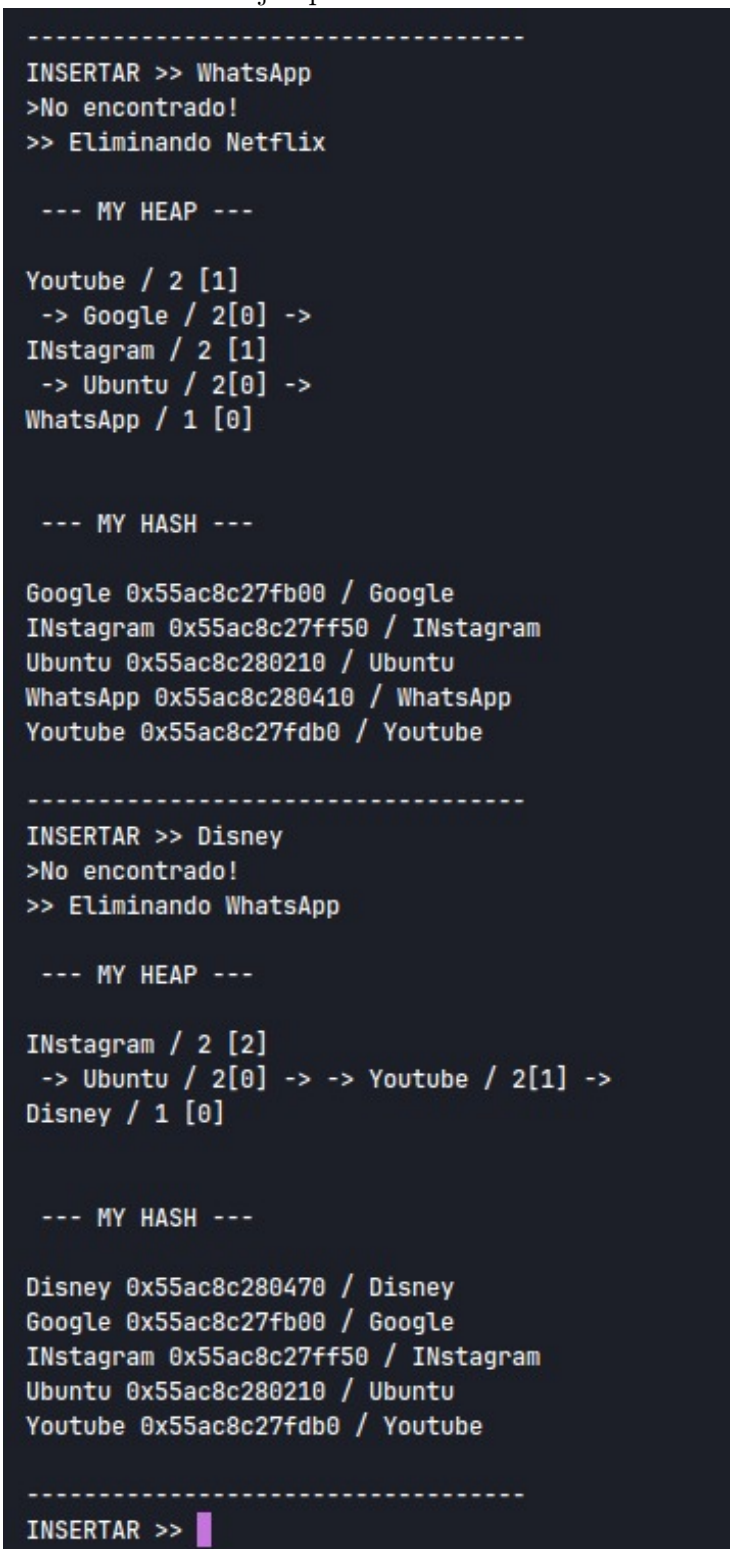
```
-----------------------------------
INSERTAR >> Anlu
>No encontrado!
>> Eliminando Zavalaga

 --- MY HEAP ---

Vanessa / 1 [1]
 -> Orozco / 1[0] ->
Anlu / 1 [0]


 --- MY HASH ---

Anlu 0x6247af9baff0 / Anlu
Orozco 0x6247af9baf70 / Orozco
Vanessa 0x6247af9badb0 / Vanessa

-----------------------------------
INSERTAR >> Orozco
>Encontrado!
Orozco 1
Se actualizo el valor de Orozco a 2

 --- MY HEAP ---

Anlu / 1 [1]
 -> Vanessa / 1[0] ->
Orozco / 2 [0]


 --- MY HASH ---

Anlu 0x6247af9baff0 / Anlu
Orozco 0x6247af9baf70 / Orozco
Vanessa 0x6247af9badb0 / Vanessa

-----------------------------------
INSERTAR >> Melanie
>No encontrado!
>> Eliminando Anlu

 --- MY HEAP ---

Vanessa / 1 [1]
 -> Orozco / 2[0] ->
Melanie / 1 [0]


 --- MY HASH ---

Melanie 0x6247af9bb0d0 / Melanie
Orozco 0x6247af9baf70 / Orozco
Vanessa 0x6247af9badb0 / Vanessa

-----------------------------------
INSERTAR >>
```

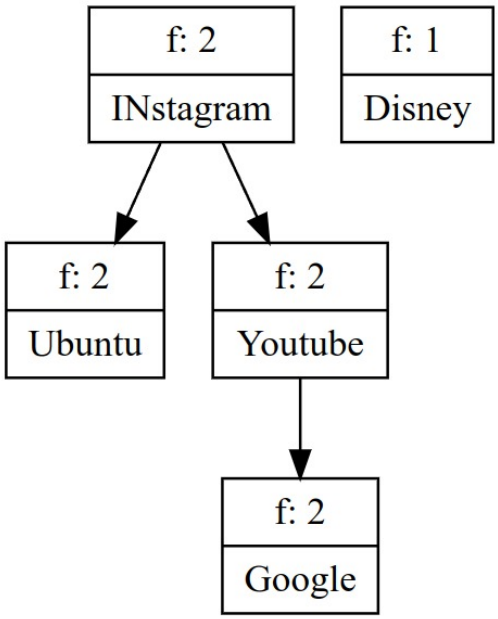
```
-----------------------------------
INSERTAR >> Melanie
>Encontrado!
Melanie 1
Se actualizo el valor de Melanie a 2

 --- MY HEAP ---

Vanessa / 1 [1]
 -> Orozco / 2[0] ->
Melanie / 2 [0]


 --- MY HASH ---

Melanie 0x6247af9bb0d0 / Melanie
Orozco 0x6247af9baf70 / Orozco
Vanessa 0x6247af9badb0 / Vanessa

-----------------------------------
INSERTAR >> Rolando
>No encontrado!
>> Eliminando Vanessa

 --- MY HEAP ---

Orozco / 2 [1]
 -> Melanie / 2[0] ->
Rolando / 1 [0]


 --- MY HASH ---

Melanie 0x6247af9bb0d0 / Melanie
Orozco 0x6247af9baf70 / Orozco
Rolando 0x6247af9bb1b0 / Rolando

-----------------------------------
INSERTAR >>
```



Fibonacci Heap

Otro ejemplo con mas nodos





Fibonaci Heap

1.2.3. Complejidad Computacional

El algoritmo **Least Frequently Used (LFU)**, implementado con un hash map y un Fibonacci Heap, ofrece una gran eficiencia en la gestión de la frecuencia de uso de los elementos. A continuación, se detalla la complejidad de las operaciones principales:

- **Acceso a datos:** O(1). El hash map permite acceder a los datos en tiempo constante O(1), manteniendo la relación entre los elementos y sus frecuencias.
- **Inserción/Actualización de frecuencia:** Amortizado O(1). Al utilizar un Fibonacci Heap, se obtiene una eficiencia amortizada de O(1) para la actualización de la frecuencia de los elementos, ya que esta estructura permite disminuir claves de manera eficiente y mantener la jerarquía de frecuencias.
- **Reemplazo (eliminación del elemento menos frecuentemente usado):** O(log n). Para eliminar el elemento menos frecuentemente usado, se extrae el nodo con la menor frecuencia del Fibonacci Heap, lo que tiene una complejidad de O(log n) en el peor caso, debido al proceso de consolidación y reestructuración del heap.
- **Búsqueda del mínimo (elemento con menor frecuencia):** O(1). El Fibonacci Heap permite encontrar el elemento con la menor frecuencia de uso en tiempo constante O(1), ya que siempre mantiene un puntero al nodo con el valor mínimo.

En resumen, el uso de un hash map junto con un Fibonacci Heap permite realizar las operaciones más comunes en el algoritmo LFU de manera extremadamente eficiente. El

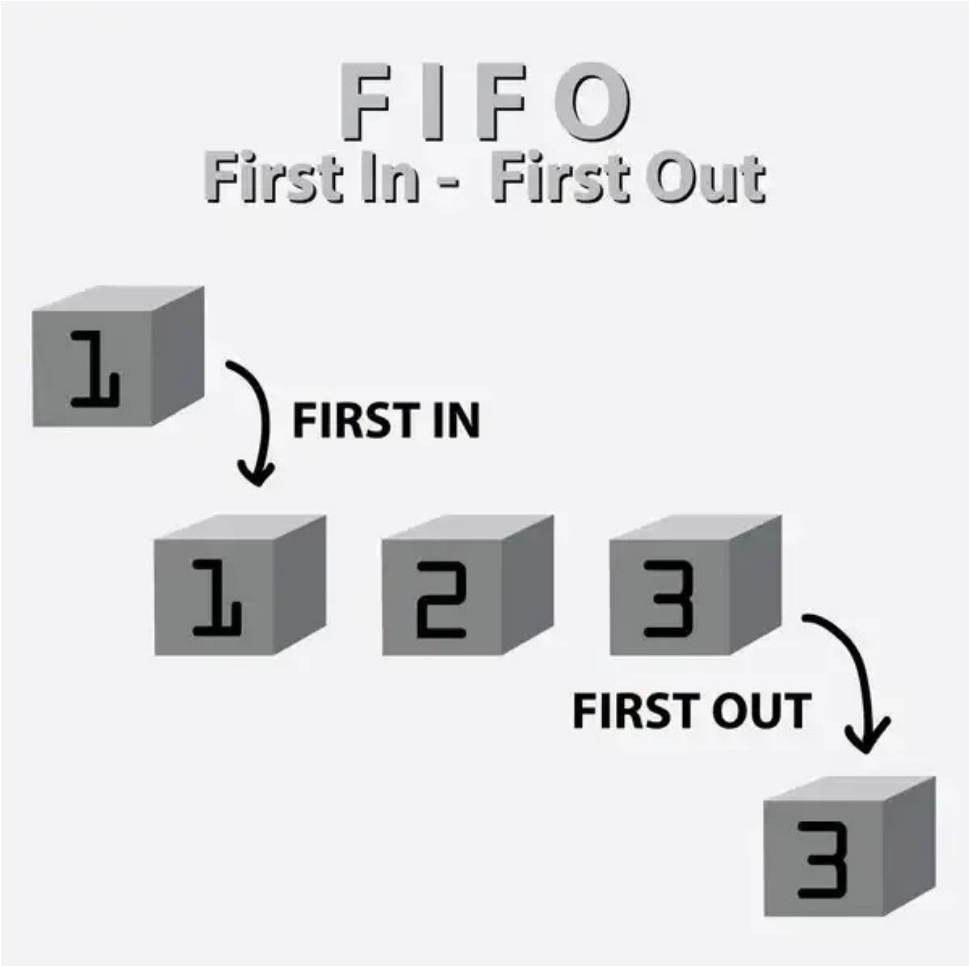
acceso a datos y la actualización de la frecuencia se realizan en tiempo constante O(1), mientras que la eliminación del elemento menos frecuentemente usado tiene una complejidad O(log n), lo que mejora significativamente el rendimiento frente a otras estructuras de datos.

1.2.4. Complejidad del espacio

La complejidad del espacio es O(n), donde n es el número máximo de bloques que se pueden almacenar en la caché, además de la sobrecarga adicional para almacenar la frecuencia de uso y los punteros asociados.

1.3. FIFO

Este algoritmo reemplaza el bloque que ha estado en la caché por más tiempo, sin considerar el uso reciente o la frecuencia de acceso. Es simple y fácil de implementar.



1.3.1. Implementación:

```
#include<bits/stdc++.h>

using namespace std;

int main(){
    int max;cin>>max;
    list<string> my_list;
    string str;

    cout<<"INSERTAR >> ";
    while(cin>>str && str != "0"){
        bool found = false;
        for(auto it=my_list.begin();it!=my_list.end();it++){
            if(*it == str){
                cout<<">Encontrado!\n";
                my_list.erase(it);
                my_list.push_back(str);
                found = true;
            }
        }
        if(!found){
            cout<<">No encontrado!\n";
            if(my_list.size() == max){
                cout<<">>Eliminando: "<<my_list.front()<<endl;
                my_list.pop_front();
            }
            my_list.push_back(str);
        }

        cout<<"\n --- FIFO --- \n\n";
        for(auto it=my_list.begin();it!=my_list.end();it++){
            cout<<*it<<" ";
        }
        cout<<endl;
        cout<<"----------------------------------\n";

        cout<<"INSERTAR >> ";
    }
}
```


rushh ~/Documentos/CODE/S6/Estructur
3
INSERTAR >> Google
>No encontrado!
--- PRINTING FIFO ---
Google --
INSERTAR >> Youtube
>No encontrado!
--- PRINTING FIFO ---
Google -- Youtube --
INSERTAR >> Netflix
>No encontrado!
--- PRINTING FIFO ---
Google -- Youtube -- Netflix --
INSERTAR >> Disney
>No encontrado!
>>Eliminando: Google
--- PRINTING FIFO ---
Youtube -- Netflix -- Disney --
INSERTAR >> Instagram
>No encontrado!
>>Eliminando: Youtube
--- PRINTING FIFO ---
Netflix -- Disney -- Instagram --
INSERTAR >>

1.3.2. Complejidad Computacional

El algoritmo **First-In, First-Out (FIFO)**, utilizado para gestionar la política de reemplazo de páginas o elementos en estructuras de datos basadas en colas, tiene la siguiente complejidad computacional:

- **Acceso a datos:** O(1). Si se usa una cola o lista doblemente enlazada para implementar FIFO, el acceso al primer o último elemento se realiza en tiempo constante, O(1).
- **Reemplazo:** O(1). Reemplazar el primer elemento (el más antiguo) de la cola en el algoritmo FIFO se puede realizar en O(1), ya que solo implica eliminar el nodo en la cabeza de la cola y agregar uno nuevo al final.

En resumen, el uso de una estructura de cola eficiente para el algoritmo FIFO permite tanto el acceso como el reemplazo de elementos en tiempo constante O(1), lo que lo convierte en una opción de bajo costo computacional.

1.3.3. Complejidad del espacio

La complejidad del espacio es O(n), donde n es el número máximo de bloques que se pueden almacenar en la caché, junto con la estructura de datos utilizada para mantener la cola.