

仪式黑刃

计算机科学并不只是关于计算机，就像天文学并不只是关于望远镜一样。

博客园

首页

新随笔

联系

订阅

管理

开放定址法——平方探测(Quadratic Probing)

为了消除一次聚集，我们使用一种新的方法：平方探测法。顾名思义就是冲突函数F(i)是二次函数的探测方法。通常会选择 $f(i)=i^2$ 。和上次一样，把{89,18,49,58,69}插入到一个散列表中，这次用平方探测看看效果，再复习一下探测规则： $h_i(x) = (Hash(x) + F(i)) \% TableSize \ (i=0,1,2,...)$

| | Empty Table | After 89 | After 18 | After 49 | After 58 | After 69 |
|---|-------------|----------|----------|----------|----------|----------|
| 0 | | | | 49 | 49 | 49 |
| 1 | | | | | | |
| 2 | | | | | 58 | 58 |
| 3 | | | | | | 69 |
| 4 | | | | | | |
| 5 | | | | | | |
| 6 | | | | | | |
| 7 | | | | | | |
| 8 | | | 18 | 18 | 18 | 18 |
| 9 | | 89 | 89 | 89 | 89 | 89 |

脑内调试一下：49和89冲突时，下一个空闲位置是0号单元。58和18冲突时， $i=1$ 也冲突，再试 $i=2$ ， $h_2(58)=(8+4)\%10=2$ 是空的可以放。69同理。

对于线性探测法而言，我们得避免元素几乎填满的情况，因为这时候性能会急剧降低。对于平方探测法，这会更糟：如果表超过一半被填满，那当表的规模不是素数时，甚至在表被填满一般之前就已经不能一下找到空单元了，需要试探好几次才能找到一个空单元。原因是表最多有一半位置可以用来解决冲突。凭什么如此断言呢？Talk is cheap,show me your....proof.

定理

公告

昵称： 仪式黑刃
园龄： 3年11个月
粉丝： 28
关注： 2
[+加关注](#)

| | | | | | | |
|--------------|----|----|----|----|----|----|
| < 2021年10月 > | | | | | | |
| 日 | 一 | 二 | 三 | 四 | 五 | 六 |
| 26 | 27 | 28 | 29 | 30 | 1 | 2 |
| 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 1 | 2 | 3 | 4 | 5 | 6 |

搜索

找找看

谷歌搜索

我的标签

- 数据结构(20)
- 树(10)
- 散列(6)

如果使用平方探测，且表的规模是素数，那么当表至少有一半是空的时候，总能插入新的元素。

我们假设表的Size是一个大于3的素数，直接拿着定理证明有点让人不知所措，那把这个定理的证明转化为：证明“前 $\frac{Size}{2}$ 个备选位置是互异的”，然后用反证法。从所有前 $\frac{Size}{2}$ 个的位置里选两个： $(h(x) + i^2) \% Size$ 和 $(h(x) + j^2) \% Size$ ，其中 $0 < i, j \leq \frac{Size}{2}$ 。假设这两个位置相同，且 $i \neq j$ ，然后让他们位置相等，推出矛盾就行了，因为都mod Size，根据等式性质我们只需要考察括号里的项就行了。

$$\begin{aligned}(h(x) + i^2) &= (h(x) + j^2) \\ \Rightarrow i^2 &= j^2 \\ \Rightarrow (i-j)(i+j) &= 0\end{aligned}$$

前面说了 $i \neq j$ ，所以只可能 $i = -j$ 。但是这和他们的定义域矛盾，所以也是不可能的。所以前半位置互异，可供选择，任何元素都有 $\frac{Size}{2}$ 个可能被放的位置。综上，如果最多有一半的位置可用，那么空闲单元总是能找到的。反过来讲，哪怕表里有一半+1个位置被填上，那么插入都有可能失败（虽然这比较偶然，但还是有可能的），这一点是十分重要的，要拿小本本记下来，说不定校招或考研就出题了哈哈哈。另外保证Size是素数也是非常重要的，如果不是的话，那遭遇冲突时可供选择的空单元个数会锐减到你难以置信的地步，远比一半少，这样一来，我们的战略纵深就太小了，难以迂回，这种情况没人希望见到。



Size=16的时候，找备选的单元只能取 $i=1,2,3$ ，也就是距离冲突单元1,4,9个单位的位置了。

另外，在开放定址的散列表里，我们之前意义上的删除操作是不能进行的，因为某个数对应的单元可能已经引起过冲突了，然后他探测跑到别的位置了。比如我们要删除69，你find一下，定位到9，发现那躺着89，那我们只能跟着平方探测的思路再找找 $9+1^2$ ，结果发现还不对，在那的是58。得，继续找吧，试试 $9+2^2$ ，这才找到。想想吧，这才Size=10就这么费劲了，那企业级软件要处理千万级甚至亿级的数据怎么办，比如头条app的数据量，那程序还不跑到天荒地老。。。因此开放定址散列表需要懒惰删除。

谈谈怎么实现吧，先给出类型声明。在这里我们不用结构体数组，而使用散列表单元的数组，而且单元是动态分配地址这和分离链接一样。



组合数学(2)

链表(2)

栈(1)

随笔档案

2018年9月(5)

2018年8月(7)

2018年7月(3)

2017年12月(3)

2017年11月(1)

2017年10月(5)

阅读排行榜

- 1. 开放定址法——线性探测(Linear Probing)(14470)
- 2. 开放定址法——平方探测(Quadratic Probing)(12746)
- 3. 分离链接法(Separate Chaining)(5946)
- 4. 母函数简介(4982)
- 5. 双散列和再散列暨散列表总结(2630)

评论排行榜

- 1. 红黑树——以无厚入有间(4)
- 2. 二叉树及其实现(基础版)(4)
- 3. 分离链接法(Separate Chaining)(3)
- 4. 红黑树——首身离兮心不惩(2)

```
#ifndef HashQuad_h
#define HashQuad_h
typedef unsigned int Index;
typedef Index Position;
struct HashTbl;
typedef struct HashTbl *HashTable;

HashTable Init(int size);
void DestroyTable(HashTable H);
void Insert(int key, HashTable H);
Position Find(int key, HashTable H);
int Retrieve(Position P);
HashTable ReTable(HashTable H);
#endif /* HashQuad_h */

enum KindOfEntry{
    Legitimate,
    Empty,
    Deleted
};

struct HashEntry {
    int value;
    enum KindOfEntry Info;
};

typedef struct HashEntry Cell;

/*Cell *TheCells will be an array of
HashEntry cells,allocated later
*/
struct HashTbl {
    int TableSize;
    Cell *TheCells;
};
```



顺便一说，Hash函数还是设置为简单的%Size

```
Index Hash(int key,int size) {
    return key%size;
}
```

初始化由2步组成：分配空间，然后将每个单元的Info设置为Empty。

```
#define aPrime 307
#define MinTableSize 5

HashTable Initial(int size){
    HashTable H;
    int i;
    if (size<MinTableSize) {
        printf("Table size too small\n");
        return NULL;
    }

    //Allocate table
    H=(HashTable)malloc(sizeof(struct HashTbl));
```

5. B-树 分合之道(2)

推荐排行榜

- 1. 二叉堆(4)
- 2. 散列——动机引入(4)
- 3. 母函数简介(3)
- 4. 左式堆(2)
- 5. 开放定址法——线性探测(Linear Probing)(2)

最新评论

1. Re:开放定址法——线性探测(Linear Probing)

good

--codworm

2. Re:分离链接法(Separate Chaining)

我实现了下两种销毁哈希表函数，麻烦楼主看下有无需更改的地方 #if 0 void DestroyTable(HashTable *H) //销毁哈希表 { Position P_List, P_Next...

--HOWU

3. Re:分离链接法(Separate Chaining)

附上我的销毁表的函数 void DestroyTable(HashTable *H) //销毁哈希表 { Position P_List, P_Next; int i; for (i = 0; i < ...

--HOWU

4. Re:分离链接法(Separate Chaining)

List header=(List)malloc(H->TableSize * sizeof(struct ListNode)); //Allocate list headers for (i=0; ...

--HOWU

```
H->TableSize=aPrime;

//Allocate array of cells
H->TheCells=(Cell*)malloc(sizeof(Cell)*H->TableSize);

//Allocate list headers
for (i=0; i<H->TableSize; i++)
    H->TheCells[i].Info=Empty;

return H;
}
```



5. Re:母函数应用

连续顶

--lcl1997

和分离链接一样，Find返回key在散列表里的单元号码。而且因为被标记了Empty，我们想表达查找失败也很容易。



```
1 Position Find(int key,HashTable H){
2     Position cur;
3     int CollisionNum=0;
4     cur=Hash(key,H->TableSize);
5     while (H->TheCells[cur].Info != Empty &&
6           H->TheCells[cur].value!= key)
7     {
8         cur+= (++CollisionNum<<1) - 1;
9         if (cur>=H->TableSize)
10            cur-=H->TableSize;
11     }
12     return cur;
13 }
```



第8行到第10行是进行平方探测的快速方法，因为在实现的时候不太好判断进行到第几次探测了，所以直接算 i^2 不容易，另设个变量监测倒也可以，不过那样挺麻烦的，还占用空间，还多了一次监测变量的++，还多了一次判断，还多了一次平方运算，尤其是算平方开销太大了。所有的这些都会让效率变低。所以我们要把平方计算转化为单纯的+-计算，用 $i^2 - (i - 1)^2$ 算出他们之间的差距是 $2 * i - 1$ ，所以 $F(i) = F(i - 1) + 2 * i - 1$ 这个几乎全是加减，乘法用移位代替速度就快多了。如果新的定位越过数组，那么可以通过-Size把它拉回到数组的范围里。这比通常办法快多了，因为他避免了看似要做的乘法和平方。第行的判断顺序很重要，别翻过来，不然短路性质就用不上了。

然后说插入，如果Key存在，就什么也不做，否则就把插入元素放在Find的位置。



```
void Insert(int key, HashTable H){
    Position P=Find(key, H);
    if (H->TheCells[P].Info != Legitimate)
    {
        H->TheCells[P].Info=Legitimate;
        H->TheCells[P].value=key;
    }
}
```



虽然平方探测法排除了一次聚集，但是散列到同一位置上的元素将探测相同的备选单元，这么说有点抽象，就是探测的时候都会踩同样的坑，比如说89，49，69这三个数往散列表里放， $h_0(49)$ 撞到89了，试试 $i=1$ ，可以了。69撞到89了然后试试 $i=1$ ，算完之后 $h_1(69)=0$ 和 $h_1(49)$ 又撞了，这就叫“探测到相同的备选单元”，再试一次69才被安置。想想规模更大的表，相撞次数会更多，用 $f(i)=i^2$ 探测的时候分批扎堆，这就叫二次聚集，和之前相比，不是0，1，2，3这样连着一整块扎堆，而是在 $i=1，4，9，16$ 附近扎堆。这是这两种聚集的区别。

二次聚集是理论上的一个缺憾，下一篇里我们继续讨论如何排除这个缺憾，从而对散列表冲突问题的排解更为高效和优美。不过这需要花费另外一些时间去做乘除法，比平方探测单纯的加减法慢一些，有利有弊吧，实际场景里因地制宜地选择不同模型就好。

标签：散列， 数据结构

好文要顶

关注我

收藏该文

仪式黑刀

关注 - 2

粉丝 - 28

1

1

+加关注

« 上一篇： 开放定址法——线性探测(Linear Probing)
» 下一篇： 双散列和再散列暨散列表总结

posted @ 2018-08-06 20:49 仪式黑刀 阅读(12746) 评论(0) 编辑 收藏 举报

[刷新评论](#) [刷新页面](#) [返回顶部](#)

发表评论

编辑 预览

B

支持 Markdown

自动补全

提交评论

 退出 订阅评论

[Ctrl+Enter快捷键提交]

- 【推荐】**并行超算云面向博客园粉丝推出“免费算力限时申领”特别活动
- 【推荐】**百度智能云超值优惠：新用户首购云服务器1核1G低至69元/年
- 【推荐】**跨平台组态\工控\仿真\CAD 50万行C++源码全开放免费下载！

【推荐】和开发者在一起：华为开发者社区，入驻博客园科技品牌专区

【注册】App开发者必备：打造增长变现闭环，高效成长，收入提升28%



编辑推荐：

- 理解ASP.NET Core - 选项(Options)
- 跳槽一年后的回顾
- 在 Unity 中渲染一个黑洞
- 理解 ASP.NET Core - 配置(Configuration)
- CSS 奇技淫巧 | 妙用 drop-shadow 实现线条光影效果

最新新闻：

- 量子物理学家：如果宇宙中所有物体突然消失，会剩下一个「空宇宙」吗？（2021-10-12 22:15）
- 深度学习正改变物理系统模拟，速度最高提升20亿倍那种（2021-10-12 22:00）
- 因果推断研究获2021诺贝尔经济学奖，图灵奖得主祝贺并反对（2021-10-12 21:45）
- 天价遗产税！卖了116亿元股票还差得远，三星家族选择5年分期（2021-10-12 21:30）
- 苹果将Face ID等信息写入底层硬件后 第三方维修市场或迎来寒冬（2021-10-12 21:20）
- » 更多新闻...

Copyright © 2021 仪式黑刃

Powered by .NET 6 on Kubernetes