数据结构与算法

Data Structure and Algorithm

极夜酱

目录

Part I

基础篇

Chapter 1 哈希表

1.1 哈希表

1.1.1 哈希表（Hash Table）

例如开发一个学生管理系统，需要有通过输入学号快速查出对应学生的姓名的功能。这里不必每次都去查询数据库，而可以在内存中建立一个缓存表，这样做可以提高查询效率。

学号	姓名
001121	张三
002123	李四
002931	王五
003278	赵六

表 1.1: 学生名单

再例如需要统计一本英文书里某些单词出现的频率，就需要遍历整本书的内容，把这些单词出现的次数记录在内存中。

单词	出现次数
this	108
and	56
are	79
by	46

表 1.2: 词频统计

因为这些需要，一个重要的数据结构诞生了，这个数据结构就是哈希表。哈希表也称散列表，哈希表提供了键（key）和值（value）的映射关系，只要给出一个 key，就可以高效地查找到它所匹配的 value。

哈希表的时间复杂度几乎是常量 $O(1)$，即查找时间与问题规模无关。

哈希表的两项基本工作：

1. 计算位置：构造哈希函数确定关键字的存储位置。
2. 解决冲突：应用某种策略解决多个关键字位置相同的问题。

1.2 哈希函数

1.2.1 哈希函数（Hash Function）

哈希的基本思想是将键 key 通过一个确定的函数，计算出对应的函数值 value 作为数据对象的存储地址，这个函数就是哈希函数。

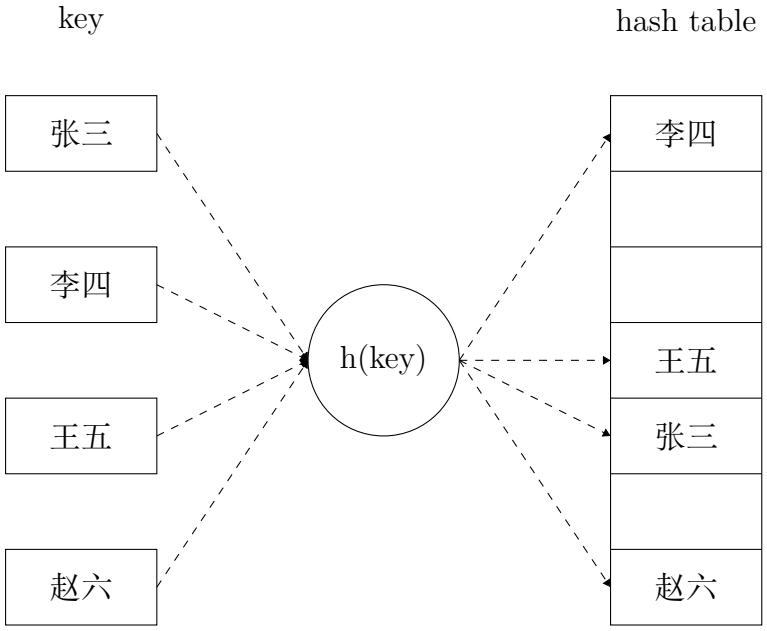

图 1.1: 哈希函数

哈希表本质上也是一个数组，可是数组只能根据下标来访问，而哈希表的 key 则是以字符串类型为主的。

在不同的语言中，哈希函数的实现方式是不一样的。假设需要存储整型变量，转化为数组的下标就不难实现了。最简单的转化方式就是按照数组长度进行取模运算。

一个好的哈希函数应该考虑两个因素：

1. 计算简单，以便提高转换速度。
2. 关键字对应的地址空间分布均匀，以尽量减少冲突。

1.2.2 数字关键字的哈希函数构造方法

对于数字类型的关键字，哈希函数有以下几种常用的构造方法：

直接定址法

取关键字的某个线性函数值为散列地址。

$$h(key) = a * key + b$$

例如根据出生年份计算人口数量 h(key) = key - 1990：

地址	出生年份	人数
0	1990	1285 万
1	1991	1281 万
2	1992	1280 万
...	...	...
10	2000	1250 万
...	...	...
21	2011	1180 万

表 1.3: 直接定址法

除留余数法

哈希函数为 h(key) = key % p，p 一般取素数。

例如 h(key) = key % 17：

地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
关键字	34	18	2	20			23	7	42		27	11		30		15	

表 1.4: 除留余数法

数字分析法

分析数字关键字在各位上的变化情况，取比较随机的位作为散列地址。

例如取 11 位手机号码的后 4 位作为地址 h(key) = int(key + 7)。

再例如取 18 位身份证号码中变化较为随机的位数：

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**
3	3	0	1	0	6	1	9	9	0	1	0	0	8	0	4	1	9
省		市		区		年				月		日		辖			校验

表 1.5: 数字分析法

折叠法

把关键字分割成位数相同的几个部分，然后叠加。

例如将整数 56793542 每三位进行分割：

$$\begin{array}{rr} & 542 \\ & 793 \\ + & 056 \\ \hline = & 1319 \end{array}$$

$$h(56793542) = 319$$

平方取中法

计算关键字的平方，取中间几位。

例如整数 56793542：

$$\begin{array}{rr} & 56793542 \\ * & 56793542 \\ \hline = & 3225506412905764 \end{array}$$

$$h(56793542) = 641$$

1.2.3 字符串关键字的哈希函数构造方法

对于字符串类型的关键字，哈希函数有以下几种常用的构造方法：

ASCII 码加和法

$$h(key) = \left(\sum key[i]\right) \ mod \ N$$

但是对于某些字符串会导致严重冲突，例如：a3、b2、c1 或 eat、tea 等。

移位法

取前 3 个字符移位。

$$h(key) = \left(key[0] \times 27^2 + key[1] \times 27 | key[2]\right) \ mod \ N$$

对于一些字符串仍然会冲突，例如 string、strong、street、structure 等。

一个有效的改进是涉及关键字中所有 n 个字符：

$$h(key) = \left(\sum_{i=0}^{n-1} key[n-i-1] \times 32^i\right) \ mod \ N$$

哈希函数

快速计算 $h('abcde') = a * 32^4 + b * 32^3 + c * 32^2 + d * 32 + e$

```
int hash(char *key, int tableSize) {
    int h = 0;          // hash value
    int i = 0;
    while(key[i] != '\0') {
        h = (h << 5) + key[i];
        i++;
    }
    return h % tableSize;
}
```

凯撒加密

```
/**
```

```
2  * @brief  凯撒加密
3  * @note   加密算法: ciphertext[i] = (plaintext[i] + Key) % 128
4  * @param  plaintext: 明文
5  * @retval 密文
6  */
7  char* encrypt(char *plaintext) {
8      int n = strlen(plaintext);
9      char *ciphertext = (char *)malloc((n + 1) * sizeof(char));
10     for(int i = 0; i < n; i++) {
11         ciphertext[i] = (plaintext[i] + KEY) % 128;
12     }
13     ciphertext[n] = '\0';
14     return ciphertext;
15 }
16
17 /**
18 * @brief  凯撒解密
19 * @note   解密算法: plaintext[i] = (ciphertext[i] - key + 128) % 128
20 * @param  ciphertext: 密文
21 * @retval 明文
22 */
23 char* decrypt(char *ciphertext) {
24     int n = strlen(ciphertext);
25     char *plaintext = (char *)malloc((n + 1) * sizeof(char));
26     for(int i = 0; i < n; i++) {
27         plaintext[i] = (ciphertext[i] - KEY + 128) % 128;
28     }
29     plaintext[n] = '\0';
30     return plaintext;
31 }
```

1.3 冲突处理

1.3.1 装填因子（Load Factor）

假设哈希表空间大小为 m，填入表中元素个数是 n，则称 $\alpha = n/m$ 为哈希表的装填因子。

当哈希表元素太多，即装填因子 α 太大时，查找效率会下降。实用最大装填因子一般取 $0.5 \leq \alpha \leq 0.85$。当装填因子过大时，解决的方法是加倍扩大哈希表，这个过程叫作再散列（rehashing）。

再散列的过程需要遍历原哈希表，把所有的关键字重新散列到新数组中。为什么需要重新散列呢？因为长度扩大以后，散列的规则也随之改变。经过扩容，原本拥挤的哈希表重新变得稀疏，原有的关键字也重新得到了尽可能均匀的分配。

装填因子也是影响产生哈希冲突的因素之一。当不同的关键字可能会映射到同一个散列地址上，就导致了哈希冲突（collision），即 $h(key_i) = h(key_j)$，$key_i \neq key_j$，因此需要某种冲突解决策略。

例如有 11 个数据对象的集合 {18, 23, 11, 20, 2, 7, 27, 30, 42, 15, 34, 35}，哈希表的大小为 17，哈希函数选择 h(key) = key % size。

地址	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
关键字	34	18	2	20			23	7	42		27	11		30		15	

在插入最后一个关键字 35 之前，都没有产生任何冲突。但是 h(35) = 1，位置已有对象，就导致了冲突。

常用的处理冲突的思路有两种：

1. 开放地址法（open addressing）：一旦产生了冲突，就按某种规则去寻找另一空地址。开放地址法主要有线性探测法、平方探测法（二次探测法）和双散列法。

2. 分离链接法：将相应位置上有冲突的所有关键字存储在同一个单链表中。

1.3.2 线性探测法（Linear Probing）

当产生冲突时，以增量序列 1, 2, 3, ..., n - 1 循环试探下一个存储地址。

例如序列 {47, 7, 29, 11, 9, 84, 54, 20, 30}，哈希表表长为 13，哈希函数 h(key) = key % 11，用线性探测法处理冲突。

key	**47**	**7**	**29**	**11**	**9**	**84**	**54**	**20**	**30**
h(key)	3	7	7	0	9	7	10	9	8
冲突次数	0	0	1	0	0	3	1	3	6

地址	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**Δ**
插入 47				47										0
插入 7				47				7						0
插入 29				47				7	29					1
插入 11	11			47				7	29					0
插入 9	11			47				7	29	9				0
插入 84	11			47				7	29	9	84			3
插入 54	11			47				7	29	9	84	54		1
插入 20	11			47				7	29	9	84	54	20	3
插入 30	11	30		47				7	29	9	84	54	20	6

表 1.6: 线性探测法

线性探测法的缺陷在于容易出现聚集现象。

1.3.3 平方探测法（Quadratic Probing）

平方探测法也称为二次探测法，以增量序列 $1^2, -1^2, 2^2, -2^2, \ldots, q^2, -q^2$ $(q \leq \lfloor N/2 \rfloor)$ 循环试探下一个存储地址。

例如序列 {47, 7, 29, 11, 9, 84, 54, 20, 30}，哈希表表长为 11，哈希函数 h(key) = key % 11，用平方探测法处理冲突。

key	47	7	29	11	9	84	54	20	30
h(key)	3	7	7	0	9	7	10	9	8
冲突次数	0	0	1	0	0	2	0	3	3

地址	0	1	2	3	4	5	6	7	8	9	10	Δ
插入 47				47								0
插入 7				47				7				0
插入 29				47				7	29			1
插入 11	11			47				7	29			0
插入 9	11			47				7	29	9		0
插入 84	11			47			84	7	29	9		-1
插入 54	11			47			84	7	29	9	54	0
插入 20	11		20	47			84	7	29	9	54	4
插入 30	11	30	20	47			84	7	29	9	54	4

表 1.7: 平方探测法

但是只要还有空间，平方探测法就一定能找到空闲位置吗?

例如对于以下哈希表，插入关键字 11，哈希函数 h(key) = key % 5，用平方探测法处理冲突。

下标	0	1	2	3	4
key	5	6	7		

表 1.8: 平方探测法存在的问题

对关键字 11 进行平方探测的结果一直在下标 0 和 2 之间波动，永远无法达到其它空的位置。

但是有定理证明，如果哈希表长度是满足 $4k+3$ $(k \in Z^+)$ 形式的素数时，平方探测法就可以探查到整个哈希表空间。

1.3.4 双散列探测法（Double Hashing）