

哈希表

TreeMap分析

- 时间复杂度（平均）

- 添加、删除、搜索： $O(\log n)$

- 特点

- Key 必须具备可比较性

- 元素的分布是有顺序的

- 在实际应用中，很多时候的需求

- Map 中存储的元素不需要讲究顺序

- Map 中的 Key 不需要具备可比较性

- 不考虑顺序、不考虑 Key 的可比较性，Map 有更好的实现方案，平均时间复杂度可以达到 $O(1)$

- 那就是采取[哈希表](#)来实现 Map

需求

- 设计一个写字楼通讯录，存放所有公司的通讯信息
- 座机号码作为 key（假设座机号码最长是 8 位），公司详情（名称、地址等）作为 value
- 添加、删除、搜索的时间复杂度要求是 $O(1)$

```
private Company[] companies = new Company[100000000];
public void add(int phone, Company company) {
    companies[phone] = company;
}
public void remove(int phone) {
    companies[phone] = null;
}
public Company get(int phone) {
    return companies[phone];
}
```

- 存在什么问题？
- 空间复杂度非常大
- 空间使用率极其低，非常浪费内存空间
- 其实数组 `companies` 就是一个哈希表，典型的【空间换时间】

索引	数据
0	
1	
...	
40089008	小码哥
...	
68485438	大码哥
...	
99999999	

哈希表 (Hash Table)

■ 哈希表也叫做散列表 (hash 有“剁碎”的意思)

■ 它是如何实现高效处理数据的?

□ `put("Jack", 666);`

□ `put("Rose", 777);`

□ `put("Kate", 888);`

■ 添加、搜索、删除的流程都是类似的

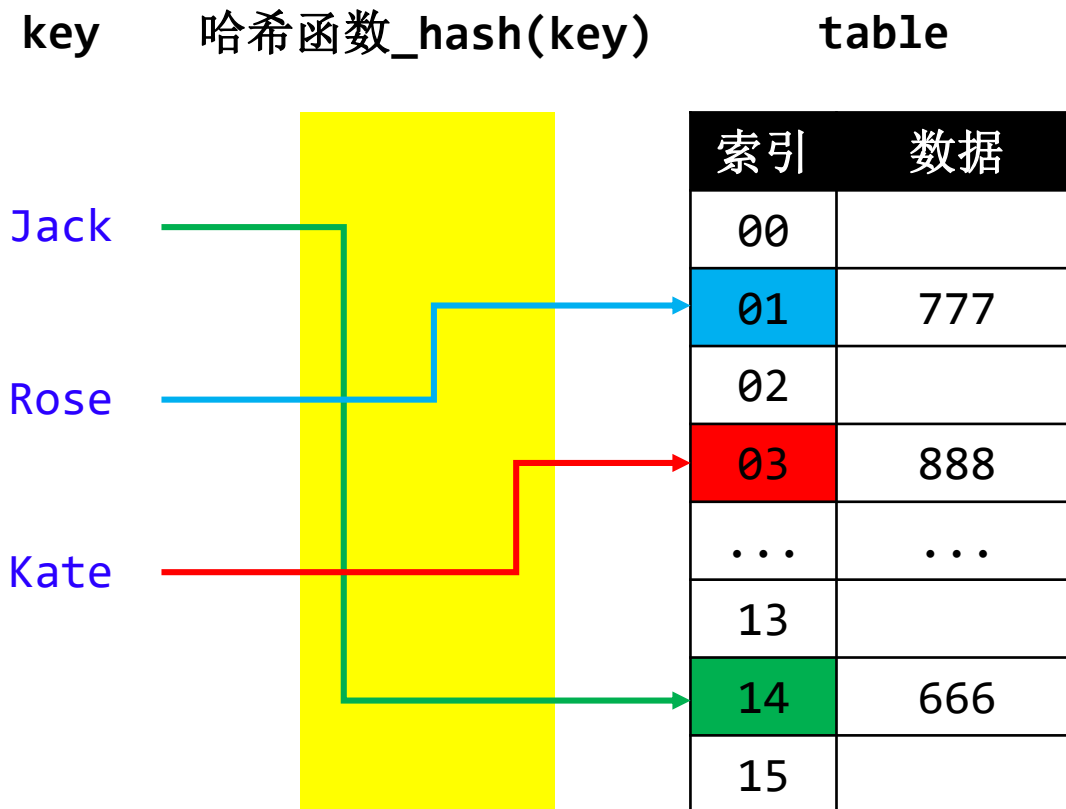
1. 利用哈希函数生成 key 对应的 index 【 $O(1)$ 】

2. 根据 index 操作定位数组元素 【 $O(1)$ 】

■ 哈希表是【空间换时间】的典型应用

■ 哈希函数, 也叫做散列函数

■ 哈希表内部的数组元素, 很多地方也叫 Bucket (桶), 整个数组叫 Buckets 或者 Bucket Array



哈希冲突 (Hash Collision)

■ 哈希冲突也叫做哈希碰撞

□ 2 个不同的 key, 经过哈希函数计算出相同的结果

□ $\text{key1} \neq \text{key2}$, $\text{hash}(\text{key1}) = \text{hash}(\text{key2})$

■ 解决哈希冲突的常见方法

1. 开放定址法 (Open Addressing)

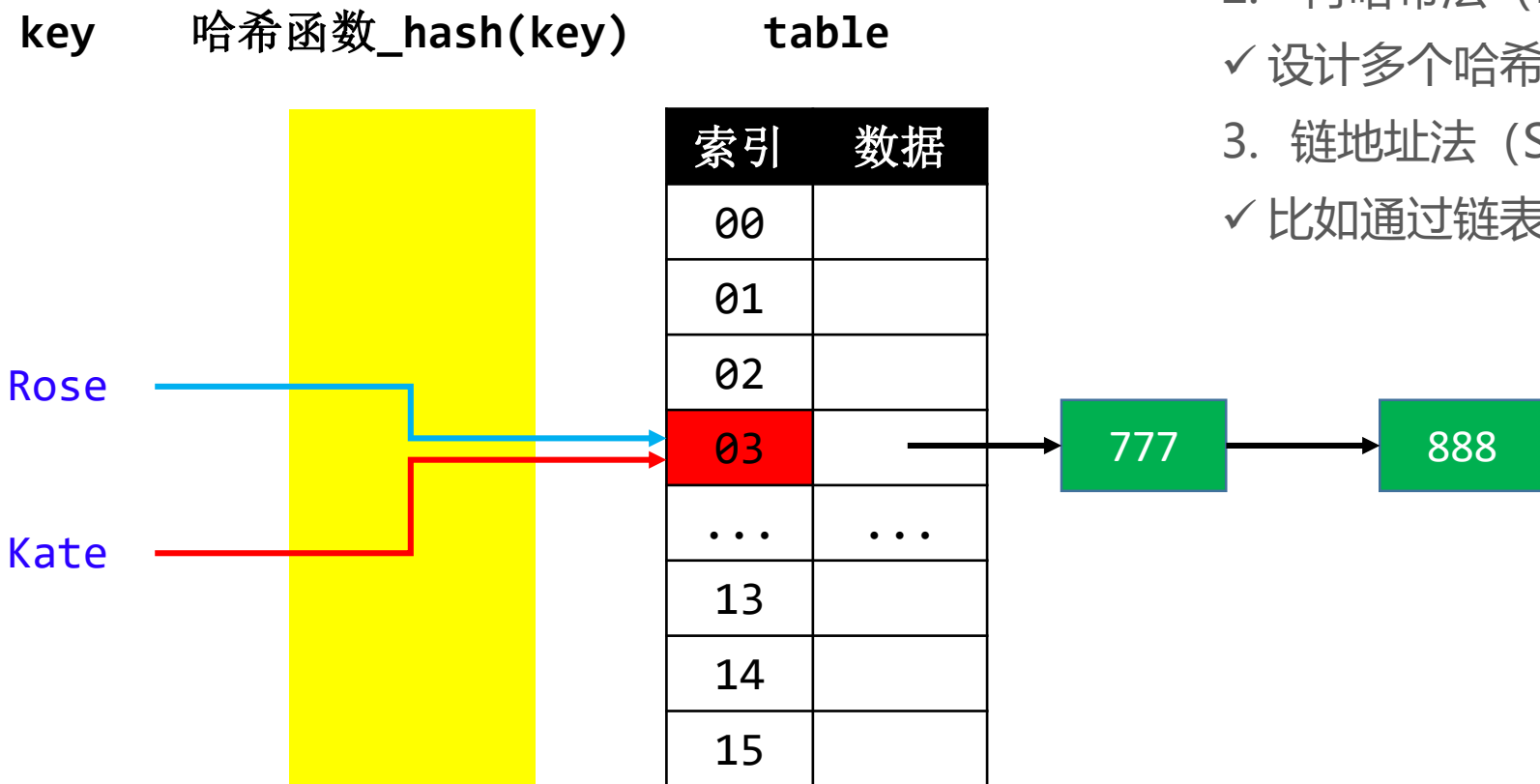
✓ 按照一定规则向其他地址探测, 直到遇到空桶

2. 再哈希法 (Re-Hashing)

✓ 设计多个哈希函数

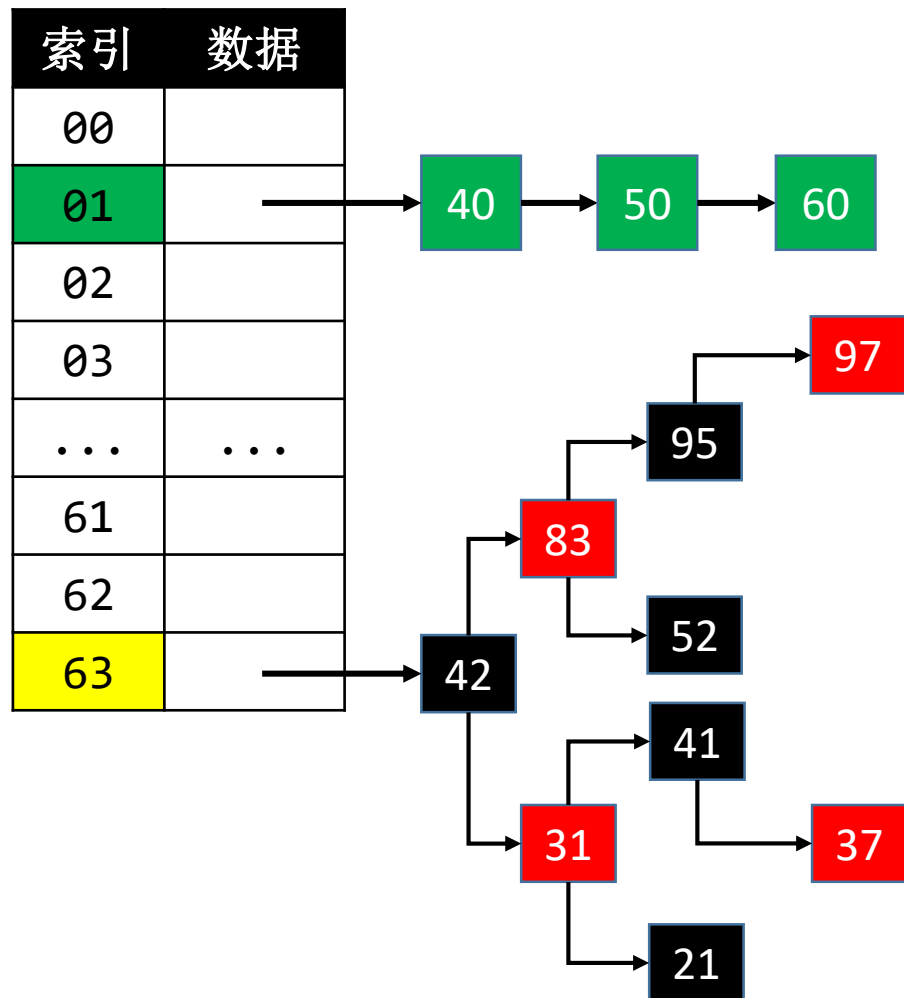
3. 链地址法 (Separate Chaining)

✓ 比如通过链表将同一index的元素串起来



JDK1.8的哈希冲突解决方案

table



- 默认使用单向链表将元素串起来
- 在添加元素时，可能会由单向链表转为红黑树来存储元素
 - 比如当哈希表容量 ≥ 64 且 单向链表的节点数量大于 8 时
- 当红黑树节点数量少到一定程度时，又会转为单向链表
- JDK1.8中的哈希表是使用链表+红黑树解决哈希冲突
- 思考：这里为什么使用单链表？
 - 每次都是从头节点开始遍历
 - 单向链表比双向链表少一个指针，可以节省内存空间

哈希函数

■ 哈希表中哈希函数的实现步骤大概如下

1. 先生成 **key 的哈希值** (必须是**整数**)
2. 再让 **key 的哈希值**跟**数组的大小**进行相关运算, 生成一个**索引值**

```
public int hash(Object key) {  
    return hash_code(key) % table.length;  
}
```

■ 为了提高效率, 可以使用 **&** 位运算取代 **%** 运算【前提: 将数组的长度设计为 **2 的幂** (2^n) 】

```
public int hash(Object key) {  
    return hash_code(key) & (table.length - 1);  
}
```

1100 1010	1011 1100
& 1111	& 1111
1010	1100

■ 良好的哈希函数

□ 让哈希值更加均匀分布 → 减少哈希冲突次数 → 提升哈希表的性能

如何生成key的哈希值

- key 的常见种类可能有

- 整数、浮点数、字符串、自定义对象
- 不同种类的 key，哈希值的生成方式不一样，但目标是一致的
- ✓ 尽量让每个 key 的哈希值是唯一的
- ✓ 尽量让 key 的所有信息参与运算

- 在Java中，HashMap 的 key 必须实现 hashCode、equals 方法，也允许 key 为 null

- 整数

- 整数值当做哈希值
- 比如 10 的哈希值就是 10

```
public static int hashCode(int value) {  
    return value;  
}
```

- 浮点数

- 将存储的二进制格式转为整数值

```
public static int hashCode(float value) {  
    return floatToIntBits(value);  
}
```


Long和Double的哈希值

```
public static int hashCode(long value) {  
    return (int)(value ^ (value >>> 32));  
}
```

```
public static int hashCode(double value) {  
    long bits = doubleToLongBits(value);  
    return (int)(bits ^ (bits >>> 32));  
}
```

■ >>> 和 ^ 的作用是？

□ 高32bit 和 低32bit 混合计算出 32bit 的哈希值

□ 充分利用所有信息计算出哈希值

value	1111 1111 1111 1111 1111 1111 1111 1111 1011 0110 0011 1001 0110 1111 1100 1010
value >>> 32	0000 0000 0000 0000 0000 0000 0000 0000 1111 1111 1111 1111 1111 1111 1111 1111
value ^ (value >>> 32)	1111 1111 1111 1111 1111 1111 1111 1111 0100 1001 1100 0110 1001 0000 0011 0101

字符串的哈希值

■ 整数 5489 是如何计算出来的？

□ $5 * 10^3 + 4 * 10^2 + 8 * 10^1 + 9 * 10^0$

■ 字符串是由若干个字符组成的

□ 比如字符串 jack，由 j、a、c、k 四个字符组成（字符的本质就是一个整数）

□ 因此，jack 的哈希值可以表示为 $j * n^3 + a * n^2 + c * n^1 + k * n^0$ ，等价于 $[(j * n + a) * n + c] * n + k$

□ 在JDK中，乘数 n 为 31，为什么使用 31？

✓ 31 是一个奇素数，JVM会将 $31 * i$ 优化成 $(i \ll 5) - i$

```
String string = "jack";
int hashCode = 0;
int len = string.length();
for (int i = 0; i < len; i++) {
    char c = string.charAt(i);
    hashCode = 31 * hashCode + c;
}
```

```
String string = "jack";
int hashCode = 0;
int len = string.length();
for (int i = 0; i < len; i++) {
    char c = string.charAt(i);
    hashCode = (hashCode << 5) - hashCode + c;
}
```

关于31的探讨

■ $31 * i = (2^5 - 1) * i = i * 2^5 - i = (i \ll 5) - i$

■ 31不仅仅是符合 $2^n - 1$ ，它是个奇素数（既是奇数，又是素数，也就是质数）

□ 素数和其他数相乘的结果比其他方式更容易产成唯一性，减少哈希冲突

□ 最终选择31是经过观测分布结果后的选择

自定义对象的哈希值

```
public class Person {
    private int age;
    private float height;
    private String name;
    private Car car;
    @Override
    public int hashCode() {
        int hash = Integer.hashCode(age);
        hash = 31 * hash + Float.hashCode(height);
        hash = 31 * hash + name.hashCode();
        hash = 31 * hash + car.hashCode();
        return hash;
    }
    @Override
    public boolean equals(Object obj) {
        if (obj == this) return true;
        if (obj == null || obj.getClass() != getClass()) return false;
        Person person = (Person) obj;
        return person.age == age && person.height == height
            && valueEquals(person.name, name)
            && valueEquals(person.car, car);
    }
    private boolean valueEquals(Object v1, Object v2) {
        return v1 == null ? v2 == null : v1.equals(v2);
    }
}
```

■ 思考几个问题

□ 哈希值太大，整型溢出怎么办？

✓ 不用作任何处理

自定义对象作为key

■ 自定义对象作为 key, 最好同时重写 hashCode 、 equals 方法

□ equals : 用以判断 2 个 key 是否为同一个 key

✓ 自反性: 对于任何非 null 的 x, x.equals(x)必须返回true

✓ 对称性: 对于任何非 null 的 x、y, 如果 y.equals(x) 返回 true, x.equals(y) 必须返回 true

✓ 传递性: 对于任何非 null 的 x、y、z, 如果 x.equals(y)、y.equals(z) 返回 true, 那么x.equals(z) 必须返回 true

✓ 一致性: 对于任何非 null 的 x、y, 只要 equals 的比较操作在对象中所用的信息没有被修改, 多次调用 x.equals(y) 就会一致地返回 true, 或者一致地返回 false

✓ 对于任何非 null 的 x, x.equals(null) 必须返回 false

□ hashCode : 必须保证 equals 为 true 的 2 个 key 的哈希值一样

□反过来 hashCode 相等的 key, 不一定 equals 为 true

■ 不重写 hashCode 方法只重写 equals 会有什么后果?

✓ 可能会导致 2 个 equals 为 true 的 key 同时存在哈希表中

哈希值的进一步处理：扰动计算

```
private int hash(K key) {  
    if (key == null) return 0;  
    int h = key.hashCode();  
    return (h ^ (h >>> 16)) & (table.length - 1);  
}
```

装填因子

- 装填因子 (Load Factor) : $\text{节点总数量} / \text{哈希表桶数组长度}$, 也叫做负载因子
- 在JDK1.8的HashMap中, 如果装填因子超过0.75, 就扩容为原来的2倍

TreeMap vs HashMap

- 何时选择TreeMap?

- 元素具备可比较性且要求升序遍历（按照元素从小到大）

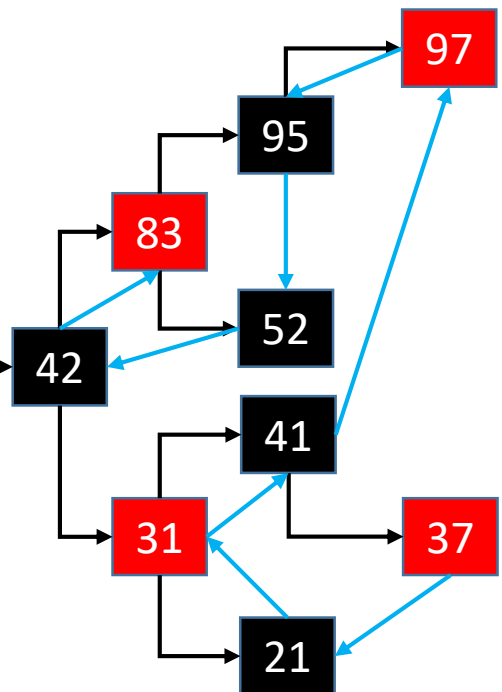
- 何时选择HashMap?

- 无序遍历

LinkedHashMap

■ 在HashMap的基础上维护元素的添加顺序，使得遍历的结果是遵从添加顺序的

索引	数据
00	
01	
02	
03	
...	...
61	
62	
63	



■ 假设添加顺序是

□ 37、21、31、41、97、95、52、42、83

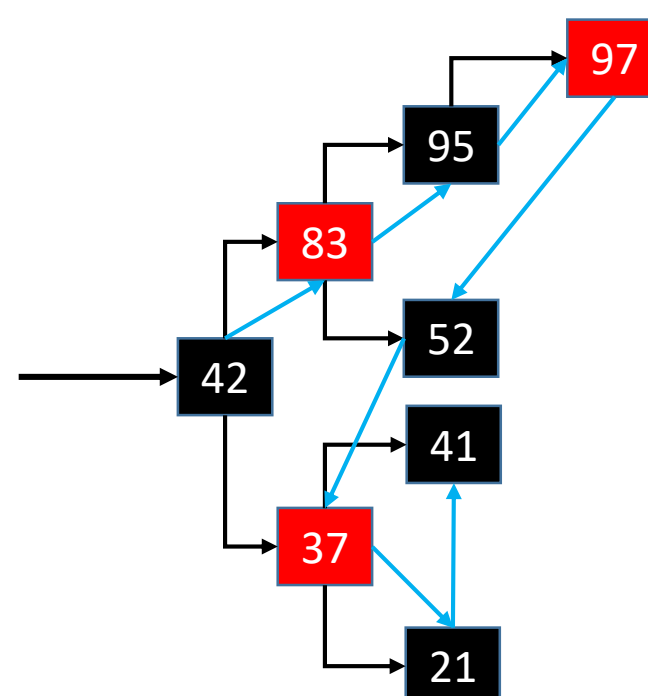
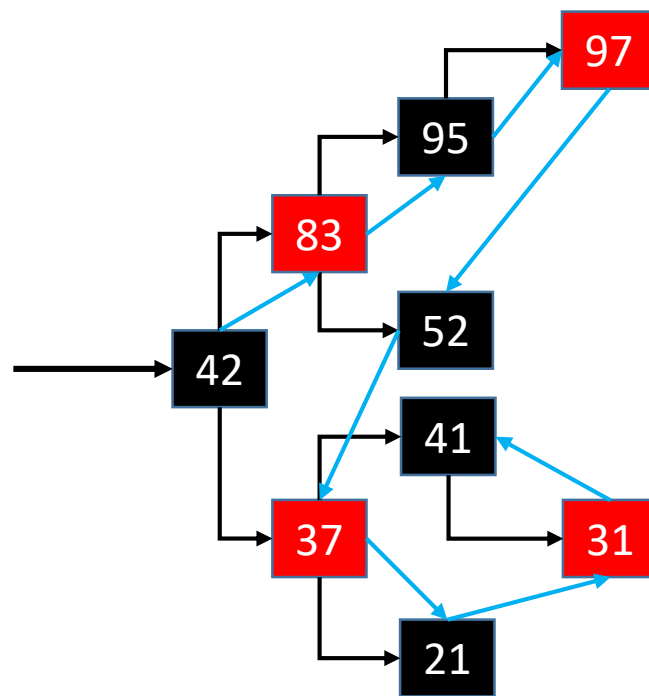
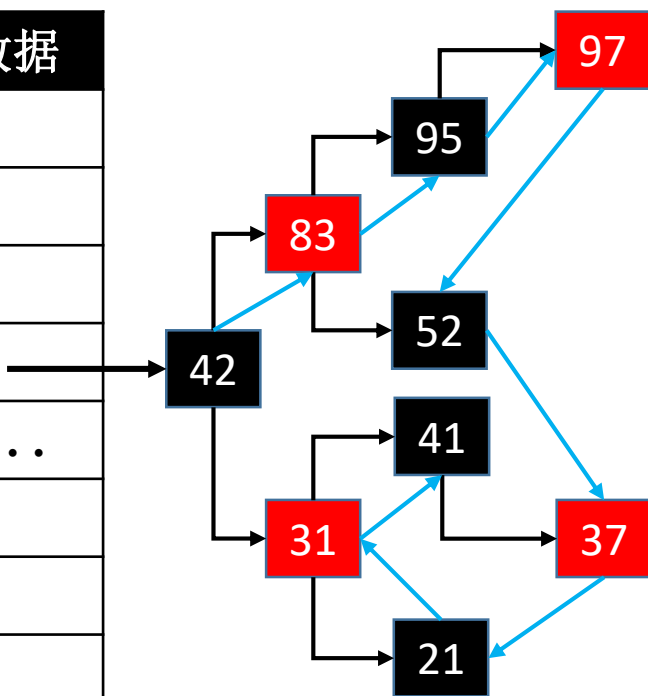
■ 删除度为2的节点node时

□ 需要注意更换 node 与 前驱\后继节点 的连接位置

LinkedHashMap – 删除注意点

- 删除度为2的节点node时 (比如删除31)
- 需要注意更换 node 与 前驱\后继节点 的连接位置

索引	数据
00	
01	
02	
03	
...	...
61	
62	
63	



LinkedHashMap – 更换节点的连接位置



```
// 交换prev
LinkedNode<K, V> tmp = node1.prev;
node1.prev = node2.prev;
node2.prev = tmp;
if (node1.prev != null) {
    node1.prev.next = node1;
} else {
    first = node1;
}
if (node2.prev != null) {
    node2.prev.next = node2;
} else {
    first = node2;
}
```

```
// 交换next
tmp = node1.next;
node1.next = node2.next;
node2.next = tmp;
if (node1.next != null) {
    node1.next.prev = node1;
} else {
    last = node1;
}
if (node2.next != null) {
    node2.next.prev = node2;
} else {
    last = node2;
}
```

关于使用%来计算索引

- 如果使用%来计算索引
- 建议把哈希表的长度设计为素数（质数）
- 可以大大减小哈希冲突

$10\%8 = 2$	$10\%7 = 3$
$20\%8 = 4$	$20\%7 = 6$
$30\%8 = 6$	$30\%7 = 2$
$40\%8 = 0$	$40\%7 = 5$
$50\%8 = 2$	$50\%7 = 1$
$60\%8 = 4$	$60\%7 = 4$
$70\%8 = 6$	$70\%7 = 0$

- 右边表格列出了不同数据规模对应的最佳素数，特点如下
- 每个素数略小于前一个素数的2倍
- 每个素数尽可能接近2的幂（ 2^n ）

下界	上界	素数
2^5	2^6	53
2^6	2^7	97
2^7	2^8	193
2^8	2^9	389
2^9	2^{10}	769
2^{10}	2^{11}	1543
2^{11}	2^{12}	3079
2^{12}	2^{13}	6151
2^{13}	2^{14}	12289
2^{14}	2^{15}	24593
2^{15}	2^{16}	49157
2^{16}	2^{17}	98317
2^{17}	2^{18}	196613
2^{18}	2^{19}	393241
2^{19}	2^{20}	786433
2^{20}	2^{21}	1572869
2^{21}	2^{22}	3145739
2^{22}	2^{23}	6291469
2^{23}	2^{24}	12582917
2^{24}	2^{25}	25165843
2^{25}	2^{26}	50331653
2^{26}	2^{27}	100663319
2^{27}	2^{28}	201326611
2^{28}	2^{29}	402653189
2^{29}	2^{30}	805306457
2^{30}	2^{31}	1610612741