UNIVERSIDAD DE MÁLAGA
ESCUELA TÉCNICA SUPERIOR DE
INGENIERÍA DE TELECOMUNICACIÓN

PROYECTO FIN DE CARRERA:

EVALUACIÓN DE POLÍTICAS DE REEMPLAZO EN CACHES WEB

INGENIERÍA DE TELECOMUNICACIÓN

Málaga, 2007 **Javier Sanz Bustamante**

UNIVERSIDAD DE MÁLAGA
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Titulación: Ingeniería de Telecomunicación

Reunido el tribunal examinador en el día de la fecha, constituido por:

D. _____

D. _____

D. _____

para juzgar el Proyecto Fin de Carrera titulado:

EVALUACIÓN DE POLÍTICAS DE REEMPLAZO EN CACHES WEB

del alumno D. Javier Sanz Bustamante

dirigido por D. Francisco Javier González Cañete

ACORDÓ POR: _____ OTORGAR LA CALIFICACIÓN DE

_____

y, para que conste, se extiende firmada por los componentes del Tribunal, la presente diligencia.

Málaga, a ____de_____de Año

El Presidente: El Vocal: El Secretario:

Fdo. _____ Fdo. _____ Fdo. _____

UNIVERSIDAD DE MÁLAGA
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

EVALUACIÓN DE POLÍTICAS DE REEMPLAZO EN CACHES WEB

REALIZADO POR:

Javier Sanz Bustamante

DIRIGIDO POR:

Francisco Javier González Cañete

DEPARTAMENTO DE: Tecnología Electrónica.

TITULACIÓN: Ingeniería de Telecomunicación

Palabras claves: Políticas de reemplazo, *Web Proxy Cache*, simulador, evaluación de rendimiento.

RESUMEN:

En el presente proyecto fin de carrera se ha realizado un estudio de catorce políticas de reemplazo para cachés Web. Estas políticas de reemplazo pueden agrupase en tres tipos: políticas clásicas (LRU, LFU,…), políticas específicas para la Web (GDS, GDSF, …) y políticas aleatorias (RAND, HARM,…). Se ha estudiado el rendimiento de estas políticas de reemplazo para lo que se ha implementado un simulador que calcula el hit ratio y byte hit ratio de cada política de reemplazo usando una muestra de entrada. Por último se ha realizado un estudio comparativo entre todas las políticas de reemplazo a evaluación, deduciendo las ventajas y desventajas, así como sus escenarios Web más convenientes.

Málaga, Marzo de 2007

"Sé que estáis ahí, percibo vuestra presencia. Sé que tenéis miedo, nos teméis a nosotros, teméis el cambio. Yo no conozco el futuro, no he venido para deciros como acabará todo esto, al contrario, he venido a deciros como va a comenzar. Voy a colgar el teléfono y luego voy a enseñarles a todos lo que vosotros no queréis que vean, les enseñaré un mundo sin vosotros, un mundo sin reglas y sin controles, sin límites ni fronteras, un mundo donde cualquier cosa sea posible….lo que hagamos después es una decisión que dejo en vuestras manos."

INDICE

INDICE DE FIGURAS

INDICE DE CÓDIGO

CAPÍTULO 1: Introducción

En este apartado se pretende realizar una breve descripción del trabajo seguido tras la realización del proyecto, dando a conocer cuál es la naturaleza de su necesidad y los objetivos que se han pretendido alcanzar con el mismo; así como un pequeño resumen acerca de cuál será la estructura de la propia memoria y sus principales apartados.

Con todo, lo que se ha buscado realizar, es una calmada evaluación de diversas políticas de reemplazo en *cache Web*, comparando los resultados arrojados por cada una de ellas, de manera que se pueda realizar un estudio objetivo de las ventajas y desventajas proporcionadas por cada política; así como los escenarios más convenientes para cada una de las mismas.

Pero, comenzando por el principio, ¿qué es una cache *Web*? ¿En qué consisten las diversas políticas de reemplazo? A continuación se arrojará algo de luz ante tales incógnitas.

1.1. CACHE WEB

En los últimos años se ha producido un crecimiento desmesurado de la popularidad de la *World Wide Web* (WWW). Esto ha redundado en un crecimiento proporcional del tráfico en la red. Un aumento del tráfico de documentos *Web*, que provoca que gran parte del ancho de banda disponible en la red se 'desperdicie' en un continuo movimiento de estos documentos.

El principio de funcionamiento de una *cache Web*, se basa en considerar que la gran mayoría de estos documentos tienen una naturaleza estática, es decir, no suelen modificarse en su contenido a menudo. De tal manera, podríamos situar un sistema intermedio entre el servidor del documento y el cliente que realiza la petición del mismo, de forma que repetidas peticiones del mismo documento, fueran servidas por este sistema intermedio, sin colapsar el tráfico *Web* hasta el servidor.

Éste es, en líneas generales, el funcionamiento sintetizado de una cache *Web*. La cache, se situará entre el servidor de documentos *Web* y el cliente, cuando el cliente haga una petición al servidor, éste le servirá el documento solicitado; pero en este proceso la *cache Web* almacenará una copia de este documento, de forma que futuras

peticiones por parte del cliente ya no serán servidas por el propio servidor, sino por la *cache Web*.

En la Figura 1.1, se puede observar de forma gráfica el esquema representativo de un escenario habitual cliente-servidor, en el que se incorpora el uso de un sistema de *cache Web*.

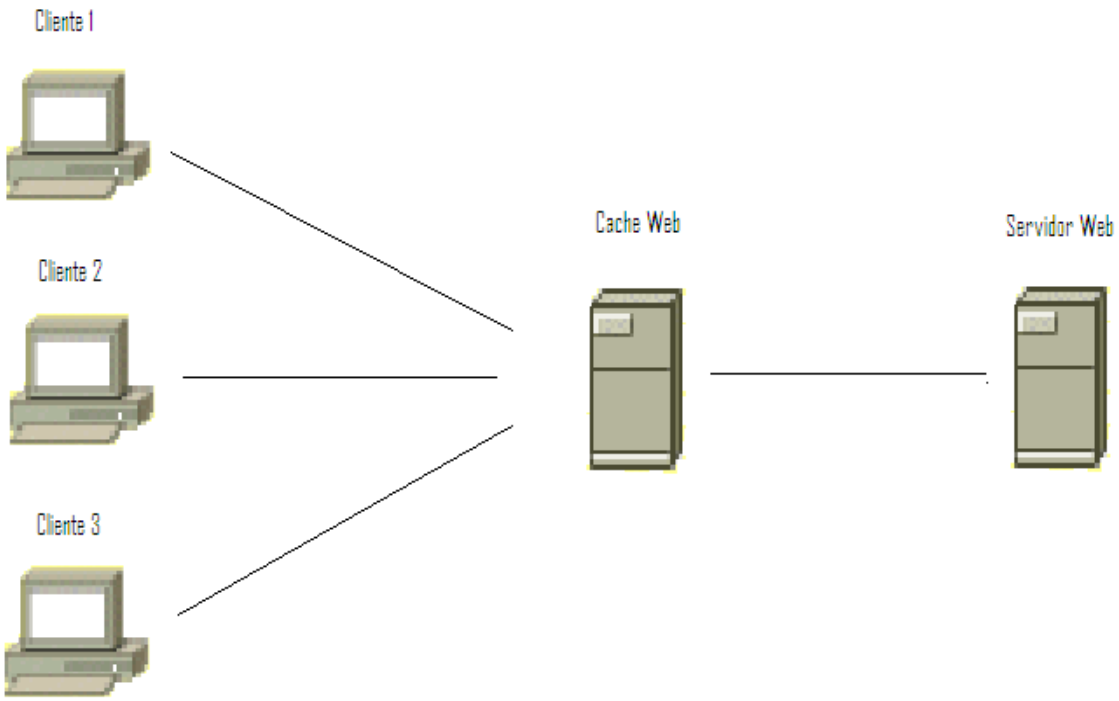


Figura 1.1. Escenario de una cache Web

Las ventajas se observan de manera muy intuitiva, pero ¿qué desventajas arroja este sistema? A continuación pormenorizaremos las ventajas y posibles desventajas del empleo de una *cache Web*.

1.1.1. VENTAJAS

Si situamos una *cache* entre uno o más servidores *Web*, y uno o más clientes, guardando copias de las respuestas, conseguiremos dos efectos [6]:

- *Reducir la latencia*: ya que la respuesta se satisface desde la *cache*, que se encuentra más cercana al cliente que el propio servidor, reduciéndose el tiempo de respuesta.
- *Reducir el tráfico*: ya que los documentos se vuelven a usar, reduciendo el ancho de banda usado por el cliente.

Las ventajas parecen, a priori, altamente favorecedoras. Conseguimos respuestas más rápidas ante peticiones de documentos *Web*, y a la vez que estas peticiones no redunden en una saturación del ancho de banda empleado por el cliente, o del propio servidor.

1.1.2. DESVENTAJAS

Las desventajas propias de este sistema son menos intuitivas, pero no por ello inexistentes.

Los principales problemas planteados por una *cache Web*, hacen referencia a la pérdida de control por parte de los *Websmaster* de sus documentos, es decir, la *cache Web* hace que sea más dificultoso el control de los usuarios que hacen peticiones sobre estos documentos *Web*, hacen más compleja la generación de unas estadísticas fiables para sus espacios en Internet.

Otro de los problemas hace referencia a la premisa que hemos considerado de que los documentos tienen una naturaleza aparentemente estática, es decir, si éstos se modificasen también debieran hacerlo en la *cache*. Todo esto fuerza el empleo de cabeceras para asegurar el refresco de los documentos almacenados en *cache* de manera conveniente, complicando el sistema global.

1.2 POLÍTICAS DE REEMPLAZO

Pero, como se comentó anteriormente, con este proyecto hemos pretendido la realización de un simulador de *cache Web*. Herramienta que nos permitiría abordar el objetivo último del mismo, evaluar diversas políticas de reemplazo, extrayendo las conclusiones pertinentes de este análisis

¿En qué consiste una política de reemplazo? Se ha pretendido abordar la naturaleza propia de una *cache Web*, un sistema que almacena documentos *Web*, para así facilitar el acceso a éstos por parte del usuario final. Pero como todo sistema real, es finito, es decir, tendrá un tamaño determinado. La principal consecuencia que se desprende de tal circunstancia es que la *cache* acabará llenándose, llegado tal caso, ¿cómo actuará? La respuesta parece evidente, se hará necesario la eliminación de algún documento de los que conserva en memoria, lo que no parece tan evidente es cuál.

Esa es la principal característica de cada política de reemplazo, llegado el momento, qué elemento eliminará (otra característica diferenciadora de una política u otra, es la posición en la que insertará los documentos nuevos). De tal manera, que ante

un mismo conjunto de documentos *Web* a almacenar (lo que a partir de ahora denominaremos muestra de entrada) cada política manifestará un comportamiento diametralmente opuesto, y se obtendrá un conjunto de documentos almacenados finalmente, distinto.

Con todo esto, el estudio se realizará sobre las siguientes políticas de reemplazo:

- *LRU (Least Recently Used)*
- *LFU (Least Frequently Used)*
- *LFU-DA (Least Frequently Used with dynamic Aging)*
- *GDS (Greedy Dual Size)*
- *GDSF (Greedy Dual Size Frequency)*
- *GD**
- *RAND (Random)*
- *HARM (Harmonic)*
- *LRU-C*
- *LRU-S*
- *RRGVF (Randomized replacement with general value functions)*

Cada una de las cuáles, se analizará al detalle en sucesivos capítulos.

1.3 MEMORIA DEL PROYECTO

Para la presentación de la siguiente memoria, se ha realizado un trabajo estructurado en capítulos, cada uno de los cuales hace referencia a algún aspecto que se ha considerado importante para la comprensión de las diversas políticas de reemplazo que en la memoria se tratan. El primero de los cuales es el propio de la introducción. A continuación se realizará un breve resumen del contenido de cada uno de los capítulos restantes.

- **Capítulo 2.Políticas de Reemplazo:** En esta sección se realizará una completa disección de cada una de las diversas políticas de reemplazo, se dará a conocer su comportamiento, así como las características fundamentales de cada una de ellas.
- **Capítulo 3.El Simulador:** En este apartado se propondrá un completo estudio del simulador implementado. Se elaborará un pequeño manual de usuario en el que se dará a conocer el funcionamiento del mismo. De igual

manera se detallarán algunos de los elementos y mecanismos relativos a la programación orientada a objetos, empleados para su creación.

- **Capítulo 4.Plan de Pruebas:** Lo que se pretenderá en esta ocasión es verificar el correcto funcionamiento del simulador, es decir, que los resultados por él arrojados coincidan con los obtenidos de manera teórica. Para ello se creará una muestra de entrada y se emulará de manera teórica el comportamiento de cada una de las políticas.
- **Capítulo 5. Comparativa de las Políticas de Reemplazo:** En esta sección se pretende realizar una comparación detallada de todas las políticas de reemplazo. Para ello se recreará el comportamiento de cada política en gráficas, para diversos tamaños de *cache*. Una vez con todas las gráficas preparadas se realizará una intensa comparación entre ellas, de la cual se deducirán las características principales de cada política de reemplazo.
- **Capítulo 6. Conclusiones:** Se dará por finalizada esta memoria, exponiendo de manera somera y concisa las conclusiones a las que se han llegado tras la realización del proyecto, así como un pequeño resumen del mismo. De igual manera se dará alguna pincelada, describiendo cuales serían las líneas futuras de investigación.

CAPÍTULO 2: Políticas de Reemplazo

El objetivo de este capítulo es realizar una somera descripción de cada una de las políticas de reemplazo que se han implementado en el simulador. Para ello, se seguirá siempre un mismo procedimiento.

Se seleccionará una política, comenzando por la *LRU*, y en primer lugar se realizará una pequeña introducción a la misma, y el escenario en que se encuadra.

A continuación, se entrará de lleno en la descripción de la propia política, cuál es el principio de su funcionamiento y características principales; ésta será realmente la parte más interesante del estudio, ya que nos permitirá comprender con detalle cómo funciona cada política, y con ello se podrá llegar a entender claramente futuras conclusiones que se obtengan tras la realización de diversas simulaciones.

El siguiente paso, será realizar una pequeña simulación, a fin de poder observar de manera gráfica y reposada cómo el algoritmo interactúa con los documentos, a la hora de insertar o eliminar, para así afianzar el entendimiento de su rutina.

Por último se constatará un pequeño resumen, tan sólo unas líneas, de la política en cuestión, sus aspectos generales y características más destacadas.

Con todo, se pretende que con este capítulo se llegue a un entendimiento global de las políticas de reemplazo que se han tratado, como parte del objetivo último de esta memoria, alcanzar un cierto conocimiento acerca del tema que nos ocupa, la *cache Web*.

Se ha de hacer mención, también, de que a lo largo de la presente memoria, se hablará indistintamente de objetos o documentos *Web*, haciendo referencia a aquellos que serán almacenados en un sistema de *cache Web*.

2.1 IMPORTANCIA DE LAS POLÍTICAS DE REEMPLAZO

Una política de reemplazo conveniente, es un factor que se consideró absolutamente determinante en los primeros pasos de la *cache Web*.

Hoy en día, sin embargo, su importancia parece haber decaído, de forma que el tema del reemplazo en *cache* es a menudo considerado como poco importante. Esta consideración se basa en los siguientes argumentos:

- El coste asociado a la obtención de un sistema de almacenamiento masivo decrece constantemente, de tal manera, es posible adquirir sistemas de

- almacenamiento capaces de albergar la mayor parte de los documentos *cacheables*.
- Los actuales escenarios cliente-servidor fuerzan que el tráfico Web es cada vez menos susceptible de ser almacenados en sistemas de *cache Web*.
- Existen algoritmos suficientemente adecuados como para satisfacer la mayoría de las situaciones en que los sistemas de *cache Web* son empleados.

A continuación, comentaremos con más detalle cada uno de los argumentos esgrimidos.

2.1.1. TAMAÑO DE LA CACHE

Éste es el mayor argumento esgrimido a la hora de poner en duda la radical importancia que la elección de una política de reemplazo u otra, tienen a la hora de diseñar un sistema de almacenamiento.

Se basa en una ley empírica ya formulada en numerosos artículos, es aquella que dice que se consiguen sistemas de almacenamiento que doblan la capacidad cada 18 meses [1]. De tal forma, la capacidad de sistemas de *cache* crece exponencialmente. No resultando el empleo de una política u otra un factor limitador para la misma.

Ante tal premisa, a menudo un básico algoritmo de reemplazo *LRU* consigue los resultados esperados por un sistema de *cache Web.*

2.1.2. REDUCCIÓN DEL TRÁFICO 'CACHEABLE'

Como se comentó en el capítulo 1, una *cache Web* es un sistema que se ubicará entre un cliente que solicita documentos *Web* y el servidor *Web* de los mismos. Diremos que una petición realizada por un cliente es *cacheable* si es satisfecha por la *cache Web*, es decir, si ésta proporciona al cliente el documento solicitado.

Igualmente se hizo referencia a la naturaleza estática de estos documentos *cacheables*, pero, desgraciadamente, actualmente el 40% de los documentos *Web* son generados dinámicamente, es decir, varían constantemente personalizándose para cada cliente o para cada situación [1], razón por la cual se hace pertinente conseguir políticas cada vez más eficientes, que se adapten a esta constante variación en los documentos, consiguiendo que la *cache* se actualice constantemente, sin mermar el rendimiento de la misma.

Todo esto, presenta un importante problema, no sólo para el diseño de políticas de reemplazo en particular, sino para la propia *cache Web* en general.

2.1.3. ALGORITMOS 'ADECUADOS'

En los primeros días de los sistemas de *cache Web* se emplearon políticas de reemplazo muy simples. Con el paso del tiempo, los estudios se centraron en la búsqueda de estrategias de reemplazo más sofisticadas y complejas, capaces de alcanzar los objetivos propuestos.

En este ámbito, se han desarrollado un grupo de algoritmos capaces de proporcionar excelentes resultados ante diversos escenarios, es decir, distintos tamaños de almacenamiento o tráfico de documentos. Éstos son los que se han denominado algoritmos 'adecuados'.

Pero, ¿cómo estableceremos la comparación entre los resultados arrojados por una política u otra?, para ello se han desarrollado los siguientes términos [2]:

- ***Hit – Ratio (HR):*** Éste será el parámetro más usado para comparar políticas. Pretende establecer una relación entre los objetos solicitados a la *cache* y resultan un acierto, y el número total de objetos que pasan en algún momento por la misma. Así estrategias que favorezcan el almacenamiento de objetos de pequeño tamaño obtendrán mayores tasas de *HR* (el número de objetos almacenados así será mayor, y por tanto, mayor la probabilidad de obtener un acierto).
- ***Byte – Hit – Ratio (BHR):*** Éste será un caso opuesto al anterior. Políticas que favorezcan el almacenamiento de objetos de pequeño tamaño conseguirán una baja tasa de *BHR.* El *BHR* no será más que la relación entre los bytes acertados y el número total de bytes llegados a la cache. Por lo tanto, aciertos de objetos de gran tamaño son los que consiguen una alta *BHR.*

Estos parámetros son los que emplearemos en futuros capítulos para realizar las pertinentes comparaciones entre las políticas de reemplazo implementadas.

2.2 LRU [2]

Como se dijo en el *capítulo 1*, *LRU* es el acrónimo inglés de *Least Recently Used*, es decir, '*el menos recientemente usado'. LRU* es una política encuadrada en el marco de las políticas 'basadas en el uso reciente'. Éstas políticas se basan en analizar el uso reciente o no de los objetos almacenados, como factor principal, de tal forma que la

mayor parte de los algoritmos que forman parte de este grupo, son en mayor o menor medida una extensión de la estrategia *LRU*.

Estas estrategias se basan en el principio de localidad para justificar su comportamiento. Existen dos tipos diferenciados de localidad, la localidad espacial y temporal, que describiremos a continuación [1]:

- **Localidad Espacial:** Hace referencia al principio que realiza la siguiente suposición, referencias a determinadas elementos de la cache implican necesariamente que exista una mayor probabilidad de realizar una llamada a un determinado grupo de elementos en la cache. De tal manera, la llamada a ciertos objetos podría servir como un predictor de futuras referencias.
- **Localidad Temporal:** Se basa en el hecho de que recientes aciertos de elementos de la cache son mucho más probables referenciarlos de nuevo en el futuro que el resto.

El algoritmo *LRU* se basa el principio de localidad temporal, es decir, pretende mantener siempre en memoria los objetos que más recientemente se han referenciado, o dicho de otra manera, los objetos que siguiendo el principio de localidad temporal, más probabilidad tienen de volverse a referenciar.

Como se ha dicho, este algoritmo forma parte de aquellos basado en el 'uso reciente', pero desde luego no es el único. Algunos de los algoritmos que también forman parte de este grupo son *LRU-Treshold, LRU-Min, Size, Value Aging EXP1, PSS, LRU-LSC, Partitioned Caching, Pitkow/Reckers strategy o HLRU [1]*.

Las ventajas de las estrategias basadas en el 'uso reciente' son las siguientes:

- Ellas consideran la localidad temporal como un factor principal. Dado que las peticiones *Web* muestran cierto orden por localidad temporal, parece un sistema ventajoso.
- Son estrategias fáciles de implementar y rápidas. Muchas de tales estrategias emplean una lista *LRU*. Nuevos objetos son insertados en la cabeza de la *cache*. Un acierto provoca que el objeto en cuestión sea eliminado de su posición actual y se reinserte en la cabeza de la *cache*. Los objetos más susceptibles de ser reemplazados se aglutinarán en torno al final de la cache. De tal manera, la inserción y eliminado se convierten en operaciones con una baja complejidad. Igualmente, la búsqueda puede ser soportada por técnicas de búsqueda *hashing*.

A continuación se exponen las desventajas características de este grupo de políticas de reemplazo:

- No consideran información alguna de la frecuencia con la que se referencian los elementos de la *cache*. Lo cual será una seria desventaja a tener en cuenta, ya que éste es un importante indicador en escenarios de tráfico *Web*.
- Otra de las desventajas atañe en mayor medida a las estrategias derivadas de la *LRU*, ya que no combinan este almacenamiento de objetos recientemente llamados y el tamaño de manera balanceada. Los documentos *Web* suelen ser de diferente tamaño, por lo tanto el tamaño es un factor a tener en cuenta en cada reemplazo.

Dicho todo esto, se procede a continuación a la presentación del algoritmo en cuestión.

Como su propio nombre indica, la estrategia *LRU* reemplaza el documento que menos recientemente haya sido referenciado., y éste será el principio fundamental de su funcionamiento. Pero, es conveniente desarrollar más ampliamente éste método, para ello se describirá como actúa esta política ante diversas circunstancias (inicialmente la *cache* esta vacía):

1) **Llega un elemento nuevo** → Se analiza si el nuevo elemento cabe en la cache. De ser así, se almacenará en la cabeza de la *cache* (es decir, en la posición superior de la misma). El resto de elementos almacenados serán desplazados una posición.
2) **Llega un elemento nuevo que no cabe** → Se eliminan objetos de la *cache*, comenzando desde el último, hasta que el espacio liberado sea el suficiente como para insertar el nuevo elemento (en la cabeza de la *cache*).
3) **Se produce un acierto en la *cache*** → O lo que es lo mismo, el documento solicitado está en la *cache*, y por tanto será servido por la misma. En tal caso se realizará una búsqueda de este objeto almacenado, y una vez localizado (se consigue su índice) se reubica de nuevo en la cabeza del sistema (se inserta como si se tratase de un elemento nuevo).

Por supuesto en caso de llegar un elemento cuyo tamaño excediese el propio de la *cache*, éste sería directamente descartado para ser almacenado (esto será común para todas las políticas de reemplazo).

Se puede comprobar cómo efectivamente los objetos más recientes, ya sea porque son nuevos, ya sea porque son elementos acertados, se ubican siempre en la zona superior de la cache, en torno a su cabeza. Por el contrario, los elementos menos recientemente referenciados permanecerán en el final del sistema, siendo candidatos óptimos para su eliminación.

A continuación se realizará una pequeña demostración:

Cache Web

	(1) – Inicialmente la *cache* estará vacía
OBJETO 1	***(2) - OBJETO 1*** llega a la *cache.* Se sitúa en la cabeza
OBJETO 2 ***OBJETO 1***	***(3) – OBJETO 2*** llega a la cache. Se sitúa en la cabeza
OBJETO 3 ***OBJETO 2*** ***OBJETO 1***	***(4) – OBJETO 3*** llega a la *cache*. Se sitúa en la cabeza
OBJETO 4 ***OBJETO 3*** ***OBJETO 2***	***(5) – OBJETO 4*** llega a la *cache*. No cabe. Se elimina el último objeto de la misma (OBJETO1). Ya cabe. Se inserta en la cabeza.
OBJETO 3 ***OBJETO 4*** ***OBJETO 2***	***(6) – OBJETO 3*** llega a la caché. Éste elemento ya está almacenado, es un acierto. Se reubica en la cabeza del sistema.

En resumen, esta política tradicional es, en la práctica, la más frecuentemente usada, y ofrece un gran comportamiento para *cache* CPU y sistemas de memoria virtual. Su rendimiento no es tan conveniente para sistemas de *cache Web,* debido a que la localidad temporal del tráfico *Web* a menudo presenta patrones muy diversos.

2.3 LFU [3]

Se ha de examinar en primer lugar el significado del acrónimo inglés, *Least Frequently Used,* o traducido al español 'menos frecuentemente usado'. *LFU* es una política de reemplazo agrupada dentro de las políticas 'basadas en la frecuencia'.

Estas estrategias usan la frecuencia como característica principal, es decir, el número de veces que se ha realizado una llamada a un documento determinado, y son, como ocurría en las políticas basadas en el uso reciente con la política *LRU*, en mayor o menor medida una extensión de la propia *LFU.* Se basan en el hecho de que los diversos documentos *Web* poseen una distinta popularidad, popularidad que se traduce en un mayor o menor indicador de frecuencia. Estas políticas emplean dicho indicador para tomar futuras decisiones.

Este grupo de políticas pueden implementarse de dos formas distintas [1]:

- ***Perfect LFU:*** Se crea un contador que se incrementa con cada llamada a un objeto, este factor permanecerá inalterable aún cuando el elemento haya sido eliminado de la *cache,* de esta manera, este factor mantiene una cuenta de todas las referencias a dicho objeto, presentes y pasadas.
- ***In-Cache LFU:*** En este caso, dicho contador se reseteará cada vez que el elemento en cuestión sea eliminado de la *cache*, de tal manera que el contador mantendrá la cuenta de las veces que se han referenciado únicamente los elementos que se encuentren en *cache*.

En el desarrollo de este proyecto, se ha optado por utilizar la variante *In-Cache LFU.*

Como se ha dicho, *LFU* forma parte de una serie de algoritmos basados en la frecuencia', pero como sucedió con *LRU* no es el único. Algunos de los cuáles son *LFU Aging, LFU-DA* (que se estudiará a continuación), *α-Aging o sw-LFU [1]*.

De igual manera, el empleo de políticas basadas en la frecuencia genera una serie de ventajas e inconvenientes. Algunas de las ventajas son:

- Tienen en cuenta la frecuencia de acceso a los elementos almacenados, es decir, objetos más veces referenciados tendrán una mayor prioridad sobre el resto, a la hora de evitar ser eliminados.

En contrapartida, también se darán una serie de desventajas, son las siguientes:

- Estas son políticas de una mayor complejidad, ya que requieren un tratamiento mas elaborado de la *cache.* En estas estrategias suele ser extendido el uso de colas de prioridad.
- Muchos de los objetos almacenados pueden llegar a tener el mismo contador de frecuencia, en tal caso sería necesario algún factor que reordenase los elementos en la lista.
- Estas estrategias son invariantes a los cambios de tráfico, es decir, es posible que documentos muy referenciados en el pasado pero totalmente denostados actualmente, permanezcan en la zona superior de la *cache,* y por lo tanto no serán eliminados. Existen técnicas de envejecimiento o A*ging,* que buscan solucionar tal circunstancia, a costa de complicar aún más el algoritmo.

Dicho todo esto, se procederá a continuación a la presentación del algoritmo en cuestión.

Como su propio nombre indica, la estrategia *LFU* reemplaza el documento que menor número de veces haya sido referenciado., y éste será el principio fundamental de su funcionamiento. A continuación se desarrollará más ampliamente éste método, para ello se describirá como actúa esta política ante diversas circunstancias:

1) **Llega un elemento nuevo** → Se analiza si el nuevo elemento cabe en la cache. De ser así, se almacenará en la misma, de manera ordenada, en función de su contador de frecuencia, lo que a partir de ahora denominaremos contador de referencia. Así elementos con un mayor contador de referencia se situarán siempre en la zona superior de la *cache*, mientras que elementos que han sido pocas veces referenciados estarán en la zona inferior de la misma y serán futuros candidatos para ser eliminados.
2) **Llega un elemento nuevo que no cabe** → Se eliminan objetos de la *cache*, comenzando desde el último, hasta que el espacio liberado sea el suficiente como para insertar el nuevo elemento (en la posición que le corresponda en función de su contador de referencia).
3) **Se produce un acierto en la *cache*** → O lo que es lo mismo, el documento solicitado está en la *cache*, y por tanto será servido por la misma. En tal caso se realizará una búsqueda de este objeto almacenado, y una vez localizado (se consigue su índice) se incrementa su contador de

referencia (ya que ha sido llamado una vez más) y se reubica de nuevo en el sistema en función de su nuevo contador de referencia.

Con todo esto, se obtendrá una *cache* ordenada en función del valor del contador de referencia de cada uno de los elementos que la componen, de tal manera que los objetos más 'populares' (más veces referenciados) quedarán en la zona superior del sistema; mientras que los elementos menos 'populares' quedarán en la zona inferior, listos para ser eliminados.

Se ha supuesto que, en caso de existir elementos con un mismo contador de referencia, éstos se ordenarán en función de cómo de recientemente fueron llamados, es decir, el objeto que más recientemente se referenció estará por encima del resto.

A continuación se realizará una pequeña demostración (entre paréntesis se muestra el valor del contador de referencia del elemento en cuestión):

Cache Web

	(1) – Inicialmente la *cache* estará vacía
OBJETO 1(1)	***(2) - OBJETO 1*** llega a la *cache.* Se sitúa en la cabeza
OBJETO 2(1) ***OBJETO 1(1)***	***(3) – OBJETO 2*** llega a la cache. Se ubica de manera ordenada en la *cache.*
OBJETO 3(1) ***OBJETO 2(1)*** ***OBJETO 1(1)***	***(4) – OBJETO 3*** llega a la *cache*. Se ubica de manera ordenada en la *cache.*
OBJETO 2(2) ***OBJETO 3(1)*** ***OBJETO 1(1)***	***(5) – OBJETO 2*** llega a la *cache*. Se localiza en el sistema. Se incrementa su contador y se reubica en la zona superior de la *cache*.
OBJETO 2(2)	

OBJETO 4(1)
OBJETO 3(1)

(6) – OBJETO 4 llega a la *cache*. No cabe. Se elimina el último objeto de la misma (OBJETO1). Ya cabe. Se inserta de manera ordenada en la *cache* en función del contador. .

OBJETO 3(2)
OBJETO 2(2)
OBJETO 4(1)

(7) – OBJETO 3 llega a la *cache*. Se localiza en el sistema. Se incrementa su contador y se reubica en la zona superior de la *cache*.

En resumen, el algoritmo *LFU* reemplaza el documento que ha sido referenciado un menor número de veces. Esta estrategia intenta mantener los documentos más populares y reemplazar los más raramente referenciados. Sin embargo algunos documentos pueden alcanzar un alto contador de referencia y nunca ser referenciados de nuevo, es lo que se suele llamar polución de *cache*, y ante la cual la estrategia *LFU* no provee ningún mecanismo de defensa.

2.4 LFU-DA [3]

Se trata de una política muy parecida a la anteriormente expuesta *LFU,* con ciertas variaciones proporcionadas por la coletilla *DA,* acrónimo inglés de *Dynamic Aging* o en español 'envejecimiento dinámico'.

Al igual que la estrategia *LFU* se engloba dentro de las políticas 'basadas en la frecuencia', es decir, son políticas que priman la popularidad de los documentos almacenados, de tal manera que los documentos menos populares son los que con una mayor probabilidad serán eliminados si fuese necesario.

Dado que las características generales de las políticas 'basadas en la frecuencia' se describieron en el apartado anterior omitiremos su repetición en esta ocasión, y nos centraremos en diseccionar el comportamiento de la propia política.

El comportamiento básico será igual al de la *LFU,* es decir, elementos nuevos se ubicarán en la cache de forma ordenada en función de su contador de referencia. Al producirse un acierto, el objeto acertado modificará su contador de referencia y se reubicará en función de éste. Por último, en caso de ser necesario liberar espacio en memoria, lo haremos eliminando elementos comenzando desde el final de la caché,

lugar donde se encontrarán los elementos con un menor contador de referencia, o lo que es lo mismo, los elementos menos populares de los que se han almacenado.

Pero entonces, ¿dónde radica la diferencia con *LFU*? Para darla a conocer, será necesario recordar antes dónde radicaba el principal inconveniente del algoritmo *LFU.* Y éste se observaba en el hecho de que existía la posibilidad de que ciertos elementos almacenados pudieran haberse referenciado en el pasado numerosas veces, de forma que su contador de referencia fuese tremendamente alto, manteniéndose el documento en la zona de privilegio de la cache (alejado de la zona de eliminación) aún cuando en la actualidad dichos documentos hayan caído claramente en desuso (deberían ser candidatos válidos para ser eliminados). Es lo que en el apartado anterior se llamó 'polución de *cache'*.

Pues bien, el objetivo básico de este *Dynamic Aging,* es evitar en la medida de lo posible el efecto de esta polución. Para ello, almacena el valor del contador de referencia del último elemento eliminado (inicialmente cero); valor con el que modificará el contador de referencia de los elementos nuevos que lleguen a la *cache* o de aquellos que resulten ser un acierto en la misma.

Pero, se ha de desarrollar con más detalle el algoritmo, para ello se describirá como actúa esta política ante diversas circunstancias:

1) **Llega un elemento nuevo** → Se analiza si el nuevo elemento cabe en la cache. De ser así, se almacenará en la misma, de manera ordenada, en función de su contador de referencia. Así elementos con un mayor contador de referencia se situarán siempre en la zona superior de la *cache*, mientras que elementos que han sido pocas veces referenciados estarán en la zona inferior de la misma y serán futuros candidatos para ser eliminados. Inicialmente elementos nuevos entrarán en la *cache* con un contador de referencia igual a uno, en el momento en que se produzca una eliminación (el factor de envejecimiento deja de ser nulo), los objetos nuevos se ubicarán en *cache* con un contador de referencia igual a este factor de referencia.
2) **Llega un elemento nuevo que no cabe** → Se eliminan objetos de la *cache*, comenzando desde el último, hasta que el espacio liberado sea el suficiente como para insertar el nuevo elemento (en la posición que le corresponda en función de su contador de referencia). Se actualiza el valor del factor de envejecimiento al del contador de referencia del último elemento que se haya eliminado. El nuevo objeto tendrá como contador de

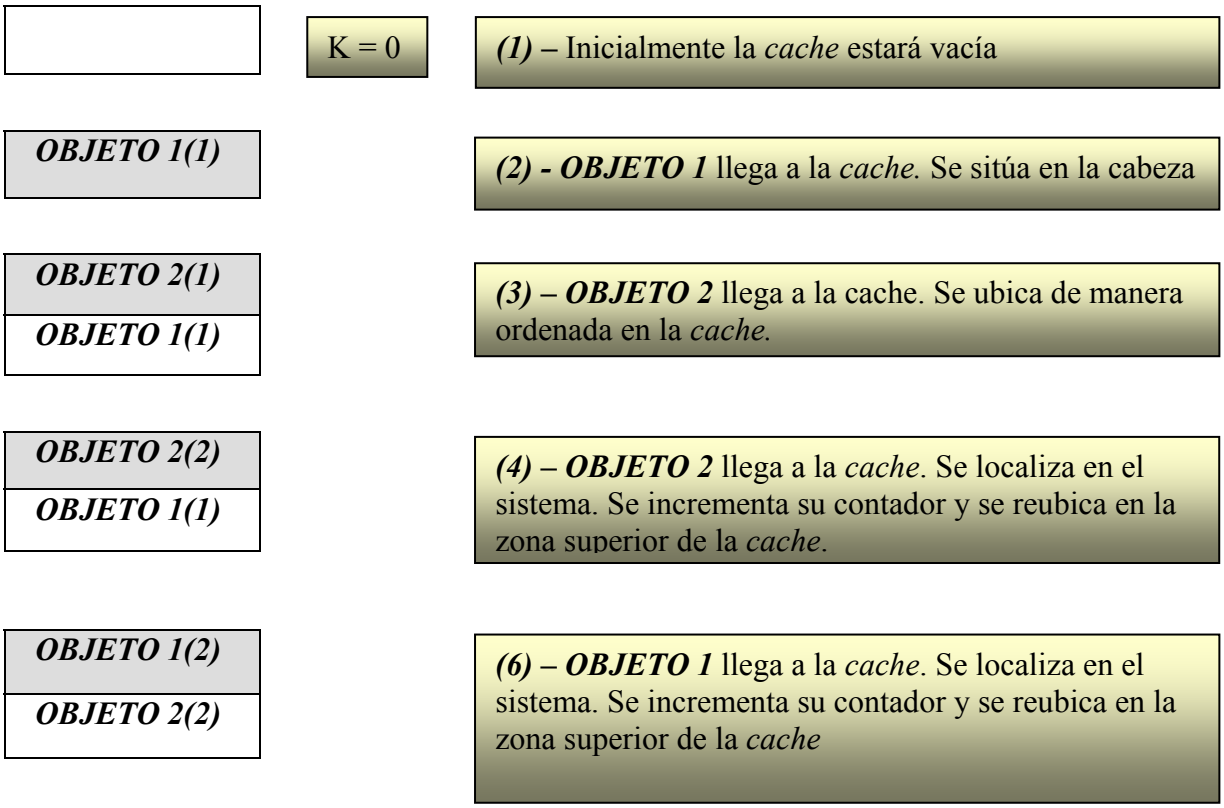
referencia el nuevo factor de envejecimiento, y se ubicará en la *cache* en función del mismo.

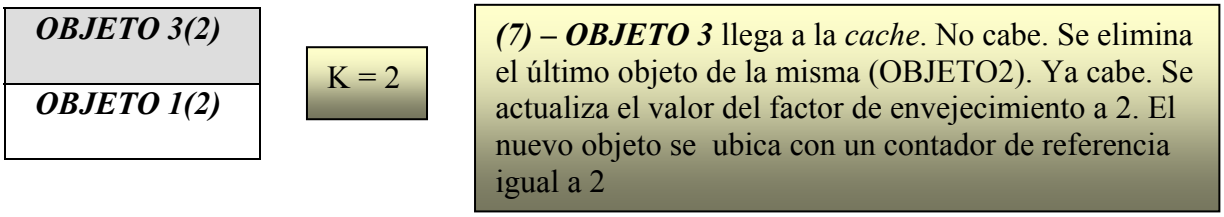
3) **Se produce un acierto en la *cache*** → O lo que es lo mismo, el documento solicitado está en la *cache*, y por tanto será servido por la misma. En tal caso se realizará una búsqueda de este objeto almacenado, y una vez localizado (se consigue su índice) se incrementa su contador de referencia sumándole el valor del factor de envejecimiento.

Con todo esto, se obtendrá una *cache* ordenada en función del valor del contador, tal y como sucedía en la política *LFU,* con la incorporación de este envejecimiento con el que se intenta solucionar el problema de la polución de *cache*.

A continuación se realizará una pequeña demostración (entre paréntesis se muestra el valor del contador de referencia del elemento en cuestión, en caso de que varios elementos tengan el mismo contador, el elemento nuevo se insertará en la posición superior). Cada vez que se actualice el valor del factor de envejecimiento (K) haremos una mención a su nuevo valor (inicialmente cero):

Cache Web





Resumiendo, el algoritmo *LFU-DA* funciona de manera semejante al *LFU,* almacenando los documentos en cache en función de su popularidad, con la salvedad de que con la introducción de este factor de envejecimiento en el algoritmo se pretende solucionar el problema de la polución de *cache*.

2.5 GDS [5]

Acrónimo inglés de *Greedy Dual Size***.** Esta estrategia, así como las siguientes (*GDSF y GD**) forman parte de lo que se denomina políticas 'basadas en función', es decir, estrategias que emplean alguna función con la que establecer la posición de cada objeto en la *cache.*

Tal y como ocurría en las estrategias *LFU* y *LFU-DA,* los elementos dentro de la cache estarán ordenados, la diferencia estriba que el factor de ordenación , en este caso, no será el contador de referencia del elemento en cuestión, sino el valor arrojado por la función para cada objeto.

Algunas otras políticas que forman parte de este grupo son (a parte de las ya citadas), *Server-assisted cache replacement, TSP (Taylor Series Prediction), Bolot/Hoschka's strategy, MIX, M-Metric, HYBRID, LRV, LUV, LR (Logistic Regressión-Model) [1]*.

Como sucedió en los apartados anteriores, este grupo de políticas presenta una serie de ventajas y desventajas. Las ventajas más relevantes son:

- Consideran un número de factores para manejar diversas situaciones de carga de trabajo.
- No asumen una combinación fija de factores o un uso fijo de estructuras de datos. Con la opción apropiada de parámetros, uno puede intentar optimizar cualquier comportamiento, es decir, con la función adecuada se puede optimizar su funcionamiento ante cualquier escenario de tráfico Web.

A continuación se resaltarán las desventajas más notables:

- Elegir la función apropiada para cada caso es tarea difícil Ésta se suele deducir de los estudios de las muestras *Web,* pero alcanzar la función idónea para cada

carga de tráfico es realmente complejo; además el tráfico *Web* es muy variable con el tiempo, por lo tanto sería necesario una configuración adaptativa de los parámetros. Adaptación que por otro lado, sin llegar a solucionar por completo los problemas de tráfico variable, añaden una gran complejidad al proceso de reemplazo.

- El empleo de información de latencia para el cálculo de parámetros introduce grandes problemas. Cálculos relativos a frecuencia o como de recientemente se han referenciado ciertos objetos pueden ser calculados fácilmente partiendo de las peticiones hechas a la *cache.* La latencia es calculada en la *cache,* pero influenciada por muchos factores, como el camino entre el servidor y la *cache.* De tal manera que fluctuaciones en el cálculo de éste parámetro pueden complicar cualquier decisión tomada.

Dicho todo esto, se procederá a continuación a la presentación del algoritmo en cuestión.

La estrategia *GDS (1)* se emplea en *caches* que mantendrán sus objetos ordenados en función de un cierto factor, que denominaremos ***clave***, elementos con un campo clave más alto se ubicarán en la zona superior del sistema, y aquellos con una menor clave se situarán en la zona inferior, y serán los objetos a eliminar en caso de necesitar liberar espacio.

Pero, ¿cómo se calculará esta clave?, para ello emplearemos función mostrada en la *Ecuación 2.1*:

$$clave = \left(\frac{\cos te}{tamaño\_objeto} \right) \qquad \textbf{Ecuación 2.1}$$

Dentro de esta familia de políticas de reemplazo, se distinguirán dos versiones, la *GDS (1)* y *GDS (p).* Cada una de las cuales hará uso de una función de coste distinta. Por un lado la *GDS(1)* que asigna una función coste igual a1(se entiende por coste, al coste asociado a que se produzca un fallo de *cache*, es decir, que haya que traer el documento directamente del servidor), por lo tanto, el factor clave quedará como sigue:

$$clave = \left(\frac{1}{tamaño\_objeto} \right) \qquad \textbf{Ecuación 2.2}$$

Así, elementos de mayor tamaño generarán claves menores, y se insertarán en la zona inferior de la *cache,* o lo que es lo mismo, esta es una política que beneficia a los objetos pequeños sobre los de mayor tamaño.

Debido a esto, cada vez que sea necesario liberar espacio, eliminaremos, los objetos de mayor tamaño, es decir, nos quedaremos con una cache con un mayor número de objetos; aunque de menor tamaño. La conclusión principal que se desprende de esto es clara, una función coste igual a 1 parece conseguir el mejor *HR,* por el contrario, el *BHR* será bajo. Con todo, lo que *GDS (1)* busca, será minimizar los fallos de cache.

Por otro lado, la estrategia *GDS (p)* asocia un coste a cada objeto igual al mostrado en la *Ecuación 2.3*:

$$\cos te = \left(2 + \frac{tamaño\_objeto}{536}\right) \qquad \textbf{Ecuación 2.3}$$

Con lo que el valor final de la clave resulta ser el de la *Ecuación 2.4*:

$$clave = \left(\frac{\cos te}{tamaño\_objeto}\right) = \left(\frac{\left(2 + \frac{tamaño\_objeto}{536}\right)}{tamaño\_objeto}\right) = \left(\frac{2}{tamaño\_objeto} + \frac{1}{536}\right)$$

Ecuación 2.4

la nueva función coste es tal ya que es el número estimado de paquetes de red enviados y recibidos necesarios hasta que se produzca un fallo en la petición de un documento.

La estrategia *GDS (p)* consigue ambos, un *HR* y un *BHR* alto. *GDS (p)* trata de minimizar el tráfico *Web* resultante de los fallos de *cache*.

Exceptuada esta característica diferenciadora, el funcionamiento global del algoritmo es idéntico para ambas políticas de reemplazo. A continuación se describirá este funcionamiento básico a la hora de realizar una inserción, una eliminación, o al producirse un acierto:

1) **Llega un elemento nuevo** → Se analiza si el nuevo elemento cabe en la cache. De ser así, se almacenará en la misma, de manera ordenada, en función de su campo clave. Así elementos con una mayor clave (menor tamaño) se situarán siempre en la zona superior de la *cache*, mientras que elementos de menor clave

(mayor tamaño) se ubicarán en la zona inferior de la misma y serán futuros candidatos para ser eliminados.

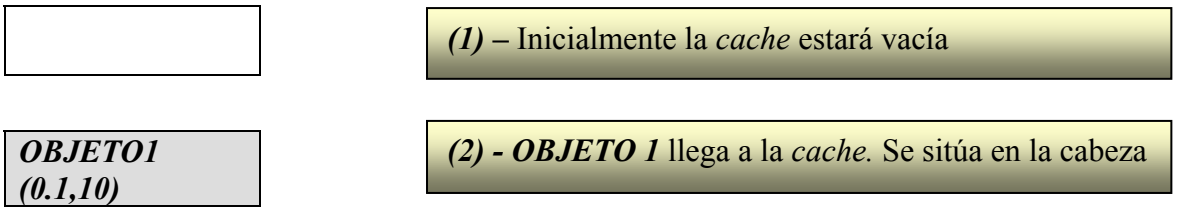
2) **Llega un elemento nuevo que no cabe** → Se eliminan objetos de la *cache*, comenzando desde el último, hasta que el espacio liberado sea el suficiente como para insertar el nuevo elemento. Se toma la clave del último elemento eliminado, valor que será restado a las claves de todos los objetos almacenados en el sistema. Una vez recalculadas todas las claves en la *cache*, el nuevo elemento se introduce en la misma (en la posición que le corresponda en función de su clave).
3) **Se produce un acierto en la *cache*** → O lo que es lo mismo, el documento solicitado está en la *cache*, y por tanto será servido por la misma. En tal caso se realizará una búsqueda de este objeto almacenado, y una vez localizado (se consigue su índice) se restaura el valor de su clave al original, y se reubica en función de ese nuevo valor.

Como resultado, tal y como se indicó anteriormente, la cache aparecerá ordenada en función del valor de las claves de cada uno de los elementos del sistema. El hecho de que tras una eliminación, se recalcule las claves de todos los elementos no es más que un mecanismo de envejecimiento de documentos.

Por supuesto en caso de llegar un elemento cuyo tamaño excediese el propio de la *cache*, éste sería directamente descartado para ser almacenado.

A continuación se realizarán dos pequeñas simulaciones, con la que se pretende alcanzar una total comprensión de cómo funciona el algoritmo. La primera de las cuales hace referencia a una política de reemplazo *GDS (1),* mientras que en la segunda se trata una *GDS (p)* (entre paréntesis se indicará el valor, ya calculado, de la clave de cada uno de los objetos, y a continuación el del tamaño del mismo):

Cache Web *(GDS (1))*



OBJETO1 (0.1,10)
OBJETO2 (0.05,20)

(3) – OBJETO 2 llega a la cache. Se ubica de manera ordenada en la *cache.*

OBJETO1 (0.1,10)
OBJETO3 (0.08,12.5)
OBJETO2 (0.05,20)

(4) – OBJETO 3 llega a la cache. Se ubica de manera ordenada en la *cache.*

OBJETO4 (0.2,5)
OBJETO1 (0.05,10)
OBJETO3 (0.03,12.5)

(5) – OBJETO 4 llega a la *cache*. No cabe. Se elimina el último objeto de la misma (OBJETO2). Se actualiza el valor de la clave del resto de elementos almacenados. Ya cabe. Se ubica de manera ordenada en la cache.

OBJETO4 (0.2,5)
OBJETO3 (0.08,12.5)
OBJETO1 (0.05,10)

(6) – OBJETO 3 llega a la caché. Éste elemento ya está almacenado, es un acierto. Se restaura su clave al valor original (0.08).Se reubica de manera ordenada.

Como se observa, el algoritmo *GDS (1)* gestiona la cache manteniendo los objetos ordenados en función del valor de su clave. Elementos de gran tamaño (clave pequeña) serán más propensos a ser eliminados, es decir, se ubicarán en la zona inferior de la memoria; mientras que elementos de menor tamaño permanecerán en la zona superior, y su eliminación será mucho más improbable.

Cache Web (GDS (p))

(1) – Inicialmente la *cache* estará vacía

OBJETO1 (0.20186,10)

(2) - OBJETO 1 llega a la *cache.* Se sitúa en la cabeza

OBJETO1 (0.20186,10)
OBJETO2 (0.10186,20)

(3) – OBJETO 2 llega a la cache. Se ubica de manera ordenada en la *cache.*

OBJETO1 (0.20186,10)
OBJETO3 (0.16186,12.5)
OBJETO2 (0.10186,20)

(4) – OBJETO 3 llega a la cache. Se ubica de manera ordenada en la *cache.*

OBJETO4 (0.40186,5)
OBJETO1 (0.1,10)
OBJETO3 (0.06,12.5)

(5) – OBJETO 4 llega a la *cache*. No cabe. Se elimina el último objeto de la misma (OBJETO2). Se actualiza el valor de la clave del resto de elementos almacenados. Ya cabe. Se ubica de manera ordenada en la cache..

OBJETO4 (0.40186,5)
OBJETO3 (0.16186,12.5)
OBJETO 1(0.1,10)

(6) – OBJETO 3 llega a la caché. Éste elemento ya está almacenado, es un acierto. Se restaura su clave al valor original (0.16186).Se reubica de manera ordenada.

Resumiendo, vemos que el algoritmo *GDS (p)* aún cuando calcula la función coste de distinta manera, acaba ordenando los objetos en *cache* de manera similar. Elementos grandes, tendrán claves pequeñas y se situarán en la zona superior de la *cache*; mientras que elementos pequeños se situarán en la zona superior.

2.6 GDSF [2]

Otra de las estrategias que forman parte de aquellas 'basadas en función'. El comportamiento es muy similar a *GDS,* la diferencia radica en que en esta política el factor frecuencia que se vio en la *LFU,* es parte fundamental del cálculo de la clave de cada elemento. Por tanto el hecho de que un objeto ocupe una posición u otra ya no depende únicamente del tamaño del mismo, sino también de las veces que es referenciado.

En líneas generales, la cache se ordenará, como viene siendo costumbre, por la clave de cada objeto. Así claves mayores situarán al objeto en la zona superior de la

cache, mientras que claves bajas lo ubicarán en la zona inferior, por donde se comenzará a eliminar los objetos.

Como se ha comentado, en este caso, a la hora de calcular la llave de cada objeto se tendrá en cuenta la frecuencia con que se solicita, y como sucedió en el *apartado 2.5* , se describirán dos tipos de políticas de reemplazo *GDSF*, la *GDSF (1)* y la *GDSF (p).* El cálculo de la clave para la estrategia *GDSF (1)* se rige por la *Ecuación 2.5*:

$$clave = \left(L + \left(contadorx \left(\frac{\cos te}{tamaño\_objeto} \right) \right) \right) \overset{\cos te=1}{=} \left(L + \left(\frac{contador}{tamaño\_objeto} \right) \right) \quad \textbf{Ecuación 2.5}$$

Así, elementos de pequeño tamaño, o frecuentemente llamados serán los que conseguirán las claves más altas. Como se ha dicho, en este caso la función coste es la unidad, lo que quiere decir que *GDSF (1)* es una política que prima el *HR* sobre el *BHR,* lo que parece mejorar la latencia, es decir, reducir el tiempo de respuesta.

La variable L no es más que un factor de inflación. Comenzará valiendo 0 y es actualizado al valor de clave del último elemento eliminado de la *cache.* Éste no es mas que un factor de envejecimiento, al igual que se ha visto anteriormente, cuyo objetivo es evitar la polución de cache, es decir, que elementos muy referenciados en el pasado, pero caídos en desuso actualmente puedan llegar a ser eliminados.

Mientras tanto, para el caso de la estrategia *GDSF (p)* la función coste se calcula tal y como se describió en la *Ecuación 2.3*, con lo cual, el valor final de la clave en este caso vendrá definido por la *Ecuación 2.6*:

$$clave = \left(L + contadorx \left(\frac{\cos te}{tamaño\_objeto} \right) \right) = \left(L + contadorx \left(\frac{\left(2 + \frac{tamaño\_objeto}{536} \right)}{tamaño\_objeto} \right) \right) =$$

$$= \left(L + contadorx \left(\frac{2}{tamaño\_objeto} + \frac{1}{536} \right) \right) \quad \textbf{Ecuación 2.6}$$

Como se enunció en la *GDS (p),* el valor que adquiere la función coste es muy característico, ya que es el número estimado de paquetes de red enviados y recibidos necesarios hasta que se produzca un fallo en la petición de un documento.

Con este valor de la función coste conseguimos tanto un *HR* como un *BHR* alto, minimizando, en la medida de lo posible, el tráfico *Web* debido a peticiones que provoquen fallos de *cache*.

A continuación se detallará el funcionamiento general del algoritmo, y que como se indicó en el *apartado 2.5*, es común para ambas políticas de reemplazo (L inicialmente vale 0):

1) **Llega un elemento nuevo** → Se analiza si el nuevo elemento cabe en la cache. De ser así, se almacenará en la misma, de manera ordenada, en función de su campo clave y teniendo en cuenta el valor actual del factor L (el contador de referencia inicialmente vale 1).
2) **Llega un elemento nuevo que no cabe** → Se eliminan objetos de la *cache*, comenzando desde el último, hasta que el espacio liberado sea el suficiente como para insertar el nuevo elemento. Se actualizará el factor L al valor de la clave del último documento eliminado, y se calculará la clave del nuevo elemento, clave que ubicará al elemento dentro de la *cache*.
3) **Se produce un acierto en la *cache*** → O lo que es lo mismo, el documento solicitado está en la *cache*, y por tanto será servido por la misma. En tal caso se realizará una búsqueda de este objeto almacenado, y una vez localizado (se consigue su índice) se recomputa el valor de su clave, teniendo en cuenta el valor actual del factor L, y que el contador de referencia se ha de incrementar en una unidad.

Finalmente, como ya era de esperar, la cache aparecerá ordenada en función del valor de las claves de cada uno de los elementos del sistema.

Por último se realizará dos pequeñas simulaciones para comprender mejor el funcionamiento de ambas estrategias (entre paréntesis se indicará, por este orden, el valor, ya calculado, de la clave de cada uno de los objetos, el del tamaño del mismo y su contador de referencia. L inicialmente vale 0 y se hará referencia a él cuando se actualice):

Cache Web (GDSF (1))

	L=0	***(1) –*** Inicialmente la *cache* estará vacía
OBJETO1 (0.1,10,1)		***(2) - OBJETO 1*** llega a la *cache.* Se sitúa en la cabeza
OBJETO1 (0.1,10,1) ***OBJETO2 (0.05,20,1)***		***(3) – OBJETO 2*** llega a la cache. Se ubica de manera ordenada en la *cache.*
OBJETO1 (0.1,10,1) ***OBJETO3 (0.08,12.5,1)*** ***OBJETO2 (0.05,20,1)***		***(4) – OBJETO 3*** llega a la cache. Se ubica de manera ordenada en la *cache.*
OBJETO4 (0.25,5,1) ***OBJETO1 (0.1,10,1)*** ***OBJETO3 (0.08,12.5,1)***	L=0.05	***(5) – OBJETO 4*** llega a la *cache*. No cabe. Se elimina el último objeto de la misma (OBJETO2). Se actualiza el valor de la clave del resto de elementos almacenados. Ya cabe. Se ubica de manera ordenada
OBJETO4 (0.25,5,1) ***OBJETO3 (0.21,12.5,2)*** ***OBJETO1 (0.01,10,1)***		***(6) – OBJETO 3*** llega a la caché. Éste elemento ya está almacenado, es un acierto. Se recalcula su clave, teniendo en cuenta el nuevo valor del factor L e incrementando su contador de referencia.

Resumiendo, el algoritmo *GDSF (1)* gestiona la cache manteniendo los objetos ordenados en función de el valor de su clave, y su clave será mayor cuanto mas veces sean referenciados o menor sea su tamaño. Incorpora también un facto r de inflación L que pretender aliviar los efectos de la polución de *cache*.

Cache Web (GDSF (p))

	L=0	***(1) –*** Inicialmente la *cache* estará vacía
OBJETO1 (0.20186,10,1)		***(2) - OBJETO 1*** llega a la *cache.* Se sitúa en la cabeza
OBJETO1 (0.20186,10,1) ***OBJETO2 (0.10186,20,1)***		***(3) – OBJETO 2*** llega a la cache. Se ubica de manera ordenada en la *cache.*
OBJETO1 (0.20186,10,1) ***OBJETO3 (0.16186,12.5,1)*** ***OBJETO2 (0.10186,20,1)***		***(4) – OBJETO 3*** llega a la cache. Se ubica de manera ordenada en la *cache.*
OBJETO4 (0.50372,5,1) ***OBJETO1 (0.20186,10,1)*** ***OBJETO3 (0.16186,12.5,1)***	L=0.10186	***(5) – OBJETO 4*** llega a la *cache*. No cabe. Se elimina el último objeto de la misma (OBJETO2). Se actualiza el valor de la clave del resto de elementos almacenados. Ya cabe. Se ubica de manera ordenada
OBJETO4 (0.50372,5,1) ***OBJETO3 (0.42558,12.5,2)*** ***OBJETO1 (0.20186,10,1)***		***(6) – OBJETO 3*** llega a la caché. Éste elemento ya está almacenado, es un acierto. Se recalcula su clave, teniendo en cuenta el nuevo valor del factor L e incrementando su contador de referencia.

El algoritmo *GDSF (p)* ordena la cache ubicando los objetos en función de el valor de su clave, y su clave será mayor cuanto mas veces sean referenciados o menor sea su tamaño, tal y como sucedía en *GDSF (1)*, sólo distinguiéndose en la manera en que calculan la función coste.

2.7 GD* [4]

Como ocurrió en los últimos apartados, el algoritmo *GD** forma parte del grupo de políticas 'basadas en función'.

Tal y como sucedió en *GDS,* esta política se basa en el cálculo de una clave por documento, con la que éstos serán ordenados dentro de la cache. El valor de esta clave dependerá del tamaño del propio objeto, así como de las veces que haya sido solicitado o del factor de inflación L; pero en este caso también entra en liza el uso de una constante β, que se comentará más adelante.

Como es de suponer, se estudiarán dos versiones para esta política de reemplazo. En la primera de ellas se supondrá una función coste constante y de valor igual a la unidad (favorece el HR y por tanto los tiempos de respuesta). Con todo, el cálculo del campo clave será muy similar a los hasta ahora vistos e igual al mostrado en la *Ecuación 2.7*:

$$clave = \left(L + \left(contadorx \frac{\cos te}{tamaño\_objeto} \right)^{\frac{1}{\beta}} \right) \overset{\cos te=1}{=} \left(L + \left(\frac{contador}{tamaño\_objeto} \right)^{\frac{1}{\beta}} \right) \quad \textbf{Ecuación 2.7}$$

Mientras que para el caso de una política *GD * (p)* el cálculo de la clave será el propuesto en la *Ecuación 2.8*:

$$clave = \left(L + \left(contadorx \left(\frac{\cos te}{tamaño\_objeto} \right) \right)^{\frac{1}{\beta}} \right) = \left(L + \left(contadorx \left(\frac{\left(2 + \frac{tamaño\_objeto}{536} \right)}{tamaño\_objeto} \right) \right)^{\frac{1}{\beta}} \right) =$$

$$= \left(L + \left(contadorx \left(\frac{2}{tamaño\_objeto} + \frac{1}{536} \right) \right)^{\frac{1}{\beta}} \right) \quad \textbf{Ecuación 2.8}$$

Como ya se dijo, con este valor de la función coste se consigue tanto un *HR* como un *BHR* alto, minimizando, en la medida de lo posible, el tráfico *Web* causado por los fallos de *cache*.

¿Cuál es la razón de esa constante β? β es un factor de peso, el cual es estimado como una moda *online,* y cuyo objetivo será diferenciar la probabilidad de que elementos de igual contador de referencia (igual de populares) sean llamados en cada momento, esto es, intenta adecuar la clave de cada objeto a la probabilidad de que un objeto sea, o no, referenciado (intenta aunar en la clave la popularidad de un elemento y como de recientemente ha sido llamado). Es una forma más de controlar el envejecimiento de los documentos de la *cache*. El cálculo de este parámetro es

determinado estudiando el contador de referencia de cada objeto, como función de las referencias hechas entre dos llamadas sucesivas al mismo documento para documentos igualmente populares.

Así estas políticas intentan solventar el problema de polución de cache, apelando

a dos distintas formas de solventarlo, el empleo del factor de inflación L, y la constante β.

Dicho todo esto, el siguiente paso será realizar un pequeño resumen de cuál es el funcionamiento general común a ambas estrategias:

1) **Llega un elemento nuevo** → Se analiza si el nuevo elemento cabe en la cache. De ser así, se almacenará en la misma, de manera ordenada, en función de su campo clave y teniendo en cuenta el valor actual del factor L (el contador de referencia inicialmente vale 1).
2) **Llega un elemento nuevo que no cabe** → Se eliminan objetos de la *cache*, comenzando desde el último, hasta que el espacio liberado sea el suficiente cómo para insertar el nuevo elemento. Se actualizará el factor L al valor de la clave del último documento eliminado, y se calculará la clave del nuevo elemento, teniendo en cuenta el nuevo valor del factor L (en la posición que le corresponda en función de su clave), clave que ubicará al elemento dentro de la *cache*.
3) **Se produce un acierto en la *cache*** → O lo que es lo mismo, el documento solicitado está en la *cache*, y por tanto será servido por la misma. En tal caso se realizará una búsqueda de este objeto almacenado, y una vez localizado (se consigue su índice) se recomputa el valor de su clave, teniendo en cuenta el valor actual del factor L, y que el contador de referencia se ha de incrementar en una unidad.

A continuación se realizará un ejemplo con el que se entenderá con mayor claridad el algoritmo (entre paréntesis se indicará por este orden el valor, ya calculado, de la clave de cada uno de los objetos, el del tamaño del mismo y su contador de referencia. L inicialmente vale 0 y se hará referencia a él cuando se actualice), el valor que emplearemos para la constante β será 0.5:

Cache Web (GD* (1))

	L=0	***(1) –*** Inicialmente la *cache* estará vacía
OBJETO1 (0.01,10,1)		***(2) - OBJETO 1*** llega a la *cache.* Se sitúa en la cabeza
OBJETO1 (0.01,10,1) ***OBJETO2 (0.0025,20,1)***		***(3) – OBJETO 2*** llega a la cache. Se ubica de manera ordenada en la *cache.*
OBJETO1 (0.01,10,1) ***OBJETO3 (0.0064,12.5,1)*** ***OBJETO2 (0.0025,20,1)***		***(4) – OBJETO 3*** llega a la cache. Se ubica de manera ordenada en la *cache.*
OBJETO4 (0.0425,5,1) ***OBJETO1 (0.01,10,1)*** ***OBJETO3 (0.0064,12.5,1)***	L =0.00255	***(5) – OBJETO 4*** llega a la *cache*. No cabe. Se elimina el último objeto de la misma (OBJETO2). Se actualiza el valor de la clave del resto de elementos almacenados. Ya cabe. Se ubica de manera ordenada
OBJETO4 (0.0425,5,1) ***OBJETO3 (0.0281,12.5,2)*** ***OBJETO1 (0.01,10,1)***		***(6) – OBJETO 3*** llega a la caché. Éste elemento ya está almacenado, es un acierto. Se recalcula su clave, teniendo en cuenta el nuevo valor del factor L e incrementando su contador de referencia.

Resumiendo, en *GD* (1)* los objetos quedan ordenados en *cache* en función de su clave. Emplea coste unidad y diversos mecanismos de control del envejecimiento del sistema como es el factor de inflación L o la constante β.

Cache Web (GD* (p))

	L=0	***(1) –*** Inicialmente la *cache* estará vacía

OBJETO1 (0.04075,10,1)

(2) - OBJETO 1 llega a la *cache.* Se sitúa en la cabeza

OBJETO1 (0.04075,10,1)
OBJETO2 (0.01037,20,1)

(3) – OBJETO 2 llega a la cache. Se ubica de manera ordenada en la *cache.*

OBJETO1 (0.04075,10,1)
OBJETO3 (0.0262,12.5,1)
OBJETO2 (0.01037,20,1)

(4) – OBJETO 3 llega a la cache. Se ubica de manera ordenada en la *cache.*

OBJETO4 (0.1719,5,1)	L =0.01037
OBJETO1 (0.04075,10,1)	
OBJETO3 (0.0262,12.5,1)	

(5) – OBJETO 4 llega a la *cache*. No cabe. Se elimina el último objeto de la misma (OBJETO2). Se actualiza el valor de la clave del resto de elementos almacenados. Ya cabe. Se ubica de manera ordenada

OBJETO4 (0.1719,5,1)
OBJETO3 (0.1152,12.5,2)
OBJETO1 (0.04075,10,1)

(6) – OBJETO 3 llega a la caché. Éste elemento ya está almacenado, es un acierto. Se recalcula su clave, teniendo en cuenta el nuevo valor del factor L e incrementando su contador de referencia.

GD (p)* es una política casi idéntica a *GD* (1)*, por tanto, hereda casi todas sus características, e incorpora las propias de su nueva función coste.

2.8 RAND [1]

Esta política estará encuadrada en el marco de las 'políticas aleatorias', es decir, aquellas políticas que liberan espacio en la *cache* seleccionando documentos a eliminar de manera aleatoria.

Este conjunto de estrategias intentan reducir la complejidad de los procesos de reemplazo, intentando no sacrificar en demasía el rendimiento de los mismos. Como sucedió en anteriores apartados, en este grupo se englobarán otras muchas estrategias, cada una de las cuales se detallarán con mayor atención en apartados posteriores.

Es evidente intuir, que este grupo de políticas presentarán una serie de ventajas e inconvenientes. Las ventajas más relevantes son las siguientes:

- Las estrategias aleatorias no necesitan estructuras de datos especiales para insertar o borrar documentos.
- Son estrategias muy simples de implementar.

Por el contrario las desventajas que se presentan son las siguientes:

- Son estrategias mucho más difíciles de evaluar, ya que diversas simulaciones con una misma muestra de entrada arrojarían resultados muy diversos.

Con todo, el funcionamiento del algoritmo es tremendamente simple, en caso de ser necesario eliminar documentos, estos se seleccionarán de forma aleatoria. Dado que el algoritmo elimina objetos aleatoriamente, la forma en que éstos serán insertados no es relevante, ya que los objetos tendrán la misma probabilidad de ser eliminados sea cual sea la posición en que se encuentren.

De esta manera obtenemos una *cache* que no estará ordenada de ninguna manera. Por supuesto, en este caso el campo clave no tiene sentido y no es calculado.

Por comodidad, y para establecer un criterio común a todas las políticas aleatorias, se considerará que la inserción se realizará siempre al final de la *cache* (en este caso elementos acertados no se reubican).

Dicho lo cual, se procederá a continuación a la presentación del algoritmo en cuestión:

1) **Llega un elemento nuevo** → Se analiza si el nuevo elemento cabe en la cache. De ser así, se almacenará en la cabeza de la *cache* (es decir, en la posición superior de la misma). El resto de elementos almacenados serán desplazados una posición.
2) **Llega un elemento nuevo que no cabe** → Se eliminan objetos de la *cache*, hasta que el espacio liberado sea el suficiente como para insertar el nuevo elemento. Los objetos a eliminar se elegirán de manera aleatoria.
3) **Se produce un acierto en la *cache*** → O lo que es lo mismo, el documento solicitado está en la *cache*, y por tanto será servido por la misma. En tal caso se deja en la posición en que estaba (ya que no tiene ningún sentido reubicarlo).

A continuación realizaremos una pequeña demostración (la *cache* originariamente estará vacía):

Cache Web

	(1) – Inicialmente la *cache* estará vacía
OBJETO 1	***(2) - OBJETO 1*** llega a la *cache.* Se sitúa en la última posición
OBJETO 1 ***OBJETO 2***	***(3) – OBJETO 2*** llega a la cache. Se sitúa en la última posición
OBJETO 1 ***OBJETO 2*** ***OBJETO 3***	***(4) – OBJETO 3*** llega a la *cache*. Se sitúa en la última posición
OBJETO 1 ***OBJETO 2*** ***OBJETO 4***	***(5) – OBJETO 4*** llega a la *cache*. No cabe. Se elimina un documento aleatoriamente (OBJETO3). Ya cabe. Se inserta en la última posición.
OBJETO 2 ***OBJETO 4*** ***OBJETO 3***	***(6) – OBJETO 3*** llega a la caché. No cabe. Se elimina un documento aleatoriamente (OBJETO1). Ya cabe. Se sitúa en la cabeza

En resumen, esta es la política aleatoria de mayor sencillez, y se basa, básicamente, en (siempre que sea necesario) eliminar objetos de la *cache* seleccionándolos de manera aleatoria.

2.9 HARM [10]

Otra de las llamadas 'políticas aleatorias', tremendamente similar a la *RAND* pero con una particular diferencia que se expondrá a continuación.

En el apartado anterior, los objetos a eliminar se elegían de manera aleatoria; pero suponiendo una distribución de probabilidad constante, es decir, todos y cada uno de los objetos almacenados tenían la misma probabilidad de ser escogidos, independiente de su tamaño, contador de referencia o cualquier otro de sus campos.

En esta ocasión, la probabilidad de eliminar un objeto es inversamente proporcional a su campo clave. Dado que se ha supuesto, en este caso una función coste constante e igual a la unidad, la probabilidad de que un documento sea eliminado, será directamente proporcional al tamaño de cada objeto, como se indica en la *Ecuación 2.9*:

$$probabilidad \propto \left(\frac{\cos te}{tamaño\_objeto} \right)^{-1} \xrightarrow{\cos te=1} probabilidad \propto (tamaño\_objeto)$$

Ecuación 2.9

Así, elementos muy grandes tendrán una mayor probabilidad de ser eliminados, es decir, se prima a los documentos pequeños sobre los grandes, o dicho de otra manera, dado que se busca almacenar muchos documentos, aunque de menor tamaño, se estará consiguiendo altos *HR* a coste de perjudicar el *BHR*. Premisa que coincide con la arrojada por el hecho de asignar una función coste igual a 1.

Exceptuando este comportamiento, el resto del algoritmo será el mismo que el mostrado en la política *RAND*. Aun así describiremos su funcionamiento básico una vez más:

1) **Llega un elemento nuevo** → Se analiza si el nuevo elemento cabe en la cache. De ser así, se almacenará en la cabeza de la *cache* (es decir, en la posición superior de la misma). El resto de elementos almacenados serán desplazados una posición.
2) **Llega un elemento nuevo que no cabe** → Se eliminan objetos de la *cache*, hasta que el espacio liberado sea el suficiente como para insertar el nuevo
3) elemento. Los objetos a eliminar se elegirán de manera aleatoria, teniendo en cuenta que el elemento más pesado será también el que más probablemente será eliminado.
4) **Se produce un acierto en la *cache*** → O lo que es lo mismo, el documento solicitado está en la *cache*, y por tanto será servido por la misma. En tal caso se deja en la posición en que estaba (ya que no tiene ningún sentido reubicarlo).

A continuación se realizará una pequeña simulación para comprender mejor el algoritmo (entre paréntesis se indicará el valor del tamaño del mismo):

Cache Web

	(1) – Inicialmente la *cache* estará vacía
OBJETO1 (10)	***(2) - OBJETO 1*** llega a la *cache.* Se sitúa en la última posición
OBJETO 1(10) ***OBJETO 2(20)***	***(3) – OBJETO 2*** llega a la cache. Se sitúa en la última posición
OBJETO 1(10) ***OBJETO 2(20)*** ***OBJETO 3(12.5)***	***(4) – OBJETO 3*** llega a la cache. Se sitúa en la última posición
OBJETO 1(10) ***OBJETO 3(12.5)*** ***OBJETO 4(5)***	***(5) – OBJETO 4*** llega a la *cache*. No cabe. Se elimina un elemento de manera aleatoria, siendo más probables los más pesados (OBJETO2).Ya cabe. Se sitúa en la última posición
OBJETO 1(10)) ***OBJETO 3(12.5)*** ***OBJETO 4(5***	***(6) – OBJETO 3*** llega a la caché. Éste elemento ya está almacenado, es un acierto. Se deja tal y como estaba

Resumiendo, el algoritmo *HARM (1)* elimina los documentos de forma aleatoria; pero con una probabilidad directamente proporcional a su tamaño. Por lo tanto, documentos mayores serán descartes más probables, o dicho de otra forma, otorga preferencia a documentos pequeños sobre los de mayor tamaño.

2.10 LRU-C [9]

Versión aleatoria del algoritmo *LRU.* Como sucedió en dicha estrategia, los elementos se insertan en la cabeza de la *cache*, y a la hora de provocar alguna extracción, ésta se realizará comenzando desde el final del sistema.

Pero, entonces, ¿dónde radica el componente aleatorio del algoritmo? Su comprensión es simple, cada vez que se produzca la petición de un documento, y éste sea un acierto de *cache*, se localizará el elemento, y se hará la siguiente operación, dividiremos el valor de su coste, entre el coste máximo de entre todos los elementos guardados. Con esto obtendremos una valor entre 0 y 1, que será la probabilidad (en tanto por uno) de que ese elemento acertado sea movido a la cabeza, o no.

La función coste será la misma que la exhibida en apartados anteriores:

$$\cos te = \left(2 + \frac{tamaño\_objeto}{536}\right) \quad \rightarrow \quad probabilidad = \frac{\cos te_i}{\cos te_{\max}} \qquad \textbf{Ecuación 2.10}$$

Evidentemente, se ha descartado el empleo de la función coste unitaria, ya que en ese caso la probabilidad de mover el objeto a la cabeza sería siempre 1, es decir, los documentos acertados serían movidos a la cabeza de la *cache* constantemente.

La conclusión más evidente que se puede obtener tras lo dicho es la siguiente, el algoritmo *LRU* no es más que el *LRU-C* con un coste unitario.

Pero, para una mejor compresión del algoritmo se describirá como actúa ante diversas circunstancias de tráfico *Web*:

1) **Llega un elemento nuevo** → Se analiza si el nuevo elemento cabe en la cache. De ser así, se almacenará en la cabeza de la *cache* (es decir, en la posición superior de la misma). El resto de elementos almacenados serán desplazados una posición.
2) **Llega un elemento nuevo que no cabe** → Se eliminan objetos de la *cache*, comenzando desde el último, hasta que el espacio liberado sea el suficiente como para insertar el nuevo elemento (en la cabeza de la *cache*).
3) **Se produce un acierto en la *cache*** → O lo que es lo mismo, el documento solicitado está en la *cache*, y por tanto será servido por la misma. En tal caso se realizará una búsqueda de este objeto almacenado, y una vez localizado (se consigue su índice) será movido a la cabeza de la *cache* con una probabilidad $p = \frac{\cos te_i}{\cos te_{\max}}$; en otro caso no se hará nada.

Por último se realizará una breve simulación, el objetivo de ésta será ver de manera gráfica como se comporta el algoritmo (entre paréntesis se indicará, por este orden, el valor de la función coste y a continuación el tamaño del propio documento):

Cache Web

	(1) – Inicialmente la *cache* estará vacía
OBJETO1 (2.0186,10)	***(2) - OBJETO 1*** llega a la *cache.* Se sitúa en la cabeza
OBJETO2 (2.0373,20) ***OBJETO1 (2.0186,10)***	***(3) – OBJETO 2*** llega a la cache. Se sitúa en la cabeza
OBJETO3 (2.0233,12.5) ***OBJETO2 (2.0373,20)*** ***OBJETO1 (2.0186,10)***	***(4) – OBJETO 3*** llega a la *cache*. Se sitúa en la cabeza
OBJETO4 (2.0093,5) ***OBJETO3 (2.0233,12.5)*** ***OBJETO2 (2.0373,20)***	***(5) – OBJETO 4*** llega a la *cache*. No cabe. Se elimina el último objeto de la misma (OBJETO1). Ya cabe. Se inserta en la cabeza.
OBJETO3 (2.0233,12.5) ***OBJETO4 (2.0093,5)*** ***OBJETO2 (2.0373,20)***	$C_{\max} = 2.0373$ ***(6) – OBJETO 3*** llega a la caché. Éste elemento ya está almacenado, es un acierto. Se calcula la probabilidad con que será movido, $p = 98.6\%$. El documento se reubica en la cabeza.

Resumiendo, ésta no es más que una versión aleatoria del popular *LRU,* inserta en la cabeza, elimina comenzando por el final de la *cache.* La única diferencia se produce al darse un acierto, ya que el elemento acertado se moverá o no según un índice de probabilidad dependiente del coste de los documentos.

2.11 LRU-S [9]

LRU-S es una nueva variante aleatoria del algoritmo *LRU,* y por tanto, es tremendamente similar al estudiado en el apartado anterior, *LRU-C.*

La única variación radica en la manera en que esta política calcula la probabilidad con que un objeto acertado será, o no, movido a la cabeza del sistema. Por tanto, documentos nuevos se insertarán en la cabeza de la *cache*, y en caso de necesitar extraer elementos se hará comenzando por el final de la misma.

La probabilidad con que un objeto acertado será movido a la cabeza se calculará como el cociente entre el tamaño mínimo de entre todos los objetos almacenados y el tamaño del objeto acertado, es decir:

$$probabilidad = \frac{tamaño\_objeto_{\min}}{tamaño\_objeto_i} \qquad \textbf{Ecuación 2.11}$$

Así, cuanto más pequeño sea el documento que ha producido el acierto en *cache*, mayor probabilidad habrá de que el elemento sea movido.

A continuación, se realizará una síntesis del funcionamiento, en líneas generales, del algoritmo:

1) **Llega un elemento nuevo** → Se analiza si el nuevo elemento cabe en la cache. De ser así, se almacenará en la cabeza de la *cache* (es decir, en la posición superior de la misma). El resto de elementos almacenados serán desplazados una posición.
2) **Llega un elemento nuevo que no cabe** → Se eliminan objetos de la *cache*, comenzando desde el último, hasta que el espacio liberado sea el suficiente como para insertar el nuevo elemento (en la cabeza de la *cache*).
3) **Se produce un acierto en la *cache*** → O lo que es lo mismo, el documento solicitado está en la *cache*, y por tanto será servido por la misma. En tal caso se realizará una búsqueda de este objeto almacenado, y una vez localizado (se consigue su índice) será movido a la cabeza de la *cache* con una probabilidad $p = \frac{tamaño_{\min}}{tamaño_i}$; en otro caso no se hará nada.

Por supuesto en caso de llegar un elemento cuyo tamaño excediese el propio de la *cache*, éste sería directamente descartado para ser almacenado.

Para terminar, se realizará una breve simulación, cuyo objetivo será ver de manera gráfica como se comporta el algoritmo (entre paréntesis se indicará el tamaño de cada documento):

Cache Web

	(1) – Inicialmente la *cache* estará vacía
OBJETO1 (10)	***(2) - OBJETO 1*** llega a la *cache.* Se sitúa en la cabeza
OBJETO2 (20) ***OBJETO1 (10)***	***(3) – OBJETO 2*** llega a la cache. Se sitúa en la cabeza
OBJETO3 (12.5) ***OBJETO2 (20)*** ***OBJETO1 (10)***	***(4) – OBJETO 3*** llega a la *cache*. Se sitúa en la cabeza
OBJETO4 (5) ***OBJETO3 (12.5)*** ***OBJETO2 (20)***	***(5) – OBJETO 4*** llega a la *cache*. No cabe. Se elimina el último objeto de la misma (OBJETO1). Ya cabe. Se inserta en la cabeza.
OBJETO4 (5) ***OBJETO3 (12.5)*** ***OBJETO2 (20)***	$tam_{\min} = 5$ ***(6) – OBJETO 3*** llega a la caché. Éste elemento ya está almacenado, es un acierto. Se calcula la probabilidad con que será movido, $p = 40\%$. No se hace nada.

Resumiendo, *LRU-S* es una nueva variante aleatoria del *LRU* y la única diferencia con *LRU-C* radica en la manera en que calculará la probabilidad que existe de que un documento acertado sea movido o no a la cabeza de la *cache*. En este caso, esa probabilidad se calculará a partir del tamaño de los objetos almacenados y del propio del objeto acertado.

2.12 RRGVF [7]

Acrónimo inglés de la expresión *Randomized Replacement with General Value Functions*, o traducido 'Reemplazo aleatorio con funciones de valor general'.

Ésta es, posiblemente, la política de naturaleza aleatoria más compleja que se tratará. Dada esa naturaleza aleatoria, la forma en que se insertará en la *cache* no es determinante, es decir, la cache no tendrá ningún tipo de orden, ya que de tenerlo éste se perdería al extraer elementos sin ningún tipo de regla, es decir, aleatoriamente.

Por comodidad, se ha establecido que elementos nuevos, o los que resulten un acierto serán insertados en la última posición de la *cache*.

A continuación, se describirá qué proceso se seguirá para seleccionar los elementos a eliminar, en caso de ser necesario. Esta estrategia selecciona N elementos aleatoriamente de la *cache* y selecciona los M menos útiles, guardándolos en memoria. Eliminaremos elementos de entre estos M (comenzando por los menos útiles) hasta liberar el espacio suficiente. En siguientes reemplazos, N-M nuevos documentos son seleccionados de la *cache* y se seleccionan los M menos útiles de entre estos N-M y los M previamente guardados. Eliminaremos elementos de entre estos M (comenzando por los menos útiles), hasta liberar el espacio suficiente. Así sucesivamente.

Se dice que se produce un error cuando el documento eliminado no forma parte del *n%* de los documentos guardados menos útiles de la *cache*. Así la probabilidad de error se podría calcular como $\left(1-\frac{n}{100}\right)^N$, que es aproximadamente igual a $\exp^{\frac{-nN}{100}}$, o lo que es lo mismo, incrementando N se puede conseguir que la probabilidad de error tienda exponencialmente a 0.

El valor de n y N se definirán al iniciar el algoritmo, pero, ¿Cuál es el valor óptimo de M? M será calculado a partir de N y n, a través de la *Ecuación 2.12*:

$$M = N - \sqrt{\frac{(N+1)100}{n}} \qquad \textbf{Ecuación 2.12}$$

por tanto, ya tenemos todos los factores necesarios para definir el algoritmo. N indica el número de elementos que chequearemos antes de guardar los de menor utilidad en una memoria secundaria. Si N es mayor, el nº de objetos a examinar será mayor, y por tanto, la probabilidad de error se hace más pequeña. Con la constante n definiremos cuando entenderemos que se ha producido un error o no, y a partir de estos dos términos, podremos obtener el valor de M a partir de una Ecuación cerrada.

Pero, ¿cómo decidiremos que un documento resulta ser menos útil que otro? Existen multitud de funciones de utilidad, muchas de ellas ya han sido estudiadas, como el número de referencias de un objeto, el tiempo desde la última vez que fue llamado, o funciones más complejas como las vistas en *GDS (1)* o *GDS (p)* al calcular la clave. Cualquiera de ellas definiría una política *RRGVF* válida. En nuestro caso se optó por tomar el contador de referencia de un objeto como la función que definiría como de útil resulta, o no, el documento.

El propósito de esta estrategia es combinar los beneficios de estos algoritmos que emplean una función de utilidad como las antes expuestas, con las ventajas de las políticas aleatorias.

Dicho todo esto, el siguiente paso será realizar un pequeño resumen de cuál es el funcionamiento general de este algoritmo, para ello habría que definir antes el valor que tomarán las constantes N y M:

1) **Llega un elemento nuevo** → Se analiza si el nuevo elemento cabe en la cache. De ser así, se almacenará al final de la *cache* (es decir, en la última posición de la misma). El resto de elementos almacenados serán desplazados una posición.
2) **Llega un elemento nuevo que no cabe** → En caso de ser la primera vez que se hace eliminar algún elemento, tendremos que seleccionar N elementos aleatoriamente, de ellos tomaremos los M menos útiles e iremos eliminando objetos comenzando por el de menor contador de referencia. En caso de no ser la primera vez que eliminamos, seleccionaremos N-M objetos aleatoriamente, y elegiremos los M menos útiles de entre estos nuevos N-M y los M que ya tendríamos memorizados de anteriores iteraciones, e iremos eliminando objetos comenzando por el de menor contador de referencia hasta que el nuevo objeto quepa en la *cache*.
3) **Se produce un acierto en la *cache*** → O lo que es lo mismo, el documento solicitado está en la *cache*, y por tanto será servido por la misma. En tal caso se realizará una búsqueda de este objeto almacenado, y una vez localizado (se consigue su índice) se reubica en la última posición de la *cache*.

En caso de llegar un elemento cuyo tamaño excediese el propio de la *cache*, éste sería directamente descartado para ser almacenado.

Por último, una pequeña simulación que nos ayude a comprender mejor el algoritmo. Para ello supongamos N = 2 y M = 1(entre paréntesis se indicará el contador de referencia del elemento):

Cache Web

	(1) – Inicialmente la *cache* estará vacía
OBJETO 1(1)	***(2) - OBJETO 1*** llega a la *cache.* Se sitúa en la última posición
OBJETO 1 (1) ***OBJETO 2 (1)***	***(3) – OBJETO 2*** llega a la cache. Se sitúa en la última posición
OBJETO 1 (1) ***OBJETO 2 (1)*** ***OBJETO 3 (1)***	***(4) – OBJETO 3*** llega a la *cache*. Se sitúa en la última posición
OBJETO 1 (1) ***OBJETO 2 (1)*** ***OBJETO 3 (2)***	***(6) – OBJETO 3*** llega a la caché. Es un acierto. Se incrementa su contador y se reubica en la última posición de la lista
OBJETO 2 (1) ***OBJETO 3 (2)*** ***OBJETO 4 (1)***	***(5) – OBJETO 4*** llega a la *cache*. No cabe. Se selecciona N (2) elementos aleatoriamente (OBJETO1 y OBJETO3), y se guardan los M (1) de menor contador de referencia (OBJETO1). Se eliminan elementos de entre los guardados comenzando por el de menor contador (OBJETO1). Ya cabe. Se inserta el nuevo elemento al final de la lista.

En resumen, esta es una política aleatoria que intenta aunar también las ventajas de algoritmos que emplean funciones de utilidad. Su comportamiento aleatorio se hace patente a la hora de eliminar objetos. Esta eliminación es ciertamente compleja, y se basa en elegir una serie de candidatos a los que eliminar, de entre los cuales optaremos únicamente por los que muestren una función de utilidad menor.

CAPÍTULO 3: El Simulador

En el presente capítulo se pretende hacer un estudio detallado del simulador implementado.

Para ello se ha dividido el capítulo en tres apartados. El primero de ellos explica, a modo de manual de usuario, cuál es el funcionamiento del simulador, los distintos menús, como se realizará la entrada de datos o la selección de algoritmo. En resumen, trata de aportar la información necesaria para que un usuario final consiga realizar las simulaciones pertinentes con éxito.

En el segundo apartado, se pormenorizará cómo se ha realizado cada una de las políticas detalladas en el capítulo 2, teniendo en cuenta las características propias y las posibilidades que aporta el lenguaje de programación utilizado.

Por último, dedicaremos una sección a estudiar algunas de las estructuras empleadas para la realización del simulador, tales como el uso de clases de datos o hebras de ejecución.

3.1. MANUAL DE USUARIO

En primer lugar se hará un análisis del lenguaje de programación utilizado (C++), así como la herramienta usada para su desarrollo (Borland C++ Builder 6.0).

3.1.1. PROGRAMACIÓN VISUAL

Borland C++ Builder es una completa herramienta de programación visual para aplicaciones Windows de 32 bits, es decir, para funcionar en Windows 95 o posteriores.

La diferencia entre C++ Builder y otras herramientas de programación visual como Visual Basic, Delphi, Visual J++, etc., radica en que se utiliza el lenguaje de programación C++ para generar el código de las aplicaciones.

El lenguaje C++ es, sin lugar a dudas, el más potente a la hora de programar en Windows, aunque también presenta mayor complejidad que otros. Sin embargo, al utilizar una herramienta visual como C++ Builder, no es necesario preocuparse para nada del código correspondiente a la creación de ventanas, de situar uno u otro control, etc.

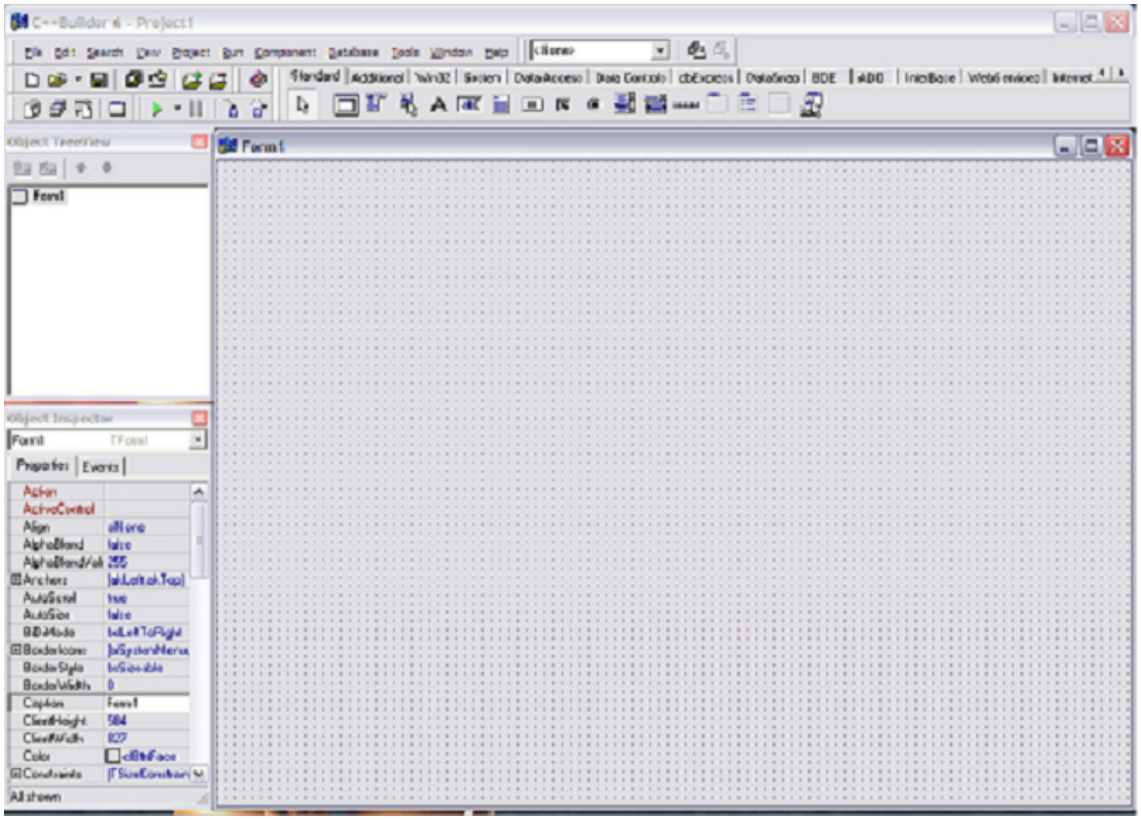


Figura 3.1. El entorno de programación

Ésta es la gran ventaja de C++ Builder, y no es poca cosa.

Pero, ¿por qué Borland C++ Builder 6.0 y no otras herramientas de programación visual en C++? la razón probablemente es algo subjetiva, y radica en la gran diversidad de funciones y librerías que ésta herramienta incorpora, y que resultan de gran utilidad, sobre todo, debido a la necesidad de diseñar algoritmos de gran eficacia, que manejan grandes archivos de entrada en un tiempo razonable.

3.1.2. FORMULARIO PRINCIPAL

En el diseño del simulador se han utilizado dos formularios principales, con los que el usuario podrá interactuar con el simulador, indicar la política de reemplazo a utilizar, la muestra de entrada, el tamaño de la cache…

A continuación mostraremos una pequeña introducción a cada uno de ellos.

Si ejecutamos el simulador, el primer formulario que aparecerá en pantalla será el correspondiente a la *Figura 3.2*:

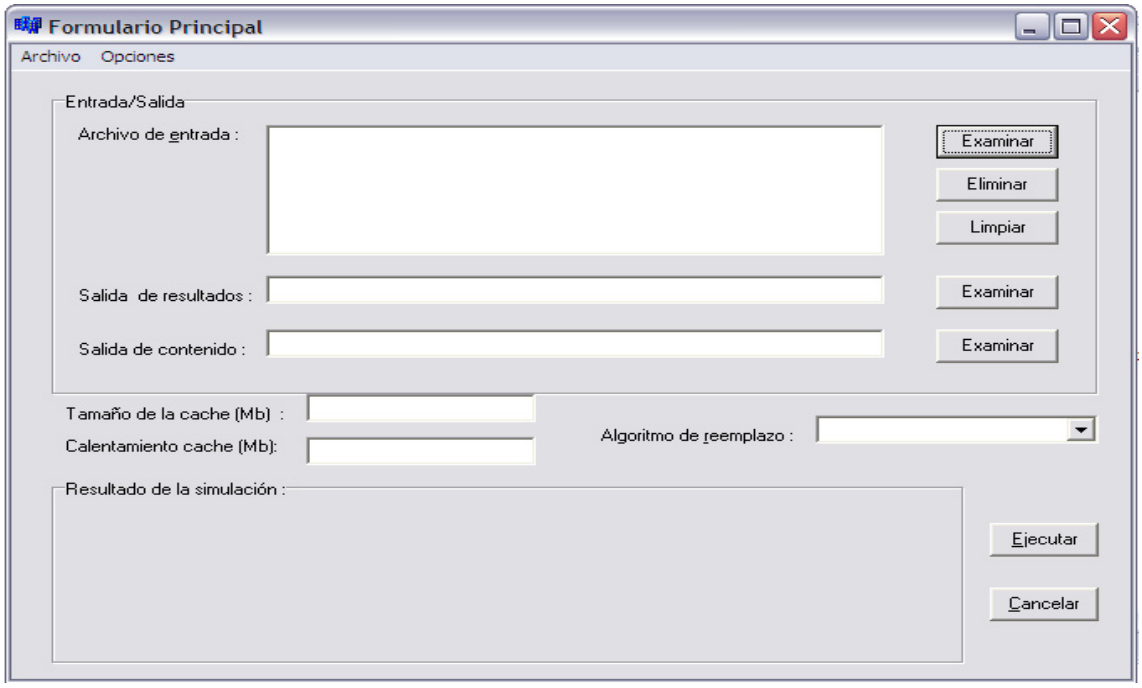


Figura 3.2. Formulario Principal

Como puede observarse en el título de la ventana, este será el *Formulario Principal* de la aplicación. Pasaremos a diseccionarlo con mayor detalle.

En la zona superior se puede observar un grupo denominado *Entrada/Salida:*

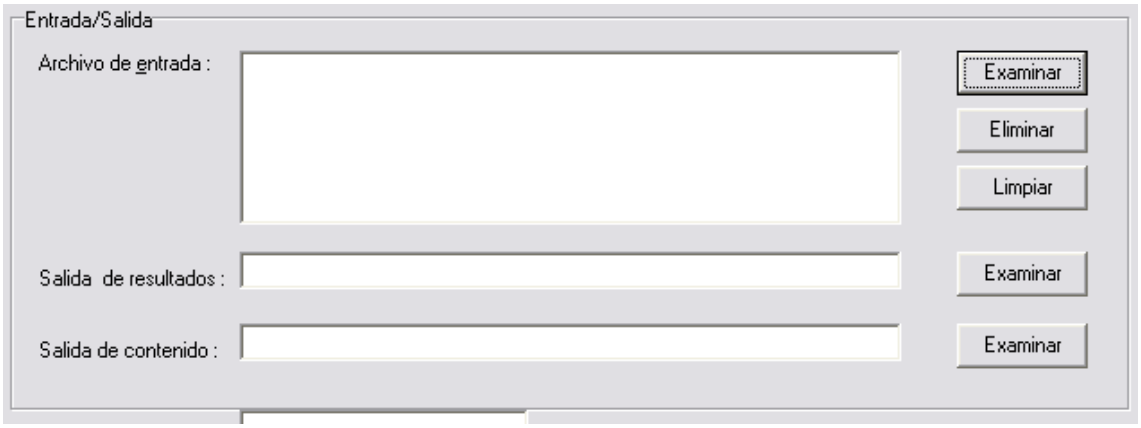


Figura 3.3. Grupo Entrada/Salida

Éste se compone a su vez de otros tres campos principales, *Archivo de entrada*, *Salida de Resultado* y *Salida de Contenido:*

- **Archivo de entrada:** En este campo asignaremos el (los) archivo(s) que contendrán la muestra de entrada que el algoritmo empleará a la hora de realizar la simulación. Este será un campo obligatorio para comenzar la simulación El botón *Examinar* que le acompaña abre el cuadro de diálogo mostrado en la *Figura 3.4*, para hacer la búsqueda de éste (estos) archivo(s).

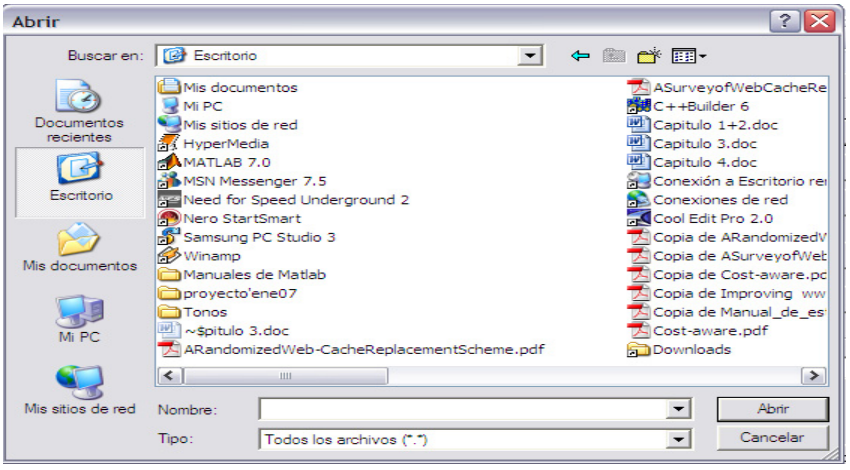


Figura 3.4. Cuadro de diálogo *Abrir*

El botón *Eliminar,* elimina el archivo de entrada marcado en negrita en la lista, mientras que el botón *Limpiar* los elimina todos.

- **Salida de resultado:** En este campo se seleccionará el archivo de salida que contendrá en formato de texto, los resultados más relevantes arrojados tras la conclusión de la simulación. El botón Examinar que le acompaña abre el cuadro de diálogo pertinente. tal y como se aprecia en la *Figura 3.5*:

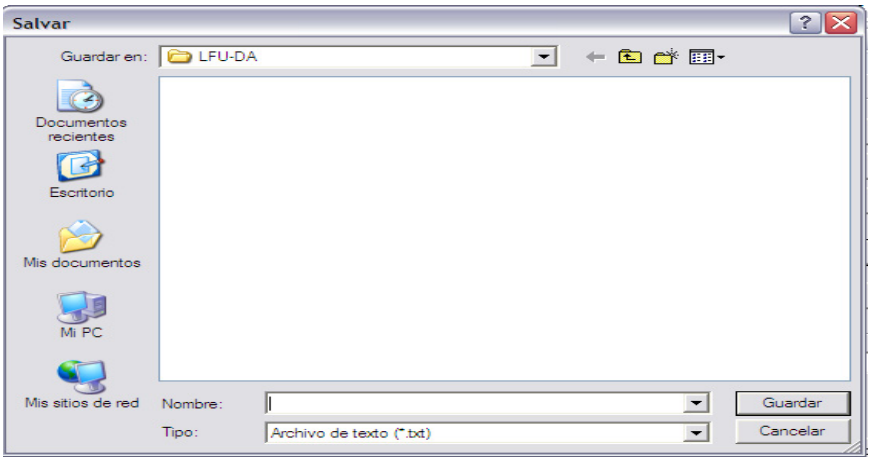


Figura 3.5. Cuadro de diálogo *Salvar*

- **Salida de contenido:** En este campo se seleccionará el archivo de salida que contendrá en formato de texto, el contenido de la *cache* al terminar la simulación, es decir, una lista en formato de texto con los documentos almacenados y sus campos más relevantes. El botón *Examinar* que le acompaña, abre un cuadro de diálogo idéntico al visto en la *Salida de resultado.*

A continuación, existe una segunda zona, donde se realizarán las decisiones fundamentales a la hora de definir la naturaleza y fin de la política seleccionada. Esta se describe en la *Figura 3.6*:

Figura 3.6. Tamaño cache, tamaño calentamiento y algoritmo Reemplazo

Como puede observarse, existen tres campos fundamentales, *Tamaño de la cache*, *Calentamiento cache* y *Algoritmo de Reemplazo*.

- **Tamaño de la *cache*:** En este campo se introduce el tamaño que tendrá la cache que emplearemos a la hora de realizar la simulación. Como ya se vio en el capítulo2, el tamaño de la *cache* es fundamental a la hora de definir el rendimiento de la misma, incluso más determinante que el propio algoritmo de reemplazo a emplear. Este campo será obligatorio para poder comenzar la simulación.
- **Calentamiento cache:** Campo en que se definirá el tamaño que se empleará para producir el calentamiento de *cache.* Campo obligatorio para comenzar la *simulación.*
- **Algoritmo de Reemplazo:** Tal y como se observa en la *Figura 3.7, s*e trata de una lista desplegable en la que seleccionaremos la política que se empleará para realizar la simulación. Este campo también será obligatorio completarlo para poder comenzar una simulación.

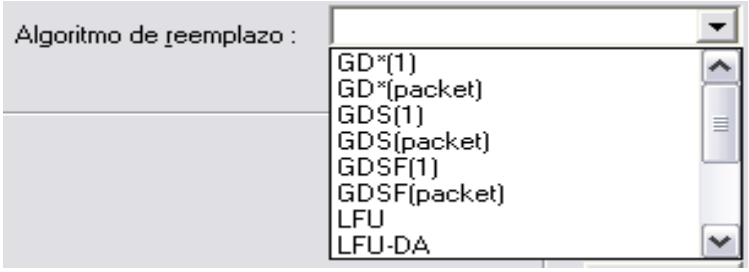


Figura 3.7. Lista desplegable 'Algoritmo reemplazo'

En la zona inferior, se observa la existencia de un nuevo grupo denominado *Resultado de la Simulación*

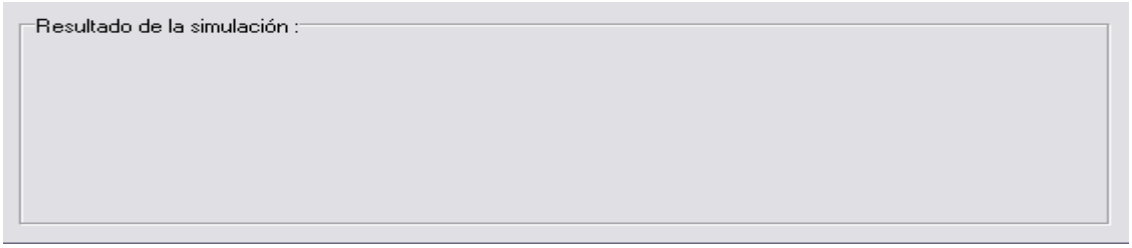
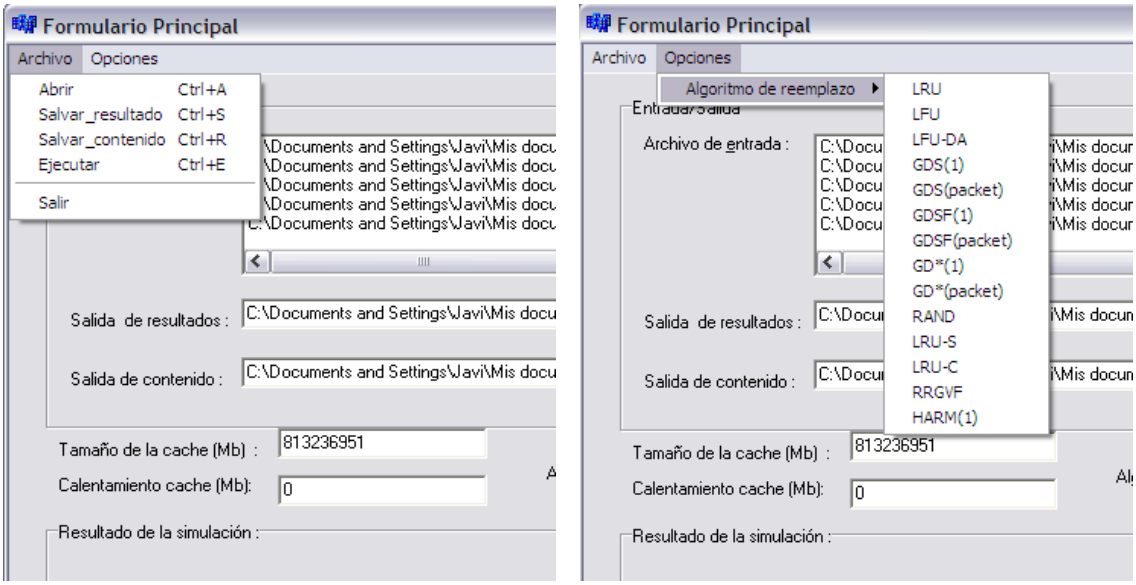


Figura 3.8. Grupo Resultado de la simulación

En este panel, se ilustrarán los resultados más destacados conseguidos tras la simulación, por tanto, aparecerá en principio vacío y sólo se rellenará al terminar una simulación.

En la barra de menú superior, aparecen dos casillas *Archivo* y *Opciones,* que seleccionarán mediante menús las principales opciones que hasta ahora se han destacado del *Formulario Principal.*



(a) **(b)**

Figura 3.9. Barra de menú

opciones tales como seleccionar la muestra de entrada, el algoritmo de reemplazo o el tamaño de la cache.

Por último, en la *Figura 3.10* se pueden observar dos botones situados en el margen inferior derecho del formulario, y que aún quedan por describir:

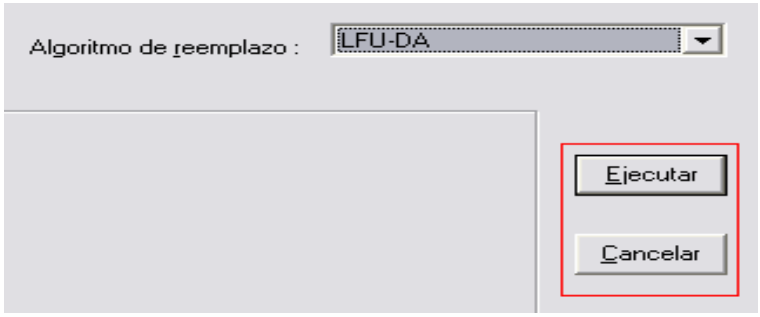


Figura 3.10. Botones Ejecutar y Cancelar

El botón *Ejecutar* da paso al segundo formulario, desde donde se comenzará la simulación. El botón *Cancelar* será usado para salir de la aplicación.

3.1.3. FORMULARIO SECUNDARIO

Como se dijo en el apartado anterior, tras pulsar el botón *Ejecutar* llamamos al este formulario Secundario, el cual tendrá el aspecto de la *Figura 3.11*:

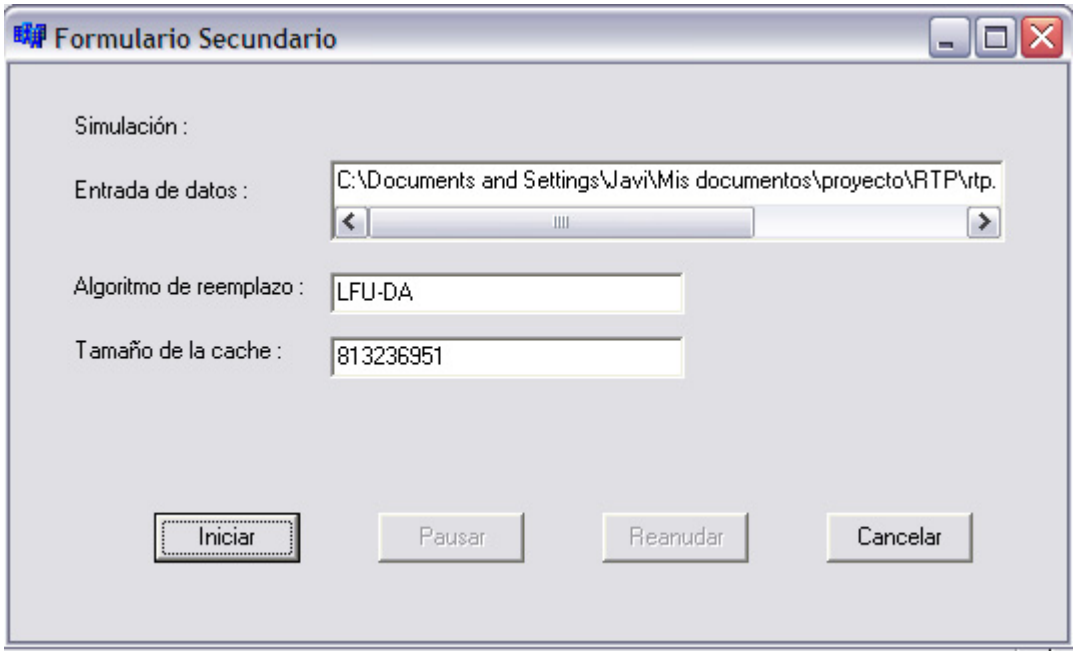


Figura 3.11. Formulario Secundario

Como puede observarse, el formulario estará compuesto de tres campos y cuatro botones que pasaremos a describir con detalle:

- **Entrada de datos:** Campo en que se mostrará en cada instante el archivo de entrada (campo *Entrada de datos*) que el simulador está procesando en cada momento. Si la simulación aún no ha comenzado, se mostrará el primer archivo que se computará.

- **Algoritmo de reemplazo:** Campo que aporta información acerca del algoritmo de reemplazo que se seleccionó en el *Formulario Principal*.
- **Tamaño de la cache:** Campo que hace referencia al tamaño de cache que se seleccionó en el *Formulario Principal.*

Existe un cuarto campo, sin título alguno. Éste aporta información del estado del simulación, así en principio aparece vacío, pudiendo presentar otros estados, como 'simulando' o 'simulación pausada'.

El próximo paso será describir qué función desempeñan cada uno de los botones de la zona inferior:

- **Iniciar:** comienza la simulación, con los parámetros descritos en los campos superiores.
- **Pausar:** este botón aparecerá inicialmente sombreado, y sólo se permitirá cuando la simulación este en ejecución. Con el conseguimos pausar temporalmente la ejecución de la simulación, sin perder los datos hasta ese momento conseguidos.
- **Reanudar:** este botón tan solo estará visible en caso de haber pausado la simulación. Con él se devolverá la simulación al estado de ejecución.
- **Cancelar:** este es un botón con dos estados posibles. Si lo pulsamos antes de ejecutar la simulación o mientras está en ejecución, se conseguirá salir del formulario secundario, cancelándose la ejecución de la simulación en curso, si esta se inició. Tras el término de una simulación, cambiará su nombre por el de **Resultados**, y nos permitirá salir del *Formulario Secundario* para poder observar los resultados obtenidos en el grupo *Resultado de la simulación,* del *Formulario Principal.*

Con todo, y tras el término de la simulación se podrá observar como el contenido del grupo *Resultado de la simulación,* del *Formulario Principal,* se completa con los resultados obtenidos en la misma, tal y como se aprecia en la *Figura 3.12*.

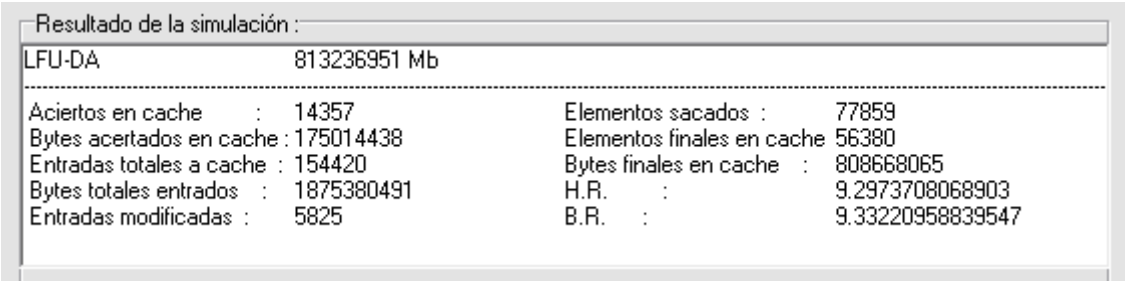


Figura 3.12. Grupo Resultado de la simulación II

Por último destacar que tras finalizar una simulación, aparecerá un mensaje por pantalla indicando el tiempo que se necesitó en completarla.

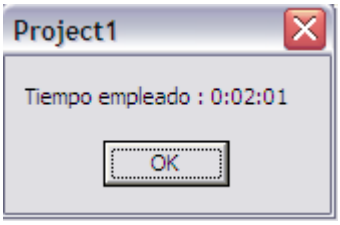


Figura 3.13. Tiempo empleado

3.2. IMPLEMENTACIÓN DE LAS POLÍTICAS

A continuación se realizará una breve descripción de cómo se han implementado las políticas de reemplazo, es decir, a menudo el comportamiento de ciertas políticas se hace muy poco eficiente, computacionalmente hablando, y es necesario el empleo de librerías especializadas de búsqueda u ordenación, así como el diseño, en ocasiones, de las políticas de manera que no se ajustan estrictamente a su descripción dada en el *Capítulo 2*, pero que consiguen unos resultados finales equivalente.

Para ello, se repasará, de nuevo, cada una de las políticas. Se realizará primero una breve descripción de cómo se comporta la política de manera teórica, para luego pasar a comentar cómo se ha realizado su diseño en el simulador, y en caso de requerirlo, el por qué; aunque antes, se comentarán aspectos generales que se han seguido para realizar los algoritmos, es decir, detalles de la programación que resultan comunes a todas las políticas.

Optimizar los algoritmos es absolutamente necesario, ya que, como se dijo en el capítulo anterior, el simulador tendrá que manejar muestras de entrada en las que aparecerán millones de documentos *Web*. Por tanto, ante cualquier política de reemplazo o tamaño de *cache* los tiempos de ejecución se harían interminables si no forzásemos esta optimización de los algoritmos para acelerar su finalización.

En caso de ser preciso, se mostrarán líneas de código en que aparezca alguna función interesante, a fin de comprender mejor el diseño del algoritmo.

Es necesario presentar un término desconocido hasta ahora, el de *documento modificado.* Este es un concepto similar al del documento acertado, pero con sutiles diferencias:

- **Documento Acertado:** Se entenderá como tal, los documentos que tengan el mismo identificador que alguno de los ya guardados, y cuyo tamaño sea exactamente el del documento ya almacenado, o bien, lo supere en más de un 5%, o bien, sea menor de un 5% inferior [8]. Es decir:

  Tamaño documento nuevo = Tamaño documento almacenado

  Tamaño documento nuevo < 0.95·(Tamaño documento almacenado)

  Tamaño documento nuevo > 1.05·(Tamaño documento almacenado)

- **Documento Modificado:** Será cualquier otro caso, es decir, documentos que produzcan un acierto y cuyo tamaño esté comprendido entre el ±5% del tamaño del objeto ya en memoria, será considerado como un corte de conexión [8]:

  0.95·(Tam.doc.almacenado)<Tam.doc.nuevo< 1.05·(Tam.doc. almacenado)

3.2.1. ASPECTOS GENERALES

Como se explicó en el apartado anterior, la aplicación se compone de dos formularios principales, la descripción de los mismos, desde el punto de vista de la programación, se realizará en el *apartado 3.3*; en este, se centrará la atención en presentar aspectos detallados de la programación de los algoritmos.

En primer lugar, ¿qué estructura se ha empleado para el almacenamiento de los documentos?

Se ha empleado una lista (*TList*), la razón es clara, es una estructura capaz de almacenar cualquier tipo de objeto, en este caso esos objetos serán información acerca de los documentos *Web,* e incorpora gran variedad de propiedades y funciones para insertar o eliminar elementos, ordenarlos, o acceder a los objetos almacenados. ¿Cómo se determinará si un objeto está o no almacenado? Y en caso de estarlo ¿dónde?

3.2.1.1. Búsqueda de objetos

Como ya se indicó anteriormente, es necesario optimizar tremendamente la simulación, por ello se optó por mantener también una estructura gemela a la lista, que almacena tan sólo los identificadores de los documentos (el identificador del documento

será un número que identificará de manera universal al documento en toda la *Web*), y que optimice su búsqueda, para identificar rápidamente si un objeto está o no almacenado.

Así, se empleo un *array de THashedStringList.* Estas son estructuras que emplean tablas *hash* internamente para agilizar el proceso de localización de elementos. El hecho de mantener un *array* de estas listas de búsqueda se debe a que de esta manera repartimos todos los objetos almacenados entre las diversas listas que compondrán el *array*. Así se insertará un identificador en la tabla que le corresponda en función de su último dígito (el *array* será de 10 listas de búsqueda), de manera que conseguimos que en cada lista el número de identificadores almacenados sea mucho menor, y así su búsqueda se agiliza.

El esquema de la estructura de datos comentada es el indicado en la *Figura 3.14*:

Id = '0'	***Id = '1'***	***Id = '2'***		***Id = '7'***	***Id = '8'***	***Id = '9'***
13200	**6501**		…….	**997**	**5498**	**99**
9800	**6601**				**3298**	**199**
200					**198**	
7900						
9400						

Figura 3.14. El array de listas de búsqueda

Al llegar un elemento nuevo se analizará su identificador, y se comprobará cual es su último dígito. Conseguido éste, se insertará en la lista de búsqueda correspondiente. En el *Código 3.1* se observa el pseudocódigo necesario para conseguir éste último dígito:

```
Id = Convertir(identificador);      // convertimos el identificador a un tipo entero
Dígito = Resto de dividir entre 10(Id); // digito será el último dígito del identificador
```

Código 3.1. Obtención del índice de lista de búsqueda

Otro mecanismo, para agilizar la búsqueda de objetos se basa en el estudio de una muestra de entrada típica. Hecho esto, se comprueba que de llegar un objeto cuyo identificador sea mayor que cualquiera de los ya insertados, ese objeto no estará en la *cache*. En caso contrario, habría que analizar si el objeto está o no. Todo esto se realiza tal y como muestra *Código 3.2*:

```
si (Identificador máximo >= objeto _ identificador) {
        Esta = Comprobar en lista correspondiente((objeto_ identificador);
}
en otro caso {
        Identificador máximo =  objeto_ identificador); Objeto en cache = falso;
 }
```

Código 3.2. Optimización de la búsqueda de un documento

Pero, ¿si es posible que un elemento esté en la cache, cómo se localiza en la lista de búsqueda correspondiente?, y ¿para que tener la lista original, si ya guardamos los objetos en las listas de búsqueda?

Si, efectivamente, el identificador del nuevo objeto es menor al máximo de los identificadores que tenemos almacenados, será posible que ese nuevo elemento ya esté en la *cache.* Para verificarlo se hace una llamada a una función de búsqueda llamada ***Find***. Este algoritmo de búsqueda devuelve un **booleano** indicando si el identificador está o no, y el índice en caso necesario. Para el empleo de este método es necesario que las listas estén ordenadas y que se inserten los identificadores con el método ***Add***, es decir, por la última posición de la lista de búsqueda.

Pero estas listas de búsqueda presentan dos problemas fundamentales. El primero es que como ya se ha dicho, es necesario que estén ordenadas de manera que se facilite la búsqueda con el método ***Find,*** y esto choca diametralmente con la ejecución de la mayoría de las políticas, que precisan un orden muy concreto en la lista**.** Otro de los inconvenientes es que en estas listas sólo se insertarán los identificadores de los elementos, y no el resto de los campos. Campos que resultan absolutamente imprescindibles para completar el algoritmo, tal y como se aprecia en la *Figura 3.15*:

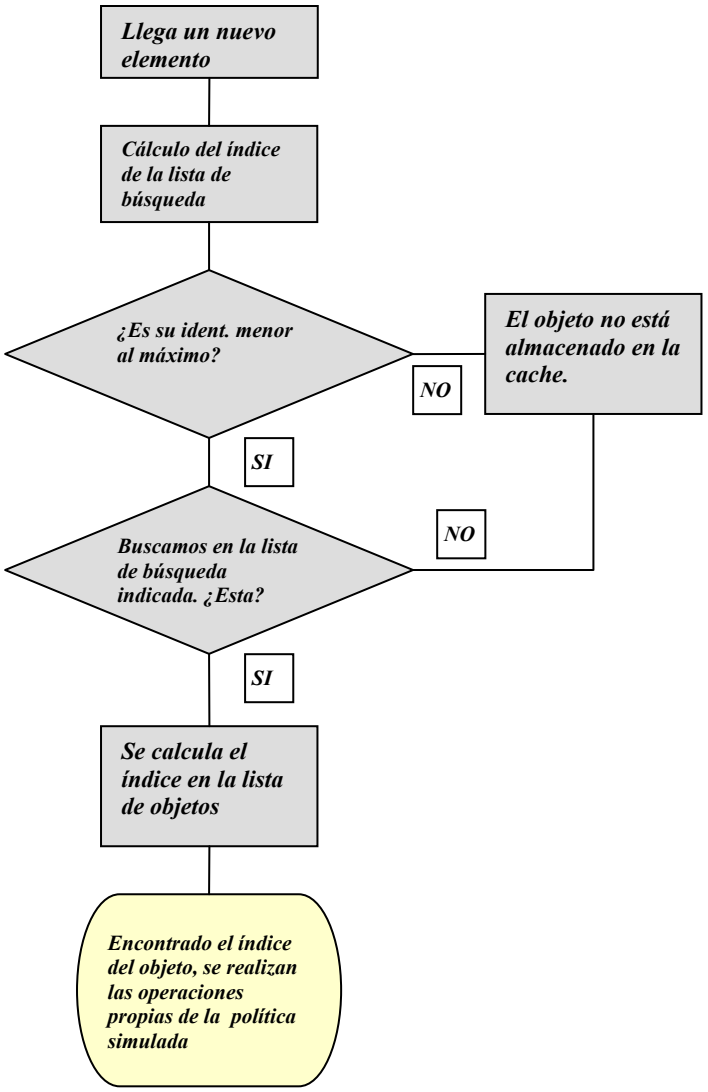


Figura 3.15. Diagrama de flujo de la búsqueda

3.2.1.2. Liberar memoria

Si suponemos ahora que la simulación ha finalizado, ¿Qué habría que hacer con todo este sistema de listas?

Por supuesto, sería necesario eliminar todas estas estructuras de datos empleadas. Por lo tanto, habría que eliminar todos y cada uno de los elementos que las componen. Para estructuras que tengan objetos asociados a cada posición, será necesario destruir también esos objetos de manera explícita.

El pseudocódigo utilizado para liberar memoria es el mostrado en *Código 3.3*:

listas de búsqueda

```
para (lista de búsqueda 0)  hasta (lista de búsqueda 9){
      lista = lista de búsqueda actual;
      para (primer elemento(lista)) hasta (ultimo elemento(lista)){
            eliminar(elemento);
      }
      Eliminar(lista);
}
```

lista de objetos

```
para (primer elemento(lista)) hasta (ultimo elemento(lista)){
       eliminar(elemento);
}
Eliminar(lista);
```

Código 3.3. Liberación de memoria

3.2.1.3. Salida a fichero

Existe la posibilidad, de que una vez terminada la simulación, cierta información fundamental desprendida de esta se muestre por fichero, es decir, se crearán dos tipos distintos de ficheros de texto.

En uno aparecerá la misma información que se detalla en el cuadro de lista *Resultado de la simulación.* Esto es, número de acierto, modificaciones, elementos sacados, *HR*, *HBR*, etc.…

En el otro se mostraría el contenido de la *cache* en el instante mismo de finalizar la simulación. Sería una lista de líneas de texto (una por documento almacenado), que contendrían la información relativa a cada uno de los campos propios de los objetos. Por supuesto la posición relativa que ocupasen estas líneas, sería equivalente a la posición que ocuparon en la *cache.* Siendo la primera línea la correspondiente al objeto de índice 0.

El proceso de escritura en fichero es el mismo en ambos casos. Primero se crea un array con todos los datos correspondientes a los resultados, o a los campos de un objeto. Luego, por cada uno de los elementos de este *array*, se crea una cadena de

caracteres del mismo tamaño, y se copia el contenido del elemento del *array* correspondiente (método *strcpy()).* Por último, no hay más que copiar cada uno de los caracteres de esta nueva cadena al fichero, para ello se emplea el método *put().*

A continuación se mostrarán el contenido de un fichero de resultado y otro de contenido, resultantes de la realización de una simulación con una muestra de entrada pequeña:

Salida de contenido

1085356818.305 390 4 2 1 3 0 |0
1085356817.424 396 3 2 1 3 0 |1
1085356867.246 27357 20 2 3 1 0 |2
1085356864.925 2302 19 2 3 1 0 |3
1085356864.905 2280 18 2 3 1 0 |4
1085356863.775 25369 17 2 3 1 0 |5
1085356862.705 2270 16 2 3 1 0 |6
1085356862.685 1479 15 2 3 1 0 |7
1085356862.657 11718 14 2 3 1 0 |8
1085356859.635 15824 13 4 4 1 0 |9
1085356856.134 3238 12 2 3 1 0 |10
1085356853.794 3064 11 2 3 1 0 |11
1085356851.794 2978 10 2 3 1 0 |12
1085356850.864 2560 9 2 3 1 0 |13
1085356849.334 2996 8 2 3 1 0 |14
1085356846.244 457575 7 5 4 1 0 |15
1085356824.583 24020 6 4 4 1 0 |16
1085356819.753 41217 5 4 4 1 0 |17
1085356814.514 1482 2 2 1 1 0 |18

Salida de resultados

LRU 81323695 | Aciertos : 4 | Bytes Acertados : 1572 |Entradas Totales : 24 | Bytes Totales : 630545 | Elementos Finales en Cache : 21 | Bytes de cache ocupados : 629429 | HR. : 16.6666666666667 | B.R.: 0.249308138197908 |Entradas Modificadas: 0 |Elementos Sacados: 0

3.2.1.4. Diagrama general

Por último, la *Figura 3.16* corresponde a un diagrama de flujo con el comportamiento general del simulador, independientemente de la política implementada:

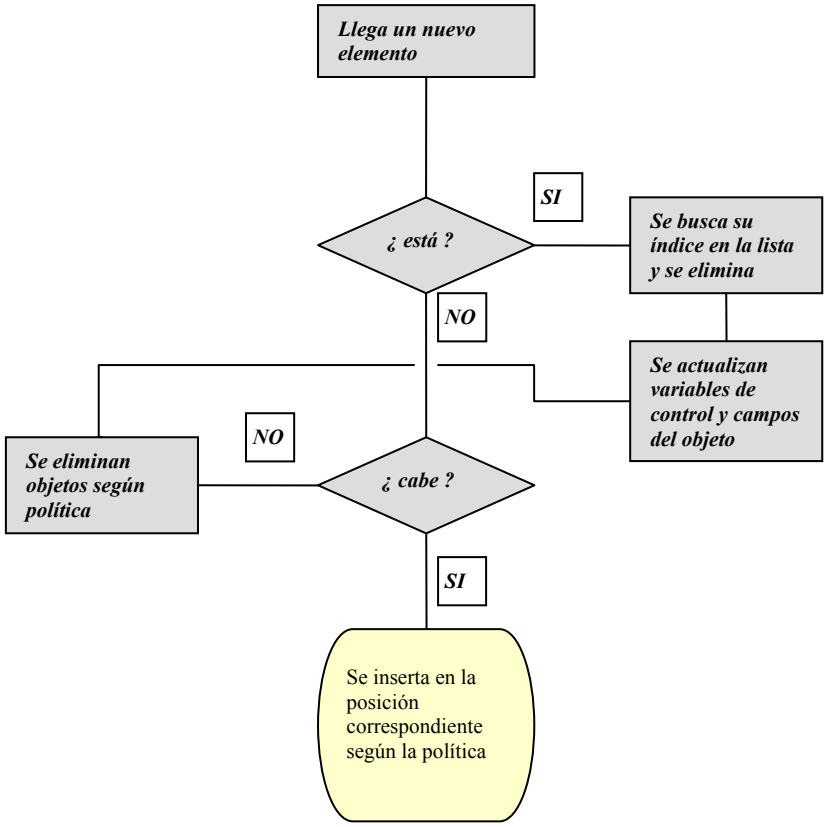


Figura 3.16. Diagrama de flujo del algoritmo generalizado

3.2.2. LRU

3.2.2.1. Descripción de la política

Como ya se estudió en el Capítulo 2, *LRU* es una política tremendamente simple, que inserta documentos nuevos en la cabeza de la *cache* (posición superior), y en caso de necesitarse liberar espacio, se comenzará extrayendo elementos comenzando por el último de la lista (posición inferior).

En caso de producirse un acierto (o una modificación), el documento se reubicará en la cabeza del sistema.

3.2.2.2. El algoritmo implementado

En principio, la programación del algoritmo parece bastante evidente, insertaremos en la lista en la posición inicial (0), mientras que extraeremos de ella

comenzando por la última posición (*lista->Count -1*).

Pero, se puede demostrar que existen maneras más eficientes de implementarlo. Para comprender todo esto analicemos como funcionaría el algoritmo de implementarlo de la manera más evidente. Para su mejor compresión se mostrará en *Código 3.4* el pseudocódigo equivalente:

```
si (objeto esta){
    lista_buscamosindice(objeto);
    lista_eliminamos(indice);
}
mientras (no cabe(objeto)){
    lista_eliminamos(ultima posición);
}
lista_insertamos(objeto en posición inicial);
```

Código 3.4. Algoritmo LRU teórico

En caso de llegar un nuevo elemento, o producirse un acierto o modificación, éste se insertaría en la posición inicial de la *lista,* para ello se emplea la siguiente instrucción:

```
Lista->Insert(0,objeto);
```

¿Cómo se comporta? Esta instrucción desplazaría todos los elementos almacenados una posición hacia abajo para así insertar en la primera posición al nuevo elemento, hecho que se describe en la *Figura 3.17*:

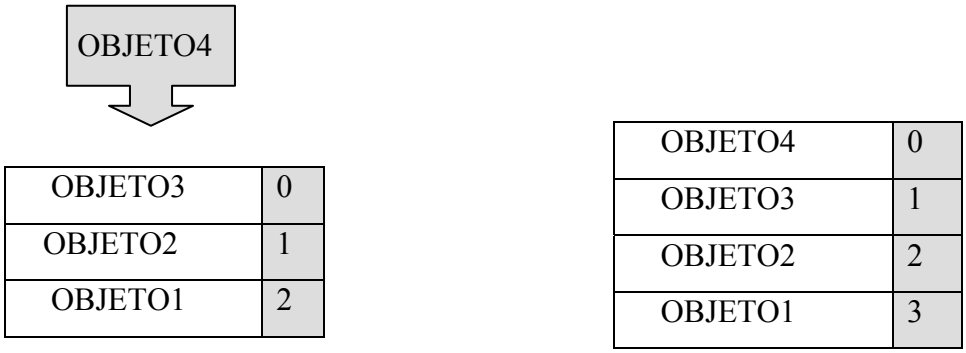


Figura 3.17. Funcionamiento del método Insert()

A simple vista, este parece un comportamiento sensato, pero si manejamos una lista con millones de elementos almacenados, la llegada de un nuevo objeto implicaría

tener que desplazar de posición todos los documentos. Lo cual resulta tremendamente ineficaz. ¿Cómo solucionarlo?

La solución propuesta es la siguiente, insertar en la posición inicial es tremendamente costoso, sin embargo, insertar en la última posición de la lista sería lo más eficiente, ya que no habría que desplazar ningún objeto, tan sólo insertar el nuevo al final de la lista.

LRU requiere insertar en la cabeza de la lista, pero el concepto 'cabeza' es ciertamente relativo, podríamos considerar que esa ultima posición fuera la cabeza, mientras que la posición 0 fuera la cola de la lista, es decir, de cierta manera invertimos la lista. De esta manera, la lista estaría igualmente ordenada, pero invertida, lo cual no es importante, ya que los resultados obtenidos serían los mismos.

De esta manera conseguimos un algoritmo equivalente, pero mucho más eficiente. ¿Cómo podría mejorarse? Hay que pensar que al invertir la lista se ha resuelto el problema de la inserción de elementos, pero cuando haya que eliminar, habrá que hacerlo comenzando por la posición 0, es decir, se traslada el problema de tener que mover todos los elementos al eliminar (mucho menos traumático).

Podríamos mejorarlo si eliminásemos este efecto ligado al hecho de eliminar objetos, para ello lo que se hizo fue evitar tener que eliminar objetos, es decir, establecer una variable que indicase donde comenzaba la lista, sin que se tomaran a cuenta los objetos anteriores. De tal manera si hubiese que eliminar un objeto, simplemente se incrementaría este contador. Con lo que se consigue que se supriman las eliminaciones, y se comprueba experimentalmente que los tiempos de ejecución mejoran sensiblemente (dado que el número de eliminaciones por simulación es mucho menor al número de inserciones, la mejora conseguida es mucho menor que la que se alcanzó al modificar la manera de insertar).

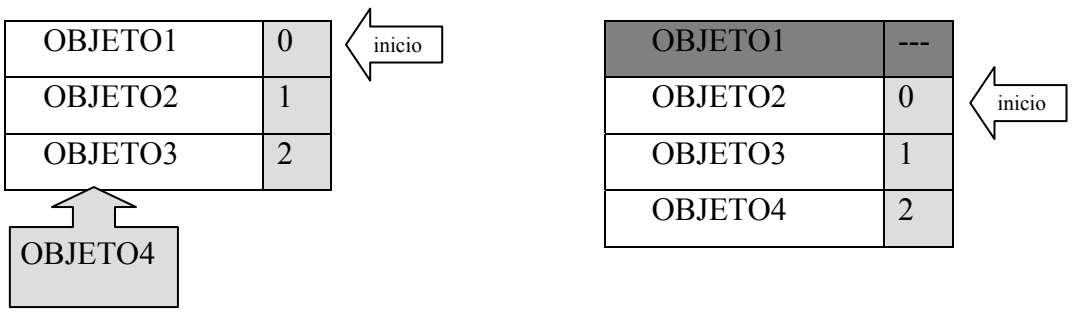


Figura 3.18 Descripción del nuevo método de inserción

Esta nueva manera de insertar arroja un problema. Para localizar un objeto en la lista se emplea un método ya implementado:

indice = lista->IndexOf(objeto);

este método busca el objeto de entre toda la lista, pero ahora que realmente no extraeremos objetos, el método buscará de entre toda la lista sin tener en cuenta el índice que apunta al comienzo válido de nuestra lista. Por ello se elaboró un nuevo procedimiento de búsqueda:

indice = lista->Index(objeto,referencia);

El cual tiene un comportamiento similar al anterior, solo que comienza a buscar tomando como inicio de la lista el dado por la variable de referencia.

Con todo conseguimos un algoritmo rápido y robusto que simula perfectamente el comportamiento de la política *LRU.*

3.2.3. LFU

3.2.3.1. Descripción de la política

El funcionamiento de esta política consistía en mantener una lista ordenada por el contador de referencia de los objetos.

Así en la zona superior de la lista se encontrarán los objetos de mayor contador, mientras que en la zona inferior estarán los de menor contador de referencia. Al llegar un nuevo elemento insertaremos de manera ordenada en función del contador de referencia. Al
producirse un acierto o una modificación, recalcularemos su contador, y reinsertaremos el objeto en función de ese nuevo contador.

En caso de necesitar liberar espacio, eliminaremos objetos comenzando por el de menor contador (final de la lista).

Hay que hacer notar que mientras un acierto provoca que se incremente el contador, una modificación consigue reiniciar el contador del objeto a uno (es como si el elemento fuera nuevo).

3.2.3.2. El algoritmo implementado

En este caso, la inserción no se realiza necesariamente en la cabeza de la lista, sino que se insertará en la lista de manera ordenada, en función del contador de referencia del documento a insertar. De hecho, dado que la lista se ordenará por el contador de referencia, elementos nuevos tendrán un contador igual a 1, por lo que se deduce que elementos nuevos se insertarán en la zona inferior de la misma.

Tan sólo en caso de producirse un acierto, caso en que podríamos tener un contador elevado, la inserción se produciría en la zona superior.

Por tanto, dado que las inserciones se hacen mayoritariamente en la zona inferior, no es necesario aplicar la técnica puesta en práctica al implementar el algoritmo *LRU.*

A la hora de llegar un objeto, se evalúa si ya está, o no, en *cache*. De ser así, se analiza si es un acierto (se incrementa el contador) o una modificación (contador igual a la unidad). Se elimina el objeto de la lista (por supuesto también de la lista de búsqueda correspondiente) y se reinserta en su posición correspondiente en función del nuevo contador. En caso de que el objeto nuevo no estuviese en *cache*, se chequea si cabe y de ser

así se inserta en la posición correspondiente; sin no cupiese habría que eliminar espacio hasta que lo hiciese.

En principio, el diseño de este algoritmo no dista mucho del comportamiento teórico de la política, el único proceso destacable es como el algoritmo busca la posición ordenada que le corresponde al objeto a insertar en la lista. Para ello se diseñó un procedimiento de búsqueda, cuyo esquema en pseudocódigo se muestra en *Código 3.5* a continuación:

```
superior = ultima posición;
centro = inferior = 0;
mientras (no_encontrado){
        centro = (superior+inferior)/2;
        si (objeto_contador < lista(centro)_contador){
            inferior = centro - 1;
        }
        o si (objeto_contador > lista(centro)_contador){
            superior = centro + 1;
        }
        en otro caso{
```

```
            encontrado;
        }
}
```

Código 3.5. Búsqueda binaria

este método de búsqueda es el conocido ***método de búsqueda binario,*** con él implementaremos una búsqueda de la posición del nuevo objeto de manera eficiente.

3.2.4. LFU-DA

3.2.4.1. Descripción de la política

Esta política es tremendamente similar a la *LFU,* la única variación radica en la inserción de un factor de envejecimiento para así evitar la polución de *cache*.

Resumiendo, los objetos se insertan en función de su contador de referencia. Los elementos de mayor contador, más populares, estarán en las primeras posiciones de la lista. Los objetos menos populares ocuparán las últimas posiciones y serán los elementos más susceptibles de ser eliminados. Al producirse una eliminación se actualizará el valor del factor L de envejecimiento al del contador de referencia de este elemento eliminado.

Si se produce un acierto, el contador se incrementa, pero esta vez con el valor de la constante de inflación L. Si se produce una modificación el objeto modificado pasará a tener un contador de referencia igual a esta constante L. Tras esto, se reubica el objeto según su nuevo contador de referencia.

3.2.4.2. El algoritmo implementado

La implementación de esta política es exactamente igual a la de la *LFU,* de hecho comparten la mayoría de las líneas de código.

La única diferencia radica en la necesidad de mantener una referencia al valor del contador de referencia de último elemento extraído. Así, el pseudocódigo asociado corresponde al expuesto en *Código 3.6*:

```
mientras (no cabe(nuevo objeto)){
        L = contador_objeto(ultima posición);
        lista_eliminar(ultima posición);
}
```

Código 3.6. Actualización del factor L

y a la hora de insertar el nuevo elemento(o el acertado/modificado):

```
objeto_ contador = objeto_ contador_ + L
indice =  buscar_indice(lista,objeto);
lista_insertar(indice,objeto);
```

Código 3.7. Cálculo del índice de inserción de un nuevo objeto

Donde la llamada a la función *buscar_indice,* busca la posición del nuevo objeto en la lista según su contador de referencia (método descrito en el apartado anterior).

3.2.5. GDS

3.2.5.1. Descripción de la política

En esta política, los elementos se ordenan en la *cache,* en función de su campo clave. Esta clave se calculará como el cociente entre la función coste y el tamaño del objeto. Cada vez que se produzca una eliminación, las claves de los elementos en la lista se modificarán, restándoles la clave del elemento eliminado. En caso de producirse un acierto o una modificación, la clave del elemento retornará a su valor original y el objeto será reubicado en función de este nuevo valor de clave.

Por último, hay que recordar que se implementan dos funciones coste típicas.

$$\cos te(1) = 1$$
$$\cos te(p) = \left(2 + \frac{tamaño\_objeto}{536}\right) \qquad \textbf{Ecuación 3.1}$$

3.2.5.2. El algoritmo implementado

El algoritmo simula fielmente la descripción dada de las políticas, inserta en la lista en función del valor calculado del campo clave. Al eliminar, extraeremos objetos comenzando por el de menor clave, es decir, el que ocupe la última posición de la lista.

La única diferencia radica en la manera en que realizan el envejecimiento de los documentos. La política indica que cada vez que se elimine un objeto, se modificarán las claves de todos los objetos almacenados en la lista.

Es evidente, que esta es una operación muy ineficiente, computacionalmente hablando, ya que en caso de tener una lista con gran cantidad de elementos, cada vez

que se produzca una eliminación, es decir, bastante a menudo, habría que referenciar todos y cada uno de los objetos almacenados, para modificar uno de sus campos.

¿Cómo se podría realizar una operación equivalente, pero más eficiente? lo que busca el algoritmo es envejecer la clave de los documentos ya memorizados, pero en lugar de disminuir estos, se podría recompensar al nuevo objeto aumentando su clave en la misma medida. De esta manera se conseguiría que la relación entre en nuevo objeto y el resto de los insertados se mantuviera constante, o lo que es lo mismo, el objeto se insertará en la misma posición que según la política original.

El único inconveniente de realizar este procedimiento es que, aún cuando los elementos se insertarán en la misma posición, y los resultados finales sean los mismos, la clave de los objetos finales no coincidirán en el algoritmo original y en el implementado.

Este no será realmente un problema de importancia, ya que el campo clave no aporta ninguna información a la simulación, su única función es asegurar que los elementos se insertan en su lugar correspondiente cada vez. Y esa es una premisa que se cumple absolutamente.

Por último, destacar, que estas familias de algoritmos emplean un método de búsqueda de los índices con que insertar en la lista, equivalente al descrito en *Código 3.5* (en este caso, buscarán el índice en función de las claves):

3.2.6. GDSF

3.2.6.1. Descripción de la política

Las políticas *GDSF*, tiene el mismo comportamiento de las políticas *GDS*, tan sólo que incorporan el factor de la frecuencia a la hora de insertar objetos.

Así, cada vez que llegue un objeto nuevo, o se produzca un acierto o una modificación, se recalculará el valor del campo clave del objeto y se ubicará en la lista.

Con todo, la clave se calculara como:

$$clave = \left(L + \left(contador \cdot \frac{\cos te}{tamaño\_objeto} \right) \right) \qquad \textbf{Ecuación 3.2}$$

el valor del factor L se actualiza con la clave del último elemento eliminado, de manera que así se controlará el envejecimiento de la *cache*. En este caso el factor L se añade a la clave de los objetos.

Esta familia de políticas se ha implementado empleando las dos funciones de coste típicas:

$$\cos te(1) = 1$$
$$\cos te(p) = \left(2 + \frac{tamaño\_objeto}{536}\right) \qquad \textbf{Ecuación 3.3}$$

3.2.6.2. El algoritmo implementado

La implementación del algoritmo sigue fielmente la descripción original de la política. Por tanto no existen aspectos destacables del mismo.

Sólo comentar que emplea la función de búsqueda de la posición en *cache* del nuevo objeto, implementada en el algoritmo *GDS.*

Dado que el cálculo de la clave, depende directamente del valor del contador de referencia del objeto en cuestión, se realizará una descripción, empleando pseudocódigo de cómo se actualiza este campo. Esta descripción es igualmente válida para todos y cada uno de los algoritmos que empleen el contador de referencia como argumento.

```
si (objeto está)){
   si (objeto = acierto){
       contador_objeto++;
   }
  o si (objeto = modificación){
       contador_objeto = 1;
  }
}
en otro caso{
       contador_objeto  = 1;
}
```

Código 3.8. Modificación del contador de referencia

Dado que en este caso el factor L se suma a la clave de elementos nuevos, acertados o modificados, los campos clave resultantes tras ejecutar el algoritmo implementado coinciden completamente con los obtenidos de manera teórica, es decir, no se da la circunstancia exhibida en el apartado anterior.

3.2.7. GD*

3.2.7.1. Descripción de la política

Estas serán las últimas políticas *Greedy Dual* implementadas. Su funcionamiento será idéntico al del algoritmo *GDSF,* la única variación será la manera en que se calculará el campo clave, con que insertaremos de manera ordenada en la lista.

En este caso la clave se calcula tal y como se indica en la *Ecuación 3.4*:

$$clave = \left(L + \left(contador \cdot \frac{\cos te}{tamaño\_objeto} \right)^{\frac{1}{\beta}} \right) \qquad \textbf{Ecuación 3.4}$$

Siendo la función de coste, la mostrada en la *Ecuación 3.5*:

$$\cos te(1) = 1$$
$$\cos te(p) = \left(2 + \frac{tamaño\_objeto}{536} \right) \qquad \textbf{Ecuación 3.5}$$

3.2.7.2. El algoritmo implementado

Como se dijo en el apartado anterior, el diseño de este algoritmo no presenta novedades importantes, comparado con su política original.

Emplea el mismo método con el que buscar índices en función del campo clave, la única característica relevante es el empleo de la función *pow,* rutina importada de la librería matemática, que se utiliza para elevar un número a otro.

3.2.8. RAND

3.2.8.1. Descripción de la política

Con esta, comienza el estudio de la implementación de las políticas de naturaleza aleatoria. La característica principal radica en el hecho de que los elementos a eliminar se elegirán de manera aleatoria. Debido a esta naturaleza aleatoria, nada más se dice acerca de cómo insertar o qué hacer en caso de acierto o modificación de un objeto existente.

3.2.8.2. El algoritmo implementado

Pues bien, como se ha dicho, la particularidad fundamental de esta política radica en el hecho de eliminar objetos eligiéndolos en la lista de manera aleatoria. Pero, ¿cómo implementaremos esta aleatoriedad?

La selección aleatoria de objetos se realizará por medio del método *random*. Su sintaxis es la siguiente:

indice_aleatorio = random(indice_maximo +1);

es decir, *random* devolverá un entero comprendido entre 0 y el número pasado como parámetro a la función menos 1. Así si hacemos una llamada a la función como la anterior, conseguiremos un índice aleatorio válido de entre todos los de la lista. *Random* es una función importada desde la librería matemática.

Pero esta función es necesaria inicializarla, para ello se ejecuta el comando *Randomize()* cada vez que se vaya a iniciar un proceso de selección aleatoria. Este comando proporciona un valor aleatorio inicial, a partir del reloj del sistema, desde el cual la función *random* construya la secuencia aleatoria total.

En caso de no proporcionarse este valor inicial, *random* generaría siempre la misma secuencia de índices aleatorios.

Por último comentar que con esta función se generan números aleatorios con igual probabilidad, es decir, la distribución de probabilidad es constante para todos los índices.

Ya es conocido cómo se consiguen generar índices aleatorios válidos. Ahora bien, se conoce la metodología seguida para eliminar objetos, pero, ¿Cómo se inserta? ¿qué hacer en caso de producirse un acierto? ¿por qué no son importantes estos factores que en anteriores políticas eran determinantes?

Será más interesante comenzar por el final. La manera en que insertaremos en la lista, o cómo se reubicarán objetos acertados no es importante dado que eliminamos de manera aleatoria. Ya que la eliminación es impredecible, no tiene sentido mantener la lista ordenada de ninguna forma, es decir, no serviría de nada mantener un criterio de inserción para que los documentos más importantes permanezcan en cierta posición, si estos se eliminarán con la misma probabilidad que los menos importantes, independientemente de donde estén ubicados.

Al producirse un acierto o llegar un elemento modificado, se podría dar una justificación similar.

Dicho lo cual, es necesario deducir la manera más eficiente de insertar los elementos. Ya que la posición donde se haga es indiferente, resulta bastante evidente pensar, que tras lo comentado en el apartado *LRU,* la inserción más eficiente sería la conseguida con el método *Add*, es decir, insertar en la última posición de la lista, para que así no sea necesario desplazar ninguno de los objetos ya almacenados.

En cuanto al hecho de producirse un acierto o una modificación, la primera idea es clara, repetir lo dicho para la inserción de elementos nuevos, reubicar al final de la lista. Pero incluso más eficiente resulta no reubicar estos elementos, es decir, si se produce un acierto el elemento acertado no se toca, se mantiene inalterable en la misma posición. Dado que no será necesario modificar ninguno de sus campos (contador de referencia, clave), ésta es sin duda la manera más rápida de tratar estos objetos.

3.2.9. HARM

3.2.9.1. Descripción de la política

Como sucedió en el apartado anterior la característica principal de esta política radica en la aleatoriedad con que se eliminan objetos cuando es necesario liberar espacio de la *cache.*

La única diferencia radica en el hecho de que esta política no asume que todos los objetos son igualmente probables a la hora de eliminarse, sino que esta probabilidad es inversamente proporcional al cociente $\frac{\cos te}{tamaño}$. Dado que se ha supuesto una función coste constante, e igual a la unidad, la probabilidad de que un objeto sea seleccionado para ser eliminado es directamente proporcional al tamaño de los objetos.

Igualmente, no se dice nada más acerca de la manera en que se inserta o se reubican los elementos

3.2.9.2. El algoritmo implementado

Por tanto, el aspecto más relevante del diseño de este algoritmo es sin duda el ser capaces de generar números aleatorios en función del tamaño de los objeto.

Para ello, se ha resuelto emplear el método *RandG.* Éste produce números aleatorios con una distribución Gaussiana sobre la media.

La idea es la siguiente, ordenar la lista en función del tamaño de los objetos, el primer elemento (índice 0) será el mayor, mientras que el último será el de menor

tamaño. Tras lo cual conseguiremos una distribución de tamaño tal y como se muestra en la *Figura 3.19*:

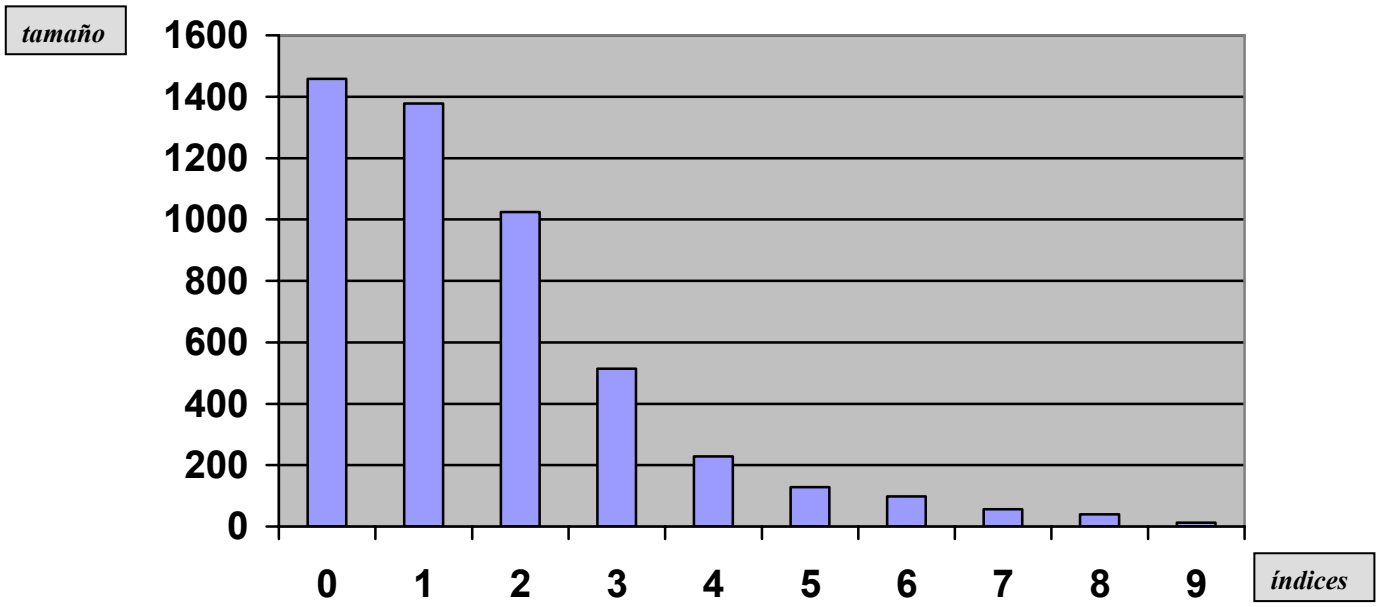


Figura 3.19. Distribución de los tamaños de objetos según índices

Se puede observar como esta distribución se parece tremendamente a una Gaussiana, por tanto si ajustamos correctamente la media y desviación típica de los elementos que insertamos en la lista, se podrá conseguir el diseño de una función de Gauss.

Es conveniente recordar de manera somera cómo se calculan los factores media y desviación típica de una secuencia de valores x_i, todo esto se muestra en la *Ecuación 3.6*:

$$media = \bar{x} = \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)$$

$$\text{var}ianza = S^2 = \frac{1}{n}\sum_{i=1}^{n}\left(x_i - \bar{x}\right)^2 = \frac{1}{n}\sum_{i=1}^{n}\left(x_i^2 - 2x_i\bar{x} + \bar{x}^2\right)$$

Ecuación 3.6

$$= \frac{1}{n}\sum_{i=1}^{n} x_i^2 - 2\bar{x}\frac{1}{n}\sum_{i=1}^{n}\left(x_i\right) + \frac{1}{n}n\bar{x}^2 = \frac{1}{n}\sum_{i=1}^{n} x_i^2 - 2\bar{x}^2 + \bar{x}^2$$

$$= \left(\frac{1}{n}\sum_{i=0}^{n} x^2{}_i\right) - \bar{x}^2 \longrightarrow desviacion\_tipica = \sqrt{\text{var}ianza}$$

a continuación, se presentará en *Código 3.9* cómo se ha realizado el cálculo de estas variables:

```
media = media + objeto_tamaño;
varianza = varianza + (objeto_tamaño)^2;
si (objeto está){
   media = media - objeto_tamaño;       //si el objeto ya está, no habría que volver a
   varianza = varianza - (objeto_tamaño)^2;// añadir el tamaño en la media y varianza
}
en otro caso{
    media2 = media / (lista_longitud);
   varianza2 = (varianza / (lista:longitud)) – media2^2;
    desviación = raiz(varianza2);
    mientras (no cabe){
            indice = RandG(media,varianza);
             lista_eliminar(índice);
    }
}
```

Código 3.9. Cálculo de media y varianza

Así, manteniendo estas variables *media* y *varianza* constantemente actualizadas, se podrá definir una función Gaussiana que represente perfectamente la distribución de los objetos en la lista, y por tanto, podremos llamar a la función *RandG* que proveerá índices aleatorios según esta función, es decir, se obtendrán índices aleatorios válidos que tendrán una probabilidad directamente proporcional al tamaño de cada objeto.

Solucionada esta faceta del algoritmo, solo habría que destacar cómo se realiza la inserción o se reubican los objetos en la lista.

Como sucedió en el *apartado 3.2.8*, ya que el método de extracción es aleatorio (aunque en cierta manera dependa del tamaño de los documentos), la manera de insertar o reubicar nos resulta indiferente.

De tal manera, se optó por insertar elementos al final de la lista, ya que, como se ha indicado en numerosas ocasiones, esta es la manera menos costosa computacionalmente hablando. En cuanto a los aciertos o modificaciones, lo más eficiente es conservar dichos objetos en la posición original, es decir, no reubicarlos y mantenerlos inalterables.

De esta forma, conseguimos un algoritmo *HARM* muy eficiente y perfectamente válido.

3.2.10. LRU-C

3.2.10.1. Descripción de la política

Esta no es más que una variante aleatoria de la popular política *LRU.* Por tanto, como sucedía en la *LRU,* se eliminarán los elementos menos recientemente referenciados.

El concepto de aleatoriedad se introduce ya que al producirse dichos aciertos estos objetos se reubicarán a la cabeza pero con una probabilidad dada, es decir, no siempre se reordenarán estos objetos, habrá veces en que no se realice ninguna operación. Esta probabilidad se calculará como el cociente entre el coste del elemento, y el coste máximo de entre todos los que hay en la lista.

En esta política el coste asociado a un objeto se calculará como indica la *Ecuación 3.7*:

$$\cos te(p) = \left(2 + \frac{tamaño\_objeto}{536}\right) \qquad \textbf{Ecuación 3.7}$$

3.2.10.2. El algoritmo implementado

Como se ha dicho, la implementación de este es prácticamente idéntica a la del algoritmo *LRU, s*e insertará al final de la lista y se eliminará comenzando por el primer elemento de la lista, es decir, se utiliza la técnica de la lista invertida para conseguir el mismo resultado, mejorando notablemente el rendimiento del algoritmo.

La particularidad más notable se encuentra en el hecho de que haya que decidir si un elemento hay que reubicarlo o no. Para ello se calcula el coste de cada objeto que entre en la lista, y se evalúa si este coste es mayor que el máximo de referencia, en caso de serlo, se actualiza el valor del máximo de referencia al del coste del nuevo objeto:

```
si (objeto_coste > maximo){
   objeto_coste = maximo;
}
probabilidad = (objeto_tamaño/máximo);
```

Código 3.10. Actualización del coste

Cada vez que haya que reubicar, se evalúa el cociente entre el coste del objeto y el coste máximo, obteniendo un número entre 0 y 1, que representa la probabilidad en tanto por uno de que el elemento sea movido al final de la lista.

Pero, ¿cómo se decidirá si se cumple la probabilidad o no?

Para ello, se genera un número aleatorio entre 0 y 100 a través de la función *random*, y se evalúa si este número supera a la probabilidad de cambio en tanto por cien (en tal caso no se hará nada), o no (el objeto se reubicará al final de la lista).

Así el pseudocódigo asociado a esta función de decisión se muestra en *Código 3.11*:

```
Funcion Decidir(probabilidad);
{
  Reubicar = falso;
  indice aleatorio = random(101) ;
  si (indice aleatorio < (probabilidad)){
     Reubicar = verdadero;
   }
 devolver  Reubicar;
}
```

Código 3.11. Función Decidir()

3.2.11. LRU-S

3.2.11.1. Descripción de la política

Una nueva variante aleatoria de la política *LRU.* Se elimina el elemento menos recientemente referenciado.

El factor de aleatoriedad se introduce ya que, tal y como sucedía en el algoritmo *LRU-C* no siempre que se produzca un acierto o modificación el objeto será desplazado, sino que existirá la probabilidad de que se quede tal y como estaba.

Así, la probabilidad de que el objeto sea reubicado viene dada por el cociente entre el tamaño menor de todos los objetos almacenados en la lista y el tamaño del objeto en cuestión. Se conseguirá así un número entre 0 y 1 que indica esa probabilidad en tanto por uno.

3.2.11.2. El algoritmo implementado

El algoritmo es idéntico al *LRU-S*, la única variación radica en la manera en que se calculará la probabilidad existente de mover los objetos acertados.

Para ello es necesario mantener una referencia con el valor del tamaño del objeto más pequeño existente en la cache, y que se actualice con cada entrada:

```
si (objeto_tamaño < minimo){
    minimo = objeto_tamaño;
}
probabilidad = (minimo /objeto_tamaño);
```

Código 3.12. Actualización del tamaño mínimo

A partir de esta probabilidad ya calculada, sólo habría que llamar a la función *Decidir,* descrita en *Código 3.12*, que determinará si se cumple la probabilidad para reubicar el objeto acertado/modificado, o no.

3.2.12. RRGVF

3.2.12.1. Descripción de la política

Este es el último algoritmo de los implementados en este proyecto. *RRGVF* marca su complejidad a la hora de elegir qué objetos se han de eliminar.

De tal manera, esta estrategia selecciona N elementos aleatoriamente de la *cache* y toma los M menos útiles, guardándolos en memoria. Eliminaremos elementos de entre estos M (comenzando por los menos útiles) hasta liberar el espacio suficiente. En siguientes reemplazos, N-M nuevos objetos son seleccionados de la lista y se seleccionan los M menos útiles de entre estos N-M y los M previamente guardados. Eliminaremos elementos de entre estos M (comenzando por los menos útiles), hasta liberar el espacio suficiente. Así sucesivamente.

La utilidad de un objeto vendrá determinada por el número de veces que ha sido referenciado, es decir, el valor de su contador de referencia. La inserción, o reubicación de elementos acertados o modificados, no es determinante en la especificación del algoritmo.

3.2.12.2. El algoritmo implementado

Como se acaba de indicar, dada la naturaleza aleatoria del algoritmo, no es importante como introduzcamos los objetos nuevos o recoloquemos los acertados, ya que este tipo de extracción aleatoria inutiliza cualquier tipo de inserción ordenada.

Por tanto, es conveniente realizar la inserción más rápida posible, es decir, en la última posición de la lista. Los objetos acertados serán movidos también a esta última posición. ¿Por qué no se dejan estos objetos acertados en la misma posición en que estaban, como se ha hecho otras veces?

La razón es poco intuitiva. Dado que este algoritmo tiene en cuenta el contador de referencia a la hora de extraer cualquier objeto acertado o modificado, sería necesario modificarlo para incrementar su contador. Por tanto habría que eliminar este elemento y reinsertarlo en la misma posición con el nuevo contador. Puestos a tener que reinsertar, mucho más rápido hacerlo en la última posición que en la original.

Pero, lo realmente interesante de este algoritmo es la manera de extraer elementos. Para ello se ha definido una estructura auxiliar más. Una nueva lista (*TList*) auxiliar que almacenará los M elementos de menor contador.

Si se parte ahora del hecho de la existencia de una lista llena de objetos, y la necesidad de liberar espacio, para almacenar un nuevo elemento (entre paréntesis el contador de referencia del objeto):

OBJETO1(2)
OBJETO4(3)
OBJETO2(1)
……
OBJETO19(1)
OBJETO14(3)

Figura 3.20. La *cache* llena

- ***Primera extracción de la cache***

  1) Tomamos *N* elementos de manera aleatoria de la lista, tal y como se aprecia en la *Figura 3.21*:

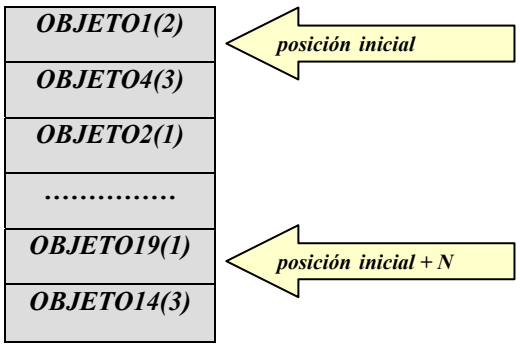


Figura 3.21. Selección de N objetos

2) En la *Figura 3.22* se describe como se eligen, a continuación, los *M* de menor contador de entre estos *N*, y los guardamos en la lista auxiliar de manera ordenada:

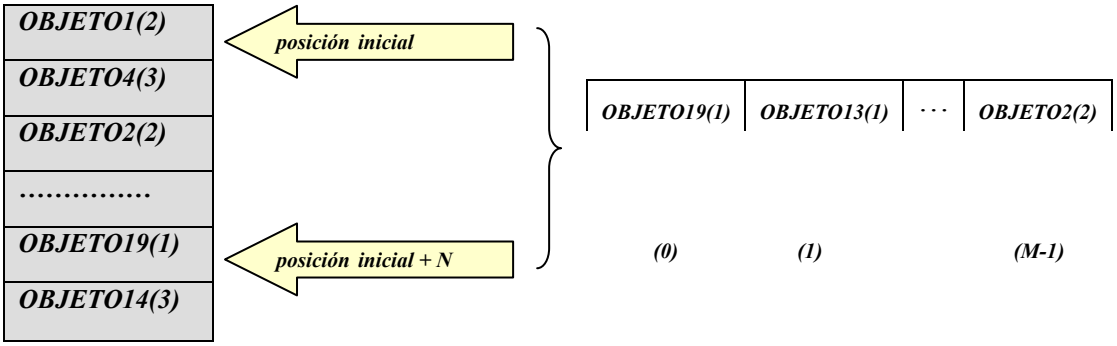


Figura 3.22. Inserción en la lista auxiliar

Elementos de menor contador estarán situados en las primeras posiciones de la lista. Cuando haya varios objetos con un mismo contador de referencia, el más recientemente llevado a la lista auxiliar tendrá un índice menor.

3) Eliminamos elementos de la *cache,* comenzando por el primer elemento (índice 0) de la lista auxiliar. Y así sucesivamente hasta liberar el espacio necesario, hecho que se describe en la *Figura3.23*:

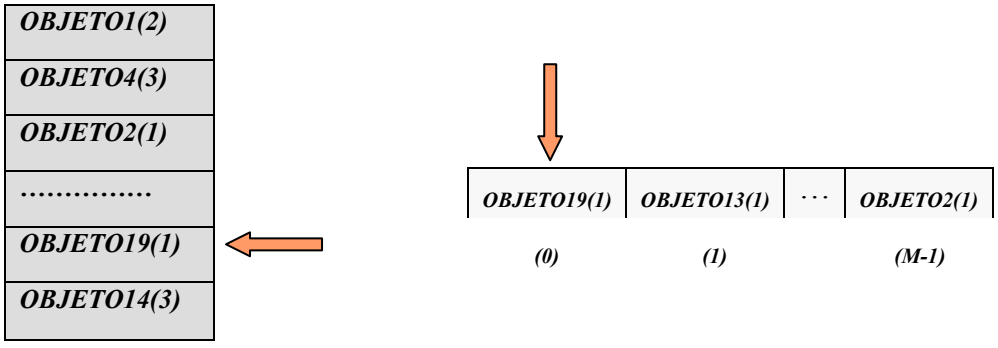


Figura 3.23. Eliminación de la *cache*

- ***Siguientes eliminaciones en la cache***

1) Tomamos *N-M* elementos de manera aleatoria de la lista, tal y como se indica en la *Figura 3.24*:

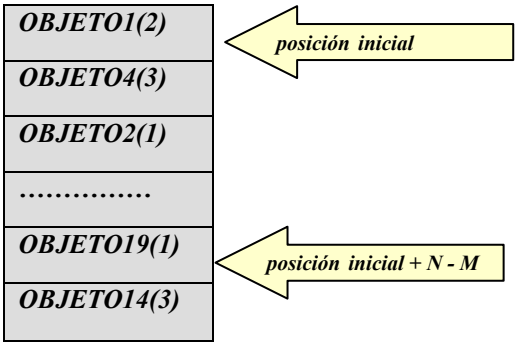


Figura 3.24. Selección N-M objetos de la *cache*

2) Elegimos los *M* de menor contador de entre estos *N-M*, y los M ya guardados en la lista auxiliar. Y los almacenamos en esta lista auxiliar. La manera es simple, con cada uno de los nuevos elementos seleccionados de la *cache,* se verifica si podría estar entre estos M elementos de la lista auxiliar, es decir, los de menor contador. Si es así, se inserta, y se elimina el último objeto de esta lista auxiliar. Si no es así, se rechaza, todo esto se detalla en la *Figura 3.25*:

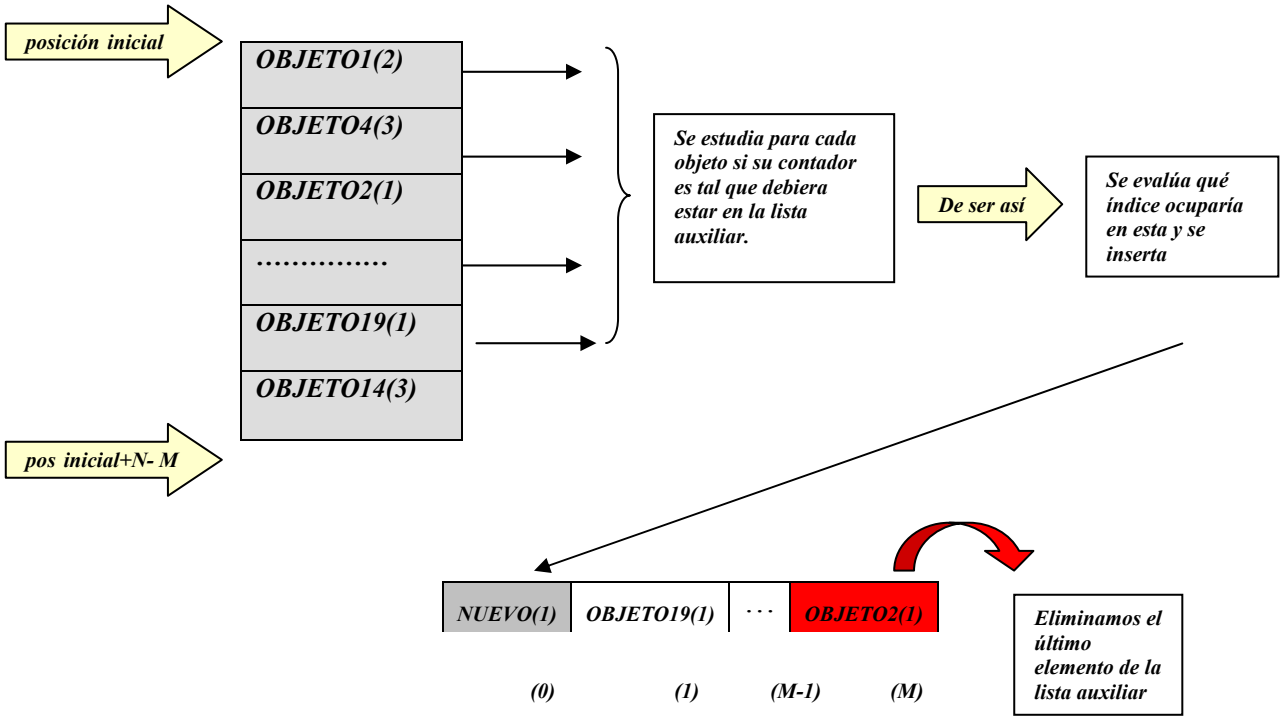


Figura 3.25. Inserción en la lista auxiliar II

3) Eliminamos elementos de la *cache,* comenzando por el primer elemento (índice 0) de la lista auxiliar. Y así sucesivamente hasta liberar el espacio necesario. La *Figura 3.26*, describe perfectamente este proceso:

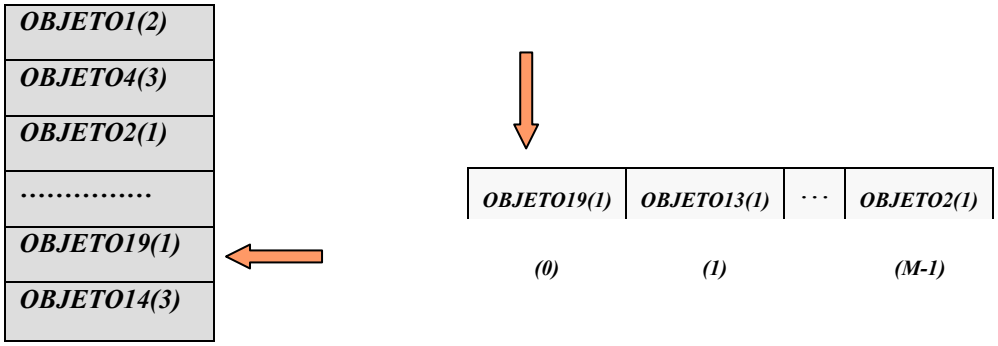


Figura 3.26. Eliminación de la *cache II*

Este es, en líneas generales, el procedimiento aleatorio que esta política sigue a la hora de eliminar objetos de la lista.

A continuación, en *Código 3.13,* se presentará la función que se encarga de realizar el proceso de eliminación de esta política:

```
Función lista_aux(lista,lista_aux,N,M)
{
para (índice aleatorio) hasta (índice aleatorio + N) // N-M a partir de la 1ª eliminación
{
    objeto = lista[índice aleatorio];
    si (objeto ya está en lista_aux){
        lista_aux_eliminar(objeto);
    }
    lista_aux_insertar_ordenado(objeto);
    si (nº elementos en lista_aux > M){
        lista_aux_eliminar(ultimo_elemento);
    }
}
}
```

Código 3.13. Inserción en la lista auxiliar del algoritmo RRGVF

3.3. EL ENTORNO DE PROGRAMACIÓN

Como ya se dijo en el primer capítulo, para la realización de este proyecto se ha requerido el uso de un entorno de programación de C++.

Ya se han visto las características principales de los algoritmos implementados para la simulación de las políticas. En este capítulo lo que se pretende es aportar cierta información acerca de las clases, estructuras de datos, o funciones que se han utilizado al desarrollar la programación del simulador.

El objetivo es aportar una información más técnica acerca de cómo se ha desarrollado el simulador, es decir, se pretende hacer una revisión de lo ya explicado pero con un enfoque más centrado en el ambiente de programación.

3.3.1. CLASES

Estas son el elemento clave de todo lenguaje orientado a objetos. Y se define como una plantilla o modelo que define los objetos.

La creación de una clase se divide en dos etapas principalmente. Por una parte se proporcionará su interfaz a través de un archivo de cabecera, donde realizaremos la definición de la clase, mientras que por otra se escribirá su implementación en un archivo .cpp.

A continuación se presentarán las clases más importantes que se han creado al desarrollar el simulador

3.3.1.1. La clase *Talgoritmo*

Esta es una clase creada para gestionar el comportamiento de los algoritmos propios de cada política.

Define una serie de métodos generales, algunos de los cuales se expondrán a continuación:

class Talgoritmo

void_algoritmo(THashedStringList*tabla[100],TList*lista,double TamagnoCache…);

Algoritmo general al que se llamará al simular las políticas. Realiza operaciones generales como comprobar si un objeto está ya en cache, actualizar las variables de

control (elementos acertados, elementos sacados, elementos modificados…), así como la extracción de elementos cuando sea necesario, para insertar uno nuevo.

Existen diversas variantes de este *'algoritmo',* según las características propias de cada grupo de políticas.

int buscar_indice(TList *lista,int ref);

Este método se encarga de la búsqueda de la posición adecuada en algoritmos que precisan listas ordenadas, como la *LFU* o *LFU-DA*. Como sucedió anteriormente, existen varias versiones según se pretenda encontrar una posición ordenada por contador, clave…

int digitos(AnsiString S);

Método con el que conseguimos el último dígito del identificador de un elemento. Pudiendo así conocer que lista de búsqueda le corresponde.

int Index(TList *lista,Tobjeto *objeto,int iIndice);

Variante del método *IndexOf,* que pretende la búsqueda de un índice partiendo de una posición de comienzo pasada como parámetro.

bool probabilidad(double dProbabiolidad);

Indica si se verifica la probabilidad que se le pasa como parámetro o no. Se emplea en *LRU-C* y *LRU-S* para decidir si los objetos son reubicados o no.

void guardar_M(TList *lista,TList *lista_aux,int N,int M,int iIndiceAleatorio);

Se emplea en el algoritmo RRGVF, para almacenar los posibles elementos a eliminar de la *cache* en caso de ser necesario.

int calcular_indice(TList *lista,double dMedia,double dMedia2);

Esta función devuelve un índice aleatorio empleando una distribución Gaussiana. Es empleado únicamente en la política *HARM*

Asimismo tras observar que la gran mayoría de las políticas comparten el método *algoritmo(),* se optó por la creación de subclases, cada una representativa de cada una de las políticas en liza. Subclases que heredan todas las características de la clase *Talgoritmo.*

Con todo el modelo de subclase empleado es:

```
class Tlru:Talgoritmo
 {
  public:
    //métodos
   void lru(THashedStringList *tabla[100],TList *lista,double dTamagnoCache,…);
 };
```

Código 3.14. La clase Tlru

y existirán tantas como políticas se han implementado.

3.3.1.2. La clase *Tobjeto*

Esta es una clase definida con el fin de identificar el concepto general que se pretende, de los documentos que se guardan en la *cache*.

Como toda clase, esta estará definida por una serie de propiedades:

```
class Tobjeto
{
   private:
    //propiedades
   long tamagno;  //  Tamaño del objeto
   int ident;  //  Identificador del objeto
   double tiempo;  //  Información temporal de cuándo se solicitó el objeto
   int contador;  //  Contador de referencia del objeto
   int ctI;     //   Content Type I, indica la naturaleza del objeto (imágenes, video,
                texto)
   int ctII; //  Content Type II, información más detallada del tipo de
                objeto(formato,extensión…)
   double clave;  // Clave del documento Web
}
```

Código 3.15. La clase Tobjeto

Cada uno de los cuales se identifican con cada uno de los campos que identifican a un documento.

Igualmente la clase definirá una serie de métodos relativos al objeto:

Tobjeto(long tam,int id,double tiemp,int cont,double cI,int cII,double key);

Se trata del constructor de la clase. Realizando una llamada a este se conseguirá la creación de un elemento objeto.

int get_contador();

Con este método accederemos al contenido del campo contador del objeto referenciado.

void set_contador(cont);

Modificará el contenido del campo contador de un objeto en cuestión

Existirán esta pareja de métodos por cada propiedad de la clase *Tobjeto.*

3.3.1.3. La clase *Tlectura*

Esta clase se diseñó tan sólo con el fin de mantener cierto orden al desarrollar la totalidad del código, es decir, con esta clase se busca realizar la lectura de los archivos de entrada y conseguir elementos con sus campos correspondientes, y dado que no se encuadraba en el marco de ninguna otra clase, se decidió incluirla en la suya propia.

Con todo el método principal de esta clase es:

```
class Tlectura
{
   public:

  AnsiString lectura(TStringList*entrada, int*iCampo, double&dTiempo,…);
}
```

Código 3.16. La clase Tlectura

y como ya se ha dicho, se encarga de la lectura del archivo y su conversión al modelo de objetos.

3.3.1.4. La clase *THebraCache*

Esta es una clase que se creó con el objetivo de realizar la llamada a una nueva hebra cada vez que se solicitase la ejecución de una nueva simulación.

Pero, ¿qué es una hebra? ¿para qué se utilizan? Todas estas dudas serán resueltas con más calma en un apartado posterior. Ahora parece más conveniente centrarse en los métodos y propiedades de esta nueva clase.

Las propiedades principales son:

```
class THebraCache : public TThread
{
    private:

    double dBytesTotales,dBytesAcumulados,dBytesAcertados,dElementosSacados
    double dElementosModificados,dElementosAcertados,dElementosTotales
    double dElementosEntrados,dPorcentajes[2];
    bool bContinuarHebra;
}
```

Código 3.17. La clase THebraCache

Estas propiedades son útiles en tanto en cuanto, se tratan de los resultados finales de la simulación, es decir, al terminar la simulación los resultados arrojados por esta se verán reflejados en el valor que adquieran cada una de estas propiedades.

Mientras que sus métodos son:

double get_aciertos();

Este método dará acceso al contenido de la propiedad aciertos, y aportará información acerca del número de aciertos producidos.

double get_bytes_acertados();

Método idéntico al anterior pero en este caso la información aportada hace referencia al número de bytes acertados.

Como éstos, habrá un método para conocer el valor de cada una de las propiedades de la clase.

C++ Builder proporciona la clase abstracta TThread para la creación de hebras.

Para definir una hebra se debe crear un descendiente de *TThread*, para lo cual se ha de crear una clase derivada de *TThread.* Ésta es una clase abstracta porque posee un método virtual puro denominado *Execute(),* que será el que se rellenará con el código asociado a la hebra.

Obligatoriamente, una clase derivada de *TThread,* debe incluir los miembros:

- El ***constructor*** de la clase:

```
fastcall THebraCache::THebraCache(bool CreateSuspended): TThread(false)
{
    Priority      = tpNormal;
}
```

- El método ***Execute,*** que contendrá el código asociado a la tarea que tenga que realizar la hebra:

```
void __fastcall THebraCache::Execute()
{
    // código asociado a la tarea que implementa la hebra
}
```

Con todo, el esquema global, con el que interactúan las diversas clases hasta ahora definidas:

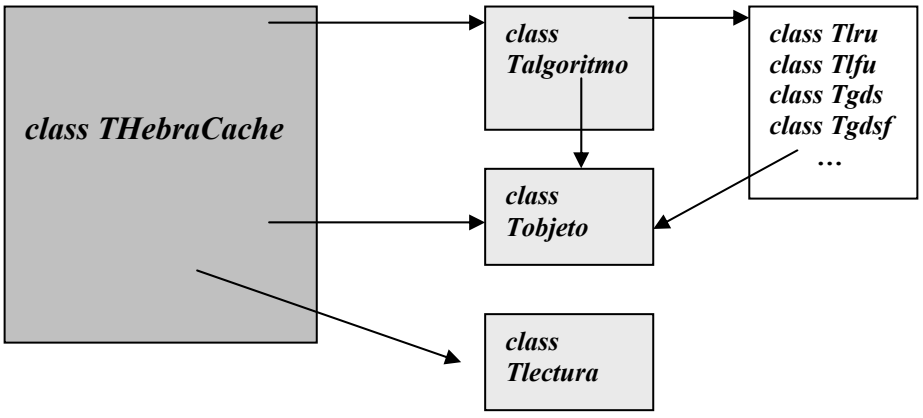


Figura 3.27. Relación entre las diversas clases

3.3.2. PROGRAMACIÓN CON HEBRAS

Una aplicación multihebra es aquella que contiene varias vías de ejecución.

Su uso (del paralelismo en general), proporciona una serie de ventajas frente a las limitaciones de los sistemas monotarea:

- **Aprovechamiento de los recursos del sistema**: Cuando se utiliza una sola hebra, el programa debe detener completamente la ejecución mientras espera a que se realice cada tarea. La *CPU* permanece ocupada completamente (o inactiva) hasta que el proceso actual termine. Si se utilizan varias hebras el sistema puede usarse para realizar varias tareas simultáneamente.
- **Establecimiento de las prioridades:** Asignando mayor prioridad a las tareas más importantes.
- **Multiprocesamiento real:** En un sistema multiprocesador, si la aplicación se descompone en varias hebras, el sistema operativo podrá asignar cada una a una de las CPU's del sistema.
- **Estructuración – Paralelismo implícito:** En muchas ocasiones, un programa puede diseñarse como varios procesos paralelos que funcionen de forma independiente. La descomposición de una aplicación en varias hebras puede resultar muy beneficiosa en términos de desacoplamiento de código (tareas independientes se implementan por separado) y de calidad del diseño (frente a futuras ampliaciones, por ejemplo).

En definitiva, con el empleo de hebras de ejecución se busca que el sistema tenga la capacidad de lanzar varios procesos, varias simulaciones de manera concurrente.

CAPÍTULO 4: Plan de Pruebas

El objetivo de este apartado es verificar el correcto funcionamiento de cada una de las políticas, es decir, corroborar que el simulador maneja las muestras de entrada tal y como debiera de forma teórica.

Para ello se prepara una muestra de entrada de veinte objetos, y se recrea el comportamiento de cada una de las políticas, observándose cómo actúa ante la llegada de un nuevo objeto, al producirse un acierto, una modificación o al necesitar eliminar objetos de la cache para liberar espacio, de esta manera se podrá estudiar cómo cada política maneja su contenido y dónde radica la diferencia entre todas ellas. Al tratarse de una muestra de veinte elementos, son veinte los pasos que se seguirán hasta llegar a los resultados finales, pasos que se indicarán numerándolos.

El objetivo último de esta tarea es comprobar que los resultados que se consiguen así son idénticos a los conseguidos usando el simulador.

Podremos observar también, que se utiliza cierto código de colores. Éste se describe más abajo, y con él se pretende resaltar cuando se produzca un acierto o una modificación.

Se identificará un objeto por sus campos característicos (tamaño, identificador, ctI y ctII), excepto por el referente al tiempo, que por razones de simplicidad se ha obviado, ya que no aporta información alguna a este estudio. También se podrá observar que en determinadas políticas, aparecen acompañando a un objeto ciertos valores entre paréntesis, esta información estará ligada al propio funcionamiento y necesidades de la política en cuestión, y se detallará para una mayor comprensión del algoritmo. En cualquier caso se especificará la naturaleza de estos valores antes de comenzar a recrear el algoritmo.

Una vez terminado el estudio del algoritmo, se detallarán los resultados más relevantes, así como el contenido final de la cache. Tras esto se empleará el simulador, de manera que se podrán contrastar los resultados por él arrojados con los obtenidos con la simulación 'manual'.

Por último, destacar que el tamaño asignado a la cache es de ***5000*** bytes. En caso de llegar una entrada de un tamaño superior a estos 5000 bytes, simplemente se descartará.

A continuación mostramos cual serán la muestra de entrada (campos de cada elemento – tiempo, tamaño, identificador, content type I, content type II) que emplearemos en todo este estudio.

Muestra de entrada:

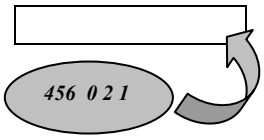
(tiempo)	*(tam)*	*(id)*	*(ctI)*	*(ctII)*	
1085356812.473	456	0	2	1	**(1)**
1085356814.078	458	1	2	1	**(2)**
1085356814.514	1458	2	2	1	**(3)**
1085356817.424	2456	4	2	1	**(4)**
1085356818.305	256	5	1	1	**(5)**
1085356819.753	425	1	2	1	**(6)**
1085356824.583	490	7	0	0	**(7)**
1085356846.244	512	9	1	1	**(8)**
1085356849.334	1450	2	2	1	**(9)**
1085356850.864	1550	2	2	1	**(10)**
1085356851.794	490	7	0	0	**(11)**
1085356853.794	120	11	0	2	**(12)**
1085356856.134	460	0	2	1	**(13)**
1085356859.635	458	1	2	1	**(14)**
1085356862.657	1260	13	1	1	**(15)**
1085356862.685	440	1	2	1	**(16)**
1085356862.705	514	9	1	1	**(17)**
1085356863.775	2570	4	1	1	**(18)**
1085356864.905	430	1	1	1	**(19)**
1085356864.925	100	15	0	0	**(20)**

4.1 LRU

Se inserta en el comienzo de la caché (posición superior), en caso de ser necesario se eliminará comenzando por el último elemento de la misma. Cuando se produzca un acierto o una modificación, también se moverá el objeto al comienzo de la cache.

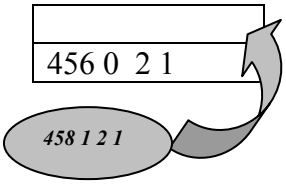
La cache en origen está vacía, llega el primer elemento (cache = 456)

(1)



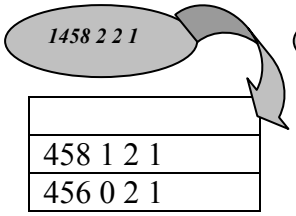
Llega un segundo objeto a la cache (cache = 914)

(2)



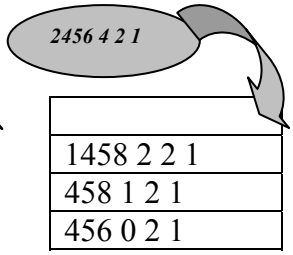
Llega un tercer objeto a la cache (cache = 2372)

(3)



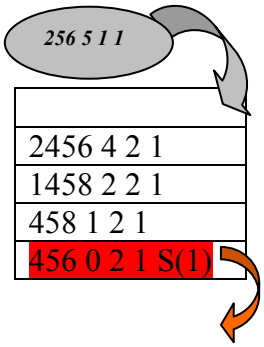
Llega el cuarto elemento (cache = 4828)

(4)



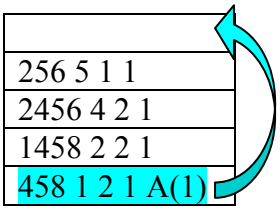
El 5º elemento no cabe, se elimina el último objeto de la cache (cache = 4628)

(5)



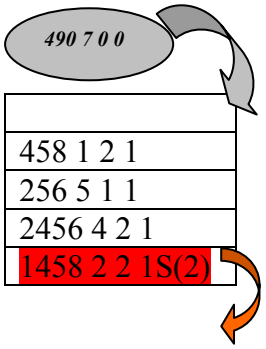
El 6º elemento es un acierto en cache (cache = 4628) (425 1 2 1)

(6)



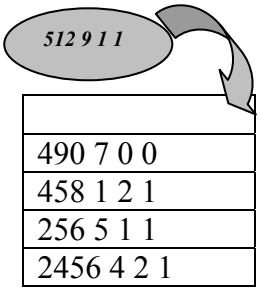
El 7º elemento no cabe, se elimina el último objeto de la cache (cache = 3660)

(7)



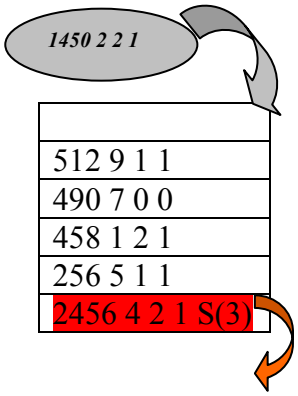
Llega el 8º elemento (cache = 4172)

(8)



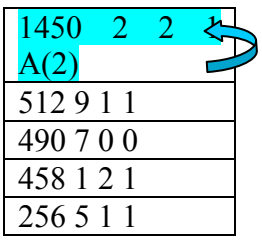
El 9º elemento no cabe, se elimina el último objeto de la cache (cache = 3166)

(9)



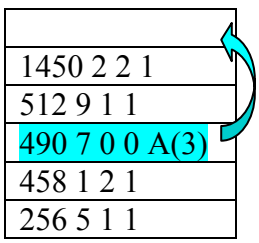
El 10º elemento es un acierto en cache (cache =3166) (1550 2 2 1)

(10)



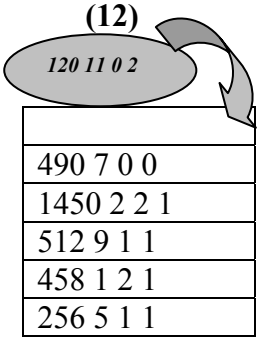
El 11º elemento es un acierto en cache (cache = 3166)

(11)



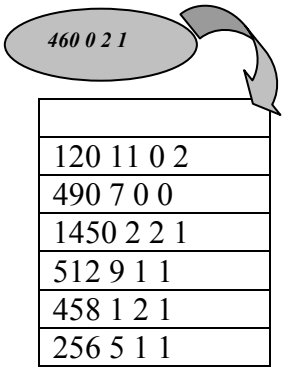
Llega 12º elemento (cache = 3286)

(12)



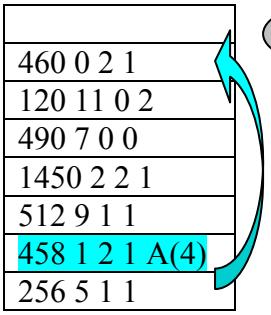
Llega el 13º elemento (cache = 3746)

(13)



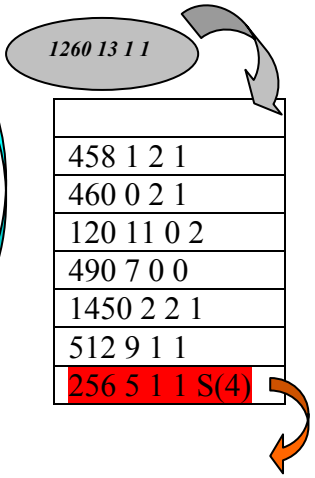
El 14º elemento es un acierto en cache (cache = 3746) (458 1 2 1)

(14)



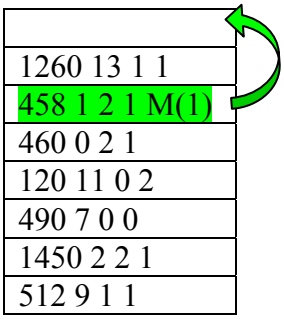
El 15º elemento no cabe, se elimina el último objeto de la cache (cache = 4750)

(15)



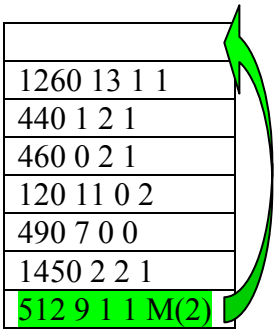
El 16º elemento es una modificación de un elemento existente (cache = 4732) (440 1 2 1)

(16)



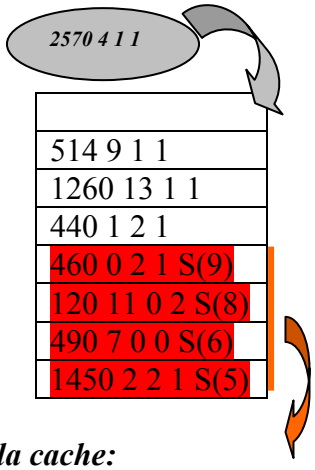
El 17º elemento es una modificación de un elemento existente (cache = 4734) (514 9 1 1)

(17)



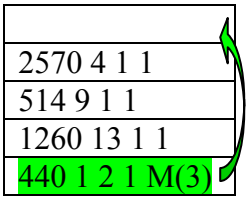
El 18º elemento no cabe, se eliminan objetos comenzando desde el último (cache = 4784)

(18)



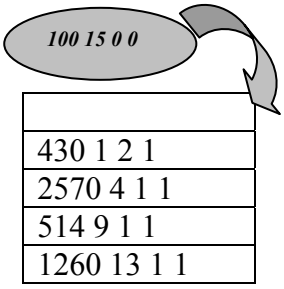
El 19º elemento es una modificación de un elemento existente (cache = 4774) (430 1 2 1)

(19)



Llega el 20º elemento (cache = 4874)

(20)



Contenido final de la cache:

100 15 0 0
430 1 2 1
2570 4 1 1
514 9 1 1
460 0 2 1

Resultados:

Aciertos :	4
Elementos Sacados :	8
Elementos Modificados :	3
Elementos finales en cache :	5
Bytes finales en cache :	4874

Empleando el simulador:

Contenido final de la cache:

1085356864.925 **100** 15 0 0 1 0 |0
1085356864.905 **430** 1 1 1 5 0 |1
1085356863.775 **2570** 4 1 1 1 0 |2
1085356862.705 **514** 9 1 1 2 0 |3
1085356862.657 **1260** 13 1 1 1 0 |4

Resultados

LRU 5000

Aciertos: 4
Bytes Acertados: 4333
Entradas Totales: 20
Bytes Totales: 16353
Elementos Finales en Cache: 5
Bytes **de cache ocupados: 4874**
HR. : 20
B.R.: 17.8743961352657
Entradas **Modificadas: 3**
Elementos Sacados: 8

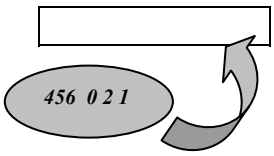
4.2. LFU

Se inserta de manera ordenada, en función del contador de referencia de cada elemento; así elementos que hayan sido referenciados un mayor número de veces permanecerán en la zona superior de la tabla. Eliminaremos elementos del final de la tabla (zona inferior de la tabla).

Entre paréntesis ilustraremos el valor del contador de un elemento.

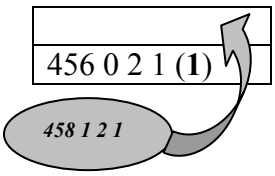
La cache en origen está vacía, llega el primer elemento (cache = 456)

(1)



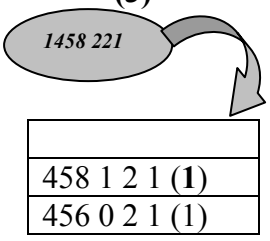
Llega el 2º elemento (cache = 914)

(2)



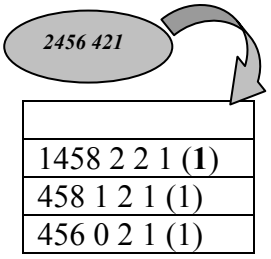
Llega el 3ª elemento (cache = 2372)

(3)



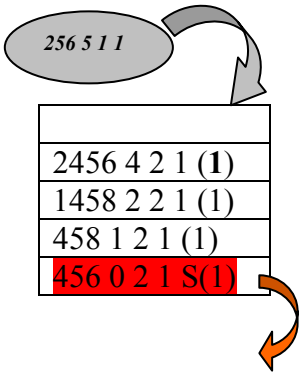
Llega el 4º elemento (cache = 4828)

(4)



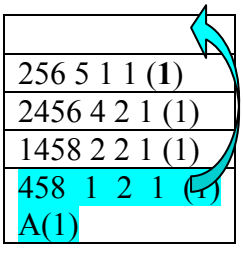
El 5º objeto no cabe, eliminamos el último elemento de la cache. (cache = 4628)

(5)



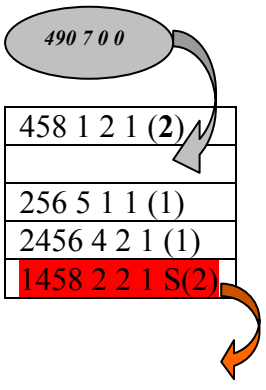
El 6º elemento es un acierto en cache(aumenta el contador) (cache = 4628) (425 1 2 1)

(6)



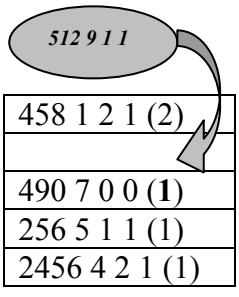
El 7º objeto no cabe, eliminamos el último elemento de la cache. (cache = 3660)

(7)



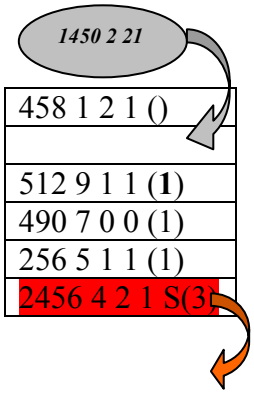
Llega el 8º elemento. (cache = 4172)

(8)



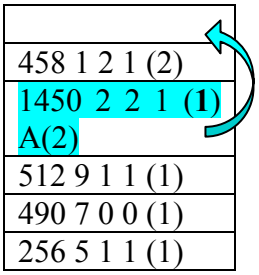
El 9º elemento no cabe, eliminamos de la cache comenzando por la cola.
(cache = 3166)

(9)



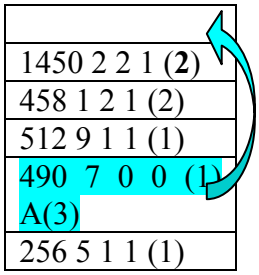
El 10º elemento es un acierto en la cache (reubicamos)
(cache = 3166)
(1550 2 2 1)

(10)



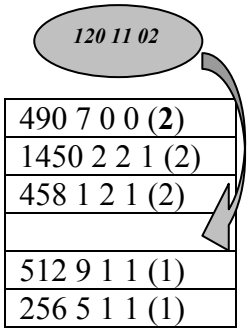
El 11º elemento es un acierto en la cache (reubicamos)
(cache =3166)
(490 7 0 0)

(11)



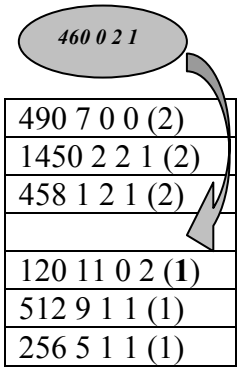
Llega el 12º elemento.
(cache = 3286)

(12)



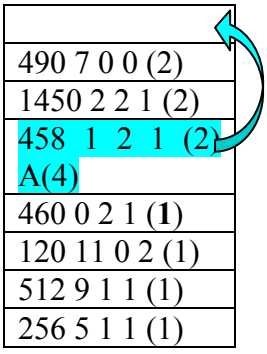
Llega el 13º elemento.
(cache = 3746)

(13)



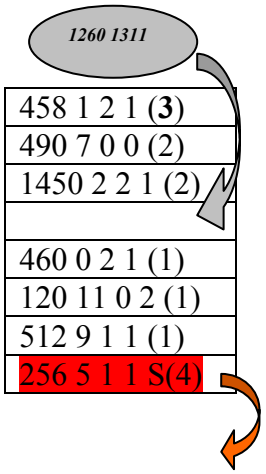
El 14º elemento es un acierto en cache (reubicamos)
(cache = 3746)
(458 1 2 1)

(14)



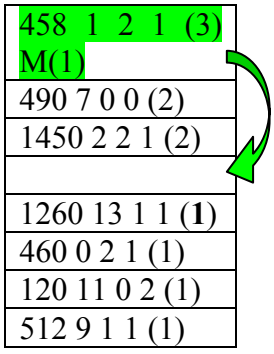
El 15º elemento no cabe, eliminamos el elemento con menor contador.
(cache = 4750)

(15)



El elemento 16º es una modificación de un objeto ya existente.
(cache = 4732)
(440 1 2 1)

(16)

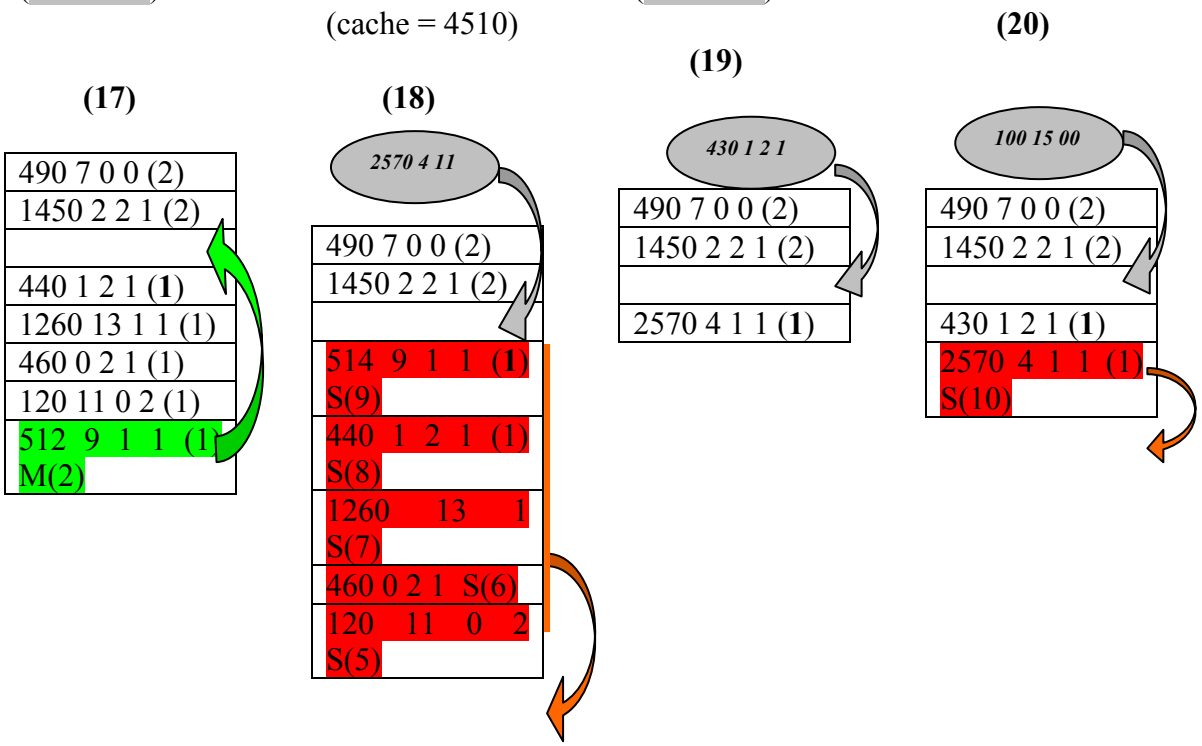


El 17º elemento es una modificación de un elemento ya existente (cache = 4734) (514 9 1 1)

18º elemento no cabe, eliminamos objetos comenzando por el de menor contador. (cache = 4510)

El 19º elemento es una modificación de un elemento ya existente. (cache = 4940) (430 1 2 1)

Llega el 20º elemento (cache = 2470)



Contenido final de la cache:

490 7 0 0 (2)
1450 2 2 1 (2)
100 15 0 0 (1)
430 1 2 1 (1)

Resultados:

Aciertos :	4
Elementos Sacados :	10
Elementos Modificados :	2
Elementos finales en cache :	4
Bytes finales en cache :	2470

Empleando el simulador:

Contenido final de la cache:

1085356851.794 **490** 7 0 0 2 0 |0
1085356850.864 **1450** 2 2 1 2 0 |1
1085356864.925 **100** 15 0 0 1 0 |2
1085356864.905 **430** 1 1 1 1 0 |3

Resultados

LFU 5000

Aciertos: 4
Bytes Acertados: 3793
Entradas Totales: 20
Bytes Totales: 16353
Elementos Finales en Cache: 4
Bytes de cache ocupados: 2470
HR. : 20
B.R.: 23.1945208830184
Entradas Modificadas: 2
Elementos Sacados: 10

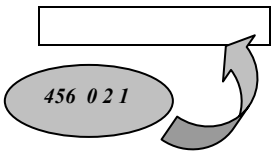
4.3. LFU-DA

Igual al LFU, la única variación radica en que se conserva una variable con el valor del contador de referencia del último elemento eliminado de la cache (contenido que se sombreará). Este valor se sumará al contador de referencia de todos los elementos acertados, y será el valor del contador de referencia de todos los elementos nuevos.

Entre paréntesis se ilustrará el valor del contador de referencia de un elemento.

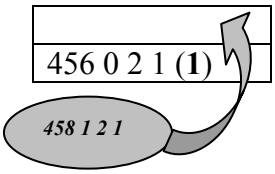
La cache en origen está vacía, llega el primer elemento (cache = 456)

(1)



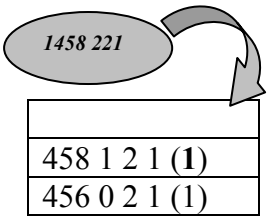
Llega el 2º elemento (cache = 914)

(2)



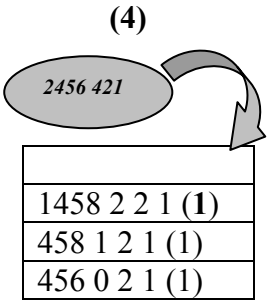
Llega el 3ª elemento (cache = 2372)

(3)



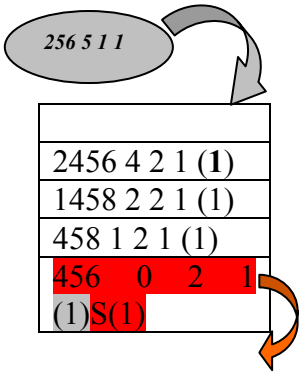
Llega el 4º elemento (cache = 4828)

(4)



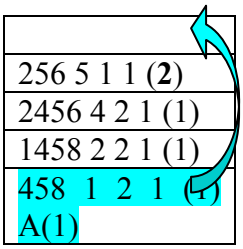
El 5º objeto no cabe, eliminamos el último elemento de la cache. (cache = 4628)

(5)



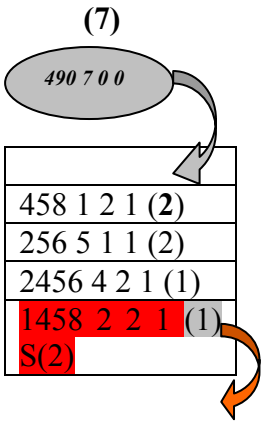
El 6º elemento es un acierto en cache(aumenta el contador) (cache = 4628) (425 1 2 1)

(6)



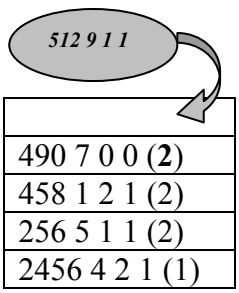
El 7º objeto no cabe, eliminamos el último elemento de la cache. (cache = 3660)

(7)



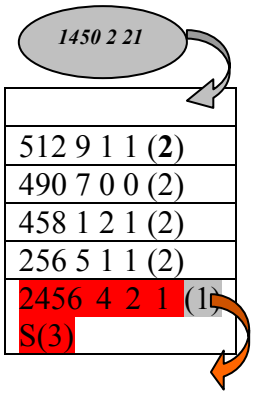
Llega el 8º elemento. (cache = 4172)

(8)



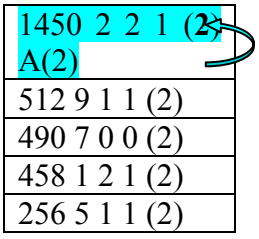
El 9º elemento no cabe, eliminamos de la cache comenzando por la cola.
(cache = 3166)

(9)



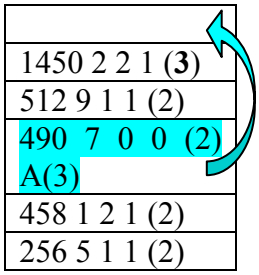
El 10º elemento es un acierto en la cache (reubicamos)
(cache = 3166)
(1550 2 2 1)

(10)



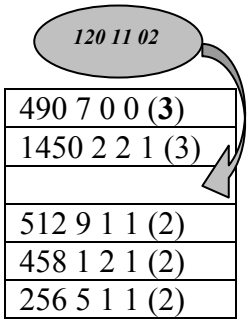
El 11º elemento es un acierto en la cache (reubicamos)
(cache =3166)
(490 7 0 0)

(11)



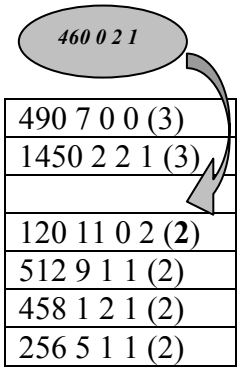
Llega el 12º elemento.
(cache = 3286)

(12)



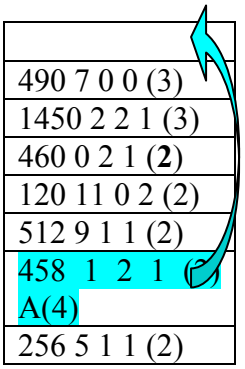
Llega el 13º elemento.
(cache = 3746)

(13)



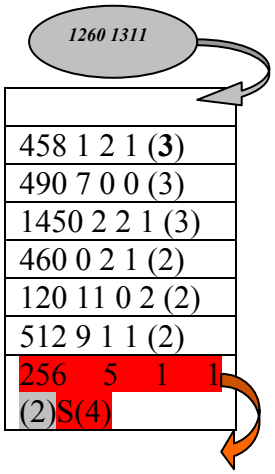
El 14º elemento es un acierto en cache (reubicamos)
(cache = 3746)
(458 1 2 1)

(14)



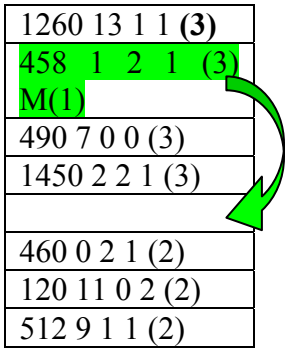
El 15º elemento no cabe, eliminamos el elemento con menor contador.
(cache = 4750)

(15)



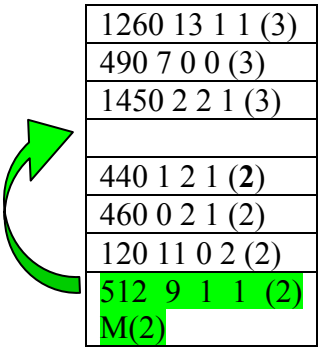
El elemento 16º es una modificación de un objeto ya existente.
(cache = 4732)
(440 1 2 1)

(16)



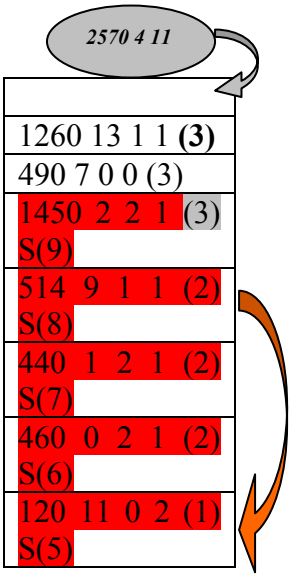
El 17º elemento es una modificación de un elemento ya existente (cache = 4734) (514 9 1 1)

(17)



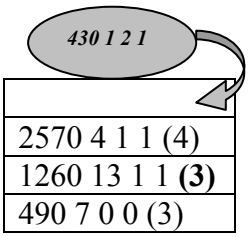
18º elemento no cabe, eliminamos objetos comenzando por el de menor contador. (cache = 4320)

(18)



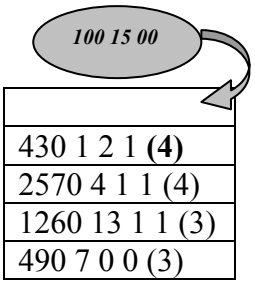
El 19º elemento es una modificación de un elemento ya existente. (cache = 4750) (430 1 2 1)

(19)



Llega el 20º elemento (cache = 4850)

(20)



Contenido final de la cache:

100 15 0 0 (4)
430 1 2 1 (4)
2570 4 1 1 (4)
1260 13 1 1 (3)
490 7 0 0 (3)

Resultados:

Aciertos :	4
Elementos Sacados :	9
Elementos Modificados :	2
Elementos finales en cache :	5
Bytes finales en cache :	4850

Empleando el simulador:

Contenido final de la cache:

1085356864.925 **100** 15 0 0 4 0 |0
1085356864.905 **430** 1 1 1 4 0 |1
1085356863.775 **2570** 4 1 1 4 0 |2
1085356862.657 **1260** 13 1 1 3 0 |3
1085356851.794 **490** 7 0 0 3 0 |4

Resultados

LFU-DA 5000

Aciertos: 4
Bytes Acertados: 2823
Entradas Totales: 20
Bytes Totales: 16353
Elementos Finales en Cache: 5
Bytes de cache ocupados: 4850
HR. : 21.0526315789474
B.R.: 17.758067559917
Entradas Modificadas: 2
Elementos Sacados: 9

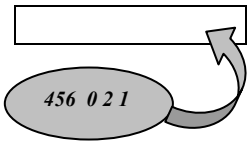
4.4. GDS(1)

Insertamos en la tabla de manera ordenada, siguiendo como criterio de ordenación la clave del elemento $\left(\frac{1}{tamaño\_objeto}\right)$. Cuando eliminemos un objeto, modificaremos la clave de todos los elementos existentes en la cache, restándoles la clave del elemento eliminado (valor que sombrearemos cada vez). Cuando un objeto ya existe o es modificado, recupera su llave original.

Entre paréntesis ilustraremos el valor de la llave de cada objeto.

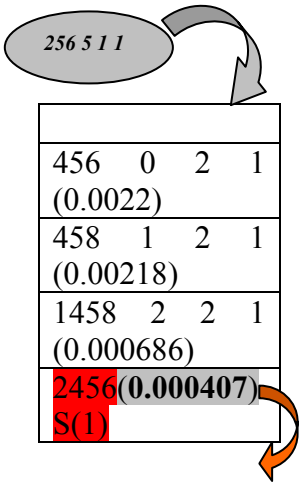
La cache en origen está vacía, llega el primer elemento (cache = 456)

(1)



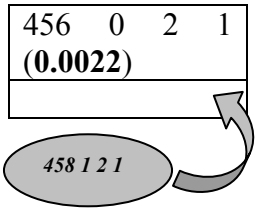
El 5º objeto no cabe, eliminamos el último elemento de la cache. (cache = 2628)

(5)



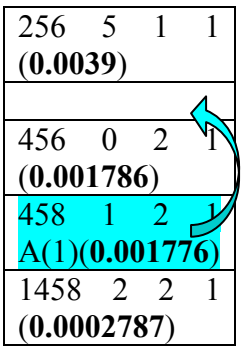
Llega el 2º elemento (cache = 914)

(2)



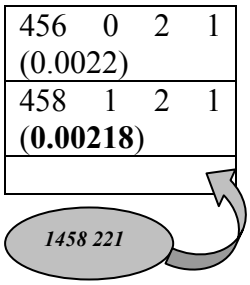
El 6º elemento es un acierto en cache(aumenta el contador) (cache = 2628) (425 1 2 1)

(6)



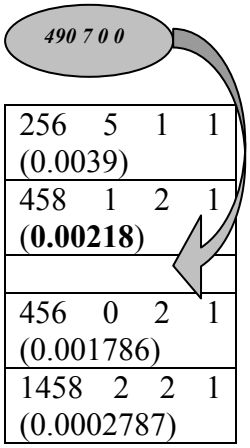
Llega el 3ª elemento (cache = 2372)

(3)



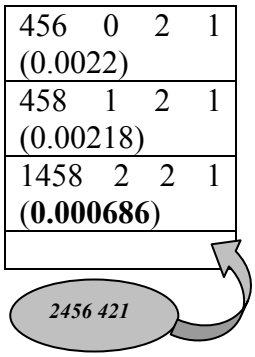
Llega el 8º elemento. (cache = 3118)

(7)



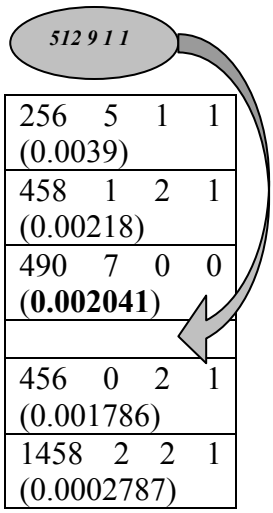
Llega el 4º elemento (cache = 4828)

(4)



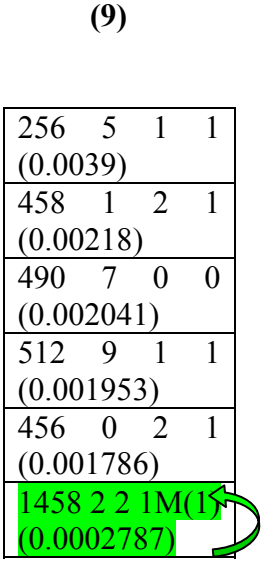
Llega el 8º elemento. (cache = 3630)

(8)



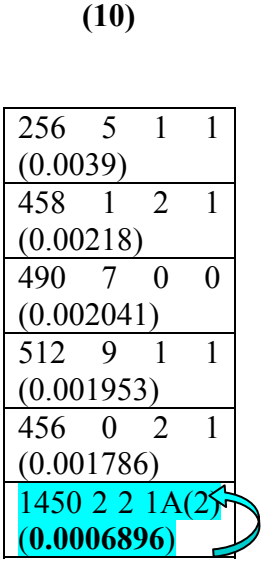
El 9º elemento es una modificación de un elemento ya existente
(cache = 3622)
(1450 2 2 1)

(9)



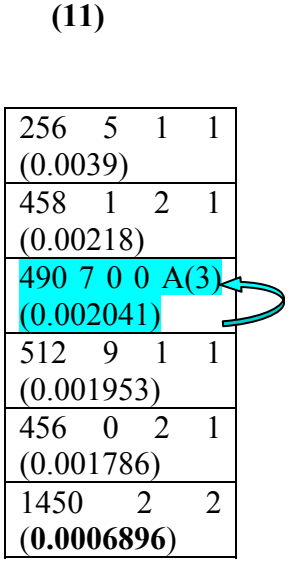
El 10º elemento es un acierto en la cache
(cache = 3622)
(1550 2 2 1)

(10)



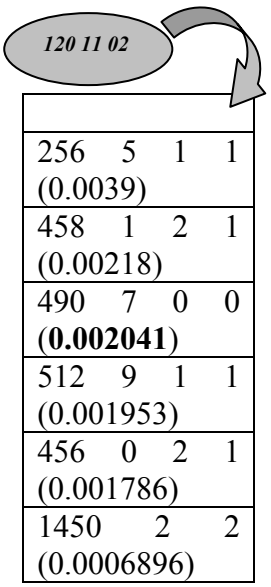
El 11º elemento es un acierto en la cache
(cache =3622)
(490 7 0 0)

(11)



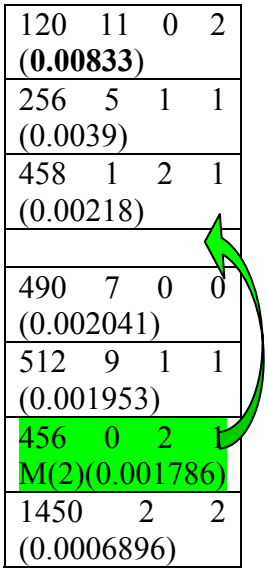
Llega el 12º elemento.
(cache = 3742)

(12)



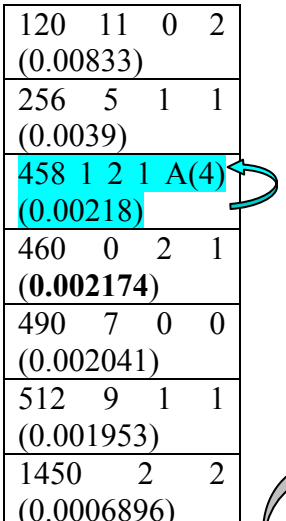
El 13º elemento es una modificación de un elemento ya existente.
(cache = 3746)
(460 0 2 1)

(13)



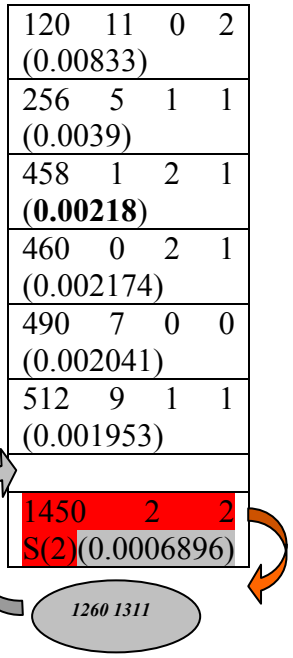
El 14º elemento es un acierto en cache (reubicamos)
(cache = 3746)
(458 1 2 1)

(14)



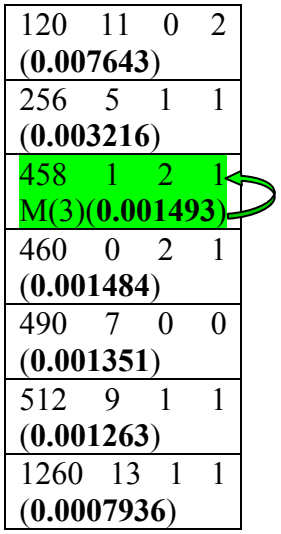
El 15º elemento no cabe, eliminamos el elemento con menor llave.
(cache = 3556)

(15)



El elemento 16º es una modificación de un objeto ya existente.
(cache = 3538)
(440 1 2 1)

(16)



El 17º elemento es una modificación de un elemento ya existente
(cache = 3540)
(514 9 1 1)

18º elemento no cabe, eliminamos objetos comenzando por el de menor contador.
(cache = 4850)

El 19º elemento es una modificación de un elemento ya existente.
(cache = 4840)
(430 1 2 1)

Llega el 20º elemento
(cache = 4940)

(17)

120 11 0 2 (0.007643)
256 5 1 1 (0.003216)
440 1 2 1 (**0.002272**)
460 0 2 1 (0.001484)
490 7 0 0 (0.001351)
512 9 1 1 M(4)(0.001263)
1260 13 1 1 (0.0007936)

(18)

120 11 0 2 (0.007643)
256 5 1 1 (0.003216)
440 1 2 1 (0.002272)
514 9 1 1 (**0.001945**)
460 0 2 1 (0.001484)
490 7 0 0 (0.001351)
1260 13 1 1 S(3)(0.0007936)

2570 4 11

(19)

120 11 0 2 (**0.00685**)
256 5 1 1 (**0.002423**)
440 1 2 1 M(5)(**0.001478**)
514 9 1 1 (**0.00115**)
460 0 2 1 (**0.00069**)
490 7 0 0 (**0.000557**)
2570 4 1 1 (**0.00039**)

(20)

100 15 00

120 11 0 2 (0.00685)
256 5 1 1 (0.002423)
430 1 2 1 (**0.002272**)
514 9 1 1 (0.00115)
460 0 2 1 (0.00069)
490 7 0 0 (0.000557)
2570 4 1 1 (0.00039)

Contenido final de la cache:

100 15 0 0 (0.01)
120 11 0 2 (0.00685)
256 5 1 1 (0.002423)
430 1 2 1 (0.002325)
514 9 1 1 (0.00115)
460 0 2 1 (0.00069)
490 7 0 0 (0.000557)
2570 4 1 1 (0.00039)

Resultados:

Aciertos :	4
Elementos Sacados :	3
Elementos Modificados :	5
Elementos finales en cache :	8
Bytes finales en cache :	4940

Empleando el simulador:

Contenido final de la cache:

1085356864.925 **100** 15 0 0 1 0.0118459782077519 |0
1085356853.794 **120** 11 0 2 1 0.00874049945711184 |1
1085356818.305 **256** 5 0 1 1 0.0043134161237785 |2
1085356864.905 **430** 1 0 1 5 0.00417155960310071 |3
1085356862.705 **514** 9 0 1 2 0.00299785270592988 |4
1085356856.134 **460** 0 0 1 2 0.00258107916725676 |5
1085356851.794 **490** 7 0 0 2 0.00244798245030911 |6
1085356863.775 **2570** 4 0 1 1 0.00223508326611763 |7

Debido a la propia idiosincrasia del algoritmo, vemos como las llaves en ambas simulaciones no coinciden. Ese es un comportamiento esperado y sin importancia, ya que la llave no es un campo propio de los objetos, sólo resulta útil en tanto en cuanto ayuda a la correcta ordenación de los elementos en la cache. Lo único realmente importante es que las llaves mantengan la misma posición relativa en ambos casos, independientemente que sus valores no coincidan.

Resultados

GDS (1) 5000

Aciertos: 4
Bytes Acertados: 6147
Entradas Totales: 20
Bytes Totales: 16353
Elementos Finales en Cache: 8
Bytes **de cache ocupados: 4940**
HR. : 20
B.R.: 37.5894331315355
Entradas Modificadas: 5
Elementos Sacados: 3

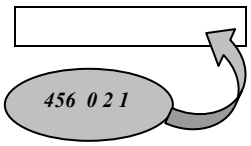
4.5. GDS(p)

Se comporta igual al GDS (1), la única diferencia radica en la manera en que calcularemos la clave, $\left(2 + \frac{tamaño\_objeto}{536} \right)$.

Entre paréntesis ilustraremos el valor de la llave de un elemento.

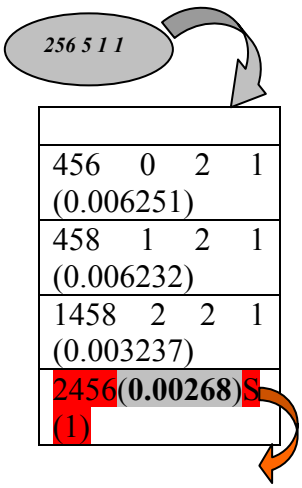
La cache en origen está vacía, llega el primer elemento (cache = 456)

(1)



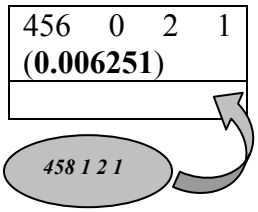
El 5º objeto no cabe, eliminamos el último elemento de la cache. (cache = 2628)

(5)



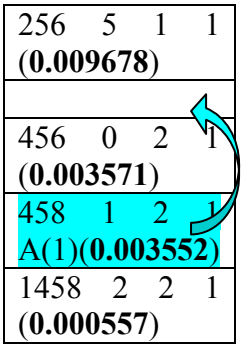
Llega el 2º elemento (cache = 914)

(2)



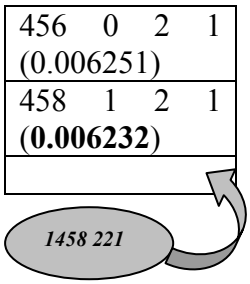
El 6º elemento es un acierto en cache (aumenta el contador) (cache = 2628) (425 1 2 1)

(6)



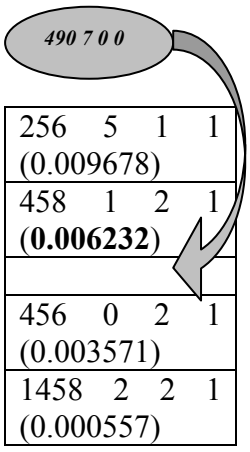
Llega el 3ª elemento (cache = 2372)

(3)



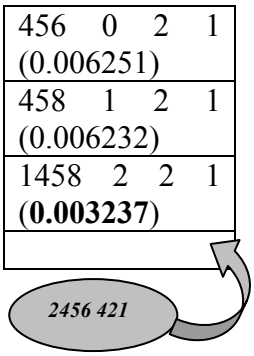
Llega el 8º elemento. (cache = 3118)

(7)



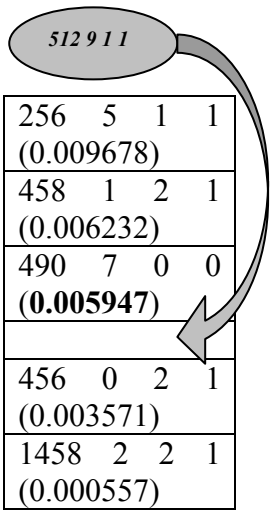
Llega el 4º elemento (cache = 4828)

(4)



Llega el 8º elemento. (cache = 3630)

(8)



El 9º elemento es una modificación de un elemento ya existente
(cache = 3622)
(1450 2 2 1)

(9)

256 5 1 1 (0.009678)
458 1 2 1 (0.006232)
490 7 0 0 (0.005947)
512 9 1 1 (**0.005772**)
456 0 2 1 (0.003571)
1458 2 2 1 M(1)(0.000557)

El 10º elemento es un acierto en la cache
(cache = 3622)
(1550 2 2 1)

(10)

256 5 1 1 (0.009678)
458 1 2 1 (0.006232)
490 7 0 0 (0.005947)
512 9 1 1 (0.005772)
456 0 2 1 (0.003571)
1450 2 2 1A(2) (**0.003245**)

El 11º elemento es un acierto en la cache
(cache =3622)
(490 7 0 0)

(11)

256 5 1 1 (0.009678)
458 1 2 1 (0.006232)
490 7 0 0 A(3) (0.005947)
512 9 1 1 (0.005772)
456 0 2 1 (0.003571)
1450 2 2 1 (**0.003245**)

Llega el 12º elemento.
(cache = 3742)

(12)

120 11 02

256 5 1 1 (0.009678)
458 1 2 1 (0.006232)
490 7 0 0 (**0.005947**)
512 9 1 1 (0.005772)
456 0 2 1 (0.003571)
1450 2 2 1 (0.003245)

El 13º elemento es una modificación de un elemento ya existente.
(cache = 3746)
(460 0 2 1)

(13)

120 11 0 2 (**0.01853**)
256 5 1 1 (0.009678)
458 1 2 1 (0.006232)
490 7 0 0 (0.00594)
512 9 1 1 (0.005772)
456 0 2 1 M(2)(0.003571)
1450 2 2 (0.003245)

El 14º elemento es un acierto en cache
(reubicamos)
(cache = 3746)
(458 1 2 1)

(14)

120 11 0 2 (**0.01853**)
256 5 1 1 (0.009678)
458 1 2 1 A(4) (0.006232)
460 0 2 1 (**0.006213**)
490 7 0 0 (0.00594)
512 9 1 1 (0.005772)
1450 2 2 (0.003245)

El 15º elemento no cabe, eliminamos el elemento con menor llave.
(cache = 3556)

(15)

1260 1311

120 11 0 2 (0.01853)
256 5 1 1 (0.009678)
458 1 2 1 (**0.006232**)
460 0 2 1 (0.006213)
490 7 0 0 (0.00594)
512 9 1 1 (0.005772)
1450 2 2 S(2) (0.003245)

El elemento 16º es una modificación de un objeto ya existente.
(cache = 3538)
(440 1 2 1)

(16)

120 11 0 2 (**0.01528**)
256 5 1 1 (**0.006433**)
1260 13 1 1 (**0.003453**)
458 1 2 1 M(3) (**0.002987**)
460 0 2 1 (**0.002968**)
490 7 0 0 (**0.002695**)
512 9 1 1 (**0.002527**)

El 17º elemento es una modificación de un elemento ya existente (cache = 3540) (514 9 1 1)

18º elemento no cabe, eliminamos objetos comenzando por el de menor contador. (cache = 3900)

El 19º elemento es una modificación de un elemento ya existente. (cache = 3890) (430 1 2 1)

Llega el 20º elemento (cache = 3990)

(17)

120 11 0 2 (0.01528)
256 5 1 1 (0.006433)
440 1 2 1 (**0.006411**)
1260 13 1 1 (0.003453)
460 0 2 1 (**0.002968**)
490 7 0 0 (**0.002695**)
512 9 1 1 M(4)(0.001263)

(18)

120 11 0 2 (0.01528)
256 5 1 1 (0.006433)
440 1 2 1 (0.006411)
514 9 1 1 (**0.005756**)
1260 13 1 1 S(5) (0.003453)
460 0 2 1 S(4) (0.002968)
490 7 0 0 S(3) (0.002695)

2570 4 11

(19)

120 11 0 2 (**0.0012585**)
256 5 1 1 (**0.003738**)
440 1 2 1 M(5)(**0.003716**)
514 9 1 1 (**0.003061**)
2570 4 1 1 (**0.002644**)

(20)

100 15 00

120 11 0 2 (0.0012585)
430 1 2 1 (**0.006517**)
256 5 1 1 (0.003738)
514 9 1 1 (0.003061)
2570 4 1 1 (0.002644)

Contenido final de la cache:

100 15 0 0 (0.02186)
120 11 0 2 (0.0012585)
430 1 2 1 (0.006517)
256 5 1 1 (0.003738)
514 9 1 1 (0.003061)
2570 4 1 1 (0.002644)

Resultados:

Aciertos :	4
Elementos Sacados :	5
Elementos Modificados :	5
Elementos finales en cache :	6
Bytes finales en cache :	3990

Empleando el simulador:

Contenido final de la cache:

1085356864.925 **100** 15 0 0 1 0.0304929798259914 |0
1085356853.794 **120** 11 0 2 1 0.0212123421978058 |1
1085356864.905 **430** 1 1 1 5 0.015144142616689 |2
1085356818.305 **256** 5 1 1 1 0.0123581755311391 |3
1085356862.705 **514** 9 1 1 2 0.0116817081014153 |4
1085356863.775 **2570** 4 1 1 1 0.0112711899427229 |5

Debido a la propia idiosincrasia del algoritmo, vemos como las llaves en ambas simulaciones no coinciden. Ese es un comportamiento esperado y sin importancia, ya que la llave no es un campo propio de los objetos, sólo resulta útil en tanto en cuanto ayuda a la correcta ordenación de los elementos en la cache. Lo único realmente importante es que las llaves mantengan la misma posición relativa en ambos casos, independientemente de que sus valores no coincidan.

Resultados

GDS (packet) 5000

Aciertos: 4 |
Bytes Acertados: 6147
Entradas Totales: 20
Bytes Totales: 16353
Elementos Finales en Cache: 6
Bytes de cache ocupados: 3990
HR. : 20
B.R.: 37.5894331315355
Entradas Modificadas: 5
Elementos Sacados: 5

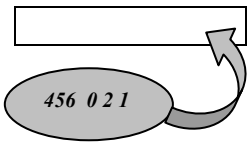
4.6. GDSF(1)

Insertamos en la tabla de manera ordenada en función de la clave $\left(L+\left(contador\times\frac{1}{tamaño\_objeto}\right)\right)$. El valor del contador inicialmente vale 1, y se incrementa por cada acierto o modificación. El valor inicial de L es cero, y se actualiza cada vez que se elimina un objeto con el valor de su clave.

Entre paréntesis se ilustrará primero el valor del contador y de la llave de un elemento. Destacamos L sombreándolo.

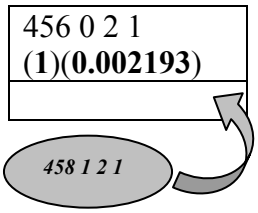
La cache en origen está vacía, llega el primer elemento (cache = 456)

(1)



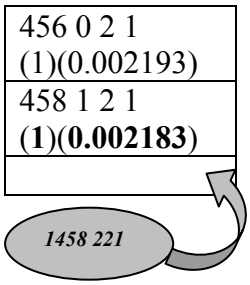
Llega el 2º elemento (cache = 914)

(2)



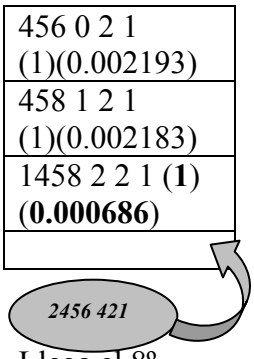
Llega el 3ª elemento (cache = 2372)

(3)



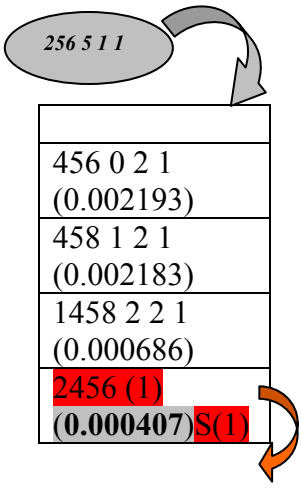
Llega el 4º elemento (cache = 4828)

(4)



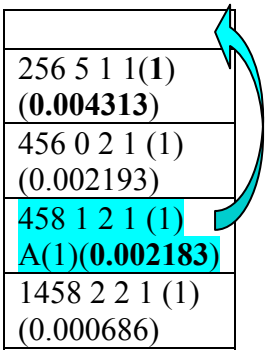
El 5º objeto no cabe, eliminamos el último elemento de la cache. (cache = 2628)

(5)



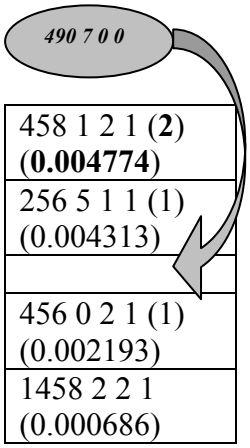
El 6º elemento es un acierto en cache (aumenta el contador) (cache = 2628) (425 1 2 1)

(6)



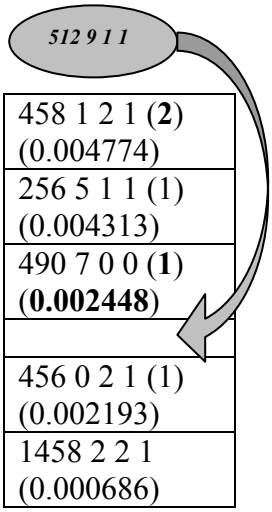
Llega el 7º elemento. (cache = 3118)

(7)



Llega el 8º elemento. (cache = 3630)

(8)



El 9º elemento es una modificación de un elemento ya existente (cache = 3622) (1450 2 2 1)

(9)

458 1 2 1 (**2**) (0.004774)
256 5 1 1 (1) (0.004313)
490 7 0 0 (**1**) (0.002448)
512 9 1 1 (**1**) (**0.00236**)
456 0 2 1 (1) (0.002193)
1458 2 2 1 M(1) (1) (0.0006858)

El 10º elemento es un acierto en la cache (cache = 3622) (1550 2 2 1)

(10)

458 1 2 1 (2) (0.004774)
256 5 1 1 (1) (0.004313)
490 7 0 0 (1) (0.002448)
512 9 1 1 (1) (0.00236)
456 0 2 1 (1) (0.002193)
1450 2 2 1A(2) (**2**)(**0.001093**)

El 11º elemento es un acierto en la cache (cache =3622) (490 7 0 0)

(11)

458 1 2 1 (2) (0.004774)
256 5 1 1 (1) (0.004313)
490 7 0 0 A(3) (1) (0.002448)
512 9 1 1 (1) (0.00236)
456 0 2 1 (1) (0.002193)
1450 2 2 2 (2) (**0.001786**)

Llega el 12º elemento. (cache = 3742)

(12)

120 11 02

458 1 2 1 (2) (0.004774)
490 7 0 0 (**2**) (**0.004488**)
256 5 1 1 (1) (0.004313)
512 9 1 1 (1) (0.00236)
456 0 2 1 (1) (0.002193)
1450 2 2 2 (2) (0.001786)

El 13º elemento es una modificación de un elemento ya existente. (cache = 3746) (460 0 2 1)

(13)

120 11 0 2 (**1**) (**0.00874**)
458 1 2 1 (2) (0.004774)
490 7 0 0 (2) (0.004488)
256 5 1 1 (1) (0.004313)
512 9 1 1 (1) (0.00236)
456 0 2 1 M(2) (1) (0.002193)
1450 2 2 2 (3) (0.001786)

El 14º elemento es un acierto en cache (reubicamos) (cache = 3746) (458 1 2 1)

(14)

120 11 0 2 (1) (0.00874)
458 1 2 1 A(4) (2) (0.004774)
490 7 0 0 (2) (0.004488)
256 5 1 1 (1) (0.004313)
460 0 2 1 (**1**) (**0.00258**)
512 9 1 1 (1) (0.00236)
1450 2 2 2 (3) (0.001786)

El 15º elemento no cabe, eliminamos el elemento con menor llave. (cache = 3556)

(15)

120 11 0 2 (1) (0.00874)
458 1 2 1 (**3**) (**0.006957**)
490 7 0 0 (2) (0.004488)
256 5 1 1 (1) (0.004313)
460 0 2 1 (1) (0.00258)
512 9 1 1 (1) (0.00236)
1450 2 2 2 (3) (0.001786) S(2)

1260 1311

El elemento 16º es una modificación de un objeto ya existente. (cache = 3538) (440 1 2 1)

(16)

120 11 0 2 (1) (0.00874)
458 1 2 1 M(3) (3) (0.004748)
490 7 0 0 (2) (0.004488)
256 5 1 1 (1) (0.004313)
460 0 2 1 (1) (0.00258)
1260 13 1 1 (**1**) (**0.002579**)
512 9 1 1 (1) (0.00236)

Llega el 17º elemento (cache = 4990) (514 9 1 1)

(17)

18º elemento no cabe, eliminamos objetos comenzando por el de menor contador. (cache = 4850)

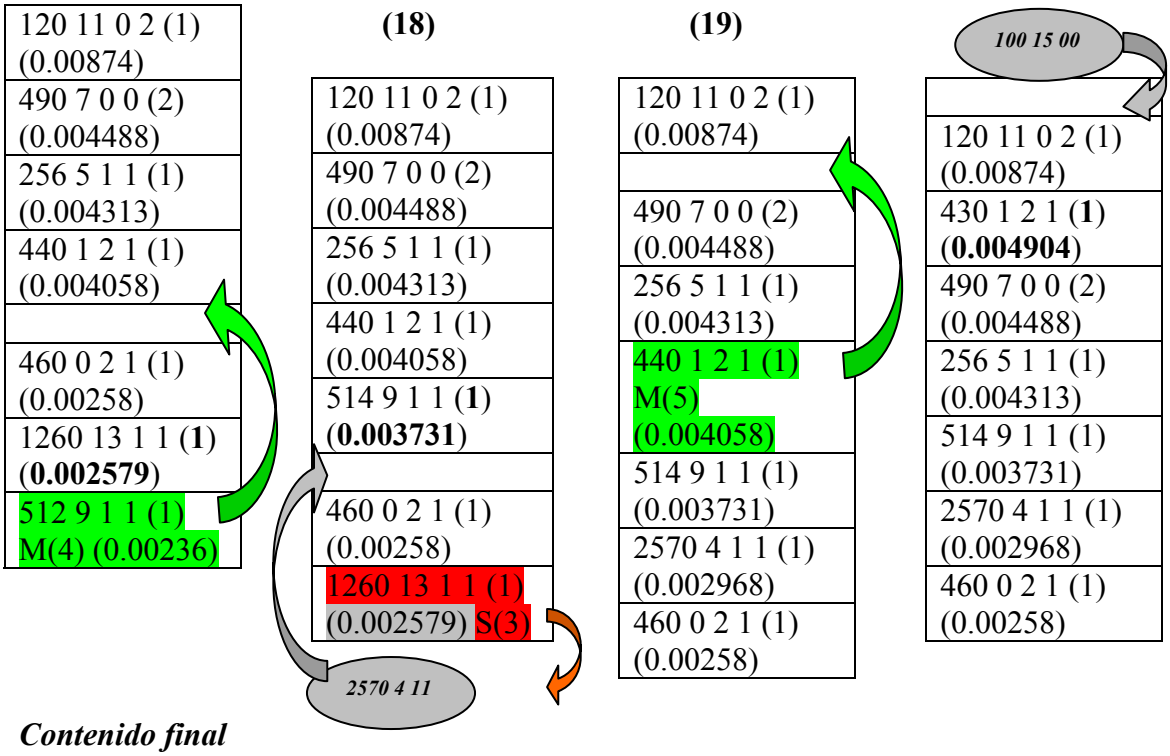
(18)

El 19º elemento es una modificación de un elemento ya existente. (cache = 4840) (430 1 2 1)

(19)

Llega el 20º elemento (cache = 4940)

(20)



Contenido final de la cache:

100 15 0 0 (1) (0.01258)
120 11 0 2 (1) (0.00874)
430 1 2 1 (1) (0.004904)
490 7 0 0 (2) (0.004488)
256 5 1 1 (1) (0.004313)
514 9 1 1 (1) (0.003731)
2570 4 1 1 (1) (0.002968)
460 0 2 1 (1) (0.00258)

Resultados:

Aciertos :	4
Elementos Sacados :	3
Elementos Modificados :	5
Elementos finales en cache :	8
Bytes finales en cache :	4940

Empleando el simulador:

Contenido final de la cache:

1085356864.925 **100** 15 0 0 1 0.0125801272622569 |0
1085356853.794 **120** 11 0 2 1 0.00874049945711184 |1
1085356864.905 **430** 1 1 1 1 0.00490570865760572 |2
1085356851.794 **490** 7 0 0 2 0.00448879877683973 |3
1085356818.305 **256** 5 1 1 1 0.0043134161237785 |4
1085356862.705 **514** 9 1 1 1 0.00373200176043488 |5
1085356863.775 **2570** 4 1 1 1 0.00296923232062264 |6
1085356856.134 **460** 0 2 1 1 0.00258107916725676 |7

Resultados

GDSF (1) 5000

Aciertos: 4
Bytes Acertados: 6117
Entradas Totales: 20
Bytes Totales: 16353
Elementos Finales en Cache: 8
Bytes de cache ocupados: 4940
HR. : 20
B.R.: 37.4059805540268
Entradas Modificadas: 5
Elementos Sacados: 3

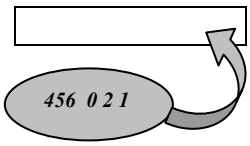
4.7. GDSF(p)

Se comporta igual al *GDSF (1)*, pero en este caso calcularemos la llave siguiendo la fórmula $\left(L + \left(contadorx \left(\frac{2 + \frac{tamaño\_objeto}{536}}{tamaño\_objeto} \right) \right) \right)$.

Entre paréntesis se ilustrará primero el valor del contador y de la llave de un elemento. Se resaltará L sombreándolo.

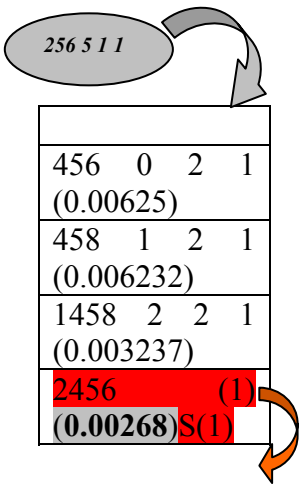
La cache en origen está vacía, llega el primer elemento (cache = 456)

(1)



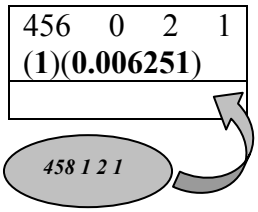
El 5º objeto no cabe, eliminamos el último elemento de la cache. (cache = 2628)

(5)



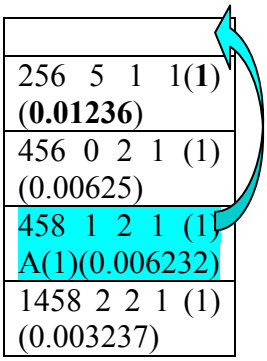
Llega el 2º elemento (cache = 914)

(2)



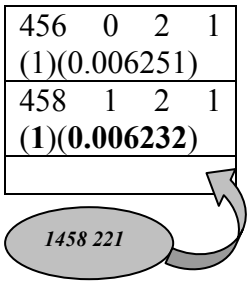
El 6º elemento es un acierto en cache (aumenta el contador) (cache = 2628) (425 1 2 1)

(6)



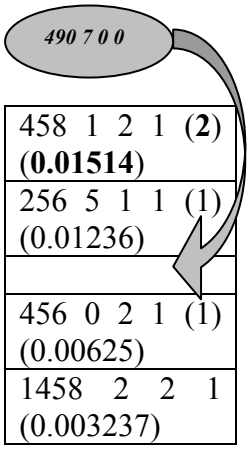
Llega el 3ª elemento (cache = 2372)

(3)



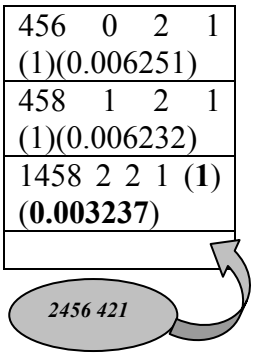
Llega el 8º elemento. (cache = 3118)

(7)



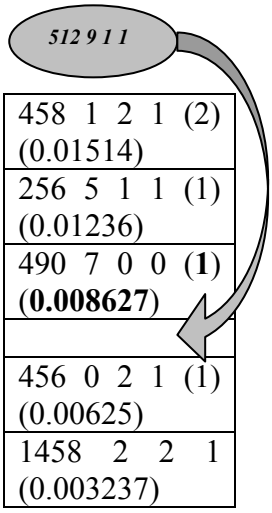
Llega el 4º elemento (cache = 4828)

(4)



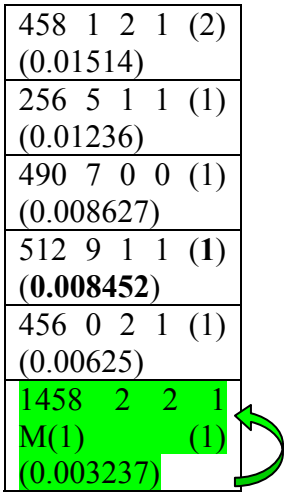
Llega el 8º elemento. (cache = 3630)

(8)



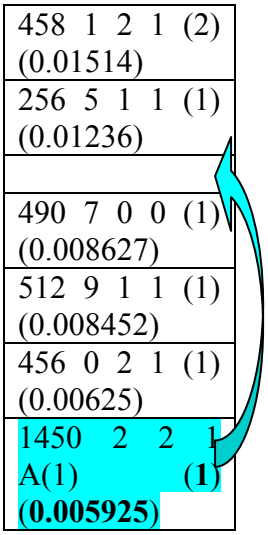
El 9º elemento es una modificación de un elemento ya existente
(cache = 3622)
(1450 2 2 1)

(9)



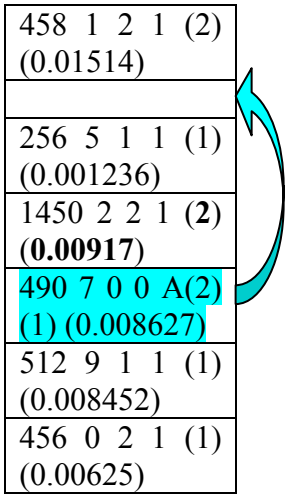
El 10º elemento es un acierto en la cache
(cache = 3622)
(1550 2 2 1)

(10)



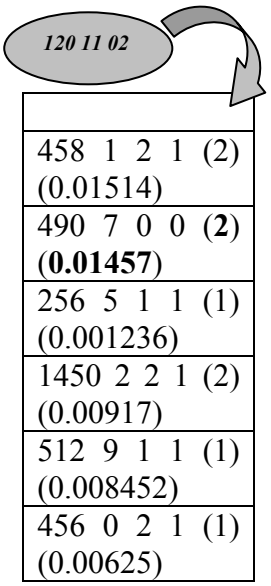
El 11º elemento es un acierto en la cache
(cache =3622)
(490 7 0 0)

(11)



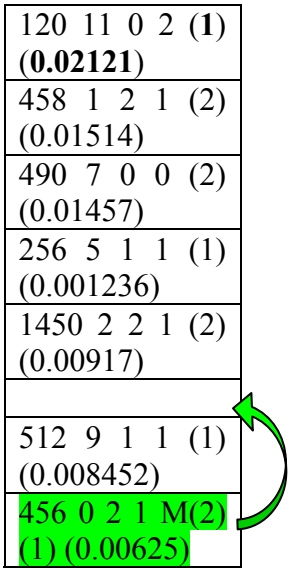
Llega el 12º elemento.
(cache = 3742)

(12)



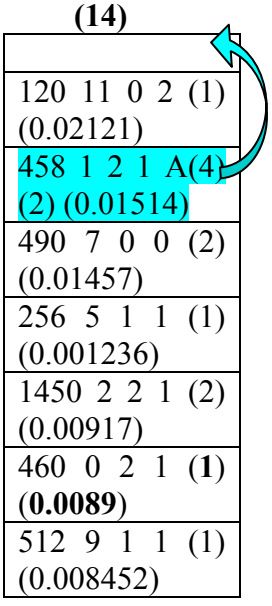
El 13º elemento es una modificación de un elemento ya existente.
(cache = 3746)
(460 0 2 1)

(13)



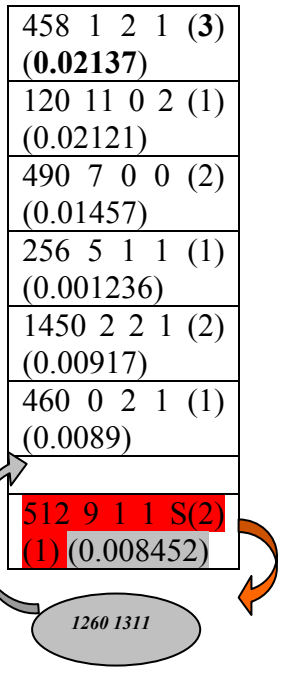
El 14º elemento es un acierto en cache
(reubicamos)
(cache = 3746)
(458 1 2 1)

(14)



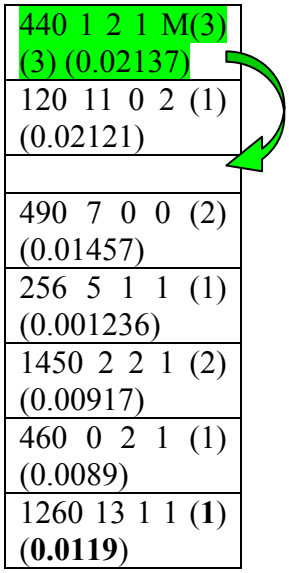
El 15º elemento no cabe, eliminamos el elemento con menor llave.
(cache = 4494)

(15)



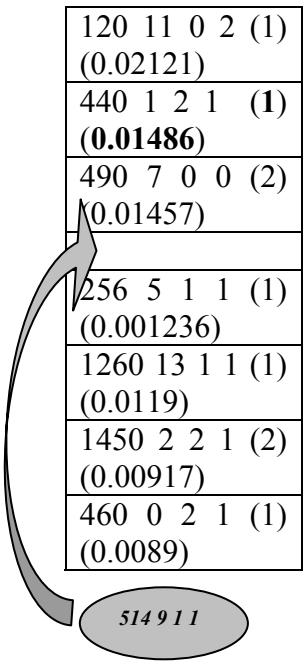
El elemento 16º es una modificación de un objeto ya existente.
(cache = 4476)
(440 1 2 1)

(16)



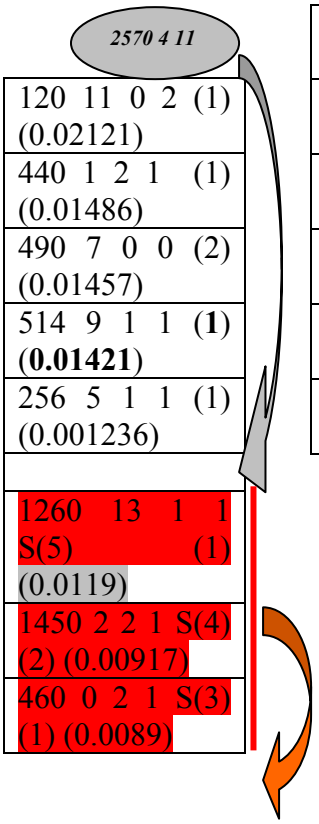
Llega el 17º elemento (cache = 4990)

(17)



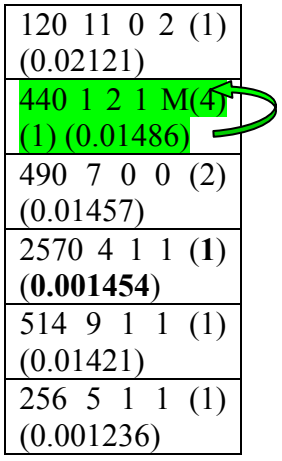
18º elemento no cabe, eliminamos objetos comenzando por el de menor contador. (cache = 4390)

(18)



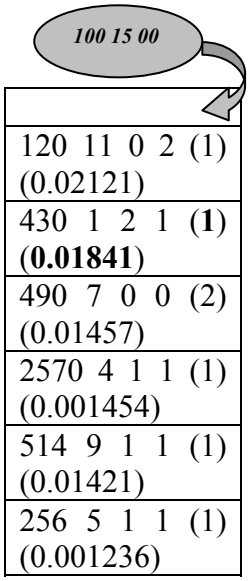
El 19º elemento es una modificación de un elemento ya existente. (cache = 4380) (430 1 2 1)

(19)



Llega el 20º elemento (cache = 4480)

(20)



Contenido final de la cache:

100 15 0 0 (1) (0.03376)
120 11 0 2 (1) (0.02121)
430 1 2 1 (1) (0.01841)
490 7 0 0 (2) (0.01457)
2570 4 1 1 (1) (0.001454)
514 9 1 1 (1) (0.01421)
256 5 1 1 (1) (0.001236)

Resultados:

Aciertos:	4
Elementos Sacados:	5
Elementos Modificados:	4
Elementos finales en cache:	7
Bytes finales en cache:	4480

Empleando el simulador:

Contenido final de la cache:

1085356864.925 **100** 15 0 0 1 0.0337705704020228 |0
1085356853.794 **120** 11 0 2 1 0.0212123421978058 |1
1085356864.905 **430** 1 1 1 1 0.0184217331927204 |2
1085356851.794 **490** 7 0 0 2 0.0145746124790526 |3
1085356863.775 **2570** 4 1 1 1 0.0145487805187543 |4
1085356862.705 **514** 9 1 1 1 0.0142086477565877 |5
1085356818.305 **256** 5 1 1 1 0.0123581755311391 |6

Resultados

GDSF (packet) 5000

Aciertos: 4
Bytes Acertados: 5603
Entradas Totales: 20
Bytes Totales: 16353
Elementos Finales en Cache: 7
Bytes de cache ocupados: 4480
HR. : 20
B.R.: 34.2628263927108
Entradas Modificadas: 4
Elementos Sacados: 5

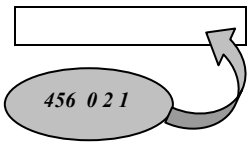
4.8. GD*(1)

Sigue el mismo comportamiento que *GDSF (1)*, variando la manera en que se calcula la llave de cada objeto $\left(L + \left(contador x \left(\frac{1}{tamaño\_objeto} \right) \right) \right)^{\frac{1}{\beta}}$

El valor de β por defecto será 0.55. Entre paréntesis se ilustrará primero el valor del contador y de la llave de un elemento. Se resaltará L sombreándolo.

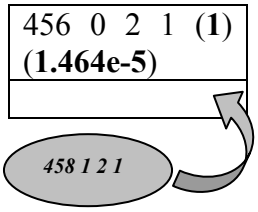
La cache en origen está vacía, llega el primer elemento (cache = 456)

(1)



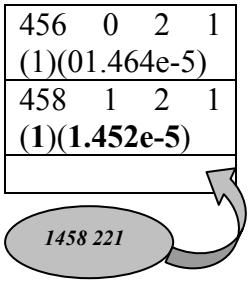
Llega el 2º elemento (cache = 914)

(2)



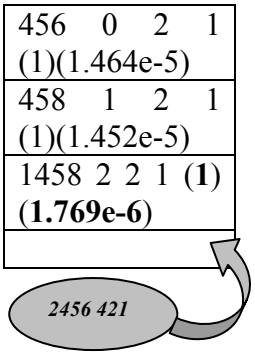
Llega el 3ª elemento (cache = 2372)

(3)



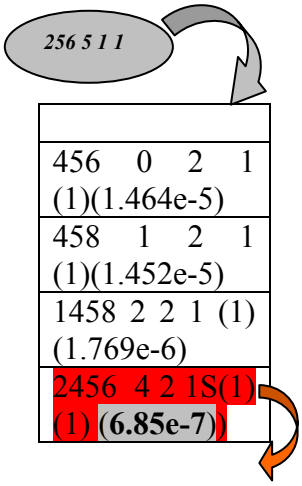
Llega el 4º elemento (cache = 4828)

(4)



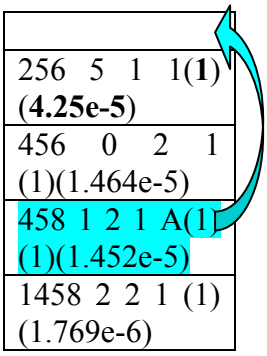
El 5º objeto no cabe, eliminamos el último elemento de la cache. (cache = 2628)

(5)



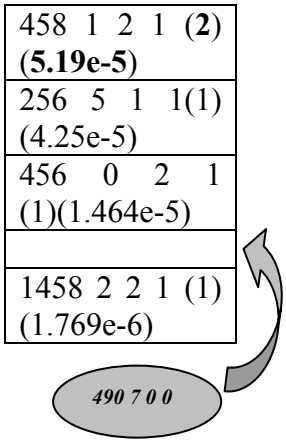
El 6º elemento es un acierto en cache (aumenta el contador) (cache = 2628) (425 1 2 1)

(6)



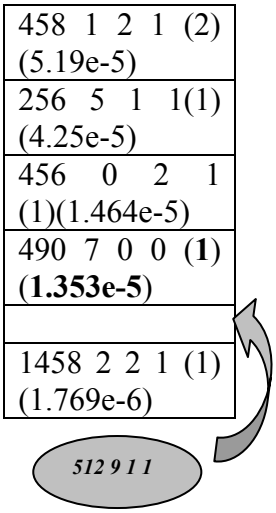
Llega el 8º elemento. (cache = 3118)

(7)



Llega el 8º elemento. (cache = 3630)

(8)



El 9º elemento es una modificación de un elemento ya existente
(cache = 3622)
(1450 2 2 1)

(9)

458 1 2 1 (2) (5.19e-5)
256 5 1 1(1) (4.25e-5)
456 0 2 1 (1)(1.464e-5)
490 7 0 0 (1) (1.353e-5)
512 9 1 1 (**1**) (**1.2544e-5**)
1458 2 2 1 M(1) (1) (1.769e-6)

El 10º elemento es un acierto en la cache
(cache = 3622)
(1550 2 2 1)

(10)

458 1 2 1 (2) (5.19e-5)
256 5 1 1(1) (4.25e-5)
456 0 2 1 (1)(1.464e-5)
490 7 0 0 (1) (1.353e-5)
512 9 1 1 (**1**) (**1.2544e-5**)
1450 2 2 1 A(1) (**1**) (**1.78e-6**)

El 11º elemento es un acierto en la cache
(cache =3622)
(490 7 0 0)

(11)

458 1 2 1 (2) (5.19e-5)
256 5 1 1(1) (4.25e-5)
456 0 2 1 (1)(1.464e-5)
490 7 0 0 A(2) (1) (1.353e-5)
512 9 1 1 (1) (1.2544e-5)
1450 2 2 1 (**2**) (**6.98e-6**)

Llega el 12º elemento.
(cache = 3742)

(12)

120 11 02

458 1 2 1 (2) (5.19e-5)
490 7 0 0 (**2**) (**4.598e-5**)
256 5 1 1(1) (4.25e-5)
456 0 2 1 (1)(1.464e-5)
512 9 1 1 (1) (1.2544e-5)
1450 2 2 1 (2) (6.98e-6)

El 13º elemento es una modificación de un elemento ya existente.
(cache = 3746)
(460 0 2 1)

(13)

120 11 0 2 (1) (1.665e-4)
458 1 2 1 (2) (5.19e-5)
490 7 0 0 (2) (4.598e-5)
256 5 1 1(1) (4.25e-5)
460 0 2 1 M(2) (1)(1.51e-5)
512 9 1 1 (1) (1.2544e-5)
1450 2 2 1 (2) (6.98e-6)

El 14º elemento es un acierto en cache
(reubicamos)
(cache = 3746)
(458 1 2 1)

(14)

120 11 0 2 (1) (1.665e-4)
458 1 2 1 A(4) (2) (5.19e-5)
490 7 0 0 (2) (4.598e-5)
256 5 1 1(1) (4.25e-5)
460 0 2 1 (1) (1.51e-5)
512 9 1 1 (1) (1.2544e-5)
1450 2 2 1 (2) (6.98e-6)

El 15º elemento no cabe, eliminamos el elemento con menor llave.
(cache = 4494)

(15)

120 11 0 2 (1) (1.665e-4)
458 1 2 1 (**2**) (**5.19e-5**)
490 7 0 0 (2) (4.598e-5)
256 5 1 1(1) (4.25e-5)
460 0 2 1 (1) (1.51e-5)
512 9 1 1 (1) (1.2544e-5)
1450 2 2 1 S(2) (2) (6.98e-6)

1260 1311

El elemento 16º es una modificación de un objeto ya existente.
(cache = 4476)
(440 1 2 1)

(16)

120 11 0 2 (1) (1.665e-4)
458 1 2 1 M(3) (**2**) (**5.19e-5**)
490 7 0 0 (2) (4.598e-5)
256 5 1 1(1) (4.25e-5)
460 0 2 1 (1) (1.51e-5)
512 9 1 1 (1) (1.2544e-5)
1260 13 1 1 (**1**) (**9.29e-6**)

El 17º elemento es una modificación de un elemento ya existente.
(cache = 4990)
(514 9 1 1)

(17)

120 11 0 2 (1) (1.665e-4)
490 7 0 0 (2) (4.598e-5)
256 5 1 1(1) (4.25e-5)
440 1 2 1 (**1**) (**2.26e-5**)
460 0 2 1 (1) (1.51e-5)
512 9 1 1 M(4) (1) (1.2544e-5)
1260 13 1 1 (**1**) (**9.29e-6**)

(9.92e-6)
18º elemento no cabe, eliminamos objetos comenzando por el de menor contador.
(cache = 4850)

(18)

120 11 0 2 (1) (1.665e-4)
490 7 0 0 (2) (4.598e-5)
256 5 1 1(1) (4.25e-5)
440 1 2 1 (1) (2.26e-5)
514 9 1 1 (**1**) (**1.87e-5**)
460 0 2 1 (1) (1.51e-5)
1260 13 1 1 S(3) (1) (9.29e-6)

2570 4 11

El 19º elemento es una modificación de un elemento ya existente.
(cache = 4840)
(430 1 2 1)

(19)

120 11 0 2 (1) (1.665e-4)
490 7 0 0 (2) (4.598e-5)
256 5 1 1(1) (4.25e-5)
440 1 2 1 M(5) (1) (2.26e-5)
514 9 1 1 (1) (1.87e-5)
460 0 2 1 (1) (1.51e-5)
2570 4 1 1 (**1**) (**9.92e-6**)

Llega el 20º elemento
(cache = 4940)

(20)

100 15 00

120 11 0 2 (1) (1.665e-4)
490 7 0 0 (2) (4.598e-5)
256 5 1 1(1) (4.25e-5)
430 1 2 1 (**1**) (**2.56e-5**)
514 9 1 1 (1) (1.87e-5)
460 0 2 1 (1) (1.51e-5)
2570 4 1 1 (1) (9.92e-6)

Contenido final de la cache:

100 15 0 0 (1) (2.4e-4)
120 11 0 2 (1) (1.665e-4)
490 7 0 0 (2) (4.598e-5)
256 5 1 1(1) (4.25e-5)
430 1 2 1 (1) (2.56e-5)
514 9 1 1 (1) (1.87e-5)
460 0 2 1 (1) (1.51e-5)
2570 4 1 1 (1)

Resultados:

Aciertos:	4
Elementos Sacados:	3
Elementos Modificados:	5
Elementos finales en cache:	8
Bytes finales en cache:	4940

Empleando el simulador:

Contenido final de la cache:

1085356864.925 **100** 15 0 0 1 0.000240305576899323 |0
1085356853.794 **120** 11 0 2 1 0.000166518217565367 |1
1085356851.794 **490** 7 0 0 2 4.59815959049795E-5 |2
1085356818.305 **256** 5 1 1 1 4.25052000712817E-5 |3
1085356864.905 **430** 1 1 1 1 2.55808984781436E-5 |4
1085356862.705 **514** 9 1 1 1 1.87614710803214E-5 |5
1085356856.134 **460** 0 2 1 1 1.50940180908804E-5 |6
1085356863.775 **2570** 4 1 1 1 9.9237409159097E-6 |

Resultados

GD*(1) 5000

Aciertos: 4
Bytes Acertados: 6117
Entradas Totales: 20
Bytes **Totales: 16353**
Elementos Finales en Cache: 8
Bytes de cache ocupados: 4940
HR. : 20
B.R.: 37.4059805540268
Entradas Modificadas: 5
Elementos Sacados: 3

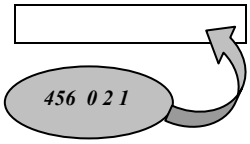
4.9 GD*(p)

Sigue el mismo comportamiento que *GDSF (1)*, variando la manera en que calculamos la clave de cada objeto $\left(L + \left(contadorx \left(\frac{2 + \frac{tamaño\_objeto}{536}}{tamaño\_objeto} \right) \right) \right)^{\frac{1}{\beta}}$

El valor de β por defecto será 0.55. Entre paréntesis ilustraremos primero el valor del contador y a continuación de la llave. Resaltaremos el valor de L sombreándolo.

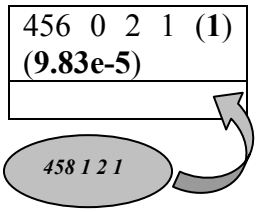
La cache en origen está vacía, llega el primer elemento (cache = 456)

(1)



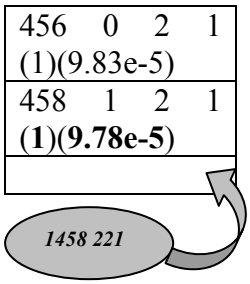
Llega el 2º elemento (cache = 914)

(2)



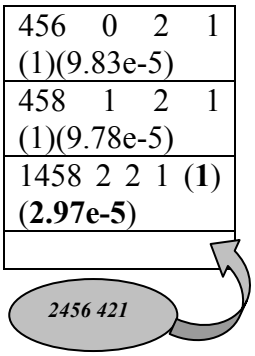
Llega el 3ª elemento (cache = 2372)

(3)



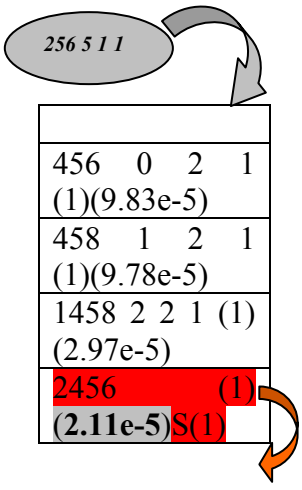
Llega el 4º elemento (cache = 4828)

(4)



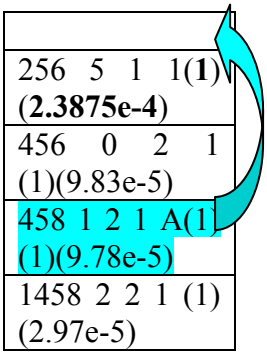
El 5º objeto no cabe, eliminamos el último elemento de la cache. (cache = 2628)

(5)



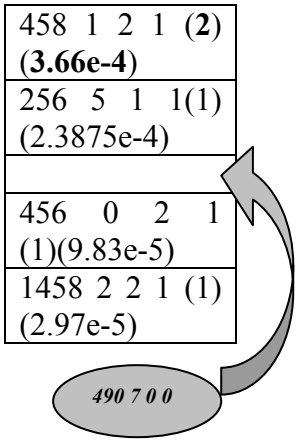
El 6º elemento es un acierto en cache(aumenta el contador) (cache = 2628) (425 1 2 1)

(6)



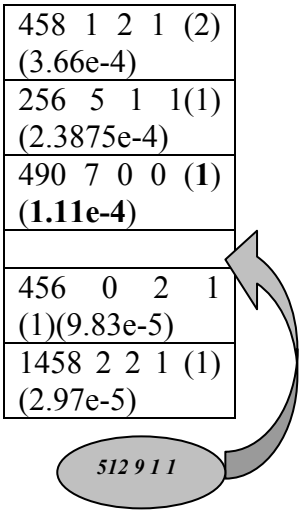
Llega el 8º elemento. (cache = 3118)

(7)



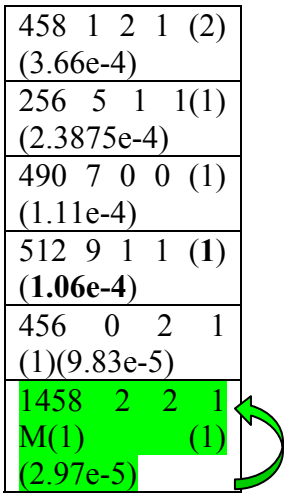
Llega el 8º elemento. (cache = 3630)

(8)



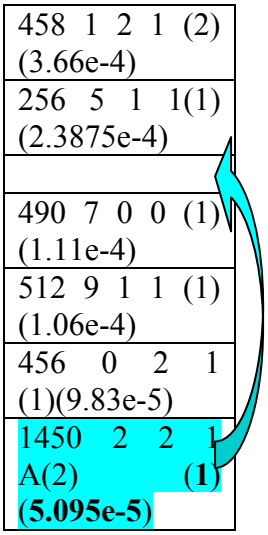
El 9º elemento es una modificación de un elemento ya existente
(cache = 3622)
(1450 2 2 1)

(9)



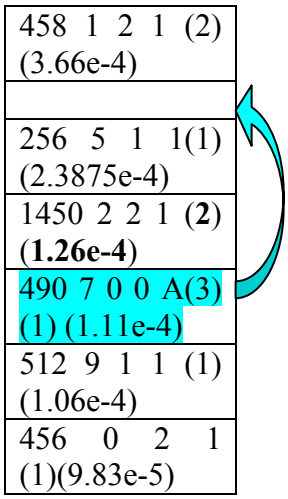
El 10º elemento es un acierto en la cache
(cache = 3622)
(1550 2 2 1)

(10)



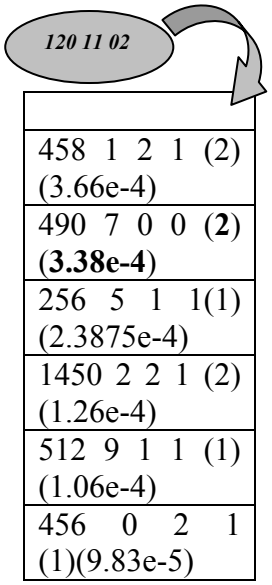
El 11º elemento es un acierto en la cache
(cache =3622)
(490 7 0 0)

(11)



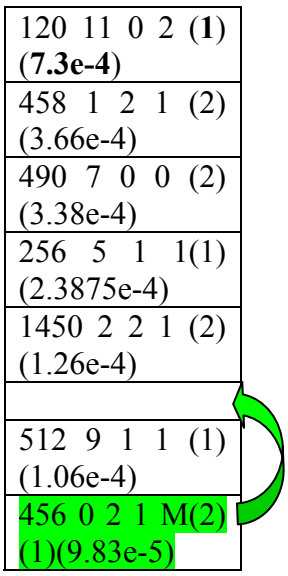
Llega el 12º elemento.
(cache = 3742)

(12)



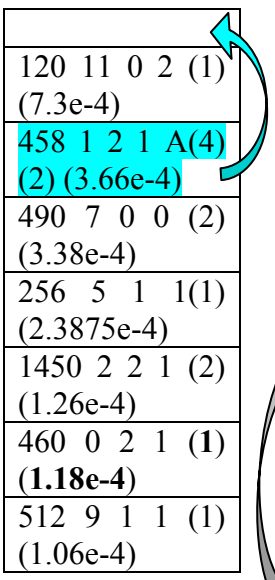
El 13º elemento es una modificación de un elemento ya existente.
(cache = 3746)
(460 0 2 1)

(13)



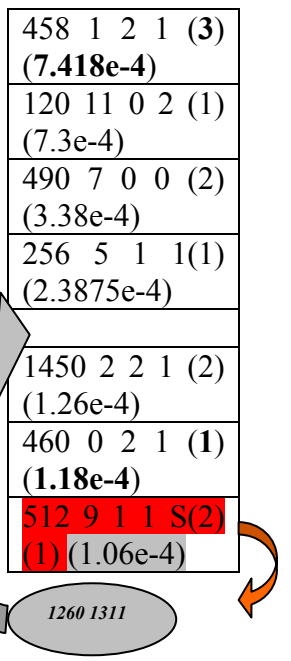
El 14º elemento es un acierto en cache
(reubicamos)
(cache = 3746)
(458 1 2 1)

(14)



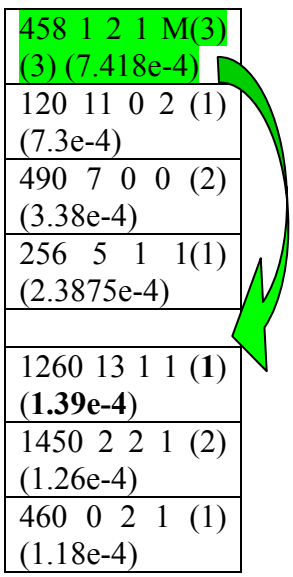
El 15º elemento no cabe, eliminamos el elemento con menor llave.
(cache = 4494)

(15)



El elemento 16º es una modificación de un objeto ya existente.
(cache = 4476)
(440 1 2 1)

(16)



Llega el 17º elemento (cache = 4546)

(17)

120 11 0 2 (1) (7.3e-4)
490 7 0 0 (2) (3.38e-4)
256 5 1 1(1) (2.3875e-4)
440 1 2 1 (**1**) (**2.09e-4**)
1260 13 1 1 (1) (1.39e-4)
1450 2 2 1 (2) (1.26e-4)
460 0 2 1 (1) (1.18e-4)

514 9 1 1

18º elemento no cabe, eliminamos objetos comenzando por el de menor contador. (cache = 4390)

(18)

120 11 0 2 (1) (7.3e-4)
490 7 0 0 (2) (3.38e-4)
256 5 1 1(1) (2.3875e-4)
440 1 2 1 (1) (2.09e-4)
514 9 1 1 (**1**) (**1.906e-4**)
1260 13 1 1 S(5) (1) (1.39e-4)
1450 2 2 1 S(4) (2) (1.26e-4)
460 0 2 1 S(3) (1) (1.18e-4)

2570 4 11

El 19º elemento es una modificación de un elemento ya existente. (cache = 4380) (430 1 2 1)

(19)

120 11 0 2 (1) (7.3e-4)
490 7 0 0 (2) (3.38e-4)
256 5 1 1(1) (2.3875e-4)
440 1 2 1 M(4) (1) (2.09e-4)
514 9 1 1 (1) (1.906e-4)
2570 4 1 1 (**1**) (**1.596e-4**)

Llega el 20º elemento (cache = 4480)

(20)

100 15 00

120 11 0 2 (1) (7.3e-4)
490 7 0 0 (2) (3.38e-4)
430 1 2 1 (**1**) (**2.45e-4**)
256 5 1 1(1) (2.3875e-4)
514 9 1 1 (1) (1.906e-4)
2570 4 1 1 (1) (1.596e-4)

Contenido final de la cache:

100 15 0 0 (1) (1.097e-3)
120 11 0 2 (1) (7.3e-4)
490 7 0 0 (2) (3.38e-4)
430 1 2 1 (1) (2.45e-4)
256 5 1 1(1) (2.3875e-4)
514 9 1 1 (1) (1.906e-4)
2570 4 1 1 (1) (1.596e-4)

Resultados:

Aciertos:	4
Elementos Sacados:	5
Elementos Modificados:	4
Elementos finales en cache:	7
Bytes finales en cache:	4480

Empleando el simulador:

Contenido final de la cache:

1085356864.925 **100** 15 0 0 1 0.00109759627137177 |0
1085356853.794 **120** 11 0 2 1 0.000730302612119596 |1
1085356851.794 **490** 7 0 0 2 0.000337772256127363 |2
1085356864.905 **430** 1 1 1 1 0.000245601559037236 |3
1085356818.305 **256** 5 1 1 1 0.000238754320235509 |4
1085356862.705 **514** 9 1 1 1 0.000190773492223843 |5
1085356863.775 **2570** 4 1 1 1 0.000160115804622543 |6

Resultados

GD*(packet) 5000

Aciertos: 4
Bytes Acertados: 5603
Entradas Totales: 20
Bytes Totales: 16353
Elementos Finales en Cache: 7
Bytes **de cache ocupados: 4480**
HR. : 20
B.R.: 34.2628263927108
Entradas Modificadas: 4
Elementos Sacados: 5

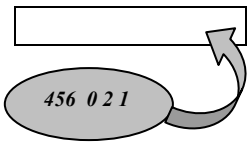
4.10. RAND

En este caso se liberará espacio en la cache eliminando objetos de manera aleatoria. Para conseguir los mismos resultados finales ambas simulaciones (automática y manual) deben eliminar los mismos objetos, por lo tanto, se utiliza primero el simulador, sacando por pantalla los índices de los objetos aleatorios eliminados, siendo éstos los mismos que se utilizan al recrear la política en la tabla.

Al eliminar de manera aleatoria, la forma de introducir o reubicar los elementos es indiferente. Se ha optado por introducir siempre al final de la tabla (zona inferior de la misma), sin reubicar los objetos acertados (ya que no se modifican en ningún campo), pero sí los modificados.

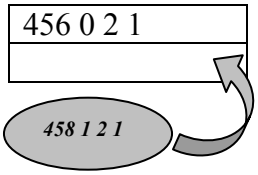
La cache en origen está vacía, llega el primer elemento (cache = 456)

(1)



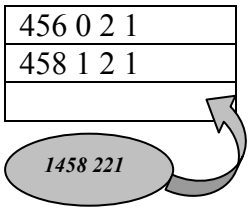
Llega el 2º elemento (cache = 914)

(2)



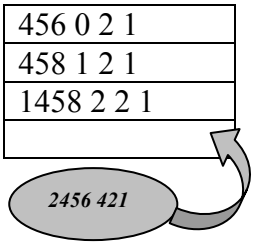
Llega el 3ª elemento (cache = 2372)

(3)



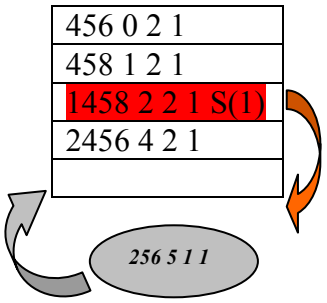
Llega el 4º elemento (cache = 4828)

(4)



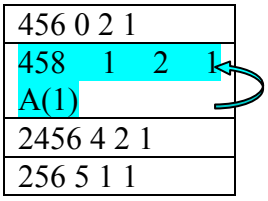
El 5º objeto no cabe, eliminamos un objeto de forma aleatoria. (cache = 3626)

(5)



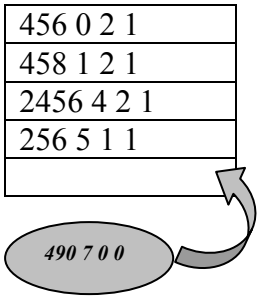
El 6º elemento es un acierto en cache (aumenta el contador) (cache = 3626) (425 1 2 1)

(6)



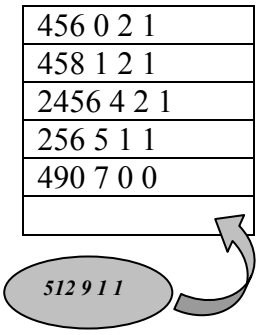
Llega el 8º elemento. (cache = 4116)

(7)



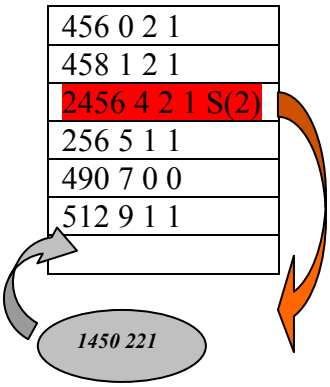
Llega el 8º elemento. (cache = 4628)

(8)



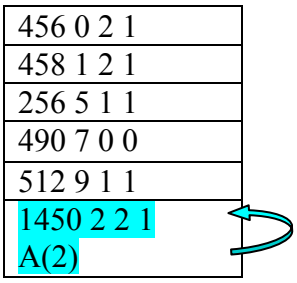
El 9º elemento no cabe, eliminamos un objeto de forma aleatoria
(cache = 3622)

(9)



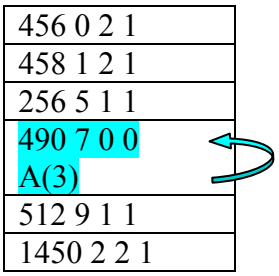
El 10º elemento es un acierto en la cache
(cache = 3622)
(1550 2 2 1)

(10)



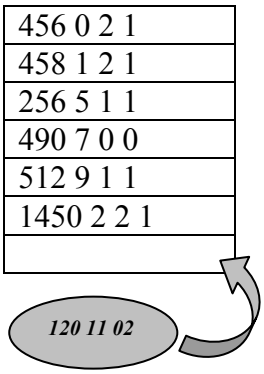
El 11º elemento es un acierto en la cache
(cache =3622)
(490 7 0 0)

(11)



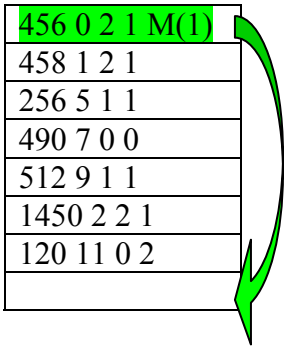
Llega el 12º elemento.
(cache = 3742)

(12)



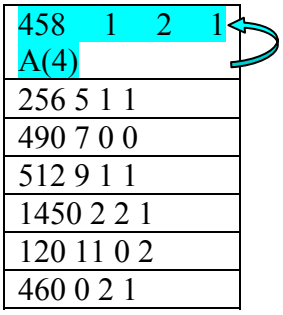
El 13º elemento es una modificación de un elemento ya existente.
(cache = 3746)
(460 0 2 1)

(13)



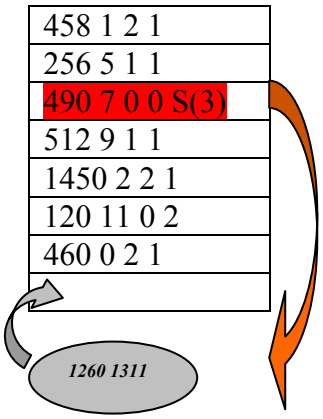
El 14º elemento es un acierto en cache
(reubicamos)
(cache = 3746)
(458 1 2 1)

(14)



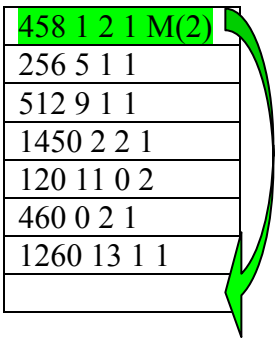
El 15º elemento no cabe, eliminamos un objeto de forma aleatoria
(cache = 4516)

(15)



El elemento 16º es una modificación de un objeto ya existente.
(cache = 4498)
(440 1 2 1)

(16)



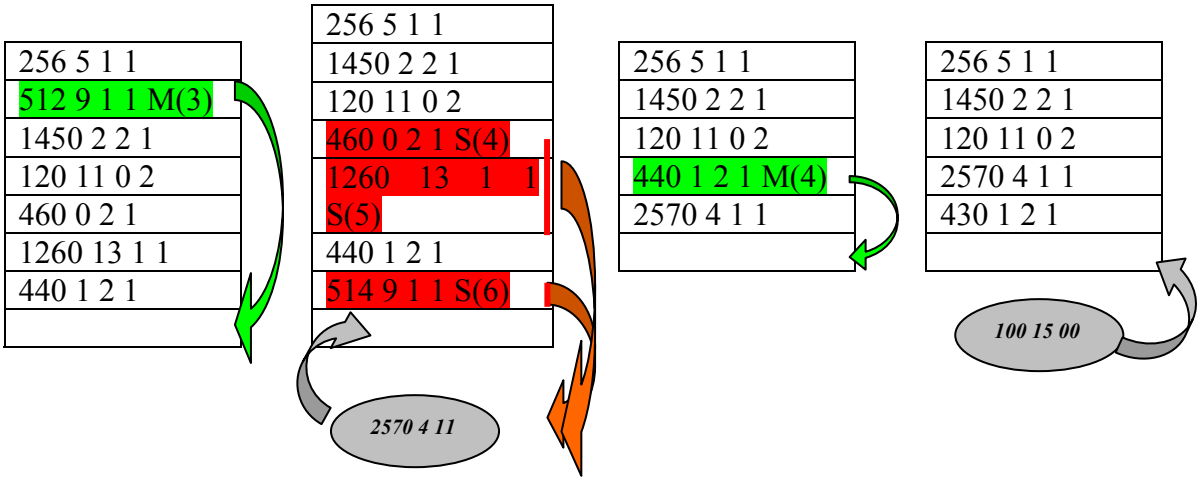
El 17º elemento es una modificación de un objeto ya existente.
(cache = 4500)
(514 9 1 1)

18º elemento no cabe, eliminamos objetos de manera aleatoria
(cache = 4836)

El 19º elemento es una modificación de un elemento ya existente.
(cache = 4826)
(430 1 2 1)

Llega el 20º elemento
(cache = 4926)

(17) **(18)** **(19)** **(20)**



Contenido final de la cache:

256 5 1 1
1450 2 2 1
120 11 0 2
2570 4 1 1
430 1 2 1
100 15 0 0

Resultados:

Aciertos:	4
Elementos Sacados:	6
Elementos Modificados:	4
Elementos finales en cache:	6
Bytes finales en cache:	4926

Empleando el simulador:

Contenido final de la cache:

1085356818.305 **256** 5 1 1 1 0 |0
1085356849.334 **1450** 2 2 1 1 0 |1
1085356853.794 **120** 11 0 2 1 0 |2
1085356863.775 **2570** 4 1 1 1 0 |3
1085356864.905 **430** 1 1 1 3 0 |4
1085356864.925 **100** 15 0 0 1 0 |5

Resultados

RAND 5000

Aciertos: 4
Bytes Acertados: 2923
Entradas Totales: 20
Bytes Totales: 16353
Elementos Finales en Cache: 6
Bytes de cache ocupados: 4926
HR. : 20
B.R.: 17.8743961352657
Entradas Modificadas: 4
Elementos Sacados: 6

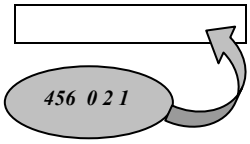
4.11. HARM

Igual a la política anterior, la diferencia radica en que con la política RAND seleccionamos objetos de manera aleatoria, con una distribución uniforme; mientras que ahora la probabilidad de seleccionar un objeto es directamente proporcional al tamaño del mismo. Para ello emplearemos un generador de números aleatorios que sigue una distribución Gaussiana, debido a lo cual ordenamos la tabla en función del tamaño de los objetos y siempre teniendo en cuenta que al tratarse de un método de eliminación aleatoria, la manera de ubicar los objetos en la tabla no es relevante.

Se puede observar como los objetos eliminados corresponden casi siempre a los de mayor tamaño.

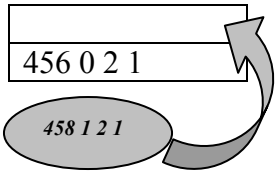
La cache en origen está vacía, llega el primer elemento (cache = 456)

(1)



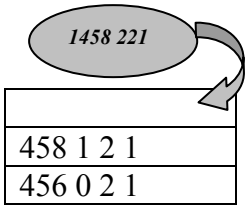
Llega el 2º elemento (cache = 914)

(2)



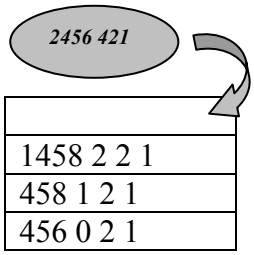
Llega el 3ª elemento (cache = 2372)

(3)



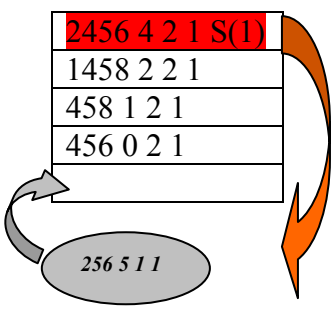
Llega el 4º elemento (cache = 4828)

(4)



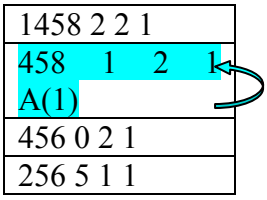
El 5º objeto no cabe, eliminamos un objeto de forma aleatoria. (cache = 3626)

(5)



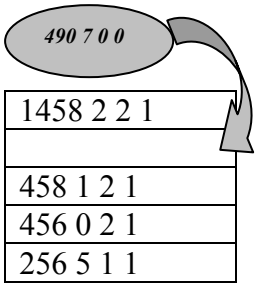
El 6º elemento es un acierto en cache (aumenta el contador) (cache = 3626) (425 1 2 1)

(6)



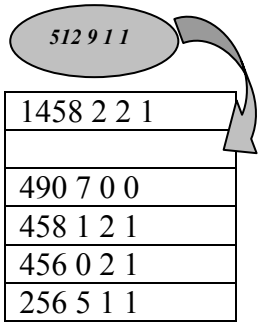
Llega el 8º elemento. (cache = 4116)

(7)



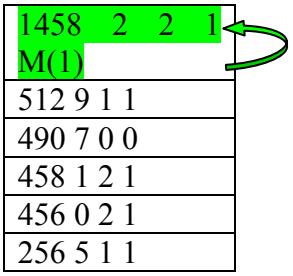
Llega el 8º elemento. (cache = 4628)

(8)



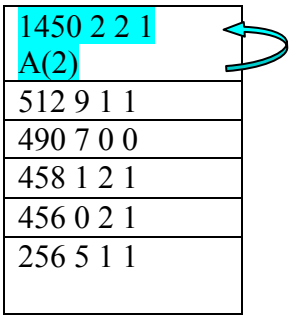
El 9º elemento es una modificación de un objeto ya existente
(cache = 3622)
(1450 2 2 1)

(9)



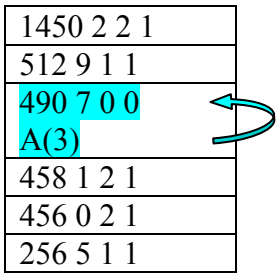
El 10º elemento es un acierto en la cache
(cache = 3622)
(1550 2 2 1)

(10)



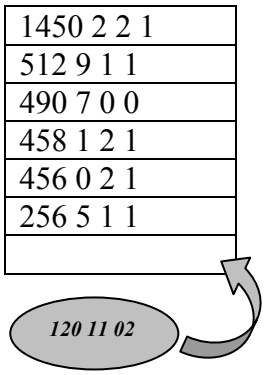
El 11º elemento es un acierto en la cache
(cache =3622)
(490 7 0 0)

(11)



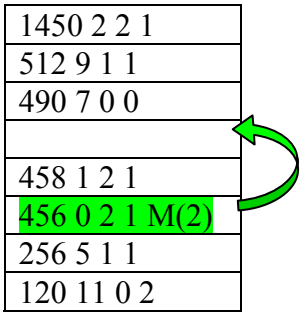
Llega el 12º elemento.
(cache = 3742)

(12)



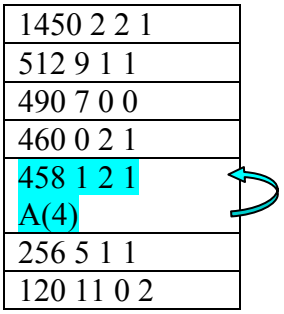
El 13º elemento es una modificación de un elemento ya existente.
(cache = 3746)
(460 0 2 1)

(13)



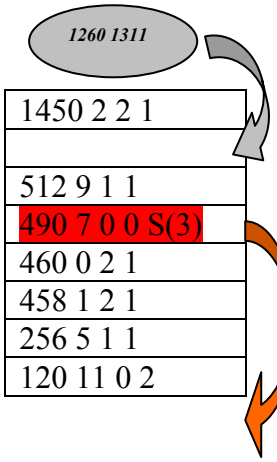
El 14º elemento es un acierto en cache
(reubicamos)
(cache = 3746)
(458 1 2 1)

(14)



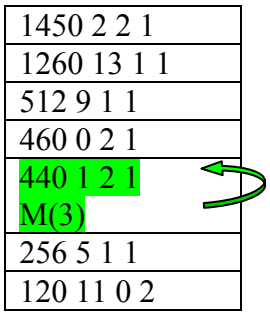
El 15º elemento no cabe, eliminamos un objeto de forma aleatoria
(cache = 4516)

(15)



El elemento 16º es una modificación de un objeto ya existente.
(cache = 4498)
(440 1 2 1)

(16)

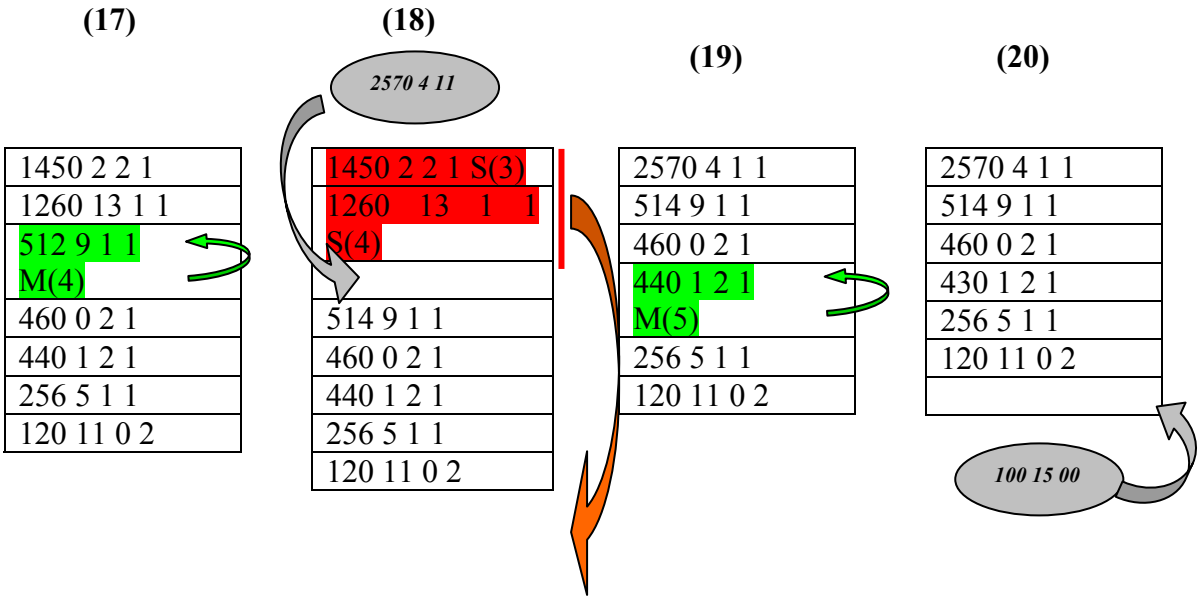


El 17º elemento es una modificación de un objeto ya existente.
(cache = 4500)
(514 9 1 1)

18º elemento no cabe, eliminamos objetos de manera aleatoria
(cache = 4836)

El 19º elemento es una modificación de un elemento ya existente.
(cache = 4826)
(430 1 2 1)

Llega el 20º elemento
(cache = 4926)



Contenido final de la cache:

2570 4 1 1
514 9 1 1
460 0 2 1
430 1 2 1
256 5 1 1
120 11 0 2
100 15 0 0

Resultados:

Aciertos:	4
Elementos Sacados:	6
Elementos Modificados:	5
Elementos finales en cache:	7
Bytes finales en cache:	4450

Empleando el simulador:

Contenido final de la cache:

1085356863.775 **2570** 4 1 1 1 2570 |0

1085356862.705 **514** 9 1 1 2 514 |1

1085356856.134 **460** 0 2 1 2 460 |2

1085356864.905 **430** 1 1 1 3 430 |3

1085356818.305 **256** 5 1 1 1 256 |4

1085356853.794 **120** 11 0 2 1 120 |5

1085356864.925 **100** 15 0 0 1 100 |6

Resultados

HARM 5000

Aciertos: 4

Bytes Acertados: 2923

Entradas Totales: 20

Bytes Totales: 16353

Elementos Finales en Cache: 7

Bytes de cache ocupados: 4450

HR. : 20

B.R.: 17.8743961352657

Entradas Modificadas: 5

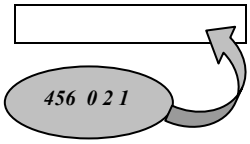
Elementos Sacados: 6

4.12. LRU-C

Política igual a la LRU, con la diferencia de que al acertar o modificar un elemento, se reinsertará en la cabeza de la tabla (zona superior) según una probabilidad igual $\left(\frac{\cos te_i}{\cos te_{\min}}\right)$ (se marcará un elemento con un doble asterisco cuando no haya que moverlo a la cabeza de la tabla), siendo $\cos te_i = \left(2 + \frac{tamaño\_objeto_i}{536}\right)$, y $\cos te_{\min}$ el menor coste de entre todos los existentes en la tabla (mostraremos siempre cual es el $\cos te_{\min}$ sombreándolo). Entre paréntesis se ilustrará el coste del elemento.

La cache en origen está vacía, llega el primer elemento (cache = 456)

(1)



Llega el 2º elemento (cache = 914)

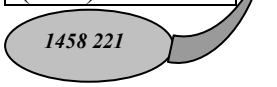
(2)

456 0 2 1 (**2.85**)

Llega el 3ª elemento (cache = 2372)

(3)

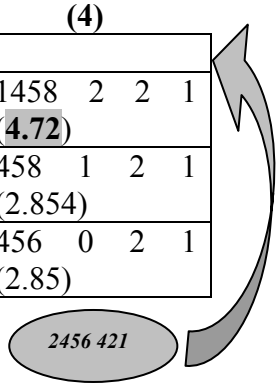
458 1 2 1 (**2.854**)
456 0 2 1 (2.85)



Llega el 4º elemento (cache = 4828)

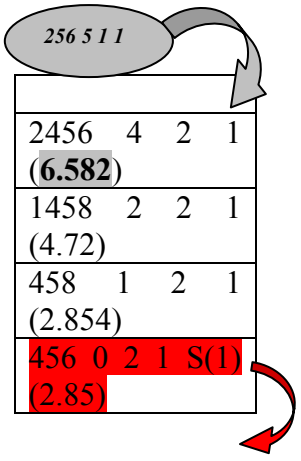
(4)

1458 2 2 1 (**4.72**)
458 1 2 1 (2.854)
456 0 2 1 (2.85)



El 5º objeto no cabe, eliminamos el último elemento de la cache. (cache = 4628)

(5)



2456 4 2 1 (**6.582**)
1458 2 2 1 (4.72)
458 1 2 1 (2.854)
456 0 2 1 S(1) (2.85)

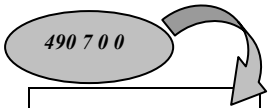
El 6º elemento es un acierto en cache (cache = 4628) (425 1 2 1)

(6)

256 5 1 1 (**2.4776**)
2456 4 2 1 (6.582)
1458 2 2 1 (4.72)
458 1 2 1 A(1)(2.854)

El 7º objeto no cabe, eliminamos el último elemento de la cache. (cache = 3660)

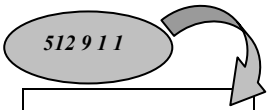
(7)



458 1 2 1 (**2.854**)
256 5 1 1 (2.4776)
2456 4 2 1 (6.582)
1458 2 2 1 S(2) (4.72)

Llega el 8º elemento. (cache = 4172)

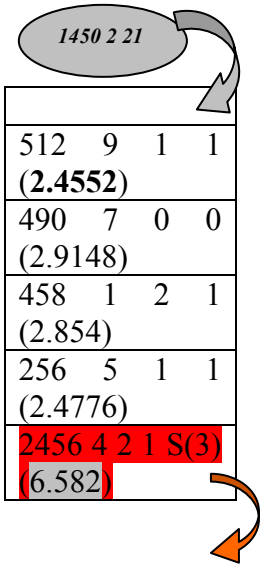
(8)



490 7 0 0 (**2.9148**)
458 1 2 1 (2.854)
256 5 1 1 (2.4776)
2456 4 2 1 (6.582)

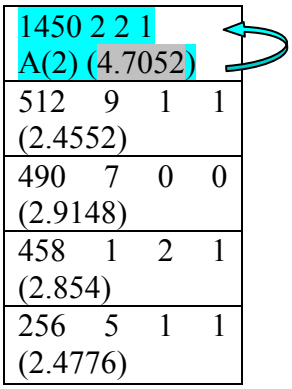
El 9º elemento no cabe, eliminamos el último objeto de la cache
(cache = 3166)
(1450 2 2 1)

(9)



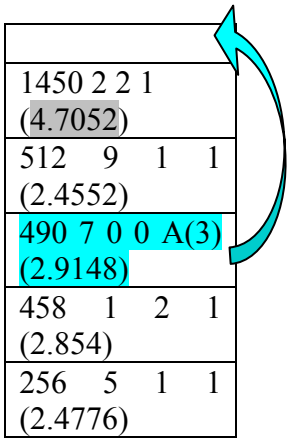
El 10º elemento es un acierto en la cache
(cache = 3166)
(1550 2 2 1)

(10)



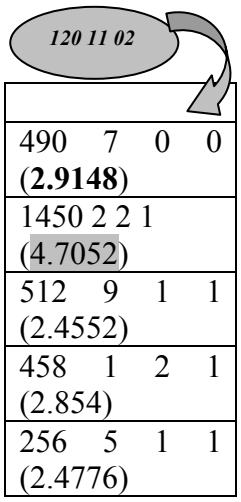
El 11º elemento es un acierto en la cache
(cache =3166)
(490 7 0 0)

(11)



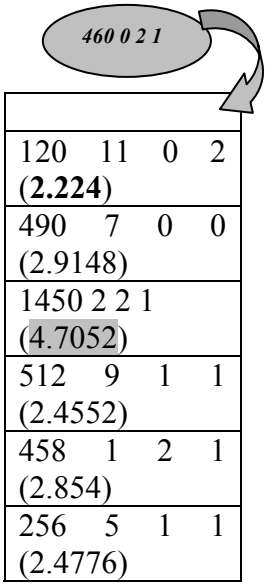
Llega el 12º elemento.
(cache = 3286)

(12)



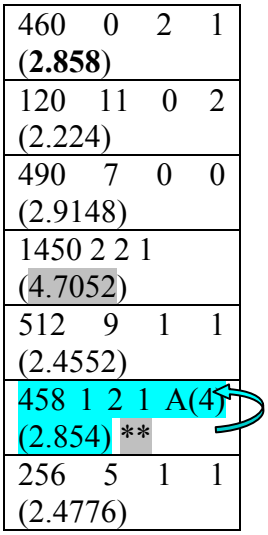
Llega el 13º elemento.
(cache = 3746)

(13)



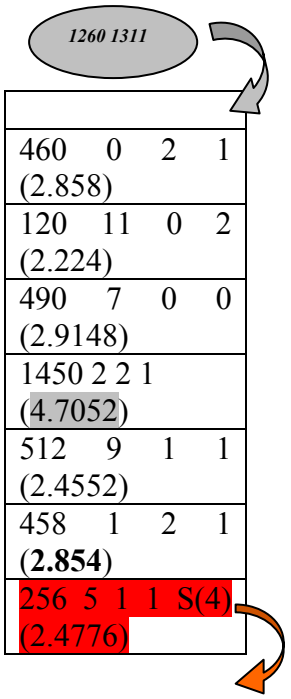
El 14º elemento es un acierto en cache
(cache = 3746)
(458 1 2 1)

(14)



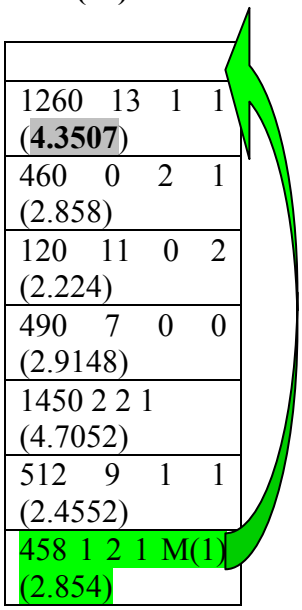
El 15º elemento no cabe, eliminamos el último objeto de la cache.
(cache = 4750)

(15)



El elemento 16º es una modificación de un objeto ya existente.
(cache = 4732)
(440 1 2 1)

(16)



El 17º elemento es una modificación de un objeto ya existente.
(cache = 4734)
(514 9 1 1)

18º elemento no cabe, eliminamos objetos comenzando por el final.
(cache = 4850)

El 19º elemento es una modificación de un elemento ya existente.
(cache = 4840)
(430 1 2 1)

Llega el 20º elemento
(cache = 4940)

(17)

440 1 2 1 (**2.821**)
1260 13 1 1 (4.3507)
460 0 2 1 (2.858)
120 11 0 2 (2.224)
490 7 0 0 (2.9148)
1450 2 2 1 (4.7052)
512 9 1 1 M(2) (2.4552) **

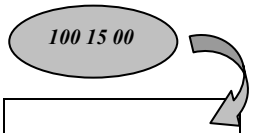
(18)

440 1 2 1 (2.821)
1260 13 1 1 (4.3507)
460 0 2 1 (2.858)
120 11 0 2 (2.224)
490 7 0 0 S(7) (2.9148)
1450 2 2 1 S(6) (4.7052)
514 9 1 1 S(5) (**2.4552**)

(19)

2570 4 1 1 (6.7947)
440 1 2 1 M(3) (2.821) **
1260 13 1 1 (4.3507)
460 0 2 1 (2.858)
120 11 0 2 (2.224)

(20)



2570 4 1 1 (6.7947)
430 1 2 1 (**2.821**)
1260 13 1 1 (4.3507)
460 0 2 1 (2.858)
120 11 0 2 (2.224)

Contenido final de la cache:

100 15 0 0 (2.184)
2570 4 1 1 (6.7947)
430 1 2 1 (2.821)
1260 13 1 1 (4.3507)
460 0 2 1 (2.858)
120 11 0 2 (2.224)

Resultados:

Aciertos:	4
Elementos Sacados:	7
Elementos Modificados:	3
Elementos finales en cache:	6
Bytes finales en cache:	4940

Empleando el simulador:

Contenido final de la cache:

1085356864.925 **100** 15 0 0 1 0 |0
1085356863.775 **2570** 4 1 1 1 0 |1
1085356864.905 **430** 1 1 1 4 0 |2
1085356862.657 **1260** 13 1 1 1 0 |3
1085356856.134 **460** 0 2 1 1 0 |4
1085356853.794 **120** 11 0 2 1 0 |5

Resultados

LRU-C 5000

Aciertos: 4
Bytes Acertados: 2923
Entradas Totales: 20
Bytes Totales: 16353
Elementos Finales en Cache: 6
Bytes de cache ocupados: 4940
HR. : 20
B.R.: 17.8743961352657
Entradas Modificadas: 3
Elementos Sacados: 7

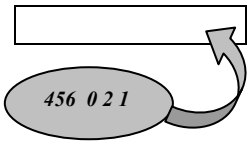
4.13. LRU-S

Presenta el mismo comportamiento que LRU-C, en éste caso se calculará la probabilidad de mover un elemento a la cabeza como el cociente $\left(\frac{tamaño_{\min}}{tamaño_i}\right)$ (marcaremos un elemento con un doble asterisco cuando no haya que moverlo a la cabeza de la tabla), siendo $tamaño_{\min}$ el tamaño del menor elemento de la tabla, y $tamaño_i$ el tamaño del elemento que se ha acertado/modificado.

Tal y como sucedió en la política LRU-C, se hizo necesario utilizar primero el simulador para luego simular la política manualmente. Se resaltará también el $tamaño_{\min}$ de cada iteración sombreándolo.

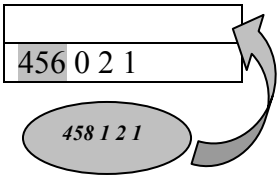
La cache en origen está vacía, llega el primer elemento (cache = 456)

(1)



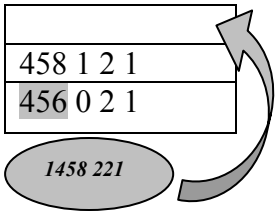
Llega el 2º elemento (cache = 914)

(2)



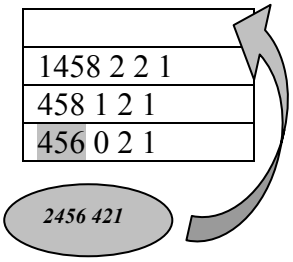
Llega el 3ª elemento (cache = 2372)

(3)



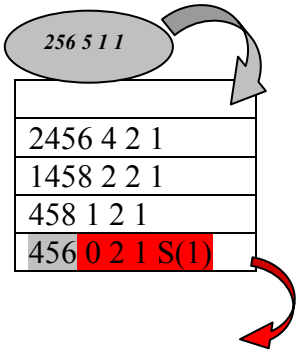
Llega el 4º elemento (cache = 4828)

(4)



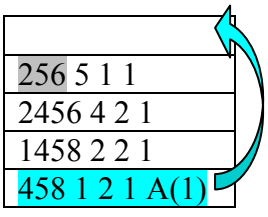
El 5º objeto no cabe, eliminamos el último elemento de la cache. (cache = 4628)

(5)



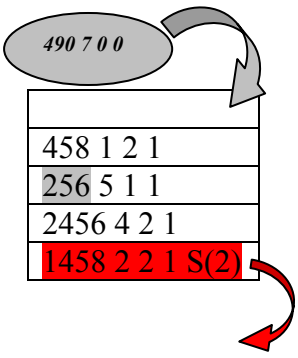
El 6º elemento es un acierto en cache (cache = 4628) (425 1 2 1)

(6)



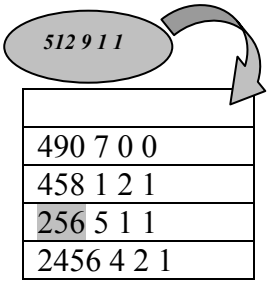
El 7º objeto no cabe, eliminamos el último elemento de la cache. (cache = 3660)

(7)



Llega el 8º elemento. (cache = 4172)

(8)



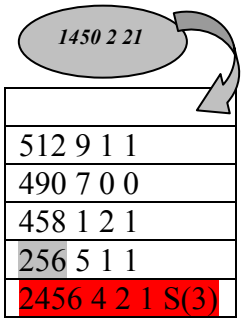
El 9º elemento no cabe, eliminamos el último objeto de la cache (cache = 3166) (1450 2 2 1)

El 10º elemento es un acierto en la cache (cache = 3166) (1550 2 2 1)

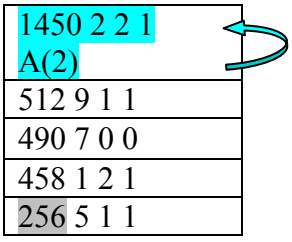
El 11º elemento es un acierto en la cache (cache =3166) (490 7 0 0)

Llega el 12º elemento. (cache = 3286)

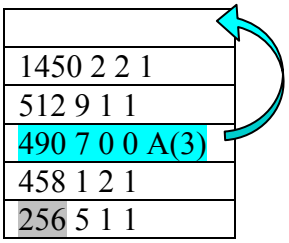
(9)



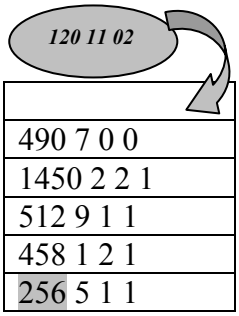
(10)



(11)



(12)



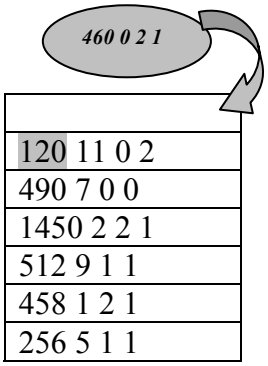
Llega el 13º elemento. (cache = 3746)

El 14º elemento es un acierto en cache (cache = 3746) (458 1 2 1)

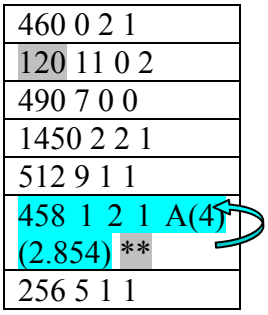
El 15º elemento no cabe, eliminamos el último objeto de la cache. (cache = 4750)

El elemento 16º es una modificación de un objeto ya existente. (cache = 4732) (440 1 2 1)

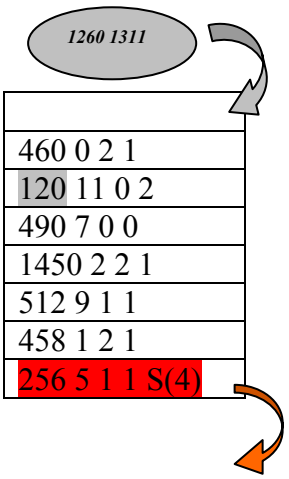
(13)



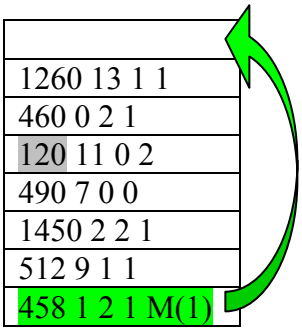
(14)



(15)



(16)



El 17º elemento es una modificación de un objeto ya existente.
(cache = 4734)
(514 9 1 1)

18º elemento no cabe, eliminamos objetos comenzando por el final.
(cache = 4850)

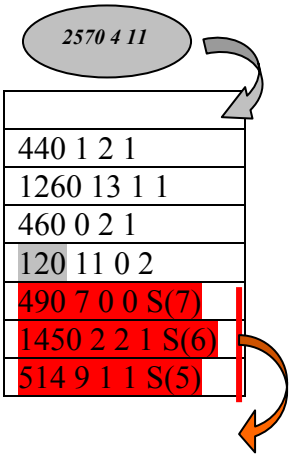
El 19º elemento es una modificación de un elemento ya existente.
(cache = 4840)
(430 1 2 1)

Llega el 20º elemento
(cache = 4940)

(17)

440 1 2 1
1260 13 1 1
460 0 2 1
120 11 0 2
490 7 0 0
1450 2 2 1
512 9 1 1 M(2) **

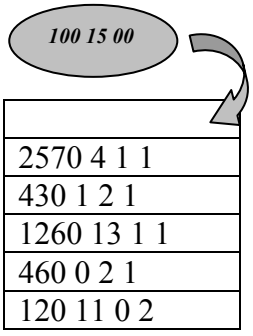
(18)



(19)

2570 4 1 1
440 1 2 1 M(3) **
1260 13 1 1
460 0 2 1
120 11 0 2

(20)



Contenido final de la cache:

100 15 0 0
2570 4 1 1
430 1 2 1
1260 13 1 1
460 0 2 1
120 11 0 2

Resultados:

Aciertos:	4
Elementos Sacados:	7
Elementos Modificados:	3
Elementos finales en cache:	6
Bytes finales en cache:	4940

Empleando el simulador:

Contenido final de la cache:

1085356864.925 **100** 15 0 0 1 0 |0
1085356863.775 **2570** 4 1 1 1 0 |1
1085356864.905 **430** 1 1 1 4 0 |2
1085356862.657 **1260** 13 1 1 1 0 |3
1085356856.134 **460** 0 2 1 1 0 |4
1085356853.794 **120** 11 0 2 1 0 |5

Resultados

LRU-S 5000

Aciertos: 4
Bytes Acertados: 2923
Entradas Totales: 20
Bytes Totales: 16353
Elementos Finales en Cache: 6
Bytes de cache ocupados: 4940
HR. : 20
B.R.: 17.8743961352657
Entradas Modificadas: 3
Elementos Sacados: 7

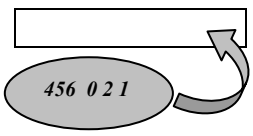
4.14. RRGVF

En esta política se insertará siempre al final de la tabla (zona inferior de la misma), tanto elementos nuevos en la cache como aquellos que han originado un acierto o una modificación. La eliminación de un objeto se realizará buscando 1º un índice aleatorio, a partir de ahí seleccionaremos N objetos y se guardarán los M con un menor contador (esa será la que se usará como función 'utilidad' de un objeto), para eliminar los elementos menos 'útiles'. En sucesivas eliminaciones, se seleccionarán N-M elementos partiendo de un índice aleatorio y se guardarán los M de menor contador de referencia de entre estos N-M y los M que ya se habían guardado (estos elementos 'candidatos' a ser eliminados los acompañaremos con índices sombreados que indiquen el orden en que serán eliminados).

En este caso N=3 y M=2(M se calcula a partir de N). Si se eliminan todos los objetos que tenemos guardados y aún así es necesario liberar más espacio se realizarán nuevas iteraciones para seleccionar nuevos N-M elementos.

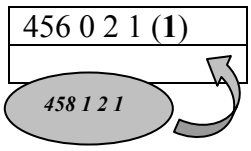
La cache en origen está vacía, llega el primer elemento (cache = 456)

(1)



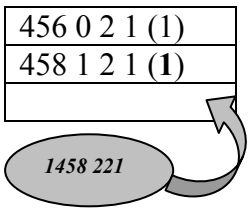
Llega el 2º elemento (cache = 914)

(2)



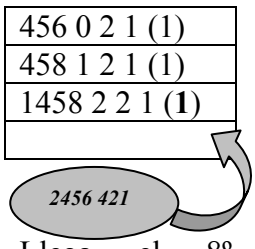
Llega el 3ª elemento (cache = 2372)

(3)



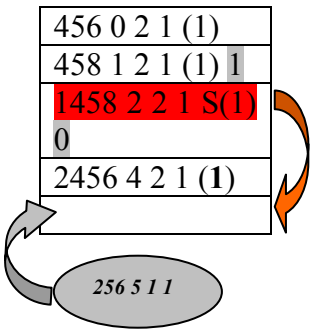
Llega el 4º elemento (cache = 4828)

(4)



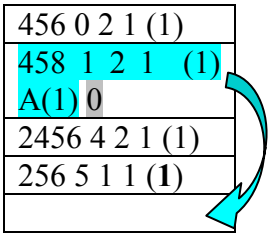
El 5º objeto no cabe, eliminamos un objeto de forma aleatoria. (cache = 3626)

(5)



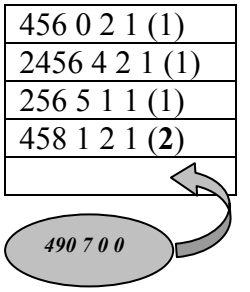
El 6º elemento es un acierto en cache (aumenta el contador) (cache = 3626) (425 1 2 1)

(6)



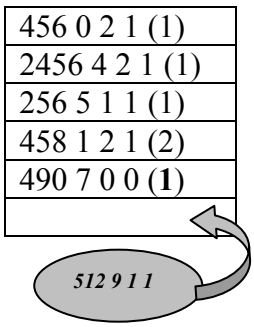
Llega el 8º elemento. (cache = 4116)

(7)



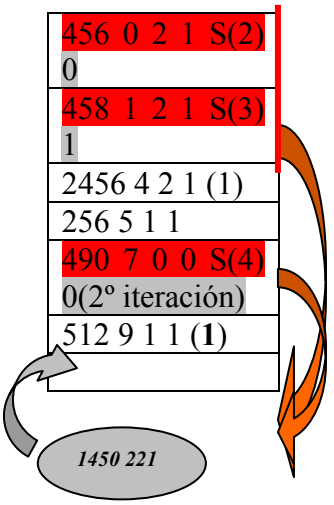
Llega el 8º elemento. (cache = 4628)

(8)



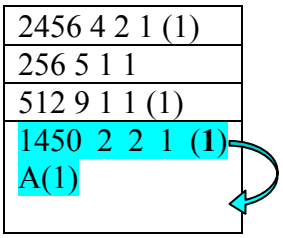
El 9º elemento no cabe, eliminamos objetos de forma aleatoria (cache = 4674)

(9)



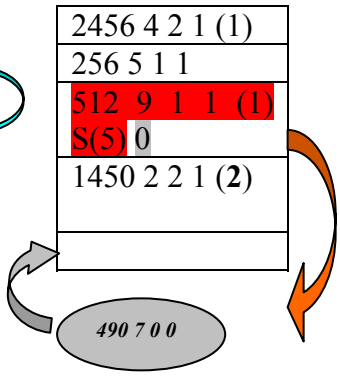
El 10º elemento es un acierto en la cache (cache = 4674) (1550 2 2 1)

(10)



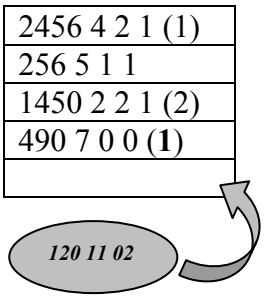
El 11º elemento no cabe, eliminamos un objeto de forma aleatoria. (cache =4652)

(11)



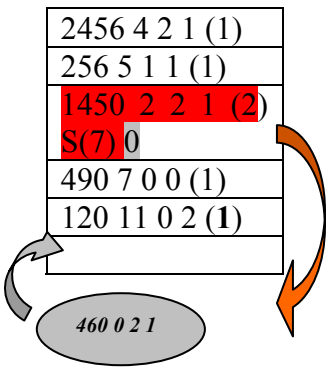
Llega el 12º elemento. (cache = 4772)

(12)



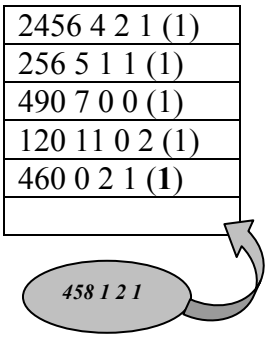
El 13º elemento no cabe, eliminamos un objeto de forma aleatoria (cache = 3782) (460 0 2 1)

(13)



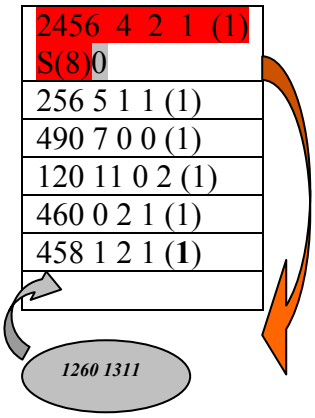
Llega el 14º elemento. (cache = 4240) (458 1 2 1)

(14)



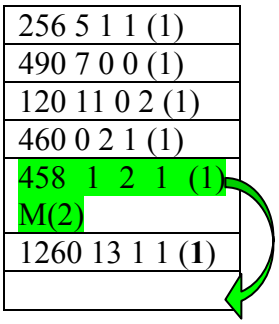
El 15º elemento no cabe, eliminamos un objeto de forma aleatoria (cache = 3044)

(15)



El elemento 16º es una modificación de un objeto ya existente. (cache = 3026) (440 1 2 1)

(16)



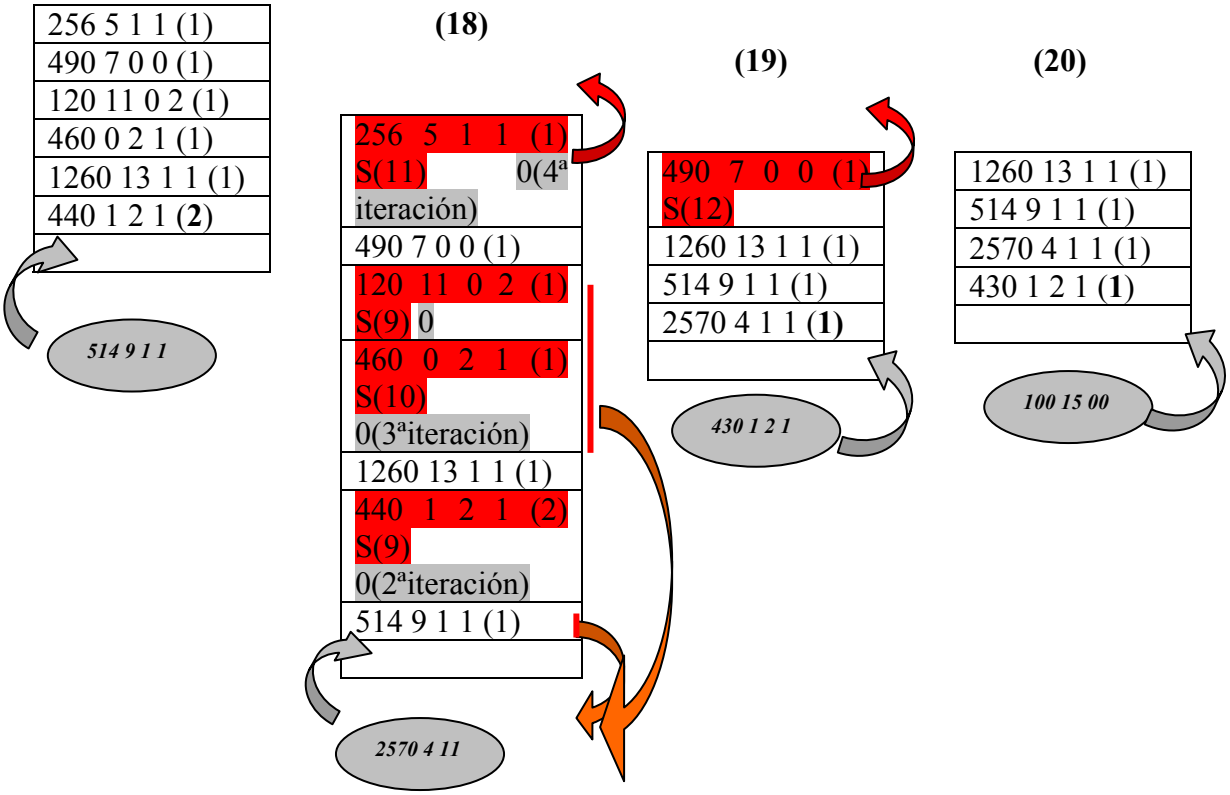
Llega el 17º elemento (cache = 3540)

(17)

18º elemento no cabe, eliminamos objetos de manera aleatoria (cache = 4834)

El 19º elemento no cabe, eliminamos un objeto de forma aleatoria. (cache = 4774) (430 1 2 1)

Llega el 20º elemento (cache = 4874)



Contenido final de la cache:

1260 13 1 1 (1)
514 9 1 1 (1)
2570 4 1 1 (1)
430 1 2 1 (1)
100 15 0 0 (1)

Resultados:

Aciertos:	2
Elementos Sacados:	12
Elementos Modificados:	1
Elementos finales en cache:	5
Bytes finales en cache:	4874

Empleando el simulador:

Contenido final de la cache:

1085356862.657 **1260** 13 1 1 1 0 |0
1085356862.705 **514** 9 1 1 1 0 |1
1085356863.775 **2570** 4 1 1 1 0 |2
1085356864.905 **430** 1 1 1 1 0 |3
1085356864.925 **100** 15 0 0 1 0 |4

Resultados

RRGVF 5000

Aciertos: 2
Bytes Acertados: 1975
Entradas Totales: 20
Bytes Totales: 16353
Elementos Finales en Cache: 5
Bytes de cache ocupados: 4874
HR. : 10
B.R.: 12.0772946859903
Entradas Modificadas: 1
Elementos Sacados: 12

CAPÍTULO 5: Comparación del Rendimiento de las Políticas de Reemplazo

El presente capítulo realiza una detallada comparación del rendimiento de las diversas políticas implementadas.

Para ello, se ha presentado una misma muestra de tráfico de entrada para todas las políticas y se han estudiado los resultados conseguidos ante diversos tamaños de *cache.* A raíz de estos resultados conseguidos por cada política, se extraerán una serie de conclusiones, es decir, se realizará una estimación del escenario más conveniente para cada una de las estrategias.

A continuación se presentarán una serie de conceptos previos, que será interesante recalcar para comprender mejor como se ha realizado cada simulación

5.1. CONCEPTOS PREVIOS

Como se ha dicho, para la realización de las simulaciones se ha empleado una misma muestra de entrada. De igual manera, el tamaño de la caché empleado será también el mismo en todos los casos, y oscilará entre el 5 y 40% del tamaño total de la muestra de entrada.

Para realizar el cálculo de este tamaño total de la muestra de entrada se realizó la simulación de una política cualquiera; pero considerando la *cache* que emplea infinita. De tal manera, todos los objetos que lleguen al sistema, serán insertados, incrementando el tamaño de la *cache*.

¿Qué algoritmo emplear para realizar esta primera simulación?

Esta es una decisión irrelevante, cualquiera sería buena. Se optó por una *LRU,* debido a su mayor sencillez, que hace que sea un algoritmo rápido y eficaz.

Con todo, el tamaño total obtenido sería de 40.426.585.400 bytes, es decir, unos 40 GB. Por tanto, por cada política, se realizarán diversas simulaciones suponiendo tamaños de *cache* de:

- 40% → 16.170.634.160 bytes
- 30% → 12.127.975.620 bytes
- 20% → 8.085.317.080 bytes
- 10% → 4.042.658.540 bytes
- 5% → 2.021.329.270 bytes

Con cada uno de ellos, y para cada política, obtendremos unos resultados de *HR y BHR,* que serán las variables con las que se realizarán las gráficas pertinentes, y se compararán una política con otra.

5.1.1. CALENTAMIENTO DE *CACHE*

Este es un concepto bastante importante a la hora de presentar las simulaciones posteriores.

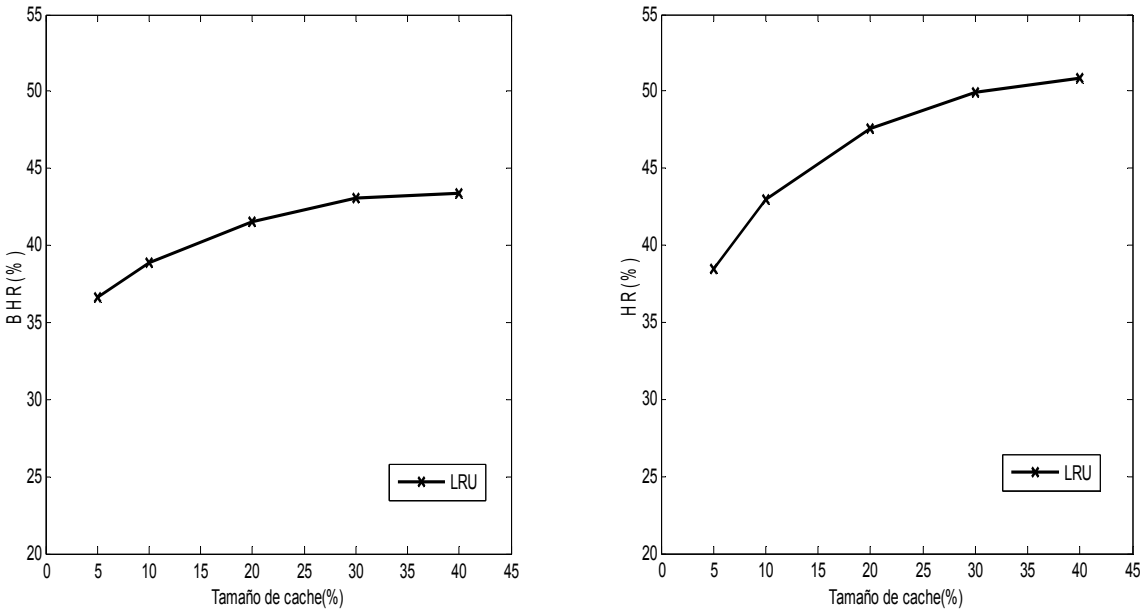
Radica en el hecho de suponer que la *cache* está inicialmente llena. En caso de comenzar con un sistema vacío, los primeros documentos causarían siempre fallos de *cache,* (los primeros documentos no favorecerán el incremento de nuestras variables a comparar, el *HR* y *HBR*) y tan sólo cuando la *cache* estuviera parcial o totalmente llena, comenzarían a producirse aciertos.

De tal manera, si provocamos un cierto calentamiento en la *cache*, la tasa de aciertos (o de bytes acertados) aumentará, y simulará mejor el comportamiento de un sistema real, ya que éste estará en estas mismas condiciones.

Dicho todo esto, se establecerá un tamaño del calentamiento del 50% del tamaño total de la muestra de entrada, es decir, 20.213.292.700 bytes. Así cuando el tamaño de la *cache* alcance o supere el del citado calentamiento, se resetearán sus variables de control, tales como el número de elementos llegados, el de aciertos, elementos modificados o sacados. O lo que es lo mismo, su *HR* y *HBR* empezará a computarse a partir de dicho momento.

5.2. LRU

A continuación se presentarán gráficas referentes al *HR* y *BHR* conseguidos con la estrategia *LRU*, para los diversos tamaños de *cache*:



a) Hit Ratio **b) Byte Hit Ratio**

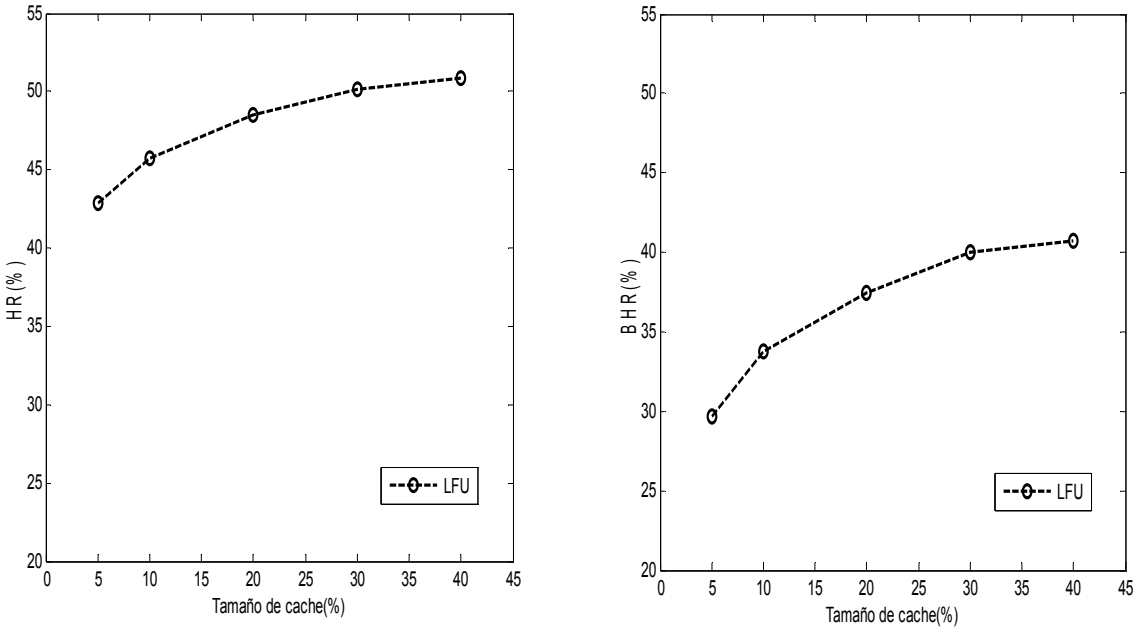
Figura 5.1. Simulación de LRU

A primera vista, las primeras conclusiones que se pueden extraer son evidentes, a medida que aumenta el tamaño de la *cache*, mayor será la probabilidad de producirse un acierto, por lo tanto, mayor serán tanto el *BR* como el *HBR*.

Ésta será la política de reemplazo que se empleará como base para realizar la comparación con el resto de estrategias.

5.3. LFU

En este apartado se mostraran los resultados obtenidos para la política de reemplazo LFU, primera de las políticas basadas en frecuencia que se analizará y que, por tanto, primará a los documentos que más veces se hayan referenciado, independientemente de su tamaño, o popularidad actual:



a) Hit Ratio **b) Byte Hit Ratio**

Figura 5.2. Simulación de LFU

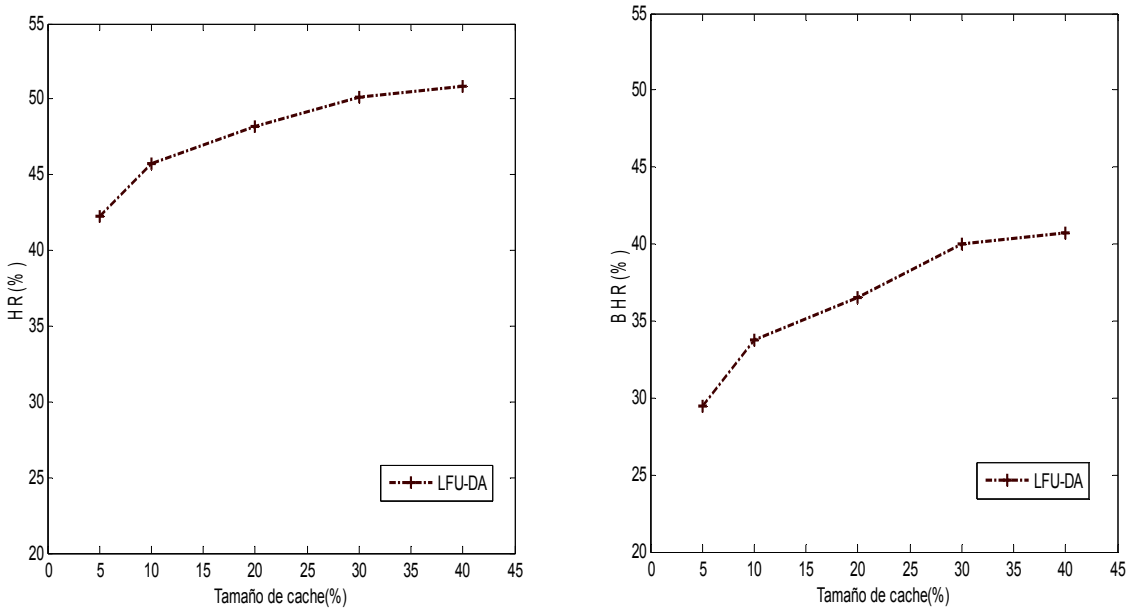
En esta ocasión se puede observar cómo el *HR* es muy similar al alcanzado en la estrategia *LRU*, tan sólo resulta algo mayor cuando el tamaño de la cache es menor, pero sin variaciones drásticas.

En cuanto al *BHR*, se comprueba el caso contrario. Cuando la cache es menor, el *BHR* del *LFU* es bastante inferior al del *LRU*, tendiendo a normalizarse con este a medida que el tamaño de la cache aumenta.

Una posible explicación a este comportamiento pudiera ser que en este caso se presenta una muestra de entrada en la que los elementos más 'populares' sean de menor tamaño. De tal forma, la cache estará compuesta por numerosos documentos, por lo que el *HR* será algo mayor, aunque de menor tamaño, por lo que el *BHR* cae.

5.4. LFU-DA

En este caso, la simulación para los diversos tamaños de *cache* arroja los siguientes resultados:



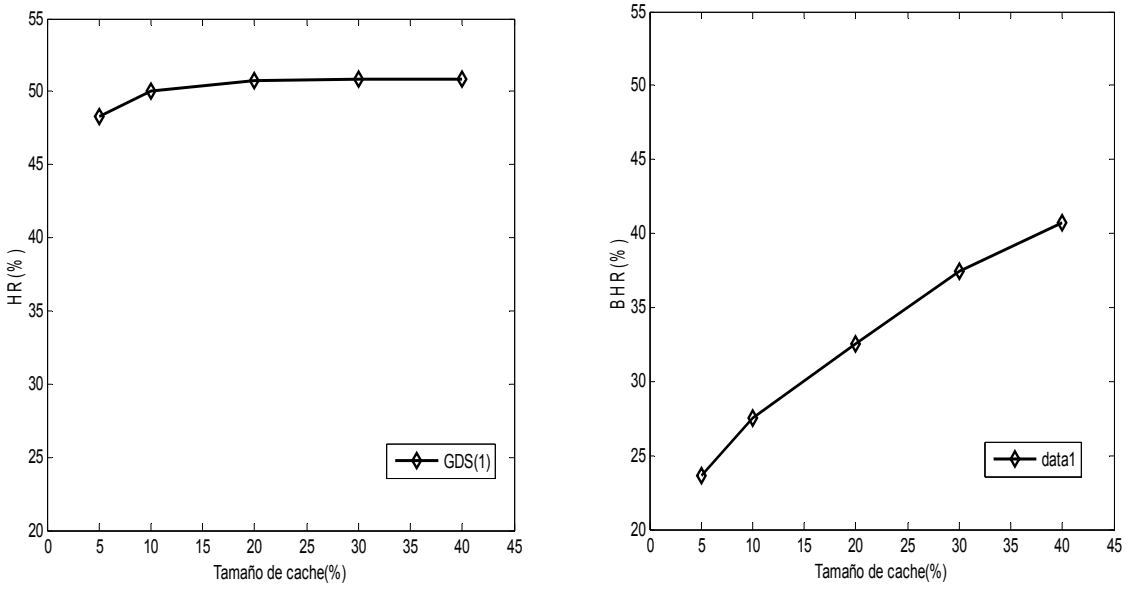
a) Hit Ratio **b) Byte Hit Ratio**

Figura 5.3. Simulación de LFU-DA

Se observa que los resultados obtenidos en la política LFU como en la LFU-DA son muy parecidos, debido a la gran similitud de ambos algoritmos.

5.5 GDS (1)

Los resultados obtenidos al realizar las simulaciones con la política *GDS (1)* son:



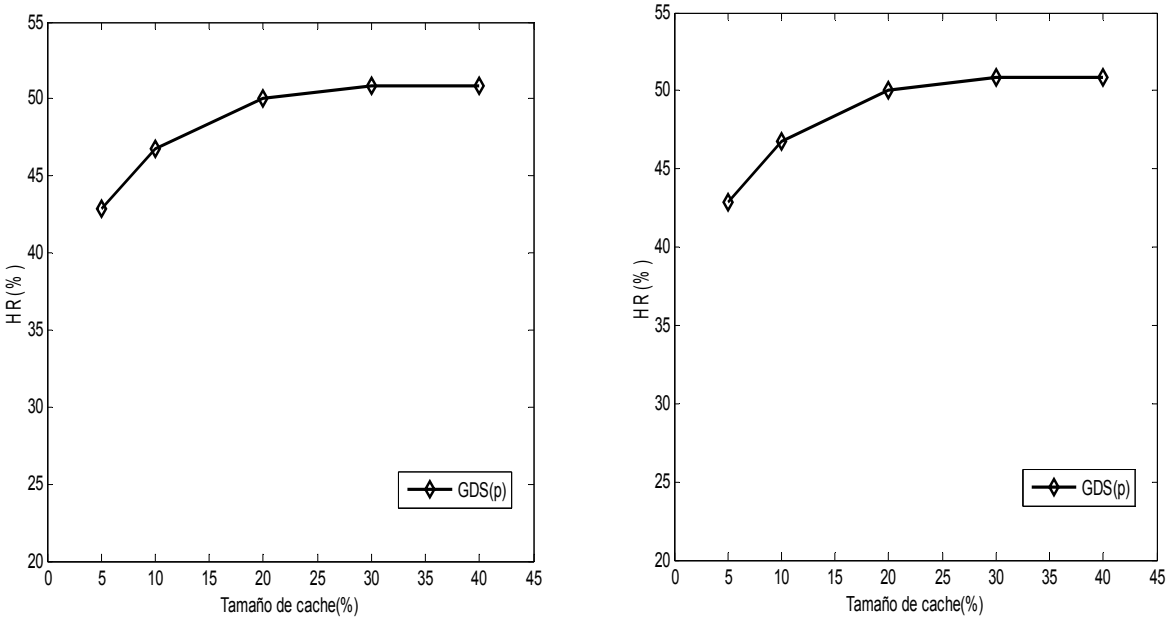
a) Hit Ratio **b) Byte Hit Ratio**

Figura5.4.Simulación de GDS (1)

Se puede observar, tal y como se comentó en el *capítulo 2*, que el hecho de emplear una función coste constante e igual a 1 parece conseguir mejores tasas de *HR*(los elementos más propensos a ser eliminados serán los de mayor tamaño, por lo que se almacenarán más elementos aunque de menor tamaño) a costa de perjudicar el *BHR*.

5.6. GDS (p)

Y si se emplea la función de coste *packets:*



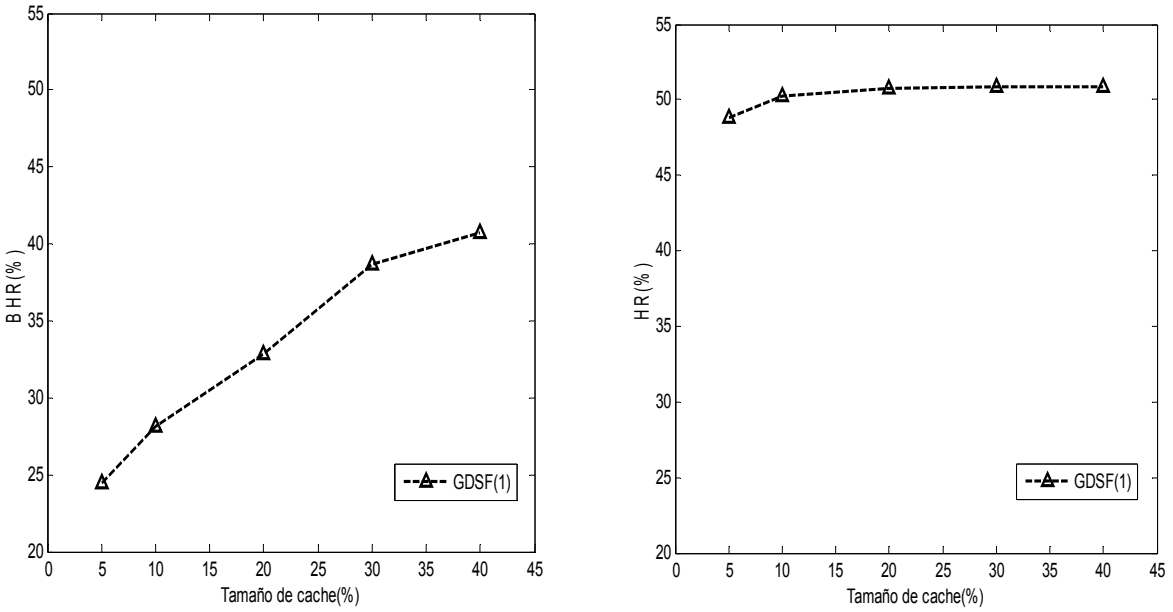
a) Hit Ratio **b) Byte Hit Ratio**

Figura 5.5. Byte Hit Ratio en GDS (p)

Como corresponde a la función de coste empleada, se observa que se consiguen mejores resultados en cuanto al *BHR* que en *GDS (1)*; sin embargo, el *HR* cae algo, sobre todo para tamaños de cache pequeños.

5.7. GDSF (1)

Tras realizar las pertinentes simulaciones para esta estrategia:



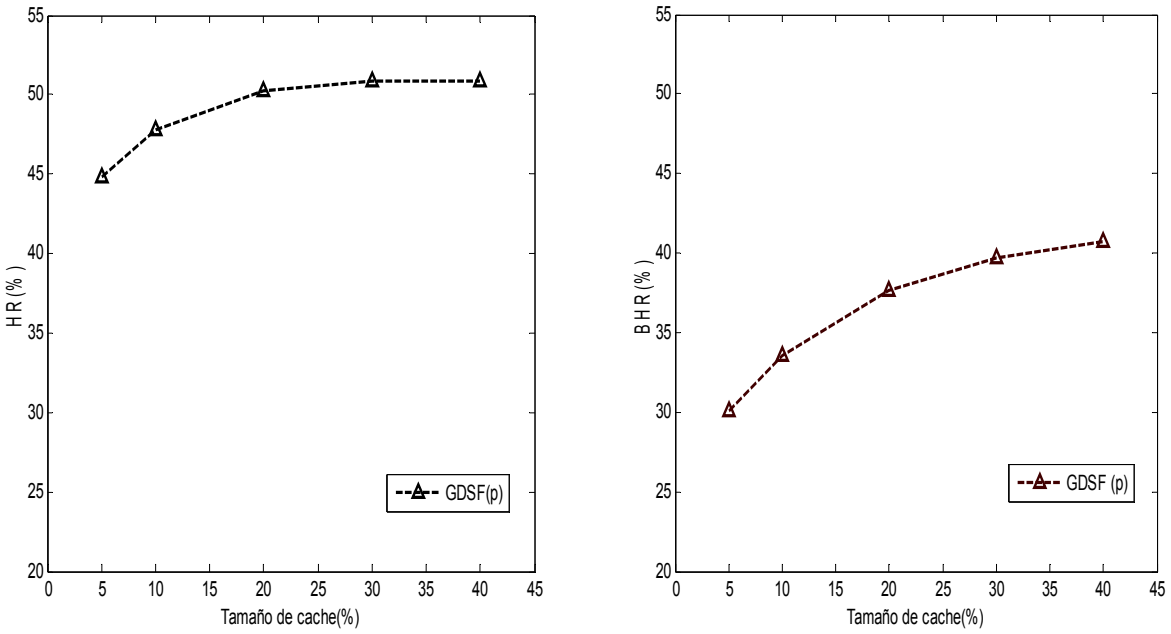
a) Hit Ratio **b) Byte Hit Ratio**

Figura 5.6. Byte Hit Ratio en GDSF (1)

Se puede observar como el *HR* es casi idéntico al conseguido con la política *GDS (1)* (y por supuesto mayor al de *GDS (p)*), de lo cual se extrae la especial relevancia que tiene el hecho de elegir una función coste u otra, sobre el resto de factores.

5.8. GDSF (p)

Si se emplea la función de coste *packets:*



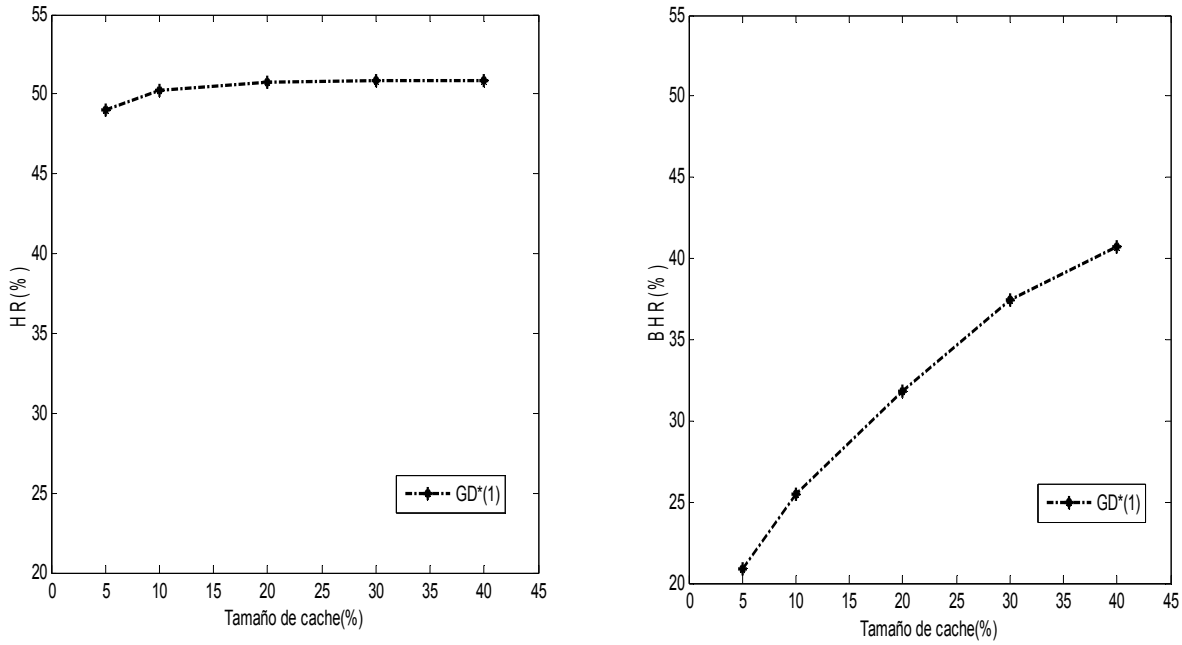
a) Hit Ratio **b) Byte Hit Ratio**

Figura 5.7. Byte Hit Ratio en GDSF (p)

Como era de esperar, se alcanzan tasas mayores de *BHR*, perjudicando el *HR*.

5.9. GD* (1)

Los resultados obtenidos en este caso son:



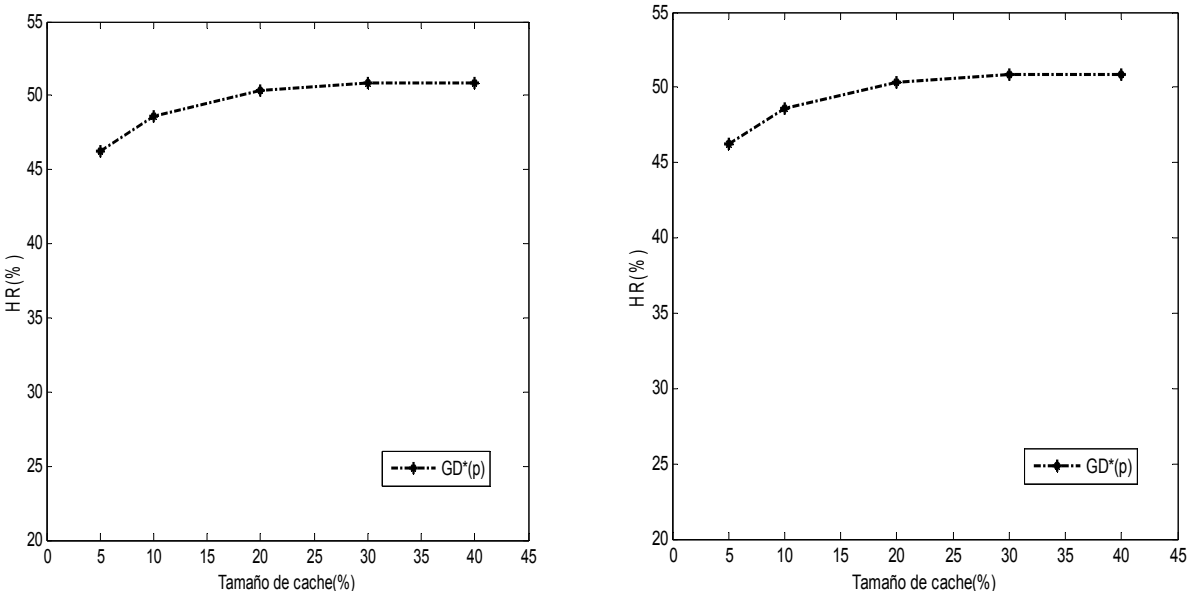
a) Hit Ratio **b) Byte Hit Ratio**

Figura 5.8. Byte Hit Ratio en GD*(1)

Como era de esperar, esta política alcanza los mejores resultados en cuanto al *HR*, prácticamente idénticos a los obtenidos con *GDS (1)* y *GDSF (1)*; por el contrario, el *BHR* es el más bajo de entre todas las políticas simuladas hasta ahora. Cabe resaltar que la β empleada para realizar estas simulaciones es de 0.55.

5.10. GD* (p)

Si se emplea una función de coste *packets*:



a) Hit Ratio **b) Byte Hit Ratio**

Figura 5.9. Byte Hit Ratio en GD*(1)

Como cabía esperar la política GD*(p) presenta un apto comportamiento en lo referente al BHR, a costa de empeorar en cierta medida el HR obtenido, como es habitual en una política *packet.* La β empleada para realizar estas simulaciones es de 0.55.

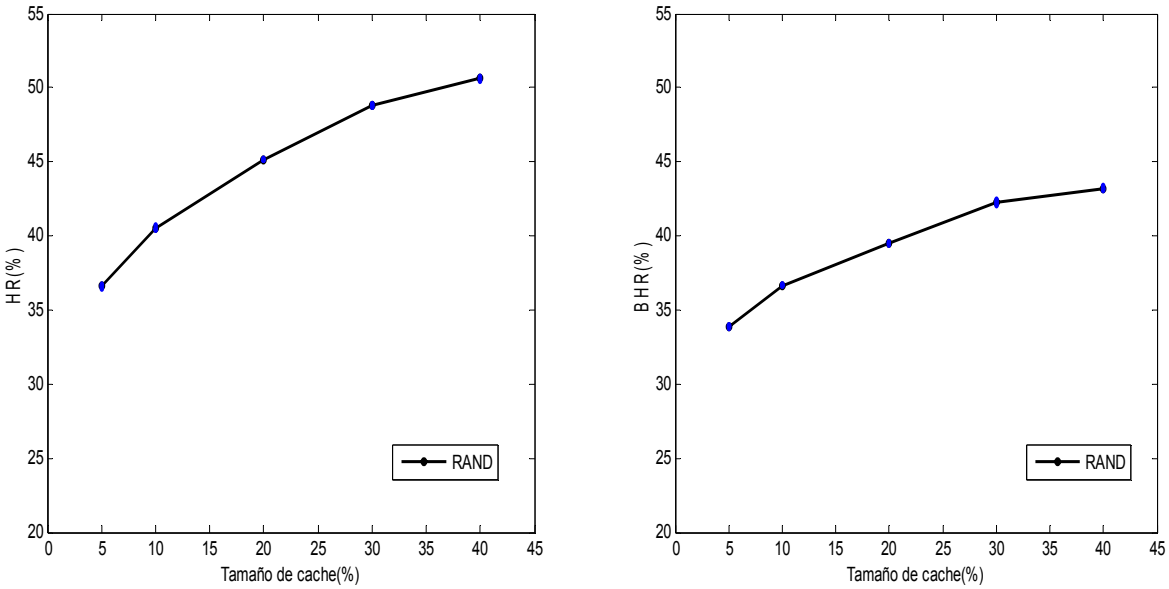
5.11. RAND

Con esta, empiezan las simulaciones de políticas de naturaleza aleatoria. Debido a lo cual, una simulación determinada arrojará resultados diversos cada vez que se ejecute, es decir, no encontraremos unos resultados definitivos que mostrar.

Por ello y como solución, se realizaron 5 simulaciones distintas, por cada política y tamaño de *cache*, de manera que se obtuvieron 5 resultados distintos para el *HR* y *BHR.*

La gráfica se construyó representando el valor medio de entre estas 5 simulaciones,

A continuación de ésta, se mostrará una ampliación de la gráfica para cada tamaño de *cache,* de manera que se puedan observar los resultados apuntados por cada simulación.



a) Hit Ratio **b) Byte Hit Ratio**

Figura 5.10. Hit Ratio en RAND

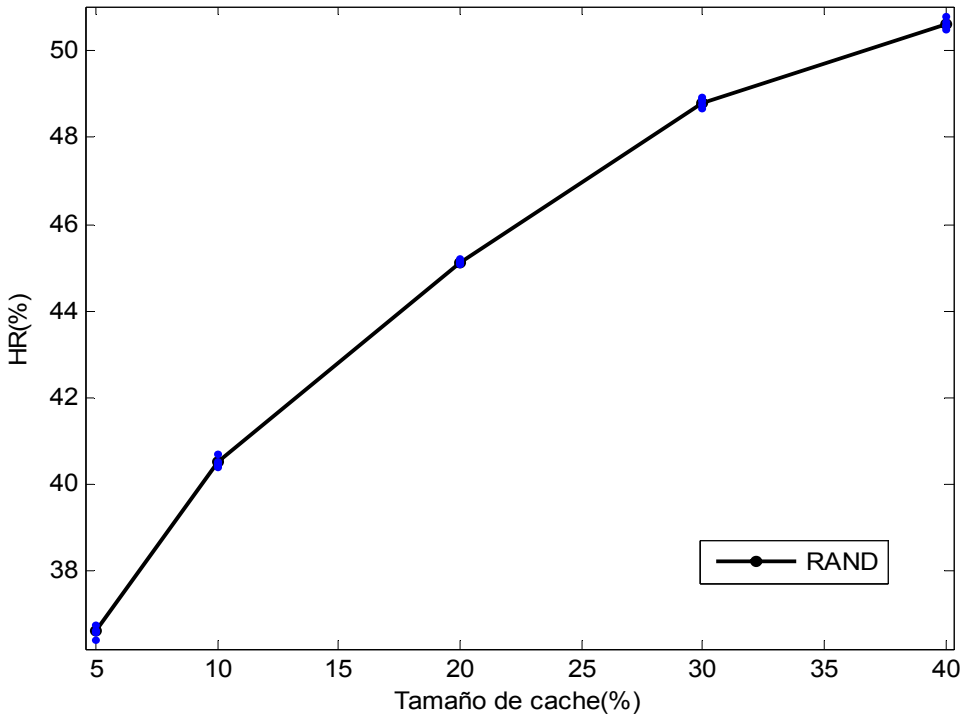


Figura 5.11. Ampliación del Hit Ratio en *RAND*

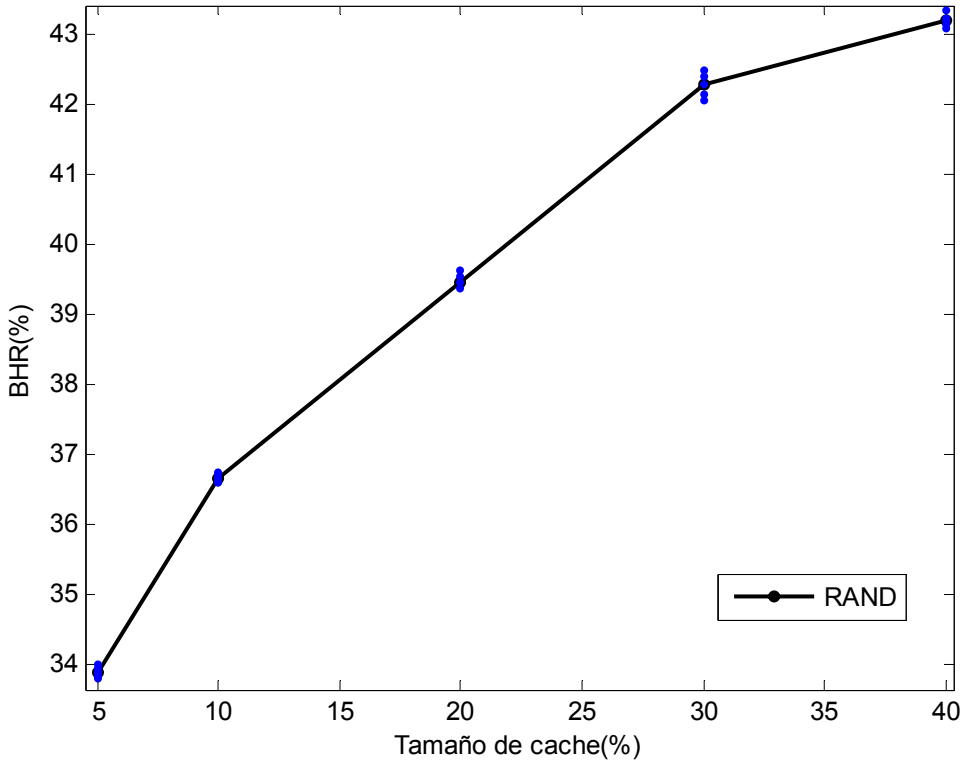
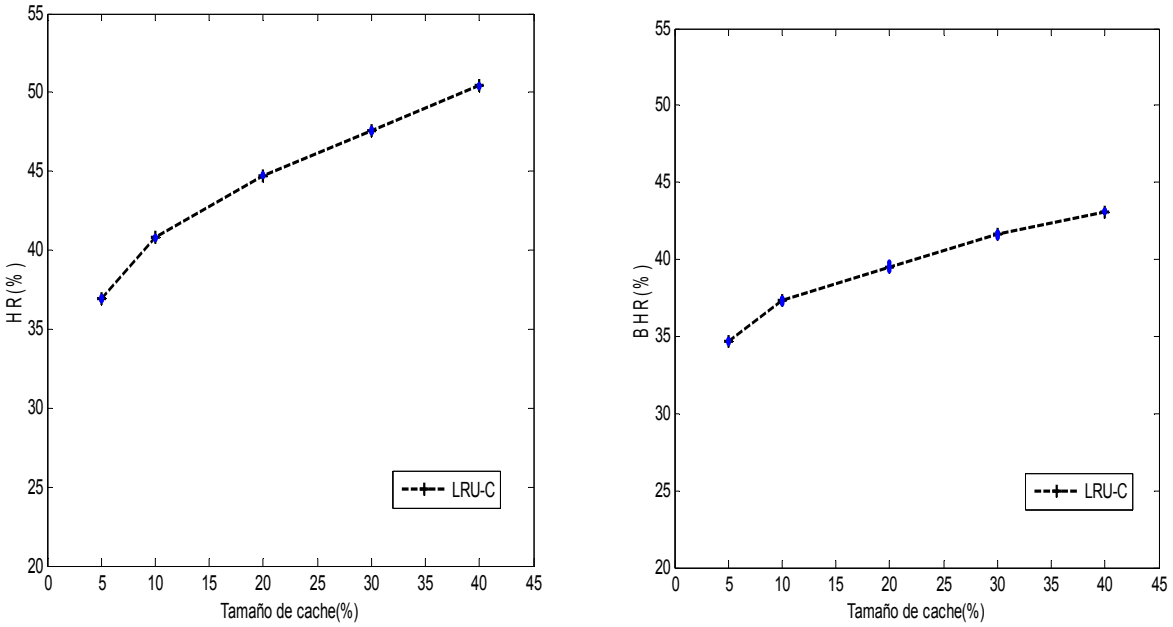


Figura 5.12 Ampliación del Byte Hit Ratio en *RAND*

Comparando con lo visto hasta ahora, se comprueba que los resultados obtenidos en cuanto al *BHR* son francamente positivos; aunque no tanto los referentes al *HR*.

5.11. LRU-C

Los resultados obtenidos al simular la variante aleatoria de la estrategia LRU, con una función coste constante e igual a la unidad, son:



a) Hit Ratio **b) Byte Hit Ratio**

Figura 5.13. Simulación de LRU-C

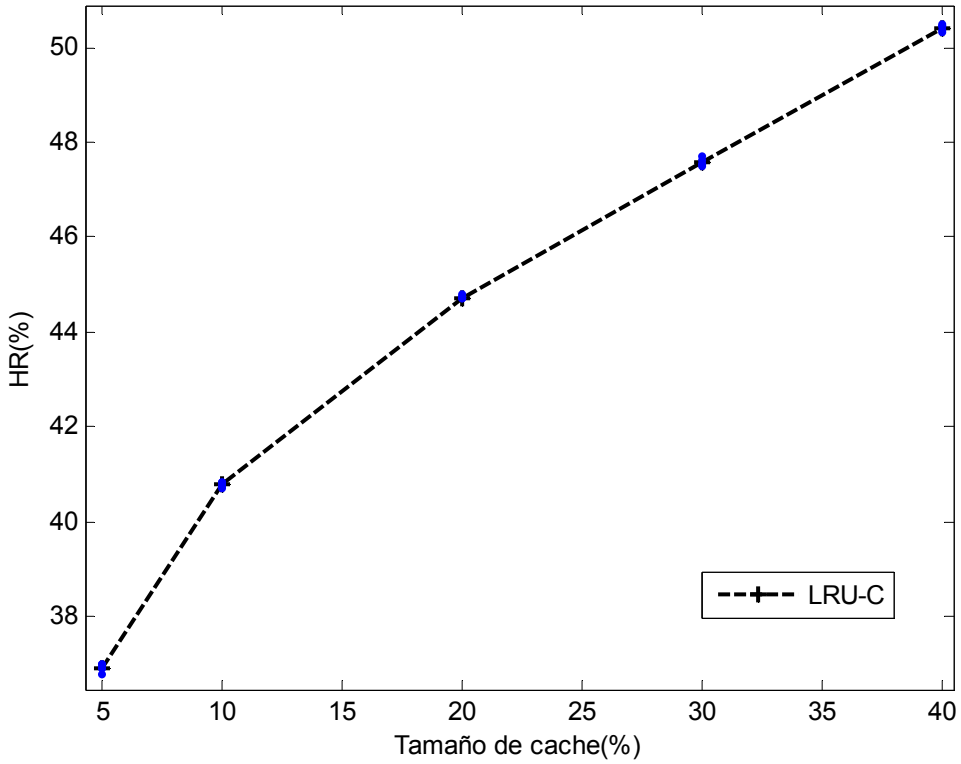


Figura 5.14 Ampliación del Hit Ratio en *LRU-C*

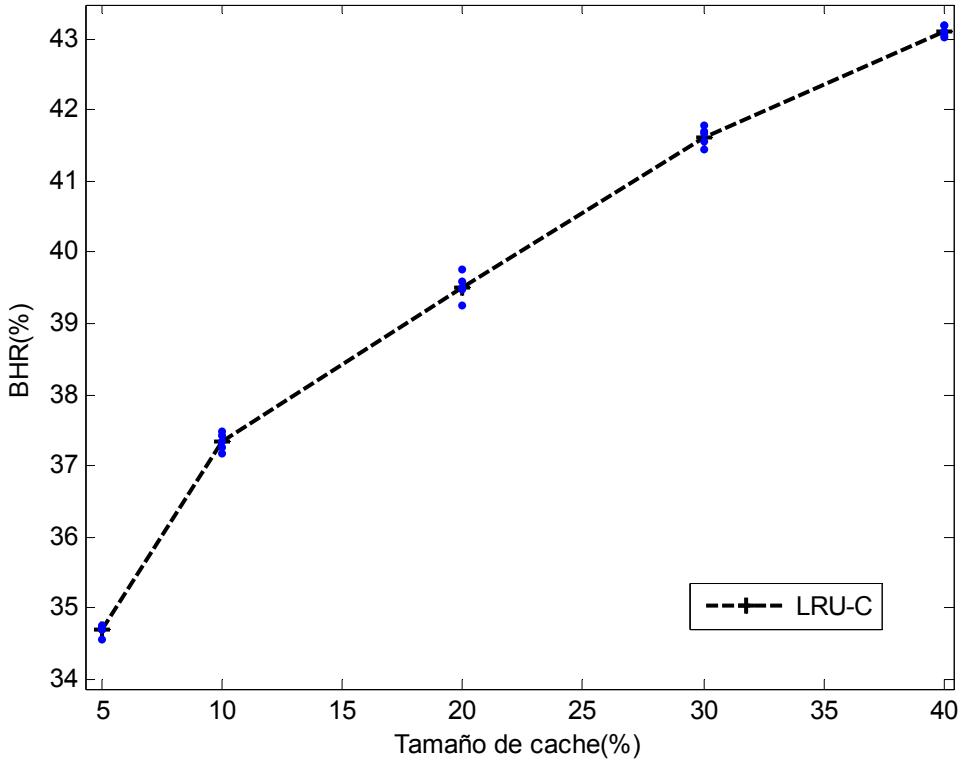
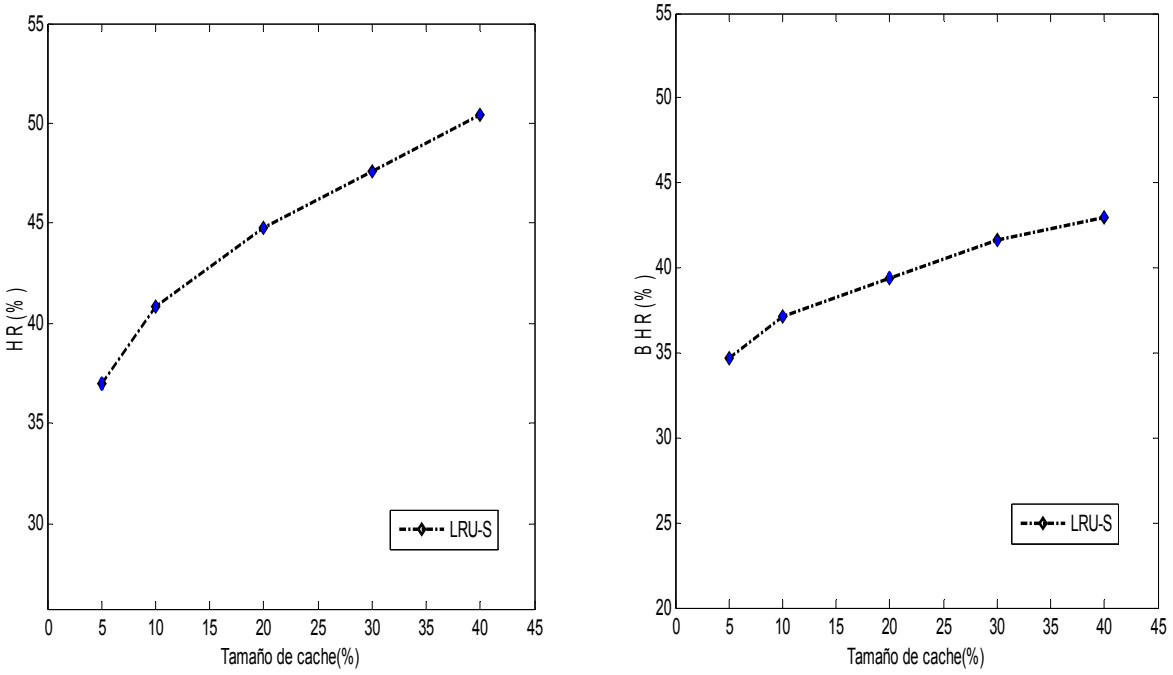


Figura 5.15 Ampliación del Byte Hit Ratio en *LRU-C*

A vista de las gráficas obtenidas, se ve que los resultados son realmente parecidos a los conseguidos con la política *LRU*, únicamente presentan un cierto desmejoramiento en cuanto al *BHR* logrado.

5.13. LRU-S

La otra variante aleatoria del *LRU* que se estudiará. En este caso, la decisión de si mover o no un documento almacenado dependerá de su tamaño:



a) Hit Ratio **b) Byte Hit Ratio**

Figura 5.16. Simulación de LRU-S

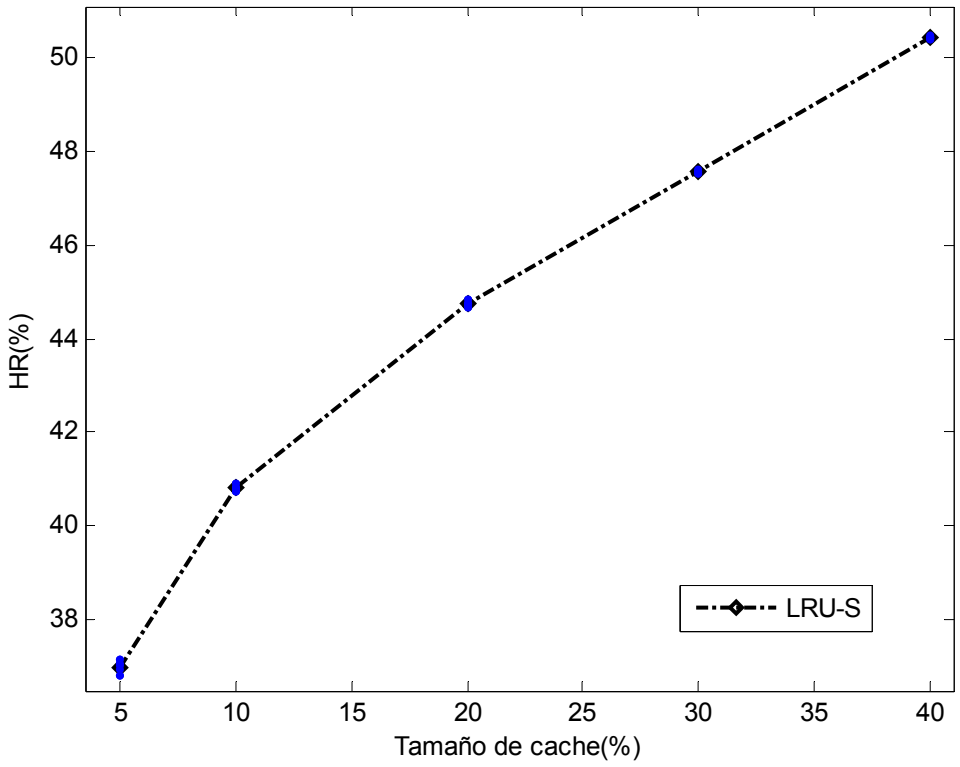


Figura 5.17 Ampliación del Hit Ratio en *LRU-S*

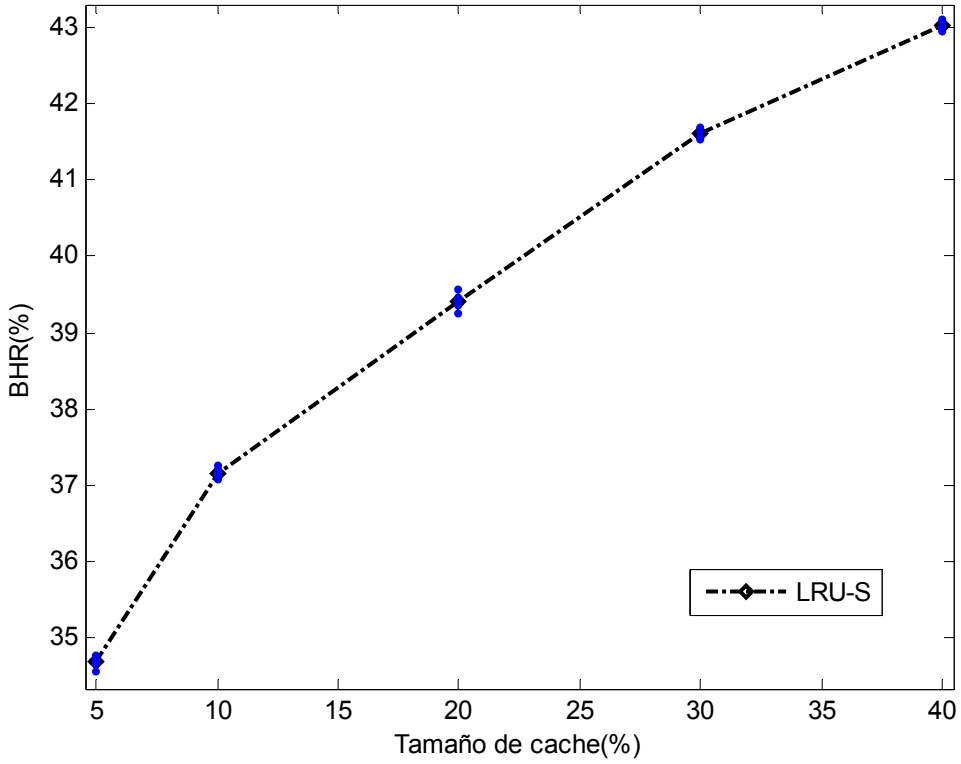
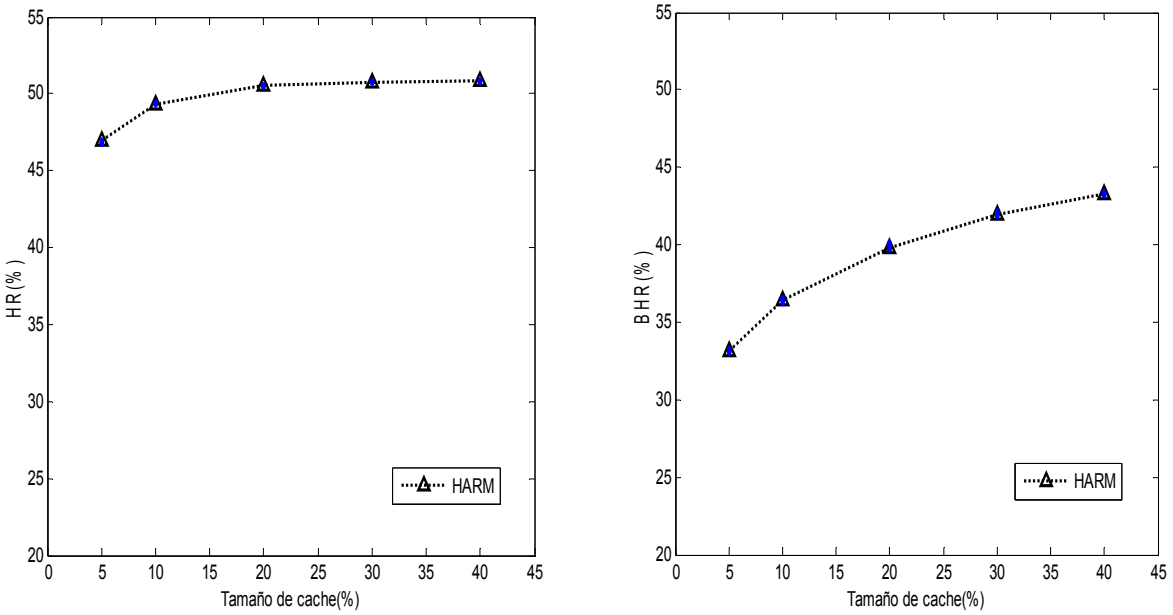


Figura 5.18. Ampliación del Byte Hit Ratio en *LRU-S*

A la vista de los resultados obtenidos, se comprueba que el comportamiento de la estrategia *LRU-S* es prácticamente idéntico al de *LRU-C* (sus resultados son tremendamente parejos), y muy similar al *LRU*.

5.14. HARM

Los resultados obtenidos para esta estrategia:



a) Hit Ratio **b) Byte Hit Ratio**

Figura 5.19. Simulación de HARM

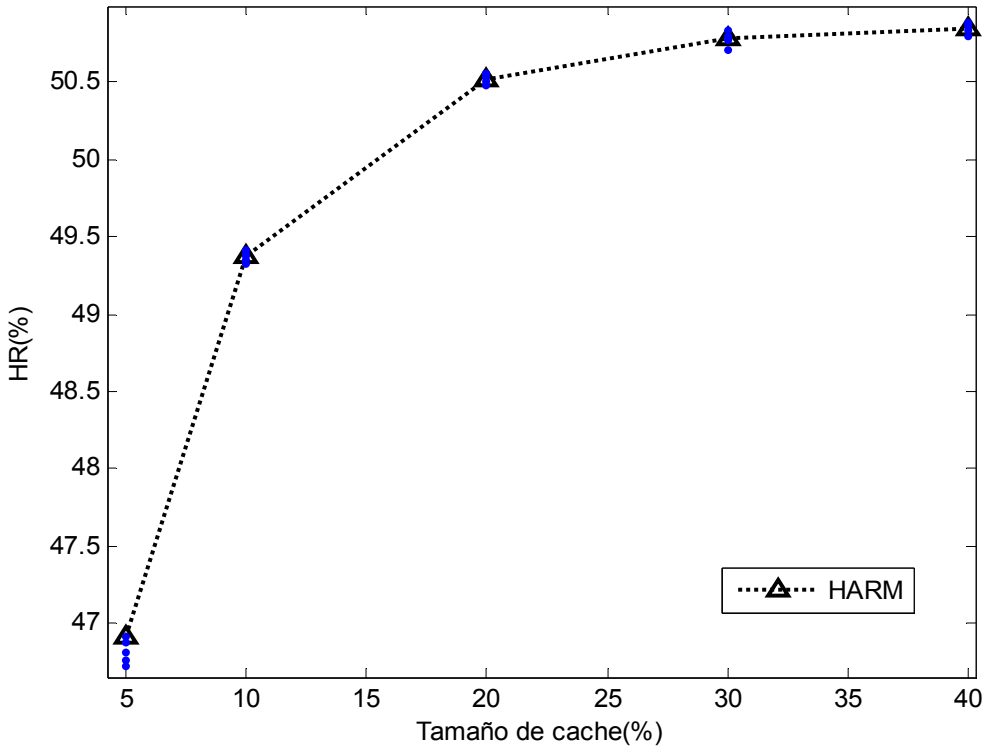


Figura 5.20. Ampliación del Hit Ratio en *HARM*

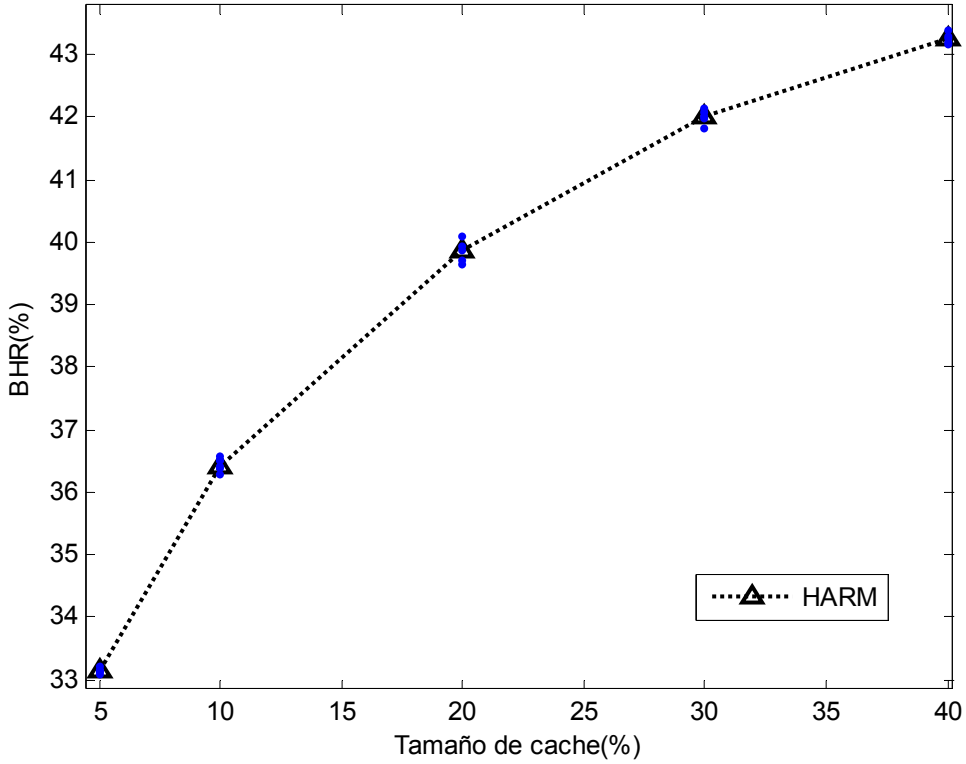
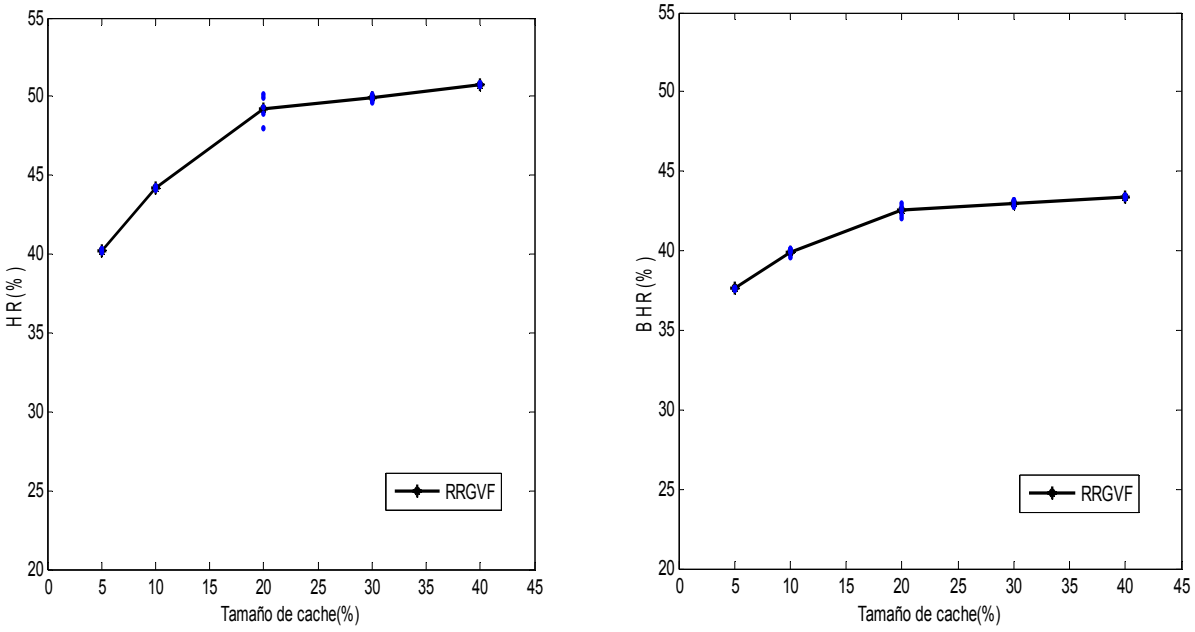


Figura 5.21 Ampliación del Byte Hit Ratio en *HARM*

Como se puede observar, el *BHR* obtenido se aproxima notablemente a los obtenidos en cada una de las políticas aleatorias estudiadas; por el contrario, el *HR*, parece ser más alto y homogéneo para los distintos tamaños de cache, su distribución recuerda a la obtenida en políticas *GDXX(1)*.

5.15. RRGVF

Ésta será la última política a evaluar, y es también la que presenta un comportamiento más complejo a la hora de seleccionar los elementos candidatos a ser eliminados. A continuación se detallará si este mecanismo de selección es influyente en cuanto a los *HR* y *BHR* obtenidos:



a) Hit Ratio **b) Byte Hit Ratio**

Figura 5.22. Simulación de RRGVF (N=30)

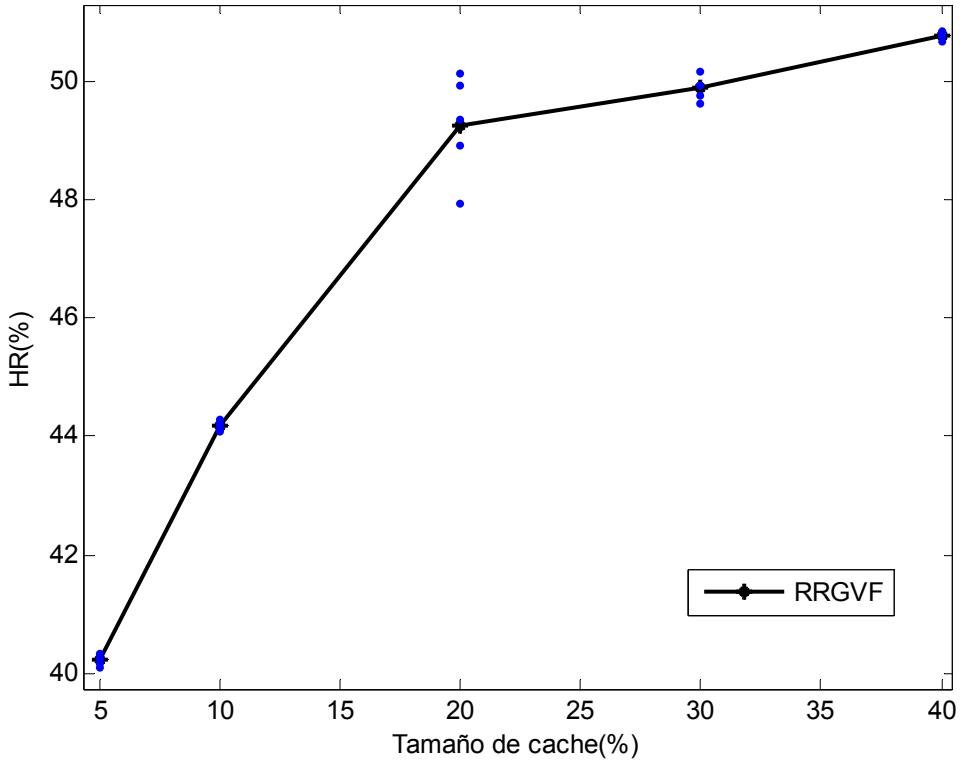


Figura 5.23 Ampliación del Hit Ratio en *RRGVF*

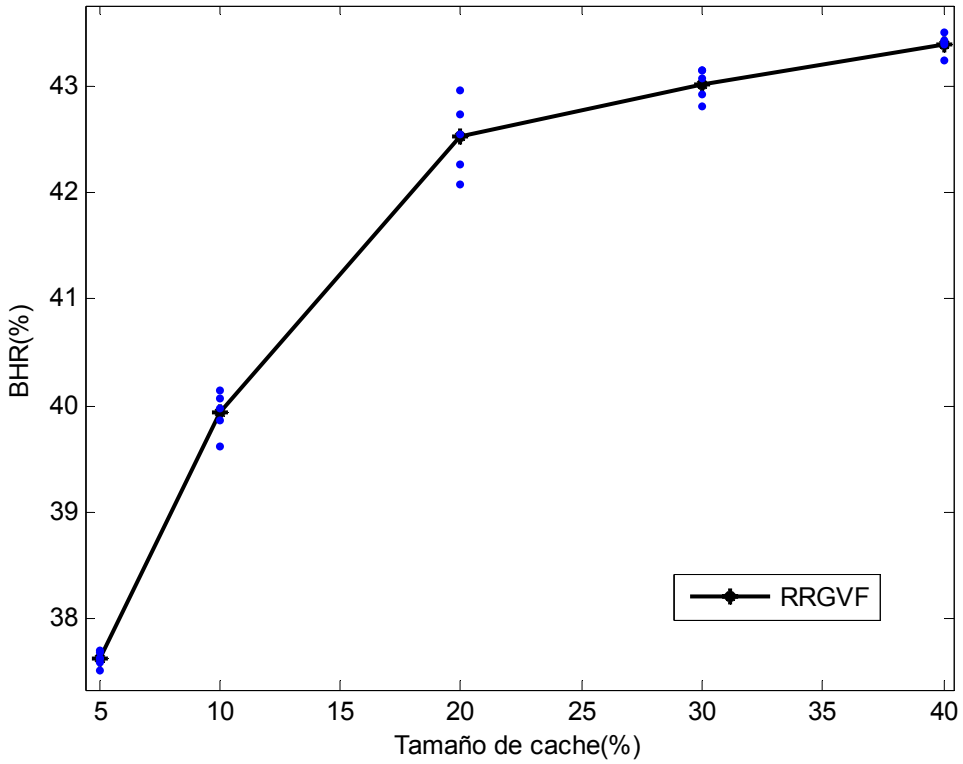


Figura 5.24 Ampliación del Byte Hit Ratio en *RRGVF*

Se comprueba que esta política consigue resultados similares a los alcanzados con el resto de políticas aleatorias, si bien, esta es la estrategia que una mayor variación de resultados ha alcanzado para las diversas simulaciones realizadas con un mismo tamaño de *cache*.

5.16. COMPARACIÓN DE POLÍTICAS DE REEMPLAZO

Para concluir este capítulo se realizará una breve comparación entre políticas, comparación de la que se extraerán una serie de conclusiones, objetivo último del presente proyecto.

Para realizar ésta, se han agrupado las políticas en tres grandes grupos:

- Políticas de Reemplazo Clásicas: *LRU, LFU, LFU-DA*
- Políticas de Reemplazo basadas en Función: *LRU, GDS (1), GDS (p), GDSF (1), GDSF (p),GD* (1), GD* (p)*
- Políticas de Reemplazo Aleatorias: *LRU, RAND, LRU-C, LRU-S, HARM, RRGVF*

5.16.1. *LRU, LFU, LFU-DA*

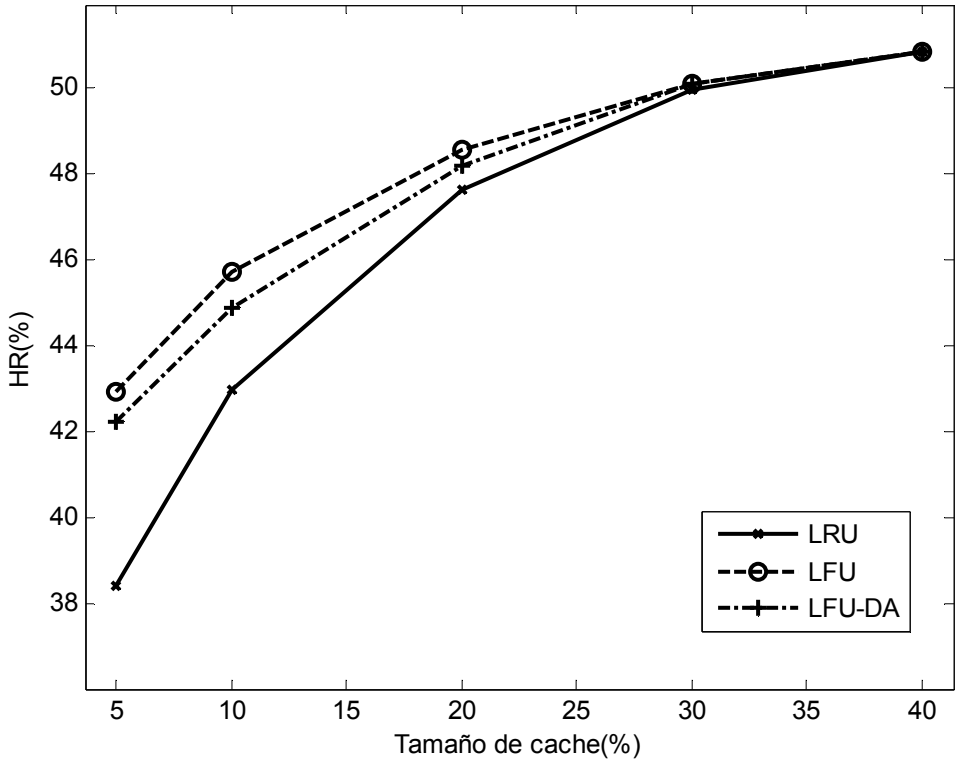


Figura 5.25 Comparación de *HR* entre políticas *LRU, LFU y LFU-DA*

Experimentalmente, se comprueba que la política que mejores estadísticas arrojas en cuanto al *HR,* es la *LFU*; aunque sus resultados son muy parejos a los logrados por *LFU-DA*. Eso puede explicarse en el hecho de que ésta última intenta eliminar documentos envejecidos, pero siempre cabe la posibilidad de que estos documentos vuelvan a ser referenciados, es decir, vuelvan a ponerse de 'moda', circunstancia en la que la estrategia *LFU-DA* se vería perjudicada, en relación a *LFU* original.

Un escalón por debajo estaría la política *LRU*, aunque sus resultados son también bastante parecidos, y en cualquier caso, tienden a igualarse a medida que el tamaño de la cache aumenta.

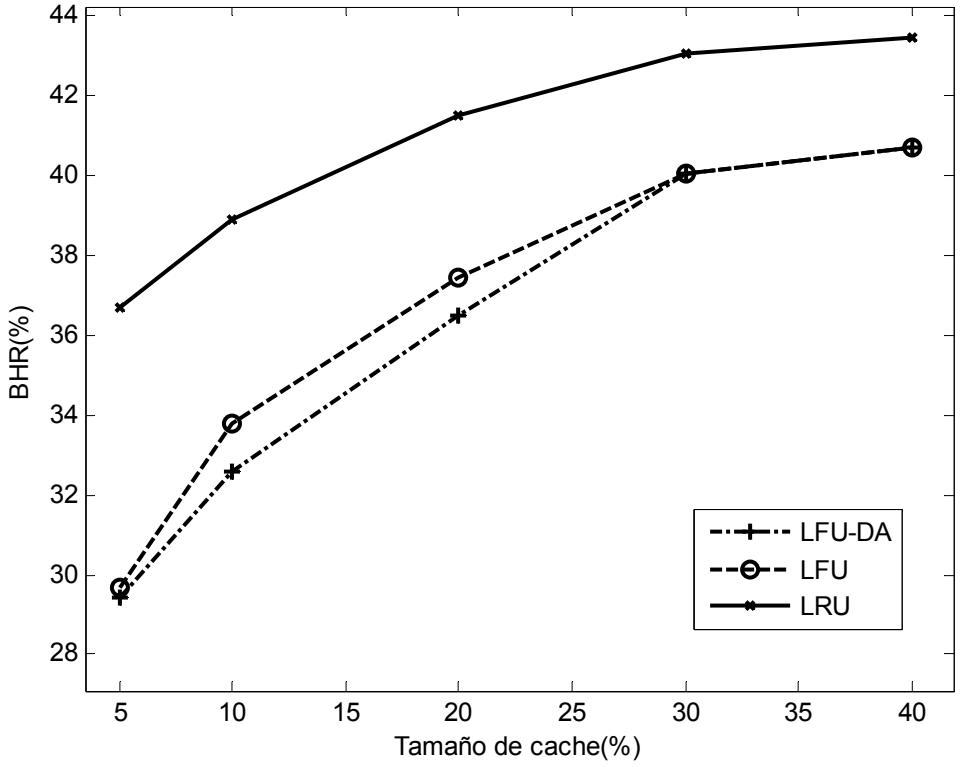


Figura 5.26. Comparación de *BHR* entre políticas *LRU, LFU y LFU-DA*

Se observa como se dan unas circunstancias diametralmente opuestas a las dadas en la *figura 5.39.*

Ahora, es la estrategia *LRU*, la que consigue tasas de bytes acertados mucho mayores, mientras que las políticas *LFU* y *LFU-DA* se mantienen en valores inferiores y muy parejos ambos.

De lo cual se deduce que estas políticas basadas en función son más convenientes en escenarios que primen el número de documentos acertados sobre el número de bytes totales acertados.

5.16.2. *LRU, GDS, GDSF, GD**

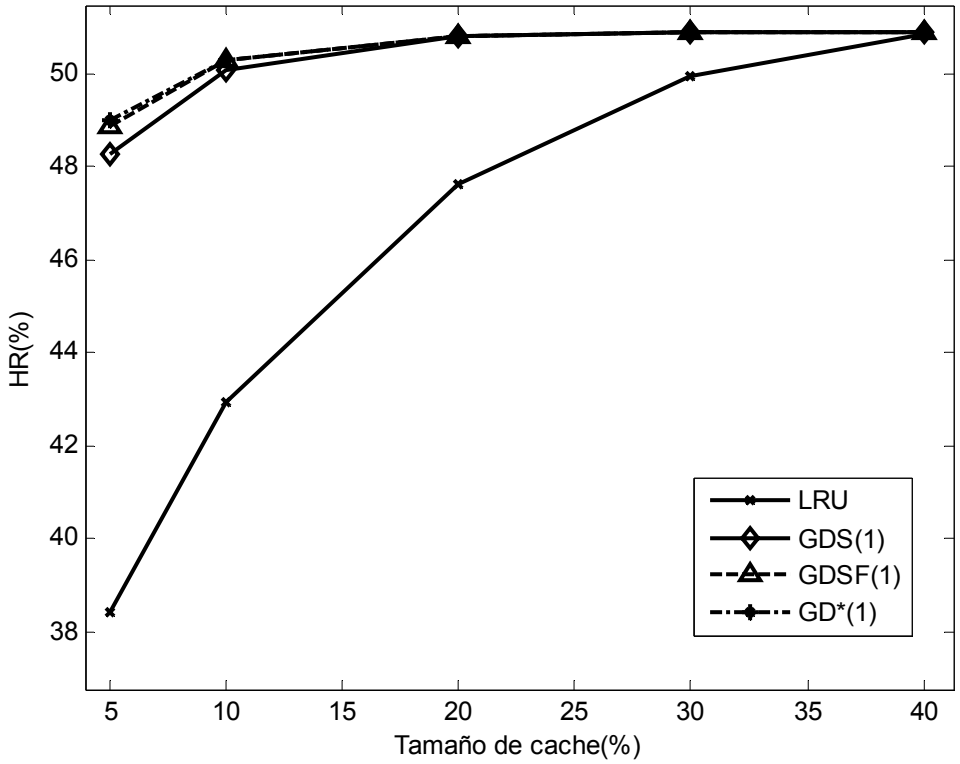


Figura 5.27 Comparación de *HR* entre políticas *LRU, GDS (1), GDSF (1) y GD* (1)*

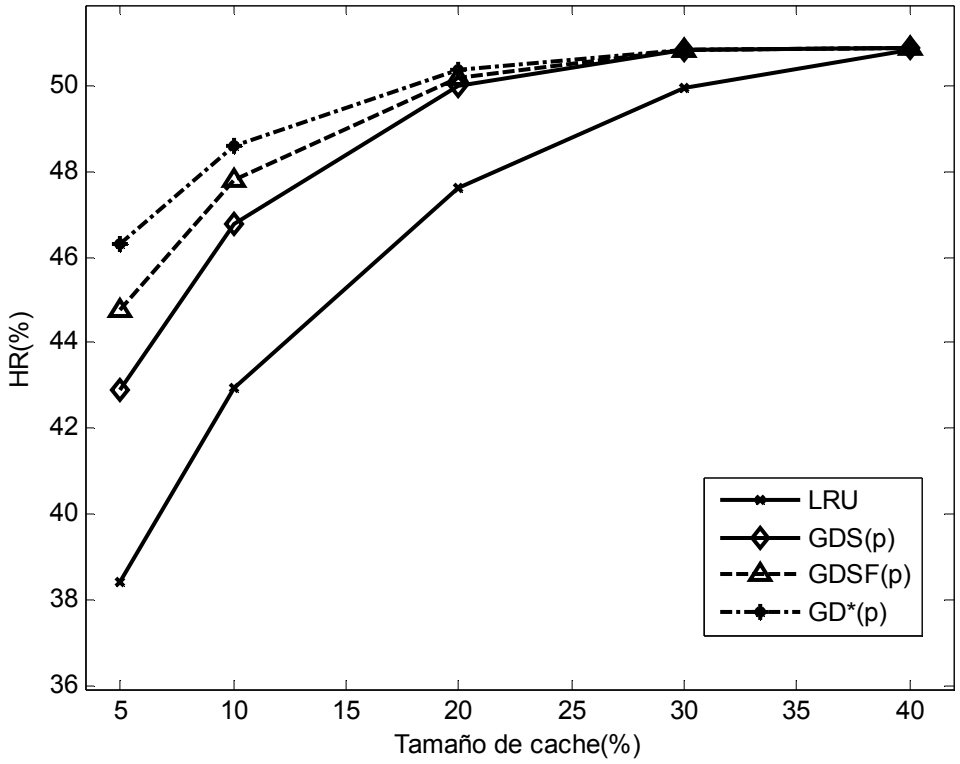


Figura 5.28 Comparación de *HR* entre políticas *LRU, GDS (p), GDSF (p) y GD* (p)*

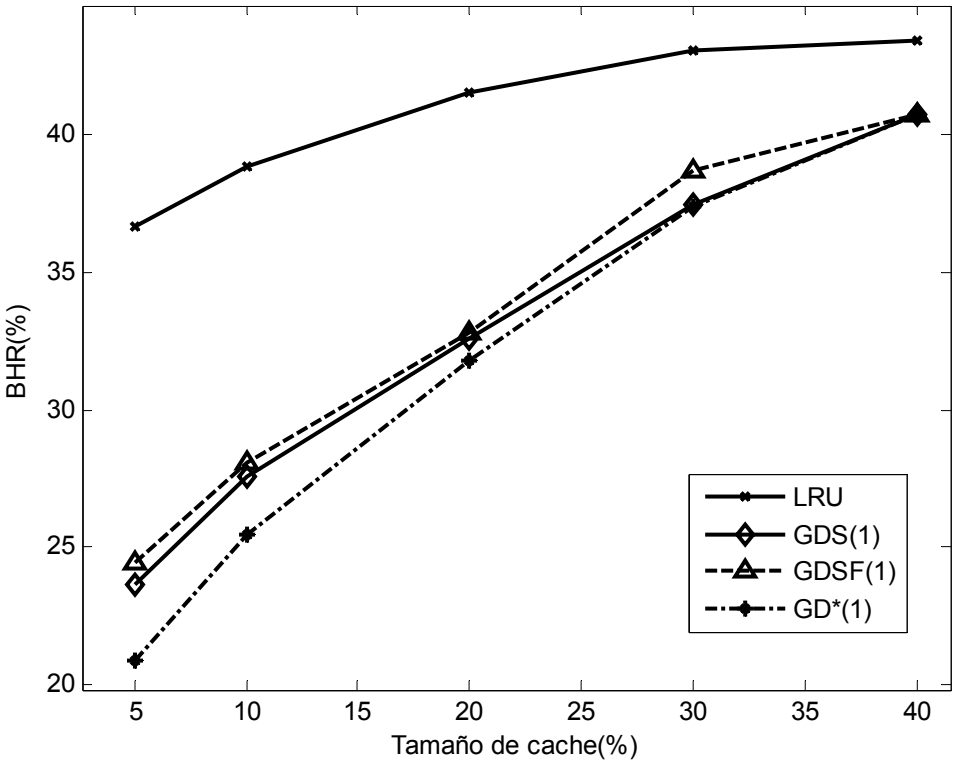


Figura 5.29. Comparación de *BHR* entre políticas *LRU, GDS (1), GDSF (1) y GD* (1)*

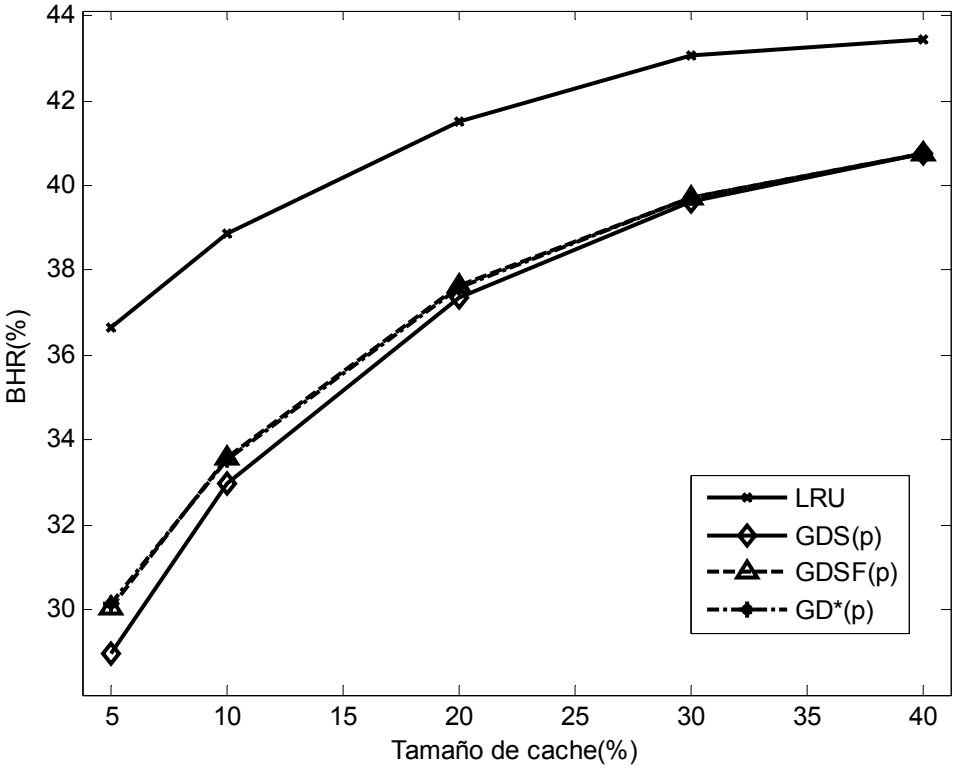


Figura 5.30. Comparación de *BHR* entre políticas *LRU, GDS (p), GDSF (p) y GD* (p)*

En este caso, se produce el efecto contrario al visto en la *figura 5.39,* en esta ocasión, como se avanzó teóricamente en el capitulo 2, son las políticas que emplean una función coste *packet* las que alcanzan un mayor *BHR*.

Aun así, será la estrategia *LRU* la que consigue una tasa de *BHR* más elevada, seguida de *GD** y *GDSF* (con resultados muy similares).

Como sucedió en la *figura 5.39*, las políticas que emplean una función coste, acercan la tasa de *BHR* conseguida a medida que aumenta el tamaño de la cache, prácticamente coincidiendo cuando ésta es del 40% del tamaño total de la muestra de entrada.

5.16.3. *LRU, RAND, HARM, LRU-C, LRU-S, RRGVF*

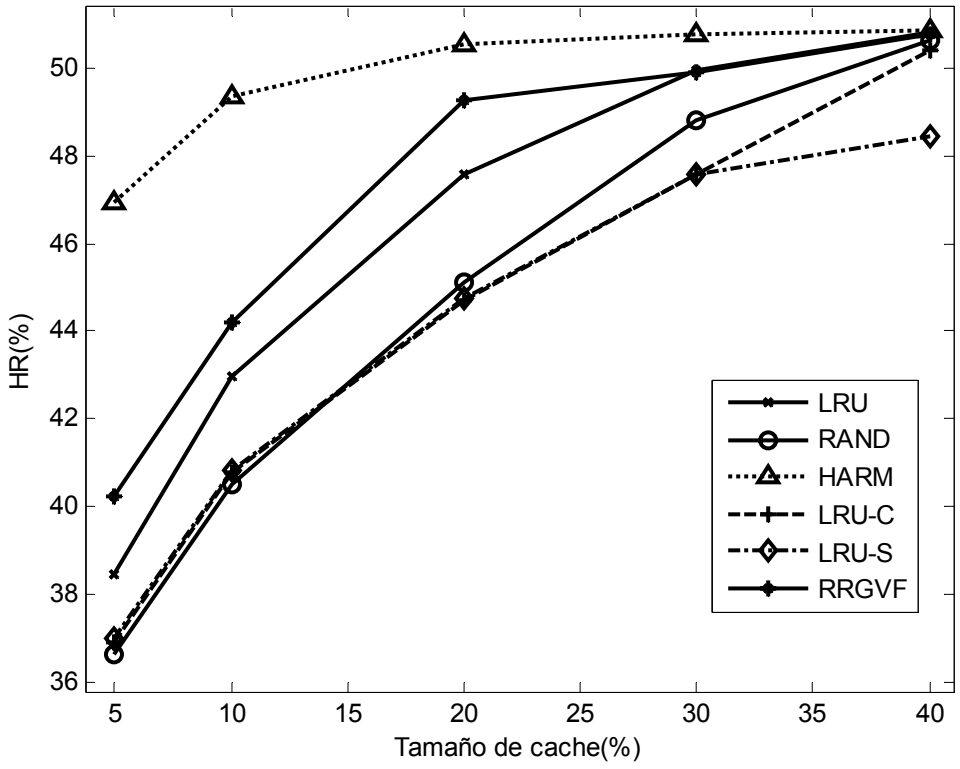


Figura 5.31 Comparación de *HR* entre políticas *LRU, RAND, HARM, LRU-C, LRU-S y RRGVF*

En este caso, se propone la comparación de los resultados obtenidos por las políticas de naturaleza aleatoria y la *LRU*. Como se puede observar, las políticas consiguen resultados muy diversos; aunque parece alcanzar valores muy similares cuando el tamaño de la cache es el mayor.

Si se revisa detenidamente la *figura 5.41*, se puede observar como es la política *HARM* la que consigue un mayor *HR*, lo cual se puede explicar en el hecho de que esta política elimina con mayor probabilidad los objetos de mayor tamaño, que no benefician a la consecución de un mayor *HR*. A continuación estarían la *RRGVF* (esta política no elimina elementos de manera estrictamente aleatoria, ya que elige el menos útil de entre un grupo aleatorio; por tanto en cierta medida favorece más el HR que el resto de políticas puramente aleatorias) y *LRU*, seguidas de *RAND*, *LRU-C* y *LRU-S.*

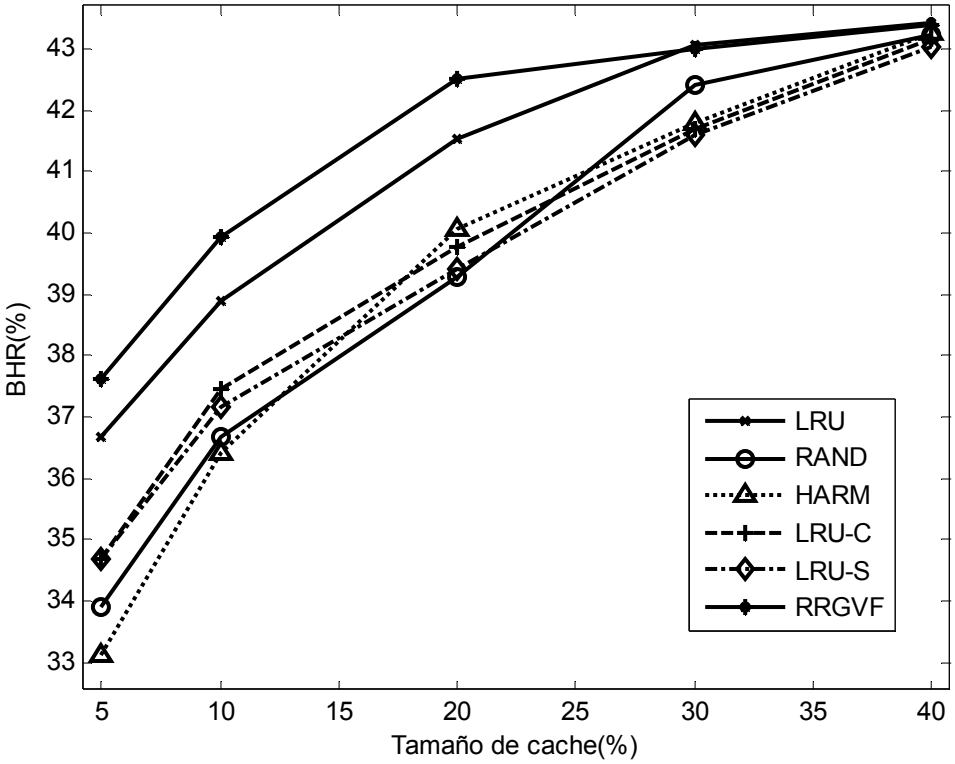


Figura 5.32. Comparación de B*HR* entre políticas *LRU, RAND, HARM, LRU-C, LRU-S y RRGVF*

Al tratar el *BHR*, se produce el efecto contrario, la política *HARM* al eliminar los objetos de mayor tamaño, será la política que consiga un menor *BHR*. En este caso, será la estrategia *RRGVF* la que alcance los mejores resultados, esta circunstancia se constata en el hecho, antes citado, de que esta política selecciona un grupo de objetos descartables de manera aleatoria, pero eliminará el menos útil de ellos, de manera que si el grupo de control es lo suficientemente amplio, la estrategia eliminará siempre objetos poco frecuentados, en otras palabras, eliminará elementos que no favorecen para nada al *BHR*. Tras la política *RRGVF*, aparecen la *LRU*, *LRU-C*, *LRU-S*, *RAND* y *HARM* respectivamente.

La política *RAND* obtiene siempre los peores resultados, y eso es debido a que selecciona los elementos a eliminar de manera absolutamente aleatoria, no sigue ningún tipo de criterio de selección que favorezca la obtención de unos mejores resultados.

Por último, se realizará la comparación de la política *LRU* y cada una de las políticas que hayan conseguido los mejores resultados de *HR* y *BHR* en los anteriores apartados.

5.16.4. *LRU* Y LAS MEJORES POLÍTICAS EN *HR y BHR*

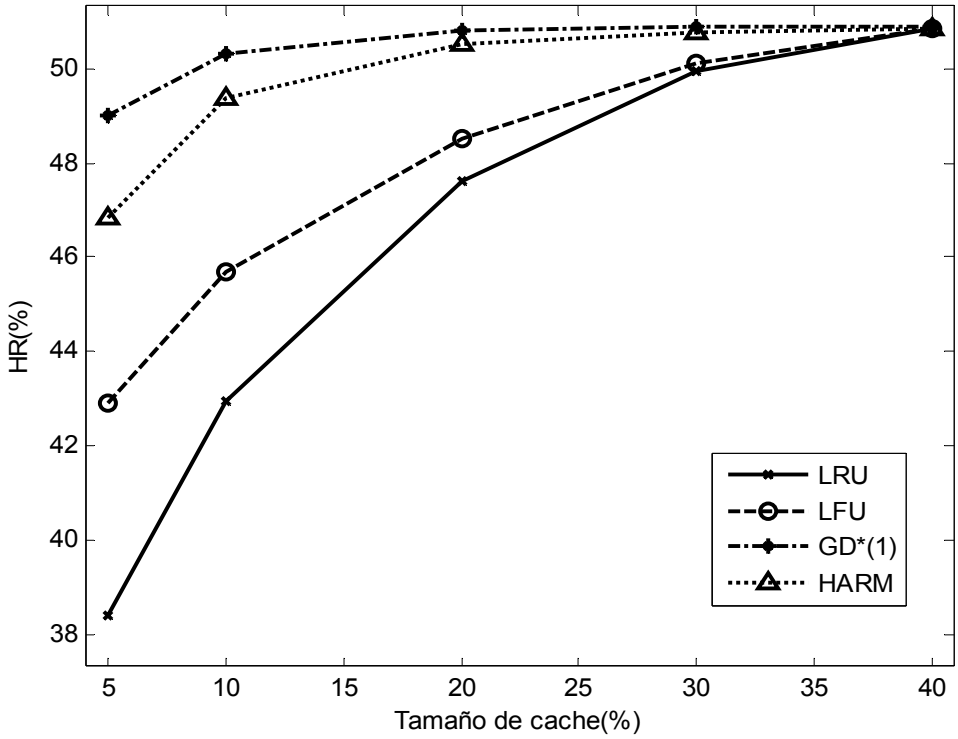


Figura 5.33. Comparación de *HR* entre políticas *LRU, LFU, HARM y GD*(1)*

Si se examina la *figura 5.47,* se concluye con la idea de que es la política *GD*(1)* (y por extensión la *GDSF (1)*), la que consigue una mejor tasa de *BHR*, como se anticipó, gracias al empleo de una función de coste unitaria. A continuación estaría la estrategia *HARM*, que alcanza tasas muy parejas, gracias al hecho de que elimina con mayor probabilidad los objetos más pesados.

Las políticas menos favorecidas, en este caso, serán la *LFU* y por último la *LRU*, que presenta valores de *HR* muy inferiores, sobre todo cuando la cache es menor.

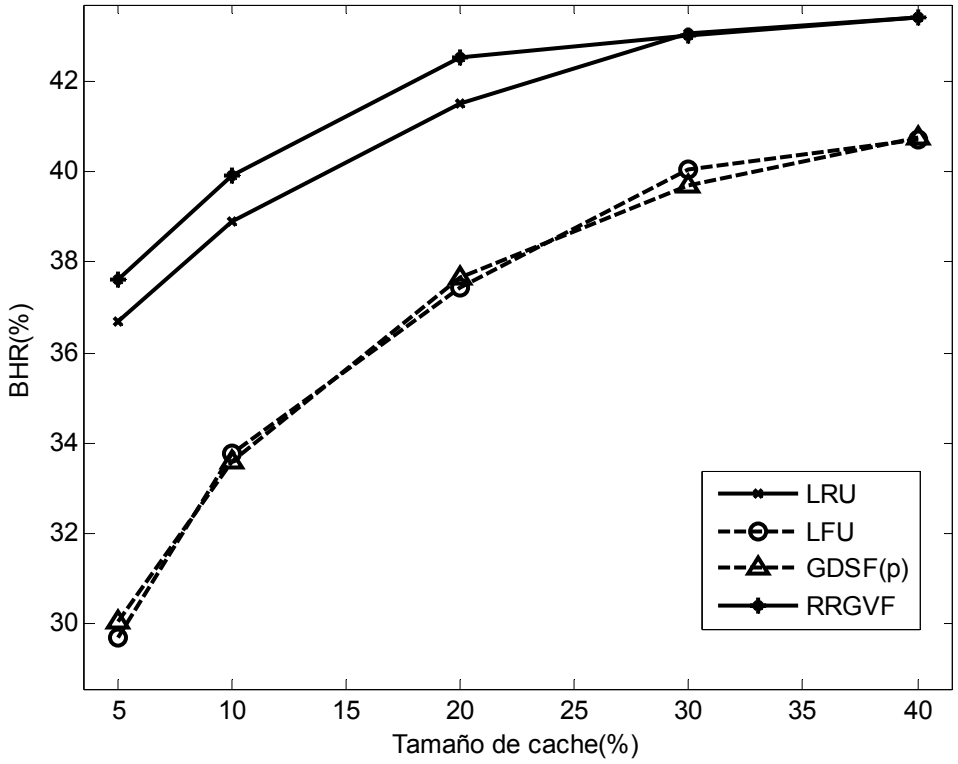


Figura 5.34. Comparación de *BHR* entre políticas *LRU, RRGVF, GDSF (p) y LFU*

En cuanto al *BHR*, será la estrategia *RRGVF* la que consiga tasas mayores, seguida muy de cerca de la *LRU*.

Por el contrario, *GDSF (p)* y *LFU*, se sitúan muy por debajo de las otras dos.

CAPÍTULO 6: Conclusiones y Líneas Futuras

Para terminar con la memoria, se expondrá este último capítulo, con el que se pretende dar un último repaso de las conclusiones a las que se han llegado tras profundos estudios de las políticas de reemplazo aquí descritas.

Con todo, este último capítulo se dividirá en dos nuevos apartados. En el primero se hará una somera descripción de cómo se podría continuar este estudio de sistemas de *cache Web,* qué otras políticas sería interesante proponer, qué otras estructuras de datos serían más convenientes implementar, etc.…

En el segundo apartado, se realizará un resumen general de todo lo estudiado, se hará mención de todo lo conseguido, así como de las conclusiones finales a las que se ha llegado tras la realización del proyecto.

6.1. CONCLUSIONES

¿Qué es una *cache Web*? ¿Para qué sirve? ¿Qué son y cómo funcionan las diversas políticas de reemplazo? Todas esas son dudas que se han intentado solventar en el desarrollo de la presente memoria.

Un *proxy cache* es un sistema que se sitúa entre un servidor *Web* y un cliente que realiza una petición de un documento *Web.* De esta manera la cache resuelve tales peticiones, mejorando la latencia vista desde el usuario, a la vez que evita la saturación del propio servidor. A su vez, la existencia de estos sistemas arroja una serie de desventajas, como el hecho de precisar un refresco continuo de los documentos almacenados.

Aún así, se ha demostrado que el uso de sistemas *cache* es ampliamente beneficioso. ¿Qué parámetros definen el comportamiento de estos sistemas? Los parámetros principales son la elección de una política de reemplazo y del tamaño de la memoria *cache* empleada. A lo largo de la memoria se ha realizado un completo estudio de estos algoritmos, sus ventajas e inconvenientes, y en definitiva, a qué escenario de tráfico *Web* se adecuan. Aunque, por otro lado, también se ha hecho mención al hecho

de que cada vez más, la elección de una política u otra es menos determinante. Esto es debido al constante crecimiento en tamaño de las memorias que se emplean para implementar estos sistemas, así el uso de una memoria de mayor tamaño resulta mucho más eficiente que resolver complejas incógnitas acerca de qué política emplear.

Las políticas de reemplazo estudiadas, se centran en tres grandes grupos, las políticas clásicas (basadas en frecuencia o cómo de recientemente se ha llamado a un objeto), *LRU, LFU y LFU-DA*; políticas basadas en función (tanto en su versión de coste unitario como con coste tipo *packet*), *GDS, GDSF, GD**; y por último las aleatorias, *RAND, HARM, LRU-C, LRU-S, y RRGVF*.

Con todo, la labor desarrollada hace hincapié en estudiar tales políticas, para eso se desarrolló un simulador que emulase su comportamiento, de tal forma que se pudieran extraer una serie de conclusiones acerca de los resultados obtenidos. Conclusiones que se expusieron en el *capítulo 5*, para ello se recrearon todas las políticas ante una misma entrada y con tamaños diversos de *cache*.

Las conclusiones que se extrajeron fueron muy diversas, por un lado se demostró que políticas basadas en la frecuencia consiguen buenos resultados en cuanto al *HR*, pero no con respecto al *BHR*, caso en el que se ven superadas por una simple estrategia *LRU*.

Por otro lado, se vio como políticas basadas en función, al emplear funciones de coste tipo *packet*, consiguen *BHR* mayores, mientras que por el contrario aquellas que emplean funciones de coste constantes e iguales a la unidad maximizan el *HR*.

En cuanto al último paquete de estrategias, las aleatorias, el mejor rendimiento se alcanza con la política *HARM*, en caso de precisar un *HR* alto, mientras que para alcanzar tasas de *BHR* mayores, sería conveniente optar por una política *RRGVF*.

En conclusión, cada grupo de estrategias ofrece ciertas características diferenciadoras, pero destaca el algoritmo *GD* (1)* (o *GDSF (1)*) en caso de precisar una estrategia que maximice el número de aciertos en cache. Por el contrario, en caso de precisar maximizar el número de bytes acertados, sería conveniente optar por una política aleatoria, del tipo *RRGVF*.

Por supuesto, el abanico de políticas existentes es mucho más amplio al aquí presentado, por tanto es sensato imaginar que existirán tantas soluciones como políticas, algunas de las cuales conseguirán resultados que superen a las hasta ahora presentadas.

Aún así, el estudio hasta aquí realizado es útil en tanto en cuanto nos ofrece una visión general de los algoritmos de reemplazo más comúnmente utilizados, su

naturaleza y comportamiento, de manera que una vez comprendidos estos, se pueda alcanzar a entender metodologías futuras con mayor sencillez.

6.2. LÍNEAS FUTURAS DE INVESTIGACIÓN

En caso de precisar continuar con el estudio hasta ahora desarrollado, sería necesario abrir nuevas líneas de investigación. Pero, ¿cuáles?

Si hacemos un repaso de lo hecho hasta ahora, es fácil ver, que la base que se ha desarrollado es la de comprobar el efecto que diferentes políticas tienen sobre la gestión de un sistema de *cache Web*.

Por tanto, es bastante intuitivo pensar, que una posible línea de continuación sería la evidente, desarrollar nuevas políticas de reemplazo. Estudiando sus características y resultados se podrían deducir nuevas conclusiones, que consiguiesen la mejora de un sistema de *cache.*

Por otro lado, al desarrollar el simulador se han hecho numerosas alusiones a la necesidad de que éste estuviera muy optimizado, para conseguir simulaciones rápidas ante grandes muestras de entrada. Por tanto, sería interesante el estudio de nuevas estructuras de datos y métodos que favorezcan aún más esta optimización

Por último, sería interesante el estudio de nuevos métodos de gestión de cache, es decir, hasta ahora se ha considerado que la llegada de un documento que no cupiese forzaba la extracción de algún elemento de la *cache*, pero, ¿y si eso no fuera así?

6.2.1. NUEVAS POLÍTICAS DE REEMPLAZO

Hasta ahora, se ha comprobado la gran importancia (aunque cada vez menor), de las políticas de reemplazo.

En el capítulo 2 se hizo una somera descripción de las políticas implementadas y los distintos grupos en los que se encuadraban. Políticas basadas en su uso reciente, en la frecuencia, en funciones, en su naturaleza aleatoria… todas ellas no eran más que una breve muestra del grueso de políticas que pertenecían a esos mismos grupos, y se podrían estudiar igualmente.

MIX, HYPER-G, SLRU, HYBRID, LRV, M-Metric, LR [1] son algunos de los ejemplos de políticas pertenecientes a esos grupos, que pudieran haberse estudiado, y que quedan propuestos para futuros estudios

Pero, el estudio podría extenderse aún más si en lugar de implementar políticas ya conocidas, se abrieran nuevos campos de investigación, posibles nuevas maneras de realizar el reemplazo:

- **Reemplazo adaptativo.** Hasta ahora se ha demostrado que no existe la estrategia de reemplazo 'definitiva'. Dependiendo de la carga de trabajo, diferentes estrategias pueden alcanzar los mejores resultados. Ya que la carga de tráfico en la *Web* es variable, sería interesante hacer que el sistema pudiera emplear diferentes estrategias de reemplazo, de manera que se adaptase a este cambio. De esta manera, políticas basadas en función podrían hacerse adaptativas modificando algunas variables de control.
- **Reemplazo coordinado.** Si se parte de la premisa de que las decisiones de reemplazo afectan al estado de un sistema de *cache Web*. La coordinación de decisiones de reemplazo en diferentes sistemas podría conseguir un rendimiento superior del sistema global.
- **Combinación de reemplazo y coherencia de *cache.*** Tradicionalmente, el reemplazo en *cache* y la coherencia de *cache* (es decir, la verificación de que el documento almacenado es el correcto, y no ha sido actualizado) son tratados como factores separados. Sin embargo existe una fuerte correlación entre ambos términos.
- ***Cache* de Reemplazo Multimedia.** En primer lugar, los documentos multimedia (especialmente videos) son de gran tamaño, ocupan gran parte de la *cache*, además de que colapsan el tráfico hasta la misma, por tanto, el empleo de técnicas de almacenamiento parcial (tan sólo se almacenan las partes más importantes) son tremendamente convenientes en estas circunstancias. Por otro lado, las *caches* multimedia emplean técnicas adaptativas para optimizar las estrategias de reemplazo.
- ***Cache* de reemplazo Diferencial.** Las *caches* vistas hasta ahora, no soportan el concepto de calidad de servicio (*QoS*). Estos sistemas son ya conocidos, y se implementan introduciendo una diferenciación en cuanto al orden de almacenamiento.

6.2.2. OPTIMIZACIÓN DE ALGORITMOS

El estudio de nuevas maneras de optimizar algoritmos es francamente importante. Nuevas estructuras de datos que permitan un acceso más rápido a cada uno de los documentos almacenados, métodos de búsqueda más optimizados, desarrollo de

nuevos algoritmos a la hora de organizar los documentos…, cualquier nuevo desarrollo que permita la simulación de las políticas más rápidamente, consiguiendo los mismos resultados.

6.2.3. GESTIÓN DE *CACHE*

Normalmente, la gestión de *cache* está asociada con el reemplazo. Así, de llegar un nuevo documento que no cupiese, se liberaría el espacio necesario del sistema.

Una nueva filosofía de gestionar el sistema, es la *cache* estática. En esta, el contenido del sistema es actualizado periódicamente. Así la popularidad de un objeto es determinada a partir de información de periodos anteriores, es decir, sus campos descriptivos no se actualizan constantemente, sino en momentos muy específicos.

Si bien esta es una política fácil de implementar, será adecuada tan sólo en sistemas en los que esté perfectamente definido el número de documentos. Su filosofía es la siguiente, cada vez que se referencie un documento almacenado se actualizará el valor de su límite de tiempo (como se ha dicho anteriormente, tan sólo se actualizará en determinados periodos de tiempo). Periódicamente la *cache* revisa los documentos y estudia si alguno de ellos ha excedido su límite de tiempo, y en tal caso los elimina del sistema.

Ésta estrategia es fácil de implementar y permite análisis muy intuitivos. El problema radica en que el espacio de memoria usado crece fuertemente al incrementarse el valor del periodo de tiempo límite.

Una posible solución es almacenar tan sólo documentos que hayan sido referenciados un número determinado de veces. De esta manera filtraremos elementos que no contribuyen excesivamente en el *HR*

El estudio de nuevas manera de gestionar la *cache*, tales como la que se acaba de describir, abriría nuevas puertas a la investigación acerca de los sistemas de cache Web y su posible evolución.

Bibliografía

[1] Stefan Podlipnig, Laszlo Böszörmenyi, “A Survey of Web Cache Replacement Strategies”. University Klagesnfurt.

[2] Ludmila Cherkasova, “Improving WWW Proxies Perfomance with Greedy-Dual-Size-Frequency Caching Policy”. Computer Systems Laboratory, November 1998.

[3] Martin F. Arlitt, Carey L. Williamson, “Trace-Driven Simulation of Document Cachin Strategies for Internet Web Servers”. Department of Computer Science, University of Saskatchewan (Canada), September 11, 1996.

[4] Shudong Jin, Azer Bestavros, “Temporal Locality in Web Request Streams. Sources, Characteristics, and Caching Implications”. Computer Science Department, Boston University, October 1999.

[5] Pei Cao, Sandy Irani, “Cost-Aware WWW Proxy Caching Algorithms”. USENIX Symposium on Internet Technologies and Systems, Monterey (California), December 1997.

[6] Mark Notthingham, “Caching Tutorial for Web Authors and Webmasters”. 1998

[7] Konstantinos Psounis, Balaji Prabhakar, “Arandomized Web-Cache Replacements Scheme”. Department of Electrical Engineering and Computer Science, Stanford University.

[8] M. Arlitt, R. Friedrich, T. Jin, “Worklload Characterization of a Web Proxy in a Cable Modem Environment”. Technical Report HPL-1999-48, Hewlet-Packard Laboratories, 1999.

[9] D. Starobinski, D. Tse. “Probabilistic methods forWeb caching”. Perform. Eval. 46, 2–3, 125–137, 2001.

[10] S. Hosseini-Khayat. “Investigation of generalized caching”, Ph.D. dissertation. Washington, University, St. Louis, MO. 1997