Implementation of LRU policy in cache memories

In computer science cache memories are additional memories that are used for temporary storage of data. Whenever a program requests data from a source (e.g the hard disk of a computer) the program stores a duplicate of the information in the cache memory. Thus, the program will be able to retrieve those data more efficiently in the future, since they can be obtained without searching the whole disk once again.

Cache hit : Whenever we look up for a specific element in the cache and it is found then we have a „cache hit".

Cache miss : Whenever we look up for a specific element in the cache and the element is **not** found then a „cache miss" occurs. In this case the data have to be retrieved from the hard disk.

However, cache memories are expensive and their size is limited. Hence, whenever a cache miss occurs, the CPU needs to eject some other entry in order to make room for the previously uncached data As a result, it is crucial, to design a **replacement policy**, according to which the least important data will be removed from the cache. One of the most efficient replacement policies is the „Least Recently Used" (**LRU**). According to the LRU policy, the removed data will be the ones that where used the furthest in the past. For this reason, there needs to be a timestamp for every entry in the cache. The timestamp of an entry is updated every time there is a request for this specific entry.

To sum up, whenever some data are requested, the computer looks for this data in the cache memory. If the data are found in the cache (cache hit) then we have to update the corresponding timestamp. If the data are not found in the cache (cache miss) then the least recently used entry is removed from the cache, the requested data are retrieved from the disk and then stored in the cache, after updating its timestamp.

The problem

The goal in this project is to implement the LRU policy in the most efficient way. Thus, we have to design a cache that retrieves, stores and removes the oldest element as fast as possible. In our approach, we are able to achieve all these three operations in constant time. For this reason we design a data structure using a **hashtable with separate chaining** and a **linked list.**

Modelling the Entries

In order to model properly the entries, we use an abstract data type, where every entry is associated with a **key** that is used for their retrieval and its corresponding **data**. As a result, a class named Data is constructed containing two variables, the key and the data. In this project ,we use java generics such that those two variables can be set as any type (String, Integer or any other type).

Experiments

For our experiments, all the data are contained in a specific file that basically represents the hard disk. For every entry, the type of key and the data are considered to be String.

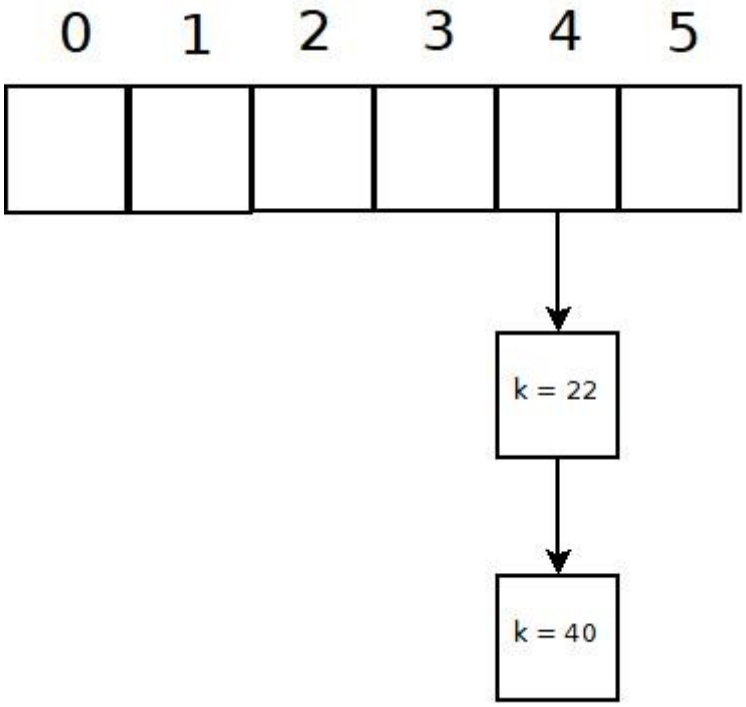
Data Structure

Our data structure needs to insert, remove and look up for elements in constant time (O(1) time). In addition, since we need to implement the LRU policy, we need to keep track of the **least recently used** (LRU) element also in constant time. Thus, whenever a **cache miss** occurs, we need to remove the LRU element from the cache, and store in the cache the element that was retrieved from the hard disk. In order to achieve all these goals we use the following data structure:

In order to insert, remove and look up elements in constant time we use **hashtable with sepatate chaining**. Therefore, for every key **k** we use the java hashCode function in order to obtain a corresponding integer value and then the actual position in the hashTable is obtained by the formula :

```
int i = key.hashCode() % hashTable.length;
```

Since two different keys might be associated with the same position in the hashTable, we have to keep a list of all the items that correspond to this position.



In this example the elements with keys 22 and 40 are linked with the same position in the hashtable since 22%6 = 40% 6 = 4. Hence, when using separate chaining, every position of the hashtable represents a list of items linked with this position.

On the other hand, in order to keep track of the LRU element of the cache we need to keep a linked list. Whenever we pull a request of a specific element, then this element is placed in the head of the linked list. As a result, **the LRU element of the list is always at the end of the list. Thus, there is no need of a specific timestamp in our implementation.**

The aforementioned data structured is implemented in the classes :

- ListNode
- List
- CacheImpl.

The class ListNode represents a node in our data structure and contains three pointers:

- ListNode<T> nextNode: The pointer nextNode is part of the linked list and points to the next node in the list.
- ListNode<T> prevNode: The pointer prevNode is part of the linked list and points to the previous node in the list.
- ListNode<T> nextHashNode: The pointer nextHashNode of a node n, points to the next item in the hashTable that shares the same position with n. Thus, in the previous example, assuming that n22 and n40 are the nodes with keys 22 and 40 equivalently, then n22.nextHashNode = n40.

The class List implements a linked list and contains methods for removal and insertion at front and at the end of the list equivalently. **Also it contains an additional method to place a specific element at the head of the list.**

The class CacheImpl is the basic class the cache functions are implemented. It contains the hashTable that is used to recover elements efficiently in constant time. In fact, **every position of the hashTable represents a pointer to an instance of the class ListNode**.

Chaining : We form a chain (list) by adding new elements at the head of this list. Concretely, if a new element n1 occurs that shares the same position with another existing element n2 in the hashTable, we place n1 at the front by modifying the hashTable pointer to point at n2 and the pointer nextHashNode of n1 points to n2. Thus, the list lying under the specific slot of the hashTable remains connected and the new element is placed at front. We can deduce then that in the previous example, node n22 was added later than n40.

Basic Algorithm

1. Initially the cache is empty.
2. Take the next request using the class WorkloadReader.
3. Look up the item with the specific key in the cache memory.
4. If the item is not found look up the item in the disk (class DataSource).
    - Store the item in the cache. (Place the item in the hashtable and place it at the front of the linked list)
5. If the item is found, just place the item first in the linked list.

The rest of the files are organised as follows:

TestCacheSpeed.java

This class contains the main method. It contains three variables determining:

- The file containing all the data, representing thus, the hard disk.
  static String *dataFile*
- The file containing the requests of specific data.
  static String *requestsFile*
- The total size of the cache memory i.e. the number of elements that the memory can store.
  static int *cacheSize*

Data.java

Every instance of this class is associated with a single element of our data. Concretely, it contains two variables representing the key of every element (**private** K key) and the actual data of every element (**private** V value) equivalently. Please take into consideration that we use generics through the entire project such that those two variables can be set as any type (String, Integer or anyt other class). In our implementation both key and value are set as String.

Cache.java

This file contains an interface that represents an instance of a cache memory. A class implementing this interface should implement the following methods:

```
public V lookUp(K key);
public V store(K key, V value);
public double getHitRatio();
public long getHits();
public long getMisses();
public long getNumberOfLookUps();
```

CacheImpl.java

This class implements the interface Cache and represents a cache memory. Apart from the methods needed to be implemented, we also include the following variables:

private long misses : this variable represents the number of cache misses, i.e. the number of times when cache memory did not contain a specific element and we had to search it in the hard disk.

private long hits : this variable represents the number of **cache hits**, i.e. the number of times when the cache memory contained a specific element.

List<Data<K, V>> list; : this is the least we use to keep track of the LRU element, which is always at the end of the queue

private ListNode<Data<K, V>>[] hashTable; : this is the hashtable, whose every position basically points to a chain of elements (all of which are associated with a specific position in the table)

private int size : variable containing the current number of elements in the cache

public V lookUp(K key) : This method looks for a specific element in the cache. It calculates the position in the hashtable for the specific key and then parses one by one the elements of the chain linked with this position. If the element is found, the requested element is placed at the front of the queue. (moveAtFront method in the class List.java)

public V store(K key, V value) : Firstly this method removes the oldest element from the list. Then it looks for the removed element in the hashtable so that it is removed from there too. And then adds the new element at the head of the queue and it is inserted in the hashtable.

DataSource.java

A DataSource instance represents the hard disk. Whenever an item is not found in the cache, then by using an instance of this class we can look up for the item in the “disk” by using the method readItem(String key).

WorkloadReader.java

This class is used in order to parse all the requests from the corresponding file.