# LRU（最近最少使用）

> 由朱悦铭于2025年设计
>
> 部分文档参考自DeepSeek

# 1. LRU的基本原理

## 1. LRU算法的基本思想

**最近最少使用（LRU）** 算法是一种广泛使用的**缓存淘汰策略**，当缓存已满时，该算法会移除最久未被访问的条目。其核心原则是：

- **最近被访问的数据更有可能在短期内被再次使用。**
- **较旧（最近未被使用）的数据可以安全移除，为新条目腾出空间。**

**核心思想：**

- 维护一个缓存条目的有序列表，其中：
  - **头部**表示**最近最多使用（MRU）**的条目。
  - **尾部**表示**最近最少使用（LRU）**的条目。
- 当缓存达到容量时，**优先淘汰LRU条目**。

## 2. LRU示例

以下是LRU缓存操作的**哈希集合表示**，展示了每个访问步骤如何改变集合状态，同时保持与双向链表实现相同的淘汰行为：

### LRU缓存模拟（哈希集合 + 链表视图）

**缓存容量 = 3**
**访问序列：** `1 → 3 → 4 → 2 → 3 → 4 → 5 → 2 → 1`

| 步骤 | 访问 | 哈希集合内容 | 链表顺序（MRU→LRU） | 操作 |
| --- | --- | --- | --- | --- |
| 1 | 1 | {1} | [1] | 插入1 |
| 2 | 3 | {1, 3} | [3, 1] | 插入3 |
| 3 | 4 | {1, 3, 4} | [4, 3, 1] | 插入4 |
| 4 | 2 | {2, 3, 4} | [2, 4, 3] | **淘汰1**，插入2 |
| 5 | 3 | {2, 3, 4} | [3, 2, 4] | 3已在集合中，移动到MRU |
| 6 | 4 | {2, 3, 4} | [4, 3, 2] | 4已在集合中，移动到MRU |
| 7 | 5 | {4, 5, 3} | [5, 4, 3] | **淘汰2**，插入5 |
| 8 | 2 | {4, 5, 2} | [2, 5, 4] | **淘汰3**，插入2 |
| 9 | 1 | {1, 4, 2} | [1, 2, 5] | **淘汰4**，插入1 |

1. **哈希集合的作用**：
    - `HashSet`（此处表示为`{...}`）**跟踪当前缓存中的页面**，支持O(1)的查找。
    - **不维护顺序**，顺序由链表维护。
2. **链表的作用**：
    - `LinkedList`（此处表示为`[...]`）**维护访问顺序**（MRU→LRU）。
    - 每次访问（命中或未命中）都会更新链表。
3. **淘汰逻辑**：
    - 插入新页面时：
        - 若缓存已满，移除链表中**最右端（LRU）的条目**。
        - 被淘汰的条目**从集合和链表中同时移除**。
4. **访问命中**：
    - 若页面存在于集合中（例如步骤5访问`3`），则将其**移动到链表头部（MRU）**，集合内容不变。

## 3. LRU算法的实现

设计要点：

- **`Set<Integer>`**：用于快速判断键是否存在（`contains(key)`操作为O(1)）。
- **`List<Integer>`**：维护访问顺序，**头部**为MRU，**尾部**为LRU。

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Set;

public class LRUCache {
```

```
  private final int capacity;
  private final Set<Integer> set;
  private final LinkedList<Integer> list;
  public LRUCache(int capacity) {
    this.capacity = capacity;
    this.set = new HashSet<>();
    this.list = new LinkedList<Integer>();
  }
  public void access(int key) {
    if (set.contains(key)) {
      list.removeFirstOccurrence(key);
      list.addFirst(key);
    } else {
      if (set.size() >= capacity) {
        int lruKey = list.removeLast();
        set.remove(lruKey);
      }
      set.add(key);
      list.addFirst(key);
    }
  }
  public String getCacheState() { return list.toString(); }
  public static void main(String[] args) {
    LRUCache cache = new LRUCache(3);
    int[] accesses = {1, 3, 4, 2, 3, 4, 5, 2, 1};
    System.out.println("步骤\t访问\t缓存状态（MRU→LRU）");
    for (int i = 0; i < accesses.length; i++) {
      cache.access(accesses[i]);
      System.out.printf("%d\t%d\t%s\n", i + 1, accesses[i],
                        cache.getCacheState());
    }
  }
}
```

**核心方法逻辑：`access(int key)`**

缓存命中（键存在）：

- 从链表中移除键并重新插入头部（使用`removeFirstOccurrence(key)` + `addFirst(key)`）。

缓存未命中（键不存在）：

- 若缓存已满：
  - 移除链表尾部（LRU）的键。
  - 同步从`HashSet`中移除。
- 插入新键：
  - 将键添加到链表头部（MRU）。
  - 插入到`HashSet`中。

## 4. LRU算法的应用场景

1. **数据库缓存（如MySQL缓冲池）**
   - 将频繁访问的数据库页面保留在内存中。
   - 需要空间时淘汰最近最少使用的页面。
2. **操作系统页面置换**
   - 决定哪些内存页交换到磁盘。
   - 通过保留最近使用的页面防止抖动。
3. **浏览器与CDN**
   - 缓存频繁访问的网站或资源。
   - 使用LRU淘汰旧缓存内容。
4. **CPU缓存管理**
   - 通过保留最近访问的数据优化缓存命中率。
5. **分布式缓存（Redis、Memcached）**
   - 使用LRU管理集群中的内存使用。

---

# 2. LRU中的固定帧（PinnedFrames）

`pinnedFrames` 是**固定页的集合**，这些页面**当前不能被LRU算法淘汰**（例如，正在被事务修改的数据库页面）。其核心功能包括：

## 1. 为什么需要`pinnedFrames`？

### (1) 典型用例

- 数据库缓冲池。
- 操作系统页面缓存。

### (2) 无`pinnedFrames`的问题

若LRU淘汰正在使用的页面：

- 数据库事务可能失败（因脏页丢失）。
- 系统可能崩溃（内核访问无效内存时）。

## 2. 主流设计方法

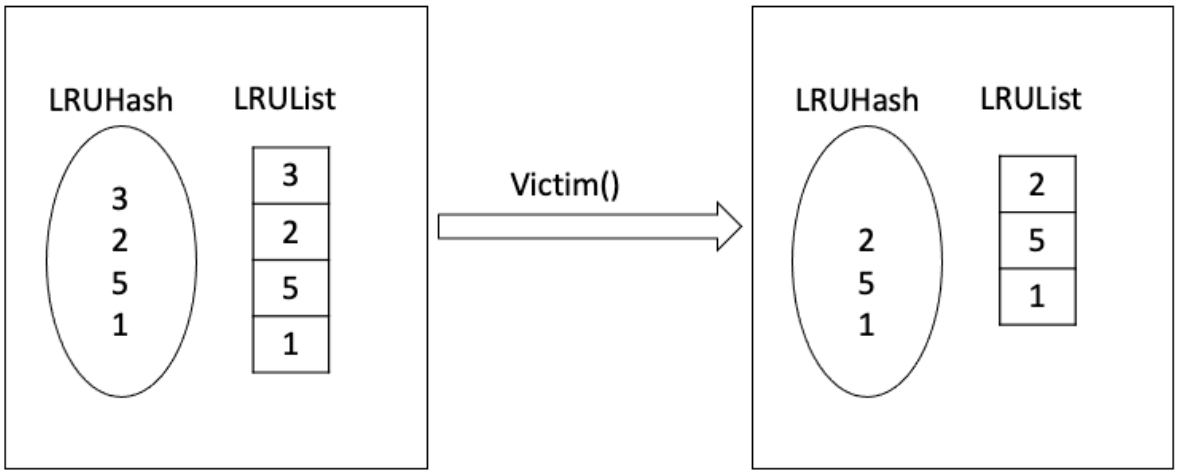
**分离LRU与固定集合**

- **LRU链表**：仅管理**可淘汰的页面**（未固定的页面）。
- **固定集合**：存储所有固定页（通常用`HashSet`或`HashMap`实现）。
- **操作逻辑**：
  - `Pin(frameId)`：从LRU链表中移除页面，并添加到`pinnedFrames`。
  - `Unpin(frameId)`：从`pinnedFrames`移除页面，并重新加入LRU链表。

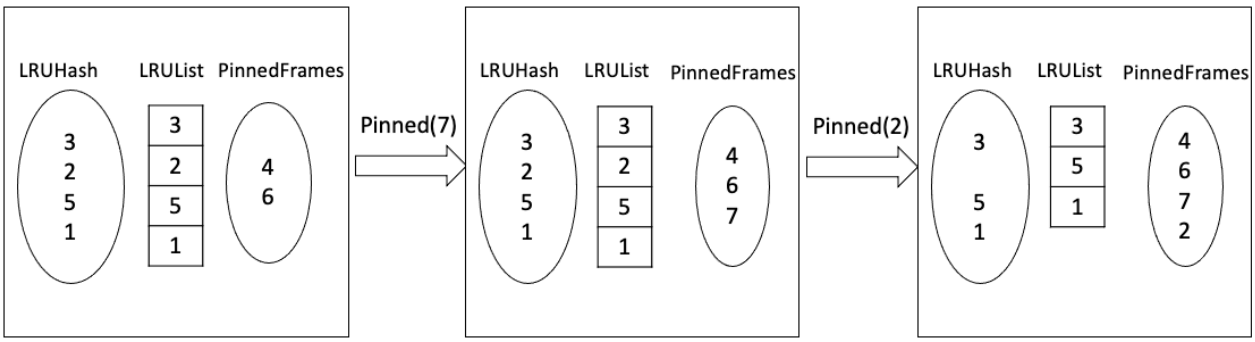
# 练习

在提供的框架中，请按要求完成类 `LRUReplacer` 中的以下三个方法，并通过JUnit测试：

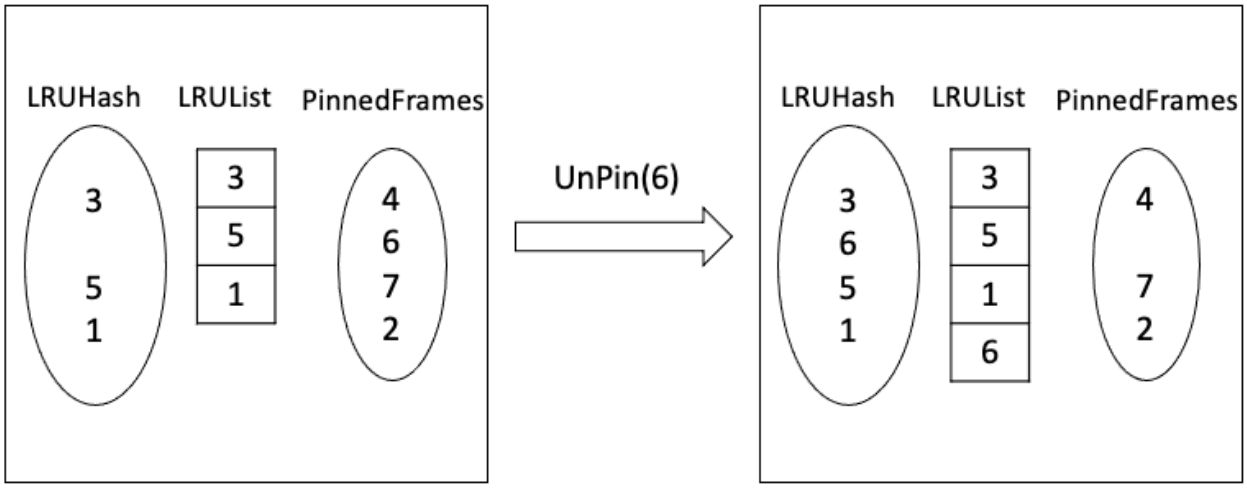
**1. public int Victim()**
从LRU中移除最近最少使用的页面。
默认情况下，LRUList在末尾添加最新页面，并从头部移除最近最少使用的页面。若流程正常，返回对应的`frameId`，否则返回-1。




**2. public void Pin(int frameId)**
固定指定帧。
若`LRUHash`和`pinnedFrames`均不包含`frameId`，则将`frameId`添加到`pinnedFrames`。
若`LRUHash`包含`frameId`，则从`LRUList`和`LRUHash`中移除`frameId`，并添加到`pinnedFrames`。




**3. public void Unpin(int frameId)**
解除固定，允许帧被淘汰。
若`LRUHash`包含`frameId`（表示帧已在缓存中），则不执行操作。
若`LRUHash`不包含`frameId`但`pinnedFrames`包含，则从`pinnedFrames`移除`frameId`，并将其加入`LRUList`和`LRUHash`。


LRUHash
LRUList
PinnedFrames
3
5
1
3
5
1
4
6
7
2
UnPin(6)
LRUHash
LRUList
PinnedFrames
3
6
5
1
3
5
1
6
4
7
2