Set Associative Cache

Requirements:

Design and implement an N-Way, Set-Associative cache, with the following features:
1. The cache itself is entirely in memory (i.e. it does not communicate with a backing store)
2. The client interface should be type-safe for keys and values and allow for both the keys and values to be of an arbitrary type (e.g., strings, integers, classes, etc.). For a given instance of a cache all keys must be the same type and all values must be the same type.
3. Design the interface as a library to be distributed to clients. Assume that the client doesn't have source code to your library.
4. Provide LRU and MRU replacement algorithms.
5. Provide a way for any alternative replacement algorithm to be implemented by the client and used by the cache.

Technologies Used:

Java 8, Eclipse Neon

How to use:

1) You can also add the SAC.jar file as an external library to your own project by right-clicking your Project Folder > Build Path > Add External Archives > SAC.jar > Finish. Be sure to import the sac.* and policy.* packages. The JAR file is located in the /jar folder.

2) Alternatively, you can import the entire source project into Eclipse through these steps:

a) Open Eclipse > File > Import > General > Projects from Folder or Archive > Next > Archive > Import Zip file of project > Finish

b) You may get errors related to JUnit. You can fix this by right-clicking the Project Folder > Build Path > Add Libraries > JUnit > Next > JUnit 4 > Finish.

3) If you wish to build and export the project once the project has been imported, right-click on the Project > Export > Java > JAR file > Finish.

To use the cache, the user must initialize a class that implements Policy.
This Policy is then used to initialize a SetAssociativeCache, along with the blocks per set and number of sets. The type used as the key for the Policy must match the key used by the cache.

Example:

```
//An MRU policy instance is created.
Policy<Integer> policy1 = new PolicyMRU<>();

//A cache is created with 3 sets of size 2 each.
SetAssociativeCache<Integer, Character> sac1 = new
SetAssociativeCache<>(2,3,policy1);

sac1.put(0, 'a');
sac1.put(1, 'b');
sac1.put(2, 'c');
sac1.put(3, 'd');
sac1.put(4, 'e');
sac1.put(5, 'f');
sac1.put(6, 'g'); //full set, replaces (3,'d')
sac1.put(1, 'z'); //overwrites (1, 'b') to (1,'z') and is now MRU within set
sac1.put(7, 'y'); //full set, replaces (1, 'z')
sac1.get(2);      //(2,'c') is now MRU within set
sac1.put(8, 'w'); //full set, replaces (2,'c')
sac1.get(1);      //returns null
sac1.get(2);      //returns null
```

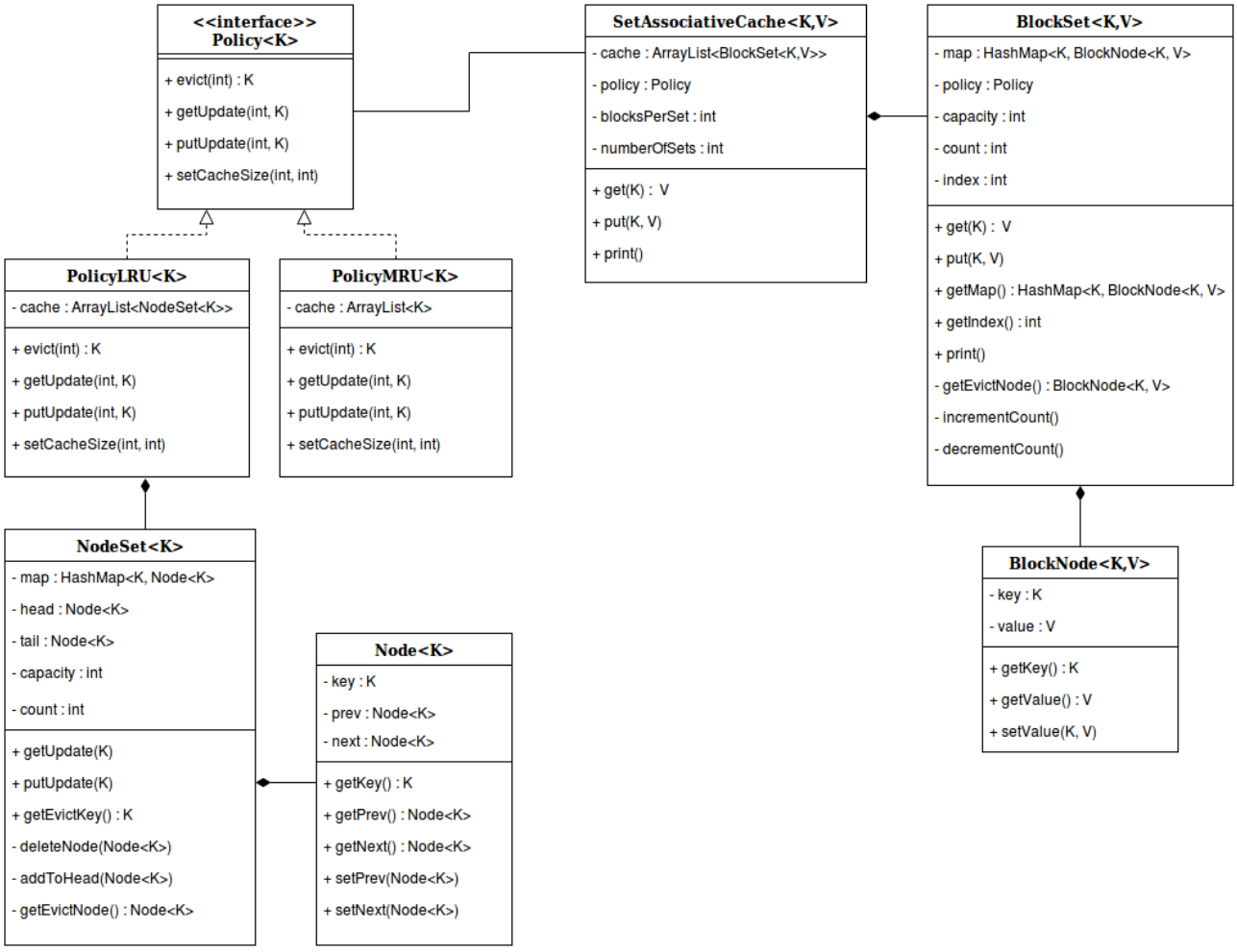
For more help:

1) Example program in the ‘example’ package labeled ExampleProgram.java. You can run it in Eclipse.

2) Test cases in the ‘tests’ package. You can run the tests in Eclipse through the ProgramTestSuite. Be sure to add the proper JUnit libraries to the project (directions on page 1).

3) JavaDocs are available in the ‘doc’ folder.

Design and Implementation:



For the cache, there are 4 relevant classes:

1) SetAssociativeCache
2) BlockSet
3) BlockNode
4) Custom Policy class that implements the Policy interface

The SetAssociativeCache is initialized with (int blocksPerSet, int numberOfSets, Policy policy), specified by the user. This is the object that the user can directly interact with using put(K key, V value) and get(K key) methods. Once initialized it will create BlockSets of size blocksPerSet in an an ArrayList of size numberOfSets.

The BlockSet - Represents a set within the cache. This object helps the cache handle put and get requests within its own set using a HashMap that stores Keys and BlockNodes. A reference to a Policy is also contained here passed by the cache. This is where the replacement policy's eviction method is called to handle evictions within the set whenever the put and get methods are called by the cache.

Each BlockSet also contains an index that represents its own location within the cache which will be passed to the Policy to determine which item to evict.

The BlockNode - Represents individual entries within the cache. It contains a Key used for look up and a Value as stored data.

The Policy interface contains 4 methods:

evict(int index) : K key - The BlockSet calls this method when evicting an item. Given an index by a BlockSet, returns a key that the cache will use to look up the item to evict.

getUpdate(int index, K key) – Is called whenever the BlockSet performs a get for the cache. This lets the policy maintain its eviction methods. The index represents the set within the cache that is being accessed. The key is used for the policy to store to return later when evicting.

putUpdate(int index, K key) - Is called whenever the BlockSet performs a put for the cache. This lets the policy maintain its eviction methods. The index represents the set within the cache that is being accessed. The key is used for the policy to store to return later when evicting.

I decided to put these updates into separate functions in case the user wishes to make a distinction between the two in their own replacement policies.

setCacheSize(int blocksPerSet, int numberOfSets) – When the cache is initialized, this method is called to let the Policy know of the cache's dimensions.

The custom policies:

The PolicyLRU implements the Policy interface. It uses a an ArrayList to maintain sets of blocks much like the SetAssociativeCache. However, the blocks themselves are nodes that form a doubly linked list amongst each other in a set. Whenever the cache calls its put and get functions, the Policy moves the accessed node to the front of the linked list. The least recently used node is the tail node and is returned when the evict method is called. PutUpdate() and getUpdate() take O(1) time to lookup and move nodes. Evict also takes O(1) time as it just needs to remove and return the tail key.

An alternative implementation is to have each set maintain timestamps of each node. However, this will require comparing timestamps when Evict() is called and is more expensive time wise.

The PolicyMRU implements the Policy interface. It uses an ArrayList to represent the cache. Each index of the list represents a set within the cache and contains a single item – a key. That key is the most recently used item of a set. Whenever the cache performs a get or put, the key used in those calls replaces the most recently used key within the array list. When an eviction is performed, it returns the key located at a particular index. Since there is only 1 element in each slot of the array, retrieving the key takes O(1) time given the index of the set.

Notes:

If there is anything I can clarify please let me know, I will be more than happy to help! Any feedback would be great. - Daniel Chunn