

## 实验 8 查找算法比较

### 一、查找的概述

查找是在大量的信息中寻找一个特定的信息元素，在计算机应用中，查找是常用的基本运算，例如编译程序中符号表的查找。本文简单概括性的介绍了常见的七种查找算法，说是七种，其实二分查找、插值查找以及斐波那契查找都可以归为一类——插值查找。插值查找和斐波那契查找是在二分查找的基础上的优化查找算法。树表查找和哈希查找会在后续的博文中进行详细介绍。<sup>[1]</sup>

**查找定义：**根据给定的某个值，在查找表中确定一个其关键字等于给定值的数据元素（或记录）。

#### 查找算法分类：

##### 1) 静态查找和动态查找

注：静态或者动态都是针对查找表而言的。动态表指查找表中有删除和插入操作的表。

##### 2) 无序查找和有序查找

无序查找：被查找数列有序无序均可；

有序查找：被查找数列必须为有序数列。

**平均查找长度（Average Search Length, ASL）：**需和指定 key 进行比较的关键字的个数的期望值，称为查找算法在查找成功时的平均查找长度。

对于含有  $n$  个数据元素的查找表，查找成功的平均查找长度为： $ASL = P_i * C_i$  的和。

$P_i$ ：查找表中第  $i$  个数据元素的概率。

$C_i$ ：找到第  $i$  个数据元素时已经比较过的次数。

### 二、查找算法

#### 1. 顺序查找

说明：顺序查找适合于存储结构为顺序存储或链接存储的线性表。

**基本思想：**顺序查找也称为线形查找，属于无序查找算法。从数据结构线形表的一端开始，顺序扫描，依次将扫描到的结点关键字与给定值  $k$  相比较，若相等则表示查找成功；若扫描结束仍没有找到关键字等于  $k$  的结点，表示查找失败。

**复杂度分析：**

查找成功时的平均查找长度为：（假设每个数据元素的概率相等） $ASL = 1/n(1+2+3+\dots+n) = (n+1)/2$ ；

当查找不成功时，需要  $n+1$  次比较，时间复杂度为  $O(n)$ ；

所以，顺序查找的时间复杂度为  $O(n)$ 。

## 2. 二分查找<sup>[2]</sup>

**说明：**元素必须是有序的，如果是无序的则要先进行排序操作。

**基本思想：**也称为是折半查找，属于有序查找算法。用给定值  $k$  先与中间结点的关键字比较，中间结点把线形表分成两个子表，若相等则查找成功；若不相等，再根据  $k$  与该中间结点关键字的比较结果确定下一步查找哪个子表，这样递归进行，直到查找到或查找结束发现表中没有这样的结点。

**复杂度分析：**最坏情况下，关键词比较次数为  $\log_2(n+1)$ ，且期望时间复杂度为  $O(\log_2 n)$ ；

**注：**折半查找的前提条件是需要有序表顺序存储，对于静态查找表，一次排序后不再变化，折半查找能得到不错的效率。但对于需要频繁执行插入或删除操作的数据集来说，维护有序的排序会带来不小的工作量，那就不建议使用。——《大话数据结构》

## 3. 插值查找

在介绍插值查找之前，首先考虑一个新问题，为什么上述算法一定要是折半，而不是折四分之一或者折更多呢？

打个比方，在英文字典里面查“apple”，你下意识翻开字典是翻前面的书页还是后面的书页呢？如果再让你查“zoo”，你又怎么查？很显然，这里你绝对不会是从中间开始查起，而是有一定目的的往前或往后翻。

同样的，比如要在取值范围  $1 \sim 10000$  之间  $100$  个元素从小到大均匀分布的数组中查找  $5$ ，我们自然会考虑从数组下标较小的开始查找。

经过以上分析，折半查找这种查找方式，不是自适应的（也就是说是傻瓜式的）。二分查找中查找点计算如下：

$mid = (low + high) / 2$ ，即  $mid = low + 1/2 * (high - low)$ ；

通过类比，我们可以将查找的点改进为如下：

$mid = low + (key - a[low]) / (a[high] - a[low]) * (high - low)$ ，

也就是将上述的比例参数  $1/2$  改进为自适应的，根据关键字在整个有序表中所处的位置，让  $mid$  值的变化更靠近关键字  $key$ ，这样也就间接地减少了比较次数。<sup>[3]</sup>

**基本思想：**基于二分查找算法，将查找点的选择改进为自适应选择，可以提高查找效率。当然，差值查找也属于有序查找。

**注：**对于表长较大，而关键字分布又比较均匀的查找表来说，插值查找算法的平均性能比折半查找要好的多。反之，数组中如果分布非常不均匀，那么插值查找未必是很合适的选择。

**复杂度分析：**查找成功或者失败的时间复杂度均为  $O(\log_2(\log_2 n))$ 。

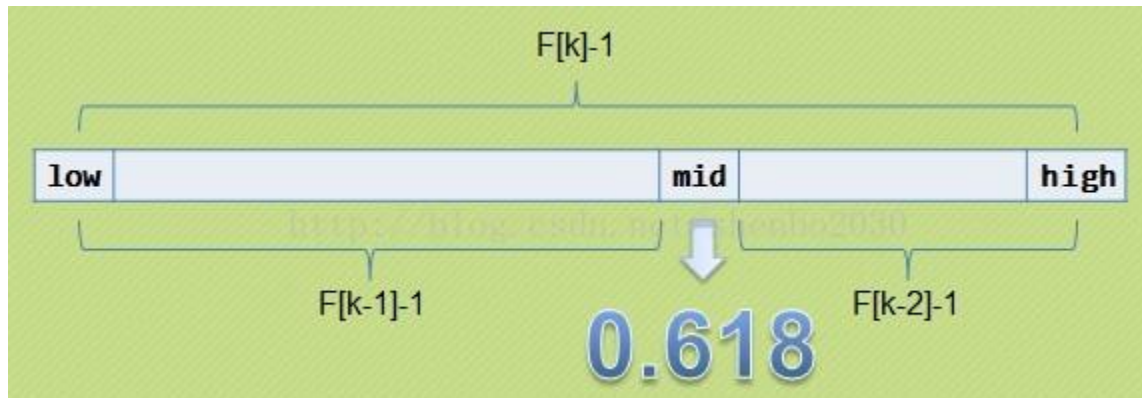
#### 4. 斐波那契查找

在介绍斐波那契查找算法之前，我们先介绍一下很它紧密相连并且大家都熟知的一个概念——黄金分割。

黄金比例又称黄金分割，是指事物各部分间一定的数学比例关系，即将整体一分为二，较大部分与较小部分之比等于整体与较大部分之比，其比值约为  $1:0.618$  或  $1.618:1$ 。<sup>[4]</sup>

$0.618$  被公认为最具有审美意义的比例数字，这个数值的作用不仅仅体现在诸如绘画、雕塑、音乐、建筑等艺术领域，而且在管理、工程设计等方面也有着不可忽视的作用。因此被称为黄金分割。

大家记不记得斐波那契数列：1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, .....（从第三个数开始，后边每一个数都是前两个数的和）。然后我们会发现，随着斐波那契数列的递增，前后两个数的比值会越来越接近  $0.618$ ，利用这个特性，我们就可以将黄金比例运用到查找技术中。



**基本思想：**也是二分查找的一种提升算法，通过运用黄金比例的概念在数列中选择查找点进行查找，提高查找效率。同样地，斐波那契查找也属于一种有序查找算法。

相对于折半查找，一般将待比较的 key 值与第  $\text{mid} = (\text{low} + \text{high}) / 2$  位置的元素比较，比较结果分三种情况：

- 1) 相等，mid 位置的元素即为所求
- 2)  $>$ ， $\text{low} = \text{mid} + 1$ ;
- 3)  $<$ ， $\text{high} = \text{mid} - 1$ 。

斐波那契查找与折半查找很相似，他是根据斐波那契序列的特点对有序表进行分割的。他要求开始表中记录的个数为某个斐波那契数小 1，及  $n = F(k) - 1$ ；开始将 k 值与第  $F(k-1)$  位置的记录进行比较(及  $\text{mid} = \text{low} + F(k-1) - 1$ )，比较结果也分为三种

- 1) 相等，mid 位置的元素即为所求
- 2)  $>$ ， $\text{low} = \text{mid} + 1, k = 2$ ;

说明： $\text{low} = \text{mid} + 1$  说明待查找的元素在  $[\text{mid} + 1, \text{high}]$  范围内， $k = 2$  说明范围  $[\text{mid} + 1, \text{high}]$  内的元素个数为  $n - (F(k-1)) = Fk - 1 - F(k-1) = Fk - F(k-1) - 1 = F(k-2) - 1$  个，所以可以递归的应用斐波那契查找。

- 3)  $<$ ， $\text{high} = \text{mid} - 1, k = 1$ 。

说明： $\text{low} = \text{mid} + 1$  说明待查找的元素在  $[\text{low}, \text{mid} - 1]$  范围内， $k = 1$  说明范围  $[\text{low}, \text{mid} - 1]$  内的元素个数为  $F(k-1) - 1$  个，所以可以递归 的应用斐波那契查找。

**复杂度分析：**最坏情况下，时间复杂度为  $O(\log_2 n)$ ，且其期望复杂度也为  $O(\log_2 n)$ 。

## 5. 树表查找

最简单的树表查找算法——二叉树查找算法。

**基本思想：**二叉查找树是先对待查找的数据进行生成树，确保树的左分支的值小于右分支的值，然后在就行和每个节点的父节点比较大小，查找最适合的范围。这个算法的查找效率很高，但是如果使用这种查找方法要首先创建树。

**二叉查找树**（BinarySearch Tree，也叫二叉搜索树，或称二叉排序树 Binary Sort Tree）或者是一棵空树，或者是具有下列性质的二叉树：

- 1) 若任意节点的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
- 2) 若任意节点的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
- 3) 任意节点的左、右子树也分别为二叉查找树。

**二叉查找树性质：**对二叉查找树进行中序遍历，即可得到有序的数列。

**复杂度分析：**它和二分查找一样，插入和查找的时间复杂度均为  $O(\log n)$ ，但是在最坏的情况下仍然会有  $O(n)$  的时间复杂度。原因在于插入和删除元素的时候，树没有保持平衡（比如，我们查找上图（b）中的“93”，我们需要进行  $n$  次查找操作）。我们追求的是在最坏的情况下仍然有较好的时间复杂度，这就是平衡查找树设计的初衷。

下图为二叉树查找和顺序查找以及二分查找性能的对比图：<sup>[5]</sup>

| implementation                  | guarantee |        |        | average case |           |            | ordered iteration? | operations on keys |
|---------------------------------|-----------|--------|--------|--------------|-----------|------------|--------------------|--------------------|
|                                 | search    | insert | delete | search hit   | insert    | delete     |                    |                    |
| sequential search (linked list) | N         | N      | N      | N/2          | N         | N/2        | no                 | equals ()          |
| binary search (ordered array)   | lg N      | N      | N      | lg N         | N/2       | N/2        | yes                | compareTo ()       |
| BST                             | N         | N      | N      | 1.39 lg N    | 1.39 lg N | $\sqrt{N}$ | yes                | compareTo ()       |

other operations also become  $\sqrt{N}$  if deletions allowed

基于二叉查找树进行优化，进而可以得到其他的树表查找算法，如平衡树、红黑树等高效算法。

| implementation                        | worst-case cost<br>(after N inserts) |           |           | average case<br>(after N random inserts) |                |                | ordered<br>iteration? | key<br>interface         |
|---------------------------------------|--------------------------------------|-----------|-----------|--|----------------|----------------|-----------------------|--------------------------|
|                                       | search                               | insert    | delete    | search hit                               | insert         | delete         |                       |                          |
| sequential search<br>(unordered list) | N                                    | N         | N         | N/2                                      | N              | N/2            | no                    | <code>equals()</code>    |
| binary search<br>(ordered array)      | $\lg N$                              | N         | N         | $\lg N$                                  | N/2            | N/2            | yes                   | <code>compareTo()</code> |
| BST                                   | N                                    | N         | N         | $1.39 \lg N$                             | $1.39 \lg N$   | ?              | yes                   | <code>compareTo()</code> |
| 2-3 tree                              | $c \lg N$                            | $c \lg N$ | $c \lg N$ | $c \lg N$                                | $c \lg N$      | $c \lg N$      | yes                   | <code>compareTo()</code> |
| red-black BST                         | $2 \lg N$                            | $2 \lg N$ | $2 \lg N$ | $1.00 \lg N^*$                           | $1.00 \lg N^*$ | $1.00 \lg N^*$ | yes                   | <code>compareTo()</code> |

### 树表查找总结：

二叉查找树平均查找性能不错，为  $O(\log n)$ ，但是最坏情况会退化为  $O(n)$ 。在二叉查找树的基础上进行优化，我们可以使用平衡查找树。平衡查找树中的 2-3 查找树，这种数据结构在插入之后能够进行自平衡操作，从而保证了树的高度在一定的范围内进而能够保证最坏情况下的时间复杂度。但是 2-3 查找树实现起来比较困难，红黑树是 2-3 树的一种简单高效的实现，他巧妙地使用颜色标记来替代 2-3 树中比较难处理的 3-node 节点问题。红黑树是一种比较高效的平衡查找树，应用非常广泛，很多编程语言的内部实现都或多或少的采用了红黑树。

除此之外，2-3 查找树的另一个扩展——B/B+平衡树，在文件系统和数据库系统中有着广泛的应用。

## 6. 分块查找

分块查找又称索引顺序查找，它是顺序查找的一种改进方法。

**算法思想：**将  $n$  个数据元素"按块有序"划分为  $m$  块 ( $m \leq n$ )。每一块中的结点不必有序，但块与块之间必须"按块有序"；即第 1 块中任一元素的关键字都必须小于第 2 块中任一元素的关键字；而第 2 块中任一元素又都必须小于第

3 块中的任一元素， .....

**算法流程：**

step1 先选取各块中的最大关键字构成一个索引表；

step2 查找分两个部分：先对索引表进行二分查找或顺序查找，以确定待查记录在哪一块中；然后，在已确定的块中用顺序法进行查找。

## 7. 哈希查找

### 什么是哈希表（Hash）？

我们使用一个下标范围比较大的数组来存储元素。可以设计一个函数（哈希函数， 也叫做散列函数），使得每个元素的关键字都与一个函数值（即数组下标）相对应，于是用这个数组单元来存储这个元素；也可以简单的理解为，按照关键字为每一个元素"分类"，然后将这个元素存储在相应"类"所对应的地方。但是，不能够保证每个元素的关键字与函数值是一一对应的，因此极有可能出现对于不同的元素，却计算出了相同的函数值，这样就产生了"冲突"，换句话说，就是把不同的元素分在了相同的"类"之中。后面我们将看到一种解决"冲突"的简便做法。

总的来说，"直接定址"与"解决冲突"是哈希表的两大特点。

### 什么是哈希函数？

哈希函数的规则是：通过某种转换关系，使关键字适度的分散到指定大小的顺序结构中，越分散，则以后查找的时间复杂度越小，空间复杂度越高。

**算法思想：**哈希的思路很简单，如果所有的键都是整数，那么就可以使用一个简单的无序数组来实现：将键作为索引，值即为其对应的值，这样就可以快速访问任意键的值。这是对于简单的键的情况，我们将其扩展到可以处理更加复杂的类型的键。

**算法流程：**

1) 用给定的哈希函数构造哈希表；

2) 根据选择的冲突处理方法解决地址冲突，常见的解决冲突的方法：拉链法和线性探测法。

3) 在哈希表的基础上执行哈希查找。

哈希表是一个在时间和空间上做出权衡的经典例子。如果没有内存限制，那么可以直接将键作为数组的索引。那么所有的查找时间复杂度为  $O(1)$ ；如果没有时间限制，那么我们可以使用无序数组并进行顺序查找，这样只需要很少的内存。哈希表使用了适度的时间和空间来在这两个极端之间找到了平衡。只需要调整哈希函数算法即可在时间和空间上做出取舍。<sup>[6]</sup>

#### 复杂度分析：

单纯论查找复杂度：对于无冲突的 Hash 表而言，查找复杂度为  $O(1)$ （注意，在查找之前我们需要构建相应的 Hash 表）。

#### 使用 Hash，我们付出了什么？

我们在实际编程中存储一个大规模的数据，最先想到的存储结构可能就是 map，也就是我们常说的 KV pair，经常使用 Python 的博友可能更有这种体会。使用 map 的好处就是，我们在后续处理数据处理时，可以根据数据的 key 快速的查找到对应的 value 值。map 的本质就是 Hash 表，那我们在获取了超高查找效率的基础上，我们付出了什么？

Hash 是一种典型以空间换时间的算法，比如原来一个长度为 100 的数组，对其查找，只需要遍历且匹配相应记录即可，从空间复杂度上来看，假如数组存储的是 byte 类型数据，那么该数组占用 100byte 空间。现在我们采用 Hash 算法，我们前面说的 Hash 必须有一个规则，约束键与存储位置的关系，那么就需要一个固定长度的 hash 表，此时，仍然是 100byte 的数组，假设我们需要的 100byte 用来记录键与位置的关系，那么总的空间为 200byte，而且用于记录规则的表大小会根据规则，大小可能是不定的。

Hash 算法和其他查找算法的性能对比：<sup>[7]</sup>



| implementation                        | worst-case cost<br>(after N inserts) |           |           | average case<br>(after N random inserts) |              |              | ordered<br>iteration? | key<br>interface         |
|---------------------------------------|--------------------------------------|-----------|-----------|--|--------------|--------------|-----------------------|--------------------------|
|                                       | search                               | insert    | delete    | search hit                               | insert       | delete       |                       |                          |
| sequential search<br>(unordered list) | N                                    | N         | N         | N/2                                      | N            | N/2          | no                    | <code>equals()</code>    |
| binary search<br>(ordered array)      | $\lg N$                              | N         | N         | $\lg N$                                  | N/2          | N/2          | yes                   | <code>compareTo()</code> |
| BST                                   | N                                    | N         | N         | $1.38 \lg N$                             | $1.38 \lg N$ | ?            | yes                   | <code>compareTo()</code> |
| red-black tree                        | $2 \lg N$                            | $2 \lg N$ | $2 \lg N$ | $1.00 \lg N$                             | $1.00 \lg N$ | $1.00 \lg N$ | yes                   | <code>compareTo()</code> |
| separate chaining                     | $\lg N^*$                            | $\lg N^*$ | $\lg N^*$ | $3-5^*$                                  | $3-5^*$      | $3-5^*$      | no                    | <code>equals()</code>    |
| linear probing                        | $\lg N^*$                            | $\lg N^*$ | $\lg N^*$ | $3-5^*$                                  | $3-5^*$      | $3-5^*$      | no                    | <code>equals()</code>    |

### 三、实验进程

#### 12 月 24 日：

1. 结合课本及 PPT 回顾理解哈希查找以及二分查找的的相关知识。
2. 通过网络学习如何使用 `fstream` 文件流实现文件的读写功能。

#### 12 月 25 日：

1. 实现了哈希查找以及二分查找的基本查找功能。
2. 通过网络了解 `windows.h` 头文件中包含的函数如何获取查找所需的时间，使用其中的 `QueryPerformanceCounter(&frequency)` 函数来获取开始查找时间和结束查找时间。

#### 12 月 26 日：

1. 通过网络了解 `time.h` 头文件中包含的函数如何生成随机数，使用其中的 `rand()` 函数来获取随机数。
2. 将文件的读写加入，实现了不同数据规模文件的创建、写入，以及查找时间的计算。
3. 完成 `cpp` 文件。

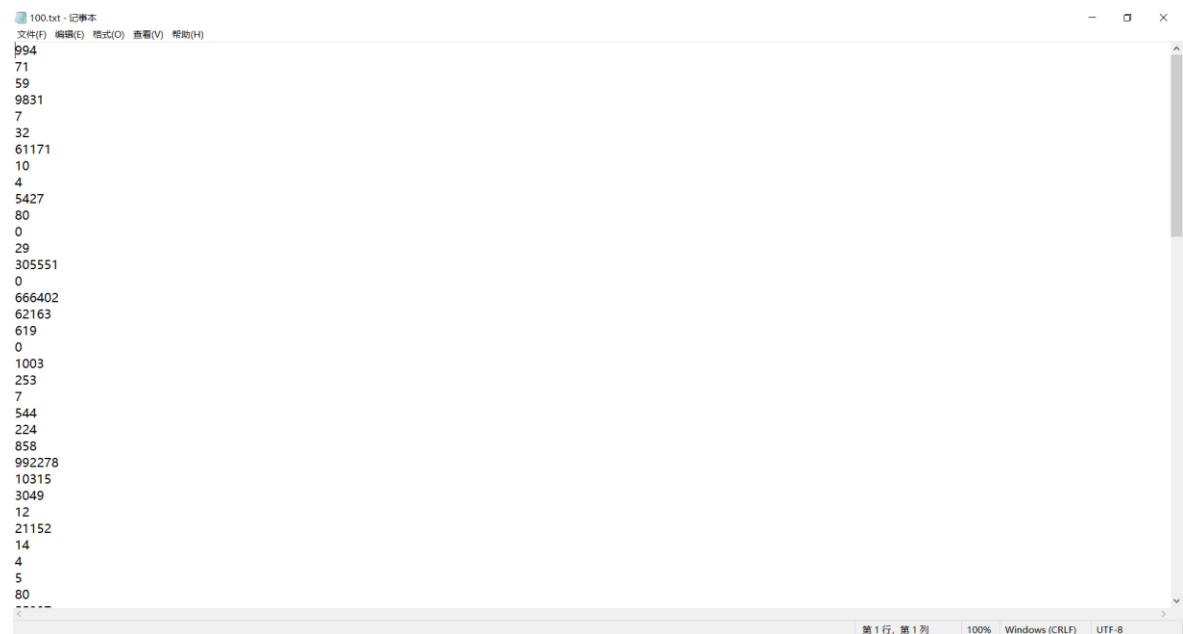
**12月27日：**

1. 通过 CSDN 进一步学习查找的相关知识。
2. 修改并完善实验日志，提交到学习通平台。

#### 四、遇到的问题及解决方法

最初调用 `rand()`函数生成随机数时，并没有指定随机数的种子，导致每次生成的随机数都是相同的，后来调用 `srand((int)time(NULL))`函数，将随机数种子设置为系统时间，解决了这个问题。

#### 五、测试用例



The image shows a Notepad++ window titled "100.txt - 记事本". The window contains 100 lines of random integers. The status bar at the bottom indicates "第 1 行, 第 1 列", "100%", "Windows (CRLF)", and "UTF-8".

```
994
71
59
9831
7
32
61171
10
4
5427
80
0
29
305551
0
666402
62163
619
0
1003
253
7
544
224
858
992278
10315
3049
12
21152
14
4
5
80
-----
```

数据规模为100的测试数据

```
out_100.txt - 记事本
文件(F)  编辑(E)  格式(O)  查看(V)  帮助(H)
数据规模为100的数据进行hash查找,结果如下
hash查找记录:4379->成功      查找次数:1 次查找时间: 0.0002ms
hash查找记录:17572->成功     查找次数:1 次查找时间: 0.0002ms
hash查找记录:3142->成功     查找次数:1 次查找时间: 0.0001ms
hash查找记录:36->成功       查找次数:1 次查找时间: 0.0001ms
hash查找记录:95->成功       查找次数:1 次查找时间: 0.0001ms
hash查找记录:5->成功        查找次数:1 次查找时间: 0.0001ms
hash查找记录:60472->成功    查找次数:1 次查找时间: 0.0002ms
hash查找记录:504348->成功   查找次数:1 次查找时间: 0.0002ms
hash查找记录:8536->成功     查找次数:1 次查找时间: 0.0002ms
hash查找记录:93176->成功    查找次数:1 次查找时间: 0.0002ms
hash查找记录:987->成功      查找次数:1 次查找时间: 0.0001ms
hash查找记录:0->成功        查找次数:1 次查找时间: 0ms
hash查找记录:46906->成功    查找次数:1 次查找时间: 0.0002ms
hash查找记录:2->成功        查找次数:1 次查找时间: 0ms
hash查找记录:151->成功      查找次数:1 次查找时间: 0.0001ms
hash查找记录:83->成功       查找次数:1 次查找时间: 0ms
hash查找记录:59->成功       查找次数:1 次查找时间: 0.0001ms
hash查找记录:0->成功        查找次数:1 次查找时间: 0ms
hash查找记录:41->成功       查找次数:1 次查找时间: 0.0001ms
hash查找记录:0->成功        查找次数:1 次查找时间: 0ms
hash查找记录:8->成功        查找次数:1 次查找时间: 0.0001ms
hash查找记录:7->成功        查找次数:1 次查找时间: 0ms
hash查找记录:692->成功     查找次数:1 次查找时间: 0.0002ms
hash查找记录:0->成功        查找次数:1 次查找时间: 0.0001ms
hash查找记录:49696->成功    查找次数:1 次查找时间: 0.0002ms
hash查找记录:73->成功       查找次数:1 次查找时间: 0.0001ms
hash查找记录:0->成功        查找次数:1 次查找时间: 0ms
hash查找记录:7086->成功     查找次数:1 次查找时间: 0.0002ms
hash查找记录:63905->成功    查找次数:1 次查找时间: 0.0002ms
hash查找记录:9->成功        查找次数:1 次查找时间: 0.0001ms
hash查找记录:87->成功       查找次数:1 次查找时间: 0.0001ms
hash查找记录:995300->成功   查找次数:1 次查找时间: 0.0003ms
hash查找记录:21->成功       查找次数:1 次查找时间: 0ms
```

数据规模为100的哈希查找结果

```
D:\文档\数据结构实验\实验8查找算法实验比较-201908010705-杨杰\二分查找和哈希查找.exe
数据规模为1e+006的数据进行binary查找,结果如下
二分查找记录->5031->成功      查找次数: 19次查找时间: 0.0003ms
二分查找记录->8698->成功      查找次数: 17次查找时间: 0.0009ms
二分查找记录->116031->失败    查找次数: 20次查找时间: 0.0024ms
二分查找记录->14352->失败     查找次数: 20次查找时间: 0.0028ms
二分查找记录->405->成功       查找次数: 16次查找时间: 0.0024ms
二分查找记录->0->成功         查找次数: 1次查找时间: 0.0003ms
二分查找记录->5->成功         查找次数: 6次查找时间: 0.0013ms
二分查找记录->954435->失败    查找次数: 20次查找时间: 0.0014ms
二分查找记录->0->成功         查找次数: 1次查找时间: 0.0002ms
二分查找记录->0->成功         查找次数: 1次查找时间: 0.0002ms
二分查找记录->64->成功        查找次数: 13次查找时间: 0.0015ms
二分查找记录->0->成功         查找次数: 1次查找时间: 0.0004ms
二分查找记录->45793->成功     查找次数: 18次查找时间: 0.0022ms
二分查找记录->398->成功       查找次数: 16次查找时间: 0.0018ms
二分查找记录->71->成功        查找次数: 4次查找时间: 0.0006ms
二分查找记录->932523->成功    查找次数: 16次查找时间: 0.0016ms
二分查找记录->0->成功         查找次数: 1次查找时间: 0.0002ms
二分查找记录->9334->成功     查找次数: 19次查找时间: 0.0022ms
二分查找记录->56459->失败    查找次数: 20次查找时间: 0.0011ms
二分查找记录->12->成功       查找次数: 10次查找时间: 0.0017ms
二分查找记录->35->成功        查找次数: 11次查找时间: 0.0024ms
二分查找记录->71->成功        查找次数: 4次查找时间: 0.0005ms
二分查找记录->256263->成功    查找次数: 18次查找时间: 0.0016ms
二分查找记录->86->成功        查找次数: 12次查找时间: 0.0011ms
二分查找记录->463->成功       查找次数: 13次查找时间: 0.0012ms
二分查找记录->35767->成功     查找次数: 19次查找时间: 0.0026ms
二分查找记录->6->成功         查找次数: 9次查找时间: 0.0019ms
二分查找记录->718802->失败   查找次数: 20次查找时间: 0.0017ms
二分查找记录->955656->失败   查找次数: 20次查找时间: 0.0014ms
```

数据规模为1M的二分查找结果

```
D:\文档\数据结构实验\实验8查找算法实验比较-201908010705-杨杰\二分查找和哈希查找.exe
数据规模为100的数据进行hash查找, 结果如下
hash查找记录:5031->失败      查找次数:1 次查找时间: 0.0002ms
hash查找记录:8698->失败      查找次数:1 次查找时间: 0.0002ms
hash查找记录:116031->失败     查找次数:1 次查找时间: 0.0003ms
hash查找记录:14352->失败     查找次数:1 次查找时间: 0.0002ms
hash查找记录:405->失败       查找次数:1 次查找时间: 0.0002ms
hash查找记录:0->成功         查找次数:1 次查找时间: 0.0001ms
hash查找记录:5->成功         查找次数:1 次查找时间: 0.0001ms
hash查找记录:954435->失败    查找次数:1 次查找时间: 0.0003ms
hash查找记录:0->成功         查找次数:1 次查找时间: 0.0001ms
hash查找记录:0->成功         查找次数:1 次查找时间: 0.0001ms
hash查找记录:64->失败        查找次数:1 次查找时间: 0.0001ms
hash查找记录:0->成功         查找次数:1 次查找时间: 0ms
hash查找记录:45793->失败     查找次数:1 次查找时间: 0.0002ms
hash查找记录:398->失败       查找次数:1 次查找时间: 0.0001ms
hash查找记录:71->失败        查找次数:1 次查找时间: 0.0001ms
hash查找记录:932523->失败    查找次数:1 次查找时间: 0.0003ms
hash查找记录:0->成功         查找次数:1 次查找时间: 0.0001ms
hash查找记录:9334->失败     查找次数:1 次查找时间: 0.0002ms
hash查找记录:56459->失败    查找次数:1 次查找时间: 0.0002ms
hash查找记录:12->失败        查找次数:1 次查找时间: 0.0002ms
hash查找记录:35->成功         查找次数:1 次查找时间: 0.0001ms
hash查找记录:71->失败        查找次数:1 次查找时间: 0.0001ms
hash查找记录:256263->失败    查找次数:1 次查找时间: 0.0003ms
hash查找记录:86->成功         查找次数:1 次查找时间: 0ms
hash查找记录:463->失败       查找次数:1 次查找时间: 0.0002ms
hash查找记录:35767->失败    查找次数:1 次查找时间: 0.0002ms
hash查找记录:6->失败         查找次数:1 次查找时间: 0ms
hash查找记录:718902->失败    查找次数:1 次查找时间: 0.0002ms
hash查找记录:955656->失败    查找次数:1 次查找时间: 0.0007ms
```

数据规模为100的哈希查找结果

```
D:\文档\数据结构实验\实验8查找算法实验比较-201908010705-杨杰\二分查找和哈希查找.exe
二分查找记录->341868->失败    查找次数: 20次查找时间: 0.0021ms
二分查找记录->39326->失败     查找次数: 20次查找时间: 0.002ms
二分查找记录->222663->失败    查找次数: 20次查找时间: 0.0024ms
二分查找记录->324381->失败    查找次数: 20次查找时间: 0.0047ms
二分查找记录->39393->失败     查找次数: 19次查找时间: 0.0028ms
二分查找记录->250404->失败    查找次数: 20次查找时间: 0.0022ms
二分查找记录->89605->失败     查找次数: 20次查找时间: 0.002ms
二分查找记录->80147->失败     查找次数: 19次查找时间: 0.0016ms
二分查找记录->348246->失败    查找次数: 20次查找时间: 0.0018ms
二分查找记录->83922->失败     查找次数: 20次查找时间: 0.0014ms
二分查找记录->73564->失败     查找次数: 20次查找时间: 0.0036ms
二分查找记录->418713->失败    查找次数: 19次查找时间: 0.0028ms
二分查找记录->411847->失败    查找次数: 20次查找时间: 0.0022ms
二分查找记录->94839->失败     查找次数: 20次查找时间: 0.0024ms
二分查找记录->300027->失败    查找次数: 20次查找时间: 0.0028ms
二分查找记录->39393->失败     查找次数: 19次查找时间: 0.0038ms
二分查找记录->532029->失败    查找次数: 20次查找时间: 0.0023ms
二分查找记录->137211->失败    查找次数: 20次查找时间: 0.0022ms
二分查找记录->65242->失败     查找次数: 20次查找时间: 0.0024ms
二分查找记录->46635->失败     查找次数: 20次查找时间: 0.0026ms
二分查找记录->625629->失败    查找次数: 20次查找时间: 0.0026ms
二分查找记录->69100->失败     查找次数: 20次查找时间: 0.0026ms
二分查找记录->13180->失败     查找次数: 20次查找时间: 0.0028ms
二分查找记录->94289->失败     查找次数: 20次查找时间: 0.0022ms
二分查找成功: 最小查找时间mints=0.0002ms      最大查找时间maxts=0.0071ms      平均查找时间averages=0.001948ms
查找结束
二分查找失败: 最小查找时间mintf=0.0012ms      最大查找时间maxtf=0.0075ms      平均查找时间averagef=0.002579ms
查找结束
```

数据规模为1M的二分查找结果

## 六、实验结果

| 查找算法 | 查找结果 | 数据规模   | 100      | 1K       | 10K      | 100K     | 1M       |
|------|------|--------|----------|----------|----------|----------|----------|
|      |      | 查找时间   |          |          |          |          |          |
| 二分查找 | 成功   | 最小查找时间 | 0.0002   | 0.0002   | 0.0001   | 0.0002   | 0.0002   |
|      |      | 最大查找时间 | 0.0009   | 0.0014   | 0.0019   | 0.0025   | 0.0036   |
|      |      | 平均查找时间 | 0.000486 | 0.000574 | 0.000705 | 0.001031 | 0.001362 |
|      | 失败   | 最小查找时间 | 0.0002   | 0.0002   | 0.0003   | 0.0005   | 0.0008   |
|      |      | 最大查找时间 | 0.0013   | 0.0014   | 0.0015   | 0.0022   | 0.0053   |
|      |      | 平均查找时间 | 0.000502 | 0.000528 | 0.000714 | 0.001134 | 0.001715 |
| 哈希查找 | 成功   | 最小查找时间 | 0        | 0        | 0        | 0        | 0        |
|      |      | 最大查找时间 | 0.0009   | 0.0007   | 0.0008   | 0.0004   | 0.0004   |
|      |      | 平均查找时间 | 0.000143 | 0.000153 | 0.00015  | 0.000136 | 0.000131 |
|      | 失败   | 最小查找时间 | 0        | 0.0001   | 0.0001   | 0.0001   | 0.0001   |
|      |      | 最大查找时间 | 0.0007   | 0.0008   | 0.0006   | 0.001    | 0.0009   |
|      |      | 平均查找时间 | 0.000263 | 0.000269 | 0.000274 | 0.000264 | 0.000296 |

（时间单位：ms）

## 七、实验分析

从上述实验数据中可以看出，当数据量较小时，哈希查找和二分查找的时间消耗相当；但是随着实验数据的增多，不难发现哈希查找所用的时间较少；同时，当数据规模一定时，查找成功所用时间要比查找失败所用时间少。

从实验结果可以看出，实验存在误差，经过分析，发现原因：一是时间函数的精确度不高，导致二分查找和小数据的哈希查找所消耗的时间判定几乎为0。二是求取随机数的函数，当数据量较大时，最终结果偏小。

总体来讲，二分查找所耗时间要比哈希查找高。

## 八、总结与心得

经过这次实验，我对于二分查找和哈希查找的相关代码已基本熟悉，算法知识得到了复习与巩固。在写代码与调试的过程中，在解决问题过程中，丰富了个人编程的经历和经验，提高了个人解决问题的能力。

通过本次报告实验，我对查找的概念有了一个新的认识，由于这段时间事情比较多，故在实验上未花很多的精力，用了三天抽空把查找的两种算法做了，有些不尽人意的地方也没有加以修正，但是实验的过程之中我又对查找的知识重新温习了一遍，获益匪浅。我希望能够做更多这样的实验，这样我才能发现自己在实际操作中的不足并加以改正。

## 九、实验说明

开发语言：C++

开发平台：Dev-C++

## 参考资料

1. 许卓群. 数据结构与算法[M]. 高等教育出版社, 2004.
2. 彭军、向毅. 数据结构与算法[M]. 人民邮电出版社, 2013.
3. Budd T. 经典数据结构（Java 语言版）[M]. 清华大学出版社, 2005.
4. 王晓东. 算法设计与分析[M]. 清华大学出版社, 2003. 5.
5. 严蔚敏. 数据结构 C 语言版[M]. 清华大学出版社, 2007.
6. 王广芳. 数据结构算法与应用-C++语言描述[M]. 机械工业出版社, 2006.
7. 严蔚敏等. 数据结构题集(C 语言版)[M]. 清华大学出版社, 1999. 2.