华东师范大学数据科学与工程学院上机实践报告

课程名称：算法设计与分析	**年级**：22 级	**上机实践成绩**：
指导教师：金澈清	**姓名**：石季凡	
上机实践名称：散列表	**学号**：10225501403	**上机实践日期**：2023.4.17
上机实践编号：No.7	**组号**：1-403	

一、目的

1．熟悉散列表的基本思想。
2. 掌握各种散列表的实现方法。

二、实验内容

1. 设计一个数据生成器，输入参数为 N；可以生成 N 个不重复的随机键或键值对。设计一个操作生成器，输入参数为 N’，method；可以生成 N’组操作 method。操作包括插入和查询。
2. 基于开放寻址法实现哈希表及其插入和查询操作，选择合适的数据规模，计算在不同表的大小和不同已占用数量下的所需时间。
3. 以顺序访问的方式实现插入和查询。选择合适的数据规模，计算在不同表的大小和不同已占用数量下的所需的时间。
4. 对比散列表（哈希表）和顺序访问。
5. （思考题）探究不同散列函数对散列表性能的影响。

三、使用环境

推荐使用 C/C++集成编译环境。

四、实验过程

1. *写出数据生成器和两种算法的源代码。*
2. *以合适的图表来表示你的实验数据。*

五、总结

首先，对于数据生成，生成一个数组 arr 用于记录数据有无出现过，然后随机生成数字并进行检测，如果没有出现过就放入数组。

其次是哈希函数的选择。

首先由于要求为开放寻址法，最简单的方法就是取 key 值为 k mod N，然后再利用线性探查法，需要注意的是，这种方法中的 search 函数的搜寻次数不是固定的，而是搜索到第一个空位置停止。但是这个方案也许对于-N~N 数据范围的数据有着很好的处理，但是对于这个范围以外的数据则很容易产生冲突，从而导致线性探查的次数不断增大。

由于第一种方案很容易产生冲突，所以我决定采用双重散列来作为散列表的 key 值函数，以 N 最大取得 100000 为例，即 size=100000，h (k , i) = (h_1(k)+i*h_2(k)) mod size，

h_1(k) = ((k)%(m_1))，h_2(k) = ((k)%(m_2)+1)，m_1=100000, m_2=99997，由于 99997 为素数，所以随着我们不断探测新位置，总能不重复地探测完所有位置。

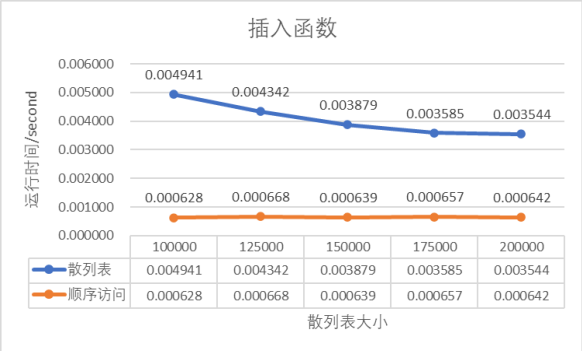
由于实验要求和顺序访问作对比，于是我开始实现顺序访问。

顺序访问很简单，插入函数用 count 记录数据数量，每次插入只需要执行 nums->arr[nums->count++]=key; 即可。简单的插入就会带来搜索的困难，顺序访问的每次搜索都需要遍历，导致耗时很多，每一次的时间复杂度为 O（n）。

methods 生成器较为简单，随机生成一个数字，method[i]=randnum%2，这样返回的数组就是一个由 0 和 1 组成的数组，1 代表插入，0 代表搜索，二者分别用不同的 key 值数组，即随机生成 key_1 数组作为插入的 key 值，再随机生成一个 key_2 数组作为搜索的 key 值，每次取的 key 值都是 key_1[i]或者 key_2[i]，其中 i 为操作数。

首先开始进行插入函数的对比，我们令表大小分别为 100000、125000、150000、175000、200000，每次都插入 80000 个随机数，得到如下数据以及图表。

其中运行时间单位为秒。

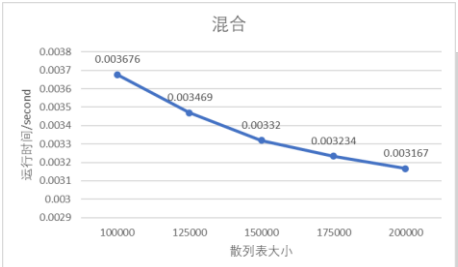


不同size	插入80000	
	散列表	顺序访问
100000	0.004941	0.000628
125000	0.004342	0.000668
150000	0.003879	0.000639
175000	0.003585	0.000657
200000	0.003544	0.000642

首先对于散列表和顺序访问的总体对比，由于 顺序访问有 count 来记录数据量，并且每次只需要在末尾插入，因此比散列表快是非常正常的。

接下来看散列函数，由算法导论中的定理 11.8 可知，对于装载因子 a<1 的一个散列表而言，一次插入的探查期望是 f(a)=1/a*ln(1/(1-a))，而 a=n/m，其中 n 为已经插入的元素个数，m 则为我们实验中的散列表大小，那么我们在不断改变散列表大小的同时其实是在改变 m，进而改变每一次插入的 a，而由公式可知 f(a)是个在定义域内单调递增的函数，因此随着 m 的增大，导致 a 的减小，导致 f(a)的减小。同时我们需要注意到，由于数组大小是线性增大的，而 a 与 m 其实为反比例关系，所以随着 m 的不断增大，a 的减小量会不断减小，同时由于 f’(a)在定义域内恒大于 0，因此 f(a)实际上会随着数组大小不断增大，不断地进行减小幅度越来越小的减小，即时间 t’(size)<0,t’’(size)>0。接下来看顺序访问，由于顺序访问对于数组大小极其不敏感，毕竟无论数组多大，0~80000 的位置会被使用，因此时间波动不大。

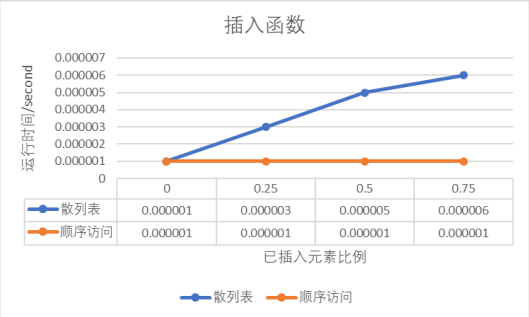
接下来我们把搜索操作也加入进来，进行随机的插入与搜索共计 80000 次，得到如下结果。



	散列表	顺序访问
100000	0.003676	1.783682
125000	0.003469	1.804972
150000	0.00332	1.781009
175000	0.003234	1.799935
200000	0.003167	1.780062

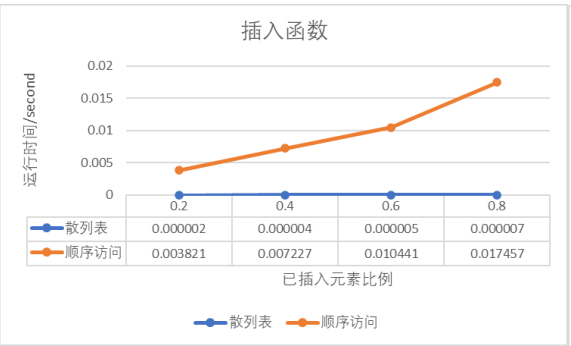
由于顺序访问变化基本不大且数量级远大于散列表便没有画图。由于散列表的搜索与插入操作逻辑基本相同，因此图像曲线依旧很好地保持着越减少越慢的形态，而顺序访问的搜索为遍历，这就相当慢了，最终导致时间数量级远大于散列表。

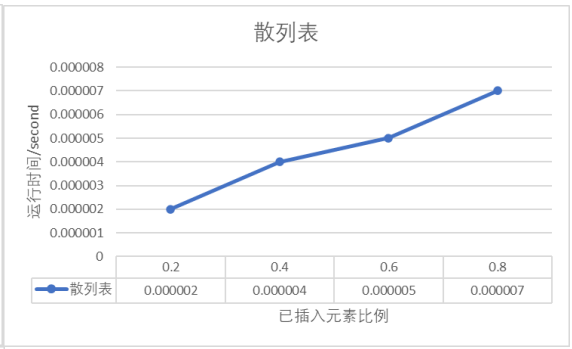
下面进行不同 n 的实验（a=n/m），其中 N=80000，散列表大小为 125000
其中，n 分别为 0、0.25N、0.5N、0.75N，然后进行 100 次插入操作（由于太多次的插入会导致 n 的改变，因此我通过减少插入次数来尽量减小后续操作对于 n 的影响，即可以较为准确地描述不同占有率下插入元素的时间规律，同时，插入次数太少导致的误差可以由多次实验来缩小）

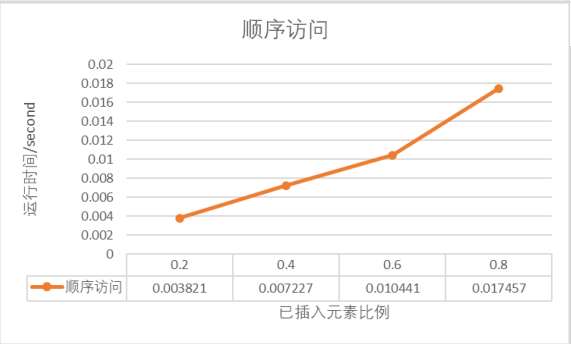


	散列表	顺序访问
0	0.000001	0.000001
0.25	0.000003	0.000001
0.5	0.000005	0.000001
0.75	0.000006	0.000001

最终得到如上结果，我们可以发现，随着 n 逐渐变大，散列表查找耗时逐渐增大，这也验证了 a=n/m，f(a)=1/a*ln(1/(1-a))，t(f(a))=kf(a)，随着 n 增大，t 不断增大，而顺序访问得益于其简单的插入逻辑，最终时间极为稳定且短。
下面来对搜索函数进行实验，即在 n 分别为 0.25N、0.5N、0.75N，然后进行 10 次搜索操作







	散列表	顺序访问
0.2	0.000002	0.003821
0.4	0.000004	0.007227
0.6	0.000005	0.010441
0.8	0.000007	0.017457

如上图可见，散列表其优秀的存储方式使其在查找时间远低于顺序访问，但是随着 n 增大，t 不断增大（其实与上面相同，毕竟搜索与插入函数逻辑极为类似）。

根据上面的实验，我们可以得知，对于开放寻址法有这么一系列函数 a=n/m，f(a)=1/a*ln(1/(1-a))，t(f(a))=kf(a)，其中最为本质的影响时间的因素就是 n 与 m，而顺序访问由于其存储时的“偷懒”，导致其在搜索时耗费非常多的时间。而对于不同的哈希函数而言，本质都是在寻找合适的 key 值以尽可能的避免冲突，而双重散列法和全域散列法等随机性较好的函数往往可以得到冲突更少的 key 值，因此选择合适的哈希函数其实为我们构建散列表时最需要考虑的事情。我们需要根据不同的数据类型与数据特点设计并使用合适的哈希函数以减少冲突。