Skip list的实现原理

最近看了一种数据结构叫做skipList，redis和levelDB都是用了它。Skip List是在有序链表的基础上进行了扩展，解决了有序链表结构查找特定值困难的问题，查找特定值的时间复杂度为O(logn)，他是一种可以代替平衡树的数据结构。

    下面是skipList的一个介绍，转载来的，源地址：<http://kenby.iteye.com/blog/1187303>，为防止源地址丢失，故拷贝一份放在这里，望作者原谅。

**———————————————转载开始—————————————————**

**为什么选择跳表**

目前经常使用的平衡数据结构有：B树，红黑树，AVL树，Splay Tree, Treep等。

想象一下，给你一张草稿纸，一只笔，一个编辑器，你能立即实现一颗红黑树，或者AVL树

出来吗？ 很难吧，这需要时间，要考虑很多细节，要参考一堆算法与数据结构之类的树，

还要参考网上的代码，相当麻烦。

用跳表吧，跳表是一种随机化的数据结构，目前开源软件 Redis 和 LevelDB 都有用到它，

它的效率和红黑树以及 AVL 树不相上下，但跳表的原理相当简单，只要你能熟练操作链表，

就能轻松实现一个 SkipList。

**有序表的搜索**

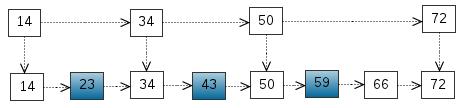
考虑一个有序表：

http://dl.iteye.com/upload/attachment/565643/d5d03b36-abff-34ea-9c40-a1fbfb709a81.jpg

从该有序表中搜索元素 < 23, 43, 59 > ，需要比较的次数分别为 < 2, 4, 6 >，总共比较的次数

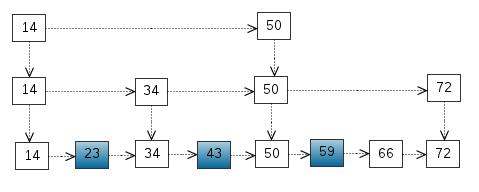
为 2 + 4 + 6 = 12 次。有没有优化的算法吗?  链表是有序的，但不能使用二分查找。类似二叉

搜索树，我们把一些节点提取出来，作为索引。得到如下结构：



 这里我们把 < 14, 34, 50, 72 > 提取出来作为一级索引，这样搜索的时候就可以减少比较次数了。

 我们还可以再从一级索引提取一些元素出来，作为二级索引，变成如下结构：

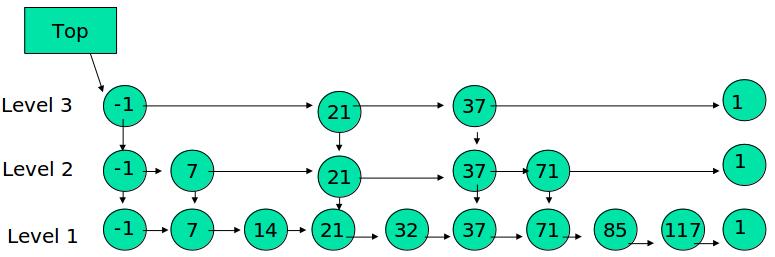


     这里元素不多，体现不出优势，如果元素足够多，这种索引结构就能体现出优势来了。

**跳表**

下面的结构是就是跳表：

 其中 -1 表示 INT\_MIN， 链表的最小值，1 表示 INT\_MAX，链表的最大值。



跳表具有如下性质：

(1) 由很多层结构组成

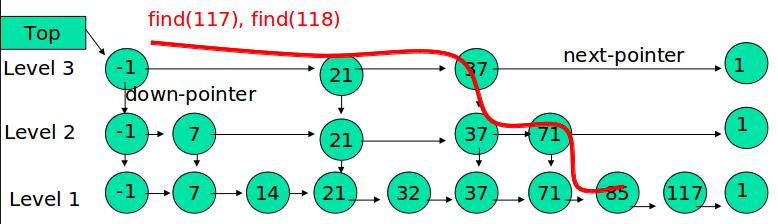
(2) 每一层都是一个有序的链表

(3) 最底层(Level 1)的链表包含所有元素

(4) 如果一个元素出现在 Level i 的链表中，则它在 Level i 之下的链表也都会出现。

(5) 每个节点包含两个指针，一个指向同一链表中的下一个元素，一个指向下面一层的元素。

**跳表的搜索**



例子：查找元素 117

(1) 比较 21， 比 21 大，往后面找

(2) 比较 37,   比 37大，比链表最大值小，从 37 的下面一层开始找

(3) 比较 71,  比 71 大，比链表最大值小，从 71 的下面一层开始找

(4) 比较 85， 比 85 大，从后面找

(5) 比较 117， 等于 117， 找到了节点。

具体的搜索算法如下：

**C代码   [收藏代码](http://kenby.iteye.com/blog/1187303)**

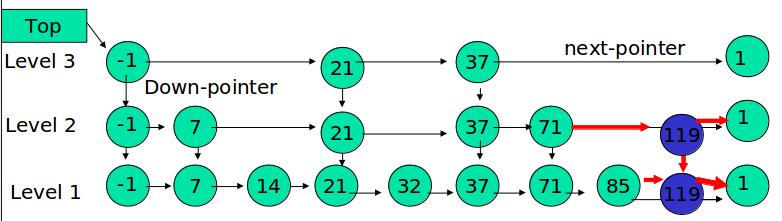
1. /\* 如果存在 x, 返回 x 所在的节点，
2. \* 否则返回 x 的后继节点 \*/
3. find(x)
4. {
5. p = top;
6. **while** (1) {
7. **while** (p->next->key < x)
8. p = p->next;
9. **if** (p->down == NULL)
10. **return** p->next;
11. p = p->down;
12. }
13. }

**跳表的插入**

先确定该元素要占据的层数 K（采用丢硬币的方式，这完全是随机的）

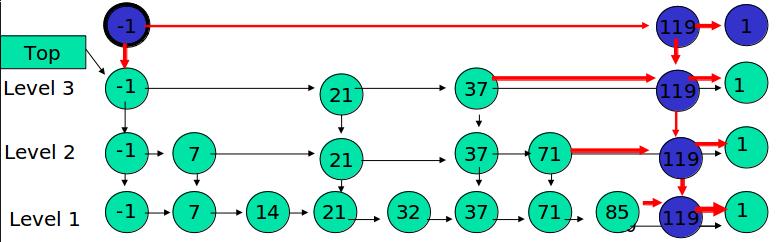
然后在 Level 1 ... Level K 各个层的链表都插入元素。

例子：插入 119， K = 2



如果 K 大于链表的层数，则要添加新的层。

例子：插入 119， K = 4



**丢硬币决定 K**

插入元素的时候，元素所占有的层数完全是随机的，通过一下随机算法产生：

**C代码   [收藏代码](http://kenby.iteye.com/blog/1187303)**

1. **int** random\_level()
2. {
3. K = 1;
5. **while** (random(0,1))
6. K++;
8. **return** K;
9. }

相当与做一次丢硬币的实验，如果遇到正面，继续丢，遇到反面，则停止，

用实验中丢硬币的次数 K 作为元素占有的层数。显然随机变量 K 满足参数为 p = 1/2 的几何分布，

K 的期望值 E[K] = 1/p = 2. 就是说，各个元素的层数，期望值是 2 层。

**跳表的高度。**

n 个元素的跳表，每个元素插入的时候都要做一次实验，用来决定元素占据的层数 K，

跳表的高度等于这 n 次实验中产生的最大 K，待续。。。

**跳表的空间复杂度分析**

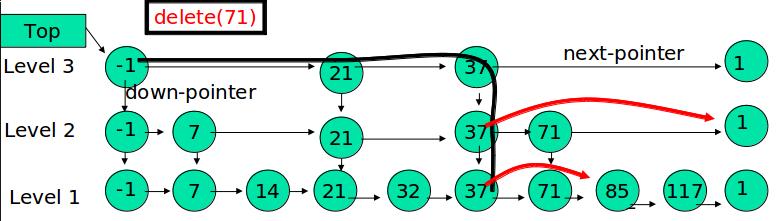
根据上面的分析，每个元素的期望高度为 2， 一个大小为 n 的跳表，其节点数目的

期望值是 2n。

**跳表的删除**

在各个层中找到包含 x 的节点，使用标准的 delete from list 方法删除该节点。

例子：删除 71



**————————————————————转载结束—————————————————————**

下面是完整的skipSet的实现，有详细的注释。

**Java代码  [收藏代码](javascript:void())**

1. /\*\*
2. \*  跳表节点数据存储结构
3. \*/
4. **class** SkipNode<E **extends** Comparable<? **super** E>> {
5. **public** **final** E value; //节点存储的数据
6. **public** **final** SkipNode<E>[] forward; //节点的指针数组
8. /\*\*
9. \* 根据节点的层级构造一个节点
10. \* @param level 节点层级
11. \* @param value 节点存储值
12. \*/
13. @SuppressWarnings("unchecked")
14. **public** SkipNode(**int** level, E value) {
15. forward = **new** SkipNode[level + 1];//level层的元素后面带着level+1的指针数组
16. **this**.value = value;
17. }
19. }
21. **public** **class** SkipSet<E **extends** Comparable<? **super** E>> {
23. /\*\*
24. \* 概率因子，实验证明p=1/e比p=0.5要好，e是个神奇的数字！
25. \*/
26. //  public static final double P = 0.5;
27. **public** **static** **final** **double** P = 1/Math.E;
28. /\*\*
29. \*  最大层级
30. \*/
31. **public** **static** **final** **int** MAX\_LEVEL = 6;
33. /\*\*
34. \* 开始节点，不存值，贯穿所有层
35. \*/
36. **public** **final** SkipNode<E> header = **new** SkipNode<E>(MAX\_LEVEL, **null**);
37. /\*\*
38. \* 当前跳表的最高层级
39. \*/
40. **public** **int** level = 0;
42. /\*\*
43. \* 插入一个元素
44. \* @param value 待插入值
45. \*/
46. @SuppressWarnings("unchecked")
47. **public** **void** insert(E value) {
48. SkipNode<E> x = header;
49. SkipNode<E>[] update = **new** SkipNode[MAX\_LEVEL + 1];
50. //查找元素的位置，这里其实做了一次contain操作，注释见contain
51. **for** (**int** i = level; i >= 0; i--) {
52. **while** (x.forward[i] != **null**
53. && x.forward[i].value.compareTo(value) < 0) {
54. x = x.forward[i];
55. }
56. //update[i]是比value小的数里面最大的，是value的前置节点
57. update[i] = x;
58. }
59. x = x.forward[0];
61. //此处不允许插入相同元素，为一个set
62. **if** (x == **null** || !x.value.equals(value)) {//跳表中不包含所要插的元素
63. //随机产生插入的层级
64. **int** lvl = randomLevel();
65. //产生的随机层级比当前跳表的最高层级大，需要添加相应的层级，并更新最高层级
66. **if** (lvl > level) {
67. **for** (**int** i = level + 1; i <= lvl; i++) {
68. update[i] = header;
69. }
70. level = lvl;
71. }
73. //生成新节点
74. x = **new** SkipNode<E>(lvl, value);
75. //调整节点的指针，和指向它的指针
76. **for** (**int** i = 0; i <= lvl; i++) {
77. x.forward[i] = update[i].forward[i];
78. update[i].forward[i] = x;
79. }
81. }
82. }
83. /\*\*
84. \* 删除一个元素
85. \* @param value 待删除值
86. \*/
87. @SuppressWarnings("unchecked")
88. **public** **void** delete(E value) {
89. SkipNode<E> x = header;
90. SkipNode<E>[] update = **new** SkipNode[MAX\_LEVEL + 1];
91. //查找元素的位置，这里其实做了一次contain操作，注释见contain
92. **for** (**int** i = level; i >= 0; i--) {
93. **while** (x.forward[i] != **null**
94. && x.forward[i].value.compareTo(value) < 0) {
95. x = x.forward[i];
96. }
97. update[i] = x;
98. }
99. x = x.forward[0];
100. //删除元素，调整指针
101. **if** (x.value.equals(value)) {
102. **for** (**int** i = 0; i <= level; i++) {
103. **if** (update[i].forward[i] != x)
104. **break**;
105. update[i].forward[i] = x.forward[i];
106. }
107. //如果元素为本层最后一个元素，则删除同时降低当前层级
108. **while** (level > 0 && header.forward[level] == **null**) {
109. level--;
110. }
112. }
113. }
114. /\*\*
115. \* 查找是否包含此元素
116. \* @param searchValue 带查找值
117. \* @return true：包含；false:不包含
118. \*/
119. **public** **boolean** contains(E searchValue) {
120. SkipNode<E> x = header;
121. //从开始节点的最高层级开始查找
122. **for** (**int** i = level; i >= 0; i--) {
123. //当到达本层级的NULL节点或者遇到比查找值大的节点时，转到下一层级查找
124. **while** (x.forward[i] != **null**
125. && x.forward[i].value.compareTo(searchValue) < 0) {
126. x = x.forward[i];
127. }
128. }
129. x = x.forward[0];
130. //此时x有三种可能，1.x=null,2.x.value=searchValue,3.x.value>searchValue
131. **return** x != **null** && x.value.equals(searchValue);
132. }
133. /\*\*
134. \* 这里是跳表的精髓所在，通过随机概率来判断节点的层级
135. \* @return 节点的层级
136. \*/
137. **public** **static** **int** randomLevel() {
138. **int** lvl = (**int**) (Math.log(1. - Math.random()) / Math.log(1. - P));
139. **return** Math.min(lvl, MAX\_LEVEL);
140. }
142. /\*\*
143. \* 输出跳表的所有元素
144. \* 遍历最底层的元素即可
145. \*/
146. **public** String toString() {
147. StringBuilder sb = **new** StringBuilder();
148. sb.append("{");
149. SkipNode<E> x = header.forward[0];
150. **while** (x != **null**) {
151. sb.append(x.value);
152. x = x.forward[0];
153. **if** (x != **null**)
154. sb.append(",");
155. }
156. sb.append("}");
157. **return** sb.toString();
158. }
159. }