LinkedHashMap实现及在LRU算法中的应用

蒋捷

2024 年 3 月 4 日

摘要

作业内容：提供HashMap的接口，LinkedHashMap的接口
和LRU(Least Recently Used)的函数接口，要求实现HashMap的功能，
LinkedHashMap的功能并以此为基础实现LRU算法。

作业限制：不可使用以下头文件及类似功能头文件

```
#include<unordered_map>
#include<map>
```

截止时间：3月24日

数据点得分：30% hashmap + 40% linked_hashmap + 30% lru

Tip：有什么问题欢迎问我。

目录

1 概念介绍

1.1 HashMap

LinkedHashMap在本质上是一个实现了按照插入顺序访问元素的HashMap(哈希表)，所以在介绍 LinkedHashMap之前，我们需要先了解HashMap。

1.1.1 哈希表相关概念

哈希表可以只存储一些数据，检索数据是否存在，也可以存储键值对，通过键来索引到所需要的数据。这两种方式本质上是一样的，存储键值对本质也是存储一些数据(如pair)，在进行哈希的时候只对键进行哈希，把整个键值对放入哈希到的地方，就可以通过键来确定它对应的键。

哈希表存储的数据可以允许重复也可以要求不重复。本次作业存储的是键值对，要求同一个的键最多只能对应一个值，新值会覆盖旧值。(所有要求会在后文汇总)

哈希值：就是把任意长度的输入，通过某种哈希算法，变换某种与之对应的输出 (通常是整数，用于定位)。

哈希碰撞：不同的输入可能会散列成相同的输出，从而不可能从散列值来唯一的确定输入值。

(在本次作业中，推荐使用 std::hash 来实现哈希函数)

HashMap 支持关键词对应元素的插入、查询和删除，并且这些操作的平均时间复杂度都为$O(1)$，只会在最差情况下退化为$O(n)$。

解决哈希碰撞有多种方法，如线性勘测法，开链表法等。

1.1.2 固定大小哈希表

有了哈希函数，我们最自然的想法就是构建一个长度为 l 的数组，数组的下标对应着哈希值。每当我们插入一对键值对 (K, V) 的时候，我们先通过哈希函数对 K 进行处理得到哈希值 h，然后将 v储存到数组的第 h 位。这就实现了一个固定大小的哈希表。

1.1.3 动态大小哈希表

我们需要实现的是一个封装好的数据结构供其他人（或自己）使用（如应用于lru），我们在开发的时候实际上并不知道数据规模的大小，所以哈希表大小的选择是一个非常关键的问题。当我们选择的哈希表大小比较小的时候，哈希碰撞概率会变高，查询复杂度会退化；当哈希表大小比较大的时候，会占用很多无用空间。

我们可以动态的改变哈希表的大小。

首先引入两个参数C, f，分别是容量(Capacity)和负载因子(LoadFactor)，代表着哈希表的大小，和对于某个特定的容量，我们所能接受的最多元素个数占容量的比例。我们一开始可以选择一个比较小（不建议太大）的容量，当元素个数大于$C * f$ 时，我们再增大C，使得我们的数据结构能够保持良好效率的同时，不占用过多空间，具体参数大家可以根据自己实现的数据结构去进行调整。

1.2 LinkedHashMap

LinkedHashMap需要在实现HashMap功能的基础上，再进行插入顺序的维护，使得我们可以按照插入顺序来访问元素。这项功能的实现比较简单，只需要维护一个双向链表，每次添加的元素除了插入 HashMap外，也需要插入到链表的末尾，这样我们按照链表顺序访问元素就可以实现这项功能。

但是LinkedHashMap的查找分两种，一种是返回Key对应的Value，一种是返回指向Key的迭代器。对于插入元素对，查询Key对应的Value，按照插入顺序访问插入的元素这三种操作，要求期望的时间复杂度为O(1)

1.3 LRU

1.3.1 LRU相关概念

LRU(Least Recently Used)即最近最少使用算法，是一种内存数据淘汰策略，使用常见场景是当内存不足时，需要淘汰最近最少使用(被插入或者被查询)的数据。（注：内存可以理解为一个有限长度的数组，正如数组由许多元素组成，内存也包含很多内存空间。）

因此，该算法需要一个参数 n 表示预设的内存的大小（允许存储的键值对的个数）。

1.3.2 LRU执行的操作

- **插入(save)**

  可以理解为把一个键值对(K, V) 放入内存。首先查找有无内存空间的键为K

  - 有

    更新节点的值

  - 无

    检查有无未被使用的内存空间

* 有
  将该内存空间标记为被该K 使用并存入V
* 无
  查找最早**被插入或者被查询**的键值对(K', V')，将该键值对替换为(K, V) (原键、值都不再存在)

- **查询(get)**

  利用Hash查找内存中有无某个内存空间该键K

  – 没找到，返回空指针。
  – 找到，返回一个指向该对象的制作

实现关键在于利用LinkedHashMap维护最早**被插入或者被查询**的键值对，即LRU。

2 代码实现

2.1 文件说明

tip:

2.1.1 编码

文件采用UTF-8编码，如果不幸添加了中文注释并且使用dev-c++打开，则会出现乱码.
一个重新编码的方式是：右击文件-打开方式-用记事本打开-另存为-
选择编码为GB18030（如果是win11,win10不清楚是不是，GB开头一般没问题）-确定，然后用dev打开新文件即可正常显示中文

2.1.2 逐个介绍

1. class-integer.hpp（无需修改）

   整数类Integer，静态变量counter记录变量构造/析构的次数。程序运行结束时应该为0

   对象构造和输出示例如下：

```
Integer a = Integer(1);
Integer *p = new Integer(2);
std::cout<<a.val<<"␣"<<(p->val)<<std::endl;
```

2. class-matrix.hpp（无需修改）

   矩阵类，对象构造示例如下：

```
Matrix<int> a = Matrix<int>(1,2,3);
Matrix<int> *p = new Matrix<int>(1,2,3);
std::cout<<a<<"␣"<<*p<<std::endl;
```

   构造1*2，填充的数字都为3的矩阵

3. exceptions.hpp（无需修改）

   用于debug，也可不用。正常的throw应该够用了。

   throw 字符串用法示例

```
try{
    throw "have␣a␣try";
}catch(const char* c){
    std::cout<<c<<std::endl;
}
```

   以上代码会输出 have a try 并且在throw后立即终止程序。

```
throw "have␣a␣try";
```

   以上代码会出现terminate called after throwing an instance of 'char const*' 并且程序会停顿几秒后才终止。

4. lru.hpp(todo)

   包含

```
sjtu::double_list<T>
sjtu::hashmap<Key,T,Hash,Equal>
sjtu::linked_hashmap<Key,T,Hash,Equal>
//derived from sjtu::hashmap<Key,T,Hash,Equal>
sjtu::lru
```

   linked_hashmap是派生类

   模板类的派生类在调用基类函数时，需要指定this

   double_list包含实现linked_hashmap所需要的双向链表的接口。

自行实现链表，你可以自己新定义一些类来辅助实现。

lru没有默认构造函数，构造时必须给定size参数。

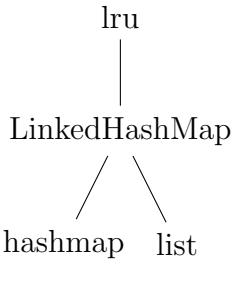
5. utility.hpp（无需修改）

   包含用到的pair类

   用法示例

```
sjtu::pair<Integer,Matrix<int> >
a(Integer(1),Matrix<int>(2,2,2));
sjtu::pair<Integer,Matrix<int> > p
=new sjtu::pair<Integer,Matrix<int> >;
```

2.2 类结构



(hashmap或许也可以用到list)

2.3 代码补充细节

1. template里Hash = ...

   解释：有等于。。。说明这是它默认的类型，如果在使用模板时不添加这个参数，则Hash这个变量类型(类似于int)就取等号后面的，如果使用时加了参数就用使用时的。

   之所以如此，是因为对于一般的int，默认的就可以处理了

   如果是Integer这种自己定义的类，std::hash不知道怎么处理，需要告诉std::hash该怎么做

   以下代码就是Key为Integer的时候，Hash类的定义。

```
class Hash {
    public:
        unsigned int operator () (Integer lhs) const {
            int val = lhs.val;
```

```
            return std::hash<int>()(val);
        }
    };
```

（本质上还是对int做std::hash）

Equal 同理。

2. Hash(equal)类怎么用

   Hash是一个类型，利用这个类型我们可以创造一个变量。

   这个变量可以像函数一样接受一个参数（其实是重载了括号运算符）

   使用示例代码如下

```
    Hash h;
    int v1= h(key);
    int v = Hash()(key);
```

   h的用法和函数一样

   Hash()是创建了一个临时对象，效果和上面一样

3. 关于迭代器指向空对象的解引用

   （以下两段内容来自ChatGPT）解引用空指针或无效指针会导致未定义行为，因此，在实际编程中，应该始终确保在对迭代器进行解引用之前，检查迭代器是否有效。

   在C++的STL（标准模板库）中，对于迭代器解引用时指向容器末尾的情况，、通常不会提供特定的保护或处理方式。这是因为STL通常遵循零成本抽象（zero-overhead abstraction）的原则，不会为不常见或异常情况提供额外的开销或处理逻辑。

   所以我们要求对于解引用空对象要throw，而在使用unordered _map的时候遇到这种情况会无事发生。

4. 关于提供的测试代码

   复制到代码根目录下编译运行即可，但不代表OJ测试结果，仅为OJ测试的子集。

5. 关于LinkedHashMap和LRU的不同

   LinkedHashMap要求实现按照插入顺序访问，所以只有在插入的时候会更改元素的“顺序”。

   而LRU在插入和查询的时候都会更改元素的顺序。

2.3.1 关于throw

要求throw，但不要求一定使用exceptions.hpp里面的类随便throw什么东西都可以
使用exceptions.hpp的好处是throw
之后不try也可以根据类型找到在哪出错，
而如果throw 字符串，需要try catch才能知道哪里出错
没有坏处。
以下是需要throw的地方的汇总

1. 如果迭代器移动到了非法位置(如begin()–,end()++)
2. 如果对指向空对象的迭代器解引用(如 *end())
3. 如果在使用[]或者at时下标越界/不存在

2.3.2 杂项

1. linked_hashmap由于是继承的hashmap，本身就有一个存储value_pair的hashmap ，在实现自身函数时可以调用基类的函数对已有的hashmap操作。基本上只需要额外维护子类的双向链表即可。
2. list里只有迭代器，没给常迭代器接口，而且也不会对list进行测试，这意味着你哪怕把list里的东西都删掉对测试也不产生影响，只要你觉得这样对你更方便。

   同时这使得如果你想要让LinkedHashMap的迭代器也采用继承（不强制要求但可行，无加分），要么就只能继承一般迭代器，要么自行完成list的常迭代器。
3. 为了统一测试格式，在提交时请让sjtu::lru::print()的形式如下（feel free to use）

```
void print(){
    sjtu::linked_hashmap<Integer,Matrix<int>,
    Hash,Equal>::iterator it;
    for(i = mem->begin(); i!= mem->end(); i++){
        std::cout<<(*i).first.val<<"␣"
        <<(*i).second<<std::endl;
    }
}
```

(剧透:hashmap底层可以使用一些复杂数据结构，可能是未来的大作业)