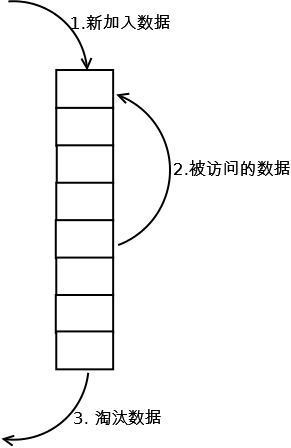
**Lru算法：**

思想：如果一个数据现在被访问，那么将来被访问的概率也更大。



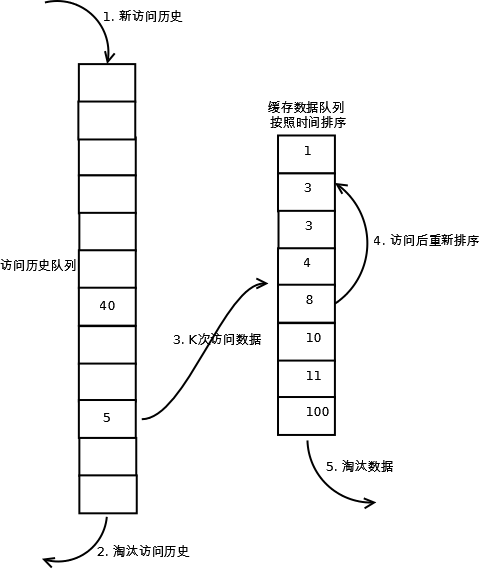
常用实现思路：

用一个链表来管理全部数据，新加入的数据放入链表头，如果有一个数据被访问，也将其放在链表头，当内存满了的时候首先丢弃链表末尾的数据。

算法分析：当存在热点数据的时候，即一部分数据使用处于被频繁访问的情况下，Lru算法情况较好。但是当存在周期性，偶发性缓存数据命中访问的时候Lru效率较低。

Lru算法可能造成**缓存污染**问题：即将不必要的数据频繁从硬盘转移到宝贵的内存资源中。

改进：Lru-K算法，将访问一次马上放入缓存则修改为访问K次才会放入缓存中。当需要淘汰时，会优先淘汰第K次访问距离当前时间最久的数据。



算法分析：

LRU-K具有LRU的优点，同时能够避免LRU的缺点，实际应用中LRU-2是综合各种因素后最优的选择，LRU-3或者更大的K值命中率会高，但适应性差，需要大量的数据访问才能将历史访问记录清除掉。

Lru算法实现：

public class LRUCache {

class CacheNode {

……

}

private int cacheSize;//缓存大小

private Hashtable nodes;//缓存容器

private int currentSize;//当前缓存对象数量

private CacheNode first;//(实现双链表)链表头

private CacheNode last;//(实现双链表)链表尾

}

最关键的移除算法。

public void put(Object key, Object value) {

CacheNode node = (CacheNode) nodes.get(key);

if (node == null) {

//缓存容器是否已经超过大小.

if (currentSize >= cacheSize) {

if (last != null)//将最少使用的删除

nodes.remove(last.key);

removeLast();

} else {

currentSize++;

}

node = new CacheNode();

}

node.value = value;

node.key = key;

//将最新使用的节点放到链表头，表示最新使用的.

moveToHead(node);

nodes.put(key, node);

}