面向对象设计 —— 插拔式缓存系统（语言不限，Java 为示例）

一、作业目标

实现一个支持多种替换策略的泛型缓存系统，具备统一接口，并支持灵活扩展。

本作业不限制编程语言，示例代码使用 Java，仅用于说明接口结构。你可以选择 C++、Python 等任意主流语言实现，前提是符合接口语义与复杂度要求。

二、接口设计

缓存类应使用如下统一接口（可为抽象类或 interface）：

```
public interface Cache<K, V> {
    void put(K key, V value);      // 若存在，则更新value
    V get(K key);                  // 若不存在，返回 null 或抛出异常，需说明处理方
式
    void erase(K key);             // 若不存在，忽略删除
    boolean contains(K key);
    int size();
}
```

每种替换策略通过实现该接口构造自己的缓存类，例如：

```
public class LRUCache<K, V> implements Cache<K, V> { ... }
public class LFUCache<K, V> implements Cache<K, V> { ... }
```

三、需要实现的替换策略

每种策略都需实现独立缓存类，支持上述统一接口：

1. LRUCache（Least Recently Used）

- 最近最少使用策略；
- 每次访问更新使用时间；
- 淘汰最久未访问的键。

2. LFUCache（Least Frequently Used）

- 最少访问次数淘汰；
- 若频率并列，淘汰最早插入者；

- 支持 O(log n) 或更优复杂度实现。

3. FIFOCache（First-In First-Out）

- 按插入顺序淘汰；
- 不考虑访问顺序。

4. RandomReplacementCache

- 淘汰时随机选择一个 key；
- 要求淘汰过程均匀，应使用均匀分布随机数，避免偏差。

5. WeightedCache

- 每个 key 对应一个权重值；
- 按权重概率决定淘汰对象（权重大者留）；
- 权重可在 put 时传入或内部设定。

四、性能要求

- 所有接口操作应尽可能满足最优时间复杂度，例如：
  - `get`, `put`: O(1) 或 O(log n)
- 请在注释中说明各策略的时间/空间复杂度。

五、异常处理与边界约定

- `get(k)` 行为：
  - 若不存在，可选择返回 `null` 或抛出异常，需在文档中说明；
- 不支持容量为 0、插入重复 key 等边界测试；

六、测试要求（必须）

请编写完整的单元测试，验证以下内容：

- 各策略下的 `put` / `get` / `erase` 是否正确；
- 淘汰顺序是否符合策略预期；
- `size()` 与实际内容一致性；
- 随机策略分布合理性（可重复实验）；
- Weighted 策略是否遵守权重分布(使用均匀分布的随机数)；
- 异常与边界情况是否被正确捕获。

推荐使用 JUnit（或 Python unittest、C++ Catch2 等）实现测试代码。

七、支持多线程版本

基于非并发版本，实现一个线程安全版本 `ConcurrentCache<K, V>`，要求：

- 接口与 `Cache<K, V>` 完全一致；
- 支持多线程并发调用下数据一致性；
- 可使用 `synchronized`、`ReentrantLock`、读写锁等标准同步机制；
- 需包含并发单元测试，验证在多线程下不会出现竞态条件或死锁。

可以提供一个代理类，内部组合并发策略

```
class ConcurrentCache<K, V> implements Cache<K, V> {
    private final Cache<K, V> delegate;
    // 内部加锁委托
}
```

八、提交内容

请提交以下文件（语言可灵活调整）：

```
src/
├── Cache/
│   ├── Cache.java                          // 缓存接口
│   └── Impl/
│       ├── FIFOCache.java
│       ├── LRUCache.java
│       ├── LFUCache.java
│       ├── RandomReplacementCache.java
│       └── WeightedCache.java
├── ConcurrentCache/
│   ├── ConcurrentCache.java                // 并发缓存接口
│   └── Impl/
│       ├── ConcurrentFIFOCache.java
│       ├── ConcurrentLRUCache.java
│       ├── ConcurrentLFUCache.java
│       ├── ConcurrentRandomReplacementCache.java
│       └── ConcurrentWeightedCache.java
test/
├── CacheTest.java                          // 策略缓存测试
└── ConcurrentTest.java                     // 并发缓存测试
README.md                                   // 项目文档
```

九、文档说明要求（README）

请在 README.md 中简要说明以下内容：

- 各策略的设计思路与核心数据结构；
- 如何选择各策略适用场景；
- 所有接口的复杂度分析；
- 异常处理与边界行为说明；

- 并发版本中锁的粒度和设计理念（如实现）；
- 可能的话提供CI文件

十、示例代码（可选参考）

```
public static void main(String[] args) {
    Cache<String, Integer> cache = new LRUCache<>(3);
    cache.put("a", 1);
    cache.put("b", 2);
    cache.put("c", 3);
    cache.get("a"); // a is now most recently used
    cache.put("d", 4); // b should be evicted
    System.out.println(cache.contains("b")); // false
}
```

实现中严禁使用任何第三方缓存库，仅允许使用语言自带的标准库。
