第6章 哈希表

这里主要介绍哈希表的原理，工作方式，但是仅仅是给个简单的示例，涉及的内容大概是
本章内容的1/5，主要就是讲明白工作原理，会调用库刷题即可，目标并不是自己写一个完整的哈希表。

In [1]:
```
#include <iostream>
#include <string>
#include <cstring>
using namespace std;
```

词条

就是保存数据的一个个节点，data就是key和value，并且支持比较(比较的是key)

In [2]:
```
template <class K, class V>
struct Entry{
    K key;
    V value;

    Entry (K k, V v): key(k), value(v) {}  // 构造函数
    bool operator< (const Entry<K,V>& e){return key < e.key;}
    bool operator> (const Entry<K,V>& e){return key > e.key;}
    bool operator= (const Entry<K,V>& e){return key = e.key;}
    bool operator!= (const Entry<K,V>& e){return key != e.key;}
};
```

In [3]:
```
Entry<int,string> x(2021111436,"zhangyouwei");
Entry<int,string> y(2021111437,"zhangsan");
```

In [4]:
```
x < y
```

Out[4]:
```
true
```

词典

就是key-value这种形式的数据，通过key快速访问数据，必须支持以下4个接口：
返回词条总数，插入词条，删除词条，读取词条,注意虚方法后面一定要加 = 0

In [5]:
```
template <class K, class V>
struct Dictionary{
public:
    virtual int size() = 0;
    virtual bool put(K k, V v) = 0;
    virtual bool remove(K k) = 0;
    virtual V& get(K k) = 0;
};
```

哈希表

对于上述词典，实际实现的数据结构有红黑树，向量等，我们这里主要通过向量实现，就是哈希表
注意，c++里面struct跟类几乎一样，也可以继承过来

In [6]:
```
template <class K, class V>
class Hashtable: public Dictionary<K,V>{
    public:
        Entry<K,V>** table;    // 存储向量
        int M;                 // 向量大小
        int N;                 // 词典大小

        Hashtable(int c=87);  // 构造函数
        int hashcode(K k);
        int size(){return N;}
        bool put(K k,V v);
        bool remove(K k);
        V& get(K k);
};
```

构造函数
这里就简单随便用一个素数87初始化一个向量，不搞那么复杂了

In [7]:
```
template <class K, class V>
Hashtable<K,V>::Hashtable(int c):M(87),N(0){
    table = new Entry<K,V>*[M];
```

```
    memset(table,0,sizeof(Entry<K,V>*)*M);
}
```

哈希映射

就是把key转换为一个数字，作为向量的下标，对于字符串，比如把所有字符的ascii编码相加，
然后对于数字，我们这里简单模M

In [8]:
```
template <class K, class V>
int Hashtable<K,V>::hashcode(K k){
    return k%M;
}
```

按key访问元素值

In [9]:
```
template <class K, class V>
V& Hashtable<K,V>::get(K k){
    return table[hashcode(k)] -> value;
}
```

插入元素

这里不考虑哈希冲突，简单写一写，返回值暂时不用理会，统一返回插入成功

In [10]:
```
template <class K, class V>
bool Hashtable<K,V>::put(K k,V v){
    table[hashcode(k)] = new Entry<K,V>(k,v);
    N++;
    return true;
}
```

删除元素

In [11]:
```
template <class K, class V>
bool Hashtable<K,V>::remove(K k){
    auto temp = table[hashcode(k)];
    table[hashcode(k)] = NULL;
    delete temp;
    N--;
    return true;
}
```

一个简单例子

In [12]:
```
auto dict1 = Hashtable<int,string>();
```

In [13]:
```
dict1.put(2021111436,"zhangyouwei")
```

Out[13]:
```
true
```

In [14]:
```
dict1.put(2021111437,"helloworld")
```

Out[14]:
```
true
```

In [15]:
```
dict1.get(2021111436)
```

Out[15]:
```
"zhangyouwei"
```

In [16]:
```
dict1.get(2021111437)
```

Out[16]:
```
"helloworld"
```

哈希表的工作原理大概就是这么一回事，就是把key转换为向量的下标，然后按下标访问，
这样就能在O(1)的时间访问，插入删除也只是在相应的位置涂改，不需要搬运。
自然而然你就发出疑问，hashcode这个函数，是不是有可能把两个不同的key映射成相同
的下标，比如上面我是把key模M=87，那么如果key是87的倍数，hashcode(key)都为0，
这不就冲突了，对的，这就是哈希冲突，构造一个hashcode函数降低哈希冲突，以及应对
哈希冲突是真正做一个哈希表的重点，不过我们这里仅仅是了解这种数据结构的其工作方式，
后面直接利用库模板解题，暂时突击没时间了解那么详细。实际上也不难，就是看了记不住，暂时不用管。

In [ ]:
```
```