Hash梳理

什么是Hash

Hash是一种在信息学竞赛中经常用到的技巧类工具。

Hash的本质是把一个极大的，不可再范围内表示的值域。通过Hash函数映射到一个值域较小、可以方便比较的值域。

一个好的Hash函数可以很大程度上提高程序的整体时间效率和空间效率。

比如最简单的hash：

```
char str[100];
int a[26];
a[str[i]-'a']++;
```

平时网络、密码学中都有使用到Hash的思想，比如：

- MD5校验 将字节的形式转换成128位16进制的性质，通过MD5码来校验文件。
- SHA-1安全散列 可以生成一个被称为消息摘要的160位（20字节）散列值，散列值通常的呈现形式为40个十六进制数。

Hash表

hash表是使用`Hash Function`对某一特定的类型或者属性，将其存储在可以用线性结构表示的容器中（比如：数组、链表等）。

一般的Hash表需要提前设计好表容量和对应的Hash函数。**如果是对于未知容量数据，就需要动态的内存管理和随机Hash函数策略。**

Hash表的**装载因子**：留给大家数据结构的老师去讲了哈 TVT.

HashMap

基于Hash表的映射结构，和传统的`map`映射不同点：

- 传统`map`基于红黑树的结构，而`HashMap`是基于Hash表的结构。
- `HashMap`在插入、查找、删除的操作上，时间复杂度均摊为：$O(1)$
  - 在特殊数据上，或者针对其Hash函数，可能退化成$O(n)$
- Hashmap的空间复杂度较大，一般都要大于时间的数据内容。

C++11，自带了Hash表的是`unordered_map<key, value>`，用法和`map`一样

- 头文件`#include <unordered_map>`
- 使用方式`unordered_map<key, value, HashFunc=std::hash<key>>`
  - 例如
  - `unordered_map<string, int>`
  - `unordered_map<int, int>`
  - `unordered_map<node, int, myHashF>`

- **写大作业 或刷 leetcode 面试题 可以使用，打比赛能不用尽量不用，熟悉STL源码的，可以把这个容器卡掉。**

JAVA中也有`HashMap`，感觉设计上比C++的`unordered_map`好，是基于`Hash表+红黑树`的形式，基本保证了均摊为$O(1)$的时间复杂度

Hash冲突

因为从大值域映射到小值域，数据量大了，必然会有冲突。

这个时候用公式表达就是：

```
key1 ≠ key2 , f(key1) = f(key2)
```

因此存入表，就需要解决这个冲突。（具体也留给老师吧）

整数的Hash

- **直接取余法**

  我们用关键字k除以$\lfloor \log_2 M \rfloor$，取余数作为在Hash表中的位置。函数表达式可以写成：

  $$h(k) = k \operatorname{Mod} M$$

- 乘积取整法
- 平方取中法

字符串Hash (重点)

用很多很多的字符串Hash方案，百度百科里，基本是密码学的内容。

在竞赛中常用的字符串Hash方法是：**进制哈希。**进制哈希的核心便是**给出一个固定进制base，将一个串的每一个元素看做一个进制位上的数字，所以这个串就可以看做一个base进制的数，那么这个数就是这个串的哈希值；则我们通过比对每个串的的哈希值，即可判断两个串是否相同。注意在算进制的过程中，数字会很大，所以应该再选取一个数字Mod对Hash值取模。**

base和Mod最好对为质数，这样出现Hash冲突的概率会小很多，至于为什么，可以参考这边文章，点击此处查看。

这里提供一些常用的素数值，最为**base**和**mod**的选取：

```
const int base[6] = {131, 56369, 99991, 999983, 16341163, 19260817};
const int mod[7] = {19260817, 999998639,  999998641, 100000007,1000000107,
1000000009, 1000000103};
// 最好别用999983 100000007 这几个出现频率极高的模数
```

这种最朴素的Hash方式，又被称为单模数Hash，其模板一般为：

```cpp
// base 和 mod 自选
void hash_value() {
    Hash[0] = 0;
    p[0] = 1;
    int len = strlen(s + 1);
    for (int i = 1; i <= len; i++) {
        Hash[i] = (1LL * Hash[i - 1] * base + s[i]) % mod;
        p[i] = 1LL * p[i - 1] * base % mod;
    }
}
```

错误率

若进行 n 次比较，每次错误率 $\frac{1}{M}$ ，那么总错误率是 $1-\left(1-\frac{1}{M}\right)^n$ 。在随机数据下，若 $M=10^9+7, n=10^6$ ，错误率约为 $\frac{1}{1000}$ ，并不是能够完全忽略不计的。

所以，进行字符串哈希时，经常会对两个大质数分别取模，这样的话哈希函数的值域就能扩大到两者之积，错误率就非常小了。

- 自然溢出法

  这种方法是利用数据类型`unsigned long long`的范围自然溢出：即当存储的数据大于`unsigned long long`的存储范围时，会自动$mod\ 2^{64}-1$，就不用mod其他质数来保证唯一性了，因其模的范围大了，所以也减少了冲突的概率。

- 双Hash

  双Hash就是用两个不同的**mod**值来计算Hash，如果两个Hash值都相等才认为是同一个字符串，Hash冲突概率降低了很多，大质数+双Hash，基本造不出冲突的数据。

  这样的结果可以用一个`pair`表示，即：

$$pair<HashValue1,HashValue2>$$

  但是常数大，因为取模的速度比加减法慢很多，所以如果题目时间很少那可能会被卡。

  当然双Hash可以扩展到多Hash，即用一个二维数据，表示多个Hash值。

  时间上：

$$\text{自然溢出法}<\text{单}\mathrm{Hash}+\text{大质数}<\text{双}\mathrm{Hash}+\text{大质数}$$

  在冲突率上：

$$\text{单}\mathrm{Hash}+\text{大质数}>\text{自然溢出法}>\text{双}\mathrm{Hash}+\text{大质数}$$

子串的Hash值

单次计算一个字符串的哈希值复杂度是 $O(n)$，其中 n 为串长，与暴力匹配没有区别，如果需要多次询问一个字符串的子串的哈希值，每次重新计算效率非常低下。

一般采取的方法是对整个字符串先预处理出每个前缀的哈希值，将哈希值看成一个 b 进制的数对 M 取模的结果，这样的话每次就能快速求出子串的哈希了：

令 $f_i(s)$ 表示 $f(s[1..i])$，那么 $f(s[l..r]) = \frac{f_r(s)-f_{l-1}(s)}{b^{l-1}}$，其中 $\frac{1}{b^{l-1}}$ 也可以预处理出来，用乘法逆元 或者是在比较哈希值时等式两边同时乘上 b 的若干次方化为整式均可。

这样的话，就可以在 $O(n)$ 的预处理后每次 $O(1)$ 地计算子串的哈希值了。

具体的证明和操作可以参考这个，点击查看

使用Hash的几个需要注意的地方

- 在复杂度允许的情况下，**尽量采用多Hash**（不过一般双Hash就够）
- **比赛时能不用自然溢出就不要**（平时刷题如果用自然溢出被卡可以及时换掉，但是比赛时如果用自然溢出，OI赛制就GG了）
- 模数用大质数这个不用说了
- **并且进制数不要选太简单的，比如 233和 13131 这样的，尽量大一点，比如1313131和233333，太小容易被卡。**
- 以及要合理应对各种卡hash方法的最好方法就是自己去卡一遍hash，详情请参考**BZOJ** hash killer 系列。