Memoria caché

Alumno: Josue Samuel Philco Puma

25 de octubre de 2024

Resumen

El siguiente trabajo contiene la implementación de las diferentes estrategias de reemplazo en la memoria caché, estas estrategias son las siguientes: LRU (Least Recently Used), LFU (Least Frequently Used) y FIFO (First In - First Out). Además vamos a analizar la complejidad de cada estrategia.

1. Implementación de la Memoria Caché

Para realizar estas implementaciones debemos tener en cuenta que cada estrategia trabaja de una mejor manera con estructuras, ya sea una lista enlazada, tablas hash u otras. En este caso vamos a trabajar con 4 estructuras en total.

1.1. LRU (Least Recently Used)

La política de LRU nos dice que va a eliminar los registros que son menos utilizados recientemente. Las operaciones que se realizan con mayor frecuencia están presentes en la memoria el mayor tiempo posible. Bueno, para esta estrategia vamos a optar por las siguientes estructuras: Una Tabla Hash y una Lista Doblemente Enlazada.

- **Estructura de la Lista Enalzada**

```
class NodeLRU {
    public:
        int key, value;
        NodeLRU* prev;
        NodeLRU* next;
        NodeLRU(int k, int v): key(k), value(v), prev(nullptr), next(
            nullptr) {}
};
```

En la clase **NodeLRU** tenemos los punteros `prev` (representa hacia el nodo anterior) y `next` (representa hacia el nodo siguiente).

- **Estructura del LRU**

```
class LRUCache {
    private:
        int capacity;
        unordered_map<int, NodeLRU*> cache;
        NodeLRU* head;
        NodeLRU* tail;

        void removeNode(NodeLRU* node) {
            if (node->prev) node->prev->next = node->next;
            if (node->next) node->next->prev = node->prev;
            if (node == head) head = node->next;
            if (node == tail) tail = node->prev;
        }

        void moveToFront(NodeLRU* node) {
            node->next = head;
            node->prev = nullptr;
            if (head) {
                head->prev = node;
            }
            head = node;
            if (!tail) {
```

```
                tail = head;
            }
        }

        void removeTail() {
            if (tail) {
                cache.erase(tail->key);
                NodeLRU* prevTail = tail;
                tail = tail->prev;
                if (tail) {
                    tail->next = nullptr;
                }
                delete prevTail;
            }
        }
    public:
        LRUCache(int capacity) : capacity(capacity), head(nullptr), tail(
            nullptr) {}

        int get(int key);
        void put(int key, int value);
        void display();

        ~LRUCache() {
            while (head) {
                NodeLRU* temp = head;
                head = head->next;
                delete temp;
            }
        }
};
```

En la clase **LRUCache** va a estar gestionando el comportamiento de la cache, moviendo nodos al frente cuando son accedidos (método `moveToFront`) y eliminando el nodo menos reciente (método `removeTail`). Los punteros `head` y `tail` permiten un acceso rápido al principio y al final de la lista, crucial para las operaciones de una cache LRU.

- **Métodos del LRU**

```
int LRUCache::get(int key) {
    if (cache.find(key) != cache.end()) {
        NodeLRU* node = cache[key];
        if (node != head) {
            removeNode(node);
            moveToFront(node);
        }
        return node->value;
    }
    return -1;
}

void LRUCache::put(int key, int value) {
    if (cache.find(key) != cache.end()) {
        NodeLRU* node = cache[key];
        node->value = value;
        removeNode(node);
        moveToFront(node);
    }
    else {
        if (cache.size() == capacity) {
            removeTail();
        }
        NodeLRU* newNode = new NodeLRU(key, value);
        moveToFront(newNode);
```

```
            cache[key] = newNode;
        }
}

void LRUCache::display() {
        NodeLRU* current = head;
        cout << "Estado actual del cache: ";
        while (current) {
            cout << "(" << current->key << ", " << current->value << ") ";
            current = current->next;
        }
        cout << endl;
}
```

- En el método `get(key)` va a buscar la clave `key` en la caché. Si la clave existe, actualiza su posición en la lista doblemente enlazada para marcarla como la más recientemente utilizada. Si no existe, devuelve -1.
- En el método `put(kry, value)` inserta un nuevo par clave-valor en el cache. Si la clave ya existe, actualiza el valor y mueve el nodo al frente. Si la clave no existe y el cache ha alcanzado su capacidad máxima, elimina el nodo menos recientemente utilizado antes de insertar el nuevo.

1.2. LFU (Least Frequently Used)

La política de LFU nos dice que los elementos menos utilizados van a ser retirados lo más pronto posible. Podemos usar la misma estructura que LRU pero con modificaciones. En este caso tenemos que optar por una estructura que obtenga siempre el nodo con menor frecuencia, en este caso podemos optar por etructuras de heap como: **Binary Heap**, **Binomial Heap** o **Fibonacci Heap**. Para esta implementación se ha usado las estructuras: Tabla Hash y Binomial Heap.

- **Estructura del Binomial Heap**

```
class Node {
    public:
        int key;
        int value;
        int freq;
        Node* parent;
        std::vector<Node*> children;
        int degree;

        Node(int k, int v, int f) : key(k), value(v), freq(f), parent(
            nullptr), degree(0) {}
};
```

La clase **Node** va a representar el nodo para el heap. Este nodo va a almacenar un valor (`value`), una clave (`key`) y una frecuencia (`freq`) y va a estar relacionado con otros nodos a través de su relación padre-hijo.

```
class BinomialHeap {
    public:
        std::vector<Node*> trees;
        Node* min_node;
        int count;

        BinomialHeap() : min_node(nullptr), count(0) {}

        void insert(Node* node) {
            // Insertar el nodo en un nuevo heap y fusionar
            BinomialHeap new_heap;
            new_heap.trees.push_back(node);
            new_heap.count = 1;
            merge(new_heap);
```

```
            find_min();
        }

        Node* extract_min() {
            // Extraer el nodo minimo
            Node* min_node = this->min_node;
            trees.erase(std::remove(trees.begin(), trees.end(), min_node),
                trees.end());

            BinomialHeap new_heap;
            new_heap.trees = min_node->children;
            min_node->children.clear();
            merge(new_heap);
            find_min();
            count--;
            return min_node;
        }

        void decrease_key(Node* node, int new_freq) {
            node->freq = new_freq;
            bubble_up(node);
            find_min();
        }

        void merge(BinomialHeap& other_heap) {
            trees.insert(trees.end(), other_heap.trees.begin(), other_heap.
                trees.end());
            count += other_heap.count;
            find_min();
        }

        void find_min() {
            min_node = nullptr;
            for (Node* tree : trees) {
                if (!min_node || tree->freq < min_node->freq) {
                    min_node = tree;
                }
            }
        }

        void bubble_up(Node* node) {
            Node* parent = node->parent;
            while (parent && node->freq < parent->freq) {
                std::swap(node->key, parent->key);
                std::swap(node->value, parent->value);
                std::swap(node->freq, parent->freq);
                node = parent;
                parent = node->parent;
            }
        }
};
```

La clase **BinomialHeap** representa lo que es la estructura como colección de árboles binomiales, sus métodos principales son:

- El método `insert` inserta un nuevo nodo en el heap creando un nuevo heap binomial con solo ese nodo y luego lo fusiona este nuevo heap con el existente y luego encontrando el nodo mínimo.
- El método `extractmin` va a extraer el nodo con la menor frecuencia, elimina su árbol de las raíces y crea un nuevo heap a partir de los hijos del nodo extraído. Luego, fusiona el nuevo heap con el heap original y vuelve a buscar el nodo mínimo.
- El método `decreasekey` actualiza la frecuencia de los nodos, realiza el proceso `bubbleup` para

mover el nodo hacia arriba si su nueva frecuencia es menor que la de su padre.

- El método `merge` fusiona el heap actual con otro heap añadiendo todos los árboles del `otherheap` y ajustando el contador de nodos. Después de realizar la fusión buscará el nodo mínimo.
- El método `findmin` busca el nodo con menor frecuencia entre todas las raíces de los árboles binomiales.
- El método `bubbleup` mueve el nodo hacia arriba en el heap si su nueva frecuencia es menor que la de su padre.

Estructura del LFU

```
class LFUCache {
    public:
        int capacity;
        std::unordered_map<int, int> cache;
        BinomialHeap freq_heap;
        std::unordered_map<int, Node*> key_node_map;

        LFUCache(int cap) : capacity(cap) {}

        int get(int key);
        void put(int key, int value);
        void display();
    private:
        void evict() {
            Node* node_to_evict = freq_heap.extract_min();
            cache.erase(node_to_evict->key);
            key_node_map.erase(node_to_evict->key);
            delete node_to_evict;
        }

        void update_freq(Node* node) {
            node->freq++;
            freq_heap.decrease_key(node, node->freq);
        }
};
```

La clase `LFUCache` va a utilizar el `Binomial Heap` para gestionar las frecuencias de los elementos almacenados. Los elementos menos usados (menor frecuencia) son eliminados primero al alcanzar la capacidad máxima. Tiene como métodos privados:

- El método `evict` llama al método `extractmin` para encontrar y extraer el nodo con menor frecuencia.
- El método `updatefreq` aumenta el contador del nodo y ajusta la posición del nodo en el heap, ya que la frecuencia a aumentado.

Métodos del LFU

```
int LFUCache::get(int key) {
    if (cache.find(key) == cache.end()) {
        std::cout << "Clave " << key << " no encontrada." << std::endl;
        return -1;
    }
    Node* node = key_node_map[key];
    std::cout << "Obteniendo clave " << key << ". Valor = " << node->value
        << ", Frecuencia = " << node->freq << std::endl;
    update_freq(node);
    return node->value;
}

void LFUCache::put(int key, int value) {
    if (capacity == 0) {
```

```
        return;
    }
    if (cache.find(key) != cache.end()) {
        Node* node = key_node_map[key];
        node->value = value;
        update_freq(node);
    }
    else {
        if (cache.size() >= capacity) {
            std::cout << "Capacidad maxima alcanzada." << std::endl;
            evict();
        }
        Node* new_node = new Node(key, value, 1);
        freq_heap.insert(new_node);
        cache[key] = value;
        key_node_map[key] = new_node;
    }
}

void LFUCache::display() {
    std::cout << "\nEstado actual de la cache (LFU Cache):\n";
    for (const auto& pair : key_node_map) {
        Node* node = pair.second;
        std::cout << "Key: " << node->key << ", Value: " << node->value <<
            ", Freq: " << node->freq << std::endl;
    }
    std::cout << std::endl;
}
```

- En el método `get(key)` recupera el valor asociado a una clave del caché. Si se encuentra la clave se actualiza la frecuencia del nodo, de lo contrario retorna -1.
- En el método `put(key, value)` inserta un nuevo nodo. Si la caché esta llena debe eliminar el elemento menos frecuente y actualizar si es que existe.

1.3. FIFO (First In - First Out)

La política de FIFO nos dice que el primer bloque de datos que entra en la cache es el primero en ser reemplazado cuando la cache está llena y se necesita espacio para un nuevo bloque. Para implementar esta estrategia usamos una estructura conocida que es la **cola** junto con la Tabla Hash.

- **Estructura de la Cola**

```
class Cola {
    struct Nodo {
        int dato;
        Nodo* siguiente;
        Nodo(int dato) : dato(dato), siguiente(nullptr) {}
    };

    Nodo* primero;
    Nodo* ultimo;

    public:
        Cola() : primero(nullptr), ultimo(nullptr) {}

        void push(int dato) {
            Nodo* nuevo = new Nodo(dato);
            if (primero == nullptr) {
                primero = nuevo;
                ultimo = nuevo;
            } else {
                ultimo->siguiente = nuevo;
```

```
                ultimo = nuevo;
            }
        }

        int pop() {
            if (primero == nullptr) {
                return -1;
            }

            int dato = primero->dato;
            Nodo* temp = primero;
            primero = primero->siguiente;
            delete temp;
            return dato;
        }

        int front() {
            if (primero == nullptr) {
                return -1;
            }
            return primero->dato;
        }

        bool empty() {
            return primero == nullptr;
        }
};
```

La clase **Cola** se encarga de crear la estructura de una cola básica usando la estructura de nodos enlazados.

- **Estructura del FIFO**

```
class FIFOCache {
    private:
        std::unordered_map<int, int> cache;
        Cola cola;
        int capacidad;
    public:
        FIFOCache(int capacidad) : capacidad(capacidad) {}

        int get(int key);
        void put(int key, int value);
        void display();
};
```

Este se va a encargar de realizar la política correcta usando la Tabla Hash para representar la memoria caché.

- **Métodos de FIFO**

```
int FIFOCache::get(int key) {
    if (cache.find(key) == cache.end()) {
        return -1;
    }
    return cache[key];
}

void FIFOCache::put(int key, int value) {
    if (cache.find(key) != cache.end()) {
        return;
    }

    if (cache.size() == capacidad) {
```

```
14          int key = cola.pop();
15          cache.erase(key);
16      }
17
18      cache[key] = value;
19      cola.push(key);
20  }
21
22  void FIFOCache::display() {
23      std::cout << "Estado actual del cache:\n";
24      if (cache.empty()) {
25          std::cout << "Cache vacio\n";
26          return;
27      }
28      for (auto it = cache.begin(); it != cache.end(); it++) {
29          std::cout << it->first << ": " << it->second << "\n";
30      }
31  }
```

- El método `get(key)` recupera el valor asociado a la clave. Si este se encuentra en la caché se devuelve ese valor, de caso contrario va a imprimir -1.
- El método `put(key, value)` inserta una nueva clave-valor en el cache siguiendo la política de reemplazo FIFO. Si la clave se encuentra en la caché se actualiza y si no existe procede a elimar el más antiguo de la caché.

2. Complejidad computacional

2.1. Complejidad de LRU

- En la función `get` vamos a buscar en el cache lo que tomaría tiempo $O(1)$, si este existe se llama a la función `removeNode` y `moveToFront` que también es $O(1)$. Entonces, la complejidad de este método es $O(1)$.
- En la función `put` lo que hace es buscar el nodo en la cache, esto toma un tiempo de $O(1)$. Si la clave existe actualiza el valor y mueve el nodo al frente con `removeNode` y `moveToFront` tomando ambos un tiempo de $O(1)$. Si la clave no existe verifica si la capacidad máxima se alcanzó, si la caché esta llena llama a la función `removeTail` que tiene un costo de $O(1)$ eliminando el nodo más antiguo. Crear el nodo nuevo y moverlo al frente toma $O(1)$ e insetarlo tamboén es $O(1)$. Entonces, la complejidad del método es $O(1)$.
- En la función `removeNode` elimina el nodo de la lista y actualiza los punteros sin nu¿inguna operación dependiente del tamaño de la lista. Por lo que su complejidad es de $O(1)$.
- En la función `moveToFront` mueve el nodo al frente de la lista, al igual que el anterior, solo se van actualizando los punteros. Por lo que la complejidad es de $O(1)$.
- En la función `removeTail` elimina el último nodo de la lista, nuevamente solo se actualizan los punteros y hacer eliminaciones. Por lo que la complejidad es de $O(1)$.

En resumen, la complejidad global de toda la implementación es de $O(1)$ lo que lo convierte en una implementación eficiente para el LRU caché.

2.2. Complejidad de LFU

- En la función `insert` crea un nuevo nodo en el heap y fusiona con uno existente. Esto puede tener una complidad de $O(\log n)$ en el peor de los casos. Posteriormente actualiza el nodo mínimo lo que genera el costo de $O(\log n)$. Por lo que la complejidad de esta función es de $O(\log n)$.
- En la función `extract_min` elimina el árbol que contiene el nodo mínimo teniendo una complejidad de $O(n)$, pero como la búsqueda del nodo mínimo se hace antes sería una complejidad de $O(\log n)$. Luego fusiona los hijos del nodo extraído que toma $O(\log n)$. Por lo que la complejidad es $O(\log n)$.

- En la función `decrease_key` actualiza la frecuencia del nodo. La altura de los árboles en este heap es $O(\log n)$. Por lo que su complejidad es $O(\log n)$.
- En la función `merge` fusiona los heaps insertando los árboles. Por lo que su complidad es de $O(\log n)$.
- En la función `find_min` recorre todos los árboles del heap para encontrar el nodo con frecuencia mínima. Por lo que la complejidad es de $O(\log n)$.
- En la función `bubble_up` disminuye la frecuencia de un nodo. El nodo sube en el árbol intercambiando lugares con su padre hasta que su frecuencia sea mayor o igual a la de su padre. Por lo que la complidad es de $O(\log n)$.
- En la función `get` busca el valor asociado a la clave en el caché lo que toma $O(1)$. Si el valor existe, se actualiza su frecuencia, lo cual implica disminuir la clave en el montículo binomial, lo que tiene una complejidad de $O(\log n)$.
- En la función `put`: verifica si la clave ya existe en el caché. Esta operación de búsqueda toma $O(1)$. Si la clave existe, actualiza el valor y la frecuencia, lo que tiene una complejidad de $O(\log n)$.
- En la función `evict` elimina el nodo con la frecuencia mínima, lo cual tiene una complejidad de $O(\log n)$.
- En la función `update_freq` incrementa la frecuencia del nodo y luego llama a decrease_key para ajustar la posición del nodo en el montículo binomial

En resumen, la implementación del LFU caché tiene un costo total de $O(\log n)$.

2.3. Complejidad de FIFO

- La función `push` inserta un elemento al final de la cola teniendo una complejidad de $O(1)$.
- La función `pop` elimina el primer elemento de la cola, el primer nodo es eliminado y el puntero de inicio se actualiza. Por lo que la complejidad es de $O(1)$.
- La función `front` retorna el primer elemento sin eliminar teniendo complejidad $O(1)$.
- La función `empty` verifica si la cola esta vacía teniendo complejidad $O(1)$.
- La función `get` realiza la búsqueda de la clave en tiempo $O(1)$, en el peor de los casos es $O(n)$ si es que hay colisiones.
- La función `put` si es que la clave no se encuentra y la capacidad esta al máximo, la eliminación tiene costo $O(1)$ e insertar también es $O(1)$.

En resumen, la implementación de FIFO caché tiene un costo total de $O(1)$.

3. Complejidad de espacio

3.1. Para LRU

Considerando la estructuras utilizadas para LRU: Lista Doblemente Enlazada y Tablas Hash:

- Cada nodo almacena una clave (`key`), un valor (`value`), un puntero al nodo anterior (`prev`) y un puntero al siguiente nodo (`next`), esto implica que el espacio ocupado por el nodo es $O(1)$ para la clave y $O(1)$ para los punteros. Si hay n nodos, la lista va a requerir un espacio de $O(n)$.
- Para la Tabla Hash, almacena un par de clave-entero y un puntero a un nodo de la lista doblemente enlazada. El espacio requerido por cada entrada en el mapa hash es $O(1)$ para almacenar la clave y $O(1)$ para el puntero. Con n, el espacio total ocupado por la tabla hash es $O(n)$.

Entonces, La complejidad de espacio es dominada por el número de nodos en la lista doblemente enlazada y las entradas en el mapa hash, ambas de tamaño proporcional a la capacidad n dando una complejidad de espacio de $O(n)$.

3.2. Para LFU

Considerando las estructuras utilizadas: Binomial Heap y Tablas Hash.

- En la clase `Node` los nodos almacenan una clave, valor y frecuencia, un puntero al nodo padre, un vector de punteros a los nodos hijos y un grado del nodo teniendo la complejidad de espacio $O(1)$. Para n nodos, el espacio usado es proporcional a $O(n)$, ya que cada nodo ocupa espacio constante, por lo que la complejidad espacial es $O(n)$.
- En la clase `BinomialHeap` la estructura principal es el vector `trees` que almacena los punteros a los árboles. Está limitado el número de árboles por $O(\log n)$ en el peor de los casos. Cada nodo en el árbol puede tener hijos, por lo que el vector `children` puede almacenar $O(n)$ hijos.
- Para el `std::unordered_map` almacena la clave-valor y el mapeo de la clave-nodo. El espacio requerido es proporcional al número de elementos asociados, lo que sería $O(n)$ en el peor de los casos.
- Para la clase `LFUCache` el caché almacena n claves. El espacio para almacenar los elementos de la caché es $O(n)$ para el mapa `cache` y $O(n)$ para el mapeo del `key_node_map`.

En resumen, la complejidad de espacio para el LFU caché es $O(n)$ debido a que cada elemento se almacena en estructuras que requieren espacio lineal.

3.3. Para FIFO

La complejidad de espacio de la implementación de `FIFOCache` y la clase `Cola` depende de las estructuras de datos utilizadas: una cola manual (`Cola`) y un `std::unordered_map`.

- La clase `Cola` utiliza nodos dinámicos, donde cada nodo contiene dos campos: `dato` para almacenar un entero y `siguiente` como un puntero a otro nodo. Entonces el espacio por cada nodo es $O(1)$. Si tenemos n elementos, la complejidad pasa a ser $O(n)$
- Para el `std::unordered_map` que almacena la clave-valor tiene un espacio por cada par $O(1)$. Si hay n claves la complejidad sería de $O(n)$.
- La clase `FIFOCache` almacena hasta n elementos, donde n es la capacidad de la caché. La cola va a almacenar los n nodos teniendo una complejidad de espacio de $O(n)$ y almacenarlos los n pares tiene complejidad de $O(n)$.

Entonces, la complejidad de espacio del FIFO caché es de $O(n)$.