ALGORITMO LEAST RECENTLY USED (LRU) EN C MEDIANTE LISTA ENLAZADA SIMPLE.

Integrantes: Eloy Quezada (GitHub/Kotaix0807)

Constanza Araya (GitHub/Cnstnnza)

Cristobal Vidal (GitHub/crividal-maker)

Maximiliano Vargas (GitHub/maxvarga-lab)

Profesor: Christian Vasquez

24 de octubre de 2025

Índice

1. Introduccion

El presente informe describe el desarrollo e implementación de un sistema de manejo de caché denominado Least Recently Used (LRU), cuyo objetivo es optimizar el uso de memoria reemplazando los datos menos utilizados recientemente.

Este proyecto fue desarrollado en lenguaje C como parte del curso Estructuras de Datos, utilizando estructuras como () para gestionar el almacenamiento y la prioridad de los datos.

A través de esta implementación se busca comprender el funcionamiento de un algoritmo de reemplazo de caché y aplicar conceptos fundamentales como el manejo dinámico de memoria, el uso de punteros y la eficiencia algorítmica.

Este proyecto se desarrolló de manera colaborativa mediante el uso de Git, creando un repositorio en GitHub. Además, se aprovechó la extensión Live Share de Visual Studio Code para facilitar el trabajo en tiempo real entre los integrantes del equipo.

2. Objetivos

- Comprender el funcionamiento del algoritmo LRU (Least Recently Used).
- Implementar estructuras de datos abstractas como [] para la gestión del caché.
- Desarrollar habilidades en programación en lenguaje C, especialmente en el manejo de memoria y punteros.
- Simular el comportamiento de un sistema de caché limitado en tamaño, que reordene y elimine datos de acuerdo con su uso reciente.

3. Estructuras de Datos Utilizadas

Para la implementación del sistema de caché LRU, se utilizaron las siguientes estructuras de datos:

- Lista enlazada (Linked List)
- Puntero
- Funciones de manipulación de nodos

3.1. Puntero

Un puntero en informática (y en lenguajes como C o C++) es una variable que almacena la dirección de memoria de otra variable, en lugar de almacenar directamente un valor. Los punteros se utilizan para:

- Acceder y modificar valores indirectamente:

```
            *ptr = 20; // Cambia el valor de x a 20

```

Listing 1: Acceder y modificar valores indirectamente

- Listas enlazadas, árboles y grafos
- Paso de parámetros por referencia: Permite que una función modifique la variable original

```
                void incrementar(int *p) { *p = *p + 1; }

```

Listing 2: Paso de parámetros por referencia

- Memoria dinamica
- Eficiencia

A continuación, se muestra una imagen que ilustra el concepto de puntero:

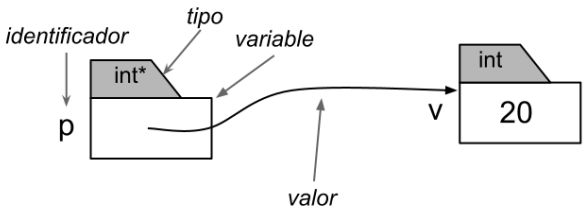

Figura 1: Puntero

3.2. Lista Enlazada

Una **lista enlazada** es una estructura de datos dinámica que consiste en una serie de elementos llamados nodos, donde cada nodo contiene un dato y una referencia al siguiente nodo en la secuencia. A diferencia de un array, las listas enlazadas permiten un crecimiento dinámico sin un tamaño fijo y facilitan la inserción y eliminación de elementos, ya que solo requieren actualizar referencias.
Para comprender mejor una lista enlazada es importante saber que es un nodo.

3.2.1. Nodo

En informática, un nodo es un elemento individual de una estructura de datos que almacena información y, en muchos casos, tiene referencias o enlaces a otros nodos. Los nodos son la base de muchas estructuras de datos dinámicas, como listas enlazadas, árboles y grafos.
Un nodo tiene dos componentes principales:

- **Dato**: La información que el nodo almacena, que puede ser de cualquier tipo de dato, como un número, una cadena de texto o un objeto más complejo.
- **Referencia**:Un puntero o referencia al siguiente nodo en la lista

En la siguiente imagen se puede apreciar de manera grafica que es un nodo

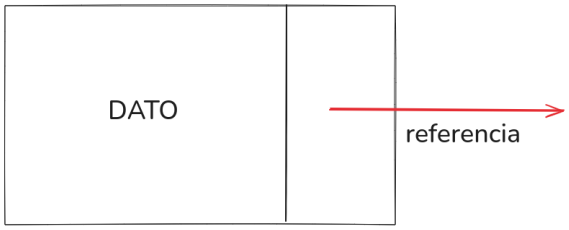

Figura 2: Nodo

Las listas enlazadas tienen un nodo inicial llamado **head** el cual apunta al primer elemento de la lista. Si se llega a un nodo que no apunta a ningún otro, se dice que dicho nodo apunta a **null** o final de la lista.
En la siguiente imagen se puede apreciar de manera grafica que es una lista enlazada

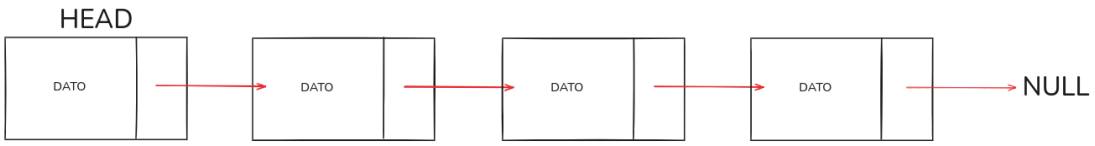

Figura 3: Lista enlazada

Se utilizó una lista enlazada para mantener el orden de los elementos en la caché, permitiendo una fácil inserción, eliminación y reordenamiento de los nodos según su uso reciente, de esta manera se garantiza que los datos más utilizados recientemente se encuentren en la parte superior de la lista, mientras que los datos menos utilizados se encuentren en la parte inferior.

3.2.2. Implementacion mediante Lista Enlazada

La lista enlazada (o lista ligada simple), definida por las estructuras en `nodes.h` y gestionada por `LRUcache`, es la **estructura de datos central** que implementa la cache LRU. Su funcion va mas alla del simple almacenamiento: es el mecanismo que mantiene el orden de recencia de uso (uso reciente).

Mantenimiento del Orden de Recencia (MRU $\rightarrow$ LRU) La lista se manipula estrictamente para que sus extremos reflejen la recencia de uso:

- **Inicio de la Lista**: Representa el **Most Recently Used** (MRU), accesible en $O(1)$ mediante el puntero `cache->primero`.
- **Final de la Lista**: Representa el **Least Recently Used** (LRU), accesible en $O(1)$ mediante el puntero `cache->ultimo`.

Uso en la Logica de la Cache La lista enlazada es esencial para todas las operaciones clave de la cache:

- **Insercion/Reubicacion**: Las operaciones de agregar y usar un dato siempre implican mover el nodo al inicio de la lista para reflejar el nuevo estado MRU.
- **Desalojo**: La eliminacion del LRU se realiza directamente desde el final de la lista, ya que es el candidato menos recientemente usado.

3.3. Manipulacion de Nodos

La manipulacion de los nodos fue fundamental para implementar la politica LRU, asegurando que la lista ligada mantuviera el orden de recencia de uso. Los nodos son gestionados directamente mediante punteros para mantener al inicio

el MRU y al final el LRU.

Reubicacion de Uso (Reubicacion MRU)

Cuando se accede a un nodo existente (`usar_dato`), este debe ser movido a la posicion MRU. El proceso implica:

1. Localizar el nodo (`actual`) y su predecesor (`prev`). Esta busqueda es $O(N)$.
2. **Desvincular**: El puntero `prev->Next` se salta `actual`.
3. **Re-insertar al inicio**: El nodo `actual` se convierte en el nuevo `cache->main->Next`.
4. **Actualizacion de LRU**: Si el nodo movido era el `cache->ultimo`, el puntero se actualiza a `prev`.

Eliminacion del Menos Usado (LRU)

El desalojo se activa cuando la capacidad se excede. Se elimina el nodo al final de la lista.

- Se recorre la lista hasta encontrar el penultimo nodo (`prev`) que precede al LRU (`actual`).
- El LRU es desvinculado al hacer que `prev->Next` apunte a `NULL`.
- El puntero `cache->ultimo` se establece a `prev`.
- Se libera la memoria del nodo eliminado (`free(actual)`).

Mecanismo de Carga

Para reconstruir la cache desde el disco sin alterar el orden guardado, se utiliza la funcion auxiliar `insertar_al_final`. Esta funcion anade nodos directamente utilizando `cache->ultimo` en $O(1)$, evitando la logica de reubicacion MRU de `agregar_dato`.

3.4. Estructura de la Cache LRU (`LRUcache`)

La gestion de la lista enlazada y el control de la capacidad se centralizan en la estructura `LRUcache`, definida en `lru.h`.

Listing 3: Estructura de Control LRUcache

```
typedef struct {
    int max_cache;
    List main;
    Position primero;
```

```
    Position ultimo;
} LRUcache;
```

- `max_cache`: Almacena la capacidad maxima permitida para la cache.
- `main` **(List)**: Puntero al nodo cabecera de la lista enlazada, sirviendo como punto de acceso para todas las operaciones.
- `primero` **(Position)**: Puntero auxiliar que apunta directamente al nodo **Mas Recientemente Usado (MRU)**. Facilita la insercion en $O(1)$.
- `ultimo` **(Position)**: Puntero auxiliar que apunta directamente al nodo **Menos Recientemente Usado (LRU)**. Facilita la eliminacion por desalojo en $O(1)$ una vez localizado su predecesor.

4. Implementacion de comandos

4.1. Lru create N

EEste comando se encarga de crear la memoria cache y designar un máximo de elementos dentro del mísmo. **¿Cómo lo hace?** En primer lugar, se implemento el uso de argumentos, y estos se ingresan como argumento para la función que está condicionada con un if, ya que, si en la ejecución de la función, existe un error, entonces la función completa devolverá 1, y por consecuencia, main.c también lo hará. Luego, se crearán:

- lru_cache (carpeta)
- metadata.txt
- data.txt

En metadata.txt se guardarán los datos del cache, mientras que en data.txt, se guardará el orden de la lista enlazada.

4.2. Lru add_data

Este comando añade elementos al cache actual (Se añaden elementos a la lista). En cada llamada de la función, primero se cargarán los datos del cache. Luego, se asegura de que, si el dato que se quiere ingresar ya existe, entonces encuentra ese nodo, y lo marca como usado (**MRU: Most Recently Used**). De no ser así, contará la cantidad de datos actuales, la comparará con la cantidad máxima, y de ser excedida esa capacidad, entonces eliminará el dato menos usado con la función ”eliminar_menos_usado”.

4.3. Lru get data <data>

Esta función cumple el rol de ’Usar’ un dato especificado, así se cambiará su lugar en la lista enlazada, marcado como **MRU** (Most Recently Used). Para ello, primero **buscará** el dato ingresado mediante un bucle `while` (Función `FindNode`), que tiene una complejidad de $O(N)$. Si el dato ya es MRU (`cache->primero`), entonces devolverá un mensaje sin realizar movimientos. Del contrario y una vez encontrado, se **invierte su recencia** a través de la función `usar_dato`.

1. **Desvinculación**: Se desvincula el nodo encontrado de su posición actual mediante la actualización de los punteros `Next` de su nodo predecesor.
2. **Reubicación**: El nodo es re-insertado inmediatamente al inicio de la lista (después de la cabecera) para establecerlo como el nuevo **MRU**. El puntero `cache->primero` es actualizado.

3. **Persistencia**: Finalmente, el nuevo orden de la lista enlazada es **guardado** en el archivo `data.txt`, garantizando que el estado de recencia actualizado se conserve en el disco.

Este proceso es fundamental para la política LRU, ya que garantiza que los datos consultados mantengan su prioridad y no sean desalojados prematuramente.

4.4. Lru add_data <data>

Este comando añade elementos al cache actual (Se añaden elementos a la lista). En cada llamada de la funcion, primero se cargan los datos del cache. Luego, se asegura de que, si el dato que se quiere ingresar ya existe, entonces encuentra ese nodo, y lo marca como usado (**MRU: Most Recently Used**) a traves de `usar_dato`. De no ser asi, contara la cantidad de datos actuales, la comparara con la cantidad maxima, y de ser excedida esa capacidad, entonces eliminara el dato menos usado con la funcion `eliminar_menos_usado`. Tras la operacion, el estado de la lista es persistido en `data.txt`.

4.5. Lru get data <data>

Esta funcion cumple el rol de 'Usar' un dato especificado, asi se cambiara su lugar en la lista enlazada, marcado como **MRU** (Most Recently Used) si es encontrado.

- **Carga y Busqueda**: Primero se cargan los datos y luego se busca el valor.
- **Reubicacion**: Si el dato existe y no es ya el MRU, se llama a la funcion `usar_dato` para desvincular el nodo de su posicion y re-insertarlo al inicio.
- **Persistencia**: Al igual que con `add_data`, el nuevo orden de la cache es guardado en `data.txt` para reflejar la recencia actualizada.

Este comando garantiza que los datos consultados mantengan su prioridad en la cache.

5. Persistencia y Manejo de Archivos

La **persistencia** es la cualidad de los datos de **sobrevivir más allá del ciclo de vida del programa** que los creó o manipula. En este proyecto, garantiza que el estado y el orden de recencia (**MRU** $\rightarrow$ **LRU**) de la caché se mantengan entre diferentes ejecuciones del programa.

5.1. Estructura de Persistencia

La cache utiliza el sistema de archivos del sistema operativo para lograr la persistencia, basándose en la siguiente estructura:

- **Directorio** `lru_cache`: Contiene todos los archivos de la cache. La creacion se maneja por la funcion `CreateDir`.
- `metadata.txt`: Archivo que almacena informacion estatica, principalmente la capacidad maxima (`max_cache`).
- `data.txt`: Archivo donde se guardan los datos de la lista enlazada, un elemento por linea, manteniendo el orden de recencia (MRU $\rightarrow$ LRU).

5.2. Mecanismos de Persistencia (Carga y Guardado)

La gestion del estado persistente se logra mediante las siguientes operaciones:

- **Carga** (`LoadCache`): Se ejecuta al inicio de cada comando de manipulacion. Lee `metadata.txt` y `data.txt`. Utiliza la funcion auxiliar `insertar_al_final` para **reconstruir la lista en memoria (RAM)** en el orden exacto en que fue guardada, evitando que la logica de reubicacion LRU altere el orden de carga.
- **Guardado (Escritura)**: Despues de cada modificacion (`add_data` o `get data`), el estado actual y orden de la lista enlazada se sobreescribe completamente en `data.txt` para reflejar la recencia mas reciente de la cache.

6. Conclusion

La implementacion demostro ser exitosa en simular la politica LRU, utilizando la lista enlazada para mantener la jerarquia de uso. La gestion mediante los punteros auxiliares `primero` y `ultimo` permitio lograr una complejidad de tiempo eficiente, de $O(1)$ para las operaciones de insercion MRU y eliminacion LRU. Sin embargo, la busqueda de datos dentro de la lista sigue siendo de $O(N)$, lo cual podria mejorarse en futuras iteraciones. Para optimizar el rendimiento del sistema, se podrian realizar los siguientes cambios a futuro:

- **Optimizar Busqueda**: Migrar la implementacion a una **Lista Doblemente Enlazada** y combinarla con un **HashMap** (tabla hash). Esto reduciria la complejidad de la busqueda de elementos de $O(N)$ a $O(1)$.
- **Gestion de Errores**: Mejorar el manejo de errores de persistencia y archivos.

Referencias

esantos. (2024, noviembre). *¿Qué es una Lista Enlazada?* Consultado el 24 de octubre de 2024, desde https://www.ervinsantos.com/post/50-%C2%BFqu%C3%A9-es-una-lista-enlazada

microsoft. (2023, octubre). *Punteros (C++)*. Consultado el 24 de octubre de 2024, desde https://learn.microsoft.com/es-es/cpp/cpp/pointers-cpp?view=msvc-170

wikipedia. (2024, diciembre). *Nodo (informática)*. Consultado el 24 de octubre de 2024, desde https://es.wikipedia.org/wiki/Nodo_(inform%C3%A1tica)