9.哈希表实现

1. 哈希概念

哈希(hash)又称散列，是一种组织数据的方式。从译名来看，有散乱排列的意思。本质就是通过哈希函数把关键字Key跟存储位置建立一个映射关系，查找时通过这个哈希函数计算出Key存储的位置，进行快速查找。

1.1 直接定址法

当关键字的范围比较集中时，直接定址法就是非常简单高效的方法，比如一组关键字都在[0,99]之间，那么我们开一个100个数的数组，每个关键字的值直接就是存储位置的下标。再比如一组关键字值都在[a,z]的小写字母，那么我们开一个26个数的数组，每个关键字acsii码-a ascii码就是存储位置的下标。也就是说直接定址法本质就是用关键字计算出一个绝对位置或者相对位置。这个方法我们在计数排序部分已经用过了，其次在string章节的下面OJ也用过了。

387. 字符串中的第一个唯一字符 - 力扣（LeetCode）

```cpp
class Solution {
public:
    int firstUniqChar(string s) {
        // 每个字母的ascii码-'a'的ascii码作为下标映射到count数组，数组中存储出现的次数
        int count[26] = {0};

        // 统计次数
        for(auto ch : s)
        {
            count[ch-'a']++;
        }

        for(size_t i = 0; i < s.size(); ++i)
        {
            if(count[s[i]-'a'] == 1)
                return i;
        }

        return -1;
    }
};
```

1.2 哈希冲突

直接定址法的缺点也非常明显，当关键字的范围比较分散时，就很浪费内存甚至内存不够用。假设我们只有数据范围是[0, 9999]的N个值，我们要映射到一个M个空间的数组中(一般情况下M >= N)，那么就要借助哈希函数(hash function)hf，关键字key被放到数组的h(key)位置，这里要注意的是h(key)计算出的值必须在[0, M)之间。

这里存在的一个问题就是，两个不同的key可能会映射到同一个位置去，这种问题我们叫做哈希冲突，或者哈希碰撞。理想情况是找出一个好的哈希函数避免冲突，但是实际场景中，冲突是不可避免的，所以我们尽可能设计出优秀的哈希函数，减少冲突的次数，同时也要去设计出解决冲突的方案。

1.3 负载因子

假设哈希表中已经映射存储了N个值，哈希表的大小为M，那么 负载因子 $= \frac{N}{M}$ ，负载因子有些地方也翻译为载荷因子/装载因子等，他的英文为load factor。负载因子越大，哈希冲突的概率越高，空间利用率越高；负载因子越小，哈希冲突的概率越低，空间利用率越低；

1.4 将关键字转为整数

我们将关键字映射到数组中位置，一般是整数好做映射计算，如果不是整数，我们要想办法转换成整数，这个细节我们后面代码实现中再进行细节展示。下面哈希函数部分我们讨论时，如果关键字不是整数，那么我们讨论的Key是关键字转换成的整数。

1.5 哈希函数

一个好的哈希函数应该让N个关键字被等概率的均匀的散列分布到哈希表的M个空间中，但是实际中却很难做到，但是我们要尽量往这个方向去考量设计。

1.5.1 除法散列法/除留余数法

- 除法散列法也叫做除留余数法，顾名思义，假设哈希表的大小为M，那么通过key除以M的余数作为映射位置的下标，也就是哈希函数为：h(key) = key % M。
- 当使用除法散列法时，要尽量避免M为某些值，如2的幂，10的幂等。如果是 2^X ，那么key % 2^X 本质相当于保留key的后X位，那么后x位相同的值，计算出的哈希值都是一样的，就冲突了。如：{63 , 31}看起来没有关联的值，如果M是16，也就是 2^4 ，那么计算出的哈希值都是15，因为63的二进制后8位是 00111111，31的二进制后8位是 00011111。如果是 10^X ，就更明显了，保留的都是10进值的后x位，如：{112, 12312}，如果M是100，也就是 10^2 ，那么计算出的哈希值都是12。
- 当使用除法散列法时，建议M取不太接近2的整数次幂的一个质数(素数)。
- 需要说明的是，实践中也是八仙过海，各显神通，Java的HashMap采用除法散列法时就是2的整数次幂做哈希表的大小M，这样玩的话，就不用取模，而可以直接位运算，相对而言位运算比模更高效一些。但是他不是单纯的去取模，比如M是2^16次方，本质是取后16位，那么用key’ = key>>16，然后把key和key' 异或的结果作为哈希值。也就是说我们映射出的值还是在[0,M)范围内，但是尽量让key所有的位都参与计算，这样映射出的哈希值更均匀一些即可。所以我们上面建议M取不太接近2的整数次幂的一个质数的理论是大多数数据结构书籍中写的理论吗，但是实践中，灵活运用，抓住本质，而不能死读书。（了解）

1.5.2 乘法散列法（了解）

- 乘法散列法对哈希表大小M没有要求，他的大思路第一步：用关键字 K 乘上常数 A (0<A<1)，并抽取出 k*A 的小数部分。第二步：后再用M乘以k*A 的小数部分，再向下取整。
- $h(key) = floor(M \times ((A \times key)\%1.0))$ ，其中floor表示对表达式进行下取整，A∈(0,1)，这里最重要的是A的值应该如何设定，Knuth认为 $A = (\sqrt{5}-1)/2 = 0.6180339887....$ (黄金分割点])比较好。
- 乘法散列法对哈希表大小M是没有要求的，假设M为1024，key为1234，A = 0.6180339887, A*key = 762.6539420558，取小数部分为0.6539420558，M×((A×key)%1.0) = 0.6539420558*1024 = 669.6366651392，那么h(1234) = 669。

1.5.3 全域散列法（了解）

- 如果存在一个恶意的对手，他针对我们提供的散列函数，特意构造出一个发生严重冲突的数据集，比如，让所有关键字全部落入同一个位置中。这种情况是可以存在的，只要散列函数是公开且确定的，就可以实现此攻击。解决方法自然是见招拆招，给散列函数增加随机性，攻击者就无法找出确定可以导致最坏情况的数据。这种方法叫做全域散列。
- $h_{ab}(key) = ((a \times key + b)\%P)\%M$ ，P需要选一个足够大的质数，a可以随机选[1,P-1]之间的任意整数，b可以随机选[0,P-1]之间的任意整数，这些函数构成了一个P*(P-1)组全域散列函数组。假设P=17，M=6，a = 3， b = 4, 则 $h_{34}(8) = ((3 \times 8 + 4)\%17)\%6 = 5$ 。
- 需要注意的是每次初始化哈希表时，随机选取全域散列函数组中的一个散列函数使用，后续增删查改都固定使用这个散列函数，否则每次哈希都是随机选一个散列函数，那么插入是一个散列函数，查找又是另一个散列函数，就会导致找不到插入的key了。

1.5.4 其他方法（了解）

- 上面的几种方法是《算法导论》书籍中讲解的方法。
- 《殷人昆 数据结构：用面向对象方法与C++语言描述 （第二版）》和 《[数据结构(C语言版)].严蔚敏_吴伟民》等教材型书籍上面还给出了平方取中法、折叠法、随机数法、数学分析法等，这些方法相对更适用于一些局限的特定场景，有兴趣可以去看看这些书籍。

1.6 处理哈希冲突

实践中哈希表一般还是选择除法散列法作为哈希函数，当然哈希表无论选择什么哈希函数也避免不了冲突，那么插入数据时，如何解决冲突呢？主要有两种两种方法，开放定址法和链地址法。

1.6.1 开放定址法

在开放定址法中所有的元素都放到哈希表里，当一个关键字key用哈希函数计算出的位置冲突了，则按照某种规则找到一个没有存储数据的位置进行存储，开放定址法中负载因子一定是小于的。这里的规则有三种：线性探测、二次探测、双重探测。

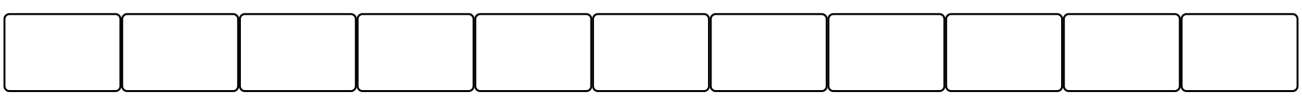
线性探测

- 从发生冲突的位置开始，依次线性向后探测，直到寻找到下一个没有存储数据的位置为止，如果走到哈希表尾，则回绕到哈希表头的位置。
- $h(key) = hash0 = key \% M$，hash0位置冲突了，则线性探测公式为：$hc(key, i) = hashi = (hash0 + i) \% M$， $i = \{1, 2, 3, ..., M-1\}$，因为负载因子小于1，则最多探测M-1次，一定能找到一个存储key的位置。
- 线性探测的比较简单且容易实现，线性探测的问题假设，hash0位置连续冲突，hash0，hash1，hash2位置已经存储数据了，后续映射到hash0，hash1，hash2，hash3的值都会争夺hash3位置，这种现象叫做群集/堆积。下面的二次探测可以一定程度改善这个问题。
- 下面演示 `{19,30,5,36,13,20,21,12}` 等这一组值映射到M=11的表中。

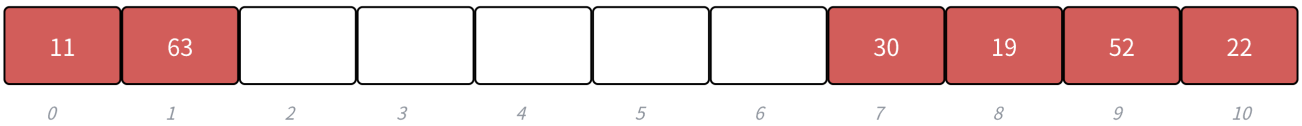
`h(19) = 8, h(30) = 8, h(5) = 5, h(36) = 3, h(13) = 2, h(20) = 9, h(21) = 10, h(12) = 1`

二次探测

- 从发生冲突的位置开始，依次左右按二次方跳跃式探测，直到寻找到下一个没有存储数据的位置为止，如果往右走到哈希表尾，则回绕到哈希表头的位置；如果往左走到哈希表头，则回绕到哈希表尾的位置；
- $h(key) = hash0 = key \% M$，hash0位置冲突了，则二次探测公式为：$hc(key, i) = hashi = (hash0 \pm i^2) \% M$， $i = \{1, 2, 3, ..., \frac{M}{2}\}$
- 二次探测当 $hashi = (hash0 - i^2) \% M$ 时，当hashi<0时，需要hashi += M
- 下面演示 `{19,30,52,63,11,22}` 等这一组值映射到M=11的表中。

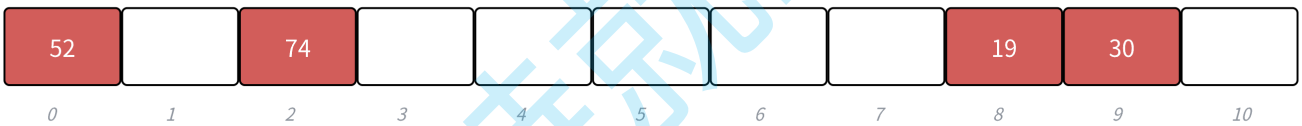


`h(19) = 8, h(30) = 8, h(52) = 8, h(63) = 8, h(11) = 0, h(22) = 0`



双重散列（了解）

- 第一个哈希函数计算出的值发生冲突，使用第二个哈希函数计算出一个跟key相关的偏移量值，不断往后探测，直到寻找到下一个没有存储数据的位置为止。
- $h_1(key) = hash0 = key \% M$，hash0位置冲突了，则双重探测公式为：$hc(key, i) = hashi = (hash0 + i * h_2(key)) \% M$， $i = \{1, 2, 3, ..., M\}$
- 要求 $h_2(key) < M$ 且 $h_2(key)$ 和M互为质数，有两种简单的取值方法：1、当M为2整数幂时，$h_2(key)$ 从[0，M-1]任选一个奇数；2、当M为质数时，$h_2(key) = key \% (M - 1) + 1$
- 保证 $h_2(key)$ 与M互质是因为根据固定的偏移量所寻址的所有位置将形成一个群，若最大公约数 $p = gcd(M, h_1(key)) > 1$，那么所能寻址的位置的个数为 $M/P < M$，使得对于一个关键字来说无法充分利用整个散列表。举例来说，若初始探查位置为1，偏移量为3，整个散列表大小为12，那么所能寻址的位置为{1, 4, 7, 10}，寻址个数为 $12/gcd(12, 3) = 4$
- 下面演示 `{19,30,52,74}` 等这一组值映射到M=11的表中，设 $h_2(key) = key\%10 + 1$



1.6.2 开放定址法代码实现

开放定址法在实践中，不如下面讲的链地址法，因为开放定址法解决冲突不管使用哪种方法，占用的都是哈希表中的空间，始终存在互相影响的问题。所以开放定址法，我们简单选择线性探测实现即可。

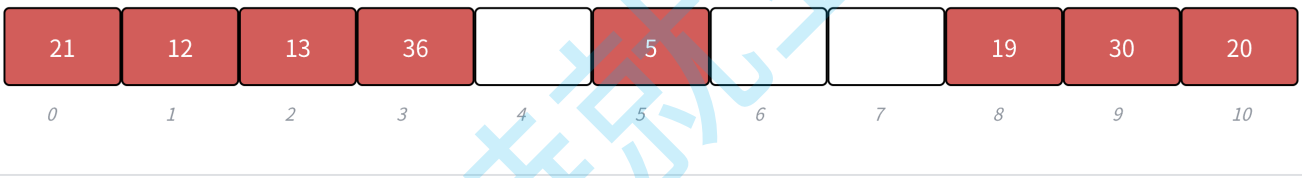
开放定址法的哈希表结构

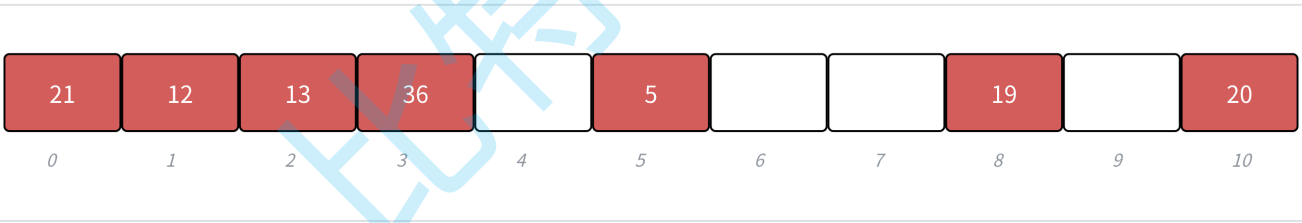
```
enum State
{
    EXIST,
    EMPTY,
    DELETE
};

template<class K, class V>
struct HashData
{
```

```
    pair<K, V> _kv;
    State _state = EMPTY;
};

template<class K, class V>
class HashTable
{
private:
    vector<HashData<K, V>> _tables;
    size_t _n = 0;  // 表中存储数据个数
};
```

要注意的是这里需要给每个存储值的位置加一个状态标识，否则删除一些值以后，会影响后面冲突的值的查找。如下图，我们删除30，会导致查找20失败，当我们给每个位置加一个状态标识`{EXIST,EMPTY,DELETE}`，删除30就可以不用删除值，而是把状态改为`DELETE`，那么查找20时是遇到`EMPTY`才能，就可以找到20。

`h(19) = 8, h(30) = 8, h(5) = 5, h(36) = 3, h(13) = 2, h(20) = 9, h(21) = 10, h(12) = 1`





扩容

这里我们哈希表负载因子控制在0.7，当负载因子到0.7以后我们就需要扩容了，我们还是按照2倍扩容，但是同时我们要保持哈希表大小是一个质数，第一个是质数，2倍后就不是质数了。那么如何解决了，一种方案就是上面1.4.1除法散列中我们讲的Java HashMap的使用2的整数幂，但是计算时不能直接取模的改进方法。另外一种方案是sgi版本的哈希表使用的方法，给了一个近似2倍的质数表，每次去质数表获取扩容后的大小。

```
inline unsigned long __stl_next_prime(unsigned long n)
{
      // Note: assumes long is at least 32 bits.
    static const int __stl_num_primes = 28;
    static const unsigned long __stl_prime_list[__stl_num_primes] =
```

```
    {
      53,         97,         193,       389,       769,
      1543,       3079,       6151,      12289,     24593,
      49157,      98317,      196613,    393241,    786433,
      1572869,    3145739,    6291469,   12582917,  25165843,
      50331653,   100663319,  201326611, 402653189, 805306457,
      1610612741, 3221225473, 4294967291
    };

    const unsigned long* first = __stl_prime_list;
    const unsigned long* last = __stl_prime_list + __stl_num_primes;
    const unsigned long* pos = lower_bound(first, last, n);
    return pos == last ? *(last - 1) : *pos;
}
```

key不能取模的问题

当key是string/Date等类型时，key不能取模，那么我们需要给HashTable增加一个仿函数，这个仿函数支持把key转换成一个可以取模的整形，如果key可以转换为整形并且不容易冲突，那么这个仿函数就用默认参数即可，如果这个Key不能转换为整形，我们就需要自己实现一个仿函数传给这个参数，实现这个仿函数的要求就是尽量key的每值都参与到计算中，让不同的key转换出的整形值不同。string做哈希表的key非常常见，所以我们可以考虑把string特化一下。

```
template<class K>
struct HashFunc
{
    size_t operator()(const K& key)
    {
            return (size_t)key;
    }
};

// 特化
template<>
struct HashFunc<string>
{
    // 字符串转换成整形，可以把字符ascii码相加即可
    // 但是直接相加的话，类似"abcd"和"bcad"这样的字符串计算出是相同的
    // 这里我们使用BKDR哈希的思路，用上次的计算结果去乘以一个质数，这个质数一般去31，131等效果会比较好
    size_t operator()(const string& key)
    {
        size_t hash = 0;
        for (auto e : key)
        {
```

```
                hash *= 131;
                hash += e;
        }

        return hash;
    }
};

template<class K, class V, class Hash = HashFunc<K>>
class HashTable
{
public:
private:
    vector<HashData<K, V>> _tables;
    size_t _n = 0;  // 表中存储数据个数
};
```

完整代码实现

```
namespace open_address
{
    enum State
    {
            EXIST,
            EMPTY,
            DELETE
    };

    template<class K, class V>
    struct HashData
    {
            pair<K, V> _kv;
            State _state = EMPTY;
    };

    template<class K, class V, class Hash = HashFunc<K>>
    class HashTable
    {
    public:
        inline unsigned long __stl_next_prime(unsigned long n)
        {
                // Note: assumes long is at least 32 bits.
                static const int __stl_num_primes = 28;
                static const unsigned long __stl_prime_list[__stl_num_primes] =
                {
```

```
27                  53,         97,         193,       389,       769,
28                  1543,       3079,       6151,      12289,     24593,
29                  49157,      98317,      196613,    393241,    786433,
30                  1572869,    3145739,    6291469,   12582917,  25165843,
31                  50331653,   100663319,  201326611, 402653189, 805306457,
32                  1610612741, 3221225473, 4294967291
33                };
34
35                const unsigned long* first = __stl_prime_list;
36                const unsigned long* last = __stl_prime_list +
   __stl_num_primes;
37                const unsigned long* pos = lower_bound(first, last, n);
38                return pos == last ? *(last - 1) : *pos;
39        }
40
41        HashTable()
42        {
43                _tables.resize(__stl_next_prime(0));
44        }
45
46        bool Insert(const pair<K, V>& kv)
47        {
48                if (Find(kv.first))
49                        return false;
50
51                // 负载因子大于0.7就扩容
52                if (_n * 10 / _tables.size() >= 7)
53                {
54                        // 这里利用类似深拷贝现代写法的思想插入后交换解决
55                        HashTable<K, V, Hash> newHT;
56
   newHT._tables.resize(__stl_next_prime(_tables.size()+1));
57                        for (size_t i = 0; i < _tables.size(); i++)
58                        {
59                                if (_tables[i]._state == EXIST)
60                                {
61                                        newHT.Insert(_tables[i]._kv);
62                                }
63                        }
64
65                        _tables.swap(newHT._tables);
66                }
67
68                Hash hash;
69                size_t hash0 = hash(kv.first) % _tables.size();
70                size_t hashi = hash0;
71                size_t i = 1;
```

```
72                 while (_tables[hashi]._state == EXIST)
73                 {
74                     // 线性探测
75                     hashi = (hash0 + i) % _tables.size();
76                     // 二次探测就变成 +- i^2
77 
78                     ++i;
79                 }
80 
81                 _tables[hashi]._kv = kv;
82                 _tables[hashi]._state = EXIST;
83                 ++_n;
84 
85                 return true;
86         }
87 
88         HashData<K, V>* Find(const K& key)
89         {
90                 Hash hash;
91                 size_t hash0 = hash(key) % _tables.size();
92                 size_t hashi = hash0;
93                 size_t i = 1;
94                 while (_tables[hashi]._state != EMPTY)
95                 {
96                         if (_tables[hashi]._state == EXIST
97                                 && _tables[hashi]._kv.first == key)
98                         {
99                                 return &_tables[hashi];
100                        }
101 
102                        // 线性探测
103                        hashi = (hash0 + i) % _tables.size();
104                        ++i;
105                }
106 
107                return nullptr;
108        }
109 
110        bool Erase(const K& key)
111        {
112                HashData<K, V>* ret = Find(key);
113                if (ret == nullptr)
114                {
115                        return false;
116                }
117                else
118                {
```

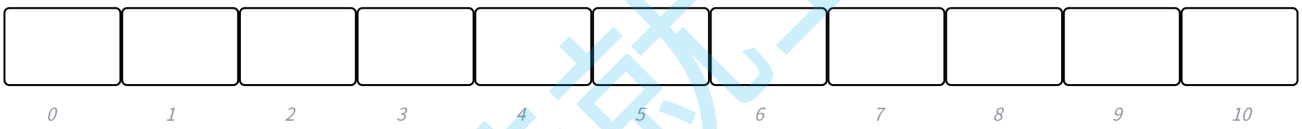
```
119                         ret->_state = DELETE;
120                         --_n;
121                         return true;
122                     }
123             }
124         private:
125             vector<HashData<K, V>> _tables;
126             size_t _n = 0;  // 表中存储数据个数
127         };
128 }
```

1.6.3 链地址法

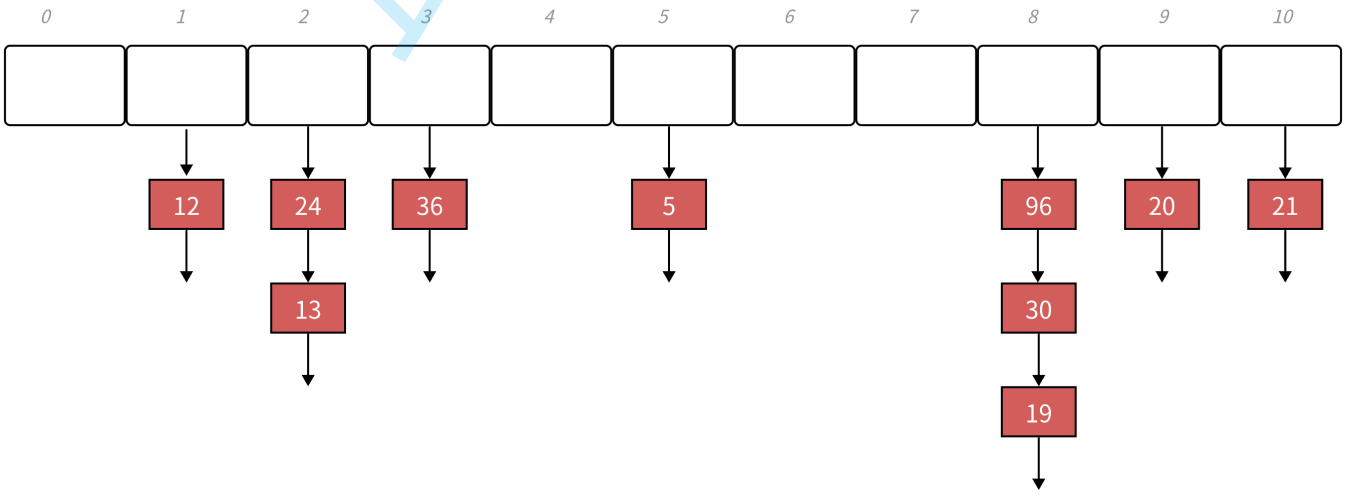
解决冲突的思路

开放定址法中所有的元素都放到哈希表里，链地址法中所有的数据不再直接存储在哈希表中，哈希表中存储一个指针，没有数据映射这个位置时，这个指针为空，有多个数据映射到这个位置时，我们把这些冲突的数据链接成一个链表，挂在哈希表这个位置下面，链地址法也叫做拉链法或者哈希桶。

- 下面演示 `{19,30,5,36,13,20,21,12,24,96}` 等这一组值映射到M=11的表中。



`h(19) = 8, h(30) = 8, h(5) = 5, h(36) = 3, h(13) = 2, h(20) = 9, h(21) = 10, h(12) = 1,h(24) = 2,h(96) = 88`



扩容

开放定址法负载因子必须小于1，链地址法的负载因子就没有限制了，可以大于1。负载因子越大，哈希冲突的概率越高，空间利用率越高；负载因子越小，哈希冲突的概率越低，空间利用率越低；stl中unordered_xxx的最大负载因子基本控制在1，大于1就扩容，我们下面实现也使用这个方式。

极端场景

如果极端场景下，某个桶特别长怎么办？其实我们可以考虑使用全域散列法，这样就不容易被针对了。但是假设不是被针对了，用了全域散列法，但是偶然情况下，某个桶很长，查找效率很低怎么办？这里在Java8的HashMap中当桶的长度超过一定阀值(8)时就把链表转换成红黑树。一般情况下，不断扩容，单个桶很长的场景还是比较少的，下面我们实现就不搞这么复杂了，这个解决极端场景的思路，大家了解一下。

1.6.4 链地址法代码实现

```
namespace hash_bucket
{
    template<class K, class V>
    struct HashNode
    {
        pair<K, V> _kv;
        HashNode<K, V>* _next;

        HashNode(const pair<K, V>& kv)
                :_kv(kv)
                ,_next(nullptr)
        {}
    };

    template<class K, class V, class Hash = HashFunc<K>>
    class HashTable
    {
        typedef HashNode<K, V> Node;

        inline unsigned long __stl_next_prime(unsigned long n)
        {

                static const int __stl_num_primes = 28;
                static const unsigned long __stl_prime_list[__stl_num_primes] =
                {
                  53,         97,         193,       389,       769,
                  1543,       3079,       6151,      12289,     24593,
                  49157,      98317,      196613,    393241,    786433,
                  1572869,    3145739,    6291469,   12582917,  25165843,
                  50331653,   100663319,  201326611, 402653189, 805306457,
                  1610612741, 3221225473, 4294967291
                };
```

```
33
34                  const unsigned long* first = __stl_prime_list;
35                  const unsigned long* last = __stl_prime_list +
    __stl_num_primes;
36                  const unsigned long* pos = lower_bound(first, last, n);
37                  return pos == last ? *(last - 1) : *pos;
38          }
39      public:
40          HashTable()
41          {
42                  _tables.resize(__stl_next_prime(0), nullptr);
43          }
44
45          // 拷贝构造和赋值拷贝需要实现深拷贝，有兴趣的同学可以自行实现
46
47          ~HashTable()
48          {
49                  // 依次把每个桶释放
50                  for (size_t i = 0; i < _tables.size(); i++)
51                  {
52                          Node* cur = _tables[i];
53                          while (cur)
54                          {
55                                  Node* next = cur->_next;
56                                  delete cur;
57                                  cur = next;
58                          }
59                          _tables[i] = nullptr;
60                  }
61          }
62
63          bool Insert(const pair<K, V>& kv)
64          {
65                  Hash hs;
66                  size_t hashi = hs(kv.first) % _tables.size();
67
68                  // 负载因子==1扩容
69                  if (_n == _tables.size())
70                  {
71                          /*HashTable<K, V> newHT;
72
    newHT._tables.resize(__stl_next_prime(_tables.size()+1);
73                          for (size_t i = 0; i < _tables.size(); i++)
74                          {
75                                  Node* cur = _tables[i];
76                                  while(cur)
77                                  {
```

```
78                                      newHT.Insert(cur->_kv);
79                                      cur = cur->_next;
80                                  }
81                              }
82                              _tables.swap(newHT._tables);*/
83
84                              // 这里如果使用上面的方法，扩容时创建新的结点，后面还要使用就结点，浪费了
85                              // 下面的方法，直接移动旧表的结点到新表，效率更好
86                              vector<Node*> newtables(__stl_next_prime(_tables.size()+1), nullptr);
87                              for (size_t i = 0; i < _tables.size(); i++)
88                              {
89                                      Node* cur = _tables[i];
90                                      while (cur)
91                                      {
92                                              Node* next = cur->_next;
93
94                                              // 旧表中节点，挪动新表重新映射的位置
95                                              size_t hashi = hs(cur->_kv.first) % newtables.size();
96                                              // 头插到新表
97                                              cur->_next = newtables[hashi];
98                                              newtables[hashi] = cur;
99
100                                             cur = next;
101                                     }
102
103                                     _tables[i] = nullptr;
104                             }
105
106                             _tables.swap(newtables);
107                     }
108
109                     // 头插
110                     Node* newnode = new Node(kv);
111                     newnode->_next = _tables[hashi];
112                     _tables[hashi] = newnode;
113                     ++_n;
114
115                     return true;
116             }
117
118             Node* Find(const K& key)
119             {
120                     Hash hs;
121                     size_t hashi = hs(key) % _tables.size();
```

```
122                 Node* cur = _tables[hashi];
123                 while (cur)
124                 {
125                         if (cur->_kv.first == key)
126                         {
127                                 return cur;
128                         }
129
130                         cur = cur->_next;
131                 }
132
133                 return nullptr;
134         }
135
136         bool Erase(const K& key)
137         {
138                 Hash hs;
139                 size_t hashi = hs(key) % _tables.size();
140                 Node* prev = nullptr;
141                 Node* cur = _tables[hashi];
142                 while (cur)
143                 {
144                         if (cur->_kv.first == key)
145                         {
146                                 if (prev == nullptr)
147                                 {
148                                         _tables[hashi] = cur->_next;
149                                 }
150                                 else
151                                 {
152                                         prev->_next = cur->_next;
153                                 }
154
155                                 delete cur;
156                                 --_n;
157                                 return true;
158                         }
159
160                         prev = cur;
161                         cur = cur->_next;
162                 }
163
164                 return false;
165         }
166
167     private:
168         vector<Node*> _tables;  // 指针数组
```

```
169         size_t _n = 0;          // 表中存储数据个数
170     };
171 }
```