A.E.D. Laboratorio 3

Guillermo Román
guillermo.roman@upm.es

Lars-Åke Fredlund
lfredlund@fi.upm.es

Manuel Carro
mcarro@fi.upm.es

Julio García
juliomanuel.garcia@upm.es

Tonghong Li
tonghong@fi.upm.es

Normas

- Fechas de entrega y penalización:

Hasta el martes 28 de octubre, 16:00 horas	0 %
Hasta el miércoles 29 de octubre, 16:00 horas	20 %
Hasta el jueves 30 de octubre, 16:00 horas	40 %

  Después la puntuación máxima será 0
- Se comprobará plagio y se actuará sobre los detectados.
- Usad las horas de tutoría para preguntar sobre programación – son oportunidades excelentes para aprender.

Entrega

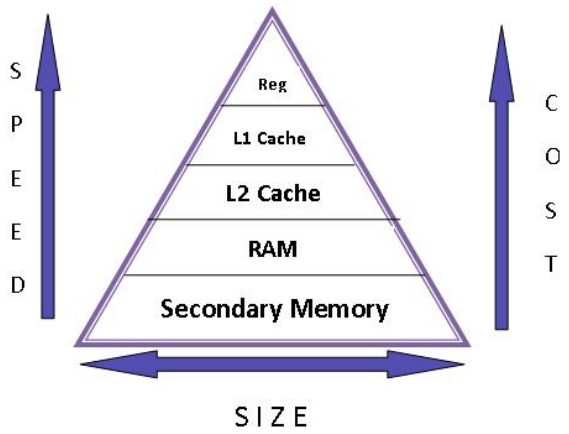
- Todos los ejercicios de laboratorio se deben entregar a través de

  `http://deliverit.fi.upm.es`

- El/los fichero(s) que hay que subir es/son `Cache.java`.
- La clase debe estar en el paquete `aed.cache` .
- La documentación de la API de `aedlib.jar` está disponible en `http://costa.ls.fi.upm.es/teaching/aed/docs/aedlib/`

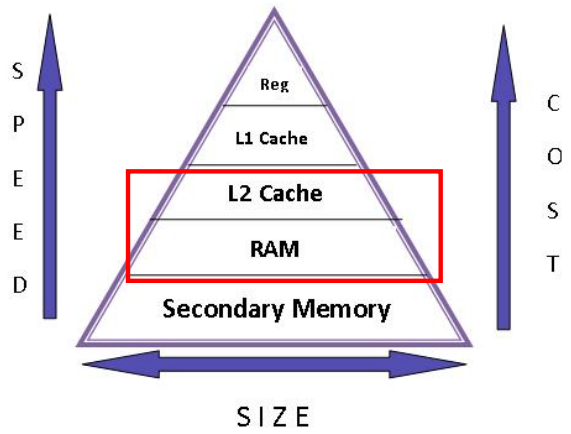
Tarea: implementar/simular una memoria *caché*

- Memoria ordenador: más rápida $\Leftrightarrow$ más cara.
- Jerarquía: mucha memoria lenta, poca memoria rápida.



Tarea: implementar/simular una memoria *caché*

- Memoria ordenador: más rápida $\Leftrightarrow$ más cara.
- Jerarquía: mucha memoria lenta, poca memoria rápida.



- Memoria caché (C): complemento a memoria principal (RAM - M):
  - Leer y escribir en C es **mucho** más rápido que en M.
  - Por tanto, es preferible acceder a C en lugar de a M.
  - Pero el tamaño de C es **mucho** menor que el de M.
  - Hay que elegir qué datos se guardan en C: C contiene copias sólo de **algunos** datos de M.
- En informática se usan memorias caché en muchas ocasiones: en procesadores (CPUs), para acceder rápidamente a páginas web, ...

Ejemplo de caché

Memoria M

key	value
8938343U	Pamela Martínez
2377373L	José González
X3773478	Stina Karlsson
X6553279	Jim Smith
2382882I	Noelia Martín
8232839G	Clara Fernández
3727347Z	George Romero
2388282H	Francisco García
3634639U	Susana Díaz

Caché (tamaño 3)

key	value
2377373L	José González
2382882I	Noelia Martín
2388282H	Francisco García

- Caché: subconjunto de los elementos (clave y valor) en M
- Operaciones: leer y escribir.

Implementación de la caché

Interfaz

Queremos implementar los algoritmos de una memoria caché simple con solo dos operaciones:

- Acceder a un valor: `get()`
- Añadir / cambiar un valor: `put()`

```
public class Cache<Key,Value> {
  // Constructor de la cache. Especifica la capacidad maxima
  // y la memoria que se va a utilizar
  public Cache(int maxSize, Storage<Key,Value> mainMemory) { ... }

  // Devuelve el valor que corresponde a una clave "Key"
  public Value get(Key key) { ... }

  // Establece un valor nuevo para la clave en la memoria cache
  public void put(Key key, Value value) { ... }
}
```

Implementación de la caché

Memoria principal

La memoria M está implementada en la clase `Storage`

```
public class Storage<Key,Value> {
  public Value read(Key key) { ... }
  public void write(Key key, Value value) { ... }
}
```

- Si `read(Key key)` devuelve `null`, no hay ningun valor asociado con `key` en la memoria
- La implementación no es realista en términos de eficiencia, pero es funcionalmente correcta

Implementación de la caché

Política de reemplazo LRU

- Cuando la caché está llena y queremos guardar otro valor más, ¿qué elemento quitamos de la caché?
- Sería válido eliminar cualquiera.
- Accesos a elementos tienden a agruparse (localidad de referencia)
  - P.e., índices en un bucle
- Si se intenta a recuperar el valor asociado a una clave que no está en la caché (por ejemplo, con `cache.get("3727347Z")`):
  - Se accede al valor en M usando `storage.read("3727347Z")`
  - Se guarda el par ⟨clave,valor⟩ en la caché.

Política LRU ("Least Recently Used")

Eliminar de la caché el dato que lleve maś tiempo sin ser accedido

Datos en caché (C) y memoria principal (M)

- Cualquier acceso (lecturas y escrituras) a un dato necesita traer ese dato a C. No se lee/escribe nunca directamente de/en M.
- Mientras un dato está en C, el acceso al mismo se realiza sólo en C.
- El valor de los datos en C puede ser diferente al de M.
- Si un dato en C ha sido modificado tras traerlo de M (está *sucio*), debe volcarse de nuevo en M cuando se desaloja de C.

Implementación de la caché

Detalles internos

- La clase Caché contiene los siguientes atributos:

```
// Para acceder a la memoria M
private Storage<Key,Value> mainMemory;

// Un 'map' que asocia una clave con un ``CacheCell''
private Map<Key,CacheCell<Key,Value>> cacheContents;

// Una PositionList que guarda las claves en orden de
// uso -- la clave mas recientemente usado sera el keyListLRU.first()
private PositionList<Key> keyListLRU;
```

- Un `CacheCell` contiene:

```
public class CacheCell<Key,Value> {
  private Value value;          // El valor asociado con la clave
  private boolean dirty;        // Necesita copiar al mainMemory?

  private Position<Key> pos;    // La posicion del keyListLRU
                                // donde esta la clave

  // y getters y setters
}
```

Objeto *Cache*

Objeto *Cache*

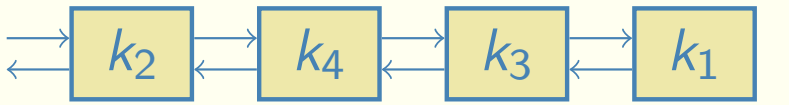
cacheContents

k_1	$\langle v_1, d_1, p_1 \rangle$
k_2	$\langle v_2, d_2, p_2 \rangle$
k_3	$\langle v_3, d_3, p_3 \rangle$
?	$\langle ?, ?, ? \rangle$
k_4	$\langle v_4, d_4, p_4 \rangle$

Objeto *Cache*

cacheContents

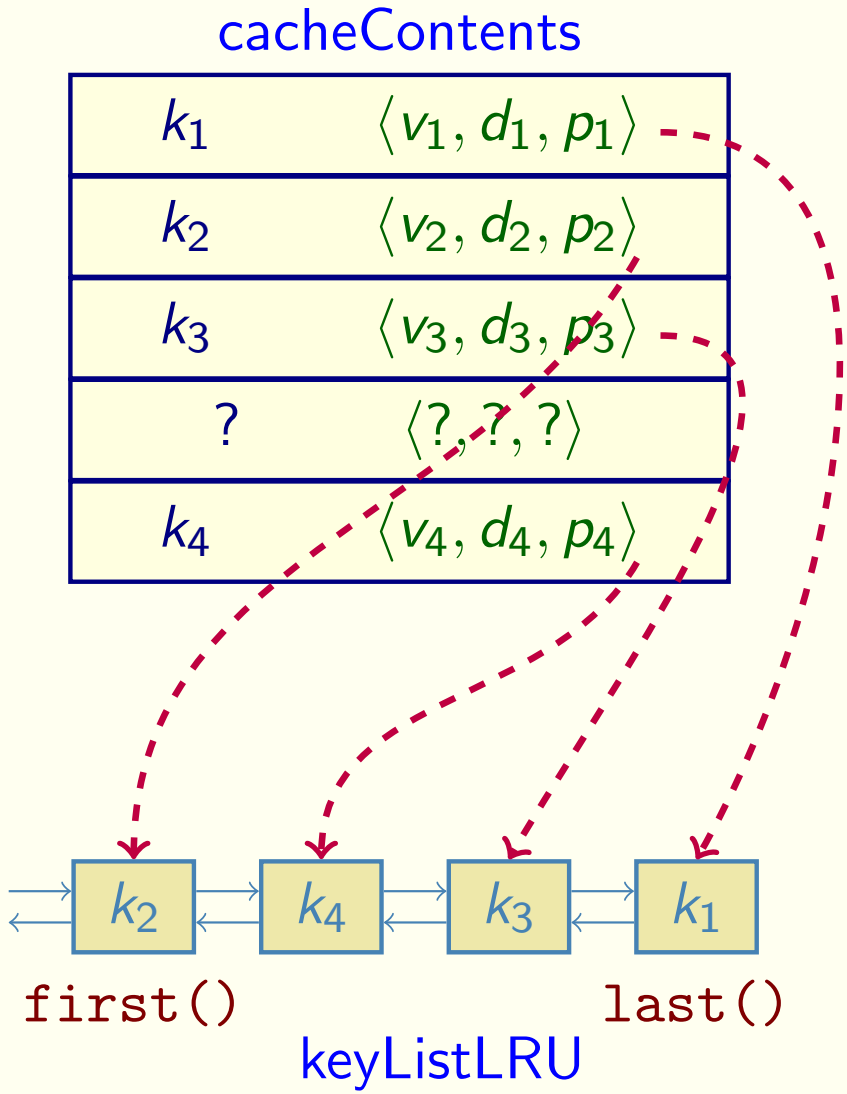
k_1	$\langle v_1, d_1, p_1 \rangle$
k_2	$\langle v_2, d_2, p_2 \rangle$
k_3	$\langle v_3, d_3, p_3 \rangle$
?	$\langle ?, ?, ? \rangle$
k_4	$\langle v_4, d_4, p_4 \rangle$



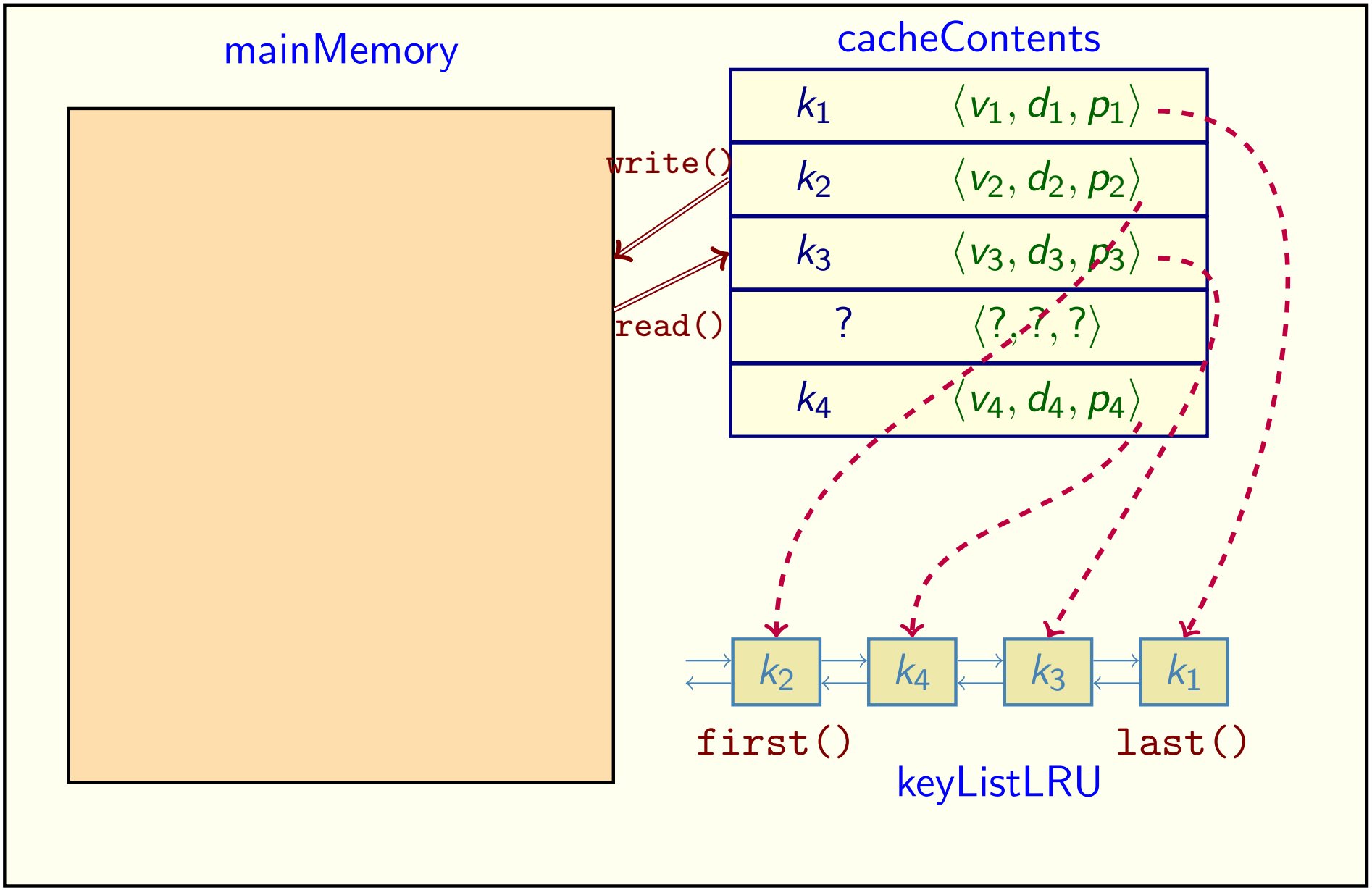
`first()` `last()`

keyListLRU

Objeto *Cache*



Objeto *Cache*



Invariantes

- Una clave está en `CacheContents` sii está en `keyListLRU`
  - Añadir y eliminar deben suceder en ambos objetos
- `keyListLRU` tiene una noción de orden:
  - keyListLRU.first(): clave más recientemente accedida.
  - keyListLRU.last(): clave accedida hace más tiempo.
- En la *CacheCell* de la clave k_i, el campo p_i apunta al nodo de la lista que contiene k_i.
- Los datos *dirty* deben volcarse a memoria principal cuando se evacúan.
- Los datos que se modifican (en la caché) deben marcarse como *dirty*.

Ejemplo

```
// <k,(v,pk,d)> representa que el cacheContents contiene una entry con clave k,
// tiene como valor un CacheCell con valor 'v', 'pk' es la Position
// de keyListLRU que contiene la clave 'k' y 'd' indica el valor de dirty.
// Abajo c.cache = c.cacheContents

M = [<k1,v1>, <k2,v2>, <k3,v3>, <k4,v4>, <k5,v5>]
Cache c = new Cache(3,M);

c.get(k2);  // c.cache = [<k2,(v2,pk2,F)>]
            // c.keyListLRU = [k2]
            // return v2
c.get(k1);  // c.cache = [<k2,(v2,pk2,F)>, <k1,(v1,pk1,F)>]
            // c.keyListLRU = [k1,k2]
            // return v1
c.get(k4);  // c.cache = [<k2,(v2,pk2,F)>, <k1,(v1,pk1,F)>, <k4,(v4,pk4,F)>]
            // c.keyListLRU = [k4,k1,k2]
            // return v4
c.get(k1);  // c.cache = [<k2,(v2,pk2,F)>, <k1,(v1,pk1,F)>, <k4,(v4,pk4,F)>]
            // c.keyListLRU = [k1,k4,k2]
            // return v1
c.get(k5);  // c.cache = [<k1,(v1,pk1,F)>, <k4,(v4,pk4,F)>, <k5,(v5,pk5,F)>]
            // c.keyListLRU = [k5,k1,k4]
            // return v5
```

Ejemplo

```
M = [<k1, v1>, <k2, v2>, <k3, v3>, <k4, v4>, <k5,v5>]
// Abajo c.cache = c.cacheContents
// c.cache = [<k1,(v1,pk1,F)>, <k4,(v4,pk4,F)>, <k5,(v5,pk5,F)>]
// c.keyListLRU = [k5,k1,k4]

c.put(k3,v8);   // c.cache = [<k1,(v1,pk1,F)>, <k3,(v8,pk3,T)>, <k5,(v5,pk5,F)>]
                // c.keyListLRU = [k3,k5,k1]
                // k3 esta dirty!!

c.get(k3);      // c.cache = [<k1,(v1,pk1,F)>, <k3,(v8,pk3,T)>, <k5,(v5,pk5,F)>]
                // c.keyListLRU = [k3,k5,k1]
                // return v8

c.put(k2,v7);   // c.cache = [<k2,(v7,pk2,T)>, <k3,(v8,pk3,T)>, <k5,(v5,pk5,F)>]
                // c.keyListLRU = [k2,k3,k5]
                // k3 y k2 estan dirty!!

c.get(k4);      // c.cache = [<k2,(v7,pk2,T)>, <k3,(v8,pk3,T)>, <k4,(v4,pk4,F)>]
                // c.keyListLRU = [k4,k2,k3]

c.get(k1);      // c.cache = [<k2,(v7,pk2,T)>, <k1,(v1,pk1,F)>, <k4,(v4,pk4,F)>]
                // c.keyListLRU = [k1,k4,k2]
                // Quitamos <k3,(v8,pk3,T)> y como estaba sucio
                // actualizamos M llevando v8 a la clave k3
                // M = [<k1, v1>, <k2, v2>, <k3, v8>, <k4, v4>, <k5,v5>]
```

Notas

- Se puede asumir que el tamaño de la memoria caché es ≥ 2
- El proyecto debe compilar sin errores y debe cumplirse la especificación de los métodos a completar
- No se debe cambiar los nombres de los atributos de la clase `Cache`
- Debe ejecutar `TesterLab3` correctamente y sin mensajes de error
  - Nota: una ejecución sin mensajes de error no significa que el método sea correcto (es decir, que funcione bien para cada posible entrada)
- Todos los ejercicios se comprueban manualmente antes de dar la nota final

Resolución de dudas: recomendaciones

- Intentad hacer preguntas lo más concretas posibles
- Si hay errores en vuestro código, intentad primero descubrir la causa:
  - Depurar código es parte del aprendizaje.
  - Podéis consultar estrategias para depurar. En muchos entornos funciona imprimir los valores de las variables y compararlos con lo esperado.
  - También os resultará útil aprender a usar el "debugger" de Eclipse.
- Eclipse permite ejecutar sólo una prueba que falle. Eso permite concentrarse en casos aislados.
- El *Tester* da mucha información. Usadla para comprender qué ha pasado — es necesario para poder corregir el código.
- Si necesitáis una consulta:
  - Enviad una descripción completa del error. Lo más útil es copiar y pegar la traza del *Tester*.
  - Enviad todo vuestro código. A veces los errores se manifiestan en un sitio diferente a dónde se cometen

Notas Generales

- El proyecto debe compilar sin errores y debe cumplirse la especificación de los métodos a completar
- Debe pasar todos los test `TesterLab3` correctamente sin mensajes de error
- **Nota:** una ejecución sin mensajes de error y que pase todas las pruebas **no** significa que la implementación sea correcta (es decir, que funcione bien para cada posible entrada)
- Todos los ejercicios se corrigen manualmente antes de dar la nota final

Evaluación

¡Seguir estos consejos os permitirá conseguir mejores resultados!

- Corrección
- Ausencia de código repetido con la misma funcionalidad (podéis usar métodos auxiliares para evitarlo)
- Concisión y claridad del código
- Legibilidad, incluida selección de nombres descriptivos para variables y métodos
- El código debe estar correctamente indentado y con comentarios *útiles* cuando lo veáis necesario
- Eficiencia:
  - Se valorará la complejidad computacional del código
  - Se valorará no iterar innecesariamente en los recorridos de las estructuras de datos