

Chapter 13 SEARCHING AND THE HASH CLASSES

回顾：红黑树查找

- 无论查找成功还是失败，最坏和平均时间复杂度都和n成对数关系

hash_map类

- hash_map类与map类相似，但前者使用的方法更少，一般为insert, erase, find（平均时间复杂度都为常数），且
- map中的value有两个部分，即每个值都是一个pair对，第一个部分为key，第二个部分为type T，key唯一，不存在两个不同的value有相同的key（key通过operator==比较）
- 哈希处理的含义：把树的值的空间映射到地址空间
- 字段
 - buckets 桶：用于放数据 目的在值和地址间直接建立关系
 - count 元素个数

buckets			count
0	?	?	3
	...		
108	108	Barrett	
	...		
351	351	Prashant	
	...		
435	435	Lin	
	...		
999			

- 上图中左列为index，buckets中的左列为key，右列为value的内容

- collision 冲突：两个不同的key有相同的index下标地址
- synonyms 同义词：冲突的key

hashing 散列

- hash function 哈希函数：通过key计算操作后返回（尽可能不同的）unsigned long，然后将此长整数转化成数组buckets里的一个index地址（模除总长度之后得到地址——此时可能会有地址冲突情况发生）
- collision handler 冲突处理机制（特有）
- ※函数类 operator()

Here is the start of the hash_map class:

```
template<class Key, class T, class HashFunc>
class hash_map
{
```

The third template parameter is a function class: a class in which the function-call operator, operator(), is overloaded.

The heading for operator() is
unsigned long operator() (const key_type& key)

For example, we can define a simple function class if each key is an int:

```
class hash_func
{
    public:

        unsigned long operator( ) (const int& key)
        {
            return (unsigned long)key;
        } // overloaded operator( )
} // class hash_func
```

注意：hash_map中的key不会排序，只用于查找，顺序随机，但map中的key是有序的。

```
extensions [5520] = "Yvonne";
extensions [5415] = "Jim";
extensions [5416] = "Penny";
extensions [5537] = "Chun Wai";
extensions [5273] = "Jim";

for (itr = extensions.begin(); itr != extensions.end(); itr++)
    cout << (*itr).first << " " << (*itr).second << endl;
```

HERE IS THE OUTPUT:

```
5520 Yvonne
5537 Chun Wai
5415 Jim
5416 Penny
5273 Jim
```

hash_func 哈希函数

- 设计哈希函数时通过乘以素数保持分散，有助于降低哈希冲突的可能性。
 - BIG_PRIME 是一个大素数，可用作哈希函数中的乘法因子
- 通过将长整数模除以bucket的长度得到operator()返回的地址索引值

```
int index = hash_func (key) % length;
```

chaining (chained hashing) 链式哈希

- 映射到相同index地址的value链成链表
- 最坏情况：全部value都对应一个index 整个就是链表 最坏时间复杂度即为O(n)

- 平均情况（不扩容）： $O(n/1000) = O(n)$
- 平均情况（扩容）： $O(n/m) = \text{常数}$ ($n \leq m$)
 - load factor 占用率：超过75%需要扩容
 - 扩容时将原容量 $\times 2 + 1$

double hashing 开放地址 - 双哈希

- 避免链表，设置开放地址哈希
- 每个index至多放置一个value，重复了就不不断往下找位置，即偏移量
- 冲突处理机制：查找表直到找到buckets中有open标记的位置
- 在value_type类中增加字段：bool occupied
 - 增加字段：bool marked_for_removal

	key	occupied	marked_for_removal	
0	?	false	false	
	...	false	false	
54	1069	true	true	$1069 \% 203 = 54$
55	460	true	false	$460 \% 203 = 54$
56	1070	true	false	$1070 \% 203 = 55$
109	312	true	false	$312 \% 203 = 109$
201	607	true	false	$607 \% 203 = 201$
202		false	false	

- primary clustering：由于冲突导致冲突聚集 最坏情况为每个元素都不在自己的位置上
- 解决由于primary clustering找不到位置放的问题——double hashing

Solution: **double hashing, that is, obtain both indices and Offsets by hashing:**

```
unsigned long hash_int = hash (key);  
int index = hash_int % length,  
offset = hash_int / length;
```

Now the offset depends on the key, so different keys will usually have different offsets, so no more primary clustering!

TO GET A NEW INDEX IF THERE IS COLLISION:

```
index = (index + offset) % length;
```

- 若offset是length的倍数，则将此时的offset设置为1
- 尽可能将length设置为素数，因为当length为多因子数时，可能导致数的存放在特定几个固定位置跳

chained hashing和double hashing的比较

-
- 从比较次数讲 链式哈希更高效
- 但从整体讲 链式哈希更复杂