Implementación de Caché LRU.

- **Asignatura**: Estructuras de Datos
- **Profesor**: Christian Vásquez
- **Fecha**: 24 de octubre de 2025
- **Integrantes**:
  - Diego Herrera
  - Simon Navarro
  - Dámaris Ahumada
  - Catalina Viñas
  - Gabriel Hernández

Contenidos:

1. INTRODUCCIÓN:

Este trabajo presenta el desarrollo de una implementación funcional del algoritmo LRU en lenguaje C, utilizando listas doblemente enlazadas como estructura de datos principal. La elección de esta estructura permite realizar operaciones de inserción, eliminación y reordenamiento en poco tiempo, lo cual es esencial para mantener la eficiencia del sistema caché.

El proyecto no solo busca implementar el algoritmo desde el punto de vista teórico, sino también desarrollar un sistema completo que incluya validación de entradas, manejo de errores, y una interfaz de línea de comandos intuitiva que permita al usuario interactuar con la caché de manera efectiva. Además, se ha puesto especial énfasis en la portabilidad del código, garantizando su funcionamiento tanto en sistemas Windows como Linux.

A lo largo de este documento se detalla el proceso de desarrollo, las decisiones de diseño tomadas, los desafíos enfrentados y las soluciones implementadas, proporcionando una visión completa del trabajo realizado.

2. OBJETIVOS:

2.1) Objetivo general:

Desarrollar un sistema funcional de gestión de caché LRU implementado en lenguaje C, que permita la manipulación eficiente de datos mediante el uso de estructuras de datos dinámicas.

2.2) Objetivos Específicos:

- Comprender y aplicar el algoritmo LRU para la gestión de memoria caché.
- Implementar estructuras de datos como listas doblemente enlazadas para el almacenamiento y manipulación de información.
- Implementar un sistema de prioridades dinámico basado en el uso de los datos.

---

3. MARCO TEÓRICO

3.1 Algoritmo LRU

El algoritmo LRU (Least Recently Used) se fundamenta en la idea de que los datos que no han sido utilizados recientemente tienen menos probabilidad de ser requeridos en el futuro próximo. Para aplicarlo, se lleva un registro del orden de acceso a las páginas. Cuando es necesario liberar espacio en la memoria, LRU reemplaza aquella página que lleva más tiempo sin ser utilizada. Por ejemplo, si se accede a las páginas A, B, C y D en ese orden, y luego se vuelve a acceder a A, al intentar cargar una nueva página E, el algoritmo eliminará la página B, ya que es la que ha permanecido más tiempo sin uso.

Este algoritmo mantiene un registro del orden de uso de todos los elementos en la caché. Cada vez que se accede a un dato:

1. Si el dato ya existe, en la caché, se mueve a la posición de máxima prioridad.
2. Si el dato no existe y hay espacio disponible se agrega con máxima prioridad.
3. Si el dato no existe y la caché está llena, se elimina el elemento con menor prioridad(el menos usado recientemente) y se inserta el nuevo dato con máxima prioridad.

Este mecanismo asegura que los datos más relevantes permanezcan disponibles en la memoria caché, optimizando así el rendimiento del sistema.

Este algoritmo es bastante útil para;

- sistemas operativos: para gestión de páginas de memoria virtual
- Bases de datos: Para mantener en caché las consultas y resultados más frecuentes
- Navegadores web: Para almacenar páginas y recursos recientemente visitados.
- Entre otros usos.

3.2 Estructuras de Datos

Una lista doblemente enlazada es una estructura de datos lineal en la que cada nodo contiene tres componentes fundamentales

- Dato: Es la información.
- Puntero al siguiente.
- Puntero al anterior.

Algunas operaciones básicas son

- Insertar al inicio: El nuevo nodo se convierte en la cabecera
- Eliminar el nodo final.
- Buscar elementos.
- Entre otras.

4. IMPLEMENTACIÓN

4.1 Estructuras Principales

- Estructura `Dato:`

```
typedef struct Dato
{
	char *valor;
	struct Dato *siguiente;
	struct Dato *anterior;
} Dato;
```

Esta estructura representa un nodo individual en la lista doblemente enlazada. Cada nodo contiene:

- Valor: Cadena de caracteres que almacena el contenido del dato.
- Siguiente: Puntero al siguiente nodo en la lista.
- Anterior: Puntero al nodo anterior en la lista.

- Estructura `Dato:`

```
typedef struct Cache
{
	int tamanho_memoria;
	int memoria_ocupada;
	Dato *cabecera;
	Dato *cola;
} Cache;
```

Esta estructura gestiona la información del caché:

- tamanho:memoria: Capacidad máxima del caché (definida por el usuario).
- memoria_ocupada: Número actual de elementos almacenados.

- cabecera: puntero al elemento más recientemente usado(Máxima prioridad).
- cola: Puntero al elemento menos usado recientemente (Mínima prioridad).

4.2 Funciones Implementadas

- Create_cache: Esta función inicializa la memoria del caché según el tamaño que quiera el usuario.

  Como parámetros tiene a:

  - Cache *cache: puntero a la estructura cache.
  - char *tamanho: puntero a cadena de caracteres, que representa el tamaño que desea el usuario.

  Funcionamiento:

  1. Revisa si ya existía un caché anteriormente, si lo tenía éste libera toda la memoria del caché anterior, mostrando cuántos elementos contiene y posteriormente creando un cache nuevo
  2. Convierte el tamaño ingresado por el usuario a un número y verifica que sea válido, en caso de que no lo sea como por ejemplo haya colocado unas letras, este mostrará un error.
  3. Verifica que el tamaño ingresado por el usuario esté dentro del rango permitido (entre 5 y MAX_CACHE_SIZE), si es válido continua y si no es muestra el mensaje de error y se cancela.
  4. Se inicializan los campos de la estructura cache:
     - cache -> memoria_ocupada = 0
     - cache -> cabecera = NULL
     - cache -> cola = NULL
  5. Muestra un mensaje diciendo que el caché fue creado y su tamaño.
- Add_data:

  Esta función agrega un nuevo dato al caché o actualiza la prioridad de uno existente.

  Como parámetros tiene a:

  - Cache *cache: puntero a la estructura caché
  - char ¨*nombre: string con la letra mayúscula a agregar.

  Su funcionamiento es de la siguiente manera:

  1. Verifica que el caché haya sido creado.
  2. Valida que el nombre sea una sola letra mayúscula (A-Z).
  3. Busca si el dato ya existe en la caché:
     - Si existe: Lo mueve a la cabecera (actualiza la prioridad).
     - Si no existe: Continúa con la inserción.
  4. Si la caché está llena:
     - Guarda referencia al nodo cola.

        - Actualiza el puntero de cola al nodo anterior.
        - Anula los enlaces del nodo eliminado.
        - Libera memoria del valor y del nodo.
        - Decrementa memoria_ocupada
    5. Crea un nuevo nodo:
        - Asigna memoria para el nodo.
        - Asigna memoria para el valor.
        - Copia la letra al valor.
    6. Inserta en la cabecera:
        - Establece anterior=NULL.
        - Establece siguiente=cabecera actual.
        - Actualiza el anterior de la cabecera anterior.
        - Si la lista está vacía, actualiza también la cola.
    7. Incrementa memoria_ocupada.

- Get_data:

  Con esta función usamos un dato existente, así actualizando su prioridad al moverlo a la cabecera.

  Sus parámetros son:

  - Cache *cache: puntero a la estructura de caché.
  - const char *nombre: String con la letra a buscar y usar.

  Su funcionamiento es el siguiente:

  1. Verifica que el caché esté creado.
  2. Valida que el nombre sea correcto (una letra mayúscula).
  3. Recorre la lista buscando el dato.
  4. Si la encuentra:
      - Llama a mover_a_cabecera para reorganizar.
      - Informa al usuario que el dato fue utilizado.
  5. Si no la encuentra:
      - Informa que el dato no está en caché.

  Sobre la función auxiliar mover_a_cabecera:

  - Si el nodo ya es cabecera, no hace nada.
  - Separa el anterior nodo de su posición actual.
  - Inserta al inicio.

- Search_data:

  Esto sirve para buscar un dato en la caché por su nombre.

  Sus parámetros son:

  Cache *cache: puntero a la estructura del caché.

  const char *nombre: String con el nombre o valor a buscar dentro del caché.

  Funcionamiento:

  1. Se comprueba si la caché fue creada revisando si cache->tamanho_memoria es igual a 0, Si no existe una caché activa, se muestra el mensaje de error, y la función retorna -1.
  2. Se verifica que el nombre entregado como parámetro sea válido con la función validar_nombre(nombre), Si el nombre no cumple con los criterios, se muestra el mensaje de error y también retorna -1.
  3. se declara una variable entera pos con valor inicial1, que sirve para contar la posición del elemento dentro de la lista enlazada
  4. Se recorre la lista utilizando un puntero t que avanza nodo por nodo mientras exista un elemento en la lista.
  5. Dentro del ciclo, se compara el contenido de cada nodo con el nombre buscado usando strcmp. Si ambos coinciden, se imprime la posición actual y se retorna el valor de pos.
  6. Si se recorre toda la lista sin hallar coincidencias, se imprime -1 y la función retorna -1, indicando que el dato no fue encontrado en la caché

- All_cache: Esta función muestra todos los datos que hay en el caché ordenados por prioridad.

  Parámetro usado:

  - Cache *cache: puntero a la estructura cache

  Funcionamiento:

  1. Verifica que el caché haya sido creado, si fue creada sigue continuando y en caso de no ser creada muestra un mensaje que esta no ha sido creada.
  2. Revisa si la caché está vacía, si no hay nada muestra un mensaje diciendo que la caché está vacía, en caso de que no esté vacía continua.
  3. Se recorre toda la lista mostrando todos los datos ordenados por prioridad de uso.
- Exit_cache: Esta función tiene como objetivo cerrar correctamente el programa liberando todos los recursos utilizados por la caché

  Los parámetros usados son:

- Cache *cache: puntero hacia la estructura caché
- int *running: puntero hacia el valor que controla el ciclo while del main

Funcionamiento:

1. Se comprueba si la caché tiene elementos utilizando un ciclo while que se ejecuta mientras el puntero cabecera no sea nulo. Esto indica que aún existen nodos en la lista enlazada que deben ser liberados.
2. El nodo apuntado por cabecera se almacena temporalmente en temp para poder liberarlo sin perder la referencia al siguiente.
3. Se actualiza cache->cabecera para que apunte al siguiente nodo.
4. Se libera primero la memoria del valor almacenado y luego la del nodo.
5. Al finalizar el ciclo, se establece cache->cola = NULL y cache->memoria_ocupada = 0, dejando la estructura vacía pero conservando el tamaño configurado.
6. Si el puntero running es válido, se cambia su valor a false y se muestra el mensaje “Saliendo del programa …”, indicando que el programa ha terminado correctamente.

4.3 Organización del Código

- Archivos .c utilizados:
  - main.c : Aquí se encuentra la unión de todas las funciones y el programa principal
  - comandos.c : Contiene todas las funciones principales del algoritmo LRU
  - funciones.c : Contiene las funciones auxiliares como la conversión de char a int.
- Archivos .h utilizados:
  - comandos.h : Declara las funciones que se implementan en comandos.c para que sean utilizables.
  - funciones.h : Declara las funciones utilizadas en funciones.c
  - estructuras.h : Se definen las estructuras de datos utilizadas (Dato y Cache)
  - constantes.h : Definición de constantes (tamaño y valores de control.

5. TÉCNICAS Y DECISIONES DE DISEÑO

La lista doblemente enlazada fue seleccionada como estructura de datos principal en este trabajo por su eficiencia algorítmica y porque se adapta perfectamente a lo que necesitábamos. Sus ventajas se notan en operaciones como insertar en la cabecera, eliminar desde la cola y mover nodos de forma rápida.

Una de las razones clave fue que permite reorganizar los elementos sin recorrer toda la lista ni copiar datos. Solo se actualizan algunos punteros, y los nodos permanecen en su lugar original en memoria. Además, al tener acceso tanto hacia adelante como hacia atrás, se puede buscar desde cualquier extremo sin complicaciones.

Al principio pensamos en usar archivos, pero nos dimos cuenta de que trabajar con listas enlazadas era más eficiente y mucho más sencillo para lo que requería el proyecto.

El manejo de prioridades se implementó de forma implícita según la posición en la lista. Por ejemplo:

A - B - C - D

Aquí, A es la cabecera (más reciente) y D es la cola (menos usada). Lo bueno es que al mover un nodo, su prioridad se actualiza automáticamente. Cada vez que se llama a la función add_data, el dato se mueve a la cabecera, y eso reorganiza el resto sin esfuerzo.

Nos interesaba que mover los datos dentro de la caché fuera rápido y sin enredos. Por eso elegimos esta estructura: con los punteros anterior y siguiente, cambiar la posición de un nodo es directo. Así, funciones como add_data, get_data o mover_a_cabecera se mantienen claras y ordenadas.

6. DESAFÍOS Y SOLUCIONES

Algunos desafíos que enfrentamos como grupo fueron:

6.1) Selección de la estructura de datos adecuada.

Al inicio del proyecto, el equipo se enfrentó a la problemática sobre qué estructura de datos utilizar para implementar el caché LRU. Se consideraron dos alternativas:

- Archivos.
    - Leer y reescribir el archivo en cada operación.
    - Usar el orden de las líneas para representar prioridades.
- Lista doblemente enlazada en memoria.
    - Mantener todos los datos en estructuras dinámicas.
    - Manipular punteros para reorganizar elementos.
    - Gestionar memoria manualmente.

Al inicio pensamos que lo mejor era archivos ya que lo manejamos mejor pero después de hablar con el grupo decidimos que por su eficiencia, y que se alinea con los objetivos de la tarea, era mucho mejor la lista doblemente enlazada como estructura de datos a utilizar en este trabajo.

7. PRUEBAS

A continuación se presentan las pruebas realizadas para validar el funcionamiento del sistema en distintos escenarios.

7.1) Tabla 01: Pruebas

N	Tipo de prueba	Descripción	Entrada	Resultado obtenido	Estado
1	Creación de caché	Crear caché válido	-c 5	Caché de tamaño 5 creado.	Éxito.
2	Creación de caché	Tamaño inválido (menor a 5)	-c 2	El tamaño debe ser entre 5 y 100	Éxito.
3	Creación de caché	Tamaño inválido (mayor a 100)	-c 120	El tamaño debe ser entre 5 y 100	Éxito.
4	Creación de caché	Entrada no numérica	-c ab	El valor ab no es un número válido.	Éxito.
5	Inserción	Agregar primer elemento	-a A	Dato agregado a la caché	Éxito.
6	Inserción	Agregar hasta llenar caché	-a A, -a B, -a C, -a D, -a E	5 Datos agregados correctamente a la caché.	Éxito.
7	Inserción	Agregar con caché lleno	-a F	Dato agregado correctamente a la caché, también se eliminó el dato con menor	Éxito.

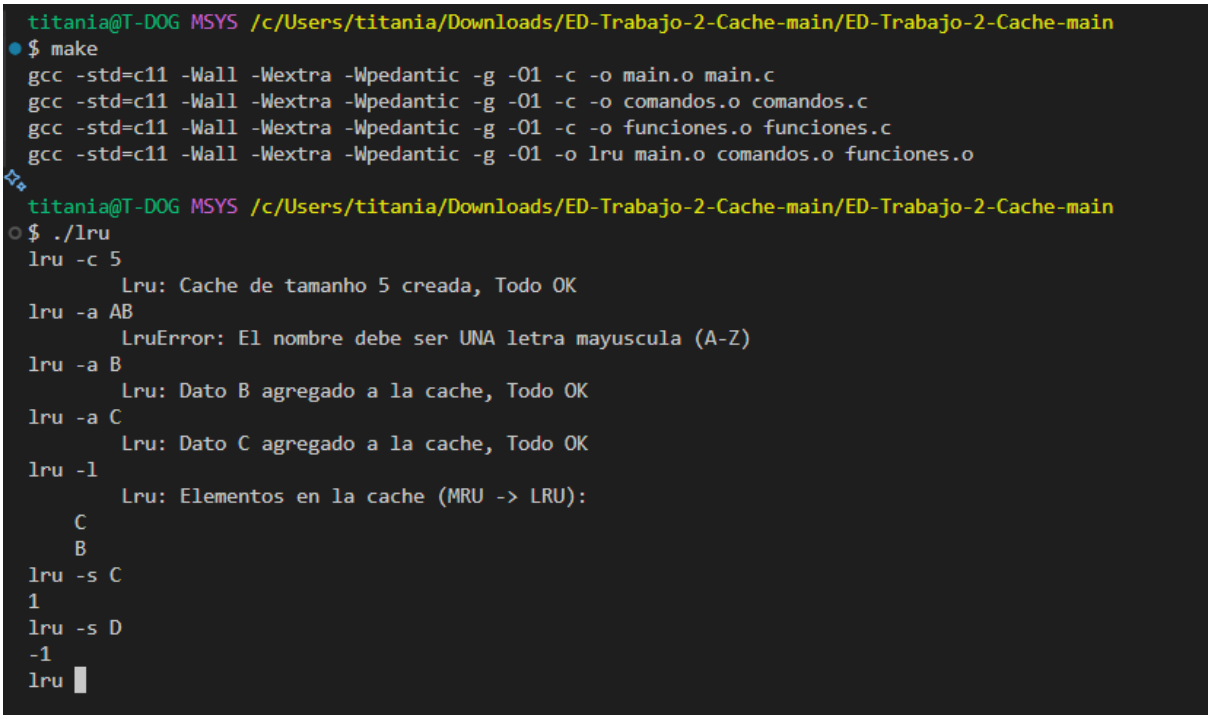
7.1) Tabla 01: Pruebas					
				prioridad : F,E,D,C,B	
8	Inserción	Agregar dato existente	-a B	Dato B utilizado (prioridad actualizada)	Éxito.
9	Inserción	Nombre inválido (minúscula)	-a a	Entrada inválida, debe ser UNA letra mayúscula (A-Z)	Éxito.
10	Inserción	Nombre inválido (múltiples chars)	-a AB	Entrada inválida, debe ser UNA letra mayúscula (A-Z)	Éxito.
11	Búsqueda	Buscar elemento existente	-s B *Caché: C,B,A	Nos entregó la posición de B en este caso 2.	Éxito.
12	Búsqueda	Buscar elemento en cabecera	-s C *Caché: C,B,A	Nos entregó la posición de C en este caso 1.	Éxito.
13	Búsqueda	Buscar elemento inexistente	-s D *Caché: C,B,A	Retorno -1, que significa que el dato no es válido.	Éxito.
14	Búsqueda	Búsqueda no modifica prioridad	-s B -l	Nos retorna la posición y luego nos muestra el caché, no se ven cambios.	Éxito.
15	Uso	Usar elemento existente	-g A *Caché: C,B,A	A, pasa a estar en la cabecera (aumenta su prioridad).	Éxito.
16	Uso	Usar elemento ya	-g A *Caché:	No cambia nada, A ya	Éxito.

7.1) Tabla 01: Pruebas					
		en cabecera	A,C,B	era el dato con mayor prioridad	
17	Uso	Usar elemento inexistente	-g D *Caché: A,C,B	D no pertenece a un dato del caché, así que no hace nada, solo muestra el mensaje de error.	Éxito.
18	Visualización	Ver caché con elementos	-l *Caché: A,C,B	Nos muestra caché por orden de mayor prioridad.	Éxito.
19	Visualización	Ver caché vacío	-l	Muestra el mensaje de que la caché está vacía	Éxito.
20	Error	Operación sin crear caché	-a sin haber llamado a -c	Error: Debemos llamar -c para crear la caché.	Éxito.
21	Reinicio	Crear caché múltiples veces	-c 5, luego -c 7	Caché reiniciado con tamaño 7	Éxito.
22	Salida	Salir del programa	-e	Memoria liberada, programa termina.	Éxito.

Las 22 pruebas realizadas permitieron validar exhaustivamente el funcionamiento del sistema de caché LRU en todos sus aspectos: creación, inserción, búsqueda, uso, visualización, reinicio y salida. Cada caso fue diseñado para cubrir tanto escenarios típicos como situaciones límite, asegurando robustez y estabilidad.

El sistema respondió correctamente ante entradas inválidas, caché lleno, uso de datos inexistentes y operaciones sin inicialización previa. Se confirmó que el algoritmo LRU reorganiza eficientemente los datos según su uso reciente, manteniendo la coherencia entre comandos como -a, -g y -s.

7.2) Compilación y pruebas en Windows y Linux



```
lru -s D
-1
lru -l
        Lru: Elementos en la cache (MRU -> LRU):
    C
    B
lru -c 7
        Lru: Reiniciando cache existente (contenia 2 elemento(s))
        Lru: Saliendo del programa ...
        Lru: Cache de tamanho 7 creada, Todo OK
lru -a 5
        LruError: El nombre debe ser UNA letra mayuscula (A-Z)
lru -l
        Lru: La cache esta vacia
lru -a A
        Lru: Dato A agregado a la cache, Todo OK
lru -a B
        Lru: Dato B agregado a la cache, Todo OK
lru -a C
        Lru: Dato C agregado a la cache, Todo OK
lru -l
        Lru: Elementos en la cache (MRU -> LRU):
    C
    B
    A
lru -g B
        Lru: Dato B utilizado
lru -l
        Lru: Elementos en la cache (MRU -> LRU):
    B
    C
    A
lru -e
        Lru: Saliendo del programa ...
```

Los comandos y la forma de ejecutar el programa es la misma tanto para Windows y para linux.

8. CONCLUSIONES

- Qué se logró: En este trabajo se logró desarrollar e implementar un sistema de gestión de caché LRU completamente funcional en lenguaje C, que permite realizar operaciones como agregar, buscar y utilizar datos de manera eficiente, respetando siempre la prioridad basada en el uso reciente de cada elemento. Gracias al uso de listas doblemente enlazadas, fue posible mantener la organización de los datos de forma dinámica, permitiendo que los elementos más utilizados se mantuvieran en la cabecera y los menos usados se eliminarán automáticamente cuando la caché alcanzaba su capacidad máxima. Se implementaron funciones modulares para crear, agregar, consultar, usar, mostrar y eliminar datos de la caché, garantizando un código organizado, legible y fácil de mantener. Además, se desarrolló un control de errores robusto para manejar situaciones como la creación de la caché, ingreso de datos inválidos o intentos de acceso a datos inexistentes, lo que asegura la estabilidad del programa durante su ejecución. El proyecto también permitió simular de manera práctica cómo un sistema real administra la memoria limitada, lo que se refleja en el ordenamiento y la eliminación eficiente de los datos menos utilizados.
- Qué se aprendió: El desarrollo de este proyecto permitió profundizar significativamente en el manejo de estructuras de datos dinámicas y en la gestión manual de memoria en C. Aprendimos a implementar listas doblemente enlazadas para resolver problemas de eficiencia, comprendiendo la importancia de mantener correctamente los punteros anterior y siguiente al reorganizar los nodos. Se adquirieron conocimientos prácticos sobre cómo aplicar el algoritmo LRU para optimizar el uso de memoria caché y cómo este algoritmo es utilizado en sistemas operativos, bases de datos y navegadores web para mejorar el rendimiento. Además, se fortalecieron habilidades de programación modular, diseño de funciones y documentación de código, asegurando que cada parte del programa cumpla con una tarea específica y pueda ser entendida y mantenida fácilmente. Por otro lado, el proyecto permitió experimentar con la toma de decisiones de diseño, evaluando alternativas como el uso de archivos frente a estructuras en memoria, y comprendiendo cómo la elección de la estructura adecuada afecta directamente la eficiencia y la simplicidad del sistema. Finalmente, se aprendió a trabajar en equipo, distribuyendo tareas de implementación, pruebas y documentación, lo que facilitó la coordinación, el intercambio de ideas y la resolución conjunta de los desafíos que surgieron durante el desarrollo del proyecto.