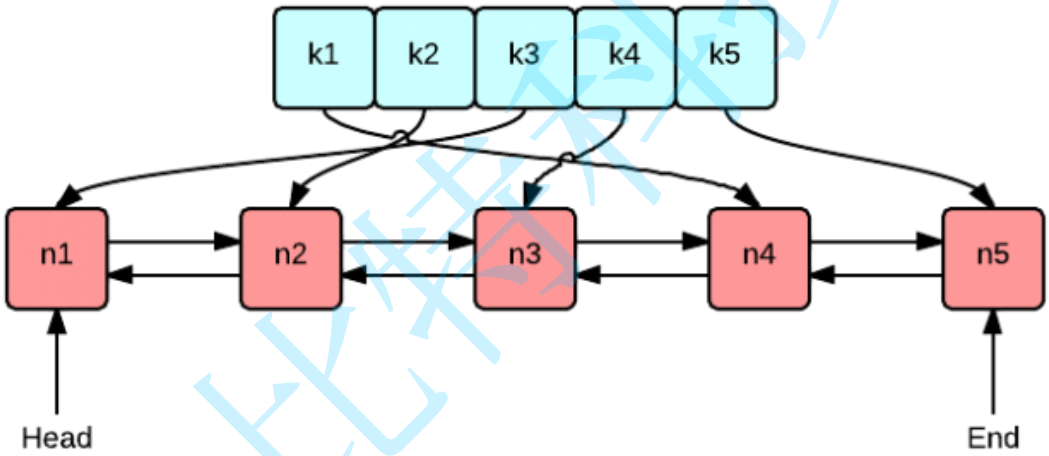
Lessson04 LRU Cache

1.什么是LRU Cache

> LRU是Least Recently Used的缩写，意思是最近最少使用，它是一种Cache替换算法。 什么是Cache？狭义的Cache指的是位于CPU和主存间的快速RAM， 通常它不像系统主存那样使用DRAM技术，而使用昂贵但较快速的SRAM技术。 广义上的Cache指的是位于速度相差较大的两种硬件之间， 用于协调两者数据传输速度差异的结构。除了CPU与主存之间有Cache， 内存与硬盘之间也有Cache，乃至在硬盘与网络之间也有某种意义上的Cache—— 称为Internet临时文件夹或网络内容缓存等。
>
> **Cache的容量有限，因此当Cache的容量用完后，而又有新的内容需要添加进来时， 就需要挑选并舍弃原有的部分内容，从而腾出空间来放新内容。LRU Cache 的替换原则就是将最近最少使用的内容替换掉。**其实，LRU译成最久未使用会更形象， 因为该算法每次替换掉的就是一段时间内最久没有使用过的内容。

2.LRU Cache的实现

> 实现LRU Cache的方法和思路很多，但是要保持高效实现O(1)的put和get，那么使用双向链表和哈希表的搭配是最高效和经典的。使用双向链表是因为双向链表可以实现任意位置O(1)的插入和删除，使用哈希表是因为哈希表的增删查改也是O(1)。
>
> 
>

3.LRU Cache的OJ

> https://leetcode-cn.com/problems/lru-cache/
>
> ```
> /*具体LRU Cache的过程和顺序，参考这个OJ题中的举例描述：
>
> LRUCache cache = new LRUCache( 2 /* 缓存容量 */ );
>
> cache.put(1, 1);  // 插入<1,1>
>
> cache.put(2, 2);  // 插入<2,2>
>
> cache.get(1);       // 返回  1
>
> cache.put(3, 3);    // 该操作会使得密钥 2 作废
>
> cache.get(2);       // 返回 -1 (未找到)
> ```

```
cache.put(4, 4);    // 该操作会使得密钥 1 作废

cache.get(1);       // 返回 -1 (未找到)

cache.get(3);       // 返回  3

cache.get(4);       // 返回  4

*/
class LRUCache {
public:
 LRUCache(int capacity) {
     _capacity = capacity;
 }

 int get(int key) {
     // 如果key对应的值存在，则listit取出，这里就可以看出hashmap的value存的是list的iterator的好处：找到key
     // 也就找到key存的值在list中的iterator，也就直接删除，再进行头插，实现O(1)的数据挪动。
     auto hashit = _hashmap.find(key);
     if(hashit != _hashmap.end())
     {
         auto listit = hashit->second;
         pair<int, int> kv = *listit;

         _list.erase(listit);
         _list.push_front(kv);
         _hashmap[key] = _list.begin();
         return kv.second;
     }
     else
     {
         return -1;
     }
 }

 void put(int key, int value) {
     // 1.如果没有数据则进行插入数据
     // 2.如果有数据则进行数据更新
     auto hashit = _hashmap.find(key);
     if(hashit == _hashmap.end())
     {
         // 插入数据时，如果数据已经达到上限，则删除链表头的数据和hashmap中的数据，两个删除操作都是O(1)
         if(_list.size() >= _capacity)
         {
             _hashmap.erase(_list.back().first);
             _list.pop_back();
         }

         _list.push_front(make_pair(key,value));
         _hashmap[key] = _list.begin();
     }
     else
     {
         // 再次put，将数据挪动list前面
         auto listit = hashit->second;
```

```
            pair<int, int> kv = *listit;
            kv.second = value;

            _list.erase(listit);
            _list.push_front(kv);
            _hashmap[key] = _list.begin();
        }
    }

private:
    list<pair<int, int>> _list;             // 将最近用过的往链表的投上移动，保持LRU
    size_t _capacity;                       // 容量大小，超过容量则换出，保持LRU

    unordered_map<int, list<pair<int, int>>::iterator> _hashmap;
    // 使用unordered_map，让搜索效率达到O(1)
    // 需要注意：这里最巧的设计就是将unordered_map的value type放成list<pair<int,
int>>::iterator，因为这样，当get一个已有的值以后，就可以直接找到key在list中对应的
iterator，然后将这个值移动到链表的头部，保持LRU。
};

/**
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache* obj = new LRUCache(capacity);
 * int param_1 = obj->get(key);
 * obj->put(key,value);
 */
```