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：**郭夏辉

上机实践名称：散列表　　**学号：**10211900416　　**上机实践日期：**2023 年 4 月 13 日

上机实践编号：No.7　　**组号：**1-学号后三位）

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。

首先来看算法导论上面关于开放寻址法的描述:

在开放寻址法(open addressing)中，所有的元素都存放在散列表里。也就是说，每个表项或包含动态集合的一个元素，或包含 NIL。当查找某个元素时，要系统地检查所有的表项，直到找到所需的元素，或者最终查明该元素不在表中。不像链接法，这里既没有链表，也没有元素存放在散列表外。因此在开放寻址法中，散列表可能会被填满，以至于不能插入任何新的元素。该方法导致的一个结果便是装载因子 α 绝对不会超过 1。

当然，也可以将用作链接的链表存放在散列表未用的槽中(见练习 11.2-4)，但开放寻址法的好处就在于它不用指针，而是计算出要存取的槽序列。于是，不用存储指针而节省的空间，使得可以用同样的空间来提供更多的槽，潜在地减少了冲突，提高了检索速度。

为了使用开放寻址法插入一个元素，需要连续地检查散列表，或称为**探查**(probe)，直到找到一个空槽来放置待插入的关键字为止。检查的顺序不一定是 0，1，…，$m-1$(这种顺序下的查找时间为 $\Theta(n)$)，而是要依赖于待插入的关键字。为了确定要探查哪些槽，我们将散列函数加以扩充，使之包含探查号(从 0 开始)以作为其第二个输入参数。这样，散列函数就变为：

$$h:U\times\{0,1,\cdots,m-1\}\rightarrow\{0,1,\cdots,m-1\}$$

对每一个关键字 k，使用开放寻址法的**探查序列**(probe sequence)

$$\langle h(k,0),h(k,1),\cdots,h(k,m-1)\rangle$$

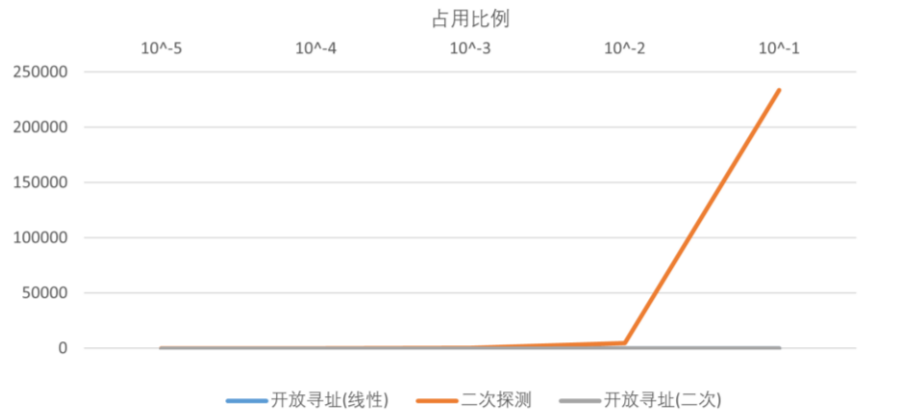
是〈0，1，…，$m-1$〉的一个排列，使得当散列表逐渐填满时，每一个表位最终都可以被考虑为用来插入新关键字的槽。在下面的伪代码中，假设散列表 T 中的元素为无卫星数据的关键字；关键字 k 等同于包含关键字 k 的元素。每个槽或包含一个关键字，或包含 NIL(如果该槽为空)。HASH-INSERT 过程以一个散列表 T 和一个关键字 k 为输入，其要么返回关键字 k 的存储槽位，要么因为散列表已满而返回出错标志。

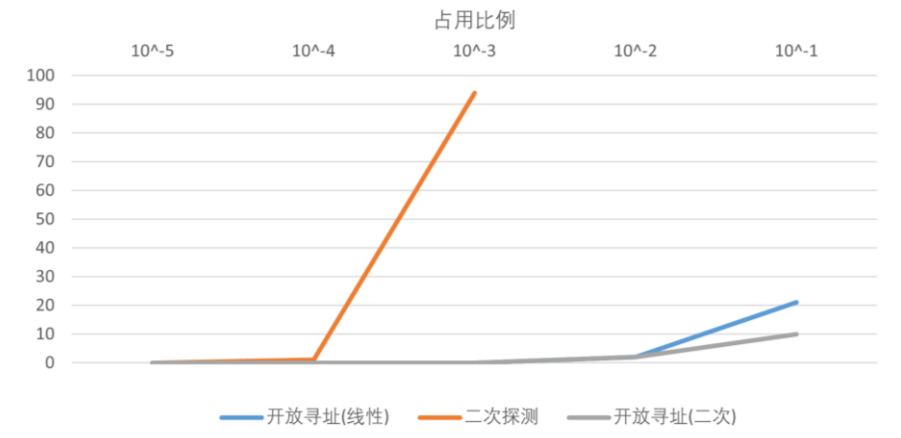
```
HASH-INSERT(T,k)
1  i=0
2  repeat
3      j=h(k,i)
4      if T[j]==NIL
5          T[j]=k
6          return j
7      else i=i+1
8  until i==m
9  error "hash table overflow"
```

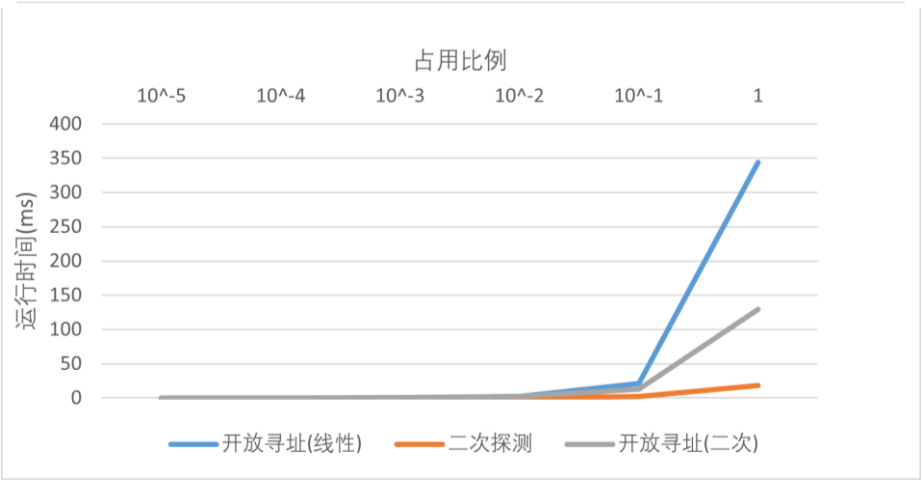
```
HASH-SEARCH(T,k)
1  i=0
2  repeat
3      j=h(k,j)
4      if T[j]==k
5          return j
6      i=i+1
7  until T[j]==NIL or i==m
8  return NIL
```

然后各种各样的散列函数我就不在此赘述了，在课本上都有较详细的描述。

在本次实验过程中，我通过调整数据规模的大小，发现了基于开放寻址法的哈希方法和直接以顺序形式访问的差异，具体如下图所示:







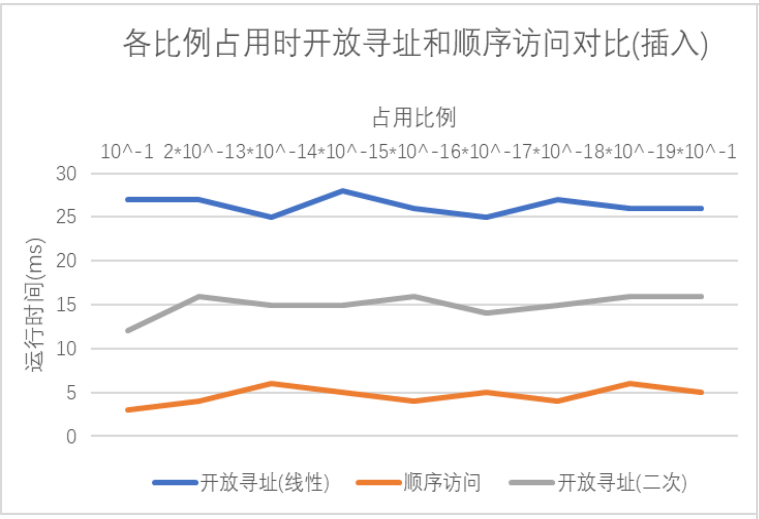
首先我锁定了表的最大容量，然后以这个最大容量为基数不断地扩大比例，实现了动态调整数据规模的目的。（这里可能有小小的歧义，但是之后我还会说不同占用比例时的情况）。

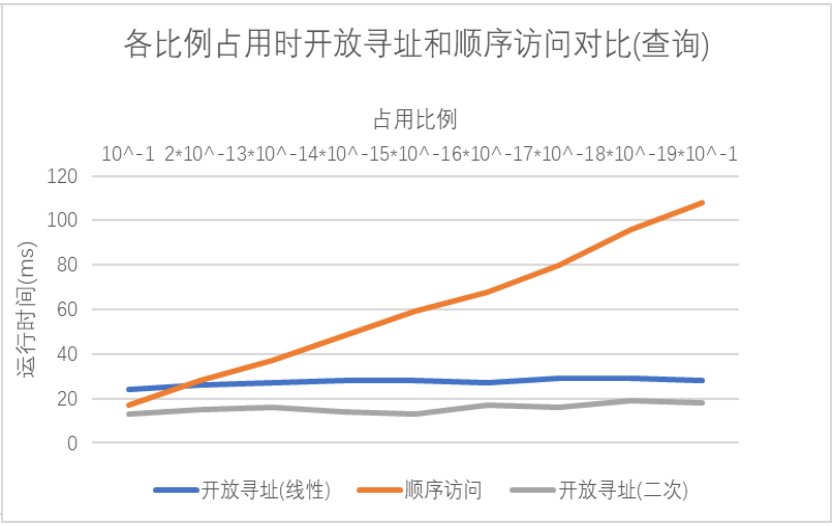
可以发现在查询时，开放寻址法的效率近乎是线性的，而且在细节处对比可以发现利用二次的探查函数效率是要比用线性的探查函数高的。

然而采用顺序访问的方法在查询时的效率就显得不可接受了，随着数据量的增大，查询的时间飙升——面对一个需要查询的数，顺序访问的方法需要一个接着一个地往后找，直到找到对应的，这显然是极其低效的。

然后来看数据的插入时间对比，由于不用关心表项是否被占这个问题，通过顺序访问构建的表只需要把需要插入的元素放到表的下一个空闲位就行了；与之相对比，采用开放寻址法，在插入一个数据时，需要不断地通过 hash 函数探查一个空余的表项来存放数据，这个预处理的过程消耗了一定的时间。我们能明显看到，随着数据量的上升，采用顺序访问的表插入时间缓慢地上升，但是此时开放寻址法所消耗的时间也并未增长很多，保持一个线性的趋势，还是可以接受的。钻研细节，我们还可以发现在插入情况下，采用二次的探查函数在较大数据量时效率也是明显优于采用线性的探查函数的。

然后接下来我会探讨不同占用比例情况下三种方式的插入和查询时间。





我们可以很明确的看到，在表的最大大小确定时，随着表的占用比例的增长，采用开放寻址法的哈希表和顺序访问的表在插入时运行时间维持了相对稳定的趋势，这个根据它们各自的原理也是容易理解的。然而，在查询时情况就有了很大的不同，顺序访问的时间呈现一个很明显的上升趋势（但是近似于线性）而采用开放寻址法的哈希表却依旧维持了一个稳定的趋势。当然这个可能主要还是因为自己的数据规模不算大，然后占用比例也没有设置的很极端，避免装载因子的极端情况（装载因子过大，会使得哈希碰撞的概率大大增大，进而大大降低了效率）。这个的主要原因还是在于它们原理的差异，查询某个值是否在哈希表中是常数时间，影响哈希表时间的决定性因素还是数据的规模；但是随着表中已有项的增多，采用顺序访问的方式要一个个地找到需要的值，这个无疑是增大了很多时间，随着数据规模地扩大，这个递增的趋势还会更加惊人。

(问题)探究不同散列函数对散列表性能的影响

在上述实验中，无论是什么情况下，都可以发现采用二次函数作为探查函数相对来说是比采用线性函数更优的，我就在此先小小地对比一下几种常见的方法吧：

1. 采用二次函数作为探查函数，冲突概率相对较低，均匀分布合理，能够有效减少哈希冲突。这个较为适合我们事先不知道关键字的分布情况，在实际应用中也有较高的安全性;
2. 采用线性定址法，该散列函数取关键字的某个线性函数值作为散列值。这个较为适用于数据较为连续地分布情况下。但是一般情况下线性函数作为探查函数产生的序列不太均匀，虽然发生冲突的概率较大，但是简单易操作，对于小规模的数据有很强的利用价值。
3. 简单余数散列函数。算法十分简单，但是十分容易产生冲突，散列值地分布也不均匀。
4. 随机数散列函数。产生的哈希值分布较为均匀，可有效避免哈希冲突；但是散列函数在单次创建和查询时的时间开销过大

在实际的应用中，还可以采用局部敏感哈希(LSH)等有效的方法，在此就不赘述了。

要论提高散列表的性能，其实主要可以从三个大方向来考虑。首先，就是要保证散列值较为均匀地分散在地址空间中，遇到冲突毕竟是一件消耗时间的事情；其次，就是要控制整个散列表的装载银子，不能过大（增大碰撞概率，影响效率）也不能过小（空间浪费）；最后就是选择合适的哈希函数，在保证散列表固有特性的同时，尽可能使得哈希函数简洁高效，不会徒增运算的时间。

五、总结

对上机实践结果进行分析，问题回答，上机的心得体会及改进意见。

这次实验的整体难度不算大，但是我却受益匪浅。哈希算法在我们日常生活中有着广泛而深刻的应用，掌握哈希函数的具体流程之后，通过详细的对比，我见识了哈希算法的高效性，也希望以此为基础在日后的学习中能最大化它的价值。