散列表下：为什么散列表和链表经常会一起使用?

链表那一节我们讲如何使用链表来实现LRU缓存淘汰算法，但是链表实现LRU缓存淘汰算法的时间复杂度是O(n),通过散列表可以将这个时间复杂度降低到O(1);

Redis的有序集合是使用跳表来实现的，跳表可以看做一种改进版的链表，当时我们也提到，Redis的有序集合不仅使用了跳表，还使用了散列表。

LRU缓存淘汰算法

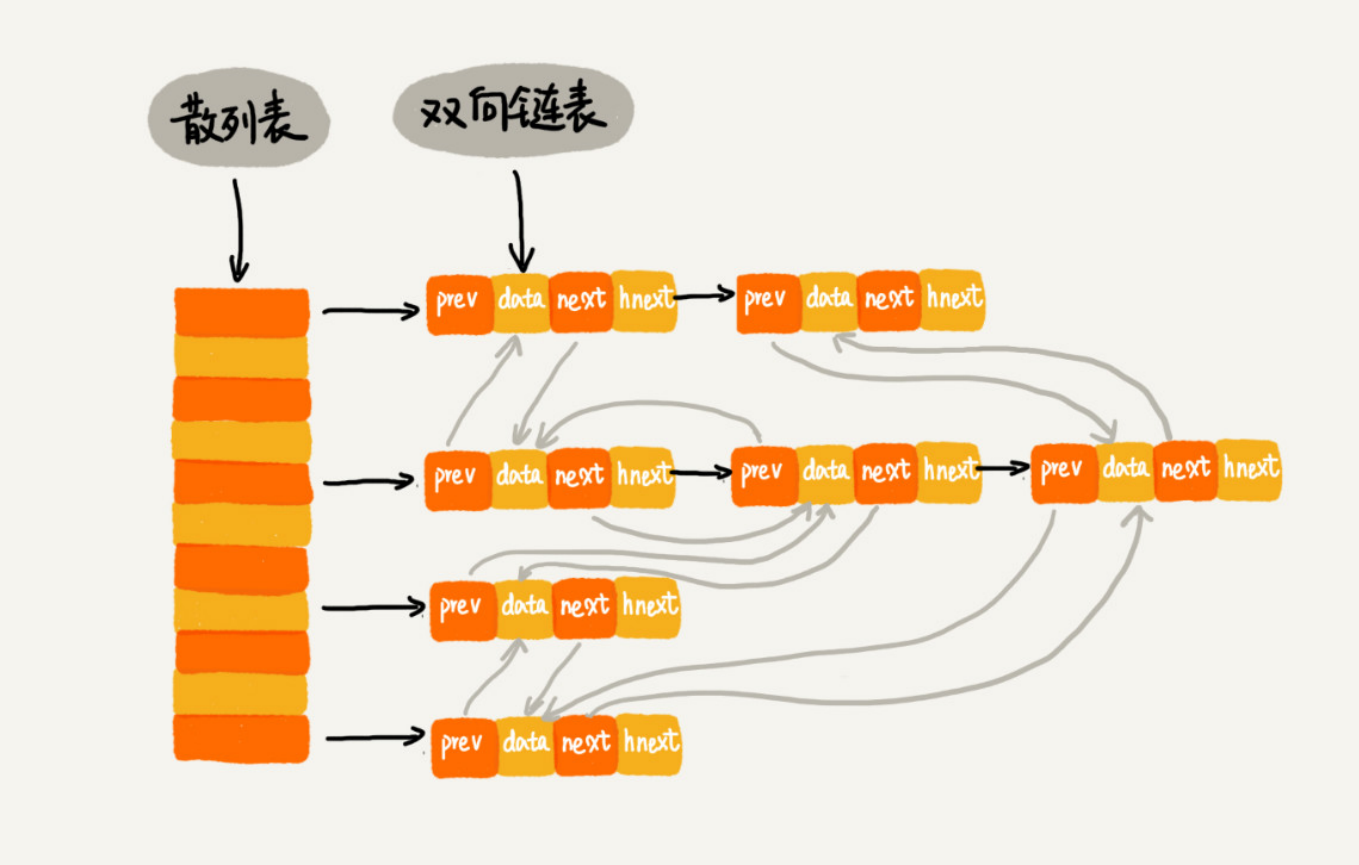
借助散列表可以将LRU缓存淘汰算法的时间复杂度降低到O(1)。

需要删除数据的时候，我们可以直接将链表头部的节点删除； 当要缓存某个数据的时候，先在链表中查找这个数据。如果没有找到，则直接将数据放到链表的尾部；如果找到了，我们，就把他移动到链表的尾部。因为要查找数据，所以单纯的使用链表实现的LRU缓存淘汰算法的时间复杂很高，是O(n);

实际上，一个缓存系统主要包括下面的几个操作：

1. 往缓存中添加一个数据；
2. 从缓存中删除一个数据；
3. 在缓存中查找一个数据；

这三种操作单纯使用链表的话，都需要查找，时间复杂度只能是O(n)； 但将散列表和链表两种数据结构相结合使用，时间复杂度就可以降低到O(1). 具体的结构是如下图：



链表的每一个结点处理存储数据data，前驱结点prev，后继结点next之外，还有一个特殊的字段hnext；

在每个结点会在两个链中，一个链式刚刚我们提到的双向链表，另一个链式散列表中的拉链。前驱和后继指针是为了将结点串在双向链表中，hnext指针是为了将结点穿在散列表的拉链中。

这样的数据结构如何实现缓存的三个操作呢，使得时间复杂度是O(1)? **如何查找一个数据** 散列表中查找数据的时间复杂度接近于O(1),这样在缓存中查找一个数据就很快。

如何删除一个数据 需要找到数据所在的结点，然后将结点删除。在O(1)的时间复杂度中找到要删除的结点，然后通过双向链表的前驱指针，实现删除结点操作，只需要O(1)的时间复杂度；

如何天剑一个数据 添加数据需要先看这个数据是否在缓存中。如果已经在其中，需要将其移动到双向链表的尾部；如果不在其中，还要看缓存有没有满。如果满了，则将双向链表的头部的节点删除，然后再将数据放到链表的尾部；如果没有满，就直接将数据放到链表的尾部。

Redis有序集合

在有序集合中，每个成员对象对两个重要的属性，key键值和score分值。我们不仅会通过score来查找数据，还会通过key来查找数据。

我们可以通过用户的ID来查找积分信息，也可以通过积分区间来查找用户ID或者姓名信息。

细化一个Redis有序集合的操作，就是下面这样：

1. 添加一个成员对象；
2. 按照键值来删除一个成员对象；
3. 按照键值来查找一个成员对象；
4. 按照分值区间查找数据，比如查找积分在[100,356]之间的成员对象；
5. 按照分值从小到大排序成员变量；

如果仅仅按照分值将成员对象组织成跳表的结构，那么按照键值来删除，查询对象就会很慢，解决方法与LRU缓存淘汰算法的解决方法类似。我么可以再按照键值构建一个散列表，这样按照key来删除，查找一个成员对象的时间复杂度就变成了O(1)。

Java LinkedHashMap

LinkedHashMap 并没有这么简单，其中的“Linked”也并不仅仅代表它是通过链表法解决散列冲突的。

```
HashMap<Integer, Integer> m = new LinkedHashMap<>();
m.put(3, 11);
m.put(1, 12);
m.put(5, 23);
m.put(2, 22);

for (Map.Entry e : m.entrySet()) {
System.out.println(e.getKey());
}
```

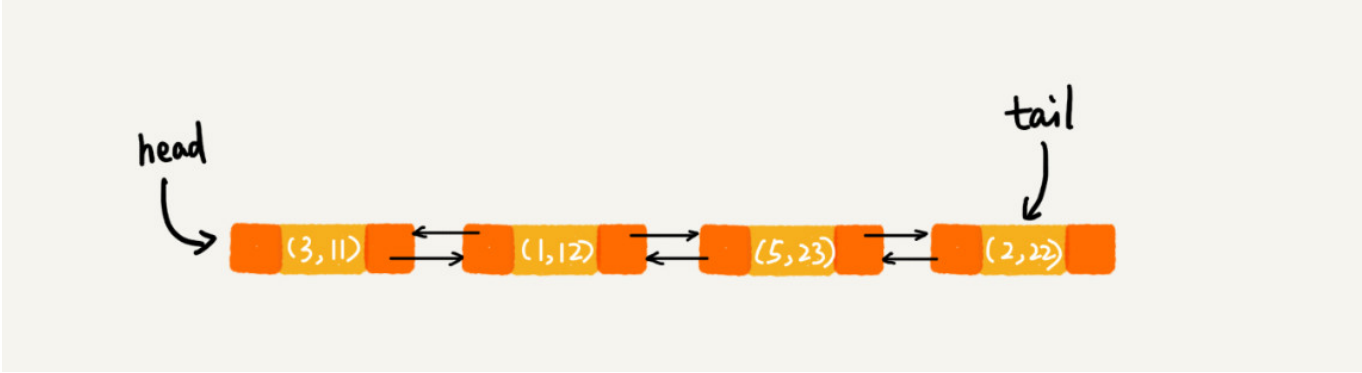
上面代码会按照数据插入的顺序依次来打印。 很奇怪，散列表中的数据经过散列函数打乱之后无规律存储的，这里如何实现按照数据的插入顺序来遍历打印的呢?

LinkedHashMap也是通过散列表和链表组合在一起实现的。实际上他不仅支持按照插入顺序遍历数据，还支持按照访问顺序来遍历数据。

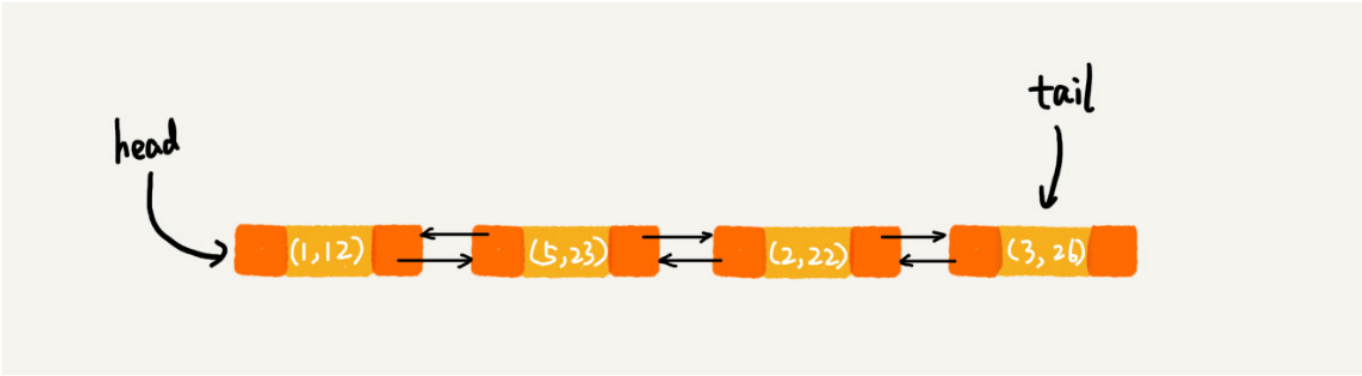
```
// 10是初始大小，0.75是装载因子，true是表示按照访问时间排序
HashMap<Integer, Integer> m = new LinkedHashMap<>(10, 0.75f, true);
m.put(3, 11);
m.put(1, 12);
m.put(5, 23);
m.put(2, 22);

m.put(3, 26);
m.get(5);

for (Map.Entry e : m.entrySet()) {
System.out.println(e.getKey());
}
```

打印结果是1,2,3,5；

每次调用put()函数，往LinkedHashMap中添加数据的时候，都会将数据添加到链表的尾部，所以就变成下图的顺序这样：



经过第8行代码中，再次将键值为3的数据放入到LinkedHashMap中的时候，会查找这个键值是否已经存在，然后将已经存在的（3,11）删除，并且将新的（3,26）放入到链表的尾部。就变成了下面这个样子：



最后打印出来的顺序是1,2,3,5；

其实按照访问时间排序的LinkedHashMap本身就是一个支持LRU缓存淘汰策略的缓存系统。

**LinkedHashMap是通过双向链表和散列表这两种数据结构组合实现的。LinkedHashMap中的“Linked”实际上指的是双向链表，并非指用链表法解决散列冲突。

解答开篇&内容小结

散列表这种数据结构虽然支持非常高效的数据插入、删除、查找操作，但是散列表中的数据都是通过散列函数打乱之后无规律存储的。也就是他无法支持按照某种顺序快速的遍历数据。又因为散列表是动态数据结构，不停的有数据的插入、删除，每当我们希望按顺序遍历的时候，都需要先排序，效率势必很低。因此我们将散列表和链表结合在一起使用。

课后思考

1. 今天讲的几个散列表和链表结合使用的例子中，使用的是双向链表。如果使用单链表，还能否正常工作呢？为什么？

答:可以实现，在删除结点的时候，虽然可以O(1)的找到目标结点，但是要删除操作的时候，需要拿到前一个结点的指针，遍历到前一个结点复杂度会变为O(N),所以还是使用双向链表合适。

2. 假设猎聘网有 10 万名猎头，每个猎头都可以通过做任务（比如发布职位）来积累积分，然后通过积分来下载简历。假设你是猎聘网的一名工程师，如何在内存中存储这 10 万个猎头 ID 和积分信息，让它能够支持这样几个操作： 根据猎头的 ID 快速查找、删除、更新这个猎头的积分信息； 查找积分在某个区间的猎头 ID 列表； 查找按照积分从小到大排名在第 x 位到第 y 位之间的猎头 ID 列表。

答：按照ID设计散列表，可以快速的在时间复杂度是O(1)的情况下实现快速查找，删除和更新这个猎头的信息。 按照积分将数据设计成双向跳表，从而实现积分区间的ID查找；跳表中的数据按照积分大小排序，实现x到y的ID列表输出。