N-Way Set-Associative Cache Design

Problem Statement:
The happy-cache library is a Scala implementation of an N-Way, set-associative cache. At its simplest, a cache is a mapping from key to value. In the case of a direct-mapped cache, a key maps to exactly one entry in the cache. The problem with this design is there is no tolerance for collision. To solve this issue, a cache may be designed associatively - a key maps to a set of N entries where its value may exist anywhere in the set. While the more associative a cache is the fewer collisions occur, the larger the set the larger the performance hit associated with locating a particular entry in that set. Further, no optimization can be done on the underlying data structure that limits support for a client-implemented replacement policy. This design aims to minimize this trade-off by storing a set of entries in both a map and a sorted set.

Requirements:
- The cache itself is entirely in memory (i.e. it does not communicate with a backing store or use any I/O)
- 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 will be the same type and all values will be the same type.
- Design the interface as a library to be distributed by clients. Assume that the client doesn't have source code to your library and that internal data structures aren't exposed to the client.
- The design should allow for any replacement algorithm to be implemented by the client. Please provide the LRU and MRU algorithms as part of your solution.

Implementation:

1. **Cache Structure:**
   The cache is an implementation of `Cache[K,V]` trait. The private, default implementation `CacheImpl[K,V]` is a simple immutable Vector with fixed-size `numSet`. The `get` function implements retrieval of elements, returning an Option to gracefully handle missing values. The `put` function implements insertion, and returns `Unit`. The runtime complexity of `get` and `put` are discussed in the next section. Conceptually, each index of the cache stores a "set". A key is deterministically mapped to a set by `key.hashCode() % numSet`.

2. **Cache Set Structure:**
   A set is implemented as a 2-tuple of immutable HashMap and immutable SortedSet, each of fixed-size `numEntry`. Both the map and the set store pointers to the entries in that set (they are not storing duplicate copies of the entry itself). The two forms of storage are respectively intended to allow for constant access time and to allow for worst-case O(log `numEntry`) insertions and removals regardless of the replacement algorithm used (Scala sorted set is a tree set under the hood). The set is sorted based on the implicit ordering provided by `CacheEntry` implementation.

3. **Cache Entry Structure:**
   An entry is an implementation of `CacheEntry[K,V]` trait. A cache entry must have `key:K`, `value:V`, and `timeUpdated:Long` fields. A cache entry must implement `copy` and `compare` functions that respectively allow for updating a field in an entry and sorting the entries for optimal replacement. The private, default implementation `CacheEntryTime[K,V]` is ordered by `timeUpdated`, which is a timestamp in nanoseconds.

4. **Factory:**
   To hide the internal implementation details of the cache, a factory method `cache` is provided to the client which takes an `update` function. `Update` takes a sorted set and an entry to add, and returns the updated sorted set and the element removed. Returning the element removed is required for the internal maintenance of the secondary storage of the entries (the map). `Cache` gives the client the flexibility to implement their own replacement strategy

5. **Replacement Algorithms:**
   The replacement policies are implemented as `update` functions which are passed to the general `cache` factory method.
   a. **LRU:**
      Given the sorted set is sorted by timeUpdate ascended, to replace least recently used simply return (tail of the set + new element, head of the set)
   b. **MRU:**
      Return (front of the set + new element, last element of set)

Included Files:
happy-cache-master.zip**:** Please (1) unzip source code and import into IDE as SBT project (2) refer to README.md to generate archive and use the library as a dependency.

Client API:
Please refer to the Scaladoc available for this package, which can be generated after importing the SBT project with `sbt doc`. This will create a file `/target/scala-2.12/api/index.html`, which can be opened in any browser.

Repository:
To clone the repo directly see https://github.com/stephanietortora/happy-cache.