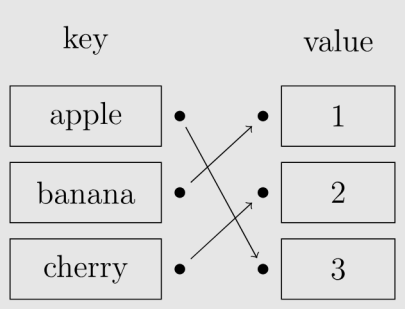
哈希表

引入



哈希表又称散列表，一种以「key-value」形式存储数据的数据结构。所谓以「key-value」形式存储数据，是指任意的键值 key 都唯一对应到内存中的某个位置。只需要输入查找的键值，就可以快速地找到其对应的 value。可以把哈希表理解为一种高级的数组，这种数组的下标可以是很大的整数，浮点数，字符串甚至结构体。

哈希函数

要让键值对应到内存中的位置，就要为键值计算索引，也就是计算这个数据应该放到哪里。这个根据键值计算索引的函数就叫做哈希函数，也称散列函数。举个例子，如果键值是一个人的身份证号码，哈希函数就可以是号码的后四位，当然也可以是号码的前四位。生活中常用的「手机尾号」也是一种哈希函数。在实际的应用中，键值可能是更复杂的东西，比如浮点数、字符串、结构体等，这时候就要根据具体情况设计合适的哈希函数。哈希函数应当易于计算，并且尽量使计算出来的索引均匀分布。

能为 key 计算索引之后，我们就可以知道每个键值对应的值 value 应该放在哪里了。假设我们用数组 a 存放数据，哈希函数是 f，那键值对 `(key, value)` 就应该放在 `a[f(key)]` 上。不论键值是什么类型，范围有多大，`f(key)` 都是在可接受范围内的整数，可以作为数组的下标。

在 OI 中，最常见的情况应该是键值为整数的情况。当键值的范围比较小的时候，可以直接把键值作为数组的下标，但当键值的范围比较大，比如以 10^9 范围内的整数作为键值的时候，就需要用到哈希表。一般把键值模一个较大的质数作为索引，也就是取 $f(x) = x \bmod M$ 作为哈希函数。

另一种比较常见的情况是 key 为字符串的情况，由于不支持以字符串作为数组下标，并且将字符串转化成数字存储也可以避免多次进行字符串比较。所以在 OI 中，一般不直接把字符串作为键值，而是先算出字符串的哈希值，再把其哈希值作为键值插入到哈希表里。关于字符串的哈希值，我们一般采用进制的思想，将字符串想象成一个 127 进制的数。那么，对于每一个长度为 n 的字符串 s，就有：

$x = s_0 \cdot 127^0 + s_1 \cdot 127^1 + s_2 \cdot 127^2 + \cdots + s_n \cdot 127^n$

我们可以将得到的 x 对 2^{64}（即 `unsigned long long` 的最大值）取模。这样 `unsigned long long` 的自然溢出就等价于取模操作了。可以使操作更加方便。

这种方法虽然简单，但并不是完美的。可以构造数据使这种方法发生冲突（即两个字符串的 x 对 2^{64} 取模后的结果相同）。
我们可以使用双哈希的方法：选取两个大质数 a, b。当且仅当两个字符串的哈希值对 a 和对 b 取模都相等时，我们才认为这两个字符串相等。这样可以大大降低哈希冲突的概率。

冲突

如果对于任意的键值，哈希函数计算出来的索引都不相同，那只用根据索引把 `(key, value)` 放到对应的位置就行了。但实际上，常常会出现两个不同的键值，他们用哈希函数计算出来的索引是相同的。这时候就需要一些方法来处理冲突。在 OI 中，最常用的方法是拉链法。

拉链法

拉链法也称开散列法（open hashing）。

拉链法是在每个存放数据的地方开一个链表，如果有多个键值索引到同一个地方，只用把他们都放到那个位置的链表里就行了。查询的时候需要把对应位置的链表整个扫一遍，对其中的每个数据比较其键值与查询的键值是否一致。如果索引的范围是 $1 \dots M$，哈希表的大小为 N，那么一次插入/查询需要进行期望 $O(\frac{N}{M})$ 次比较。

实现

C++

```cpp
constexpr int SIZE = 1000000;
constexpr int M = 999997;

struct HashTable {
  struct Node {
    int next, value, key;
  } data[SIZE];

  int head[M], size;

  int f(int key) { return (key % M + M) % M; }

  int get(int key) {
    for (int p = head[f(key)]; p; p = data[p].next)
      if (data[p].key == key) return data[p].value;
    return -1;
  }

  int modify(int key, int value) {
    for (int p = head[f(key)]; p; p = data[p].next)
```

```
      if (data[p].key == key) return data[p].value = value;
    }

    int add(int key, int value) {
      if (get(key) != -1) return -1;
      data[++size] = Node{head[f(key)], value, key};
      head[f(key)] = size;
      return value;
    }
};
```

Python

```
M = 999997
SIZE = 1000000


class Node:
    def __init__(self, next=None, value=None, key=None):
        self.next = next
        self.value = value
        self.key = key


data = [Node() for _ in range(SIZE)]
head = [0] * M
size = 0


def f(key):
    return key % M


def get(key):
    p = head[f(key)]
    while p:
        if data[p].key == key:
            return data[p].value
        p = data[p].next
    return -1


def modify(key, value):
    p = head[f(key)]
    while p:
        if data[p].key == key:
            data[p].value = value
            return data[p].value
        p = data[p].next


def add(key, value):
    if get(key) != -1:
        return -1
    size = size + 1
    data[size] = Node(head[f(key)], value, key)
```

```
    head[f(key)] = size
    return value
```

这里再提供一个封装过的模板，可以像 map 一样用，并且较短

```
struct hash_map {  // 哈希表模板

  struct data {
    long long u;
    int v, nex;
  };  // 前向星结构

  data e[SZ << 1];  // SZ 是 const int 表示大小
  int h[SZ], cnt;

  int hash(long long u) { return (u % SZ + SZ) % SZ; }

  // 这里使用 (u % SZ + SZ) % SZ 而非 u % SZ 的原因是
  // C++ 中的 % 运算无法将负数转为正数

  int& operator[](long long u) {
    int hu = hash(u);  // 获取头指针
    for (int i = h[hu]; i; i = e[i].nex)
      if (e[i].u == u) return e[i].v;
    return e[++cnt] = data{u, -1, h[hu]}, h[hu] = cnt, e[cnt].v;
  }

  hash_map() {
    cnt = 0;
    memset(h, 0, sizeof(h));
  }
};
```

在这里，hash 函数是针对键值的类型设计的，并且返回一个链表头指针用于查询。在这个模板中我们写了一个键值对类型为 `(long long, int)` 的 hash 表，并且在查询不存在的键值时返回 -1。函数 `hash_map()` 用于在定义时初始化。

闭散列法

闭散列方法把所有记录直接存储在散列表中，如果发生冲突则根据某种方式继续进行探查。

比如线性探查法：如果在 `d` 处发生冲突，就依次检查 `d + 1`，`d + 2` ……

实现

```
constexpr int N = 360007;  // N 是最大可以存储的元素数量

class Hash {
 private:
  int keys[N];
  int values[N];

 public:
```

```
  Hash() { memset(values, 0, sizeof(values)); }

  int& operator[](int n) {
    // 返回一个指向对应 Hash[Key] 的引用
    // 修改成不为 0 的值 0 时候视为空
    int idx = (n % N + N) % N, cnt = 1;
    while (keys[idx] != n && values[idx] != 0) {
      idx = (idx + cnt * cnt) % N;
      cnt += 1;
    }
    keys[idx] = n;
    return values[idx];
  }
};
```

例题

「JLOI2011」不重复数字
