1 布谷鸟哈希程序设计

参考了课本的代码，并且在其基础上进行了简化。首先，概述这里的布谷鸟哈希 insert 实现流程（与上课讲的双表型有所不同）：

1. 初始化一个哈希表，长度可以设置（默认大小 101），其中所有位置都是空的。备用哈希函数的个数可以设置（默认 3 种）
2. 当用 insert 方法插入元素时，会根据元素的类型去调用对应的哈希函数（我这里有整型和字符串两种），根据计算得到的位置尝试存入
3. 如果该位置已经有元素占用了，使用下一个备选哈希函数……若存在一个备用哈希函数，使得该元素经其映射得到的位置是空的，则将该哈希函数填入之，返回。否则，转入下一步
4. 踢除操作：任选一个备选哈希函数，该元素强行进入经其映射到的那个位置里，并把原来的老元素踢出来，再对被踢出来的老元素使用 insert 方法，如此循环。
5. 如果发生踢除操作的次数大于阈值，则认为表太小，调用 rehash 方法扩增，更换新的哈希函数，再把表的大小翻为原大小的两倍以上。然后，把原来的元素（包括刚才填不进去的那个元素），全部 insert 一遍到新表里

代码说明：

1. 一共就只有一张表（这样的实现一般来说会比双表更省空间）

2. 与课本代码不同，我简化了 rehash 的判别，如果在某一次 insert 之中，造成的碰撞次数超过阈值（阈值我设为当前元素总数的四分之一，和 100 取 min 的值）就 rehash（无需像课本代码有个内置的变量去记录累计量）

3. 与课本代码不同，调整了哈希函数的生成规则

4. 增加了打印哈希函数、打印哈希表的 public 函数，便于可视化结果与 debug

2 测试思路

基本功能正确与否：测试两种类型（字符串和整型）分别作为基本元素时，哈希表是否工作正常，首先是两个含有可视化的函数，会显示出此次的哈希函数是哪些，也会在每一步都打印出此时的表的大小、此时存有的元素的个数，也会打出整张表。此外，也会在触发布谷鸟踢除时，显示出 “kick” 了哪个元素，在触发 rehash 时会打印 “rehash”，方便检查。也测试了 remove 函数以及 contain 函数的正确实现。

效率考察：

在测试时使用 chrono 进行计时，执行前后的时间作差，使用 print_result 函数打印出时间，包括 check 的结果即可。

注意，这里的计时统一用**建堆后开始到排序结束**的时间。

3 测试效果

可直接使用 make run 命令编译且运行。

输出形如：

```
**Random Input**
sort_heap: 0.157933
my_sort: 0.15796
check result: 1
**Increasing Input**
```

```
 sort_heap: 0.063219
 my_sort: 0.0688413
 check result: 1
**Decreasing Input**
 sort_heap: 0.0810255
 my_sort: 0.0808056
 check result: 1
**Random with Repeat Input**
 sort_heap: 0.152754
 my_sort: 0.157996
 check result: 1
```

在未加入-O2 优化时，如下（单位：second）：

表 1: 运行时间对比

输入序列	**my_sort time**	**std::sort_heap time**
随机	0.733434	1.02845
递增	0.53948	0.851013
递减	0.542546	0.865929
随机（含重复）	0.716736	1.0294

在编译时加入-O2 进行优化后，如下（单位：second）：

表 2: 运行时间对比 (-O2)

输入序列	**my_sort time**	**std::sort_heap time**
随机	0.1622	0.161694
递增	0.076275	0.0677776
递减	0.0800664	0.0843663
随机（含重复）	0.160439	0.156313

用 valgrind –leak-check=full ./test 进行测试，发现没有发生内存泄露

4 时间复杂度分析

由于每一次弹出最大元是 $\Theta(1)$ 的，而耗时的主要是“向下过滤”的过程，由于最糟糕的情况是向下过滤到底，即树高 h 那么多。而共操作 N 次，因此为 $\Theta(N \log N)$

实际上这和标准库的方法一样，因此时间复杂度一致。但，由于我写的 my_sort 函数要求输入必须**已经是个堆了，不会进行检查**，而 std::sort_heap 函数则会首先进行检查，如果不是堆会先建堆。这导致在“检查阶段”耗时，所以可见在表格 1无 O2 优化时，我的函数在各种情况下都更快 0.3 秒左右，有理由猜测是检查的耗时。