Clients visit our e-commerce webpage to browse products we have for sale.

```
struct ProductDetails{
        int productId;
        std::string productName;
        std::string description;
        std::vector<uint8_t> image;
        std::vector<std::string> comments;
}
```

Our products are stored in a database which has the following interface:

- A method fetchProductDetails which given a product id, will return ProductDetails for that product (if the product exists)
- A method getProductIds that returns a vector of all available productIds in the database

Our profiling shows that fetching ProductDetails is an expensive operation.

Furthermore, we observed that clients tend to focus on the same products on a particular day,

or keep browsing back and forth between the same products.

For this reason it was proposed by a colleague to create a cache that will hold the ProductDetails.

In this exercise you need to:

Create an in-memory cache with LRU (least-recently-used) replacement policy and a predetermined capacity in terms of item count. The cache can hold only up to a maximum number of items. When the cache is full and a new item needs to be cached, the least-recently-used item must be evicted, i.e. the one that has been unused for a longer period of time than the others.

The cache component needs to expose:

- a retrieval method that requests a cached item, which may or not exist
- an insertion method that caches an item

Your code needs to be thread-safe when accessed from multiple threads

The cache implementation should be generic so that it can be reused for other use cases in the future:

- different data type for the key
- different data type for the value

Using the LRU cache implementation, create a class that routes the fetchProductDetails queries either to the cache or to the database, and a simple driver application that demonstrates their use

The actual database can be faked with hardcoded data as we are not focused on this part.

Assume that the database component is thread-safe.