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。

双散列和再散列暨散列表总结

先说明一下，她们两个属于不同的范畴，双散列属于开放定址法，仍是一种解决冲突的策略。而再散列是为了解决插入操作运行时间过长、插入失败问题的策略。简而言之，她们的区别在于：前者让散列表做的“对”（把冲突元素按规则安排到合理位置），后者让散列表具有了可扩充性，可以动态调整（不用担心填满了怎么办）。

双散列

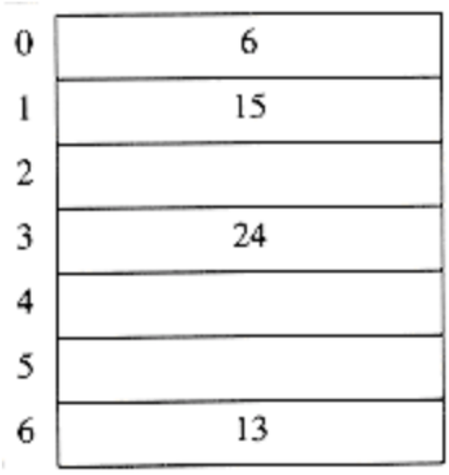
我们来考察最后一个冲突解决方法，双散列（double hashing）。常用的方法是让F（i）= i * $hash_2$(x)，这意思是用第二个散列函数算出x的散列值，然后在距离hash2(x)，2$hash_2$(x)的地方探测。$hash_2$(x)作为关键，必须要合理选取，否则会引起灾难性的后果——各种撞车。这个策略暂时不做过多分析了。

再散列

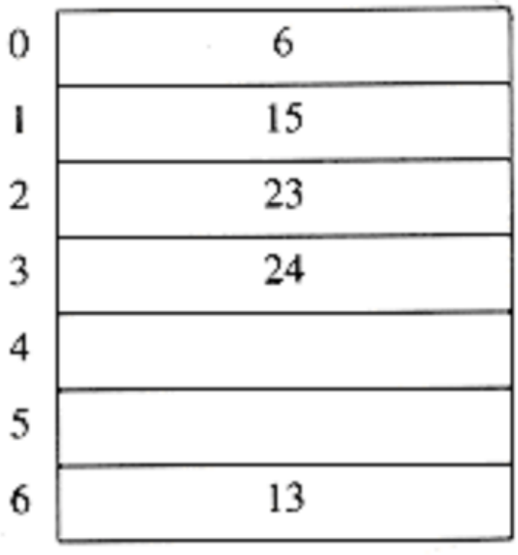
之前说过，对于使用平方探测法的闭散列里，如果元素填的太满的话后续插入将耗费过长的时间，甚至可能Insert失败，因为这里面会有太多的移动和插入混合操作。怎么办呢？一种解决方法是建立另外一个大约两倍大的表，再用一个新的散列函数，扫描整个原始表然后按照新的映射插入到新的表里。

再散列的目的是为了后续的插入方便。

比如我们把{6, 15, 23, 24,6}插入到Size=7的闭散列里，Hash（x）= x % 7，用线性探测的方法解决冲突，会得到这样一个结果；



现在还剩23，把这个插入之后，整个表里就填满了70%以上：

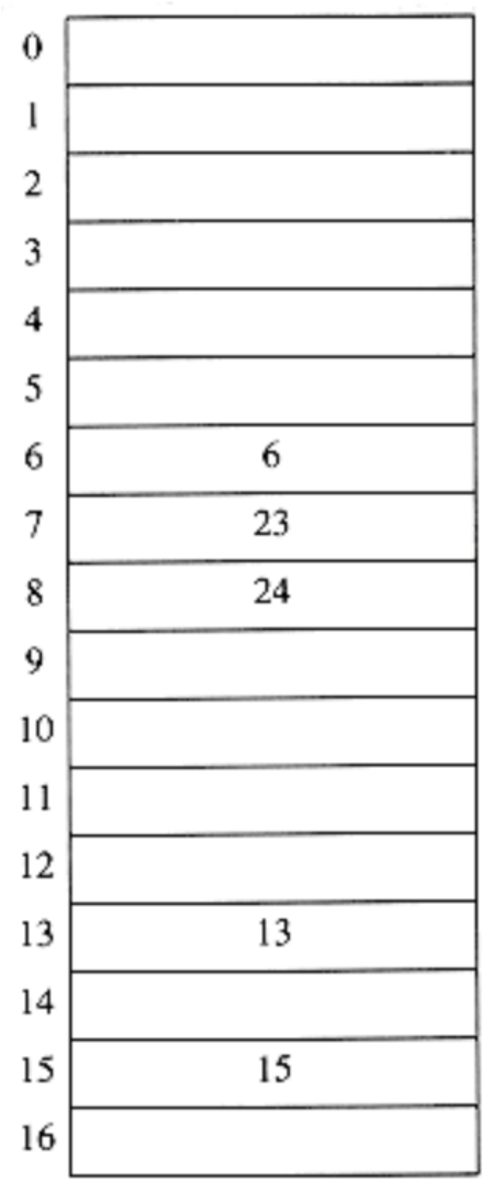


于是我们要建立一个新的表，newSize=17，这是离原规模2倍大小的最近素数。新的散列函数是Hash(x) = x % 17。扫描原来的表，把所有元素插入到新的表里，得到这个：

随笔档案

阅读排行榜

评论排行榜



这一顿操作就是再散列。可以看出这会付出很昂贵的代价：运行时间O（N），不过庆幸的是实际情况里并不会经常需要我们再散列，都是等快填满了才做一次，所以还没那么差。得说明一下，这种技术是对程序员友好而对用户不友好的。因为如果我们把这种结构应用于某个程序，那并不会有什么显著的效果，另一方面，如果再散列作为交互系统的一部分运行，可能使用户感到系统变慢。所以到底用不用还是要权衡一番的，运行速度不敏感的场景就可以用，方便自己，因为这个技术把程序员从对表规模的担心中解放出来了。

具体实现可以用平方探测以很多种方式实现

1. 只要表有一半满了就做
2. 只有当插入失败时才做（这种比较极端）
3. 途中策略：当表到达某个装填因子时再做。

由于随着装填因子的增加，表的性能会有所下降，所以第三个方法或许是最好的。再散列把程序员从对表规模的担心中解放出来了，这一点的重要之处在于在复杂程序中散列表不可能一开始就做得很大，然后高枕无忧。因为我们也不知道多大才够用，所以能使她动态调整这个特性就很有必要了。实现的时候也比较简单

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 { Postion 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

```
HashTable Rehash(HashTable H) {
    int i,OldSize;
    Cell *OldCells;

    OldCells=H->TheCells;
    OldSize=H->TableSize;

    //新建一个原规模*2的表
    H=Init(OldSize<<1);

    //扫描原表，重新插入到新表里
    for (i=0; i<OldSize; i++) {
        if (OldCells[i].Info==Legitimate) {
            Insert(OldCells[i].value, H);
        }
    }

    free(OldCells);
    return H;
}
```

散列篇的开头就说了，这不是一种单纯的技术，而是一种思想。所以我们不必机械地理解她，可以把这种思想灵活地用在其他结构中，比如在队列变满的时候，可以声明一个双倍大小的数组，然后拷贝过来，释放原来的队列。这就有点像向量的规模调整了，联系的普遍性再一次得到印证。

散列篇到这里就要结束了，在收尾之际我们不妨做一个总结，回眸下这一路沿途的风景。

散列表可以用O（1）的平均时间完成insert和Find，在使用散列的时候要尤其注意装填因子的问题，因为他是保证时间上确界的关键。对于分离链接法，尽量让λ接近1。对于开放定址法来说，不到万不得已就别让λ太大，尽量保持λ<=0.5。如果用线性探测，性能会随着λ趋向于1而急剧下降。再散列运算可以通过表的伸缩来完成，这样就会保持λ处于合理范围，而且优点还在于，如果当下空间紧缺的话，这么做是很棒的策略。

比较一下二叉查找树和散列，二叉查找树也可以实现Insert和Find，效率会比散列低一些，O（logN）。虽说这方面慢了一点，但是二叉树能支持更多的操作，比如可以FindMin和FindMax，这个散列就做不到了。还有，二叉查找树可以迅速找到在一定范围内的所有元素，散列也做不到，而且O（logN ）也不会比O（1）慢太多，因为查找树不需要做乘除法，就弥补了一些速度缺陷，综上看来她们也算是各有千秋。

说完了平均时间，再说说最坏情况散列的最坏情况一般是实现的缺憾，而二叉树的最坏情况呢，是输入序列有序的时候，那这个时候根据BST规则，二叉树会退化成一条单链，升序的输入会导致一捺的情形，降序输入会形成一撇。这要是再增删查改付出的可就是O（N）了。平衡查找树的实现相对复杂一些，所以如果不需要有序的信息以及对输入是否排序有要求的话，就该选择散列这种结构。

5. Re:母函数应用

连续顶

--lcl1997

散列还有着丰富的应用，这里举四个例子：第一个，编译器使用散列表跟踪源代中声明的变量，这种数据结构叫做符号表。散列表示这种问题的理想应用，因为只有Insert和Find操作。而且标识符一般都很短，所以根据这个短字符串能迅速算出哈希值。第二个，在图论的应用，对于节点有实际名字而不是数字的图论问题都可以用散列表来做。比如某个顶点叫计算机，那么某个特定的超算中心对应的计算机列表里有ibm1,ibm2,ibm3这样的。如果用查找树来做这个事，那效率就很滑稽了2333 第三种用途是在为游戏编制的程序里。程序搜索游戏不同的行（row，不是同行那个行）时，根据实时位置计算出一个散列值，然后跟踪这些值来确定位置。如果同样的位置再出现，那么程序会用简单的移动变换来避免重复计算，因为重复计算的代价都很大。游戏程序的这种叫做变换表。

第四个用途就是在线拼写检验程序，比如word里面的拼写检测，把整个词典预先散列，然后检测每个单词拼写对不对，这只花费O(1)时间。散列表很适合这项工作，因为以字典序排列单词并不重要，我们不关心它的顺序，就避免了散列的缺陷。

总而言之，扬长避短地选用不同结构处理工作才是我们学习数据结构的第一要义。

标签: 散列, 数据结构

仪式黑刃
关注 - 2
粉丝 - 28
+加关注

« 上一篇： 开放定址法——平方探测(Quadratic Probing)
» 下一篇： 云心出岫——Splay Tree

posted @ 2018-08-08 16:05 仪式黑刃 阅读(2630) 评论(0) 编辑 收藏 举报

[Ctrl+Enter快捷键提交]