**[跳跃表（数据结构）](http://www.cnblogs.com/zhuangli/archive/2008/08/25/1275665.html)**

二叉树是我们都非常熟悉的一种数据结构。它支持包括查找、插入、删除等一系列的操作。但它有一个致命的弱点，就是当数据的随机性不够时，会导致其树型结构的不平衡，从而直接影响到算法的效率。

跳跃表（Skip List）是1987年才诞生的一种崭新的数据结构，它在进行查找、插入、删除等操作时的期望时间复杂度均为O(logn),有着近乎替代平衡树的本领。而且最重要的一点，就是它的编程复杂度较同类的AVL树，红黑树等要低得多，这使得其无论是在理解还是在推广性上，都有着十分明显的优势。

跳跃表由多条链构成（S0，S1，S2 ……，Sh），且满足如下三个条件：

（1）           每条链必须包含两个特殊元素：+∞ 和 -∞

（2）           S0包含所有的元素，并且所有链中的元素按照升序排列。

（3）           每条链中的元素集合必须包含于序数较小的链的元素集合，即：

【基本操作】

在对跳跃表有一个初步的认识以后，我们来看一下基于它的几个最基本的操作。

一、    查找

目的：在跳跃表中查找一个元素x

    在跳跃表中查找一个元素x，按照如下几个步骤进行：

i)                 从最上层的链（Sh）的开头开始

ii)              假设当前位置为p，它向右指向的节点为q（p与q不一定相邻），且q的值为y。将y与x作比较

(1) x=y     输出***查询成功***及相关信息

(2) x>y     从p向右移动到q的位置

(3) x<y     从p向下移动一格

iii)    如果当前位置在最底层的链中（S0），且还要往下移动的话，则输出***查询失败***

二、插入

    目的：向跳跃表中插入一个元素x

    首先明确，向跳跃表中插入一个元素，相当于在表中插入一列从S0中某一位置出发向上的连续一段元素。有两个参数需要确定，即插入列的位置以及它的“高度”。

    关于插入的位置，我们先利用跳跃表的查找功能，找到比x小的最大的数y。根据跳跃表中所有链均是递增序列的原则，x必然就插在y的后面。

    而插入列的“高度”较前者来说显得更加重要，也更加难以确定。由于它的不确定性，使得不同的决策可能会导致截然不同的算法效率。为了使插入数据之后，保持该数据结构进行各种操作均为O(logn)复杂度的性质，我们引入**随机化算法**（Randomized Algorithms）。

我们定义一个**随机决策模块**，它的大致内容如下：

·产生一个0到1的随机数r                   r ← random()

·如果r小于一个常数p，则执行方案A，       if  r<p then do A

 否则，执行方案B                                   else do B

初始时列高为1。插入元素时，不停地执行随机决策模块。如果要求执行的是A操作，则将列的高度加1，并且继续反复执行随机决策模块。直到第i次，模块要求执行的是B操作，我们结束决策，并向跳跃表中插入一个高度为i的列。

**性质1：    根据上述决策方法，该列的高度大于等于k的概率为pk-1。**

此处有一个地方需要注意，如果得到的i比当前跳跃表的高度h还要大的话，则需要增加新的链，使得跳跃表仍满足先前所提到的条件。

我们来看一个例子：

假设当前我们要插入元素“40”，且在执行了随机决策模块后得到高度为4

·步骤一：找到表中比40小的最大的数，确定插入位置

·步骤二：插入高度为4的列，并维护跳跃表的结构

 三、删除

    目的：从跳跃表中删除一个元素x

    删除操作分为以下三个步骤：

（1）           在跳跃表中查找到这个元素的位置，如果未找到，则退出     **\***

（2）           将该元素所在整列从表中删除                              **\***

（3）           将多余的“空链”删除                                    **\***

所谓“记忆化”查找，就是在前一次查找的基础上进行进一步的查找。它可以利用前一次查找所得到的信息，取其中可以被当前查找所利用的部分。利用“记忆化”查找可以将一次查找的复杂度变为O(logk)，**其中k为此次与前一次两个被查找元素在跳跃表中位置的距离。**

下面来看一下记忆化搜索的具体实现方法：

假设上一次操作我们查询的元素为i，此次操作我们欲查询的元素为j。我们用一个update数组来记录在查找i时，指针在每一层所“跳”到的最右边的位置。如图4.1中橘黄色的元素。（蓝色为路径上的其它元素）

    在插入元素j时，分为两种情况：

（1）       i<=j

从S0层开始向上遍历update数组中的元素，直到找到某个元素，它向右指向的元素大于等于j，并于此处开始新一轮对j的查找（与一般的查找过程相同）

（2）       i>j

从S0层开始向上遍历update数组中的元素，直到找到某个元素小于等于j，并于此处开始新一轮对j的查找（与一般的查找过程相同）

    图4.2十分详细地说明了在查找了i=37之后，继续查找j=**15**或**53**时的两种不同情况。

【复杂度分析】

一个数据结构的好坏大部分取决于它自身的空间复杂度以及基于它一系列操作的时间复杂度。跳跃表之所以被誉为几乎能够代替平衡树，其复杂度方面自然不会落后。我们来看一下跳跃表的相关复杂度：

空间复杂度： O(n)           （期望）

跳跃表高度： O(logn)        （期望）

相关操作的时间复杂度：

    查找： O(logn)        （期望）

    插入：  O(logn)        （期望）

    删除： O(logn)        （期望）

之所以在每一项后面都加一个“期望”，是因为跳跃表的复杂度分析是基于概率论的。有可能会产生最坏情况，不过这种概率极其微小。

下面我们来一项一项分析。

一、 空间复杂度分析 O(n)

假设一共有n个元素。根据**性质1**，每个元素插入到第i层（Si）的概率为pi-1 ，则在第i层插入的期望元素个数为npi-1，跳跃表的元素期望个数为 ，当p取小于0.5的数时，次数总和小于2n。

所以总的空间复杂度为O(n)

二、跳跃表高度分析 O(logn)

根据**性质1**，每个元素插入到第i层（Si）的概率为pi ，则在第i层插入的期望元素个数为npi-1。

考虑一个特殊的层：第1+ 层。

层的元素期望个数为  = 1/n2，当n取较大数时，这个式子的值接近0，故跳跃表的高度为O(logn)级别的。

三、查找的时间复杂度分析 O(logn)

我们采用逆向分析的方法。假设我们现在在目标节点，想要走到跳跃表最左上方的开始节点。这条路径的长度，即可理解为查找的时间复杂度。

设当前在第i层第j列那个节点上。

i)                     如果第j列恰好只有i层（对应插入这个元素时第i次调用随机化模块时所产生的B决策，概率为1-p），则当前这个位置必然是从左方的某个节点向右跳过来的。

ii)                  如果第j列的层数大于i（对应插入这个元素时第i次调用随机化模块时所产生的A决策，概率为p），则当前这个位置必然是从上方跳下来的。（不可能从左方来，否则在以前就已经跳到当前节点上方的节点了，不会跳到当前节点左方的节点）

设C(k)为向上跳k层的期望步数（包括横向跳跃）

有：

C(0) = 0

C(k) = (1-p)(1+向左跳跃之后的步数)+p(1+向上跳跃之后的步数)

     = (1-p)(1+C(k)) + p(1+C(k-1))

C(k) = 1/p + C(k-1)

C(k) = k/p

    而跳跃表的高度又是logn级别的，故查找的复杂度也为logn级别。

对于记忆化查找（Search with fingers）技术我们可以采用类似的方法分析，很容易得出它的复杂度是O(logk)的（其中k为此次与前一次两个被查找元素在跳跃表中位置的距离）。

四、插入与删除的时间复杂度分析 O(logn)

    插入和删除都由查找和更新两部分构成。查找的时间复杂度为O(logn)，更新部分的复杂度又与跳跃表的高度成正比，即也为O(logn)。

    所以，插入和删除操作的时间复杂度都为O(logn)

五、实际测试效果

（1）不同的p对算法复杂度的影响

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| P | 平均操作时间 | 平均列高 | 总结点数 | 每次查找跳跃次数  （平均值） | 每次插入跳跃次数  （平均值） | 每次删除跳跃次数  （平均值） |
| 2/3 | 0.0024690 ms | 3.004 | 91233 | 39.878 | 41.604 | 41.566 |
| 1/2 | 0.0020180 ms | 1.995 | 60683 | 27.807 | 29.947 | 29.072 |
| 1/e | 0.0019870 ms | 1.584 | 47570 | 27.332 | 28.238 | 28.452 |
| 1/4 | 0.0021720 ms | 1.330 | 40478 | 28.726 | 29.472 | 29.664 |
| 1/8 | 0.0026880 ms | 1.144 | 34420 | 35.147 | 35.821 | 36.007 |

表1   进行106次随机操作后的统计结果

    从表1中可见，当p取1/2和1/e的时候，时间效率比较高[（为什么？）](http://www.cnblogs.com/zhuangli/archive/2008/08/25/1275665.html" \l "p)。而如果在实际应用中空间要求很严格的话，那就可以考虑取稍小一些的p，如1/4。

（2）运用“记忆化”查找 (Search with fingers) 的效果分析

    所谓“记忆化”查找，就是在前一次查找的基础上进行进一步的查找。它可以利用前一次查找所得到的信息，取其中可以被当前查找所利用的部分。利用“记忆化”查找可以将一次查找的复杂度变为O(logk)，其中k为此次与前一次两个被查找元素在跳跃表中位置的距离。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| P | 数据类型 | 平均操作时间（不运用记忆化查找） | 平均操作时间**（运用记忆化查找）** | 平均每次查找跳跃次数（不运用记忆化查找） | 平均每次查找跳跃次数**（运用记忆化查找）** |
| 0.5 | 随机（相邻被查找元素键值差的绝对值**较大**） | 0.0020150 ms | 0.0020790 ms | 23.262 | 26.509 |
| 0.5 | 前后具备相关性（相邻被查找元素键值差的绝对值**较小**） | 0.0008440 ms | **0.0006880 ms** | 26.157 | **4.932** |

表1   进行106次相关操作后的统计结果

从表2中可见，当数据相邻被查找元素键值差绝对值较小的时候，我们运用“记忆化”查找的优势是很明显的，不过当数据随机化程度比较高的时候，“记忆化”查找不但不能提高效率，反而会因为跳跃次数过多而成为算法的瓶颈。

合理地利用此项优化，可以在特定的情况下将算法效率提升一个层次。

【跳跃表的应用】

高效率的相关操作和较低的编程复杂度使得跳跃表在实际应用中的范围十分广泛。尤其在那些编程时间特别紧张的情况下，高性价比的跳跃表很可能会成为你的得力助手。

能运用到跳跃表的地方很多，与其去翻陈年老题，不如来个趁热打铁，拿NOI2004第一试的第一题——郁闷的出纳员(Cashier)来“小试牛刀”吧。

**例题一：NOI2004 Day1 郁闷的出纳员(Cashier)**

[[点击查看附录中的原题]](http://www.cnblogs.com/zhuangli/archive/2008/08/25/1275665.html#Cashier)

    这道题解法的多样性给了我们一次对比的机会。用不同的算法和数据结构，在效率上会有怎样的差异呢？

    首先定义几个变量

    R – 工资的范围

    N – 员工总数

我们来看一下每一种适用的算法和数据结构的简要描述和理论复杂度：

（1）       线段树

简要描述：以工资为关键字构造线段树，并完成相关操作。

I命令时间复杂度：O(logR)

A命令时间复杂度：O(1)

S命令时间复杂度：O(logR)

F命令时间复杂度：O(logR)

（2）       伸展树(Splay tree)

简要描述：以工资为关键字构造伸展树，并通过“旋转”完成相关操作。

I命令时间复杂度：O(logN)

A命令时间复杂度：O(1)

S命令时间复杂度：O(logN)

F命令时间复杂度：O(logN)

（3）       跳跃表(Skip List)

简要描述：运用跳跃表数据结构完成相关操作。

I命令时间复杂度：O(logN)

A命令时间复杂度：O(1)

S命令时间复杂度：O(logN)

F命令时间复杂度：O(logN)

    实际效果评测： （单位：秒）

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Test1 | Test2 | Test3 | Test4 | Test5 | Test6 | Test7 | Test8 | Test9 | Test10 |
| 线段树 | 0.000 | 0.000 | 0.000 | 0.031 | 0.062 | 0.094 | 0.109 | 0.203 | 0.265 | 0.250 |
| 伸展树 | 0.000 | 0.000 | 0.016 | 0.062 | 0.047 | 0.125 | 0.141 | 0.360 | 0.453 | 0.422 |
| 跳跃表 | 0.000 | 0.000 | 0.000 | 0.047 | 0.062 | 0.109 | 0.156 | 0.368 | 0.438 | 0.375 |

    从结果来看，线段树这种经典的数据结构似乎占据着很大的优势。可有一点万万不能忽略，那就是线段树是基于键值构造的，它受到键值范围的约束。在本题中R的范围只有105级别，这在内存较宽裕的情况下还是可以接受的。但是如果问题要求的键值范围较大，或者根本就不是整数时，线段树可就很难适应了。这时候我们就不得不考虑伸展树、跳跃表这类基于元素构造的数据结构。而从实际测试结果看，跳跃表的效率并不比伸展树差。加上编程复杂度上的优势，跳跃表尽显出其简单高效的特点。