屹立40年的计算机科学猜想落幕，革新出自本科生之手

原创 陈清扬 返朴 2025年04月22日 08:04 北京

加**星标**，才能不错过每日推送！方法见文末插图

2024年10月，刚刚成为剑桥大学研究生的 Andrew Krapivin与两位合作者发表论文，推翻了计算机科学领域关于哈希表一项持续 40 年的猜想。哈希表是计算机最常用的数据结构之一，他们设计的新型哈希表大幅降低了复杂度。这一成果突破了图灵奖得主姚期智 1985 年的经典理论，重新定义了哈希表的性能极限，为数据结构设计开辟了新思路。

撰文｜陈清扬

引言

2021年秋季，来自罗格斯大学的一名大二学生Andrew Krapivin意外读到了他的算法老师Martín Farach-Colton的一篇论文*Tiny Pointers*[1]，受到论文启发，在两年后Krapivin设计了一种新的哈希表，并最终和他的老师Martín Farach-Colton以及卡内基梅隆大学的William Kuszmau 在 2024 年的 IEEE Symposium on Foundations of Computer Science (FOCS) 会议上发表论文。该论文重新审视了一个计算机领域久远的问题：开放寻址哈希表的探测复杂度[2]。

对于这个问题，早在1985年中国计算机科学家姚期智就已经给出回答[3]，此后的40年间似乎没有人质疑他那个看上去十分正确的结论，直到Krapivin的这篇论文。Krapivin等人提出了一种新的哈希表，并且经过对该新哈希表的分析，发现早已被定论的结论可能并不完整。本文将对哈希表的基本原理做简单介绍，为试图进一步挖掘该话题的读者提供一些背景知识。

图1 在2024年获丘吉尔奖学金即将去剑桥大学攻读硕士学位的Andrew Krapivin。 | 图源：newbrunswick.rutgers.edu

查找问题——哈希表的起源

哈希表（Hash table，也被译为散列表）是计算机科学中最重要，也是最常用的数据结构之一，它的提出是为了解决“查找问题”。所谓查找问题，就是从一堆数据里面，查询某个特定数据是否存在；如果它存在的话，它是否关联某一个特定的值。在计算机应用中，查找问题无处不在，毫不夸张地讲，我们手机、电脑上所有常用的软件，每次使用都会涉及各种各样的查找。这里很多是程序内部数据的查找，也有许多和用户直接相关，譬如在微信中根据微信号查找好友，银行App中根据用户id找到其账户余额，浏览器中查找某个页面的资源是否被缓存，等等。

那计算机系统是如何实现查找的呢？与我们人类在图书馆里找书、在快递货架上找快递的方式一样，计算机实现查找的最简单方法，其实就是挨个找：从数据开始的地方，一直查到数据结束的地方，逐个对比，看看它是不是想找的那个。当数据不多的时候，这样挨个查找没什么问题，甚至还挺快的，我们在生活中也常常这样找东西。但是当数据量很大时，这样进行遍历查找的效率就不高了。试想，我们要从一个巨大且无序的放着100万件快递的仓库里面，一件一件地找到自己的那一件，势必如大海捞针。对计算机而言也是一样的，纵使现代计算机的速度非常快，但是当数据量较大时，这样的线性查找过程很可能会成为整个程序的性能瓶颈。哈希表的提出，就是为了计算机中能够更高效地进行查找，相比上述的线性查找（或者称为挨个遍历），使用哈希表查找，可是快得多得多。事实上，使用哈希表可以在***常数时间***内完成查找。所谓常数时间，是指查找时间不随数据量变大而变大——哪怕数据量变得很大，查找也能快速地完成。

哈希表最早由IBM工程师汉斯·彼得·卢恩（Hans Peter Luhn）发明。卢恩于1896年出生于德国巴门，早年在印刷业和纺织业工作，他学识渊博、热爱发明，对登山、美食、风景画都颇为在行。20世纪30年代，卢恩已拥有众多专利，包括一件可折叠雨衣，以及“鸡尾酒神

谕”（一种可以告诉用户用现有原料可以调制哪些饮品的指南）。卢恩尤其对文本信息的存储、通信和检索感兴趣，他最终加入IBM从事相关研究[4]。1950年代，计算机技术正蓬勃发展，计算机处理的数据量也越来越大，线性的搜索方法已经不再适用于巨大的数据量。1953年1月，卢恩撰写了一份 IBM 内部备忘录，其中使用了链式的哈希方法来进行更高效的数据检索，这就是如今哈希表的原型。

哈希表的基本思想

为了更好阐明哈希表的思想和原理，我们先简单了解一下计算机的运行模型，中央处理器（CPU）、内存和程序设计（或“编程”）这三者的关系。 CPU是计算机的核心，它能够执行各种算术、逻辑运算或条件跳转的指令；CPU内部能够存储的数据很少，数据主要是放在内存之中。CPU支持随机内存访问，可以执行读取或者写入任意内存地址的指令，而一连串的CPU指令就构成了一个计算机程序，这些指令会按顺序串行执行。于是，一个典型的计算机程序的工作流可以认为是这样：1）从内存中读取数据到CPU中；2）在CPU内部完成各种计算；3）算完以后的数据再由CPU写入内存中。这个过程可以一直重复，直到任务全部完成。对比快递驿站的例子，内存好比快递柜或者货架，包裹都存放在里面；CPU是驿站里的员工，负责存放快递，执行操作；程序则是驿站老板，负责设计快递存取的策略或者方法，由员工执行。

现在我们可以找快递了，即查找问题。假设我们有一张表，该表能够把一个键（key）映射到一个值（value），并且表的条目中不能有重复的键。这里的键和值可以是任何类型，代表任何东西，为了方便理解，我们假设键和值都是整数。对这样一张想象中的表，我们定义三种操作：

- Put(key, value)：增加一个新的条目，将key映射到value；如果key已存在，则用新value覆盖原有的value。
- Get(key)：获取key对应的值，即“查找”操作。
- Delete(key)：删除key对应的条目。

这样的表又称为字典（dictionary）或关联数组（associative array）。一开始这张表是空的，假设我们按顺序执行以下操作：

- Put(1045, 1)
- Put(1056, 2)

- Put(1067, 3)

完成上述操作之后，表的状态应该如下[每一行称为一个键-值对（Key-value pair）]。对应到取快递场景中，键可代表物流公司的快递单号，值则是用户id，现在我们有了三份快递待取。

Key	Value
1045	1
1056	2
1067	3

这张表在计算机中最简单的实现方式是：Put操作时，这些key-value对依次被写入相邻的内存地址中，即用一个数组来存储（数组是内存中的数据结构，代表连续的内存空间，其大小可动态扩展，如下表所示），数组中每个元素都是一个key-value对；查找时，即Get操作，数组被从头到尾线性扫描数，依次读取其中每个元素，将其key部分与目标key对比。如果两者相同就结束查找，返回数据的value部分，否则继续扫描下一个元素。

数据：	（1045, 1）	（1056, 2）	（1067, 3）
数组地址：	1	2	3

对于数据量小的情况，前文提到的遍历方法其实并不慢，甚至可以说是最高效的。在计算机科学中，人们用***时间复杂度***（我们文中简称为复杂度）来形式化地衡量一个算法的效率或快慢程度，它描述的是计算机完成该算法所需要的基本操作的步数（或指令数），如何随着算法处理的输入大小的增长而增长。假设输入数据的大小为n（在查找问题中，n就表示表中条目的总数），时间复杂度分析试图回答：计算机完成算法所需要的基本操作步数和n是什么关系？它和n成正比（线性增长）？和n的指数成正比（指数增长）？或者是和n的对数成正比？甚至，它也许不取决于n。

下面我们以线性查找为例，尝试分析一下它需要的操作步数和输入大小n的关系。假设所有的key都有同等概率被查找，那么最快的情况，查找只需要搜索一次，即key是第一个条目；而最坏的情况下则需要搜索n次，即key是最后一个条目。平均下来则需要搜索$\frac{n(n+1)}{2n}=\frac{n+1}{2}$次，这样的一个时间复杂度我们把它记为O(n)。O是用于描述函数渐近行为的数学符号，在计算机科学中代表算法复杂度分析的渐近上界。此外，在O符号中，加性常数和

乘性常数也都被忽略了，即$O(\frac{n}{2}+\frac{1}{2})$和O(n)是一回事，因为复杂度分析关心的是算法需要的步数随着n的增长而增长的方式，O(n)于是代表随输入线性增长。所以前面我们说线性查找“比较慢”，现在可以用一个具体的符号来表示它如何慢了，即O(n)。

那有没有什么办法能够让查找更快，甚至一次就能找到呢？读者或许已经想到，Put的时候，既然key都是整数（其他类型的key其实也可以转化为整数），可以将value直接写入key所对应的数组下标，即 。对于key-value对（1045, 1），那就在数组地址1045的地方写入值1，这样查找的时候，不就可以根据key直接定位到对应的value了吗？这样只需要一步操作。当key的可能取值范围比较小的时候，我们的确可以这样做，这种一步到位的查找操作的复杂度记为O(1)。O(1)不一定代表真的只需要一步，可能是两步、三步甚至一百步，它指的是算法的复杂度和输入大小n无关。这个办法已经初步具有了哈希表的思想，但是它有一个致命问题：一般情况下key的取值范围可能会很大，比如从0到1000亿，为了能够保存所有可能的key，哪怕实际只会用到数组很小的一部分，我们也要创建一个大小为1000亿的数组。这样的方法非常浪费内存，甚至内存容量本身就不足以构建如此之大的数组。

有的读者此时可能又想到，那就对key进行取模运算！假设数组的大小只有100，我们把value存放到（key mod 100）的地址，即A[key mod 100] = value。对于key-value（1045, 1），数组地址为 45（即1045 mod 100 = 45），在这里存入值1。这样不就解决了内存大小不够用的问题吗？但是，一个新的问题又产生了：两个不同的key在取模之后可能会对应到同一个地址，1045和2045对100取模后都是45，这样就产生了冲突。

其实到了这里，我们已经进入了哈希表的世界。哈希表就是采用类似的思想，使用一个哈希函数（或称散列函数）为每一个key计算出一个哈希值（hash value），再对数组的大小取模。由于数组的大小是有限的，实际上冲突总是会发生的，而一个好的哈希函数会使冲突不那么频繁。如何解决冲突，就是哈希表设计的一个核心问题。

哈希表的冲突解决对策

解决哈希表冲突的第一个方法称为链接法（chaining），其基本原理是使用数组来存储内存地址（也称为“指针”）。注意，这里数组存储的并不是key-value对（元素），而是一个指向内存某块空间的地址。真正的key-value对本身被存储在数组之外的其他内存区域，它们的内存地址可能是随机的。每当增加一个新的元素，系统都会申请一块新的内存区域用来存入数据key、value，以及下一个元素的内存地址（即一个指针）。如果下一个元素不存在，指针的内容则为空，在图2中显示为斜杠。每一个数组槽位的起始状态都是空，随着

第一个元素被哈希（hash）到这个槽位（并且被存储在内存的某块地方），数组槽位本身会被更新为这个元素所在的内存地址。若另一新的元素被哈希到了同一槽位，它的指针则指向槽位中原来的元素，同时让槽位本身指向新的元素。于是，具有同样哈希值的元素们就通过这样的链式结构链接了起来。这就好比在快递柜中，每一个包裹上又标明了下一个包裹存放的地址，根据这个地址你就能按图索骥再找到下一个包裹，以此类推，这些包裹就相当于“链接”起来了。

同时我们也可以看到，假如所有的元素都被哈希到了同样的一个槽位，即整个数组里面只有一个槽位被使用了，那查找某个特定元素的复杂度就是O(n)（假设一共有n个元素），因为链表的长度将会是n，此时的链接法就退化成我们前面提到的线性搜索了。这也说明了一个好的哈希函数的重要性，它会使得这些元素更加分散，减少冲突发生的频率，使得查找变得更高效。

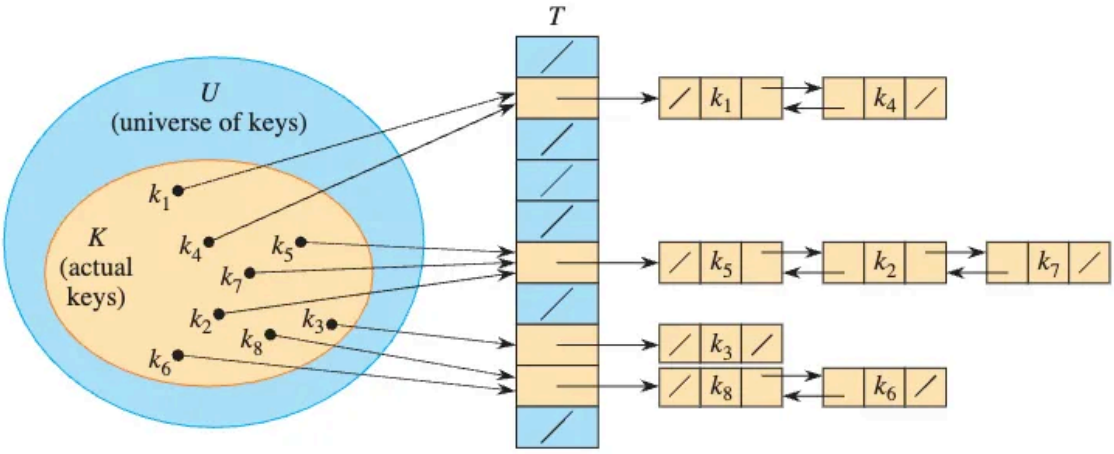

图2 基于链接法的哈希表，哈希值相同的元素们通过指针链接起来。图源：《算法导论》（第四版）[5]。

假设我们的哈希函数会将元素分散得比较均匀，那此时链表法的效率如何呢？首先，插入一个新元素是比较快的，可以直接插在链式结构（链表）最前面，这个过程可以在常数时间内完成，记为O(1)，即完成操作的快慢程度不取决于n。这里的O(1)里面也包含了计算哈希值的时间。对于查找的情况，我们考虑两种不同情况下的时间复杂度：key已存在（成功查找）和key不存在（不成功查找）。对于key不存在的情况，搜索则需要把对应槽位的整个链表都遍历一遍，这个过程的快慢取决于链表的长度。假定表中的总条目或元素数为n，总槽位数为m，则每个槽位的链表的平均长度为$\alpha = \frac{n}{m}$，α也被称为负载因子（Load factor）；不成功的查找则可以在O(1+α)时间完成。事实上，可以证明成功查找的复杂度也是O(1+α)。这个复杂度意味着，当n和m大小相当时，O(1+α)=O(1)，即可以在常数时间内完成查找，这就比前面线性查找的复杂度O(n)快得多了。（但应注意的是，这里的更

快是渐进意义上的，当n足够大时，O(1)更快；而在实际应用中，O(1)的步骤可能很大，而n足够小，这时反而是线性查找O(n)是更快的。）

不过链接法也有它自己的问题：从内存的使用上来讲，链表并不高效。因为元素所处的内存位置并不连续，而且需要额外的空间来存储指向下一个元素的指针，对于计算机而言，访问连续的内存地址会比访问随机的不相邻的地址更高效。所以在实际应用中，更多是采用另外一种方法来解决冲突，称为开放寻址法（open addressing），它也是我们开篇提到的那篇新论文所讨论的主题。

开放寻址法的思想很简单：直接将数据本身存在槽位中。如果我们发现一个槽已经被使用了，那就试探性地看看附近的一些槽位是否能用，能用的话就放在里面。事实上，开放寻址法会按照某一个预设的规则来尝试一系列的位置，直到找到一个空的位置为止，这样的一系列位置被称为探测序列（probe sequence）。当我们要查找一个元素的时候，系统会按照这个元素所对应的探测序列来挨个查找，直到找到元素本身，或者发现它不存在。最简单的探测序列就是挨个往后一个一个找，这也被称为线性探测（linear probing）；除此之外，还有二次探测（Quadratic probing），即探测序列是探测次数i的一个二次函数。在开放寻址中，由于数据放在数组本身之中，没有使用额外的内存空间，内存地址访问也更加连续，因此实际效率更高。

早在1954年，IBM工程师吉恩·阿姆达尔（Gene Amdahl）和他的同事们就在IBM 701计算机上使用了开放寻址的哈希表。此外，苏联计算机科学家、编程语言先驱安德烈·彼得罗维奇·叶尔绍夫（Andrey Ershov）也独立提出了线性探测的想法，在此后，开放寻址法成为了哈希表实现的经典方法。1985年，姚期智给出证明：无论采用何种探测序列，开放寻址法查找已存在key的最优时间复杂度为$O(\ln\left(\frac{1}{1-\alpha}\right))$，这里的α代表负载因子，或者哈希表满载的程度，且α<1。此后，这个结论似乎也成为了学界的共识。所以当大四学生Krapivin拿着他自己设计的哈希表去找算法老师，并声称其查找效率比姚期智证明的结论还要更高效时，他的老师是怀疑的。

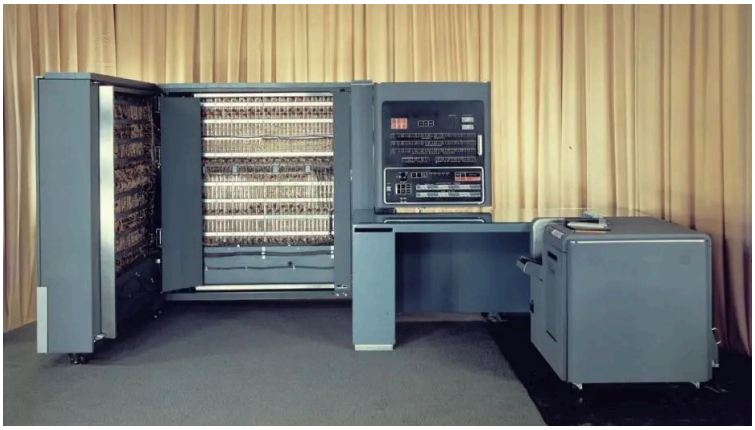

图3 1954年，IBM工程师吉恩·阿姆达尔（Gene Amdahl）和他的同事们在IBM 701计算机上使用了开放寻址的哈希表。 | 图源：IBM

开放寻址法的过程

为了更好地阐释开放寻址法的时间复杂度，我们先说明开放寻址法的基本运作过程，即它如何实现Put、Get和Delete操作。

如我们前面所说，开放寻址法也采用一个数组，并将数据（key-value对）直接存入数组中。除了数据本身，数组中还保存了每个槽位的状态（status），一共有三种可能的状态：“空”（empty）、“被使用”（used）和“被删除”（deleted）。

假设数组名为A，其大小为m，并且使用线性探测的开放寻址。Put和Get的实现方法分别如下：

> Put(key, value)
>
> 计算key的哈希值，并对m取模，得到h。依次查看槽位A[h]、A[h+1]、A[h+2]...，如果槽位的状态为“空”，或者状态为“被使用”且其元素对应的key和我们要找的一致，则将我们的key和value写入这个槽，并将其状态更新为“被使用”（如果本来状态就是被使用也可以不用更新），否则继续搜寻。

上述过程考虑了两种情况：如果key已存在，我们会找到它的那个槽位；如果未存在，则会找到第一个空槽位。

Get的过程是类似的，也需要这样的搜寻过程，完整步骤如下：

> Get(key):
>
> 计算key的哈希值，并对m取模，得到h。依次查看槽位A[h]、A[h+1]、A[h+2]...，如果槽位的状态为“空”，则返回“未找到”；如果槽位状态为“被使用”且其元素对应的key和我们要找的一致，则返回槽位元素的value，否则继续搜寻。

> 注：这里的Put和Get实现把成功查找和不成功查找合并在一起了，即同一个方法，既能应对key存在，也能应对key不存在的情况。而在复杂度分析中，为了简化分析，我们会分别分析成功查找和不成功查找的复杂度。

Delete的过程也需要进行搜索，另外它还需要用到第三个槽位状态：“被删除”。为什么delete不能直接将搜索到的槽位的状态设为“空”呢？因为在Put和Get搜索元素的过程中，只要碰到空槽位，搜索就结束了。因此，如果Delete 将某个槽位的状态设为空，则可能会导致某个元素被存在后面但搜寻却提前结束了。也就是说，Put和Get不再能正常运行，可能出现元素其实存在但Get操作却返回不存在的情况。而如果Delete将槽位的状态更新成“被删除”，则不会影响元素搜索的过程以及Put和Get的正常运行，即碰到“被删除”后依然继续往后搜索。Delete的完整步骤如下：

Delete(key):

计算key的哈希值，并对m取模，得到h。依次查看槽位A[h]、A[h+1]、A[h+2]...，如果槽位的状态为“空”，则返回“未找到”；如果槽位状态为“被使用”且其元素对应的key和我们要找的一致，则更新槽位状态为“被删除”然后返回“成功删除”，否则继续搜寻。

最后让我们来看一个具体的例子。假设哈希函数就返回key本身，数组大小为100，然后我们依次进行如下操作：

- Put(1045, 1)
  - 1045对100取模为45，将元素写入数组槽位45，并更新槽位状态为“被使用”；
- Put(1056, 2)
  - 1056对100取模为56，将元素写入数组槽位56，并更新槽位状态为“被使用”；
- Put(1067, 3)
  - 1067对100取模为67，将元素写入数组槽位67，并更新槽位状态为“被使用”；
- Put(2045, 4)
  - 2045对100取模为45，发现槽位45的状态为“被使用”，于是跳过45；探测槽位46，发现其状态为空，则将元素写入数组槽位46，并更新其槽位状态为“被使用”；
- Delete(1056)
  - 1056对100取模为56，发现槽位56的状态为“被使用”，且其存储的元素的key等于1056，则成功删除数据，并将槽位状态更新为“被删除”；

最后哈希表的状态将会是下面这样的：

地址	槽位状态	数据
....	Empty	
45	Used	(1045, 1)
46	Used	(2045, 4)

47	Empty	
...	Empty	
56	Deleted	(1056, 2)
...	Empty	
67	Used	(1067, 3)

最后再留给读者一个小练习，假如现在我们要进行操作Put(3045, 5)，这个元素会被放在何处？表的状态会变成什么样呢？之后如果再 Get(3045) 又会是怎么样的一个过程呢？

使用开放寻址法的效率取决于负载因子（即表的占用率）和探测序列。试想假如表中一半的槽位都被占用了，同时探测序列会使得每个槽位都会均匀地被探测到，那平均而言找到一个空的槽位则需要两次探测。这有点像在停车场找车位，假如50%的车位都已经被停满了，我们的查找也是均匀的，那平均而言找到一个车位就需要试两次。

更宽泛地讲，在假定独立均匀排列哈希（independent uniform permutation hashing）的情况下，即每一个key所关联的探测序列是所有哈希表索引{0,1,2,...,n-1}的一个随机排列且所有的排列是均匀分布的，寻找一个空的槽位（譬如不成功地查找或者插入新元素）所需的平均探测次数为1/(1-α)。假如负载因子α为90%，则平均需要探测10次。可以看出，随着负载因子α靠近1，需要的探测次数会急剧增多。所以在实际的哈希表实现中，当负载因子超过某个阈值（一个常用的阈值是2/3）就会触发扩容，即创建一个更大的表，然后将所有条目重新哈希到新的表中，使得后续的插入或者查找能更快。

对于成功查找的情况（即key已存在），可以证明平均探测复杂度为$O(\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right))$，为什么这个和前面的1/(1-α)不一样呢？一个直觉就是，成功查找只需要碰到对应的key就返回了，而不成功的查找或插入新元素得搜索到第一个空槽才能结束。这里证明的思路是，成功查找到一个key所需要的探测次数和之前插入它的时候经历过的探测次数相等，而这个次数则取决于当时插入时候的负载因子。假设键k是第i+1个被插入的键，那在它被插入时表的负载因子就是$\frac{i}{m}$，代入公式1/(1-α)，就得到$\frac{m}{m-i}$。假设此时表中一共有n个槽位被占用，即n=mα，我们对所有的键进行求和取平均（经过n次插入），即$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i}$。将m提出后，这个求和项其实是调和级数的部分和，可以证明其最终的结果$\leq\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})$（见参考文献[3]）。这个结果意味着，当哈希表一半满的时候，成功查找所需的平均探测次数仅为1.387；而当哈希表90%满的时候，平均探测次数也只有2.559。

在其1985年的论文*Uniform Hashing Is Optimal*中，姚期智证明了独立均匀排列哈希的假设（即我们上面的结论），带来的成功查找（record retrieval）所需的探测复杂度就是最优的，即不管使用何种哈希函数和探测序列，查找已有的key的时间复杂度至少为$O(\ln\left(\frac{1}{1-\alpha}\right))$。然而姚期智当时给出的最优结论，是默认假定了贪婪的搜索策略的，正如前文介绍，它代表发现第一个空槽位后马上返回。姚期智在论文的最末尾，也留下了一个公开猜想：对于插入新的key或不成功的查找，Uniform hashing实现的复杂度即$O(\frac{1}{1-\alpha})$是否也是最优的?

在Krapivin等人的新论文中，作者们首先回顾了这一问题的历史背景，然后给出了他们的结论：

- 在不重排的情况下（考虑非贪婪的策略），开放寻址哈希表可以实现O(1)的均摊期望探测复杂度和$O(\log\frac{1}{1-\alpha})$的最坏情况期望探测复杂度；在姚期智的术语中，这两个复杂度分别对应成功查找和不成功查找（或插入）的时间复杂度。作者们也指出，为了实现比姚期智的结论更低的复杂度，非贪婪的策略是关键。
- 对于采用贪婪策略开放寻址的哈希表，成功查找的最优复杂度已被姚期智证明；然而插入的复杂度可以达到$O(\log^2\frac{1}{1-\alpha})$，这显然是比姚期智猜想的$O(\frac{1}{1-\alpha})$更优的结果。由此，姚期智的猜想被证否。

论文的具体内容我们不再展开讨论，想进一步深入了解的读者可以自行阅读。

结语

计算机科学是一个充满惊喜的领域，即使是几十年前看似被彻底解决的问题，依然可能藏着未被发现的真相。Andrew Krapivin的研究提醒我们，科学的进步并不总是沿着既定的轨道前行，有时候，一点好奇心和深入地思考就能带来新的突破。更重要的是，这样的突破不只是资深研究者的专利，本科生同样可以做到——只要你勇于挑战权威，敢于提出问题，并愿意投入时间去探索答案。也许下一个改写教科书的发现，就来自屏幕前的你！

参考文献

[1] Tiny Pointers, Michael A. Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, Guido Tagliavini, 2021

[2] Optimal Bounds for Open Addressing Without Reordering, Martin Farach-Colton, Andrew Krapivin, William Kuszmaul, 2024

[3] Uniform Hashing Is Optimal, Andrew Yao, 1985

[4] Hans Peter Luhn and the Birth of the Hashing Algorithm, Hallam Stevens
[5] Introduction to Algorithms, fourth edition, C.L.R.S., 2022

科 普 中 国
CHINA SCIENCE COMMUNICATION

相关阅读

1 2900万，没了！——虚拟币世界的真实战斗
2 姚期智2021年京都奖演讲全文：计算机科学之旅
3 是什么让他成为现代计算机之父？丨纪念冯·诺伊曼诞辰120周年（下）
4 春运抢票，原来售票系统背后的技术这么精妙！
5 理查德·汉明：不想当数学老师的计算机科学家得不了图灵奖

近期推荐

1 顶尖华人学者王晓峰遭突击搜查已失联，“这一切都不正常”
2 中国完成全球首例基因编辑猪肝脏移植人体，我们就快用上猪器官了吗?
3 大道至简VS多者异也，物理有机化学通往何处?
4 当一颗发霉的哈密瓜成为人类救世主
5 理论物理新方向：用高一维的拓扑序来全息理解量子场论

特 别 提 示

1. 进入『返朴』微信公众号底部菜单“精品专栏”，可查阅不同主题系列科普文章。
2. 『返朴』提供按月检索文章功能。关注公众号，回复四位数组成的年份+月份，如“1903”，可获取2019年3月的文章索引，以此类推。

版权说明：欢迎个人转发，任何形式的媒体或机构未经授权，不得转载和摘编。转载授权请在「返朴」微信公众号内联系后台。

找不到《返朴》了？快加星标！！

长按下方图片关注「返朴」，查看更多历史文章

微信实行乱序推送，常点“在看”，可防失联