缓存模块要求：（1）基于内存实现，读写复杂度接近为1（2）c++封装（3）cache碰撞率非常低，设计存储的数据达到100万条以上（4）提供set,get,del接口（5）写出测试程序，测试缓存模块。写出说明文档

使用环境：IDE:CodeBlocks ；Compiler:MinGW gcc

P.s：使用了部分C++11语法，请**打开编译器选项 –std=c++11**。

本缓存模块名为LRUCache，意为Last Recently Used Cache.

根据局部性原理，在前面几条指令中使用频繁的页面很可能在后面的几条指令中频繁使用。反过来说，已经很久没有使用的页面很可能在未来较长的一段时间内不会被用到。

因此在缓存模块中，我们是用的数据结构是list + unordered\_map

双向链表cache表示最近使用的数据，cache的结点值为键值和存储的数据。

哈希表hash表示是否有数据存在于缓存中,hash的键值为数据的键值，元素值为指向cache的指针。

在构造LRUCache时，我们需要指定一个capacity作为缓存的容量。

我们提供了set,get,del接口，分别如下：

string get(int key);

void del(int key);

void set(int key, string value);

此外提供了show，用于暴露出list的内部元素，方便我们进行测试。

void show();

接口的具体实现如下“

void LRUCache::set(int key, string value) {

auto it = hash->find(key);

if(it != hash->end()){

putFirst(it->second);

cache->begin()->second = value;

}

else{

if(capacity == cache->size()){

hash->erase(cache->back().first);

cache->pop\_back();

}

cache->push\_front({key,value});

}

(\*hash)[key] = cache->begin();

}

在向cache插入数据时，需要指定一个键值key,元素值value。如果key已经存在于cache中，则将其放在cache的头部，并更新元素值为value。若不存在，则判断当前cache是否已满，如果过已满，则更新hash,并删除cache中最后一个元素，并构造以当前插入的结点，将其放在cache头部。

string LRUCache::get(int key) {

auto it = hash->find(key);

if(it == hash->end()) return "can't find";

putFirst(it->second);

(\*hash)[key] = cache->begin();

return it->second->second;

}

在获取数据时，如果在hash中未找到数据，则直接返回”can’t find”。如果找到则将其放到cache头部，并更新hash中key对应的指针。然后返回数据内容。

void LRUCache::del(int key){

auto it = hash->find(key);

if(it != hash->end()){

cache->erase(it->second);

hash->erase(it);

}

else cout << "del error : key isn't exsit" << endl;

}

在删除数据是 ，需要指定键值key。分别从hash和cache中删除数据。若不存在则输出” del error : key isn't exsit”。

void LRUCache::show(){

cout << "\*\*\*\*\*\*\*\*\*\*\*LRUCache::show\*\*\*\*\*\*\*\*\*\*" << endl;

for(auto i = cache->begin();i != cache->end();++ i){

cout << "key: " << i->first << " value: " << i->second << endl;

}

}

按照cache中由头至尾的顺序输出cache中的数据。

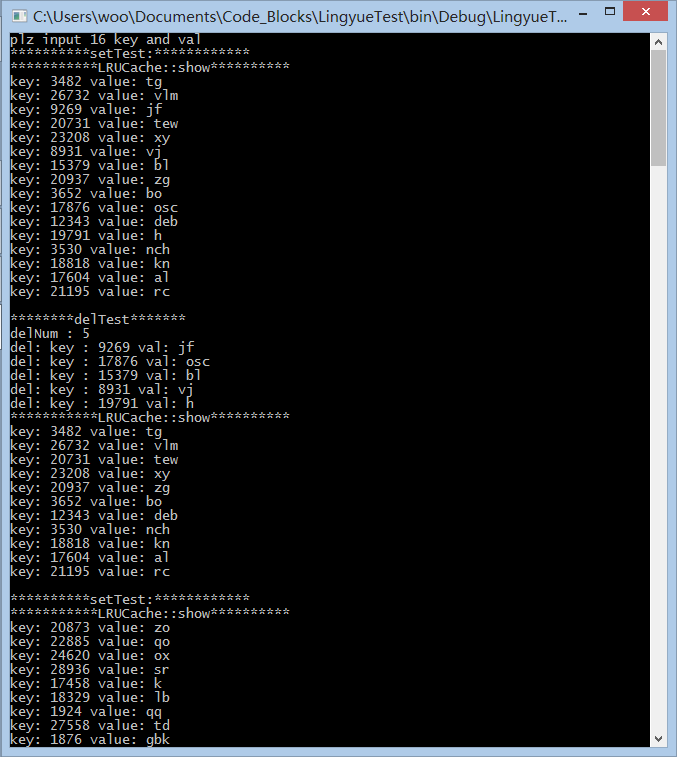
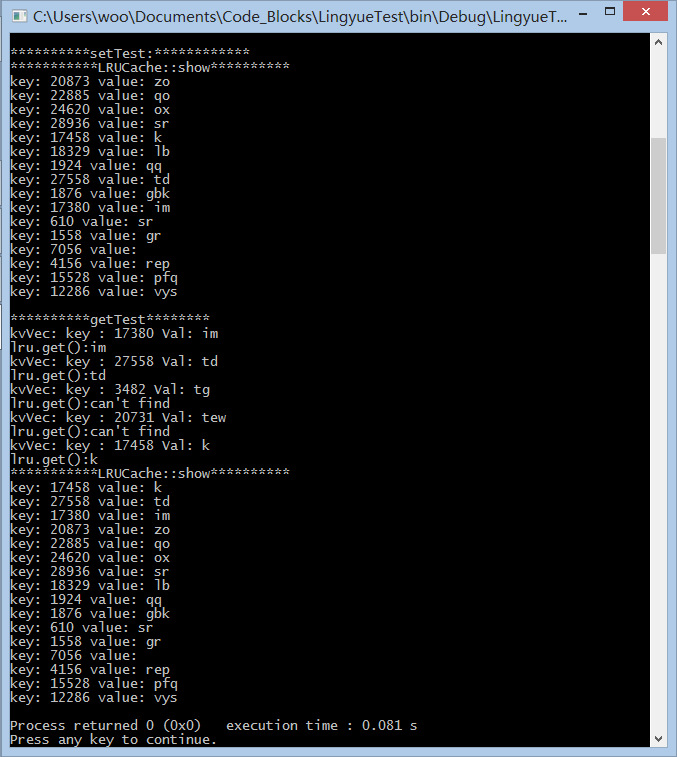
测试部分：

为方便测试，我们使用了四个函数，分别为setTest,getTest,delTest以及辅助函数getRandStr。

分别用来测试set,get,del接口。以及随机构造字符串（即数据内容）。

同时我们设置了一个vector<pair<int,string>> kvVec 作为我们的辅助器，可以把它抽象为我们的硬盘空间。

为了减少工作量，我们采用半自动化测试方法。数据的构造set,get,del均由函数自动完成。程序员需要做的就是根据打印出的数据进行判定是否合乎cache的工作原理。下面是一组测试数据，我们将cache的capacity指定为不超多20的正整数。

 在main函数中，我们依次进行了setTest, delTest, setTest, getTest。因为是在进行了两次setTest所以在我们的“硬盘“kvVec中，存在一些（至多为一半）已经不在cache中的数据。之后我们进行getTest，就会出现”can’t find”的情况。对于get得到的元素我们可以在下面的LRU::show中看到，cache中的元素排列与getTest的顺序是相反的。这就说明了我们的cache模块设计无误。

题目要求数据量在100W条以上，因为我们的数据都存储在堆上，所以对于一台有4G内存的机器来说，4 \* (10 ^ 9) / 100 \* (10 ^ 4) == 4000 bytes。即使当数据量增加到1000W条，对于每一条数据，我们仍有400bytes的大小可以用来存放单条数据。

缺陷：因为我们使用了unordered\_map和 list来作为我们的数据结构，这样会带来额外的空间开销，所以达到理想情况比较困难，一般我们认为哈希会占用实际数据大小4倍的空间。所以最坏情况下，单挑数据所能占有的空间可能只有上述情况的1 / 4.

但是考虑实际情况中，键值可能会有各种特性，比如连续。这时我们就可以采用其他数据结构，用来降低空间复杂度。

以上。