实验报告

问题描述

（个人身份证号）请完成一个基于身份证号码的个人信息系统，核心功能包括以下两个。

- 一次性插入N条个人信息记录。
- 从整个信息库中根据身份证号码查询个人信息。

个人信息的数据结构如程序示例1.10所示，其中字符串id为我国居民身份证号码，长度为18个字符，包括：6个数字字符的地址码，8个数字字符的出生日期（次序为年月日），3个数字字符的顺序码（奇数分配给男性，偶数分配给女性）和1个数字字符的校验码。

个人信息系统对外提供两个功能函数，如程序示例1.11所示。其中 `person_insert()` 函数用于插入N条个人信息，内容存储在p指针指向的个人信息数据结构中； `person_search()` 用于查找字符串类型的id对应的个人信息，如果查找到，则返回对应的个人信息数据结构指针，否则返回NULL指针。

C++STL的map方法可以建立（key，value）的映射关系，使用key快速查找到对应的value。以map方法为核心，采用下述不同的方案实现上述两个功能函数，并评估不同方案的性能提升，从而更加深刻地理解map方法的性能特征。

实验所使用的软硬件平台参数如表1-6所示。

CPU	**Intel Xeon E5-2666 v3**	**架构**	**x86_64**
主频	3.30/2.60 GHz	核心数	10c20t
操作系统	Ubuntu 22.04.4 LTS	编译器	gcc 12.3.0
三级缓存	25 MB	内存频率	1600

▲ 表1-6 实验用软硬件平台参数

方案一

key为身份证号码（字符串类型），value为个人信息（person数据结构）。

方案二

在数量N小于10^9时，可以将所有的个人信息存放在一个数组中。此时，value不再存放整个数据结构的内容，而是仅存放这个数据结构在整个数组中的索引（32位无符号整数），因此value的数据容量从112个字节减小到4个字节。

方案三

在实现方案2的基础上，可以根据身份证号码的构成规则将含18个字符的字符串id转换为64位整数。此时，key的数据类型从字符串类型转化为64位整数。

方案四

在实现方案3的基础上，可以提取身份证号码的日期（共366种可能），从而形成366个映射。这样能减少映射中元素的数量，并有可能使得key缩减为32位无符号整数。

请实现上述四种方案，并按照1.4节的格式撰写完整的测试报告。

```
struct person {
    char id[18];         // 18位身份证号码
    char name[20];       // 姓名
    char address[60];    // 地址
    char phone_num[14]; // 电话号码
};
```

▲ 程序示例1.10 实验题1.2的个人信息数据结构

```
void person_insert(struct person *p, int N);
struct person *person_search(const char *id);
```

▲ 程序示例1.11 实验题1.2的对外函数接口

实验方法

实现以上四种方案，并测量两个时间：

1. 连续插入N条记录所需要的时间T。
2. 连续进行M次查找所需要的时间T。

再由此计算出两个指标。

1. 每秒钟插入的记录数 $B = N / T$（N的取值分别为10^6、10^7、10^8、10^9）。
2. 每秒钟查找的记录数 $B = M / T$（M的取值为10^6）。

实验结果分析

根据四种优化方案在四个数量级的数据下，我们得到下列输出（其中10^9的数据量太大，内存存不下了，故无数据）。

```
Test for 10^6 records
Insert time for version 1: 2502ms
Search time for version 1: 2329ms
Insert time for version 2: 3752ms
Search time for version 2: 2264ms
Insert time for version 3: 1728ms
Search time for version 3: 1039ms
Insert time for version 4: 1011ms
Search time for version 4: 2704ms
Test for 10^7 records
Insert time for version 1: 36313ms
Search time for version 1: 3278ms
```

```
Insert time for version 2: 55478ms
Search time for version 2: 3517ms
Insert time for version 3: 25095ms
Search time for version 3: 1535ms
Insert time for version 4: 10743ms
Search time for version 4: 2723ms
Test for 10^8 records
Skip version 1 for 10^8 records, reason: Time exceeds
Skip version 2 for 10^8 records, reason: Time exceeds
Insert time for version 3: 320971ms
Search time for version 3: 2923ms
Insert time for version 4: 127087ms
Search time for version 4: 2875ms
Test for 10^9 records
error: std::bad_alloc
```

$M = 10^6$	T1_N	T1_M	T2_N	T2_M	T3_N	T3_M	T4_N	T4_M
$N = 10^6$	2502	2329	3752	2264	1728	1039	1011	2704
$N = 10^7$	36313	3278	55478	3517	25095	1535	10743	2723
$N = 10^8$	NaN	NaN	NaN	NaN	320971	2923	127087	2875
$N = 10^9$	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

> 表1-7 4种方案在不同N下的性能评测（单位：ms）

$B = N / T$	B1	B2	B3	B4
$N = 10^6$	399	266	578	***989***
$N = 10^7$	275	180	398	***930***
$N = 10^8$	NaN	NaN	311	***786***
$N = 10^9$	NaN	NaN	NaN	NaN

> 表1-8 由此得到的4种表在4个数量级的N下的16个插入性能指标

$B = M / T$	B1	B2	B3	B4
$N = 10^6$	429	441	***962***	369
$N = 10^7$	3050	2843	***6514***	3672
$N = 10^8$	NaN	NaN	34211	***34782***
$N = 10^9$	NaN	NaN	NaN	NaN

> 表1-8 由此得到的4种表在4个数量级的N下的16个检索性能指标

实验结果呈现以下特点。

1. 第四种方案的插入性能，在几个数量级中始终是最好的

2. 在$N = 10^6$与$N = 10^7$两种情况下，查询性能最好的是第三种方案；但是当N来到10^8时，查询性能最好的方案变成了第四种。
3. 在把value从直接的数据结构转为数组寻址后，仅有在$N = 10^6$的数量级下的查询有一点点性能提升，在$N = 10^6$与$N = 10^7$两种情况下，插入性能都有30%左右的下降。
4. 通过将key从字符串转为整数，插入与查询性能都有了明显的提升。