Functionality

The LUR cache consist of three Hashmaps;
The first hashmap, maps Keys to Values **<Key, Value>**;
The second hashmap, maps Keys to the round they were inserted **<Key, Round>**;
The third hashmap, maps the round the Keys were inserted with the corresponding Keys **<Round, Key>**. It is a reverse hashmap of the second one.
When a pair is inserted, all hashmaps are updated.

Insert a record <K1, V1>

<Key, Value>

K1	V1

<Key, Round>

K1	r1

<Round,Key>

r1	K1

That first hashmap serves the quick retrieval of Values given a known Key. The two others serve the quick detection of the <Key, Value> pair that needs to be deleted if the cache reaches its full capacity.

Insert record <K6, V6> when cache is at full capacity

max_capacity = 5

<Key, Value>

K1	V1
K2	V2
K3	V3
K4	V4
K5	V5

<Key, Round>

K1	1
K2	2
K3	3
K4	4
K5	5

<Round,Key>

1	K1	← Oldest item
2	K2	
3	K3	
4	K4	
5	K5	

↓

max_capacity = 5

<Key, Value>

K6	V6
K2	V2
K3	V3
K4	V4
K5	V5

<Key, Round>

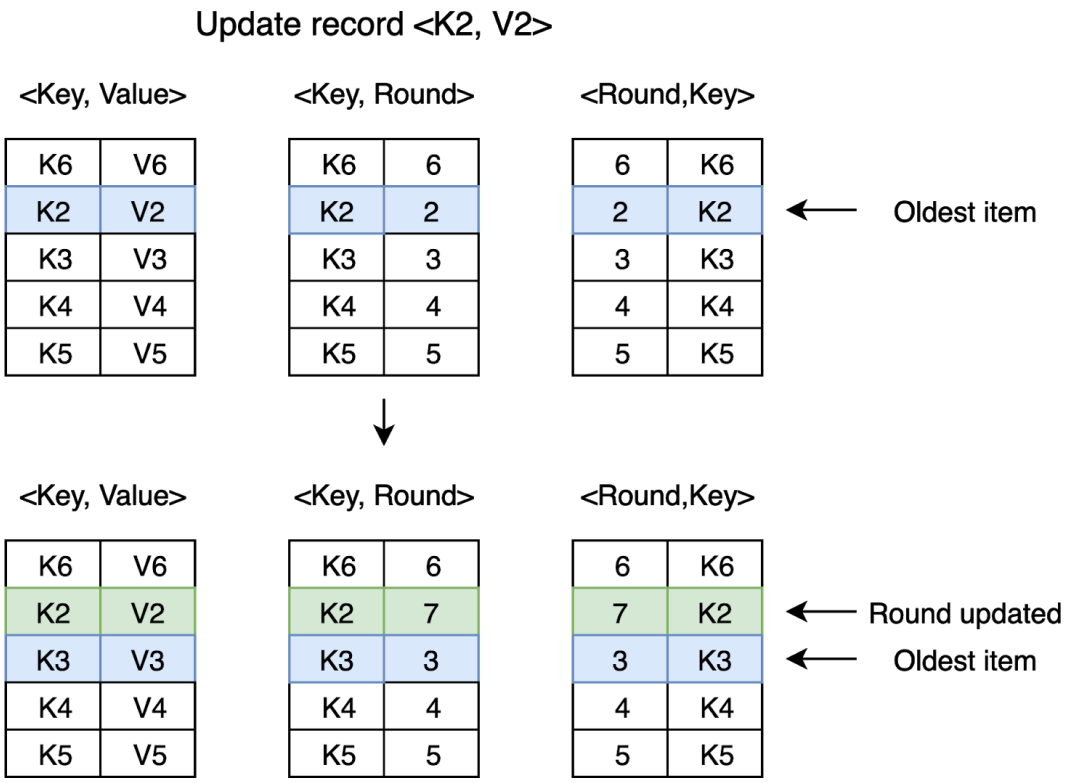
K6	6
K2	2
K3	3
K4	4
K5	5

<Round,Key>

6	K6	← New item
2	K2	← Oldest item
3	K3	
4	K4	
5	K5	

When the maximum capacity is reached, the oldest Key is requested via the round of the oldest insertion from hashmap <Round, Key>. Then, the Key (which is the oldest) is retrieved, and the record of the underlying Key is deleted from hashmaps <Key, Value>, <Key, Round> and <Round, Key>. The new Key-Value pair is inserted. The insertion round of the new Key is the current round.

When an already existing Key pair is inserted, an update of its inserted round effectively occurs. This is facilitated by the <Key, Round> hashmap. The previous round of the Key is found via <Key, Round>, and its record at <Round, Key> and <Key, Record> is updated.



Implementation

The structure has been implemented as a C++ Template Class. That makes it generic and it can be used with any type of Keys and Values.

The implementation was made in modern C++ standards, utilizing the Standard Template Library (STL) extensively.

The implementation supports multi-threaded functionality with synchronized write and read operations. The policy that was adopted was single-writer/multiple-readers. That is, either many threads can read simultaneously, or only one writer can access the structure. Mutexes and condition variables have been used to fulfill that purpose of mutual exclusion.

Several unit tests have been submitted to thoroughly test the functionality of the structure, its polymorphic flexibility and its correctness with multiple reader/writer threads. The used framework is Catch[1], a header-only library for C++ testing. The files of the library have been placed at the folder */tests/catch*.

[1] "catchorg/Catch2 - GitHub." https://github.com/catchorg/Catch2. Accessed 6 Oct. 2019.

The implementation was made in Linux (kernel version 5.3.1), and the code was compiled with g++ (version 9.1.0).

Documentation

All classes and functions are fully documented with Doxygen[2]. The documentation is located at the older */docs*. In the file */docs/html/index.html*, details regarding the classes and functions of all structures are displayed.

Execution

In the root folder of the projects two executables are included.
The first presents the requested scenario. It can be compiled and executed by running the script */compile_and_run_main.sh*. After the command, the executable is located at the path */build/lru_cache*. This demonstrates a simple use of the cache by inserting and retrieving multiple records.
The second executable consists of the unit tests. It can be compiled and executed by running the script */compile_and_run_main.sh*. After the command, the executable is located at the path */build/tests*. This demonstrates various thorough tests on the structure.

[2] "Doxygen." http://www.doxygen.nl/. Accessed 6 Oct. 2019.