



yinlili2010的专栏

目录视图 摘要视图 RSS 订阅

个人资料



Diehard_Yin



访问： 58285次
积分： 1117
等级： BLOG > 4
排名： 千里之外

原创： 48篇
转载： 30篇
译文： 1篇
评论： 13条

文章搜索

文章分类

- 算法 (14)
- C++实现 (2)
- 命令 (1)
- 数据挖掘 (15)
- 留学申请 (1)
- 数据结构 (7)
- 机器学习 (17)
- machine learning tools (2)
- tech problems in daily life (2)
- C++ language (6)
- C++工程 (9)
- 图论 (2)
- 软件行业积累 (0)
- 股票 (1)
- Tools (3)

文章存档

- 2015年11月 (1)
- 2015年10月 (2)
- 2015年09月 (3)

【活动】Python创意编程活动开始啦！！ CSDN日报20170428 ——《你的开发为何如此低效？》 深入浅出，带你学习Unity

跳跃表以及C++实现

标签： c++ 跳跃表 skip list

2014-09-24 00:32 1190人阅读 评论(0) 收藏 举报

分类： 数据结构 (6)

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录(?) [+]

首先为了方便，我大概在博友林子的博客基础上进行编辑，今天我将跳跃表实现了一下，算法导论公开课的那位年轻教授说他花了半个小时写好半个小时调试好，我的时间估计是他的4倍吧，只有结构实现看了这篇博客的插入代码，了解了只需要在n空间大小构建跳跃表的结构。对于自己几乎独立的实现了这个算法，虽然简单，但是还是很有成就感的。需要自己注意的是在独立实现代码的时候不仅仅只是需要简单的了解大概的算法，而是

- 1) 确定需要使用的**数据结构**，如今天的代码中的SKNode, SkipList.
- 2)如何**初始化这个数据结构**，这样才能得到求解问题循环结束条件。（初始化条件很重要）
- 3) 先自己按照算法流程，插入、查找、删除一个一个写好伪代码，这样才不至于在写代码的时候没有关注点，思维混乱。可以先从最简单也是最核心的查找算法写起。
- 4) 最后是调试，碰到不对的步骤，可以添加打印，我现在对于数据结构还有太底层的调试时看不出来太多信息的，只能一步一步的试探。这也是一个软肋，以后有空一定要再学学编程范式这种比较底层的东西，还有内存管理也很重要。

参考的博文地址如：

http://blog.csdn.net/u013011841/article/details/39158585

跳跃链表简介

二叉树是一种常见的数据结构。它支持包括查找、插入、删除等一系列操作。但它有一个致命的弱点，就是当数据的随机性不够时，会导致其树形结构的不平衡，从而直接影响算法的效率。

跳跃链表 (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。

跳跃链表的最大优势在于无论是查找、插入和删除都是O(logn)，不过由于跳跃链表的操作是基于概率形成的，那么它操作复杂度大于O(logn)的概率为，可以看出当n越大的时候失败的概率越小。

另外跳跃链表的实现也十分简单，在平衡树中是最易实现的一种结构。例如像复杂的红黑树，你很难在不依靠工具书的帮助下实现该算法，但是跳跃链表不一样，你可以很容易在半个小时内就完成其实现。

跳跃链表的空间复杂度的期望为O(n)，链表的层数期望为O(logn)。

如何改进普通的链表？

我们先看看一个普通的链表

2015年08月 (4)

2015年07月 (1)

展开

阅读排行

机器学习7-SVM

(6246)

分交界定法 branch-and-

(2211)

Octave安装指导

(1961)

逻辑回归--Octave实现

(1347)

线性回归--Octave实现

(1283)

数据挖掘总介与PageRank

(1275)

聚类算法（一）层次聚类

(1273)

神经网络-Octave实现

(1213)

跳跃表以及C++实现

(1190)

机器学习的相关书籍推荐

(1165)

评论排行

线性回归--Octave实现

(4)

逻辑回归--Octave实现

(3)

聚类算法（一）层次聚类

(2)

非参数学习算法之局部加

(1)

深度学习参考资料

(1)

相似项发现（一）

(1)

Eclipse C++ and Xcode

(1)

层次聚类（二）

(1)

C++模板

(0)

Special cases in C++ pr

(0)

推荐文章

* CSDN日报20170429 ——《程序修行从“拔刀术”到“万剑诀”》

* 抓取网易云音乐歌曲热门评论生成词云

* Android NDK开发之从环境搭建到Demo级十步流

* 个人的中小型项目前端架构浅谈

* 基于卷积神经网络(CNN)的中文垃圾邮件检测

* 四无年轻人如何逆袭

最新评论

非参数学习算法之局部加权回归 peizhen7095: underfitting吧

聚类算法（一）层次聚类 cust_gj: 这程序 runtime error 啊

深度学习参考资料 longbye0: 博主，把原来的转载地址放出来吧

相似项发现（一） JLOGAN: 请问前辈对于餐馆，房地产数据之类的一般用什么方法匹配比较高效呢

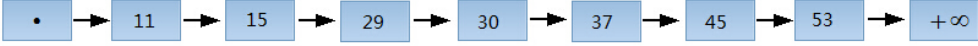
聚类算法（一）层次聚类 大号小白兔: 感谢楼主分享

层次聚类（二） 大号小白兔: 感谢楼主分享

Eclipse C++ and Xcode for Mac geekczt: 您好，您的邮件多少，我希望能和您沟通，交流，您做的这方面我也在做。我的邮箱 geekczt@163.c...

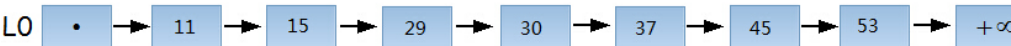
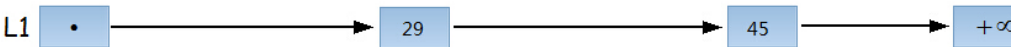
逻辑回归--Octave实现 hscspring: hx=sigmoid(X*theta);J=-1/m*sum((1-y).^1...

Link List



可以看出查询这个链表O(n),插入和删除也是O(n).因此链表这种结构虽然节省空间，但是效率不高，那有没有什么办法可以改进呢？

我们可以增加一条链表做为快速通道。这样我们使用均匀分布，从图中可以看出L1层充当L0层的快速通道，底层的结点每隔固定的几个结点出现在上面一层。



我们这里主要以查找操作来介绍，因为插入和删除操作主要的复杂度也是取决于查找，那么两条链表查找的最好的时间复杂度是多少呢？

一次查找操作首先要先在上层遍历<=|L1|次操作,然后在下层遍历<=(L0/L1)次操作,至多要经历

$$|L_1| + \frac{|L_0|}{|L_1|}$$
$$= |L_1| + \frac{n}{|L_1|}$$

次操作，其中|L1|为L1的长度,n为L0的长度.

那么最好的时间复杂度,也就怎么设置间隔距离才能使查找次数最少有

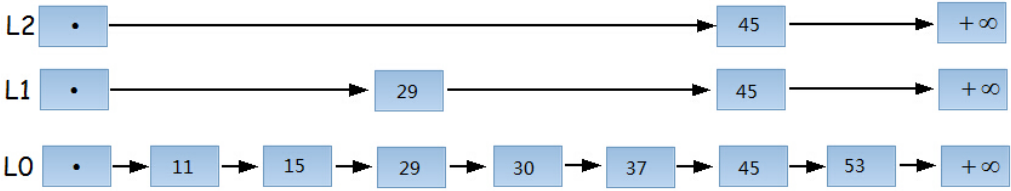
$$\min mize \quad |L_1| + \frac{n}{|L_1|}$$

我们对|L1|的长度求得

$$|L_1| = \sqrt{n}$$

把上式代入函数,查找次数最小也就是 $2\sqrt{n}$.这意味着下层每隔 \sqrt{n} 个结点在上层就有一个结点作为快速跑道。

那么三条链表呢



同理那么我们让L2/L1=L1/L0,然后同样列出方程，求导可得L2= $\sqrt[3]{n}$ ，查找次数为 $3\sqrt[3]{n}$

第k条链条.....查找次数为 $k\sqrt[k]{n}$

我们这里取k=logn，代入的查找次数为2logn.

到此为主，我们应该知道了，期望上最好的层数是logn层,而且上下层结点数比为2，这样查找次数常数最小,复杂度保持在O(logn)。

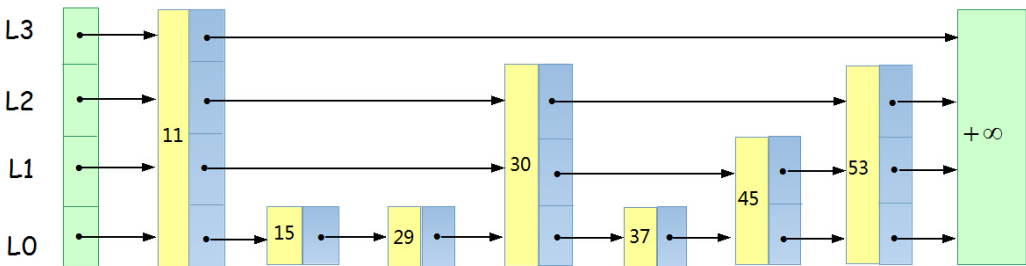
跳跃链表的结构

跳跃表由多条链构成（L0，L1，L2，Lh），且满足如下三个条件：

- 每条链必须包含两个特殊元素：+∞ 和 -∞(可以需要，可以不需要，我的实现是采用了-∞作为header之后的节点，即第一个节点，+∞作为最后一个节点)
- L0包含所有的元素，并且所有链中的元素按照升序排列。
- 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。

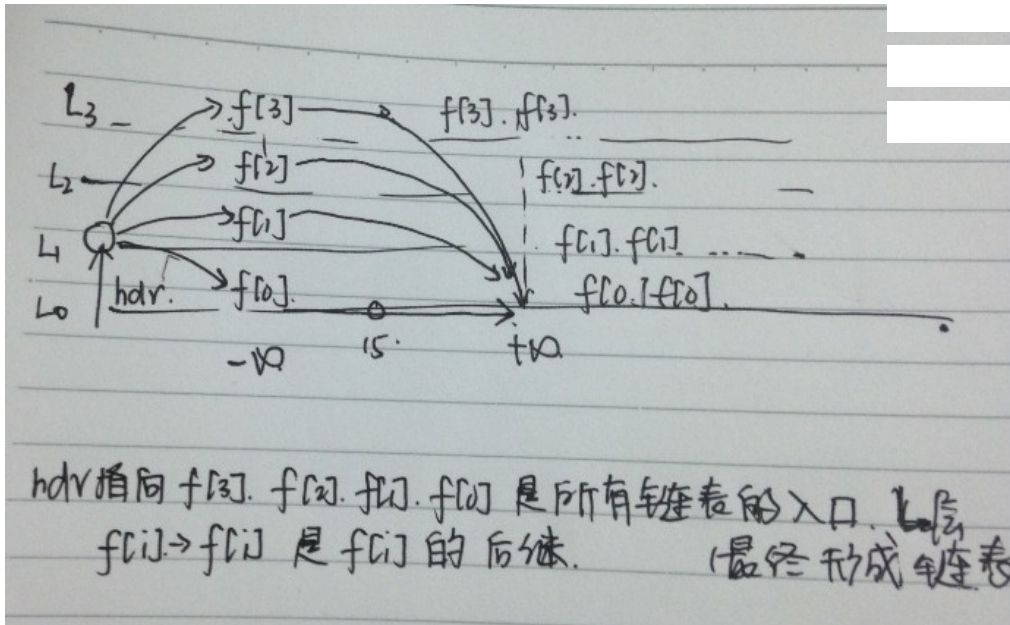
关闭

逻辑回归--Octave实现
Diehard_Yin: @hscspring:我没有测过 感觉
lambda/m*theta(2:size(X,2));应该是...
逻辑回归--Octave实现
hscspring: grad=1/m*X*(hx-y);temp=theta;temp(1)=1/m*sum(hy...



结点结构源代码

我觉得如果不考虑空间效率是可以将数据在每层上面分开存储，但是也可以办到在一个n的空间存储，跳跃表就是在这些数据之间建立links，使用links替代树形结构，如Btreap树堆，R-B Trees, AVL trees。现在我给的是基于一个n空间存储。



```
[cpp] view plain copy print ?
01. class SKNode
02. {
03. public:
04.     int key;
05.     SKNode* forward[MAXLEVEL];
06.
07.     SKNode()
08.     {
09.         key=0;
10.         for(int i =0;i<MAXLEVEL;i++)
11.         {
12.             forward[i]= NULL;
13.         }
14.     }
15.     SKNode& operator=(const SKNode* & node)
16.     {
17.         key=node->key;
18.         for(int i=0;i<MAXLEVEL;i++)
19.         {
20.             forward[i] = node->forward[i];
21.         }
22.         return *this;
23.     }
24. };
25. //skip list, it has a header, this header have maxlevel pointers
26. class SkipList
27. {
28. public:
29.     SKNode *hdr;           /* list Header */
30.     int listLevel;          /* current level of list */
31.     int insert(int key);
32.     SKNode* search(int key);
33.     int deleteNode(int key);
34.     void printList();
```

关闭

```

35.     SkipList()
36.     {
37.         hdr = new SKNode;
38.         listLevel = 0;
39.         hdr->key = -INT_MAX;
40.         SKNode* end = new SKNode;
41.         SKNode* first = new SKNode;
42.         first->key = -INT_MAX;
43.         end->key = INT_MAX;
44.         for(int i = 0; i < MAXLEVEL; i++)
45.         {
46.             hdr->forward[i] = first;
47.             hdr->forward[i]->forward[i] = end;
48.         }
49.         printList();
50.     }
51.     ~SkipList()
52.     {
53.         delete hdr;
54.     }
55. };

```

MAXlevel可以是 $\log(n)/\log(2)$

跳跃链表查找操作

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：

1、p = hdr; 从最上层的链 (Lh) 的开头开始,进入到各层

2、for i(listLevel-1, 0)，从最上层的链 (Lh) 的开头开始，如L3到L0

3、假设当前位置为p，p初始值为hdr，p在i层的下一个节点为q = p->forward[i]，它向右指向的节点为q (p与q不一定相邻)。将x和q.key做比较

(1) if $x > q.key$ 在i层继续向右走，

then p = p->forward[i].

(2) else $x \leq q.key$,

then i--，即将当前指针直接下移到下一层。

for循环结束，i变为了0，即在最后一层，p = p->forward[0]

4、判断p，

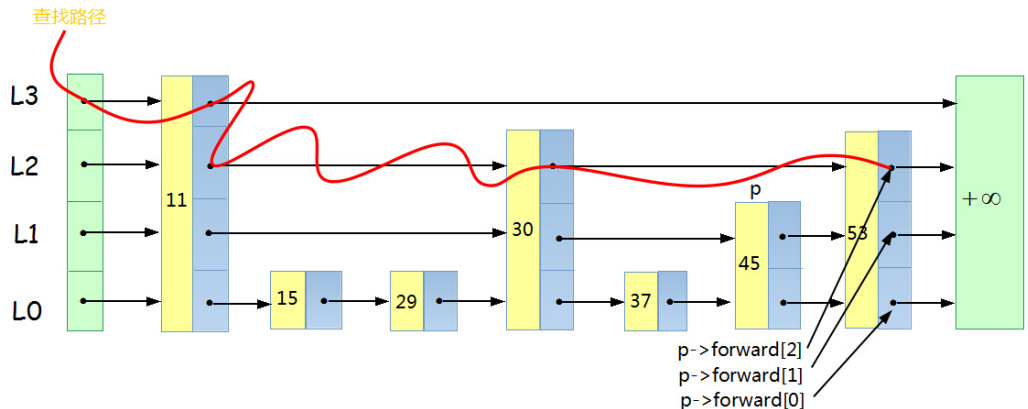
(1) if p!= NULL && p->key = x

return p

(2) return NULL

如我们查找29，先从入口到3, $29 > 11$ ，p = 11，判断知道往下走L2， $29 < 30$ ，继续往下L1， $29 < 30$ ，L1， $29 > 15$ ，前进， $29 \leq 29$ ，结束，p在15的位置，所以最后指针调整到p = p->forward[0]，判断p值。

(今天因为我从listLevel开始，浪费了好多调试时间。。。)



下面是C++实现代码：

```

[cpp] view plain copy print ?
01. SKNode* SkipList::search(int key)
02. {
03.     SKNode* current = new SKNode;
04.     current = hdr;
05.     int i = listLevel-1;
06.     for(; i >= 0; i--)

```

关闭

```

07.     {
08.         while(current->forward[i]->key != INT_MAX && key>current->forward[i]->key)//key大于下一个数据的值。转到本层下一个元素
09.         {
10.             current = current->forward[i];
11.         }
12.         //否则i--, 转到下一层
13.     }
14.     current = current->forward[0];
15.     if(current!= NULL && current->key == key)
16.     {
17.         cout<<"find"<<key<<endl;
18.         return current;
19.     }
20.     return NULL;
21.
22. }

```

跳跃链表插入操作

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链。插入时，**两个参数需要确定，即插入列的位置以及它的“高度”**。

1) 关于在L0层的插入的位置，我们先利用跳跃表的查找功能， $x \leq q.key$ ，所以x的位置一定是在x之后，q之前。同样可以推论在L1,L2,L listLevel层的位置是在循环中生成他们这层最后的位置，就是在search的while之后记录一个这个位置为S[i]。最后需要在所有的S[i]之后重连数据之间的链接。（但是需要注意的是为了，如果我们不想加入重复数据，需要判断p->forward[0]的值，如果相等，就是找到了，不需要再插入。当然如果我们不介意重复数据，也可以不加这个判断。）

2) 需要插入的高度，决定在L1-listLevel的哪些位置加。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 $O(\log n)$ 复杂度的性质，我们引入随机化算法（Randomized Algorithms）。

伪代码如下：

Skip List Insertion(x)

p = hdr;

newlevel = getlevel();

s[listLevel] = hdr(注意需要全部初始化为hdr，为了newlevel增长了，但是增长的层次s[i]却没有数据，没有初始化，应该从头结点开始)

for i(listLevel-1, 0)

while(q!=+∞ && x>key)

then p = p->forward[i].

//else x<=q.key,

//then i--

s[i]=p; (s[i]为i层探索的最后一个节点，最后需要在这之后插入x)

last = p->forward[0] //判断是否相等，。。。。

//插入数据，重连链表

if(newlevel>level) level = newlevel

for i(newlevel-1 - 0)

node->forward[i]= s[i]->forward[i]

s[i]->forward[i] = node

我定义一个随机决策模块，它的大致内容如下，但是这个代码不能保证完全随机，其实每次的运行结果都是一样的：

```

[cpp] view plain copy print ?
01. int getInsertLevel()
02. {
03.     int upcount = 0;
04.     for(int i=0;i<MAXLEVEL;i++)
05.     {

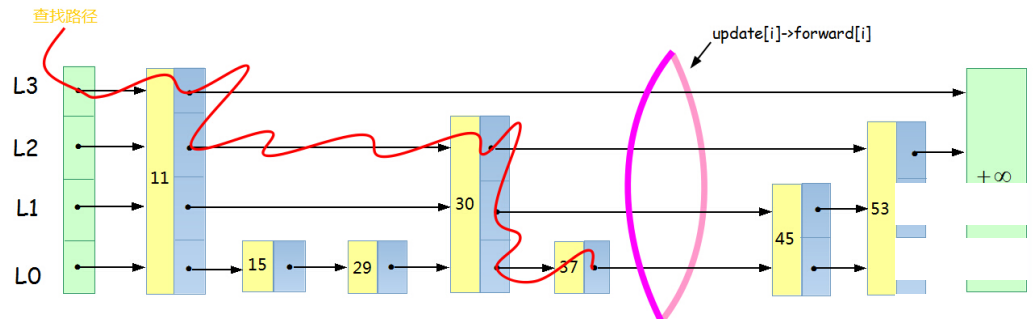
```

关闭

```

06.         int num = rand()%10;
07.         if(num<5)
08.         {
09.             upcount++;
10.         }
11.     }
12.     return upcount;
13. }

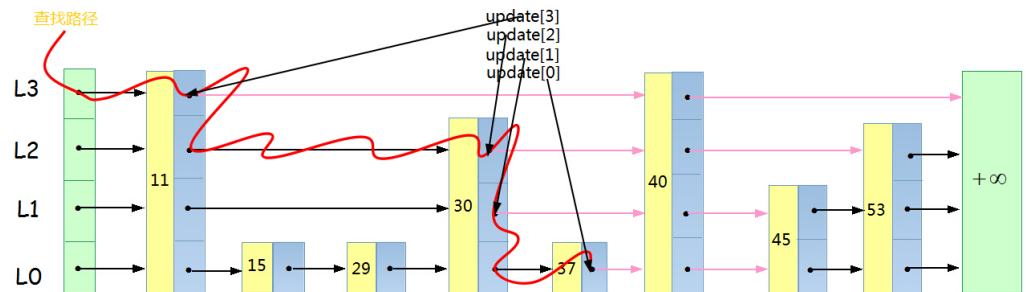
```



如插入43，查找路径如下。

43的下一个数接到的40下一个数45。

40的下一个数接到43



紫色的箭头表示更新过的指针

[cpp] view plain copy print ?

```

01. int SkipList::insert(int key)
02. {
03.     int level = getInsertLevel();
04.     SKNode* node = new SKNode;
05.     node->key=key;
06.
07.     SKNode *s[MAXLEVEL];
08.     SKNode* current = new SKNode;
09.     SKNode* last = new SKNode;
10.     for(int i =0;i<MAXLEVEL;i++)
11.     {
12.         s[i]=hdr->forward[i];//initiation
13.     }
14.     current = last = hdr;
15.     cout<<"hdr"<<hdr->key<<endl;
16.     int i = listLevel-1;
17.     for(;i>=0;i--)
18.     {
19.         while(current->forward[i]->key != INT_MAX && key>current->forward[i]->key)//key大于下一
            个数据的值。转到本层下一个元素
20.         {
21.             current = current->forward[i];
22.         }
23.         s[i] = current;//保存每一层位置上的最后指针的前驱
24.     }
25.     last=current->forward[0];
26.     if(last != NULL && last->key == key)
27.     {
28.         cout<<"inset key:"<<key<<"already existed"<<endl;
29.         return 0;
30.     }
31.     if(level>listLevel)//更新层数
32.     {
33.         listLevel = level;
34.     }
35. }

```

关闭


```

36.         for(int k = 0; k < listLevel; k++)
37.         {
38.             node->forward[k] = s[k]->forward[k];
39.             s[k]->forward[k] = node;
40.
41.         }
42.         if(level > listLevel)
43.         {
44.             listLevel = level;
45.         }
46.         return 1;
47.
48.     }

```

跳跃链表的删除

目的：从跳跃表中删除一个元素x

删除链表和插入几乎一模一样的，只是在最后重接链表不同：

在跳跃表中查找到这个元素的位置，如果未找到，则退出

否则将该元素所在整列从表中删除

将多余的“空链”删除

Skip List Deletion(x)

```
p = hdr;
```

```
//newlevel = getlevel();
```

s[listLevel] = hdr(注意需要全部初始化为hdr，为了newlevel增长了，但是增长的层次s[i]却没有数据，没有初始化，应该从头结点开始)

```
fori(listLevel-1, 0)
```

```
while(q!=+∞ && x>key)
```

```
then p = p->forward[i].
```

```
//else x<=q.key,
```

```
//then i--
```

s[i]=p; (s[i]为i层探索的最后一个节点，最后需要在这之后插入x)

```
last = p->forward[0] //判断是否相等，。。。。。
```

```
if(last->key != x)
```

```
return
```

//删除数据，重连链表

```
for i(listLevel-1 - 0)
```

```
s[i]->forward[i]=s[i]->forward[i]->forward[i];
```

这段代码如下：

```

[cpp] view plain copy print ?
01. intSkipList::deleteNode(int key)
02. {
03.     SKNode *s[MAXLEVEL];
04.     SKNode* current = new SKNode;
05.     SKNode* last = new SKNode;
06.     for(int i = 0; i < MAXLEVEL; i++)

```

关闭

```

07. {
08. s[i]=hdr->forward[i];//initiation
09. }
10. current = last = hdr;
11. for(inti = listLevel-1;i>=0;i--)
12. {
13. while(current->forward[i]->key != INT_MAX && key>current->forward[i]->key)//key大于下一个数据的
    值。转到本层下一个元素
14. {
15. current = current->forward[i];
16. }
17. s[i] = current;//保存每一层位置上的最后指针的前驱
18. }
19. last=current->forward[0];
20. if(last->key != key)
21. {
22. cout<<"delete key:"<<key<<"does not existed"<<endl;
23. }
24. return 0;
25. }
26. for(inti = 0; i<listLevel;i++)
27. {
28. s[i]->forward[i]=s[i]->forward[i]->forward[i];
29. }
30. return 1;
31. }

```

整个C++程序如下

```

[cpp] view plain copy print ?
01. #include <iostream>
02. #include <vector>
03. using namespace std;
04.
05. #define MAXLEVEL 4 //最多2 power n=16个数
06. /*skip list node,they are keys and pointers*/
07. classSKNode
08. {
09. public:
10. int key;
11. SKNode* forward[MAXLEVEL];
12.
13. SKNode()
14. {
15. key=0;
16. for(inti =0;i<MAXLEVEL;i++)
17. {
18. forward[i]= NULL;
19. }
20. }
21. SKNode& operator=(constSKNode* & node)
22. {
23. key=node->key;
24. for(inti=0;i<MAXLEVEL;i++)
25. {
26. forward[i] = node->forward[i];
27. }
28. return *this;
29. }
30. };
31. //skip list, it has a header, this header have maxlevel pointers
32. classSkipList
33. {
34. public:
35. SKNode *hdr; /* list Header */
36. intlistLevel; /* current level of list */
37. int insert(int key);
38. SKNode* search(int key);
39. intdeleteNode(int key);
40. voidprintList();
41. SkipList()
42. {
43. hdr = new SKNode;
44. listLevel = 0;
45. hdr->key = -INT_MAX;
46. SKNode* end = new SKNode;
47. SKNode* first = new SKNode;

```

关闭


```

48. first->key=-INT_MAX;
49. end->key=INT_MAX;
50. for(int i = 0; i < MAXLEVEL; i++)
51. {
52.     hdr->forward[i] = first;
53.     hdr->forward[i]->forward[i] = end;
54. }
55. printList();
56. }
57. ~Skiplist()
58. {
59.     delete hdr;
60. }
61. };
62. int getInsertLevel()
63. {
64.     int upcount = 0;
65.     for(int i = 0; i < MAXLEVEL; i++)
66.     {
67.         int num = rand() % 10;
68.         if(num < 5)
69.         {
70.             upcount++;
71.         }
72.     }
73.     return upcount;
74. }
75. SKNode* Skiplist::search(int key)
76. {
77.     SKNode* current = new SKNode;
78.     current = hdr;
79.     int i = listLevel - 1;
80.     for(; i >= 0; i--)
81.     {
82.         while(current->forward[i]->key != INT_MAX && key > current->forward[i]->key) //key大于下一个数据的
            //值。转到本层下一个元素
83.         {
84.             current = current->forward[i];
85.         }
86.         //否则i--, 转到下一层
87.     }
88.     current = current->forward[0];
89.     if(current != NULL && current->key == key)
90.     {
91.         cout << "find" << key << endl;
92.         return current;
93.     }
94.     return NULL;
95. }
96. }
97.
98. int Skiplist::insert(int key)
99. {
100.     int level = getInsertLevel();
101.     SKNode* node = new SKNode;
102.     node->key = key;
103.
104.     SKNode *s[MAXLEVEL];
105.     SKNode* current = new SKNode;
106.     SKNode* last = new SKNode;
107.     for(int i = 0; i < MAXLEVEL; i++)
108.     {
109.         s[i] = hdr->forward[i]; //initiation
110.     }
111.     current = last = hdr;
112.     cout << "hdr" << hdr->key << endl;
113.     int i = listLevel - 1;
114.     for(; i >= 0; i--)
115.     {
116.         while(current->forward[i]->key != INT_MAX && key > current->forward[i]->key) //key大于下一个数据的
            //值。转到本层下一个元素
117.         {
118.             current = current->forward[i];
119.         }
120.         s[i] = current; //保存每一层位置上的最后指针的前驱
121.     }
122.     last = current->forward[0];
123.     if(last != NULL && last->key == key)
124.     {

```

关闭

```

125. cout<<"inset key:"<<key<<"already existed"<<endl;
126. return 0;
127. }
128. if(level>listLevel)//更新层数
129. {
130. listLevel = level;
131. }
132.
133. for(int k = 0; k <listLevel;k++)
134. {
135. node->forward[k]=s[k]->forward[k];
136. s[k]->forward[k]=node;
137.
138. }
139. if(level>listLevel)
140. {
141. listLevel = level;
142. }
143. return 1;
144.
145. }
146. intSkipList::deleteNode(int key)
147. {
148. SKNode *s[MAXLEVEL];
149. SKNode* current = new SKNode;
150. SKNode* last = new SKNode;
151. for(inti =0;i<MAXLEVEL;i++)
152. {
153. s[i]=hdr->forward[i];//initiation
154. }
155. current = last = hdr;
156. for(inti = listLevel-1;i>=0;i--)
157. {
158. while(current->forward[i]->key != INT_MAX && key>current->forward[i]->key)//key大于下一个数据的
    值。转到本层下一个元素
159. {
160. current = current->forward[i];
161. }
162. s[i] = current;//保存每一层位置上的最后指针的前驱
163. }
164. last=current->forward[0];
165. if(last->key != key)
166. {
167. cout<<"delete key:"<<key<<"does not existed"<<endl;
168.
169. return 0;
170. }
171. for(inti = 0; i<listLevel;i++)
172. {
173. s[i]->forward[i]=s[i]->forward[i]->forward[i];
174. }
175. return 1;
176. }
177. voidSkipList::printList()
178. {
179. SKNode* current = hdr;
180. for(inti = listLevel -1;i>=0;i--)
181. {
182. current = hdr->forward[i];
183. cout<<"level "<<i<<"....."<<endl;
184. while(current->forward[i] != NULL)//key大于下一个数据的值。转到本层下一个元素
185. {
186. cout<<" "<<current->key;
187. current = current->forward[i];
188. }
189. cout<<" "<<current->key<<endl;
190. }
191. }
192.
193. int main()
194. {
195. SkipListsk;
196. constint n = 7;
197. intnum[n]={30,15,45,37,11,53,17};
198. cout<<"test insert....."<<endl;
199. for(inti = 0;i<n;i++)
200. {
201. sk.insert(num[i]);
202. }

```

```

203.     sk.printList();
204.     cout<<"test search....."<<endl;
205.     sk.search(17);
206.     cout<<"test delete....."<<endl;
207.     sk.deleteNode(30);
208.     sk.printList();
209.     system("pause");
210.     return 0;
211. }

```

跳跃链表的搜索时间复杂度为 $O(\log n)$

定理： n 个元素的跳跃链表的每一次搜索的时间复杂度有很高的概率为 $O(\log n)$.

高概率：事件 E 以很高的概率发生意味着对于 $a \geq 1$, 存在一个合适的常数使得事件 E 发生的概率 $\Pr\{E\} > 1 - \frac{1}{n^a}$.

其中 a 是任意选择的一个数, 不同的 a 影响搜索时间复杂度的常数, 即 $a \cdot O(\log n)$, 这个在后面介绍.

我们要证跳跃链表的时间复杂度, 不能只是证明一次搜索的复杂度为, 是要证明全部的搜索都是 $O(\log n)$. 因为这是基于概率的算法, 如果光一次有效率并没有多大作用.

我们定义时间 E_i 为某一次搜索失败的概率, 那么假设 k 次搜索, 我们先假定失败的概率为 $O(1/n^a)$, 其中至少有一次失败的概率为

$$\begin{aligned}
 & \Pr\{E_1 \cup E_2 \cup \dots \cup E_k\} \\
 & \leq \Pr\{E_1\} + \Pr\{E_2\} + \dots + \Pr\{E_k\} \\
 & = k \times \frac{1}{n^a} \\
 & \underline{\underline{\text{令 } k = n^c \frac{1}{n^{\alpha-c}}}}}
 \end{aligned}$$

可以估算出 k 次有一次失败的概率为 $1/n^{(a-c)}$, 那么我们只要让 $a > c+1$ 或者 a 取无穷大, 就可以证明每一次搜索都具有高概率成功。

跳跃链表的层数有高概率为 $O(\log n)$

类似上面的方法, 对于 n 个元素, 如果有一个层数超过 $O(\log n)$ 就算失败. 那么对于某一个元素超过 $c \log n$ 层, 即失败的概率为 $\Pr\{E\}$. 那么对于一次搜索失败的概率为

$$\begin{aligned}
 & n \times \Pr\{\text{某一结点抬升的层数} \geq c \cdot \log_2 n\} \\
 & \leq n \cdot \left(\frac{1}{2}\right)^{c \cdot \log_2 n} \\
 & = n \cdot \left(\frac{1}{2^{\log_2 n^c}}\right) \\
 & = \frac{n}{n^c} \\
 & = \frac{1}{n^{c-1}}
 \end{aligned}$$

令 $a=c-1$, 则只要 $a \geq 1$ 时, 就有高概率的可能使得层数为 $O(\log n)$

关闭

跳跃链表单次查找复杂度大于 $O(\log n)$ 的概率

每完成一次查找, 都肯定要从最顶层移动到最下面一层, 这每改变一次层数是由概率选择时候的 p 处于扔硬币中的正面决定的. 既然上面知道层数高概率为 $c \log n$ 层, 那么扔正面的次数为 $c \log n - 1$ 次.

我们假设扔了 $c \log n$ 个正面，超过 $10 \cdot c \log n$ 次是反面，则有

$$\begin{aligned} & \Pr\{\text{扔了 } c \log n \text{ 次正面, 超过 } 10c \log n \text{ 次反面}\} \\ & \leq \binom{10c \log n}{c \log n} \left(\frac{1}{2}\right)^{9c \log n}, \text{ 因为 } \left(\frac{y}{x}\right)^x \leq \left(e \frac{y}{x}\right)^x \\ & \leq e^{\left(\frac{10c \log n}{c \log n}\right)^{c \log n} \left(\frac{1}{2}\right)^{9 \log n}} \\ & = \frac{(10e)^{c \log n}}{2^{9c \log n}} \\ & = \frac{2^{\log(10e)c \log n}}{2^{9c \log n}} \\ & = 2^{[\log(10e)-9]c \log n} \\ & = \frac{1}{2^{[9-\log(10e)]c \log n}} \\ & \text{令 } \alpha = [9 - \log(10e)]c \quad \frac{1}{n^\alpha} \end{aligned}$$

因为 $9 - \log(10e)$ 中的9是线性增长，要远远大于 $\log(10e)$ 中的对数增长，因此超过 $10c \log n$ 的概率随着10的增长变得越来越小。所以全部的操作都在 $10 \log n$ 以内，我们使用 k 替代10作为常数，即查找次数为 $k \log n$ ，为 $O(\log n)$ 。

顶

1

踩

0

上一篇

聚类算法总结

下一篇

留学申请Google搜索技巧

我的同类文章

数据结构（6）			
• 二叉搜索树--进阶篇之红黑树	2014-09-25	阅读 533	• 二叉搜索树--进阶篇之平衡... 2014-09-21 阅读 540
• 二叉搜索树--基础篇	2014-09-20	阅读 504	• 哈希表初篇 2014-09-11 阅读 498
• 堆排序及C++实现	2014-09-05	阅读 446	• 实现循环单链表 2014-09-02 阅读 429

参考知识库



.NET 知识库
3791 关注 | 833 收录



算法与数据结构知识库
15893 关注 | 2320 收录

猜你在找

- 数据结构与算法在实战项目中的应用
- 转跳跃表-原理及Java实现
- 数据结构和算法
- Redis基本数据结构跳跃表实现
- 数据结构基础系列(1): 数据结构和算法
- 跳跃表-原理及Java实现
- 《C语言/C++学习指南》加密解密篇（安全相关算法）
- Skip List跳跃表原理详解与实现

数据结构基础系列(7): 图

跳跃表实现文件C语言


查看评论

暂无评论

发表评论

用户名: u010442388

评论内容:



提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

- 全部主题
- Hadoop
- AWS
- 移动游戏
- Java
- Android
- iOS
- Swift
- 智能硬件
- Docker
- OpenStack
- VPN
- Spark
- ERP
- IE10
- Eclipse
- CRM
- JavaScript
- 数据库
- Ubuntu
- NFC
- WAP
- jQuery
- BI
- HTML5
- Spring
- Apache
- .NET
- API
- HTML
- SDK
- IIS
- Fedora
- XML
- LBS
- Unity
- Splashtop
- UML
- components
- Windows Mobile
- Rails
- QEMU
- KDE
- Cassandra
- CloudStack
- FTC
- coremail
- OPhone
- CouchBase
- 云计算
- iOS6
- Rackspace
- Web App
- SpringSide
- Maemo
- Compuware
- 大数据
- aptch
- Perl
- Tornado
- Ruby
- Hibernate
- ThinkPHP
- HBase
- Pure
- Solr
- Angular
- Cloud Foundry
- Redis
- Scala
- Django
- Bootstrap

公司简介 | 招贤纳士 | 广告服务 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈

网站客服 杂志客服 微博客服 webmaster@csdn.net 400-600-2320 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 |

江苏乐知网络技术有限公司

京 ICP 证 09002463 号 | Copyright © 1999-2017, CSDN.NET, All Rights Reserved

