# Add Caching to Key-Value Store

Assume you have a subsystem, **KeyValueStore**, that can get and set key-value data in permanent storage.

* Please design and implement an in-memory cache to be used inside of **KeyValueStore**.
* The keys and values are ASCII strings.

## Requirements

* Support configuring the maximum *storage size*.
  + *storage size* is defined as the sum of the number of bytes of all keys and values
  + This definition is intentionally simplified so don't count the size of other on-disk or in-memory data structures and metadata.
* Support both read-through and write-through caching strategies.
* Support configuring the cache replacement policy. Your implementation should support FIFO and provide the flexibility for adding another policy such as LRU in the future.
* Support get and set methods.
* All operations must be thread-safe.
* Add unit tests to increase confidence in the correctness of your implementation
* Write necessary comments to help readers understand the code.
* Write a README file with step-by-step instructions on how to build and run the tests.
* Choose one of Golang, Python, C, C++, or Java as your implementation language

To simplify the implementation, use a fake **KeyValueStore** which pretends to load and store data from permanent storage but actually just use an in-memory data structure.

For example, here’s a version of **KeyValueStore.get()** using a read-through strategy in Python:

|  |
| --- |
| class **Cache**(object):  def **get**(self, key):  # TODO  return None  def **set**(self, key, value):  # TODO  pass  class **KeyValueStore**(object):  def **\_\_init\_\_**(self):  self.\_store = {}  self.\_cache = Cache()  # NOTE: this code is not thread safe. Your code should be.  def **get**(self, key):  value = self.\_cache.get(key)  if value is not None:  return value  value = self.\_store.get(key)  if value is not None:  # Use read-through strategy.  self.\_cache[key] = value  return value |

Please send us your code along with the README file.