哈希表

主讲人：邓哲也

哈希表的引入

如果要储存和使用线性表（1，75，324，43，1353，90，46）

一般情况下我们会使用一个数组 A[1..7] 来顺序存储这些数。

但是这样的存储结构会给查询算法带来 O(n) 的时间开销。

对 A 排序，使用二分查询法，时间复杂度变为 O(log n)

也可以用空间换时间的做法，用数组 A[1..1353] 来表示每个数是否出现，查找的时间复杂度变为 O(1)，但是空间上的开销变得巨大。

哈希表的引入

优化上一种做法，建立一个哈希函数 h(key) = key % 23.

(1, 75, 324, 43, 1353, 90) -> (1, 6, 2, 20, 19, 21, 0)

我们只要用一个 A[0..22] 数组就可以快速的查询每个数是否出现。

这种线性表的结构就称为哈希表（Hash Table）。

哈希表的基本原理

可以看出，哈希表的基本原理是用一个下标范围比较大的数组 A 来存储元素。

设计一个函数 h，对于要存储的线性表的每个元素 node，取一个关键字 key，算出函数值 h(key) 然后把这个值作为下标，用 A[h(key)] 来存储 node。

最常见的 h 就是模函数，也就是选定一个 m，令 h(key) = key % m.

哈希表的冲突

但是有一个问题，可能存在两个 key：k1，k2 使得 h(k1)=h(k2)，这时也称产生了“冲突”。

解决冲突有很多种办法：

可以用另一个函数 I 去计算 I(k1)，I(k2)，找到新的位置。

可以让 A 的每个元素都存一个链表，对于 h(k1)=h(k2)，我们可以让这两个 node 都接在 A[h(k1)] 的链表上。

……

哈希表的冲突

假设我们使用第二种方法解决冲突。

对于插入元素(node, key)：

- 计算 h(key)，把 node 插入 A[h(key)] 链表。

对于查询元素(node, key)：

- 计算 h(key)，如果 A[h(key)] 为空，说明 node 不存在。

  否则遍历 A[h(key)] 链表，寻找 node。

哈希表的代码实现

```
struct node{
      int next, info;
}hashNode[N];
int tot; // 哈希表节点计数
int h[M]; // 初始化为 -1
```

哈希表的代码实现

```
void insert(int key, int info){
    int u = key % M;
    hashNode[tot] = (node){h[u], info};
    h[u] = tot ++;
}
```

哈希表的代码实现

```
int find(int key, int info){
        int u = key % M;
        for(int i = h[u]; i != -1; i = hashNode[i].next){
                if (hashNode[i].info == info)
                        return 1;
        }
        return 0;
}
```

边学边练

已知 X[1..4] 是 [-T, T] 中的整数，求出满足方程

$$AX[1]+BX[2]+CX[3]+DX[4] = P$$

的解有多少组？

$|P| \leqslant 10^9$，$|A|,|B|,|C|,|D| \leqslant 10^4$，$T \leqslant 500$

边学边练

最直观的方法枚举 X[1..4]，时间复杂度 $O(n^4)$

适当优化，枚举了X[1..3] 之后，实际上 X[4] 已经确定了，

时间复杂度 $O(n^3)$

继续优化，采用 meet in the middle 策略：

一边枚举 X[1]，X[2]

一边枚举 X[4]，X[3]

然后看有哪些方案可以组成方程的解。

边学边练

枚举 X[1]，X[2]，然后算出 P-AX[1]-BX[2]

把这个值存入一个哈希表，注意要统计次数。

这一步时间复杂度 $O(n^2)$

然后枚举 X[3]，X[4]，算出 CX[3]+DX[4]

去哈希表里查找这个值出现了几次。

把次数加进答案，这一步时间复杂度 $O(n^2)$

因此总的时间复杂度是 $O(n^2)$

下节课再见