20240402-Week7-查缺补漏（1）

Updated 0100 GMT+8 Apr 7, 2024

2024 spring, Complied by Hongfei Yan

Logs：

- 4月3日的月考六个题目，会占用“2024spring每日选做”连续三天（4月2～4日）的6个slots。
- 2024/3/31 说明：周二4月2日，课堂上进行笔试（电脑上完成），考题在canvas平台，届时开放。

客观题目包括：选择、判断、填空（改造成选择）；主观题目可能没有，因为如果有的话，可能需要拍照草稿纸上传了。

请大家带笔记本（pad没有试过，估计也可以），草稿纸，cheat paper，个人独立完成。

预计考60分题目，包括15个选择题30分，10个判断题10分，10个填空题（改造为选择形式）20分。按照期末考试120分钟计算，60分题目，需要 120 * 60 / 100 = 72mins，即大约是1.5课时（50+25 = 75 mins）。

不计分。笔试后再上课，可能线下/线上同学顺序来讲解题目，每人一个。讲解时候（最好到讲台），先介绍下自己的姓名、院系，复述一下题面，然后讲解该题目答案。

笔试范围：除了图没有，其他的都有，开考之前先把 散列表 讲了。包括：栈、队列、优先队列、排序、散列表、哈夫曼、调度场、各种树的定义、操作（如各种遍历），时间复杂度等。学期过半，课程只剩下图了。

- 2024/3/18 提前进入战时状态。2024年4月3日 寒食节月考，机房15:08～17:00。

尽量按位就座，错过的自己计时完成。题目会留为当周作业。
范围主要涉及：排序、栈、队列、树。目前不考虑KMP, Trie, Segment Tree。

题目难度参考CF, 1000~1100为Easy, 1200~1300为Medium，1400～1500为Tough。

一、散列表的查找

1.1 散列表的基本概念

> 参考：数据结构（C语言版 第2版）(严蔚敏)，第7章 查找

前6周讨论了基于线性结构、树表结构的查找方法，这类查找方法都是以关键字的比较为基础的。

> 线性表是一种具有相同数据类型的有限序列，其特点是每个元素都有唯一的直接前驱和直接后继。换句话说，线性表中的元素之间存在明确的线性关系，每个元素都与其前后相邻的元素相关联。

> 线性结构是数据结构中的一种基本结构，它的特点是数据元素之间存在一对一的关系，即除了第一个元素和最后一个元素以外，其他每个元素都有且仅有一个直接前驱和一个直接后继。线性结构包括线性表、栈、队列和串等。
>
> 因此，线性表是线性结构的一种具体实现，它是一种最简单和最常见的线性结构。

在查找过程中只考虑各元素关键字之间的相对大小，记录在存储结构中的位置和其关键字无直接关系，其查找时间与表的长度有关，特别是当结点个数很多时，查找时要大量地与无效结点的关键字进行比较，致使查找速度很慢。如果能在元素的存储位置和其关键字之间建立某种直接关系，那么在进行查找时，就无需做比较或做很少次的比较，按照这种关系直接由关键字找到相应的记录。这就是散**列查找法**（Hash Search）的思想，它通过对元素的关键字值进行某种运算，直接求出元素的地址，即使用关键字到地址的直接转换方法，而不需要反复比较。因此，散列查找法又叫杂凑法或散列法。

下面给出散列法中常用的几个术语。

(1) **散列函数和散列地址**：在记录的存储位置p和其关键字 key 之间建立一个确定的对应关系 H，使p=H(key)，称这个对应关系H为散列函数，p为散列地址。

(2) **散列表**：一个有限连续的地址空间，用以存储按散列函数计算得到相应散列地址的数据记录。通常散列表的存储空间是一个一维数组，散列地址是数组的下标。

(3) **冲突和同义词**：对不同的关键字可能得到同一散列地址,即 key1≠key2,而 H(key1) = H(key2) 这种现象称为冲突。具有相同函数值的关键字对该散列函数来说称作同义词，key1与 key2 互称为同义词。

例如，在Python语言中，可以针对给定的关键字集合建立一个散列表。假设有一个关键字集合为`S1`，其中包括关键字`main`, `int`, `float`, `while`, `return`, `break`, `switch`, `case`, `do`。为了构建散列表，可以定义一个长度为26的散列表`HT`，其中每个元素是一个长度为8的字符数组。假设我们采用散列函数`H(key)`，该函数将关键字`key`中的第一个字母转换为字母表`{a,b,…,z}`中的序号（序号范围为0~25），即`H(key) = ord(key[0]) - ord('a')`。根据此散列函数构造的散列表`HT`如下所示：

```
HT = [['' for _ in range(8)] for _ in range(26)]
```

其中，假设关键字`key`的类型为长度为8的字符数组。根据给定的关键字集合和散列函数，可以将关键字插入到相应的散列表位置。

表1

0	1	2	3	4	5	...	8	...	12	...	17	18	...	22	...	25
	break	case	do		float		int		main		return	switch		while		

假设关键字集合扩充为:

S2 = S1 + {short, default, double, static, for, struct}

如果散列函数不变，新加人的七个关键字经过计算得到：H(short)=H(static)=H(struct)=18，H(default)=H(double)=3，H(for)=5，而 18、3 和5这几个位置均已存放相应的关键字，这就发生了冲突现象，其中,switch、short、static 和 struct 称为同义词；float 和 for 称为同义词；do、default 和 double 称为同义词。

集合S2中的关键字仅有 15 个，仔细分析这 15个关键字的特性，应该不难构造一个散列函数避免冲突。但在实际应用中，理想化的、不产生冲突的散列函数极少存在，这是因为通常散列表中关键字的取值集合远远大于表空间的地址集。例如，高级语言的编译程序要对源程序中的标识符建立一张符号表进行管理，多数都采取散列表。在设定散列函数时，考虑的查找关键字集合应包含所有可能产生的关键字，不同的源程序中使用的标识符一般也不相同，如果此语言规定标识符为长度不超过8的、字母开头的字母数字串，字母区分大小写，则标识符取值集合的大小为: $C_{52}^{1} \times C_{62}^{7} \times 7! = 1.09 \times 10^{12}$

而一个源程序中出现的标识符是有限的,所以编译程序将散列表的长度设为 1000 足矣。于是要将多达 10”个可能的标识符映射到有限的地址上，难免产生冲突。通常，散列函数是一个多对一的映射，所以冲突是不可避免的，只能通过选择一个“好”的散列函数使得在一定程度上减少冲突。而一旦发生冲突，就必须采取相应措施及时予以解决。综上所述，散列查找法主要研究以下两方面的问题:

(1) 如何构造散列函数；
(2) 如何处理冲突。

1.2 散列函数的构造方法

构造散列函数的方法很多，一般来说，应根据具体问题选用不同的散列函数，通常要考虑以下因素:

(1) 散列表的长度;
(2) 关键字的长度;
(3) 关键字的分布情况;
(4) 计算散列函数所需的时间;
(5) 记录的查找频率。

构造一个“好”的散列函数应遵循以下两条原则:(1)函数计算要简单，每一关键字只能有一个散列地址与之对应;(2)函数的值域需在表长的范围内，计算出的散列地址的分布应均匀，尽可能减少冲突。下面介绍构造散列函数的几种常用方法。

1.数字分析法

如果事先知道关键字集合，且每个关键字的位数比散列表的地址码位数多，每个关键字由n位数组成，如k1k2,...kn，则可以从关键字中提取数字分布比较均匀的若干位作为散列地址。

例如，有 80个记录，其关键字为8位十进制数。假设散列表的表长为100，则可取两位十进制数组成散列地址，选取的原则是分析这80个关键字，使得到的散列地址尽量避免产生冲突。假设这 80个关键字中的一部分如下所列:

⋮

8	1	3	4	6	5	3	2
8	1	3	7	2	2	4	2
8	1	3	8	7	4	2	2
8	1	3	0	1	3	6	7
8	1	3	2	2	8	1	7
8	1	3	3	8	9	6	7
8	1	3	5	4	1	5	7
8	1	3	6	8	5	3	7
8	1	4	1	9	3	5	5

⋮

① ② ③ ④ ⑤ ⑥ ⑦ ⑧

对关键字全体的分析中可以发现:第①、②位都是“81”，第③位只可能取3或 4，第⑧位可能取 2、5或7，因此这4位都不可取。由于中间的4位可看成是近乎随机的，因此可取其中任意两位，或取其中两位与另外两位的叠加求和后舍去进位作为散列地址。

数字分析法的适用情况：事先必须明确知道所有的关键字每一位上各种数字的分布情况。

在实际应用中，例如，同一出版社出版的所有图书，其ISBN号的前几位都是相同的，因此，若数据表只包含同一出版社的图书，构造散列函数时可以利用这种数字分析排除ISBN 号的前几位数字。

2.平方取中法

通常在选定散列函数时不一定能知道关键字的全部情况，取其中哪几位也不一定合适，而一个数平方后的中间几位数和数的每一位都相关，如果取关键字平方后的中间几位或其组合作为散列地址，则使随机分布的关键字得到的散列地址也是随机的，具体所取的位数由表长决定。平方取中法是一种较常用的构造散列函数的方法。

例如，为源程序中的标识符建立一个散列表，假设标识符为字母开头的字母数字串。假设人为约定每个标识的内部编码规则如下：把字母在字母表中的位置序号作为该字母的内部编码，如 I 的内部编码为 09，D 的内部编码为 04，A 的内部编码为 01。数字直接用其自身作为内部编码,如 1的内部编码为 01，2 的内部编码为 02。根据以上编码规则，可知“IDA1”的内部编码为09040101，同理可以得到“IDB2”、“XID3”和“YID4”的内部编码。之后分别对内部编码进行平方运算，再取出第7位到第9位作为其相应标识符的散列地址，如表 2所示。

表2 标识符及其散列地址

标识符	内部编码	内部编码的平方	散列地址
IDA1	09040101	081723426090201	426
IDB2	09040202	081725252200804	252
XID3	24090403	580347516702409	516
YID4	25090404	629528372883216	372

3.折叠法将

关键字分割成位数相同的几部分（最后一部分的位数可以不同），然后取这几部分的叠加和（舍去进位）作为散列地址，这种方法称为折叠法。根据数位叠加的方式，可以把折叠法分为移位叠加和边界叠加两种。移位叠加是将分割后每一部分的最低位对齐，然后相加;边界叠加是将两个相邻的部分沿边界来回折看，然后对齐相加。

例如，当散列表长为 1000 时，关键字key=45387765213，从左到右按3 位数一段分割，可以得到 4个部分:453、877、652、13。分别采用移位叠加和边界叠加，求得散列地址为 995 和914，如图 1 所示。

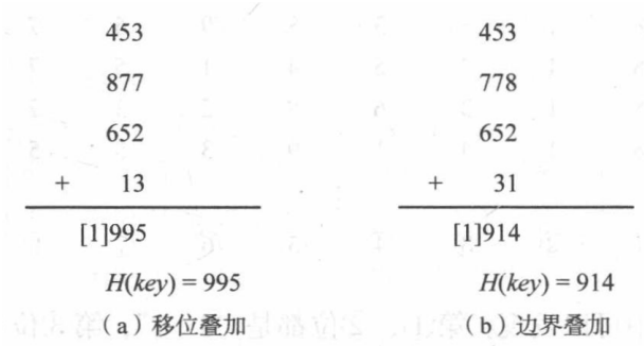

图 1由折叠法求得散列地址

折叠法的适用情况：适合于散列地址的位数较少，而关键字的位数较多，且难于直接从关键字中找到取值较分散的几位。

4.除留余数法

假设散列表表长为 m，选择一个不大于m 的数p，用p去除关键字，除后所得余数为散列地址，即
H(key) = key%p

这个方法的关键是选取适当的p，一般情况下，可以选p为小于表长的最大质数。例如，表长m=100，可取p=97。

除留余数法计算简单，适用范围非常广，是最常用的构造散列函数的方法。它不仅可以对关键字直接取模，也可在折叠、平方取中等运算之后取模，这样能够保证散列地址一定落在散列表的地址空间中。

1.3 处理冲突的方法

选择一个“好”的散列函数可以在一定程度上减少冲突，但在实际应用中，很难完全避免发生冲突，所以选择一个有效的处理冲突的方法是散列法的另一个关键问题。创建散列表和查找散列表都会遇到冲突，两种情况下处理冲突的方法应该一致。下面以创建散列表为例，来说明处理冲突的方法。

处理冲突的方法与散列表本身的组织形式有关。按组织形式的不同，通常分两大类:开放地址法和链地址法。

1.开放地址法

开放地址法的基本思想是：把记录都存储在散列表数组中，当某一记录关键字 key 的初始散列地址 H0 = H(key)发生冲突时，以 H0 为基础，采取合适方法计算得到另一个地址 H1，如果 H1 仍然发生冲突，以 H1 为基础再求下一个地址 H2，若 H2 仍然冲突，再求得 H3。依次类推，直至 Hk 不发生冲突为止，则 Hk 为该记录在表中的散列地址。

这种方法在寻找 ”下一个” 空的散列地址时，原来的数组空间对所有的元素都是开放的所以称为开放地址法。通常把寻找 “下一个” 空位的过程称为**探测**，上述方法可用如下公式表示：

Hi=(H(key) +di)%m　　i=1,2,...,k(k≤m-l)

其中，H(key)为散列函数，m 为散列表表长，d为增量序列。根据d取值的不同，可以分为以下3种探测方法。

(1) 线性探测法
di = 1, 2, 3, ..., m-1

这种探测方法可以将散列表假想成一个循环表，发生冲突时，从冲突地址的下一单元顺序寻找空单元，如果到最后一个位置也没找到空单元，则回到表头开始继续查找，直到找到一个空位，就把此元素放入此空位中。如果找不到空位，则说明散列表已满，需要进行溢出处理。

(2) 二次探测法
$d_i = 1^2, -1^2, 2^2, -2^2, 3^2, \ldots, +k^2, -k^2\ (k \leq m/2)$

(3)伪随机探测法

di = 伪随机数序列
例如，散列表的长度为 11，散列函数 H(key)=key%11，假设表中已填有关键字分别为 17、60、29 的记录，如图 7.29(a)所示。现有第四个记录，其关键字为38，由散列函数得到散列地址为 5，产生冲突。

若用线性探测法处理时，得到下一个地址6，仍冲突；再求下一个地址7，仍冲突；直到散列地址为8的位置为“空”时为止，处理冲突的过程结束，38填入散列表中序号为8的位置，如图2(b)所示。

若用二次探测法，散列地址5冲突后，得到下一个地址6，仍冲突；再求得下一个地址 4，无冲突，38填入序号为4的位置，如图 2(c)所示。

若用伪随机探测法，假设产生的伪随机数为9，则计算下一个散列地址为(5+9)%11=3，所以38 填入序号为3 的位置，如图 2(d)所示。

图 2 用开放地址法处理冲突时，关键字为38的记录插入前后的散列表

从上述线性探测法处理的过程中可以看到一个现象:当表中i, i+1, i+2位置上已填有记录时，下一个散列地址为i、i+1、i+2和i+3的记录都将填入i+3的位置，这种在处理冲突过程中发生的两个第一个散列地址不同的记录争夺同一个后继散列地址的现象称作“**二次聚集**”(或称作“**堆积**”)，即在处理同义词的冲突过程中又添加了非同义词的冲突。

可以看出，上述三种处理方法各有优缺点。线性探测法的优点是：只要散列表未填满，总能找到一个不发生冲突的地址。缺点是：会产生“二次聚集”现象。而二次探测法和伪随机探测法的优点是：可以避免“二次聚集”现象。缺点也很显然：不能保证一定找到不发生冲突的地址。

2.链地址法

链地址法的基本思想是：把具有相同散列地址的记录放在同一个单链表中，称为同义词链表。有 m个散列地址就有 m 个单链表，同时用数组 HT[0...m-1]存放各个链表的头指针，凡是散列地址为i的记录都以结点方式插入到以 HT[] 为头结点的单链表中。

【例】 已知一组关键字为 (19,14,23，1，68，20，84,27，55，11，10，79)，设散列函数H(key)=key%13，用链地址法处理冲突，试构造这组关键字的散列表。

由散列函数 H(key)=key %13 得知散列地址的值域为 0~12，故整个散列表有 13 个单链表组成，用数组 HT[0..12] 存放各个链表的头指针。如散列地址均为1的同义词 14、1、27、79 构成一个单链表，链表的头指针保存在 HT[1] 中，同理，可以构造其他几个单链表，整个散列表的结构如图 3 所示。

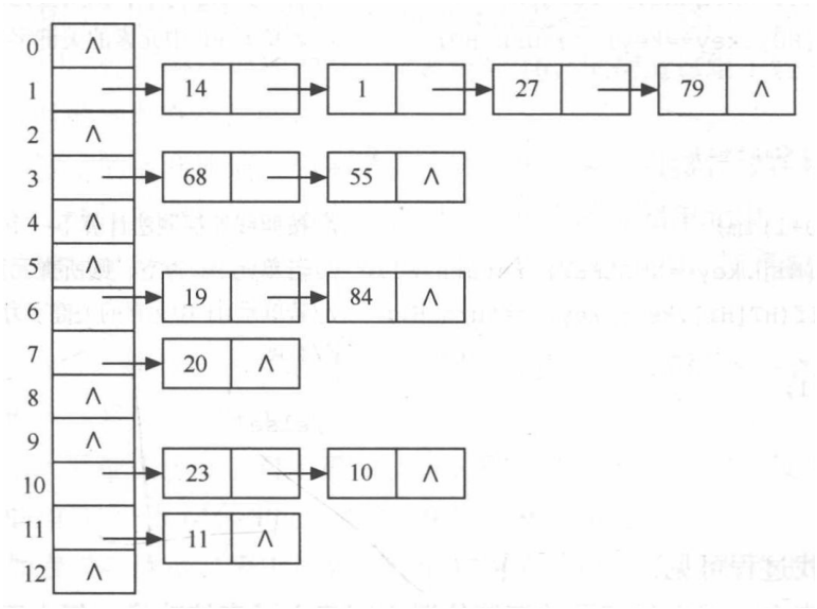

图 3 用链地址法处理冲突时的散列表

这种构造方法在具体实现时，依次计算各个关键字的散列地址，然后根据散列地址将关键字插入到相应的链表中。

> 处理散列表冲突的常见方法包括以下几种：
>
> 1. 链地址法（Chaining）：使用链表来处理冲突。每个散列桶（哈希桶）中存储一个链表，具有相同散列值的元素会链接在同一个链表上。

2. 开放地址法（Open Addressing）：
    - 线性探测（Linear Probing）：如果发生冲突，就线性地探测下一个可用的槽位，直到找到一个空槽位或者达到散列表的末尾。
    - 二次探测（Quadratic Probing）：如果发生冲突，就使用二次探测来查找下一个可用的槽位，避免线性探测中的聚集效应。
    - 双重散列（Double Hashing）：如果发生冲突，就使用第二个散列函数来计算下一个槽位的位置，直到找到一个空槽位或者达到散列表的末尾。
3. 再散列（Rehashing）：当散列表的**装载因子（load factor）**超过一定阈值时，进行扩容操作，重新调整散列函数和散列桶的数量，以减少冲突的概率。
4. 建立公共溢出区（Public Overflow Area）：将冲突的元素存储在一个公共的溢出区域，而不是在散列桶中。在进行查找时，需要遍历溢出区域。

这些方法各有优缺点，适用于不同的应用场景。选择合适的处理冲突方法取决于数据集的特点、散列表的大小以及性能需求。

双重散列（Double Hashing）是一种处理散列表冲突的方法，它使用两个散列函数来计算冲突时下一个可用的槽位位置。下面是双重散列的一个示例：

假设有一个散列表，大小为10，使用双重散列来处理冲突。我们定义两个散列函数：

1. 第一个散列函数 `hash1(key)`：将关键字 `key` 转换为散列值，使用一种合适的散列算法，比如取模运算。
2. 第二个散列函数 `hash2(key)`：将关键字 `key` 转换为一个正整数，在本例中，我们使用简单的散列函数 `hash2(key) = 7 - (key % 7)`。

现在，我们通过以下步骤来插入一个关键字 `key` 到散列表中：

1. 使用第一个散列函数 `hash1(key)` 计算关键字 `key` 的初始散列值 `hash_value = hash1(key)`。
2. 如果散列表中的槽位 `hash_value` 是空的，则将关键字 `key` 插入到该槽位中。
3. 如果槽位 `hash_value` 不为空，表示发生了冲突。在这种情况下，我们**使用第二个散列函数 `hash2(key)` 来计算关键字 `key` 的步长（step）**。
4. 通过计算 `step = hash2(key)`，我们将跳过 `step` 个槽位，继续在散列表中查找下一个槽位。
5. 重复步骤 3 和步骤 4，直到找到一个空槽位，将关键字 `key` 插入到该槽位中。

如果散列表已满而且仍然无法找到空槽位，那么插入操作将失败。

双重散列使用两个散列函数来计算步长，这样可以避免线性探测中的聚集效应，提高散列表的性能。每个关键字都有唯一的步长序列，因此它可以在散列表中的不同位置进行探测，减少冲突的可能性。

笔试例题：

有一个散列表如下图所示，其散列函数为 h(key)=key mod 13，该散列表使用再散列函数 H2(Key)=Key MOD 3 解决碰撞，问从表中检索出关键码 38 需进行几次比较（ B ）。

0	1	2	3	4	5	6	7	8	9	10	11	12
26	38			17			33		48			25

A： 1 B：2 C: 3 D： 4

1.5 程序实现

字符串构建简单的散列函数。针对异序词，这个散列函数总是得到相同的散列值。要弥补这一点，可以用字符位置作为权重因子，

```
def hash(a_string, table_size):
    sum = 0
    for pos in range(len(a_string)):
        sum = sum + (pos+1) * ord(a_string[pos])

    return sum%table_size

print(hash('abba', 11))
```

使用两个列表创建HashTable类，以此实现映射抽象数据类型。其中，名为slots的列表用于存储键，名为data的列表用于存储值。两个列表中的键与值一一对应。在本节的例子中，散列表的初始大小是11。尽管初始大小可以任意指定，但选用一个素数很重要，这样做可以尽可能地提高冲突处理算法的效率。

hashfunction实现了简单的取余函数。处理冲突时，采用“加1”再散列函数的线性探测法。put函数假设，除非键已经在self.slots中，否则总是可以分配一个空槽。该函数计算初始的散列值，如果对应的槽中已有元素，就循环运行rehash函数，直到遇见一个空槽。如果槽中已有这个键，就用新值替换旧值。

同理，get函数也先计算初始散列值。如果值不在初始散列值对应的槽中，就使用rehash确定下一个位置。注意，第46行确保搜索最终一定能结束，因为不会回到初始槽。如果遇到初始槽，就说明已经检查完所有可能的槽，并且元素必定不存在。

HashTable类的最后两个方法提供了额外的字典功能。重载`__getitem__`和`__setitem__`，以通过[]进行访问。这意味着创建HashTable类之后，就可以使用熟悉的索引运算符了。

```
class HashTable:
    def __init__(self):
        self.size = 11
        self.slots = [None] * self.size
        self.data = [None] * self.size

    def put(self,key,data):
        hashvalue = self.hashfunction(key,len(self.slots))

        if self.slots[hashvalue] == None:
            self.slots[hashvalue] = key
            self.data[hashvalue] = data
        else:
            if self.slots[hashvalue] == key:
                self.data[hashvalue] = data #replace
```

```
            else:
                nextslot = self.rehash(hashvalue,len(self.slots))
                while self.slots[nextslot] != None and self.slots[nextslot] != key:
                    nextslot = self.rehash(nextslot,len(self.slots))

                if self.slots[nextslot] == None:
                    self.slots[nextslot] = key
                    self.data[nextslot] = data
                else:
                    self.data[nextslot] = data #replace

    def hashfunction(self,key,size):
        return key%size

    def rehash(self,oldhash,size):
        return (oldhash+1)%size

    def get(self,key):
        startslot = self.hashfunction(key,len(self.slots))

        data = None
        stop = False
        found = False
        position = startslot
        while self.slots[position] != None and not found and not stop:
                if self.slots[position] == key:
                    found = True
                    data = self.data[position]
                else:
                    position=self.rehash(position,len(self.slots))
                    if position == startslot:
                        stop = True
        return data

    def __getitem__(self,key):
        return self.get(key)

    def __setitem__(self,key,data):
        self.put(key,data)


H=HashTable()
H[54]="cat"
H[26]="dog"
H[93]="lion"
H[17]="tiger"
H[77]="bird"
H[31]="cow"
H[44]="goat"
H[55]="pig"
H[20]="chicken"
print(H.slots)
```

```
print(H.data)


print(H[20])
print(H[17])

H[20] = 'duck'
print(H[20])

print(H.data)

print(H[99])

"""
[77, 44, 55, 20, 26, 93, 17, None, None, 31, 54]
['bird', 'goat', 'pig', 'chicken', 'dog', 'lion', 'tiger', None, None, 'cow', 'cat']
chicken
tiger
duck
['bird', 'goat', 'pig', 'duck', 'dog', 'lion', 'tiger', None, None, 'cow', 'cat']
None
"""
```

注意，在11个槽中，有9个被占用了。占用率被称作载荷因子（ load factor），记作λ，定义如下。

$$\lambda = \frac{\text{元素个数}}{\text{散列表大小}}$$

在本例中, $\lambda = \frac{9}{11}$.

1.6 散列表的查找

在散列表上进行查找的过程和创建散列表的过程基本一致。

二、神奇的dict

dict的value如果是list，是邻接表。dici嵌套dict 是 字典树/前缀树/Trie。

是的，你提到的两种数据结构分别是邻接表和字典树（前缀树，Trie）。

1. **邻接表**：在图论中，邻接表是一种表示图的常见方式之一。如果你使用字典（`dict`）来表示图的邻接关系，并且将每个顶点的邻居顶点存储为列表（`list`），那么就构成了邻接表。例如：

```
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D'],
    'C': ['A', 'D'],
    'D': ['B', 'C']
}
```

2. **字典树（前缀树，Trie）**：字典树是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。如果你使用嵌套的字典来表示字典树，其中每个字典代表一个节点，键表示路径上的字符，而值表示子节点，那么就构成了字典树。例如：

```
trie = {
    'a': {
        'p': {
            'p': {
                'l': {
                    'e': {'is_end': True}
                }
            }
        }
    },
    'b': {
        'a': {
            'l': {
                'l': {'is_end': True}
            }
        }
    },
    'c': {
        'a': {
            't': {'is_end': True}
        }
    }
}
```

这样的表示方式使得我们可以非常高效地搜索和插入字符串，特别是在大型数据集上。

神奇的dict

字典（dict）是Python中非常强大和灵活的数据结构之一，它可以用来存储键值对，是一种可变容器模型，可以存储任意数量的 Python 对象。

字典在 Python 中被广泛用于各种场景，例如：

1. **哈希映射**：字典提供了一种快速的键值查找机制，可以根据键快速地检索到相应的值。这使得字典成为了哈希映射（Hash Map）的理想实现。
2. **符号表**：在编程语言的实现中，字典常常被用作符号表，用来存储变量名、函数名等符号和它们的关联值。
3. **配置文件**：字典可以用来表示配置文件中的键值对，例如JSON文件就是一种常见的字典格式。
4. **缓存**：字典常常用于缓存中，可以将计算结果与其输入参数关联起来，以便后续快速地检索到相同参数的计算结果。
5. **图的表示**：如前文所述，字典可以用来表示图的邻接关系，是一种常见的图的表示方式。

由于其灵活性和高效性，字典在Python中被广泛应用于各种场景，并且被称为是Python中最常用的数据结构之一。

Algorithm for BFS

How to implement Breadth First Search algorithm in Python

https://www.codespeedy.com/breadth-first-search-algorithm-in-python/

BFS is one of the traversing algorithm used in graphs. This algorithm is implemented using a queue data structure. In this algorithm, the main focus is on the vertices of the graph. Select a starting node or vertex at first, mark the starting node or vertex as visited and store it in a queue. Then visit the vertices or nodes which are adjacent to the starting node, mark them as visited and store these vertices or nodes in a queue. Repeat this process until all the nodes or vertices are completely visited.

Advantages of BFS

1. It can be useful in order to find whether the graph has connected components or not.
2. It always finds or returns the shortest path if there is more than one path between two vertices.

Disadvantages of BFS

1. The execution time of this algorithm is very slow because the time complexity of this algorithm is exponential.
2. This algorithm is not useful when large graphs are used.

Implementation of BFS in Python (Breadth First Search)

Source Code: BFS in Python

```
graph = {'A': ['B', 'C', 'E'],
         'B': ['A','D', 'E'],
         'C': ['A', 'F', 'G'],
         'D': ['B'],
         'E': ['A', 'B','D'],
         'F': ['C'],
         'G': ['C']}

```

```
9
10 def bfs(graph, initial):
11     visited = []
12     queue = [initial]
13
14     while queue:
15         node = queue.pop(0)
16         if node not in visited:
17             visited.append(node)
18             neighbours = graph[node]
19
20             for neighbour in neighbours:
21                 queue.append(neighbour)
22     return visited
23
24 print(bfs(graph,'A'))
```

Explanation:

1. Create a graph.
2. Initialize a starting node.
3. Send the graph and initial node as parameters to the bfs function.
4. Mark the initial node as visited and push it into the queue.
5. Explore the initial node and add its neighbours to the queue and remove the initial node from the queue.
6. Check if the neighbours node of a neighbouring node is already visited.
7. If not, visit the neighbouring node neighbours and mark them as visited.
8. Repeat this process until all the nodes in a graph are visited and the queue becomes empty.

Output:

```
['A', 'B', 'C', 'E', 'D', 'F', 'G']
```

Algorithm for DFS

https://www.codespeedy.com/depth-first-search-algorithm-in-python/

This algorithm is a recursive algorithm which follows the concept of backtracking and implemented using stack data structure. But, what is backtracking.

Backtracking:-

It means whenever a tree or a graph is moving forward and there are no nodes along the existing path, the tree moves backwards along the same path which it went forward in order to find new nodes to traverse. This process keeps on iterating until all the unvisited nodes are visited.

How stack is implemented in DFS:-

1. Select a starting node, mark the starting node as visited and push it into the stack.
2. Explore any one of adjacent nodes of the starting node which are unvisited.
3. Mark the unvisited node as visited and push it into the stack.
4. Repeat this process until all the nodes in the tree or graph are visited.
5. Once all the nodes are visited, then pop all the elements in the stack until the stack becomes empty.

Implementation of DFS in Python

Source Code: DFS in Python

```
import sys

def ret_graph():
    return {
        'A': {'B':5.5, 'C':2, 'D':6},
        'B': {'A':5.5, 'E':3},
        'C': {'A':2, 'F':2.5},
        'D': {'A':6, 'F':1.5},
        'E': {'B':3, 'J':7},
        'F': {'C':2.5, 'D':1.5, 'K':1.5, 'G':3.5},
        'G': {'F':3.5, 'I':4},
        'H': {'J':2},
        'I': {'G':4, 'J':4},
        'J': {'H':2, 'I':4},
        'K': {'F':1.5}
    }

start = 'A'
dest = 'J'
visited = []
stack = []
graph = ret_graph()
path = []


stack.append(start)
visited.append(start)
while stack:
    curr = stack.pop()
    path.append(curr)
    for neigh in graph[curr]:
        if neigh not in visited:
            visited.append(neigh)
            stack.append(neigh)
            if neigh == dest :
                print("FOUND:", neigh)
                print(path)
```

```
                sys.exit(0)
print("Not found")
print(path)
```

Explanation:

1. First, create a graph in a function.
2. Intialize a starting node and destination node.
3. Create a list for the visited nodes and stack for the next node to be visited.
4. Call the graph function.
5. Initially, the stack is empty.Push the starting node into the stack (stack.append(start)).
6. Mark the starting node as visited (visited.append(start)).
7. Repeat this process until all the neighbours are visited in the stack till the destination node is found.
8. If the destination node is found exit the while loop.
9. If the destination node is not present then “Not found” is printed.
10. Finally, print the path from starting node to the destination node.

三、晴问基础题目

栈和队列（20题）

搜索专题（15题）

树专题（46题）

Todo 图算法专题（33题）

四、笔试题目

2023年有考到KMP，冒泡排序的优化。

2022年5个大题：图Dijkstra，二叉树，排序，单链表，二叉树。

2021年6个大题：森林dfs、bfs，哈夫曼树，二叉树建堆，图prim，二叉树遍历，图走迷宫。

参考

Python数据结构与算法分析(第2版)，布拉德利·米勒 戴维·拉努姆/吕能,刁寿钧译，出版时间:2019-09

Brad Miller and David Ranum, Problem Solving with Algorithms and Data Structures using Python, https://runestone.academy/ns/books/published/pythonds/index.html

https://github.com/wesleyjtann/Problem-Solving-with-Algorithms-and-Data-Structures-Using-Python

数据结构（C语言版 第2版）(严蔚敏)